

Azure Functions documentation

Azure Functions is a cloud service available on-demand that provides all the continually updated infrastructure and resources needed to run your applications. You focus on the code that matters most to you, in the most productive language for you, and Functions handles the rest. Functions provides serverless compute for Azure. You can use Functions to build web APIs, respond to database changes, process IoT streams, manage message queues, and more.

About Azure Functions

OVERVIEW

[Introduction to Azure Functions](#)

CONCEPT

[Azure Functions hosting options](#)

[Choose the right integration and automation services in Azure](#)

[What are Durable Functions?](#)

Create your first function

GET STARTED

[Getting started with Azure Functions](#)

QUICKSTART

[C#](#)

[Java](#)

[JavaScript](#)

[PowerShell](#)

[Python](#)

[Rust/Go](#)

[TypeScript](#)

Develop functions

CONCEPT

[Azure Functions developer guide](#)

[Supported languages in Azure Functions](#)

[Azure Functions triggers and bindings concepts](#)

[Code and test Azure Functions locally](#)

SAMPLE

[Azure Functions samples](#)

[Azure Serverless Community Library ↗](#)

[Azure Resource Manager templates ↗](#)

TUTORIAL

[Functions with Logic Apps](#)

[Develop Python functions with VS Code](#)

[Create serverless APIs using Visual Studio](#)

Azure Functions overview

Article • 05/24/2023

Azure Functions is a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs. Instead of worrying about deploying and maintaining servers, the cloud infrastructure provides all the up-to-date resources needed to keep your applications running.

You focus on the code that matters most to you, in the most productive language for you, and Azure Functions handles the rest.

For the best experience with the Functions documentation, choose your preferred development language from the list of native Functions languages at the top of the article.

Scenarios

Functions provides a comprehensive set of event-driven [triggers and bindings](#) that connect your functions to other services without having to write extra code.

The following are a common, *but by no means exhaustive*, set of integrated scenarios that feature Functions.

If you want to...	then...
Process file uploads	Run code when a file is uploaded or changed in blob storage.
Process data in real time	Capture and transform data from event and IoT source streams on the way to storage.
Infer on data models	Pull text from a queue and present it to various AI services for analysis and classification.
Run scheduled task	Execute data clean-up code on pre-defined timed intervals.
Build a scalable web API	Implement a set of REST endpoints for your web applications using HTTP triggers.
Build a serverless workflow	Create an event-driven workflow from a series of functions using Durable Functions.
Respond to database changes	Run custom logic when a document is created or updated in Azure Cosmos DB.

If you want to...	then...
Create reliable message systems	Process message queues using Queue Storage, Service Bus, or Event Hubs.

These scenarios allow you to build event-driven systems using modern architectural patterns. For more information, see [Azure Functions Scenarios](#).

Development lifecycle

With Functions, you write your function code in your preferred language using your favorite development tools and then deploy your code to the Azure cloud. Functions provides native support for developing in [C#, Java, JavaScript, PowerShell, Python](#), plus the ability to use [more languages](#), such as Rust and Go.

Functions integrates directly with Visual Studio, Visual Studio Code, Maven, and other popular development tools to enable seamless debugging and [deployments](#).

Functions also integrates with Azure Monitor and Azure Application Insights to provide comprehensive runtime telemetry and analysis of your [functions in the cloud](#).

Hosting options

Functions provides a variety [hosting options](#) for your business needs and application workload. [Event-driven scaling hosting options](#) range from fully serverless, where you only pay for execution time (Consumption plan), to always warm instances kept ready for fastest response times (Premium plan).

When you have excess App Service hosting resources, you can host your functions in an existing App Service plan. This kind of Dedicated hosting plan is also a good choice when you need predictable scaling behaviors and costs from your functions.

If you want complete control over your functions runtime environment and dependencies, you can even deploy your functions in containers that you can fully customize. Your custom containers can be hosted by Functions, deployed as part of a microservices architecture in Azure Container Apps, or even self-hosted in Kubernetes.

Next Steps

[Azure Functions Scenarios](#)

Get started through lessons, samples, and interactive tutorials

Azure Functions scenarios

Article • 09/26/2024

We often build systems to react to a series of critical events. Whether you're building a web API, responding to database changes, processing event streams or messages, Azure Functions can be used to implement them.

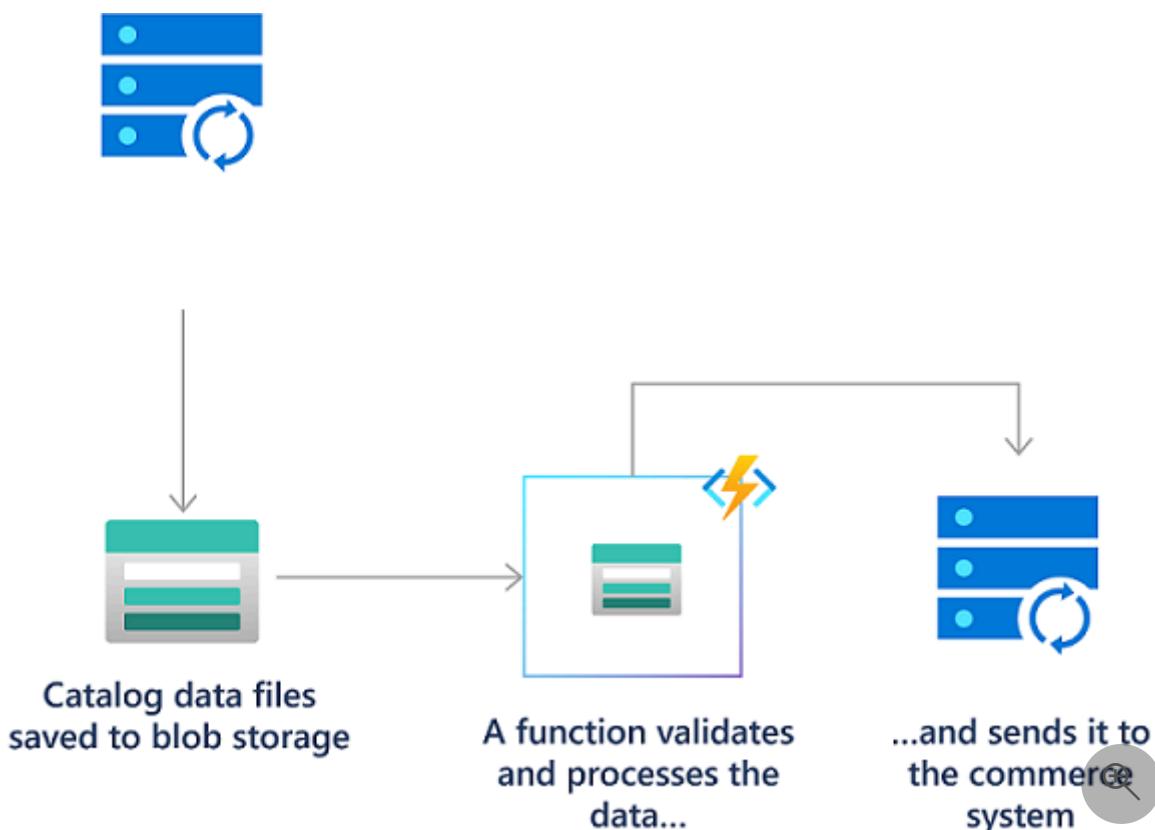
In many cases, a function [integrates with an array of cloud services](#) to provide feature-rich implementations. The following are a common (but by no means exhaustive) set of scenarios for Azure Functions.

Select your development language at the top of the article.

Process file uploads

There are several ways to use functions to process files into or out of a blob storage container. To learn more about options for triggering on a blob container, see [Working with blobs](#) in the best practices documentation.

For example, in a retail solution, a partner system can submit product catalog information as files into blob storage. You can use a blob triggered function to validate, transform, and process the files into the main system as they're uploaded.



The following tutorials use a Blob trigger (Event Grid based) to process files in a blob container:

For example, using the blob trigger with an event subscription on blob containers:

C#

```
[FunctionName("ProcessCatalogData")]
public static async Task Run([BlobTrigger("catalog-uploads/{name}", Source =
BlobTriggerSource.EventGrid, Connection = "
<NAMED_STORAGE_CONNECTION>")]Stream myCatalogData, string name, ILogger log)
{
    log.LogInformation($"C# Blob trigger function Processed blob\n Name:
{name} \n Size: {myCatalogData.Length} Bytes");

    using (var reader = new StreamReader(myCatalogData))
    {
        var catalogEntry = await reader.ReadLineAsync();
        while(catalogEntry !=null)
        {
            // Process the catalog entry
            // ...

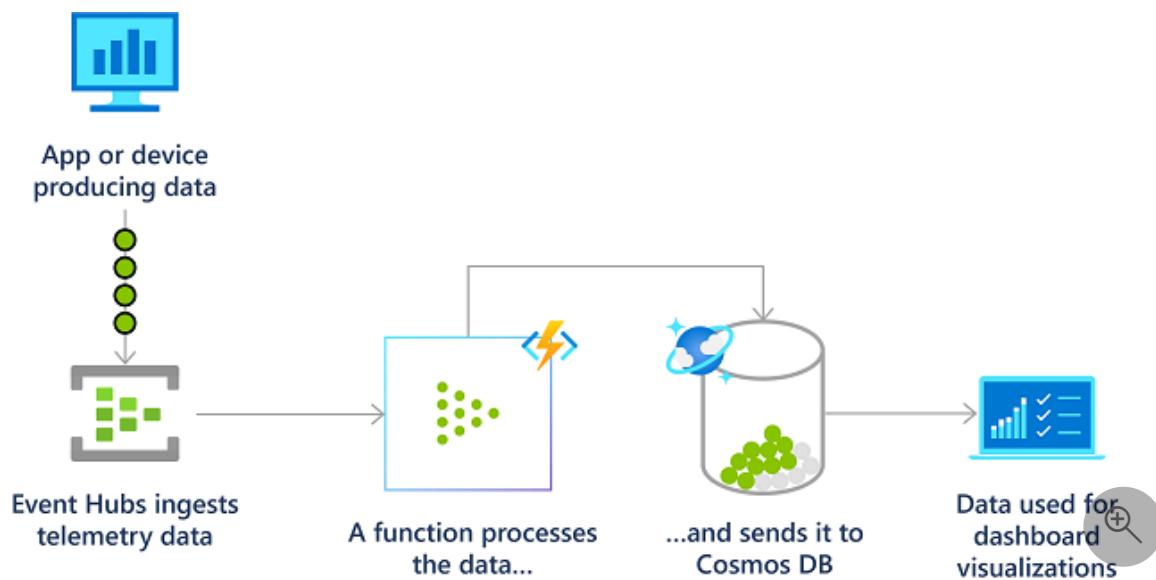
            catalogEntry = await reader.ReadLineAsync();
        }
    }
}
```

- Event-based Blob storage triggered function that converts PDF documents to text at scale ↗
- Upload and analyze a file with Azure Functions and Blob Storage
- Automate resizing uploaded images using Event Grid
- Trigger Azure Functions on blob containers using an event subscription

Real-time stream and event processing

So much telemetry is generated and collected from cloud applications, IoT devices, and networking devices. Azure Functions can process that data in near real-time as the hot path, then store it in [Azure Cosmos DB](#) for use in an analytics dashboard.

Your functions can also use low-latency event triggers, like Event Grid, and real-time outputs like SignalR to process data in near-real-time.



For example, using the event hubs trigger to read from an event hub and the output binding to write to an event hub after debatching and transforming the events:

C#

```
[FunctionName("ProcessorFunction")]
public static async Task Run(
    [EventHubTrigger(
        "%Input_EH_Name%",
        Connection = "InputEventHubConnectionString",
        ConsumerGroup = "%Input_EH_ConsumerGroup%")] EventData[]
    inputMessages,
    [EventHub(
        "%Output_EH_Name%",
        Connection = "OutputEventHubConnectionString")]
    IAsyncCollector<SensorDataRecord> outputMessages,
    PartitionContext partitionContext,
    ILogger log)
{
    var debatcher = new Debatcher(log);
    var debatchedMessages = await debatcher.Debatch(inputMessages,
partitionContext.PartitionId);

    var xformer = new Transformer(log);
    await xformer.Transform(debatchedMessages, partitionContext.PartitionId,
outputMessages);
}
```

- Service Bus trigger using virtual network integration ↗
- Streaming at scale with Azure Event Hubs, Functions and Azure SQL ↗
- Streaming at scale with Azure Event Hubs, Functions and Cosmos DB ↗
- Streaming at scale with Azure Event Hubs with Kafka producer, Functions with Kafka trigger and Cosmos DB ↗
- Streaming at scale with Azure IoT Hub, Functions and Azure SQL ↗

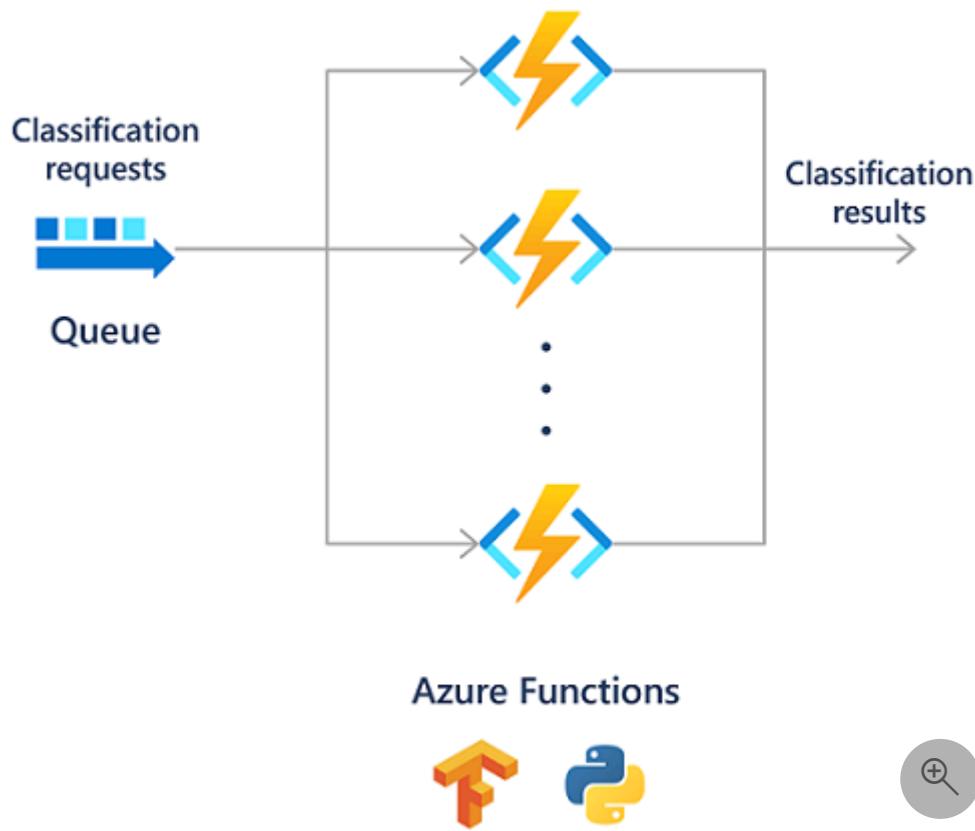
- Azure Event Hubs trigger for Azure Functions
- Apache Kafka trigger for Azure Functions

Machine learning and AI

Besides data processing, Azure Functions can be used to infer on models. The [Azure OpenAI binding extension](#) lets easily integrate features and behaviors of the [Azure OpenAI service](#) into your function code executions.

Functions can connect to an OpenAI resources to enable text and chat completions, use assistants, and leverage embeddings and semantic search.

A function might also call a TensorFlow model or Azure AI services to process and classify a stream of images.



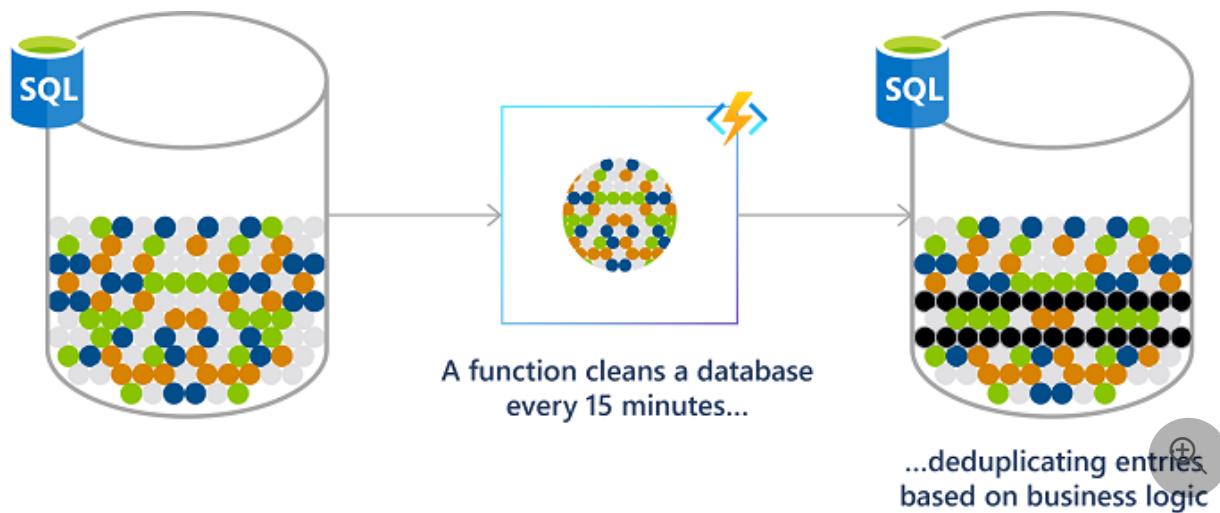
- Tutorial: [Text completion using Azure OpenAI](#)
- Sample: [Upload text files and access data using various OpenAI features ↗](#)
- Sample: [Text summarization using AI Cognitive Language Service ↗](#)
- Sample: [Text completion using Azure OpenAI ↗](#)
- Sample: [Provide assistant skills to your model ↗](#)
- Sample: [Generate embeddings ↗](#)
- Sample: [Leverage semantic search ↗](#)

Run scheduled tasks

Functions enables you to run your code based on a [cron schedule](#) that you define.

Check out how to [Create a function in the Azure portal that runs on a schedule](#).

A financial services customer database, for example, might be analyzed for duplicate entries every 15 minutes to avoid multiple communications going out to the same customer.



C#

```
[FunctionName("TimerTriggerCSharp")]
public static void Run([TimerTrigger("0 */15 * * *")]TimerInfo myTimer,
ILogger log)
{
    if (myTimer.IsPastDue)
    {
        log.LogInformation("Timer is running late!");
    }
    log.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");

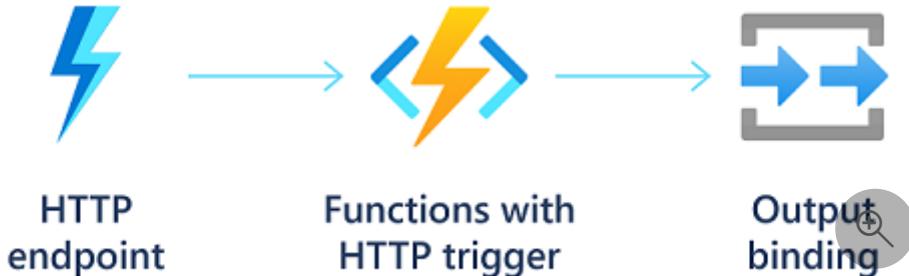
    // Perform the database deduplication
}
```

- Timer trigger for Azure Functions

Build a scalable web API

An HTTP triggered function defines an HTTP endpoint. These endpoints run function code that can connect to other services directly or by using binding extensions. You can compose the endpoints into a web-based API.

You can also use an HTTP triggered function endpoint as a webhook integration, such as GitHub webhooks. In this way, you can create functions that process data from GitHub events. To learn more, see [Monitor GitHub events by using a webhook with Azure Functions](#).



For examples, see the following:

C#

```
[FunctionName("InsertName")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post")] HttpRequest req,
    [CosmosDB(
        databaseName: "my-database",
        collectionName: "my-container",
        ConnectionStringSetting =
    "CosmosDbConnectionString")]IAsyncCollector<dynamic> documentsOut,
    ILogger log)
{
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    string name = data?.name;

    if (name == null)
    {
        return new BadRequestObjectResult("Please pass a name in the request
body json");
    }

    // Add a JSON document to the output container.
    await documentsOut.AddAsync(new
    {
        // create a random ID
        id = System.Guid.NewGuid().ToString(),
        name = name
    });
}

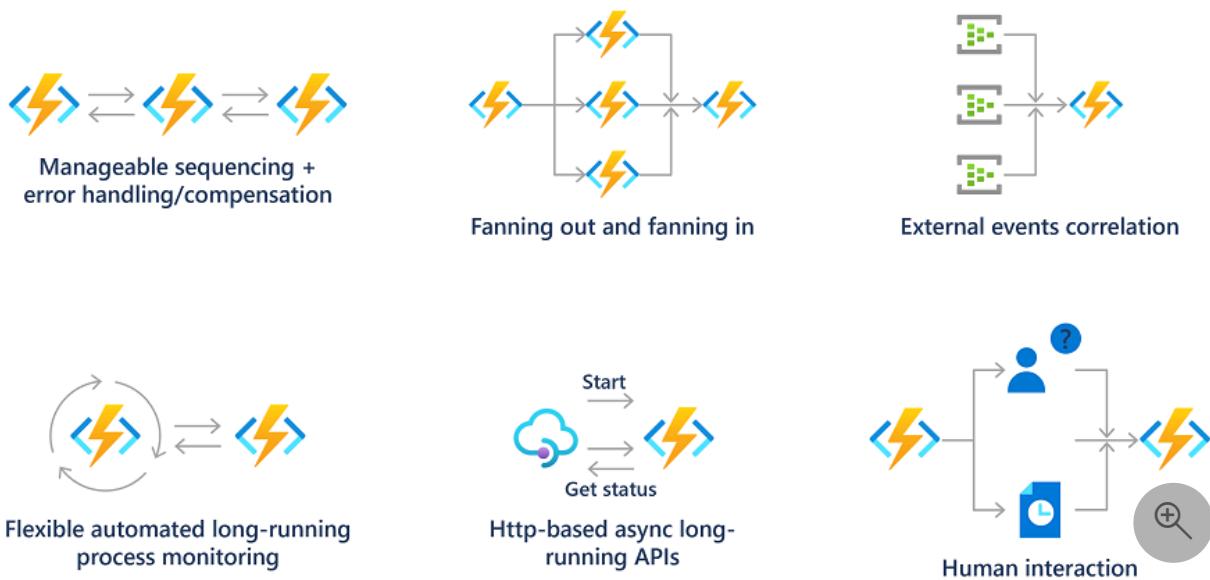
return new OkResult();
}
```

- Article: [Create serverless APIs in Visual Studio using Azure Functions and API Management integration](#)

- Training: [Expose multiple function apps as a consistent API by using Azure API Management](#)
- Sample: [Web application with a C# API and Azure SQL DB on Static Web Apps and Functions](#)
- Azure Functions HTTP trigger

Build a serverless workflow

Functions is often the compute component in a serverless workflow topology, such as a Logic Apps workflow. You can also create long-running orchestrations using the Durable Functions extension. For more information, see [Durable Functions overview](#).



- Tutorial: [Create a function to integrate with Azure Logic Apps](#)
- Quickstart: [Create your first durable function in Azure using C#](#)
- Training: [Deploy serverless APIs with Azure Functions, Logic Apps, and Azure SQL Database](#)

Respond to database changes

There are processes where you might need to log, audit, or perform some other operation when stored data changes. Functions triggers provide a good way to get notified of data changes to initial such an operation.



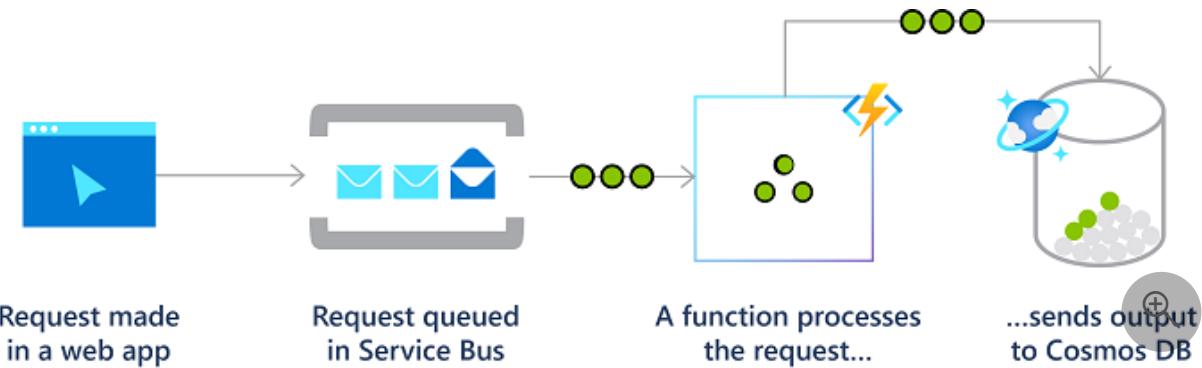
Consider the following examples:

- Article: [Connect Azure Functions to Azure Cosmos DB using Visual Studio Code](#)
- Article: [Connect Azure Functions to Azure SQL Database using Visual Studio Code](#)
- Article: [Use Azure Functions to clean-up an Azure SQL Database](#)

Create reliable message systems

You can use Functions with Azure messaging services to create advanced event-driven messaging solutions.

For example, you can use triggers on Azure Storage queues as a way to chain together a series of function executions. Or use service bus queues and triggers for an online ordering system.



The following article shows how to write output to a storage queue.

- Article: [Connect Azure Functions to Azure Storage using Visual Studio Code](#)
- Article: [Create a function triggered by Azure Queue storage \(Azure portal\)](#)

And these articles show how to trigger from an Azure Service Bus queue or topic.

- [Azure Service Bus trigger for Azure Functions](#)

Next steps

[Getting started with Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Getting started with Azure Functions

Article • 09/19/2024

Azure Functions allows you to implement your system's logic as event-driven, readily available blocks of code. These code blocks are called "functions". This article is to help you find your way to the most helpful Azure Functions content as quickly as possible. For more general information about Azure Functions, see the [Introduction to Azure Functions](#).

Make sure to choose your preferred development language at the top of the article.

Create your first function

Complete one of our quickstart articles to create and deploy your first functions in less than five minutes.

You can create C# functions by using one of the following tools:

- [Azure Developer CLI \(azd\)](#)
- [Command line](#)
- [Visual Studio](#)
- [Visual Studio Code](#)

Review end-to-end samples

These sites let you browse existing functions reference projects and samples in your desired language:

- [Awesome azd template library ↗](#)
- [Azure Community Library ↗](#)
- [Azure Samples Browser](#)

Explore an interactive tutorial

Complete one of the following interactive training modules to learn more about Functions:

- [Choose the best Azure serverless technology for your business scenario](#)
- [Well-Architected Framework - Performance efficiency](#)
- [Execute an Azure Function with triggers](#)

To learn even more, see the full listing of interactive tutorials.

Related content

Learn more about developing functions by reviewing one of these C# reference articles:

- [In-process C# class library functions](#)
- [Isolated worker process C# class library functions](#)

You might also be interested in these articles:

- [Deploying Azure Functions](#)
- [Monitoring Azure Functions](#)
- [Performance and reliability](#)
- [Securing Azure Functions](#)
- [Durable Functions](#)

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

What are Durable Functions?

Article • 08/04/2023

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless compute environment. The extension lets you define stateful workflows by writing *orchestrator functions* and stateful entities by writing *entity functions* using the Azure Functions programming model. Behind the scenes, the extension manages state, checkpoints, and restarts for you, allowing you to focus on your business logic.

Supported languages

Durable Functions is designed to work with all Azure Functions programming languages but may have different minimum requirements for each language. The following table shows the minimum supported app configurations:

Language stack	Azure Functions Runtime versions	Language worker version	Minimum bundles version
.NET / C# / F#	Functions 1.0+	In-process Out-of-process	n/a
JavaScript/TypeScript (V3 prog. model)	Functions 2.0+	Node 8+	2.x bundles
JavaScript/TypeScript (V4 prog. model)	Functions 4.16.5+	Node 18+	3.15+ bundles
Python	Functions 2.0+	Python 3.7+	2.x bundles
Python (V2 prog. model)	Functions 4.0+	Python 3.7+	3.15+ bundles
PowerShell	Functions 3.0+	PowerShell 7+	2.x bundles
Java	Functions 4.0+	Java 8+	4.x bundles

Like Azure Functions, there are templates to help you develop Durable Functions using [Visual Studio](#), [Visual Studio Code](#), and the [Azure portal](#).

Application patterns

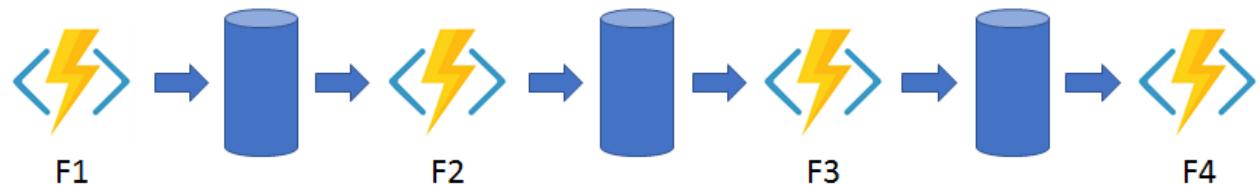
The primary use case for Durable Functions is simplifying complex, stateful coordination requirements in serverless applications. The following sections describe typical

application patterns that can benefit from Durable Functions:

- Function chaining
- Fan-out/fan-in
- Async HTTP APIs
- Monitoring
- Human interaction
- Aggregator (stateful entities)

Pattern #1: Function chaining

In the function chaining pattern, a sequence of functions executes in a specific order. In this pattern, the output of one function is applied to the input of another function. The use of queues between each function ensures that the system stays durable and scalable, even though there is a flow of control from one function to the next.



You can use Durable Functions to implement the function chaining pattern concisely as shown in the following example.

In this example, the values `F1`, `F2`, `F3`, and `F4` are the names of other functions in the same function app. You can implement control flow by using normal imperative coding constructs. Code executes from the top down. The code can involve existing language control flow semantics, like conditionals and loops. You can include error handling logic in `try/catch/finally` blocks.

C# (InProc)

C#

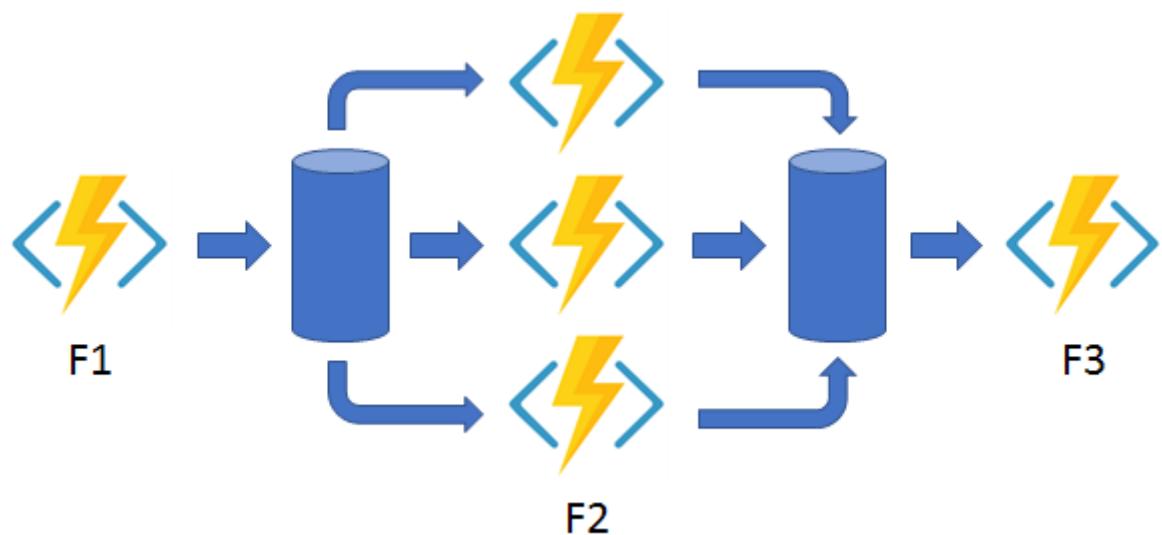
```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return await context.CallActivityAsync<object>("F4", z);
    }
}
```

```
        }
        catch (Exception)
        {
            // Error handling or compensation goes here.
        }
    }
```

You can use the `context` parameter to invoke other functions by name, pass parameters, and return function output. Each time the code calls `await`, the Durable Functions framework checkpoints the progress of the current function instance. If the process or virtual machine recycles midway through the execution, the function instance resumes from the preceding `await` call. For more information, see the next section, Pattern #2: Fan out/fan in.

Pattern #2: Fan out/fan in

In the fan out/fan in pattern, you execute multiple functions in parallel and then wait for all functions to finish. Often, some aggregation work is done on the results that are returned from the functions.



With normal functions, you can fan out by having the function send multiple messages to a queue. Fanning back in is much more challenging. To fan in, in a normal function, you write code to track when the queue-triggered functions end, and then store function outputs.

The Durable Functions extension handles this pattern with relatively simple code:

C# (InProc)

C#

```
[FunctionName("FanOutFanIn")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var parallelTasks = new List<Task<int>>();

    // Get a list of N work items to process in parallel.
    object[] workBatch = await context.CallActivityAsync<object[]>("F1",
null);
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = context.CallActivityAsync<int>("F2",
workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    int sum = parallelTasks.Sum(t => t.Result);
    await context.CallActivityAsync("F3", sum);
}
```

The fan-out work is distributed to multiple instances of the `F2` function. The work is tracked by using a dynamic list of tasks. `Task.WhenAll` is called to wait for all the called functions to finish. Then, the `F2` function outputs are aggregated from the dynamic task list and passed to the `F3` function.

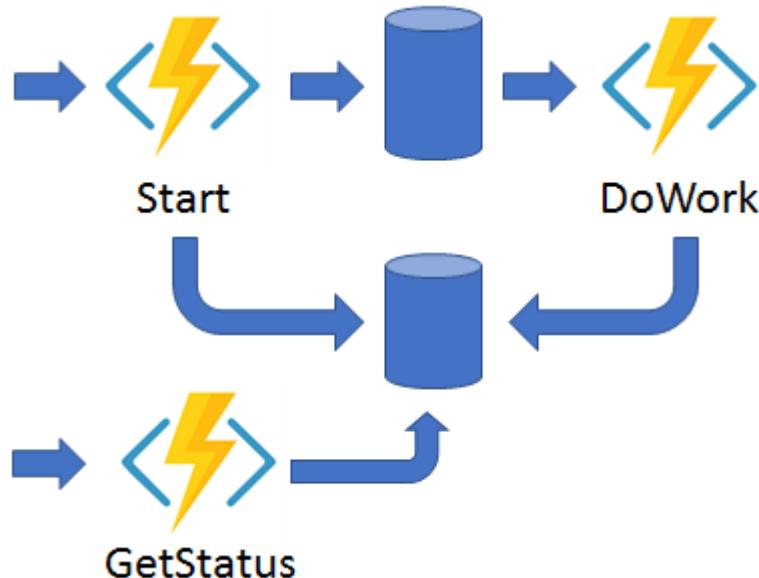
The automatic checkpointing that happens at the `await` call on `Task.WhenAll` ensures that a potential midway crash or reboot doesn't require restarting an already completed task.

ⓘ Note

In rare circumstances, it's possible that a crash could happen in the window after an activity function completes but before its completion is saved into the orchestration history. If this happens, the activity function would re-run from the beginning after the process recovers.

Pattern #3: Async HTTP APIs

The async HTTP API pattern addresses the problem of coordinating the state of long-running operations with external clients. A common way to implement this pattern is by having an HTTP endpoint trigger the long-running action. Then, redirect the client to a status endpoint that the client polls to learn when the operation is finished.



Durable Functions provides **built-in support** for this pattern, simplifying or even removing the code you need to write to interact with long-running function executions. For example, the Durable Functions quickstart samples ([C#](#), [JavaScript](#), [TypeScript](#), [Python](#), [PowerShell](#), and [Java](#)) show a simple REST command that you can use to start new orchestrator function instances. After an instance starts, the extension exposes webhook HTTP APIs that query the orchestrator function status.

The following example shows REST commands that start an orchestrator and query its status. For clarity, some protocol details are omitted from the example.

```
> curl -X POST https://myfunc.azurewebsites.net/api/orchestrators/DoWork -H "Content-Length: 0" -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec

{"id":"b79baf67f717453ca9e86c5da21e03ec", ...}

> curl
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
```

```
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79b  
af67f717453ca9e86c5da21e03ec
```

```
{"runtimeStatus": "Running", "lastUpdatedTime": "2019-03-16T21:20:47Z", ...}
```

```
> curl
```

```
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79b  
af67f717453ca9e86c5da21e03ec -i
```

```
HTTP/1.1 200 OK
```

```
Content-Length: 175
```

```
Content-Type: application/json
```

```
{"runtimeStatus": "Completed", "lastUpdatedTime": "2019-03-16T21:20:57Z", ...}
```

Because the Durable Functions runtime manages state for you, you don't need to implement your own status-tracking mechanism.

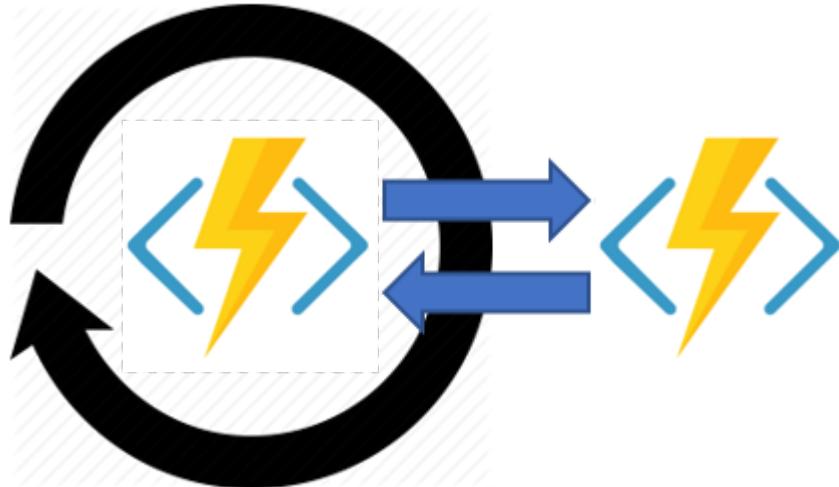
The Durable Functions extension exposes built-in HTTP APIs that manage long-running orchestrations. You can alternatively implement this pattern yourself by using your own function triggers (such as HTTP, a queue, or Azure Event Hubs) and the [durable client binding](#). For example, you might use a queue message to trigger termination. Or, you might use an HTTP trigger that's protected by an Azure Active Directory authentication policy instead of the built-in HTTP APIs that use a generated key for authentication.

For more information, see the [HTTP features](#) article, which explains how you can expose asynchronous, long-running processes over HTTP using the Durable Functions extension.

Pattern #4: Monitor

The monitor pattern refers to a flexible, recurring process in a workflow. An example is polling until specific conditions are met. You can use a regular [timer trigger](#) to address a basic scenario, such as a periodic cleanup job, but its interval is static and managing instance lifetimes becomes complex. You can use Durable Functions to create flexible recurrence intervals, manage task lifetimes, and create multiple monitor processes from a single orchestration.

An example of the monitor pattern is to reverse the earlier async HTTP API scenario. Instead of exposing an endpoint for an external client to monitor a long-running operation, the long-running monitor consumes an external endpoint, and then waits for a state change.



In a few lines of code, you can use Durable Functions to create multiple monitors that observe arbitrary endpoints. The monitors can end execution when a condition is met, or another function can use the durable orchestration client to terminate the monitors. You can change a monitor's `wait` interval based on a specific condition (for example, exponential backoff.)

The following code implements a basic monitor:

C# (InProc)

```
C#  
  
[FunctionName("MonitorJobStatus")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    int jobId = context.GetInput<int>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

    while (context.CurrentUtcDateTime < expiryTime)
    {
        var jobStatus = await context.CallActivityAsync<string>
("GetJobStatus", jobId);
        if (jobStatus == "Completed")
        {
            // Perform an action when a condition is met.
            await context.CallActivityAsync("SendAlert", jobId);
            break;
        }

        // Orchestration sleeps until this time.
        var nextCheck =
            context.CurrentUtcDateTime.AddSeconds(pollingInterval);
```

```

        await context.CreateTimer(nextCheck, CancellationToken.None);
    }

    // Perform more work here, or let the orchestration end.
}

```

When a request is received, a new orchestration instance is created for that job ID. The instance polls a status until either a condition is met or until a timeout expires. A durable timer controls the polling interval. Then, more work can be performed, or the orchestration can end.

Pattern #5: Human interaction

Many automated processes involve some kind of human interaction. Involving humans in an automated process is tricky because people aren't as highly available and as responsive as cloud services. An automated process might allow for this interaction by using timeouts and compensation logic.

An approval process is an example of a business process that involves human interaction. Approval from a manager might be required for an expense report that exceeds a certain dollar amount. If the manager doesn't approve the expense report within 72 hours (maybe the manager went on vacation), an escalation process kicks in to get the approval from someone else (perhaps the manager's manager).



You can implement the pattern in this example by using an orchestrator function. The orchestrator uses a [durable timer](#) to request approval. The orchestrator escalates if timeout occurs. The orchestrator waits for an [external event](#), such as a notification that's generated by a human interaction.

These examples create an approval process to demonstrate the human interaction pattern:

```
C#  
  
[FunctionName("ApprovalWorkflow")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("RequestApproval", null);
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = context.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = context.CreateTimer(dueTime,
            timeoutCts.Token);  
  

        Task<bool> approvalEvent = context.WaitForExternalEvent<bool>
            ("ApprovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent,
            durableTimeout))
        {
            timeoutCts.Cancel();
            await context.CallActivityAsync("ProcessApproval",
                approvalEvent.Result);
        }
        else
        {
            await context.CallActivityAsync("Escalate", null);
        }
    }
}
```

To create the durable timer, call `context.CreateTimer`. The notification is received by `context.WaitForExternalEvent`. Then, `Task.WhenAny` is called to decide whether to escalate (timeout happens first) or process the approval (the approval is received before timeout).

ⓘ Note

There is no charge for time spent waiting for external events when running in the Consumption plan.

An external client can deliver the event notification to a waiting orchestrator function by using the [built-in HTTP APIs](#):

```
curl -d "true"  
http://localhost:7071/runtime/webhooks/durabletask/instances/{instanceId}/ra  
iseEvent/ApprovalEvent -H "Content-Type: application/json"
```

An event can also be raised using the durable orchestration client from another function in the same function app:

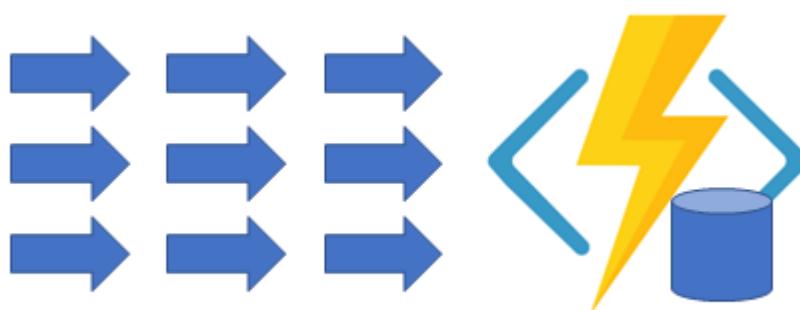
C# (InProc)

C#

```
[FunctionName("RaiseEventToOrchestration")]
public static async Task Run(
    [HttpTrigger] string instanceId,
    [DurableClient] IDurableOrchestrationClient client)
{
    bool isApproved = true;
    await client.RaiseEventAsync(instanceId, "ApprovalEvent",
        isApproved);
}
```

Pattern #6: Aggregator (stateful entities)

The sixth pattern is about aggregating event data over a period of time into a single, addressable *entity*. In this pattern, the data being aggregated may come from multiple sources, may be delivered in batches, or may be scattered over long-periods of time. The aggregator might need to take action on event data as it arrives, and external clients may need to query the aggregated data.



The tricky thing about trying to implement this pattern with normal, stateless functions is that concurrency control becomes a huge challenge. Not only do you need to worry

about multiple threads modifying the same data at the same time, you also need to worry about ensuring that the aggregator only runs on a single VM at a time.

You can use [Durable entities](#) to easily implement this pattern as a single function.

C# (InProc)

```
C#  
  
[FunctionName("Counter")]
public static void Counter([EntityTrigger] IDurableEntityContext ctx)
{
    int currentValue = ctx.GetState<int>();
    switch (ctx.OperationName.ToLowerInvariant())
    {
        case "add":
            int amount = ctx.GetInput<int>();
            ctx.SetState(currentValue + amount);
            break;
        case "reset":
            ctx.SetState(0);
            break;
        case "get":
            ctx.Return(currentValue);
            break;
    }
}
```

Durable entities can also be modeled as classes in .NET. This model can be useful if the list of operations is fixed and becomes large. The following example is an equivalent implementation of the `Counter` entity using .NET classes and methods.

C#

```
public class Counter
{
    [JsonProperty("value")]
    public int CurrentValue { get; set; }

    public void Add(int amount) => this.CurrentValue += amount;

    public void Reset() => this.CurrentValue = 0;

    public int Get() => this.CurrentValue;

    [FunctionName(nameof(Counter))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx)
```

```
=> ctx.DispatchAsync<Counter>();  
}
```

Clients can enqueue *operations* for (also known as "signaling") an entity function using the [entity client binding](#).

C# (InProc)

C#

```
[FunctionName("EventHubTriggerCSharp")]
public static async Task Run(
    [EventHubTrigger("device-sensor-events")] EventData eventData,
    [DurableClient] IDurableEntityClient entityClient)
{
    var metricType = (string)eventData.Properties["metric"];
    var delta = BitConverter.ToInt32(eventData.Body,
        eventData.Body.Offset);

    // The "Counter/{metricType}" entity is created on-demand.
    var entityId = new EntityId("Counter", metricType);
    await entityClient.SignalEntityAsync(entityId, "add", delta);
}
```

ⓘ Note

Dynamically generated proxies are also available in .NET for signaling entities in a type-safe way. And in addition to signaling, clients can also query for the state of an entity function using **type-safe methods** on the orchestration client binding.

Entity functions are available in [Durable Functions 2.0](#) and above for C#, JavaScript, and Python.

The technology

Behind the scenes, the Durable Functions extension is built on top of the [Durable Task Framework](#), an open-source library on GitHub that's used to build workflows in code. Like Azure Functions is the serverless evolution of Azure WebJobs, Durable Functions is the serverless evolution of the Durable Task Framework. Microsoft and other organizations use the Durable Task Framework extensively to automate mission-critical processes. It's a natural fit for the serverless Azure Functions environment.

Code constraints

In order to provide reliable and long-running execution guarantees, orchestrator functions have a set of coding rules that must be followed. For more information, see the [Orchestrator function code constraints](#) article.

Billing

Durable Functions are billed the same as Azure Functions. For more information, see [Azure Functions pricing](#). When executing orchestrator functions in the Azure Functions [Consumption plan](#), there are some billing behaviors to be aware of. For more information on these behaviors, see the [Durable Functions billing](#) article.

Jump right in

You can get started with Durable Functions in under 10 minutes by completing one of these language-specific quickstart tutorials:

- [C# using Visual Studio 2019](#)
- [JavaScript using Visual Studio Code](#)
- [TypeScript using Visual Studio Code](#)
- [Python using Visual Studio Code](#)
- [PowerShell using Visual Studio Code](#)
- [Java using Maven](#)

In these quickstarts, you locally create and test a "hello world" durable function. You then publish the function code to Azure. The function you create orchestrates and chains together calls to other functions.

Publications

Durable Functions is developed in collaboration with Microsoft Research. As a result, the Durable Functions team actively produces research papers and artifacts; these include:

- [Durable Functions: Semantics for Stateful Serverless](#) (OOPSLA'21)
- [Serverless Workflows with Durable Functions and Netherite](#) (pre-print)

Learn more

The following video highlights the benefits of Durable Functions:

<https://learn.microsoft.com/Shows/Azure-Friday/Durable-Functions-in-Azure-Functions/player>

For a more in-depth discussion of Durable Functions and the underlying technology, see the following video (it's focused on .NET, but the concepts also apply to other supported languages):

<https://learn.microsoft.com/Events/dotnetConf/2018/S204/player>

Because Durable Functions is an advanced extension for [Azure Functions](#), it isn't appropriate for all applications. For a comparison with other Azure orchestration technologies, see [Compare Azure Functions and Azure Logic Apps](#).

Next steps

[Durable Functions function types and features](#)

Choose the right integration and automation services in Azure

Article • 06/09/2023

This article compares the following Microsoft cloud services:

- [Microsoft Power Automate](#) (was Microsoft Flow)
- [Azure Logic Apps](#)
- [Azure Functions](#)
- [Azure App Service WebJobs](#)

All of these services can solve integration problems and automate business processes. They can all define input, actions, conditions, and output. You can run each of them on a schedule or trigger. Each service has unique advantages, and this article explains the differences.

ⓘ Note

If you're looking for a more general comparison between Azure Functions and other Azure compute options:

- [Criteria for choosing an Azure compute service](#)
- [Choosing an Azure compute option for microservices](#)

For a summary and comparison of automation service options in Azure:

- [Choose the Automation services in Azure](#)

Compare Microsoft Power Automate and Azure Logic Apps

Power Automate and Azure Logic Apps are both *designer-first* integration services that can create workflows. Both services integrate with various SaaS and enterprise applications.

Power Automate is built on the Azure Logic Apps platform. Both provide similar workflow designers and [connectors](#).

Power Automate empowers any office worker to perform simple integrations (for example, an approval process on a SharePoint Document Library) without going through

developers or IT. Logic Apps can also enable advanced integrations (for example, B2B processes) where enterprise-level Azure DevOps and security practices are required. It's typical for a business workflow to grow in complexity over time.

The following table helps you determine whether Power Automate or Logic Apps is best for a particular integration:

	Power Automate	Logic Apps
Users	Office workers, business users, SharePoint administrators	Pro integrators and developers, IT pros
Scenarios	Self-service	Advanced integrations
Design tool	In-browser and mobile app, UI only	In-browser, Visual Studio Code , and Visual Studio with code view available
Application lifecycle management (ALM)	Power Platform provides tools that integrate with DevOps and GitHub Actions to let you build automated pipelines in the ALM cycle.	Azure DevOps: source control, testing, support, automation, and manageability in Azure Resource Manager
Admin experience	Manage Power Automate environments and data loss prevention (DLP) policies, track licensing: Admin center	Manage resource groups, connections, access management, and logging: Azure portal
Security	Microsoft 365 security audit logs, DLP, encryption at rest for sensitive data	Security assurance of Azure: Azure security , Microsoft Defender for Cloud , audit logs

Compare Azure Functions and Azure Logic Apps

Functions and Logic Apps are Azure services that enable serverless workloads. Azure Functions is a serverless compute service, whereas Azure Logic Apps is a serverless workflow integration platform. Both can create complex *orchestrations*. An orchestration is a collection of functions, or *actions* in Azure Logic Apps, that you can run to complete a complex task. For example, to process a batch of orders, you might execute many instances of a function in parallel, wait for all instances to finish, and then execute a function that computes a result on the aggregate.

For Azure Functions, you develop orchestrations by writing code and using the [Durable Functions extension](#). For Azure Logic Apps, you create orchestrations by using a GUI or editing configuration files.

You can mix and match services when you build an orchestration, such as calling functions from logic app workflows and calling logic app workflows from functions. Choose how to build each orchestration based on the services' capabilities or your personal preference. The following table lists some key differences between these services:

	Durable Functions	Azure Logic Apps
Development	Code-first (imperative)	Designer-first (declarative)
Connectivity	About a dozen built-in binding types , write code for custom bindings	Large collection of connectors , Enterprise Integration Pack for B2B scenarios, build custom connectors
Actions	Each activity is an Azure function; write code for activity functions	Large collection of ready-made actions
Monitoring	Azure Application Insights	Azure portal , Azure Monitor Logs , Microsoft Defender for Cloud
Management	REST API , Visual Studio	Azure portal , REST API , PowerShell , Visual Studio
Execution context	Can run locally or in the cloud	Runs in Azure, locally, or on premises. For more information, see What is Azure Logic Apps .

Compare Functions and WebJobs

Like Azure Functions, Azure App Service WebJobs with the WebJobs SDK is a *code-first* integration service that is designed for developers. Both are built on [Azure App Service](#) and support features such as [source control integration](#), [authentication](#), and [monitoring with Application Insights integration](#).

WebJobs and the WebJobs SDK

You can use the *WebJobs* feature of App Service to run a script or code in the context of an App Service web app. The *WebJobs SDK* is a framework designed for WebJobs that simplifies the code you write to respond to events in Azure services. For example, you might respond to the creation of an image blob in Azure Storage by creating a thumbnail image. The WebJobs SDK runs as a .NET console application, which you can deploy to a WebJob.

WebJobs and the WebJobs SDK work best together, but you can use WebJobs without the WebJobs SDK and vice versa. A WebJob can run any program or script that runs in the App Service sandbox. A WebJobs SDK console application can run anywhere console applications run, such as on-premises servers.

Comparison table

Azure Functions is built on the WebJobs SDK, so it shares many of the same event triggers and connections to other Azure services. Here are some factors to consider when you're choosing between Azure Functions and WebJobs with the WebJobs SDK:

	Functions	WebJobs with WebJobs SDK
Serverless app model  with automatic scaling	✓	
Develop and test in browser	✓	
Pay-per-use pricing	✓	
Integration with Logic Apps	✓	
Trigger events	Timer Azure Storage queues and blobs Azure Service Bus queues and topics Azure Cosmos DB Azure Event Hubs HTTP/WebHook (GitHub, Slack) Azure Event Grid	Timer Azure Storage queues and blobs Azure Service Bus queues and topics Azure Cosmos DB Azure Event Hubs File system 
Supported languages	C# F# JavaScript Java Python PowerShell	C# ¹
Package managers	npm and NuGet	NuGet ²

¹ WebJobs (without the WebJobs SDK) supports languages such as C#, Java, JavaScript, Bash, .cmd, .bat, PowerShell, PHP, TypeScript, Python, and more. A WebJob can run any program or script that can run in the App Service sandbox.

² WebJobs (without the WebJobs SDK) supports npm and NuGet.

Summary

Azure Functions offers more developer productivity than Azure App Service WebJobs does. It also offers more options for programming languages, development environments, Azure service integration, and pricing. For most scenarios, it's the best choice.

Here are two scenarios for which WebJobs may be the best choice:

- You need more control over the code that listens for events, the `JobHost` object. Functions offers a limited number of ways to customize `JobHost` behavior in the `host.json` file. Sometimes you need to do things that can't be specified by a string in a JSON file. For example, only the WebJobs SDK lets you configure a custom retry policy for Azure Storage.
- You have an App Service app for which you want to run code snippets, and you want to manage them together in the same Azure DevOps environment.

For other scenarios where you want to run code snippets for integrating Azure or third-party services, choose Azure Functions over WebJobs with the WebJobs SDK.

Power Automate, Logic Apps, Functions, and WebJobs together

You don't have to choose just one of these services. They integrate with each other and with external services.

A Power Automate flow can call an Azure Logic Apps workflow. An Azure Logic Apps workflow can call a function in Azure Functions, and vice versa. For example, see [Create a function that integrates with Azure Logic Apps](#).

Between Power Automate, Logic Apps, and Functions, the integration experience between these services continues to improve over time. You can build a component in one service and use that component in the other services.

You can get more information on integration services by using the following links:

- [Leveraging Azure Functions & Azure App Service for integration scenarios by Christopher Anderson ↗](#)
- [Integrations Made Simple by Charles Lamanna ↗](#)
- [Logic Apps Live webcast ↗](#)

- [Power Automate frequently asked questions](#)

Next steps

Get started by creating your first flow, logic app workflow, or function app. Select any of the following links:

- [Get started with Power Automate](#)
- [Create an example Consumption logic app workflow](#)
- [Create your first Azure function](#)

Azure Functions hosting options

Article • 07/16/2024

When you create a function app in Azure, you must choose a hosting option for your app. Azure provides you with these hosting options for your function code:

[+] Expand table

Hosting option	Service	Availability	Container support
Consumption plan	Azure Functions	Generally available (GA)	None
Flex Consumption plan	Azure Functions	Preview	None
Premium plan	Azure Functions	GA	Linux
Dedicated plan	Azure Functions	GA	Linux
Container Apps	Azure Container Apps	GA	Linux

Azure Functions hosting options are facilitated by Azure App Service infrastructure on both Linux and Windows virtual machines. The hosting option you choose dictates the following behaviors:

- How your function app is scaled.
- The resources available to each function app instance.
- Support for advanced functionality, such as Azure Virtual Network connectivity.
- Support for Linux containers.

The plan you choose also impacts the costs for running your function code. For more information, see [Billing](#).

This article provides a detailed comparison between the various hosting options. To learn more about running and managing your function code in Linux containers, see [Linux container support in Azure Functions](#).

Overview of plans

The following is a summary of the benefits of the various options for Azure Functions hosting:

[+] Expand table

Option	Benefits
Consumption plan	<p>Pay for compute resources only when your functions are running (pay-as-you-go) with automatic scale.</p> <p>On the Consumption plan, instances of the Functions host are dynamically added and removed based on the number of incoming events.</p> <ul style="list-style-type: none"> ✓ Default hosting plan that provides true <i>serverless</i> hosting. ✓ Pay only when your functions are running. ✓ Scales automatically, even during periods of high load.
Flex Consumption plan	<p>Get high scalability with compute choices, virtual networking, and pay-as-you-go billing.</p> <p>On the Flex Consumption plan, instances of the Functions host are dynamically added and removed based on the configured per instance concurrency and the number of incoming events.</p> <ul style="list-style-type: none"> ✓ Reduce cold starts by specifying a number of pre-provisioned (always ready) instances. ✓ Supports virtual networking for added security. ✓ Pay when your functions are running. ✓ Scales automatically, even during periods of high load.
Premium plan	<p>Automatically scales based on demand using prewarmed workers, which run applications with no delay after being idle, runs on more powerful instances, and connects to virtual networks.</p> <p>Consider the Azure Functions Premium plan in the following situations:</p> <ul style="list-style-type: none"> ✓ Your function apps run continuously, or nearly continuously. ✓ You want more control of your instances and want to deploy multiple function apps on the same plan with event-driven scaling. ✓ You have a high number of small executions and a high execution bill, but low GB seconds in the Consumption plan. ✓ You need more CPU or memory options than are provided by consumption plans. ✓ Your code needs to run longer than the maximum execution time allowed on the Consumption plan. ✓ You require virtual network connectivity. ✓ You want to provide a custom Linux image in which to run your functions.
Dedicated plan	<p>Run your functions within an App Service plan at regular App Service plan rates.</p> <p>Best for long-running scenarios where Durable Functions can't be used.</p> <p>Consider an App Service plan in the following situations:</p> <ul style="list-style-type: none"> ✓ You have existing and underutilized virtual machines that are already running

Option	Benefits
	<p>other App Service instances.</p> <ul style="list-style-type: none"> ✓ You must have fully predictable billing, or you need to manually scale instances. ✓ You want to run multiple web apps and function apps on the same plan ✓ You need access to larger compute size choices. ✓ Full compute isolation and secure network access provided by an App Service Environment (ASE). ✓ Very high memory usage and high scale (ASE).
Container Apps	<p>Create and deploy containerized function apps in a fully managed environment hosted by Azure Container Apps.</p> <p>Use the Azure Functions programming model to build event-driven, serverless, cloud native function apps. Run your functions alongside other microservices, APIs, websites, and workflows as container-hosted programs. Consider hosting your functions on Container Apps in the following situations:</p> <ul style="list-style-type: none"> ✓ You want to package custom libraries with your function code to support line-of-business apps. ✓ You need to migrate code execution from on-premises or legacy apps to cloud native microservices running in containers. ✓ When you want to avoid the overhead and complexity of managing Kubernetes clusters and dedicated compute. ✓ Your functions need high-end processing power provided by dedicated GPU compute resources.

The remaining tables in this article compare hosting options based on various features and behaviors.

Operating system support

This table shows operating system support for the hosting options.

[\[\]](#) Expand table

Hosting	Linux ¹ deployment	Windows ² deployment
Consumption plan	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only <input type="checkbox"/> Container (not supported) 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only
Flex Consumption plan	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only <input type="checkbox"/> Container (not supported) 	<ul style="list-style-type: none"> <input type="checkbox"/> Not supported
Premium plan	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only <input checked="" type="checkbox"/> Container 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only

Hosting	Linux ¹ deployment	Windows ² deployment
Dedicated plan	<input checked="" type="checkbox"/> Code-only <input checked="" type="checkbox"/> Container	<input checked="" type="checkbox"/> Code-only
Container Apps	<input checked="" type="checkbox"/> Container-only	 Not supported

1. Linux is the only supported operating system for the [Python runtime stack](#).
2. Windows deployments are code-only. Functions doesn't currently support Windows containers.

Function app timeout duration

The timeout duration for functions in a function app is defined by the `functionTimeout` property in the [host.json](#) project file. This property applies specifically to function executions. After the trigger starts function execution, the function needs to return/respond within the timeout duration. For more information, see [Improve Azure Functions performance and reliability](#).

The following table shows the default and maximum values (in minutes) for specific plans:

[+] [Expand table](#)

Plan	Default	Maximum ¹
Consumption plan	5	10
Flex Consumption plan	30	Unlimited ³
Premium plan	30 ²	Unlimited ³
Dedicated plan	30 ²	Unlimited ³
Container Apps	30 ⁵	Unlimited ³

1. Regardless of the function app timeout setting, 230 seconds is the maximum amount of time that an HTTP triggered function can take to respond to a request. This is because of the [default idle timeout of Azure Load Balancer](#). For longer processing times, consider using the [Durable Functions async pattern](#) or [defer the actual work and return an immediate response](#).
2. The default timeout for version 1.x of the Functions runtime is *unlimited*.
3. Guaranteed for up to 60 minutes. [OS and runtime patching](#), vulnerability patching, and [scale in behaviors](#) can still cancel function executions so [ensure to write robust](#)

functions.

4. In a Flex Consumption plan, the host doesn't enforce an execution time limit. However, there are currently no guarantees because the platform might need to terminate your instances during scale-in, deployments, or to apply updates.
5. When the [minimum number of replicas](#) is set to zero, the default timeout depends on the specific triggers used in the app.

Language support

For details on current native language stack support in Functions, see [Supported languages in Azure Functions](#).

Scale

The following table compares the scaling behaviors of the various hosting plans. Maximum instances are given on a per-function app (Consumption) or per-plan (Premium/Dedicated) basis, unless otherwise indicated.

[+] [Expand table](#)

Plan	Scale out	Max # instances
Consumption plan	Event driven . Scales out automatically, even during periods of high load. Functions infrastructure scales CPU and memory resources by adding more instances of the Functions host, based on the number of incoming trigger events.	Windows: 200 Linux: 100 ¹
Flex Consumption plan	Per-function scaling . Event-driven scaling decisions are calculated on a per-function basis, which provides a more deterministic way of scaling the functions in your app. With the exception of HTTP, Blob storage (Event Grid), and Durable Functions, all other function trigger types in your app scale on independent instances. All HTTP triggers in your app scale together as a group on the same instances, as do all Blob storage (Event Grid) triggers. All Durable Functions triggers also share instances and scale together.	Limited only by total memory usage of all instances across a given region. For more information, see Instance memory .
Premium plan	Event driven . Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding more instances of the Functions host, based on the number of events that its functions are triggered on.	Windows: 100 Linux: 20-100 ²

Plan	Scale out	Max # instances
Dedicated plan ³	Manual/autoscale	10-30 100 (ASE)
Container Apps	Event driven. Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding more instances of the Functions host, based on the number of events that its functions are triggered on.	300-1000 ⁴

1. During scale-out, there's currently a limit of 500 instances per subscription per hour for Linux apps on a Consumption plan.
2. In some regions, Linux apps on a Premium plan can scale to 100 instances. For more information, see the [Premium plan article](#).
3. For specific limits for the various App Service plan options, see the [App Service plan limits](#).
4. On Container Apps, the default is 10 instances, but you can set the [maximum number of replicas](#), which has an overall maximum of 1000. This setting is honored as long as there's enough cores quota available. When you create your function app from the Azure portal you're limited to 300 instances.

Cold start behavior

[+] [Expand table](#)

Plan	Details
Consumption plan	Apps can scale to zero when idle, meaning some requests might have more latency at startup. The consumption plan does have some optimizations to help decrease cold start time, including pulling from prewarmed placeholder functions that already have the host and language processes running.
Flex Consumption plan	Supports always ready instances to reduce the delay when provisioning new instances.
Premium plan	Supports always ready instances to avoid cold starts by letting you maintain one or more <i>perpetually warm</i> instances.
Dedicated plan	When running in a Dedicated plan, the Functions host can run continuously on a prescribed number of instances, which means that cold start isn't really an issue.
Container Apps	Depends on the minimum number of replicas : <ul style="list-style-type: none"> • When set to zero: apps can scale to zero when idle and some requests might

Plan	Details
	<p>have more latency at startup.</p> <ul style="list-style-type: none"> When set to one or more: the host process runs continuously, which means that cold start isn't an issue.

Service limits

[\[\]](#) Expand table

Resource	Consumption plan	Flex Consumption plan ¹³	Premium plan	Dedicated plan/ASE	Container Apps
Default timeout duration (min)	5	30	30	30 ¹	30 ¹⁷
Max timeout duration (min)	10	unbounded ¹⁶	unbounded ⁸	unbounded ²	unbounded ¹⁸
Max outbound connections (per instance)	600 active (1200 total)	unbounded	unbounded	unbounded	unbounded
Max request size (MB) ³	100	100	100	100	100
Max query string length ³	4096	4096	4096	4096	4096
Max request URL length ³	8192	8192	8192	8192	8192
ACU per instance	100	varies	210-840	100-840/210-250 ⁹	varies
Max memory (GB per instance)	1.5	4 ¹⁴	3.5-14	1.75-14/3.5-14	varies
Max instance count (Windows/Linux)	200/100	1000 ¹⁵	100/20	varies by SKU/100 ¹⁰	10-300 ¹⁹
Function apps per plan ¹²	100	100	100	unbounded ⁴	unbounded ⁴

Resource	Consumption plan	Flex Consumption plan ¹³	Premium plan	Dedicated plan/ASE	Container Apps
App Service plans	100 per region ¹²	n/a	100 per resource group	100 per resource group	n/a
Deployment slots per app ¹¹	2	n/a	3	1-20 ¹⁰	not supported
Storage (temporary) ⁵	0.5 GB	0.8 GB	21-140 GB	11-140 GB	n/a
Storage (persisted)	1 GB ⁶	0 GB ⁶	250 GB	10-1000 GB ¹⁰	n/a
Custom domains per app	500 ⁷	500	500	500	not supported
Custom domain SSL support	unbounded SNI SSL connection included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included	not supported

Notes on service limits:

1. By default, the timeout for the Functions 1.x runtime in an App Service plan is unbounded.
2. Requires the App Service plan be set to [Always On](#). Pay at standard [rates](#).
3. These limits are [set in the host](#).
4. The actual number of function apps that you can host depends on the activity of the apps, the size of the machine instances, and the corresponding resource utilization.
5. The storage limit is the total content size in temporary storage across all apps in the same App Service plan. For Consumption plans on Linux, the storage is currently 1.5 GB.
6. Consumption plan uses an Azure Files share for persisted storage. When you provide your own Azure Files share, the specific share size limits depend on the storage account you set for [WEBSITE_CONTENTAZUREFILECONNECTIONSTRING](#). On Linux, you must [explicitly mount your own Azure Files share](#) for both Flex Consumption and Consumption plans.
7. When your function app is hosted in a [Consumption plan](#), only the CNAME option is supported. For function apps in a [Premium plan](#) or an [App Service plan](#), you can

- map a custom domain using either a CNAME or an A record.
8. Guaranteed for up to 60 minutes.
 9. Workers are roles that host customer apps. Workers are available in three fixed sizes: One vCPU/3.5 GB RAM; Two vCPU/7 GB RAM; Four vCPU/14 GB RAM.
 10. See [App Service limits](#) for details.
 11. Including the production slot.
 12. There's currently a limit of 5000 function apps in a given subscription.
 13. The Flex Consumption plan is currently in preview.
 14. Flex Consumption plan instance sizes are currently defined as either 2,048 MB or 4,096 MB. For more information, see [Instance memory](#).
 15. Flex Consumption plan during preview has a regional subscription quota that limits the total memory usage of all instances across a given region. For more information, see [Instance memory](#).
 16. In a Flex Consumption plan, the host doesn't enforce an execution time limit. However, there are currently no guarantees because the platform might need to terminate your instances during scale-in, deployments, or to apply updates.
 17. When the [minimum number of replicas](#) is set to zero, the default timeout depends on the specific triggers used in the app.
 18. When the [minimum number of replicas](#) is set to one or more.
 19. On Container Apps, you can set the [maximum number of replicas](#), which is honored as long as there's enough cores quota available.

Networking features

[] Expand table

Feature	Consumption plan	Flex Consumption plan	Premium plan	Dedicated plan/ASE	Container Apps*
Inbound IP restrictions	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
Inbound Private Endpoints	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Virtual network integration	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes (Regional)	<input checked="" type="checkbox"/> Yes (Regional)	<input checked="" type="checkbox"/> Yes (Regional and Gateway)	<input checked="" type="checkbox"/> Yes
Virtual network triggers (non-HTTP)	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes

Feature	Consumption plan	Flex Consumption plan	Premium plan	Dedicated plan/ASE	Container Apps*
Hybrid connections (Windows only)	✗ No	✗ No	✓ Yes	✓ Yes	✗ No
Outbound IP restrictions	✗ No	✓ Yes	✓ Yes	✓ Yes	✓ Yes

*For more information, see [Networking in Azure Container Apps environment](#).

Billing

[+] [Expand table](#)

Plan	Details
Consumption plan	Pay only for the time your functions run. Billing is based on number of executions, execution time, and memory used.
Flex Consumption plan	Billing is based on number of executions, the memory of instances when they're actively executing functions, plus the cost of any always ready instances . For more information, see Flex Consumption plan billing .
Premium plan	Premium plan is based on the number of core seconds and memory used across needed and prewarmed instances. At least one instance per plan must always be kept warm. This plan provides the most predictable pricing.
Dedicated plan	You pay the same for function apps in an App Service Plan as you would for other App Service resources, like web apps. For an ASE, there's a flat monthly rate that pays for the infrastructure and doesn't change with the size of the environment. There's also a cost per App Service plan vCPU. All apps hosted in an ASE are in the Isolated pricing SKU. For more information, see the ASE overview article .
Container Apps	Billing in Azure Container Apps is based on your plan type. For more information, see Billing in Azure Container Apps .

For a direct cost comparison between dynamic hosting plans (Consumption, Flex Consumption, and Premium), see the [Azure Functions pricing page](#). For pricing of the various Dedicated plan options, see the [App Service pricing page](#). For pricing Container Apps hosting, see [Azure Container Apps pricing](#).

Limitations for creating new function apps in an existing resource group

In some cases, when trying to create a new hosting plan for your function app in an existing resource group you might receive one of the following errors:

- The pricing tier is not allowed in this resource group
- <SKU_name> workers are not available in resource group
<resource_group_name>

This can happen when the following conditions are met:

- You create a function app in an existing resource group that has ever contained another function app or web app. For example, Linux Consumption apps aren't supported in the same resource group as Linux Dedicated or Linux Premium plans.
- Your new function app is created in the same region as the previous app.
- The previous app is in some way incompatible with your new app. This error can happen between SKUs, operating systems, or due to other platform-level features, such as availability zone support.

The reason this happens is due to how function app and web app plans are mapped to different pools of resources when being created. Different SKUs require a different set of infrastructure capabilities. When you create an app in a resource group, that resource group is mapped and assigned to a specific pool of resources. If you try to create another plan in that resource group and the mapped pool does not have the required resources, this error occurs.

When this error occurs, instead create your function app and hosting plan in a new resource group.

Next steps

- [Deployment technologies in Azure Functions](#)
- [Azure Functions developer guide](#)

Feedback

Was this page helpful?



Quickstart: Create your first C# function in Azure using Visual Studio

Article • 07/18/2024

Azure Functions lets you use Visual Studio to create local C# function projects and then easily publish this project to run in a scalable serverless environment in Azure. If you prefer to develop your C# apps locally using Visual Studio Code, you should instead consider the [Visual Studio Code-based version](#) of this article.

By default, this article shows you how to create C# functions that run on .NET 8 in an [isolated worker process](#). Function apps that run in an isolated worker process are supported on all versions of .NET that are supported by Functions. For more information, see [Supported versions](#).

In this article, you learn how to:

- ✓ Use Visual Studio to create a C# class library project.
- ✓ Create a function that responds to HTTP requests.
- ✓ Run your code locally to verify function behavior.
- ✓ Deploy your code project to Azure Functions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

This video shows you how to create a C# function in Azure.

<https://learn-video.azurefd.net/vod/player?id=efa236ad-db85-4dfc-9f1e-b353c3b09498&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Ffunctions-create-your-first-function-visual-studio>

The steps in the video are also described in the following sections.

Prerequisites

- [Visual Studio 2022](#). Make sure to select the **Azure development** workload during installation.
- [Azure subscription](#). If you don't already have an account, [create a free one](#) before you begin.

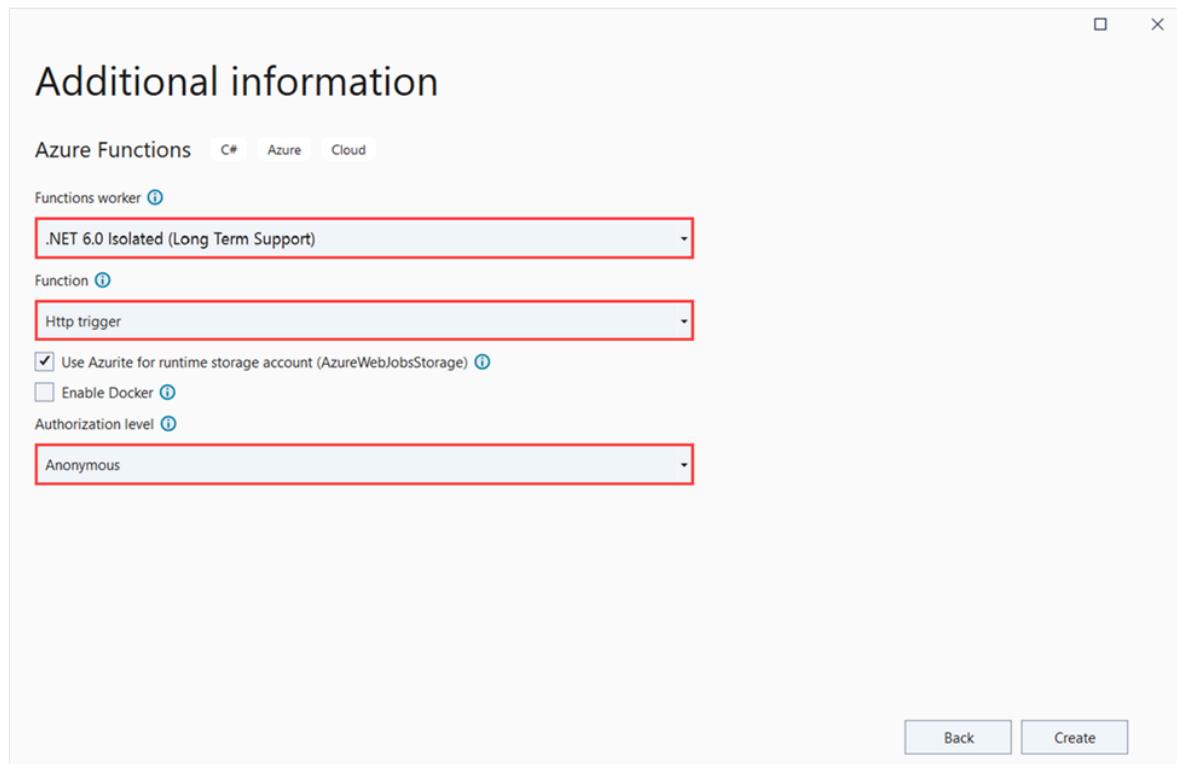
Create a function app project

The Azure Functions project template in Visual Studio creates a C# class library project that you can publish to a function app in Azure. You can use a function app to group functions as a logical unit for easier management, deployment, scaling, and sharing of resources.

1. From the Visual Studio menu, select **File > New > Project**.
2. In **Create a new project**, enter *functions* in the search box, choose the **Azure Functions** template, and then select **Next**.
3. In **Configure your new project**, enter a **Project name** for your project, and then select **Next**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other nonalphanumeric characters.
4. For the remaining **Additional information** settings,

[+] [Expand table](#)

Setting	Value	Description
Functions worker	.NET 8.0 Isolated (Long Term Support)	Your functions run on .NET 8 in an isolated worker process.
Function	HTTP trigger	This value creates a function triggered by an HTTP request.
Use Azurite for runtime storage account (AzureWebJobsStorage)	Enable	Because a function app in Azure requires a storage account, one is assigned or created when you publish your project to Azure. An HTTP trigger doesn't use an Azure Storage account connection string; all other trigger types require a valid Azure Storage account connection string. When you select this option, the Azurite emulator is used.
Authorization level	Anonymous	The created function can be triggered by any client without providing a key. This authorization setting makes it easy to test your new function. For more information, see Authorization level .



Make sure you set the **Authorization level** to **Anonymous**. If you choose the default level of **Function**, you're required to present the [function key](#) in requests to access your function endpoint in Azure.

5. Select **Create** to create the function project and HTTP trigger function.

Visual Studio creates a project and class that contains boilerplate code for the HTTP trigger function type. The boilerplate code sends an HTTP response that includes a value from the request body or query string. The `HttpTrigger` attribute specifies that the function is triggered by an HTTP request.

Rename the function

The `Function` method attribute sets the name of the function, which by default is generated as `Function1`. Since the tooling doesn't let you override the default function name when you create your project, take a minute to create a better name for the function class, file, and metadata.

1. In **File Explorer**, right-click the `Function1.cs` file and rename it to `HttpExample.cs`.
2. In the code, rename the `Function1` class to `HttpExample`.
3. In the method named `Run`, rename the `Function` method attribute to `HttpExample`.

Your function definition should now look like the following code:

C#

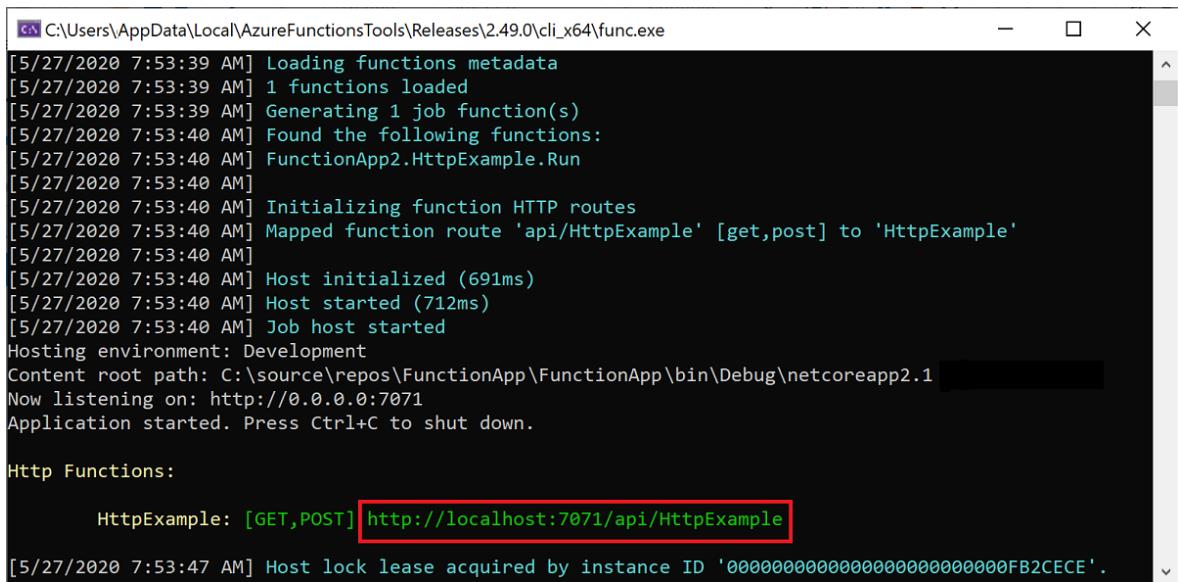
```
[Function("HttpExample")]
public IActionResult Run([HttpTrigger(AuthorizationLevel.AuthLevelValue,
"get", "post")] HttpRequest req)
{
    return new OkObjectResult("Welcome to Azure Functions!");
}
```

Now that you've renamed the function, you can test it on your local computer.

Run the function locally

Visual Studio integrates with Azure Functions Core Tools so that you can test your functions locally using the full Azure Functions runtime.

1. To run your function, press **F5** in Visual Studio. You might need to enable a firewall exception so that the tools can handle HTTP requests. Authorization levels are never enforced when you run a function locally.
2. Copy the URL of your function from the Azure Functions runtime output.



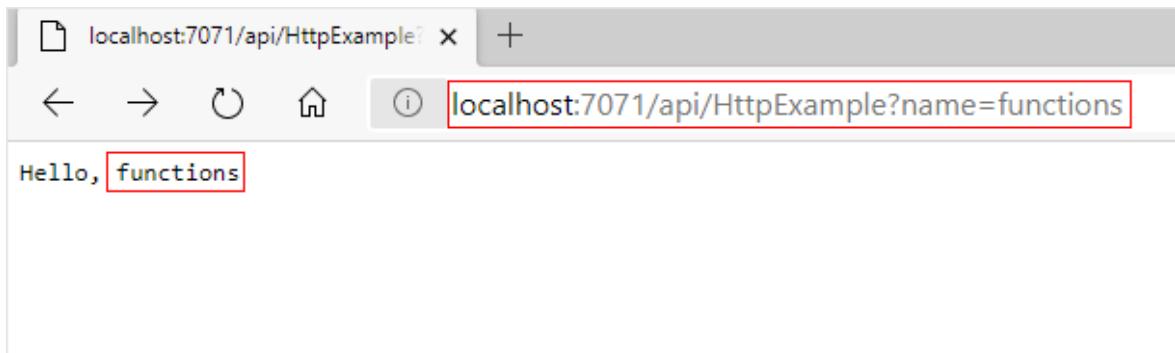
```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.49.0\cli_x64\func.exe
[5/27/2020 7:53:39 AM] Loading functions metadata
[5/27/2020 7:53:39 AM] 1 functions loaded
[5/27/2020 7:53:39 AM] Generating 1 job function(s)
[5/27/2020 7:53:40 AM] Found the following functions:
[5/27/2020 7:53:40 AM] FunctionApp2.HttpExample.Run
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Initializing function HTTP routes
[5/27/2020 7:53:40 AM] Mapped function route 'api/HttpExample' [get,post] to 'HttpExample'
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Host initialized (691ms)
[5/27/2020 7:53:40 AM] Host started (712ms)
[5/27/2020 7:53:40 AM] Job host started
Hosting environment: Development
Content root path: C:\source\repos\FunctionApp\FunctionApp\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[5/27/2020 7:53:47 AM] Host lock lease acquired by instance ID '000000000000000000000000FB2CECE'.
```

3. Paste the URL for the HTTP request into your browser's address bar and run the request. The following image shows the response in the browser to the local GET request returned by the function:



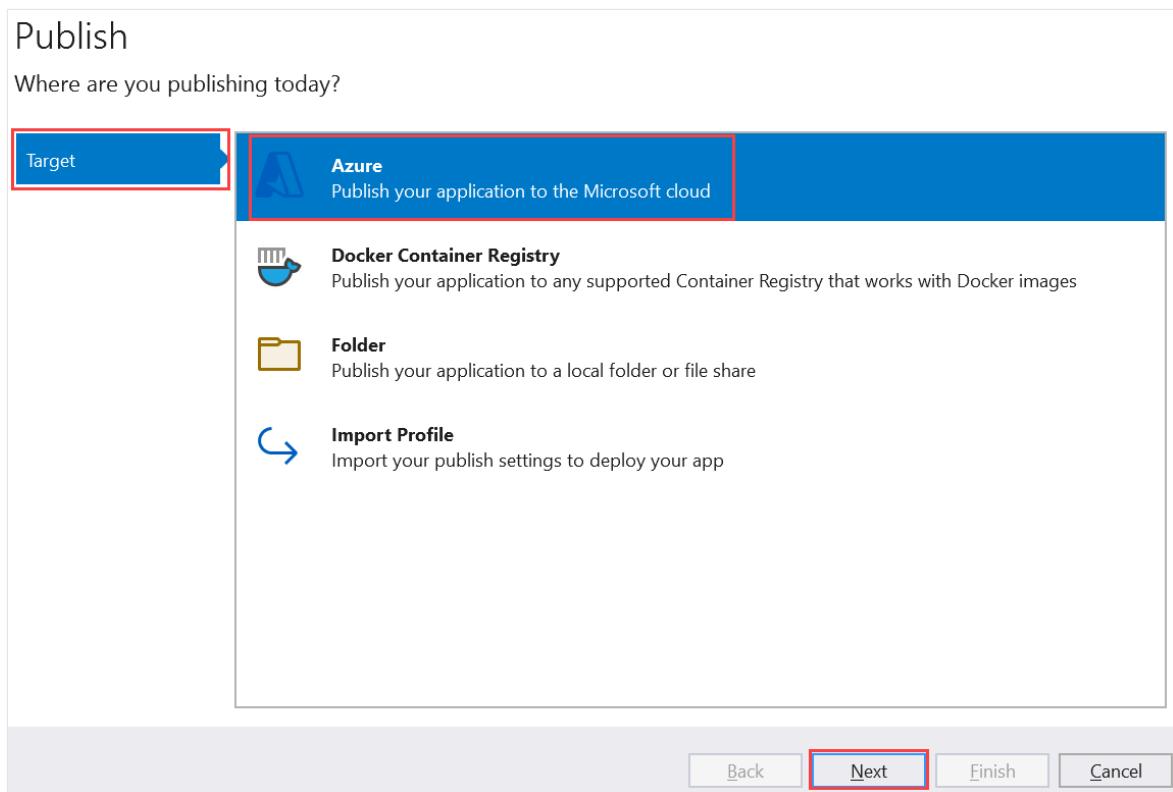
4. To stop debugging, press **Shift** + **F5** in Visual Studio.

After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

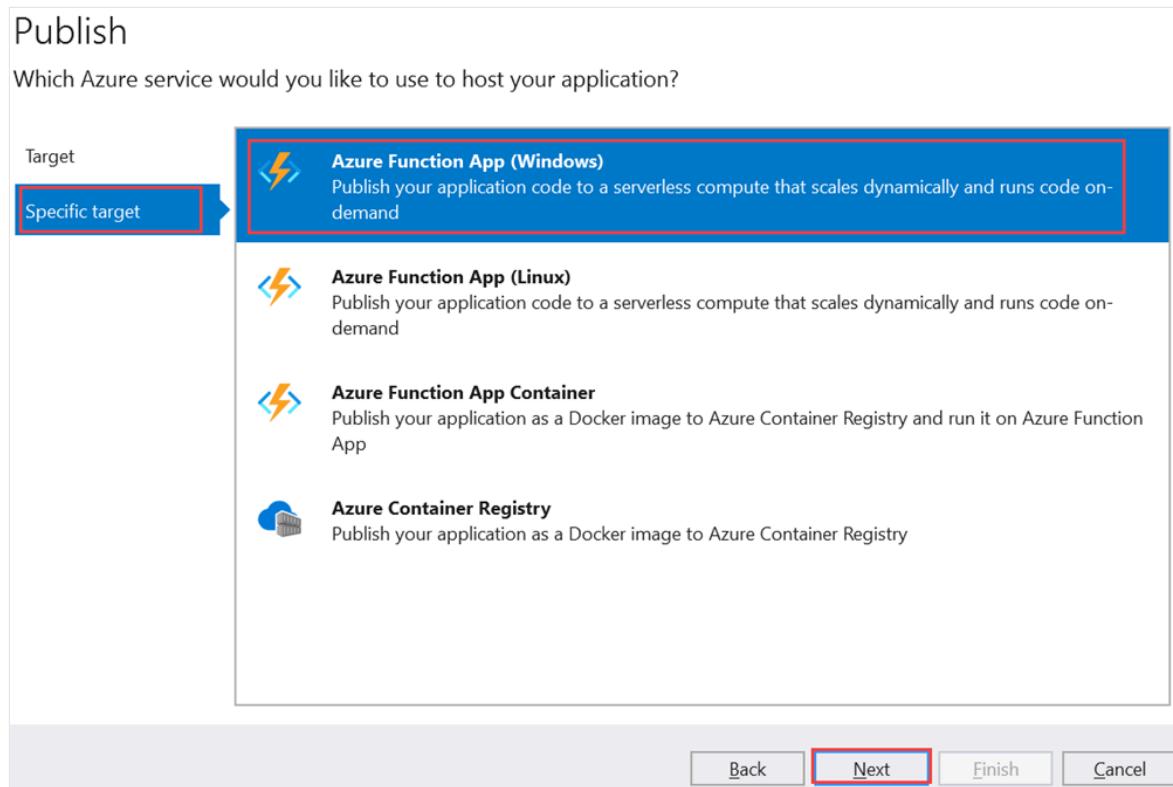
Publish the project to Azure

Visual Studio can publish your local project to Azure. Before you can publish your project, you must have a function app in your Azure subscription. If you don't already have a function app in Azure, Visual Studio publishing creates one for you the first time you publish your project. In this article, you create a function app and related Azure resources.

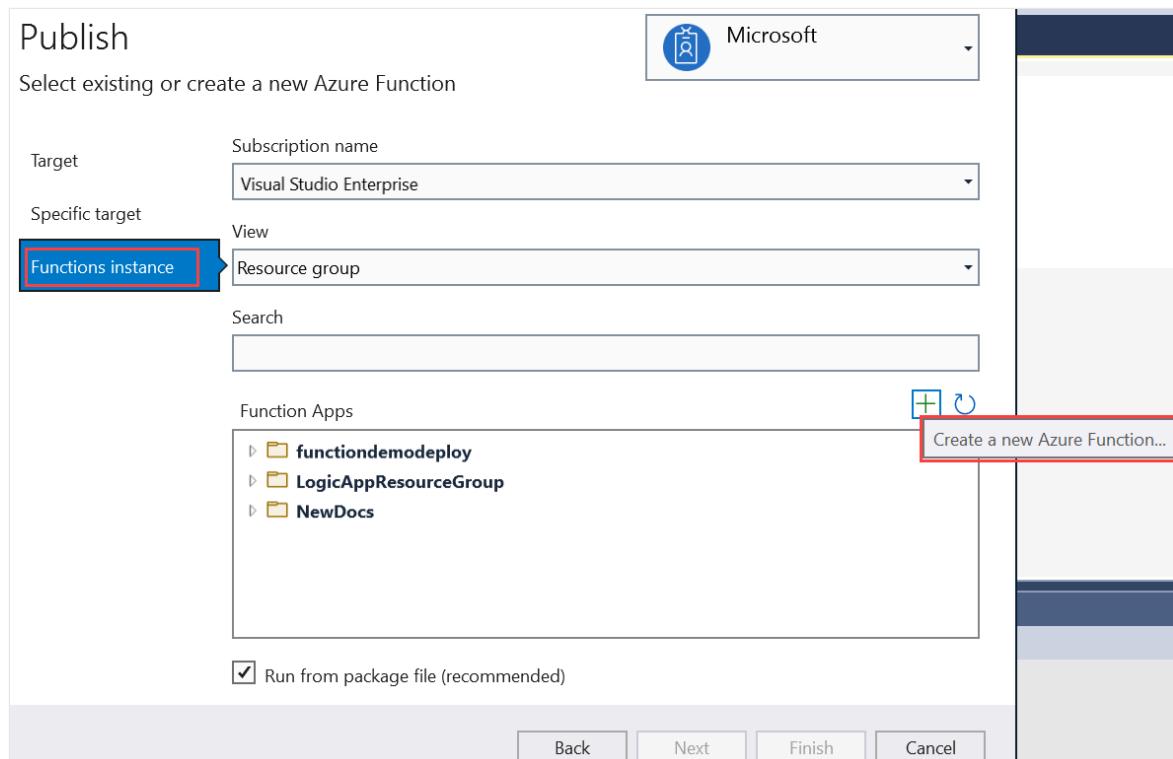
1. In **Solution Explorer**, right-click the project and select **Publish**. In **Target**, select **Azure**, and then select **Next**.



2. On **Specific target**, select **Azure Function App (Windows)**. A function app that runs on Windows is created. Select **Next**.



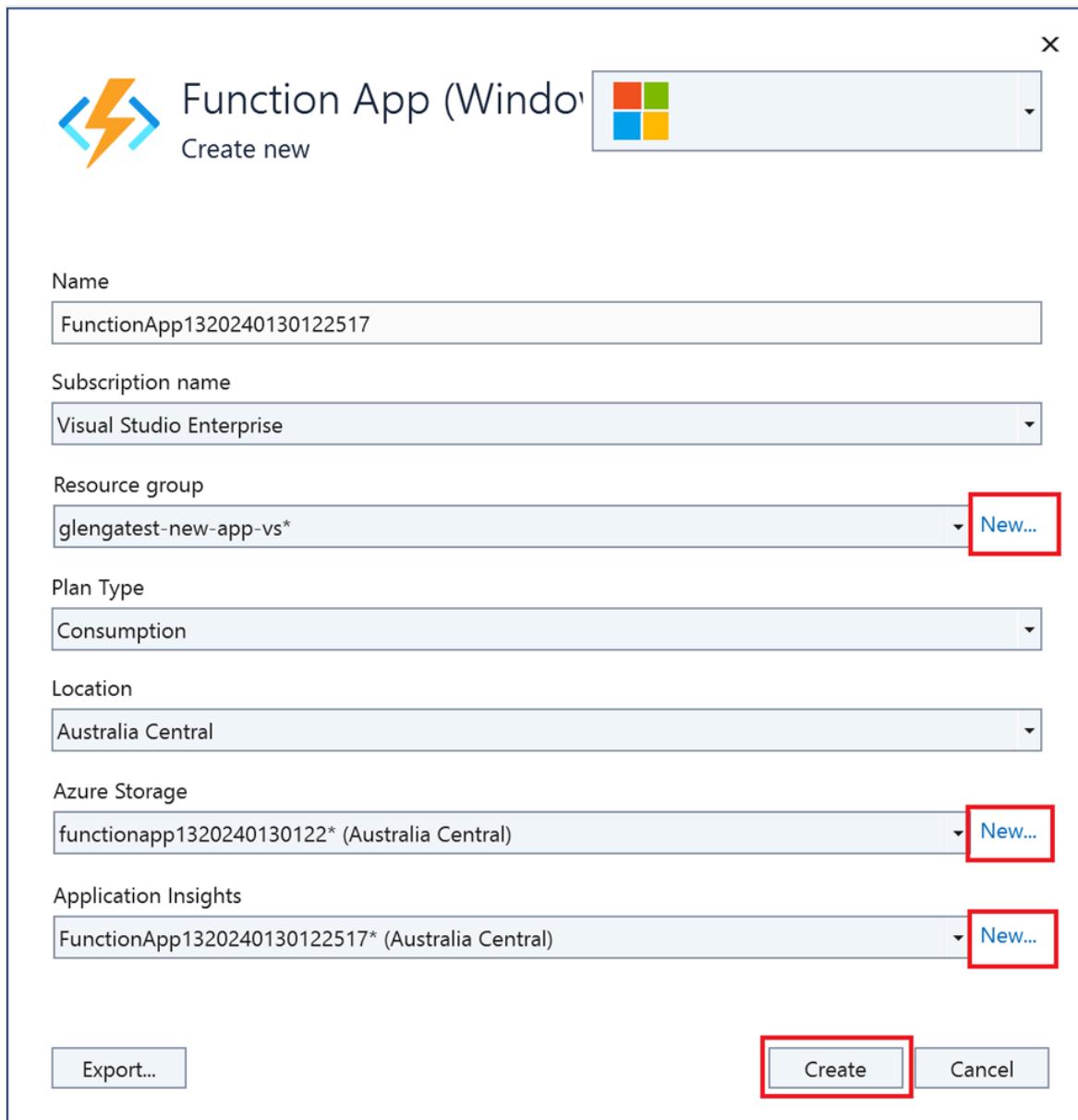
3. On **Functions instance**, select **Create a new Azure Function**.



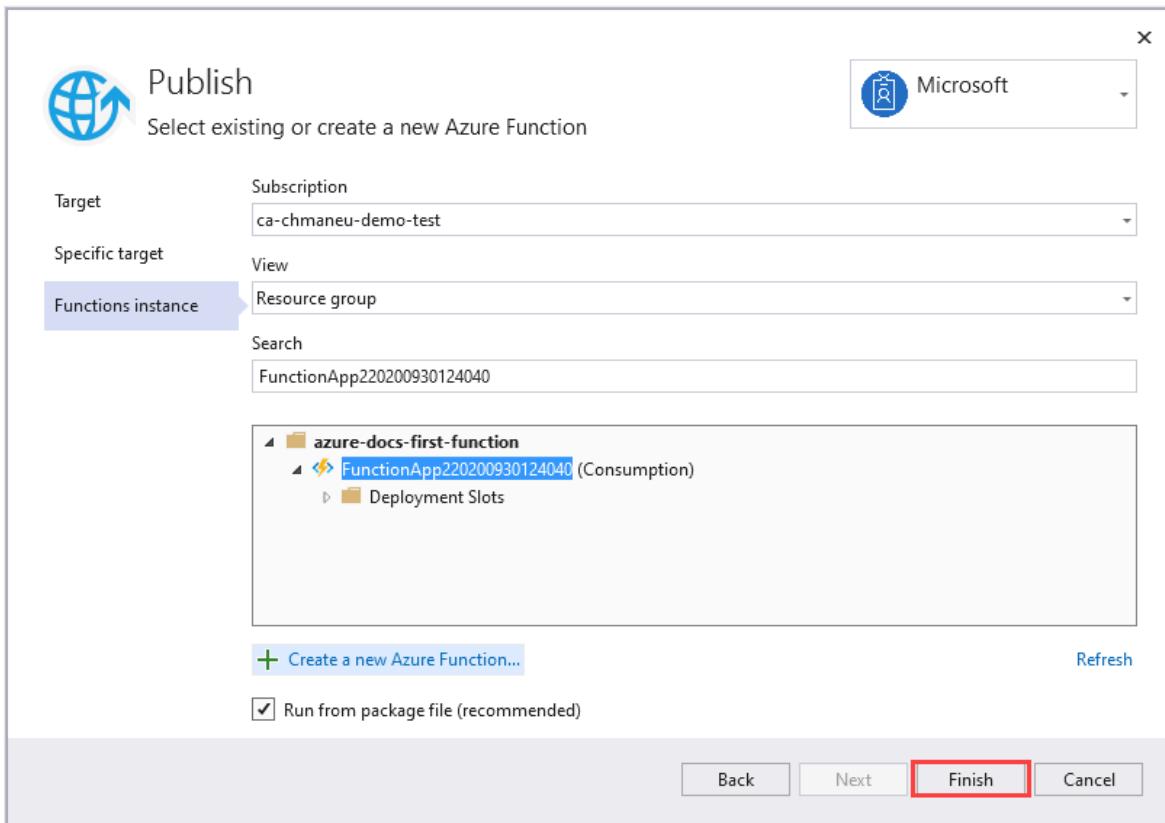
4. Create a new instance by using the values specified in the following table:

[] [Expand table](#)

Setting	Value	Description
Name	Globally unique name	Name that uniquely identifies your new function app. Accept this name or enter a new name. Valid characters are: a-z, 0-9, and -.
Subscription	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the dropdown list.
Resource group	Name of your resource group	The resource group in which you want to create your function app. Select New to create a new resource group. You can also choose to use an existing resource group from the dropdown list.
Plan Type	Consumption	When you publish your project to a function app that runs in a Consumption plan , you pay only for executions of your functions app. Other hosting plans incur higher costs.
Location	Location of the app service	Select a Location in an Azure region near you or other services your functions access.
Azure Storage	General-purpose storage account	An Azure storage account is required by the Functions runtime. Select New to configure a general-purpose storage account. You can also choose to use an existing account that meets the storage account requirements .
Application Insights	Application Insights instance	You should enable Azure Application Insights integration for your function app. Select New to create a new instance, either in a new or in an existing Log Analytics workspace. You can also choose to use an existing instance.



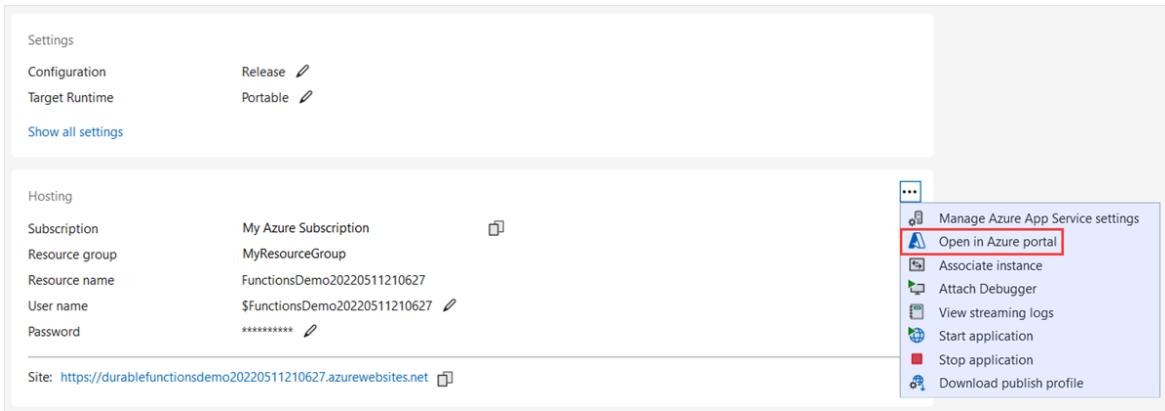
5. Select **Create** to create a function app and its related resources in Azure. The status of resource creation is shown in the lower-left corner of the window.
6. On **Functions instance**, make sure that the **Run from package file** checkbox is selected. Your function app is deployed by using [Zip Deploy](#) with [Run-From-Package](#) mode enabled. Zip Deploy is the recommended deployment method for your functions project for better performance.



7. Select **Finish**, and on the **Publish** pane, select **Publish** to deploy the package that contains your project files to your new function app in Azure.

When deployment is completed, the root URL of the function app in Azure is shown on the **Publish** tab.

8. On the **Publish** tab, in the **Hosting** section, select **Open in Azure portal**. The new function app Azure resource opens in the Azure portal.



Verify your function in Azure

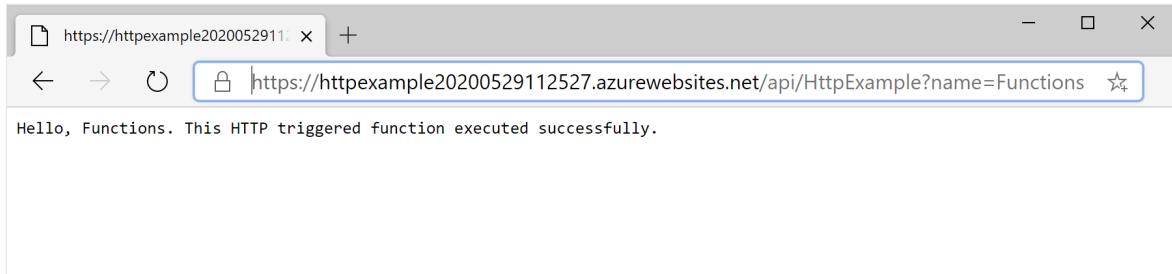
1. In the Azure portal, you should be in the **Overview** page for your new functions app.

2. Under **Functions**, select your new function named **HttpExample**, then in the function page select **Get function URL** and then the **Copy to clipboard** icon.
3. In the address bar in your browser, paste the URL you just copied and run the request.

The URL that calls your HTTP trigger function is in the following format:

```
https://<APP_NAME>.azurewebsites.net/api/HttpExample?name=Functions
```

4. Go to this URL and you see a response in the browser to the remote GET request returned by the function, which looks like the following example:



Clean up resources

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created Azure resources to complete this quickstart. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In the Visual Studio Publish dialogue, in the Hosting section, select [Open in Azure portal](#).
2. In the function app page, select the **Overview** tab and then select the link under **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with options like 'Functions', 'App keys', 'App files', and 'Proxies'. The main content area is titled 'Overview' and shows details for a resource group named 'myResourceGroup'. The details include:

- Status: Running
- Location: Central US
- Subscription: Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags: Click here to add tags
- URL: https://myfunctionapp.azurewebsites.net
- Operating System: Windows
- App Service Plan: ASP-myResourceGroup-a285 (Y1: 0)
- Properties: See More
- Runtime version: 3.0.13139.0

At the bottom, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

3. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

4. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

In this quickstart, you used Visual Studio to create and publish a C# function app in Azure with a simple HTTP trigger function.

To learn more about working with C# functions that run in an isolated worker process, see the [Guide for running C# Azure Functions in an isolated worker process](#). Check out [.NET supported versions](#) to see other versions of supported .NET versions in an isolated worker process.

Advance to the next article to learn how to add an Azure Storage queue binding to your function:

[Add an Azure Storage queue binding to your function](#)

ⓘ Note: The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

Yes

No

Quickstart: Create a C# function in Azure using Visual Studio Code

Article • 07/18/2024

This article creates an HTTP triggered function that runs on .NET 8 in an isolated worker process. For information about .NET versions supported for C# functions, see [Supported versions](#).

There's also a [CLI-based version](#) of this article.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

This video shows you how to create a C# function in Azure using VS Code.

[https://learn-video.azurefd.net/vod/player?id=be75e388-1b74-4051-8a62-132b069a3ec9&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-csharp ↗](https://learn-video.azurefd.net/vod/player?id=be75e388-1b74-4051-8a62-132b069a3ec9&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-csharp)

The steps in the video are also described in the following sections.

Configure your environment

Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [.NET 8.0 SDK ↗](#).
- [Visual Studio Code ↗](#) on one of the [supported platforms ↗](#).
- [C# extension ↗](#) for Visual Studio Code.
- [Azure Functions extension ↗](#) for Visual Studio Code.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in C#. Later in this article, you'll publish your function code to Azure.

1. In Visual Studio Code, press **F1** to open the command palette and search for and run the command **Azure Functions: Create New Project...**.
2. Select the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[+] Expand table

Prompt	Selection
Select a language for your function project	Choose C# .
Select a .NET runtime	Choose .NET 8.0 Isolated (LTS) .
Select a template for your project's first function	Choose HTTP trigger . ¹
Provide a function name	Type HttpExample .
Provide a namespace	Type My.Functions .
Authorization level	Choose Anonymous , which enables anyone to call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Select Open in current window .

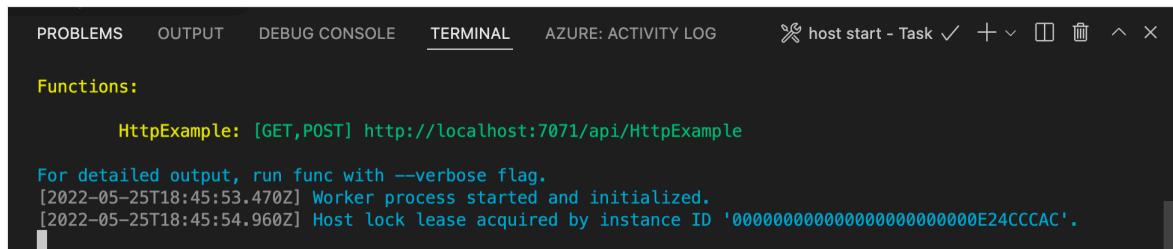
¹ Depending on your VS Code settings, you may need to use the [Change template filter](#) option to see the full list of templates.

4. Visual Studio Code uses the provided information and generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. For more information about the files that are created, see [Generated project files](#).

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure. If you don't already have Core Tools installed locally, you are prompted to install it the first time you run your project.

1. To call your function, press `F5` to start the function app project. The **Terminal** panel displays the output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The terminal window displays the following text:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    AZURE: ACTIVITY LOG
✖ host start - Task ✓ + ▾ □ ⌫ ⌄ ×

Functions:

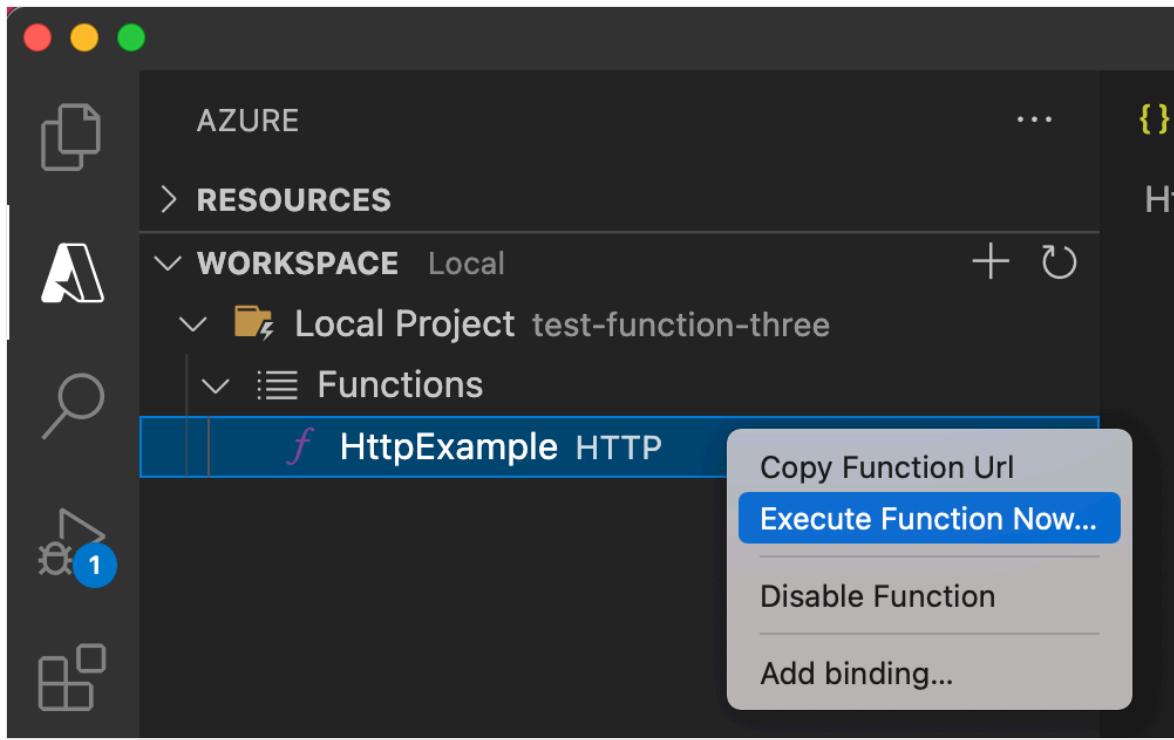
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.
```

If you don't already have Core Tools installed, select **Install** to install Core Tools when prompted to do so.

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Windows) or `ctrl -` click (macOS) the `HttpExample` function and choose **Execute Function Now....**



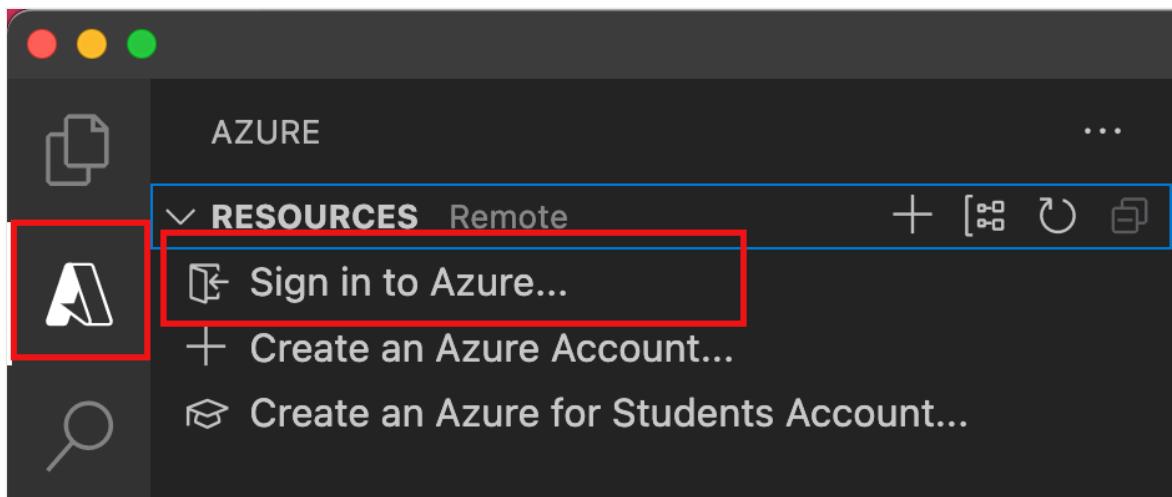
3. In the **Enter request body**, press `Enter` to send a request message to your function.
4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in the **Terminal** panel.
5. Press `ctrl + c` to stop Core Tools and disconnect the debugger.

After checking that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

2. When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors.

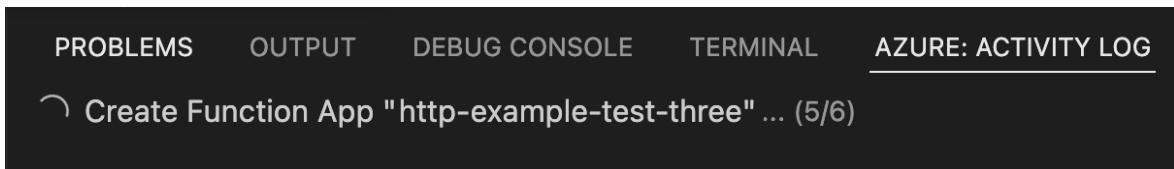
1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
2. At the prompts, provide the following information:

[] [Expand table](#)

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.

Prompt	Action
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region near you .

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

💡 Tip

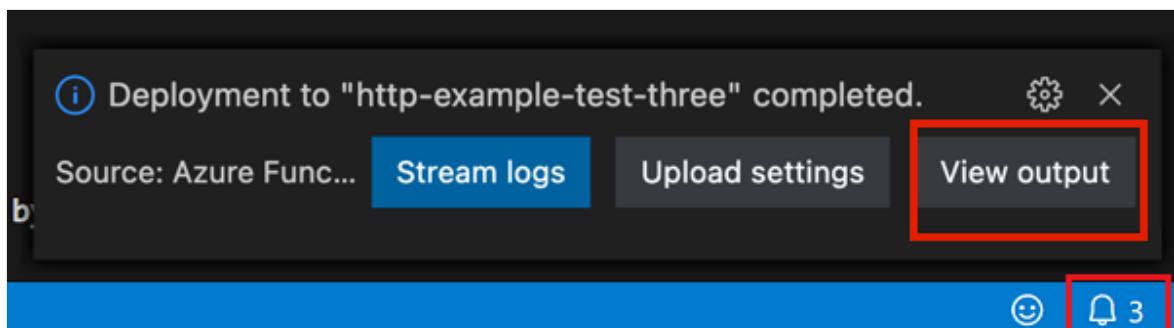
By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

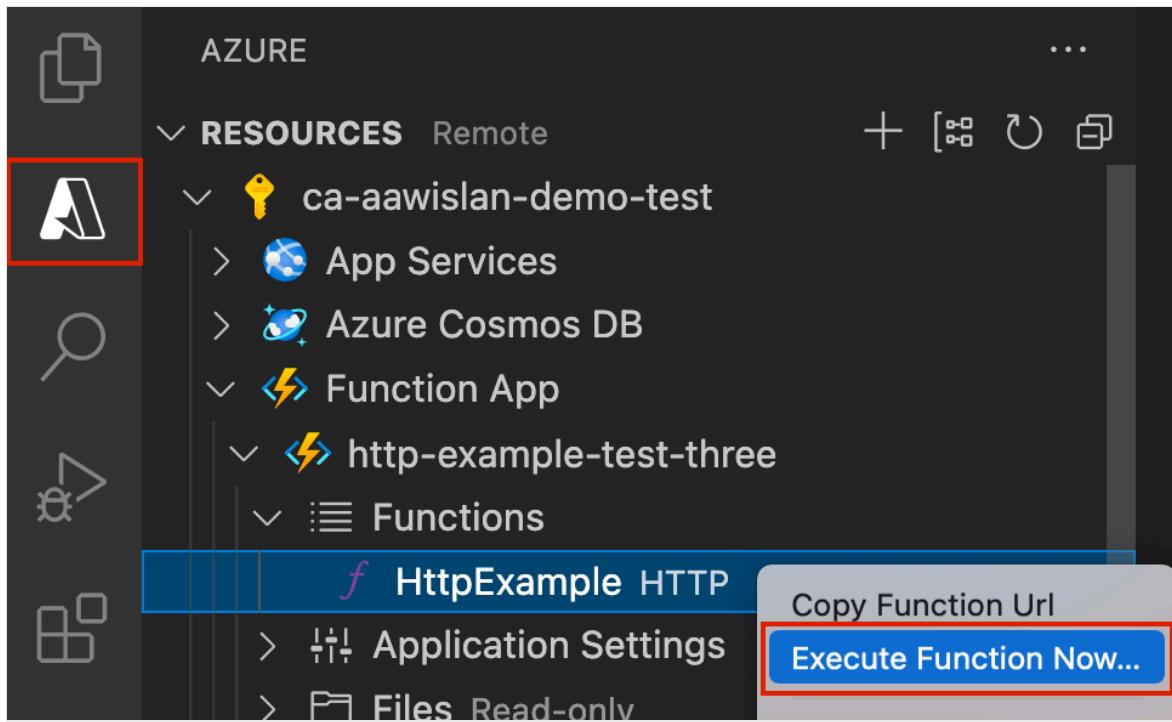
Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Back in the **Resources** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click (Windows) or **Ctrl + click** (macOS) the `HttpExample` function and choose **Execute Function Now....**



2. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
3. When the function executes in Azure and returns a response, a notification is raised in Visual Studio Code.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with options like 'Functions', 'App keys', 'App files', and 'Proxies'. The main content area is titled 'Resource group (change) myResourceGroup' and contains the following information:

Setting	Value
Status	Running
Location	Central US
Subscription	Visual Studio Enterprise
Subscription ID	11111111-1111-1111-1111-111111111111
Tags	Click here to add tags

At the bottom, there are tabs for 'Metrics', 'Features (8)', 'Notifications (0)', and 'Quickstart'. The 'Metrics' tab is currently selected.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You have used [Visual Studio Code](#) to create a function app with a simple HTTP-triggered function. In the next article, you expand that function by connecting to one of the core Azure storage services. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to Azure Cosmos DB](#)

[Connect to Azure Queue Storage](#)

ⓘ Note: The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a C# function in Azure from the command line

Article • 01/25/2024

In this article, you use command-line tools to create a C# function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

This article creates an HTTP triggered function that runs on .NET 8 in an isolated worker process. For information about .NET versions supported for C# functions, see [Supported versions](#). There's also a [Visual Studio Code-based version](#) of this article.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Configure your local environment

Before you begin, you must have the following:

- [.NET 8.0 SDK](#).
- One of the following tools for creating Azure resources:
 - [Azure CLI version 2.4](#) or later.
 - The Azure [Az PowerShell module](#) version 5.9.0 or later.

You also need an Azure account with an active subscription. [Create an account for free](#).

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit ↗](#) (Recommended. [Visual Studio Code debugging](#) requires 64-bit.)
- [v4.x - Windows 32-bit ↗](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

Create a local function project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single function.

1. Run the `func init` command, as follows, to create a functions project in a folder named *LocalFunctionProj* with the specified runtime:

```
Console  
  
func init LocalFunctionProj --worker-runtime dotnet-isolated --target-framework net8.0
```

2. Navigate into the project folder:

```
Console  
  
cd LocalFunctionProj
```

This folder contains various files for the project, including configurations files named [local.settings.json](#) and [host.json](#). Because *local.settings.json* can contain secrets downloaded from Azure, the file is excluded from source control by default in the *.gitignore* file.

3. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (HttpExample) and the `--template` argument specifies the function's trigger (HTTP).

```
Console
```

```
func new --name HttpExample --template "HTTP trigger" --authlevel  
"anonymous"
```

`func new` creates an `HttpExample.cs` code file.

(Optional) Examine the file contents

If desired, you can skip to [Run the function locally](#) and examine the file contents later.

HttpExample.cs

`HttpExample.cs` contains a `Run` method that receives request data in the `req` variable as an `HttpRequest` object. That parameter is decorated with the `HttpTriggerAttribute`, to define the trigger behavior.

C#

```
using System.Net;  
using Microsoft.Azure.Functions.Worker;  
using Microsoft.Extensions.Logging;  
using Microsoft.AspNetCore.Http;  
using Microsoft.AspNetCore.Mvc;  
  
namespace Company.Function  
{  
    public class HttpExample  
    {  
        private readonly ILogger<HttpExample> _logger;  
  
        public HttpExample(ILogger<HttpExample> logger)  
        {  
            _logger = logger;  
        }  
  
        [Function("HttpExample")]  
        public IActionResult  
Run([HttpTrigger(AuthorizationLevel.AuthLevelValue, "get", "post")]  
HttpRequest req)  
        {  
            _logger.LogInformation("C# HTTP trigger function processed a  
request.");  
  
            return new OkObjectResult("Welcome to Azure Functions!");  
        }  
    }  
}
```

The return object is an `IActionResult` object that contains the data that's handed back to the HTTP response.

To learn more, see [Azure Functions HTTP triggers and bindings](#).

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the `LocalFunctionProj` folder:

```
func start
```

Toward the end of the output, the following lines should appear:

```
...
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
...
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, navigate to the project's root folder, and run the previous command again.

2. Copy the URL of your `HttpExample` function from this output to a browser and browse to the function URL and you should receive a *Welcome to Azure Functions* message.
3. When you're done, use `Ctrl+C` and choose `y` to stop the functions host.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the following commands to create these items. Both Azure CLI and PowerShell are supported.

1. If you haven't done so already, sign in to Azure:

```
Azure CLI
az login
```

The `az login` command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in your chosen region:

```
Azure CLI
az group create --name AzureFunctionsQuickstart-rg --location
<REGION>
```

The `az group create` command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the [az account list-locations](#) command.

3. Create a general-purpose storage account in your resource group and region:

Azure CLI

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location <REGION>
--resource-group AzureFunctionsQuickstart-rg --sku Standard_LRS --
allow-blob-public-access false
```

The `az storage account create` command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#).

 **Important**

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

4. Create the function app in Azure:

Azure CLI

Azure CLI

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg
--consumption-plan-location <REGION> --runtime dotnet-isolated --
functions-version 4 --name <APP_NAME> --storage-account
<STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure.

In the previous example, replace `<STORAGE_NAME>` with the name of the account you used in the previous step, and replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default DNS domain for the function app.

This command creates a function app running in your specified language runtime under the [Azure Functions Consumption Plan](#), which is free for the amount of

usage you incur here. The command also creates an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

Deploy the function project to Azure

After you've successfully created your function app in Azure, you're now ready to deploy your local functions project by using the `func azure functionapp publish` command.

In the following example, replace `<APP_NAME>` with the name of your app.

Console

```
func azure functionapp publish <APP_NAME>
```

The publish command shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
  HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-
qs.azurewebsites.net/api/httpexample
```

Invoke the function on Azure

Because your function uses an HTTP trigger and supports GET requests, you invoke it by making an HTTP request to its URL. It's easiest to do this in a browser.

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar. When you navigate to this URL, the browser should display similar output as when you ran the function locally.

Run the following command to view near real-time streaming logs:

Console

```
func azure functionapp logstream <APP_NAME>
```

In a separate terminal window or in the browser, call the remote function again. A verbose log of the function execution in Azure is shown in the terminal.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

[Connect to Azure Cosmos DB](#)

[Connect to Azure Queue Storage](#)

Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI

Article • 09/09/2024

In this Quickstart, you use Azure Developer command-line tools to create functions that respond to HTTP requests. After testing the code locally, you deploy it to a new serverless function app you create running in a Flex Consumption plan in Azure Functions.

The project source uses the Azure Developer CLI (azd) to simplify deploying your code to Azure. This deployment follows current best practices for secure and scalable Azure Functions deployments.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

By default, the Flex Consumption plan follows a *pay-for-what-you-use* billing model, which means to complete this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Azure Developer CLI](#).
- [Azure Functions Core Tools](#).
- [.NET 8.0 SDK ↗](#)
- [Azurite storage emulator](#)
- A [secure HTTP test tool](#) for sending requests with JSON payloads to your function endpoints. This article uses `curl`.

Initialize the project

You can use the `azd init` command to create a local Azure Functions code project from a template.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

```
Console
```

```
azd init --template functions-quickstart-dotnet-azd -e flexquickstart-dotnet
```

This command pulls the project files from the [template repository](#) and initializes the project in the current folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Run this command to navigate to the `http` app folder:

```
Console
```

```
cd http
```

3. Create a file named `local.settings.json` in the `http` folder that contains this JSON data:

```
JSON
```

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"  
    }  
}
```

This file is required when running locally.

Run in your local environment

1. Run this command from your app folder in a terminal or command prompt:

```
Console
```

```
func start
```

When the Functions host starts in your local project folder, it writes the URL endpoints of your HTTP triggered functions to the terminal output.

2. In your browser, navigate to the `httpget` endpoint, which should look like this URL:

<http://localhost:7071/api/httpget>

3. From a new terminal or command prompt window, run this `curl` command to send a POST request with a JSON payload to the `httppost` endpoint:

```
Console
```

```
curl -i http://localhost:7071/api/httppost -H "Content-Type: text/json"  
-d @testdata.json
```

This command reads JSON payload data from the `testdata.json` project file. You can find examples of both HTTP requests in the `test.http` project file.

4. When you're done, press **Ctrl+C** in the terminal window to stop the `func.exe` host process.

Review the code (optional)

You can review the code that defines the two HTTP trigger function endpoints:

```
httpget
```

```
C#
```

```
[Function("httpget")]
    public IActionResult
Run([HttpTrigger(AuthorizationLevel.Function, "get")]
    HttpRequest req,
    string name)
{
    var returnValue = string.IsNullOrEmpty(name)
        ? "Hello, World."
        : $"Hello, {name}.";

    _logger.LogInformation($"C# HTTP trigger function processed
a request for {returnValue}.");
}
```

```
        return new OkObjectResult(returnValue);
    }
```

You can review the complete template project [here](#).

After you verify your functions locally, it's time to publish them to Azure.

Deploy to Azure

This project is configured to use the `azd up` command to deploy this project to a new function app in a Flex Consumption plan in Azure.

💡 Tip

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

1. Run this command to have `azd` create the required Azure resources in Azure and deploy your code project to the new function app:

```
Console
```

```
azd up
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

[\[+\] Expand table](#)

Parameter	Description
<code>Azure subscription</code>	Subscription in which your resources are created.
<code>Azure location</code>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd up` command uses your response to these prompts with the Bicep configuration files to complete these deployment tasks:

- Create and configure these required Azure resources (equivalent to `azd provision`):
 - Flex Consumption plan and function app
 - Azure Storage (required) and Application Insights (recommended)
 - Access policies and roles for your account
 - Service-to-service connections using managed identities (instead of stored connection strings)
 - Virtual network to securely run both the function app and the other Azure resources
- Package and deploy your code to the deployment container (equivalent to `azd deploy`). The app is then started and runs in the deployed package.

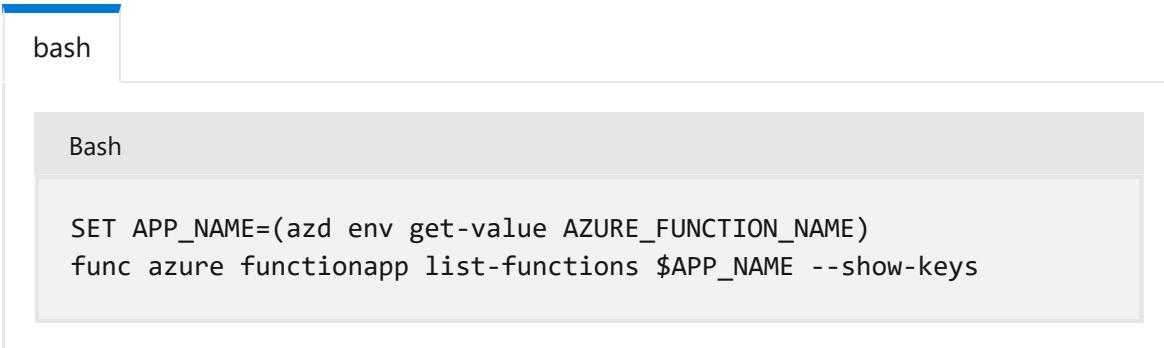
After the command completes successfully, you see links to the resources you created.

Invoke the function on Azure

You can now invoke your function endpoints in Azure by making HTTP requests to their URLs using your HTTP test tool or from the browser (for GET requests). When your functions run in Azure, access key authorization is enforced, and you must provide a function access key with your request.

You can use the Core Tools to obtain the URL endpoints of your functions running in Azure.

1. In your local terminal or command prompt, run these commands to get the URL endpoint values:



```
bash
Bash
SET APP_NAME=(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp list-functions $APP_NAME --show-keys
```

The `azd env get-value` command gets your function app name from the local environment. Using the `--show-keys` option with `func azure functionapp list-`

`functions` means that the returned **Invoke URL**: value for each endpoint includes a function-level access key.

2. As before, use your HTTP test tool to validate these URLs in your function app running in Azure.

Redeploy your code

You can run the `azd up` command as many times as you need to both provision your Azure resources and deploy code updates to your function app.

Note

Deployed code files are always overwritten by the latest deployment package.

Your initial responses to `azd` prompts and any environment variables generated by `azd` are stored locally in your named environment. Use the `azd env get-values` command to review all of the variables in your environment that were used when creating Azure resources.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs:

Console

```
azd down --no-prompt
```

Note

The `--no-prompt` option instructs `azd` to delete your resource group without a confirmation from you.

This command doesn't affect your local code project.

Related content

- [Flex Consumption plan](#)
 - [Azure Developer CLI \(azd\)](#)
 - [azd reference](#)
 - [Azure Functions Core Tools reference](#)
 - [Code and test Azure Functions locally](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Create your first function with Java and Eclipse

Article • 03/17/2023

This article shows you how to create a [serverless](#) function project with the Eclipse IDE and Apache Maven, test and debug it, then deploy it to Azure Functions.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Set up your development environment

To develop a functions app with Java and Eclipse, you must have the following installed:

- [Java Developer Kit](#), version 8.
- [Apache Maven](#), version 3.0 or above.
- [Eclipse](#), with Java and Maven support.
- [Azure CLI](#)

Important

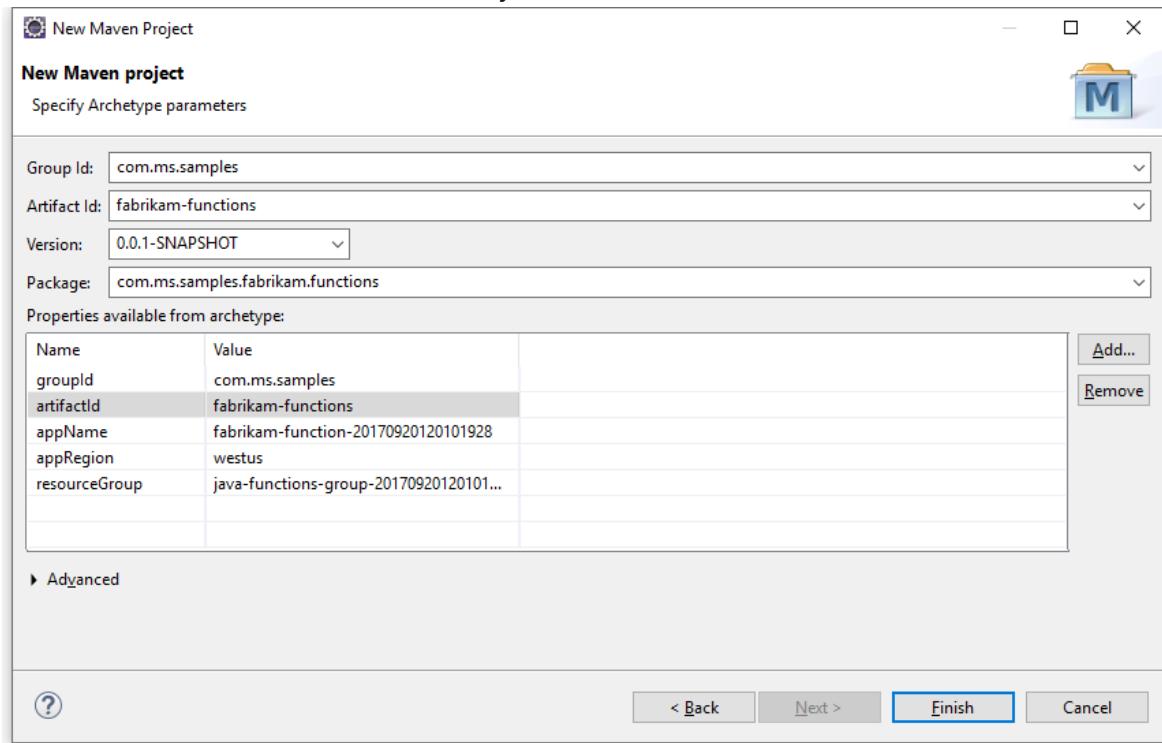
The JAVA_HOME environment variable must be set to the install location of the JDK to complete this quickstart.

It's highly recommended to also install [Azure Functions Core Tools, version 2](#), which provide a local environment for running and debugging Azure Functions.

Create a Functions project

1. In Eclipse, select the **File** menu, then select **New -> Maven Project**.
2. Accept the defaults in the **New Maven Project** dialogue and select **Next**.
3. Find and select the [azure-functions-archetype](#) and click **Next**.
4. Be sure to fill in values for all of the fields including `resourceGroup`, `appName`, and `appRegion` (please use a different `appName` other than `fabrikam-function-`)

20170920120101928), and eventually Finish.



Maven creates the project files in a new folder with a name of *artifactId*. The generated code in the project is a simple [HTTP triggered](#) function that echoes the body of the triggering HTTP request.

Run functions locally in the IDE

ⓘ Note

Azure Functions Core Tools, version 2 must be installed to run and debug functions locally.

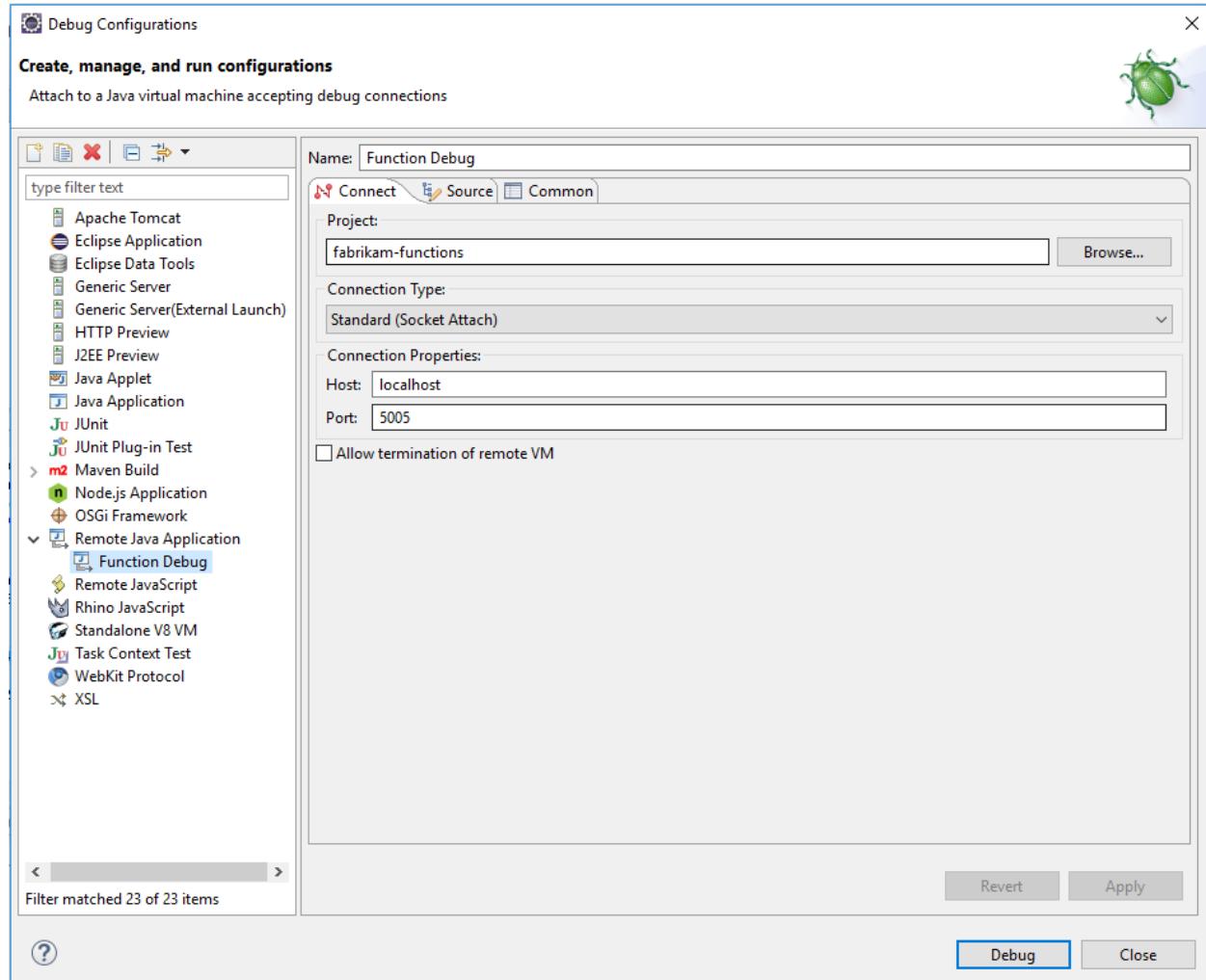
1. Right-click on the generated project, then choose **Run As** and **Maven build**.
2. In the **Edit Configuration** dialog, Enter `package` in the **Goals**, then select **Run**. This will build and package the function code.
3. Once the build is complete, create another Run configuration as above, using `azure-functions:run` as the goal and name. Select **Run** to run the function in the IDE.

Terminate the runtime in the console window when you're done testing your function. Only one function host can be active and running locally at a time.

Debug the function in Eclipse

In your **Run As** configuration set up in the previous step, change `azure-functions:run` to `azure-functions:run -DenableDebug` and run the updated configuration to start the function app in debug mode.

Select the **Run** menu and open **Debug Configurations**. Choose **Remote Java Application** and create a new one. Give your configuration a name and fill in the settings. The port should be consistent with the debug port opened by function host, which by default is `5005`. After setup, click on **Debug** to start debugging.



Set breakpoints and inspect objects in your function using the IDE. When finished, stop the debugger and the running function host. Only one function host can be active and running locally at a time.

Deploy the function to Azure

The deploy process to Azure Functions uses account credentials from the Azure CLI. [Log in with the Azure CLI](#) before continuing using your computer's command prompt.

```
Azure CLI
az login
```

Deploy your code into a new Function app using the `azure-functions:deploy` Maven goal in a new **Run As** configuration.

When the deploy is complete, you see the URL you can use to access your Azure function app:

Output

```
[INFO] Successfully deployed Function App with package.  
[INFO] Deleting deployment package from Azure Storage...  
[INFO] Successfully deleted deployment package fabrikam-function-  
20170920120101928.20170920143621915.zip  
[INFO] Successfully deployed Function App at https://fabrikam-function-  
20170920120101928.azurewebsites.net  
[INFO] -----  
---
```

Next steps

- Review the [Java Functions developer guide](#) for more information on developing Java functions.
- Add additional functions with different triggers to your project using the `azure-functions:add` Maven target.

Use Java and Gradle to create and publish a function to Azure

Article • 07/18/2024

This article shows you how to build and publish a Java function project to Azure Functions with the Gradle command-line tool. When you're done, your function code runs in Azure in a [serverless hosting plan](#) and is triggered by an HTTP request.

ⓘ Note

If Gradle is not your preferred development tool, check out our similar tutorials for Java developers using [Maven](#), [IntelliJ IDEA](#) and [VS Code](#).

Prerequisites

To develop functions using Java, you must have the following installed:

- [Java Developer Kit](#), version 8, 11, 17 or 21. (Java 21 is currently supported in preview on Linux only)
- [Azure CLI](#)
- [Azure Functions Core Tools](#) version 2.6.666 or above
- [Gradle](#), version 6.8 and above

You also need an active Azure subscription. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

ⓘ Important

The JAVA_HOME environment variable must be set to the install location of the JDK to complete this quickstart.

Prepare a Functions project

Use the following command to clone the sample project:

Bash

```
git clone https://github.com/Azure-Samples/azure-functions-samples-java.git
```

```
cd azure-functions-samples-java/
```

Open `build.gradle` and change the `appName` in the following section to a unique name to avoid domain name conflict when deploying to Azure.

Gradle

```
azurefunctions {  
    resourceGroup = 'java-functions-group'  
    appName = 'azure-functions-sample-demo'  
    pricingTier = 'Consumption'  
    region = 'westus'  
    runtime {  
        os = 'windows'  
    }  
    localDebug = "transport=dt_socket,server=y,suspend=n,address=5005"  
}
```

Open the new `Function.java` file from the `src/main/java` path in a text editor and review the generated code. This code is an [HTTP triggered function](#) that echoes the body of the request.

I ran into an issue

Run the function locally

Run the following command to build then run the function project:

Bash

```
gradle jar --info  
gradle azureFunctionsRun
```

You see output like the following from Azure Functions Core Tools when you run the project locally:

```
...
```

```
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.
```

```
Http Functions:
```

```
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

```
...
```

Trigger the function from the command line using the following cURL command in a new terminal window:

Bash

```
curl -w "\n" http://localhost:7071/api/HttpExample --data AzureFunctions
```

The expected output is the following:

```
Hello, AzureFunctions
```

⚠ Note

If you set authLevel to `FUNCTION` or `ADMIN`, the [access key](#) isn't required when running locally.

Use `Ctrl+C` in the terminal to stop the function code.

I ran into an issue

Deploy the function to Azure

A function app and related resources are created in Azure when you first deploy your function app. Before you can deploy, use the [az login](#) Azure CLI command to sign in to your Azure subscription.

Azure CLI

```
az login
```

💡 Tip

If your account can access multiple subscriptions, use [az account set](#) to set the default subscription for this session.

Use the following command to deploy your project to a new function app.

Bash

```
gradle azureFunctionsDeploy
```

This creates the following resources in Azure, based on the values in the build.gradle file:

- Resource group. Named with the *resourceGroup* you supplied.
- Storage account. Required by Functions. The name is generated randomly based on Storage account name requirements.
- App Service plan. Serverless Consumption plan hosting for your function app in the specified *region*. The name is generated randomly.
- Function app. A function app is the deployment and execution unit for your functions. The name is your *appName*, appended with a randomly generated number.

The deployment also packages the project files and deploys them to the new function app using [zip deployment](#), with run-from-package mode enabled.

The authLevel for HTTP Trigger in sample project is `ANONYMOUS`, which will skip the authentication. However, if you use other authLevel like `FUNCTION` or `ADMIN`, you need to get the function key to call the function endpoint over HTTP. The easiest way to get the function key is from the [Azure portal](#).

[I ran into an issue](#)

Get the HTTP trigger URL

You can get the URL required to trigger your function, with the function key, from the Azure portal.

1. Browse to the [Azure portal](#), sign in, type the *appName* of your function app into **Search** at the top of the page, and press enter.
2. In your function app, select **Functions**, choose your function, then click **Get Function Url** at the top right.

The screenshot shows the Azure Functions portal interface. At the top, there's a search bar with placeholder text 'Search (Ctrl+ /)' and several action buttons: 'Enable', 'Disable', 'Delete', 'Get Function Url' (which is highlighted with a red box), and 'Refresh'. Below this, there's a navigation menu with 'Overview' selected. On the left, there are three tabs: 'Developer', 'Code + Test', and 'Integration'. The main content area displays the following information:

- Function app:** azure-functions-sample-demo
- Status:** Enabled
- Resource group (change):** java-functions-group

3. Choose **default (Function key)** and select **Copy**.

You can now use the copied URL to access your function.

Verify the function in Azure

To verify the function app running on Azure using `cURL`, replace the URL from the sample below with the URL that you copied from the portal.

Console

```
curl -w "\n" http://azure-functions-sample-
demo.azurewebsites.net/api/HttpExample --data AzureFunctions
```

This sends a POST request to the function endpoint with `AzureFunctions` in the body of the request. You see the following response.

Hello, AzureFunctions

I ran into an issue

Next steps

You've created a Java functions project with an HTTP triggered function, run it on your local machine, and deployed it to Azure. Now, extend your function by...

Adding an Azure Storage queue output binding

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create your first Java function in Azure using IntelliJ

Article • 05/29/2024

This article shows you how to use Java and IntelliJ to create an Azure function.

Specifically, this article shows you:

- How to create an HTTP-triggered Java function in an IntelliJ IDEA project.
- Steps for testing and debugging the project in the integrated development environment (IDE) on your own computer.
- Instructions for deploying the function project to Azure Functions.

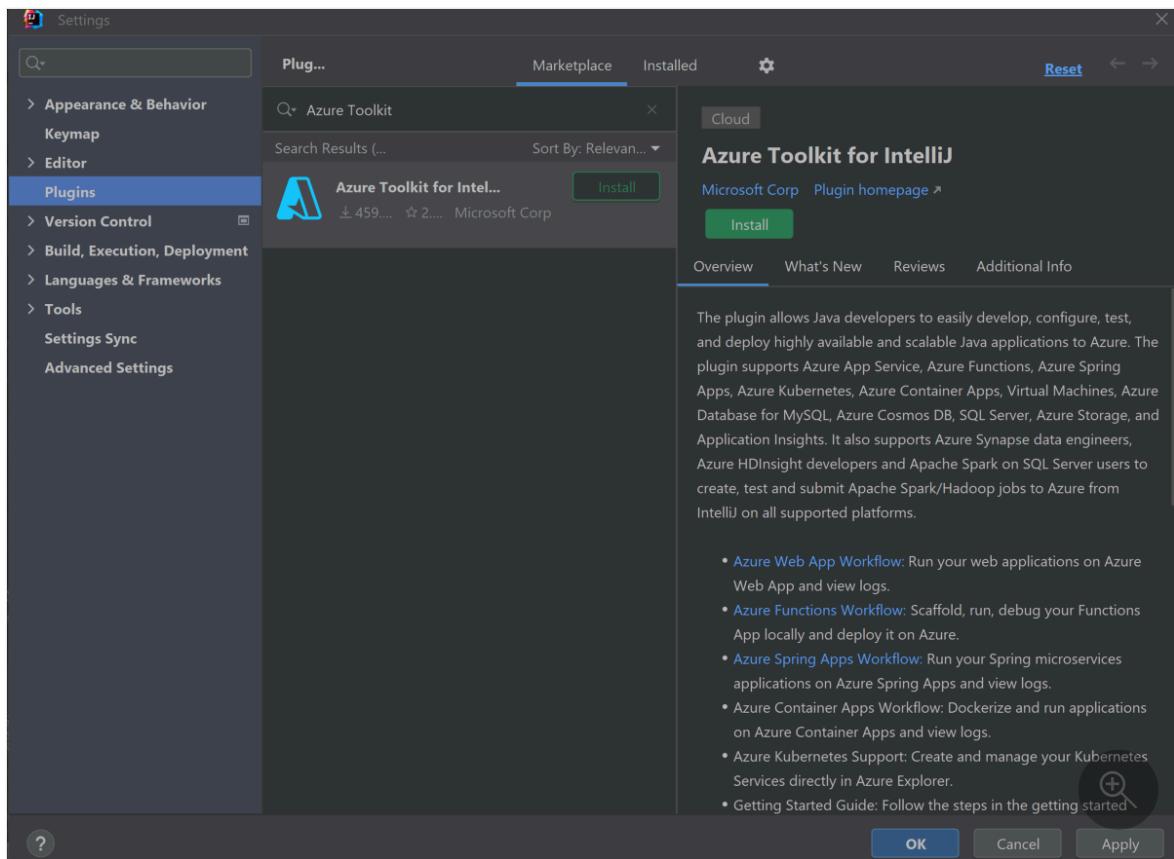
Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- An [Azure supported Java Development Kit \(JDK\)](#), version 8, 11, 17 or 21. (Java 21 is currently only supported in preview on Linux only)
- An [IntelliJ IDEA ↗](#) Ultimate Edition or Community Edition installed
- [Maven 3.5.0+ ↗](#)
- Latest [Function Core Tools ↗](#)

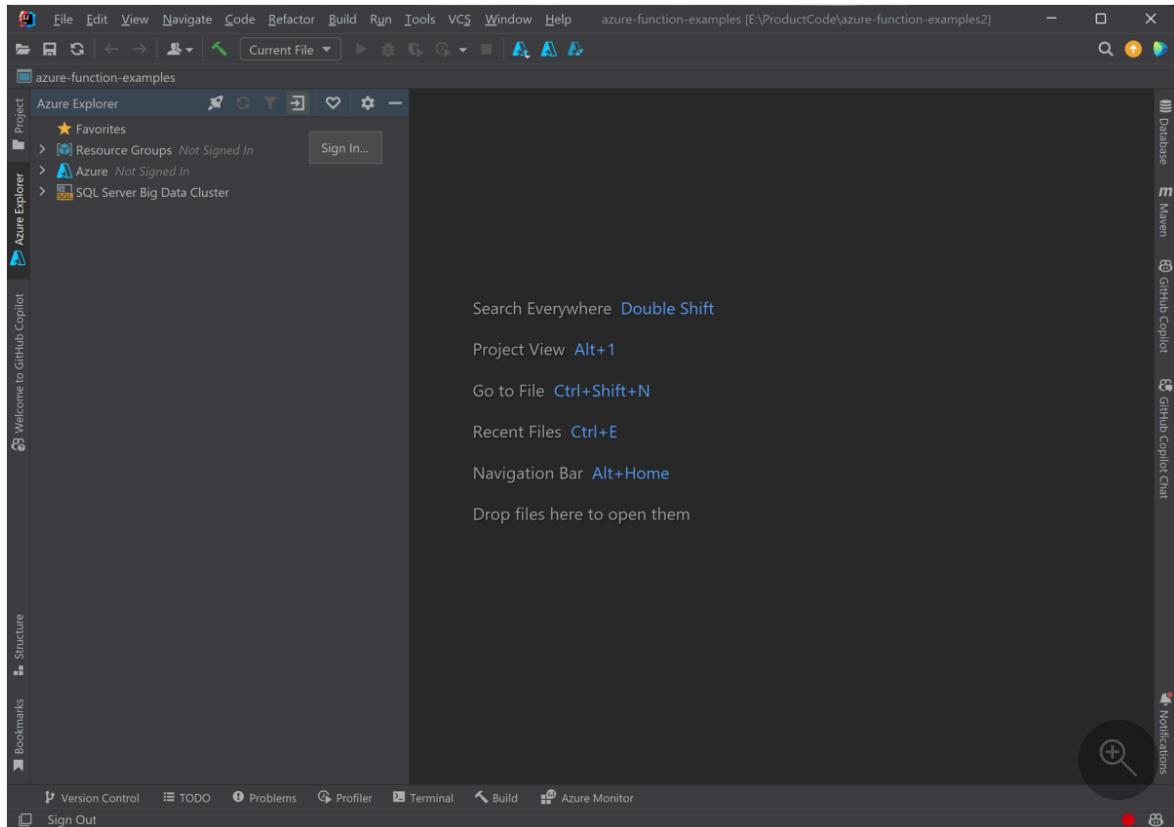
Install plugin and sign in

To install the Azure Toolkit for IntelliJ and then sign in, follow these steps:

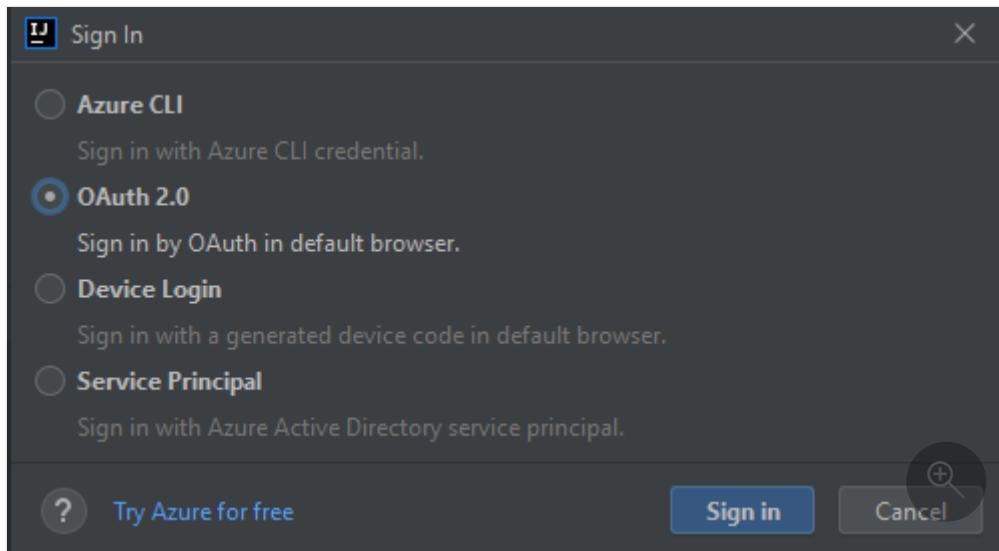
1. In IntelliJ IDEA's **Settings/Preferences** dialog (Ctrl+Alt+S), select **Plugins**. Then, find the **Azure Toolkit for IntelliJ** in the **Marketplace** and select **Install**. After it's installed, select **Restart** to activate the plugin.



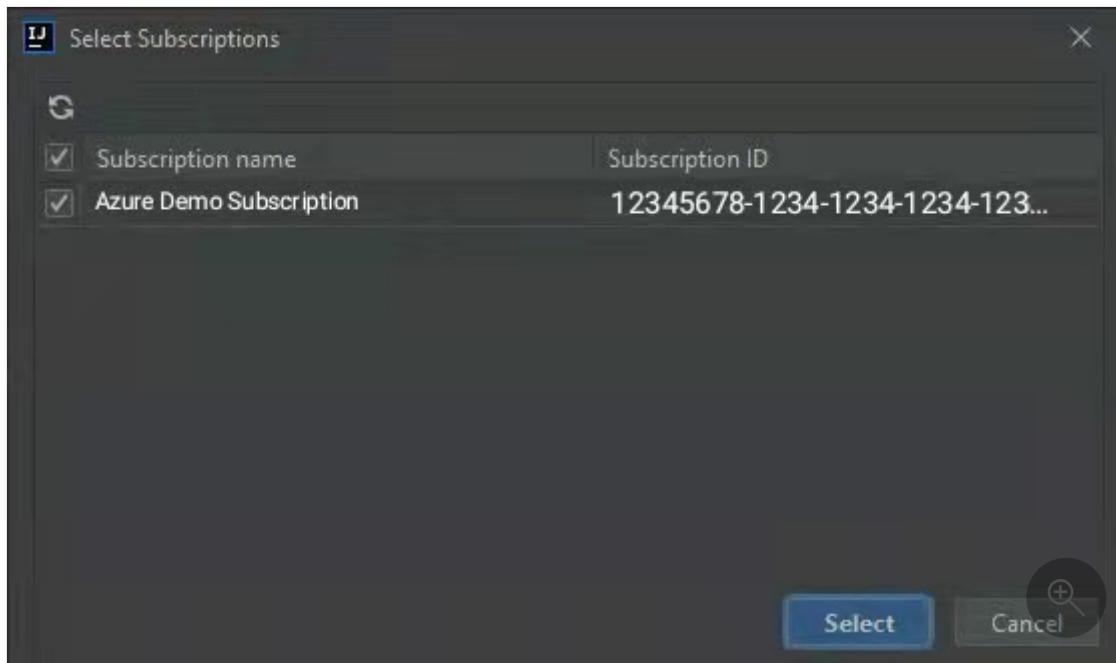
2. To sign in to your Azure account, open the **Azure Explorer** sidebar, and then select the **Azure Sign In** icon in the bar on top (or from the IDEA menu, select **Tools > Azure > Azure Sign in**).



3. In the **Azure Sign In** window, select **OAuth 2.0**, and then select **Sign in**. For other sign-in options, see [Sign-in instructions for the Azure Toolkit for IntelliJ](#).



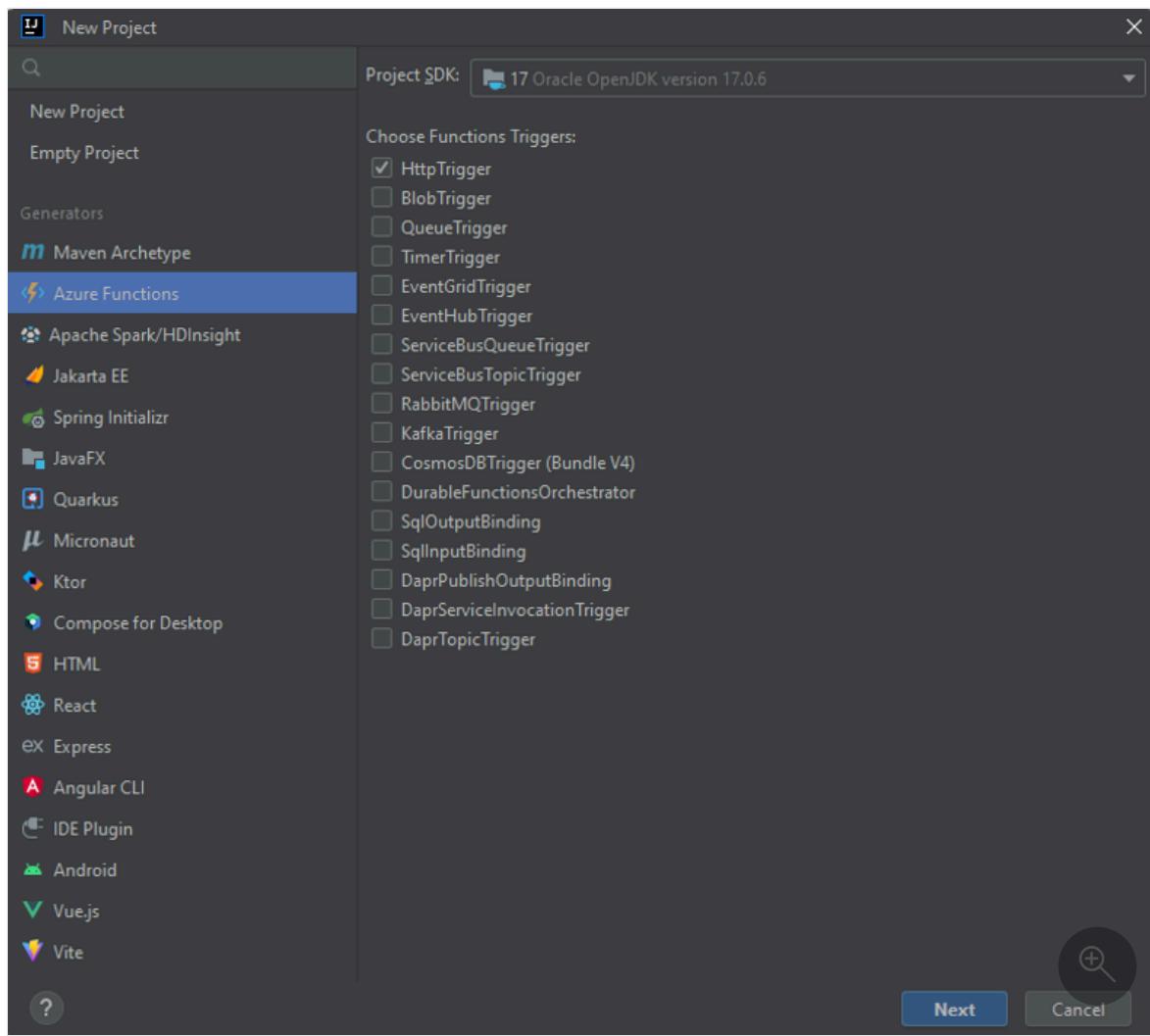
4. In the browser, sign in with your account and then go back to IntelliJ. In the **Select Subscriptions** dialog box, select the subscriptions that you want to use, then select **Select**.



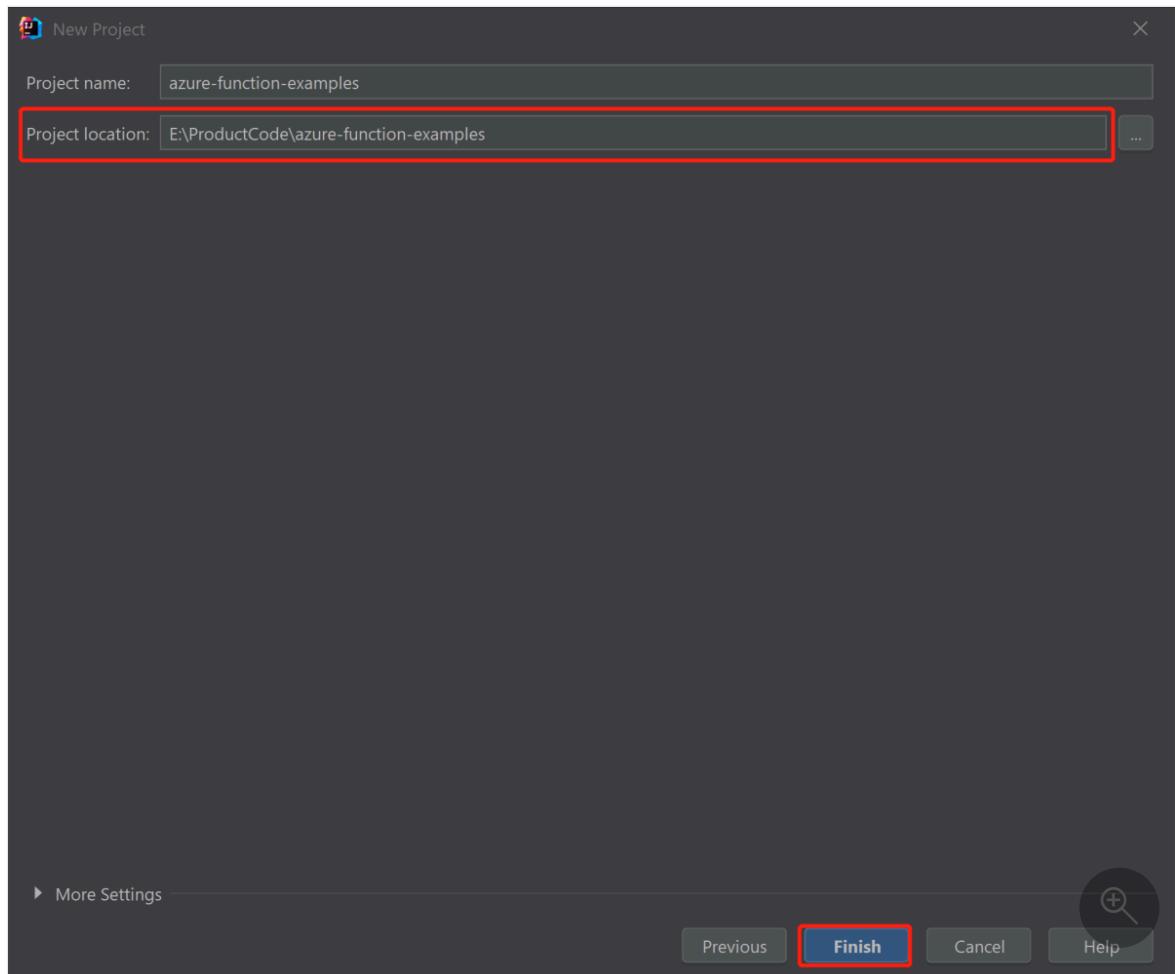
Create your local project

To use Azure Toolkit for IntelliJ to create a local Azure Functions project, follow these steps:

1. Open IntelliJ IDEA's **Welcome** dialog, select **New Project** to open a new project wizard, then select **Azure Functions**.



2. Select **Http Trigger**, then select **Next** and follow the wizard to go through all the configurations in the following pages. Confirm your project location, then select **Finish**. IntelliJ IDEA then opens your new project.



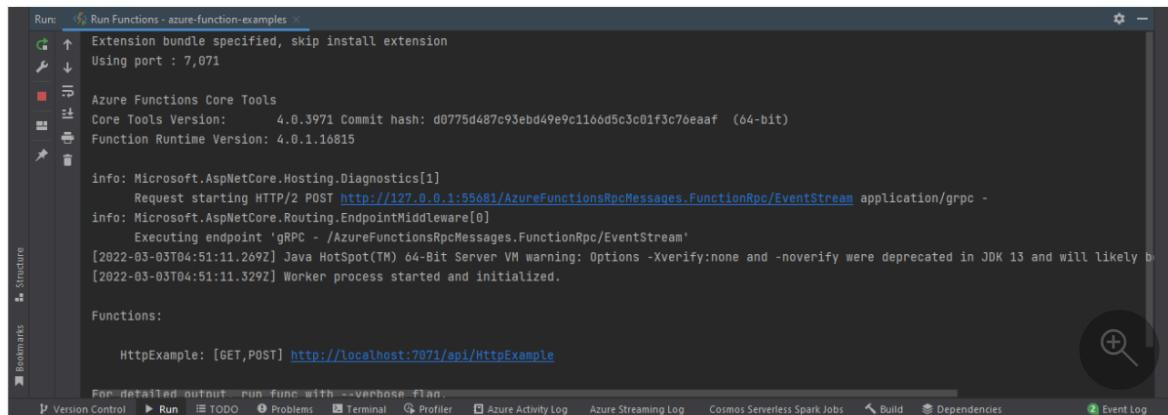
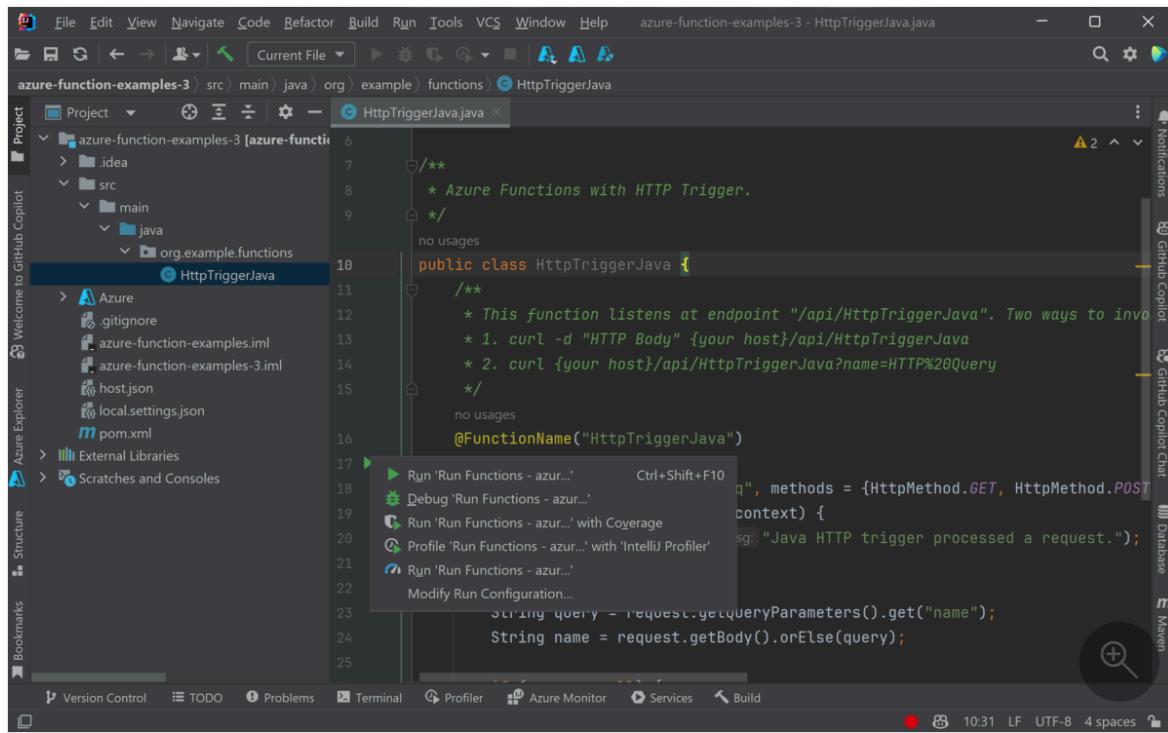
Run the project locally

To run the project locally, follow these steps:

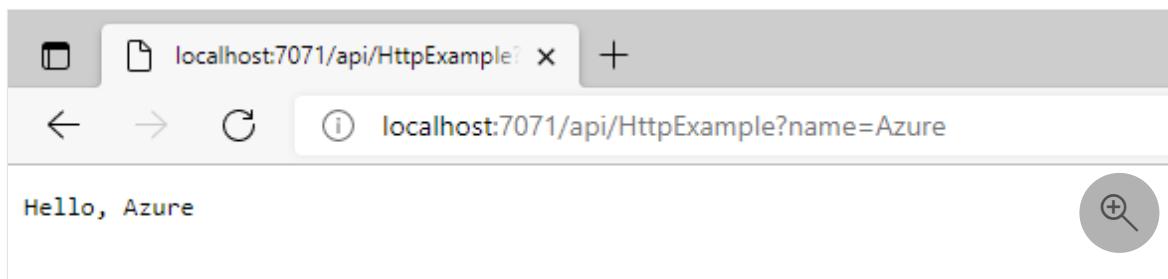
ⓘ Important

You must have the JAVA_HOME environment variable set correctly to the JDK directory that is used during code compiling using Maven. Make sure that the version of the JDK is at least as high as the `Java.version` setting.

1. Navigate to `src/main/java/org/example/functions/HttpTriggerFunction.java` to see the code generated. Beside line 24, you should see a green **Run** button. Select it and then select **Run 'Functions-azur...'**. You should see your function app running locally with a few logs.



2. You can try the function by accessing the displayed endpoint from browser, such as <http://localhost:7071/api/HttpExample?name=Azure>.



3. The log is also displayed in your IDEA. Stop the function app by selecting **Stop**.

```

Run: Run Functions - azure-function-examples
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
    Request starting HTTP/2 POST http://127.0.0.1:60051/AzureFunctionsRpcMessages.FunctionRpc/EventStream application/grpc -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
    Executing endpoint 'gRPC - /AzureFunctionsRpcMessages.FunctionRpc/EventStream'
[2022-03-03T04:56:58.644Z] Java HotSpot(TM) 64-Bit Server VM warning: Options -Xverify:none and -noverify were deprecated in JDK 13 and will likely be removed in a future release.
[2022-03-03T04:56:58.742Z] Worker process started and initialized.

Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-03-03T04:57:03.684Z] Host lock lease acquired by instance ID '000000000000000000000000003AA62615'.
[2022-03-03T04:57:04.117Z] Executing 'Functions.HttpExample' (Reason='This function was programmatically called via the host APIs.', Id=41fb3f91-c517-4522-8308-a668c5b1c4a3)
[2022-03-03T04:57:04.194Z] Java HTTP trigger processed a request.
[2022-03-03T04:57:04.194Z] Function 'HttpExample' (Id: 41fb3f91-c517-4522-8308-a668c5b1c4a3) invoked by Java Worker
[2022-03-03T04:57:04.280Z] Executed 'Functions.HttpExample' (Succeeded, Id=41fb3f91-c517-4522-8308-a668c5b1c4a3, Duration=179ms)


```

The screenshot shows the Azure Functions Java extension in Visual Studio Code. The 'Run' toolbar button is highlighted with a red box. The terminal window displays logs for a function named 'HttpExample'. The logs show the function being triggered via a Java HTTP request and successfully executed.

Debug the project locally

To debug the project locally, follow these steps:

1. Select the **Debug** button in the toolbar. If you don't see the toolbar, enable it by choosing **View > Appearance > Toolbar**.

The screenshot shows the Azure Functions Java extension in Visual Studio Code. A breakpoint is set on line 20 of the file `HttpTriggerJava.java`. The code defines an HTTP trigger function named `HttpTriggerJava` with a `@FunctionName("HttpTriggerJava")` annotation. The function body includes logic to handle requests and parse query parameters.

```

public class HttpTriggerJava {
    /**
     * This function listens at endpoint "/api/HttpTriggerJava". Two ways to invoke
     * 1. curl -d "HTTP Body" {your host}/api/HttpTriggerJava
     * 2. curl {your host}/api/HttpTriggerJava?name=HTTP%20Query
     */
    @FunctionName("HttpTriggerJava")
    public HttpResponseMessage run(
            @HttpTrigger(name = "req", methods = {HttpMethod.GET, HttpMethod.POST},
            final ExecutionContext context) {
        context.getLogger().info("Java HTTP trigger processed a request.");

        // Parse query parameter
        String query = request.getQueryParameters().get("name");
        String name = request.getBody().orElse(query);

        if (name == null) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST).body("Please provide a name query parameter");
        } else {
            return request.createResponseBuilder(HttpStatus.OK).body("Hello " + name);
        }
    }
}

```

2. Select line 20 of the file

`src/main/java/org/example/functions/HttpTriggerFunction.java` to add a breakpoint. Access the endpoint `http://localhost:7071/api/HttpTrigger-Java?name=Azure` again and you should find that the breakpoint is hit. You can then try more debug features like **Step**, **Watch**, and **Evaluation**. Stop the debug session by selecting **Stop**.

```

final ExecutionContext context) { context: ExecutionContext
context.getLogger().info( msg: "Java HTTP trigger processed a request.");
// Parse query parameter
String query = request.getQueryParameters().get("name");
String name = request.getBody().orElse(query);

if (name == null) {
    return request.createResponseBuilder(HttpStatus.BAD_REQUEST).body(
} else {
    return request.createResponseBuilder(HttpStatus.OK).body("Hello, " + name);
}

```

Create the function app in Azure

Use the following steps create a function app and related resources in your Azure subscription:

1. In Azure Explorer in your IDEA, right-click **Function App** and then select **Create**.
2. Select **More Settings** and provide the following information at the prompts:

[Expand table](#)

Prompt	Selection
Subscription	Choose the subscription to use.
Resource Group	Choose the resource group for your function app.
Name	Specify the name for a new function app. Here you can accept the default value.
Platform	Select Windows-Java 17 or another platform as appropriate.
Region	For better performance, choose a region near you.
Hosting Options	Choose the hosting options for your function app.
Plan	Choose the App Service plan pricing tier you want to use, or select + to create a new App Service plan.

ⓘ Important

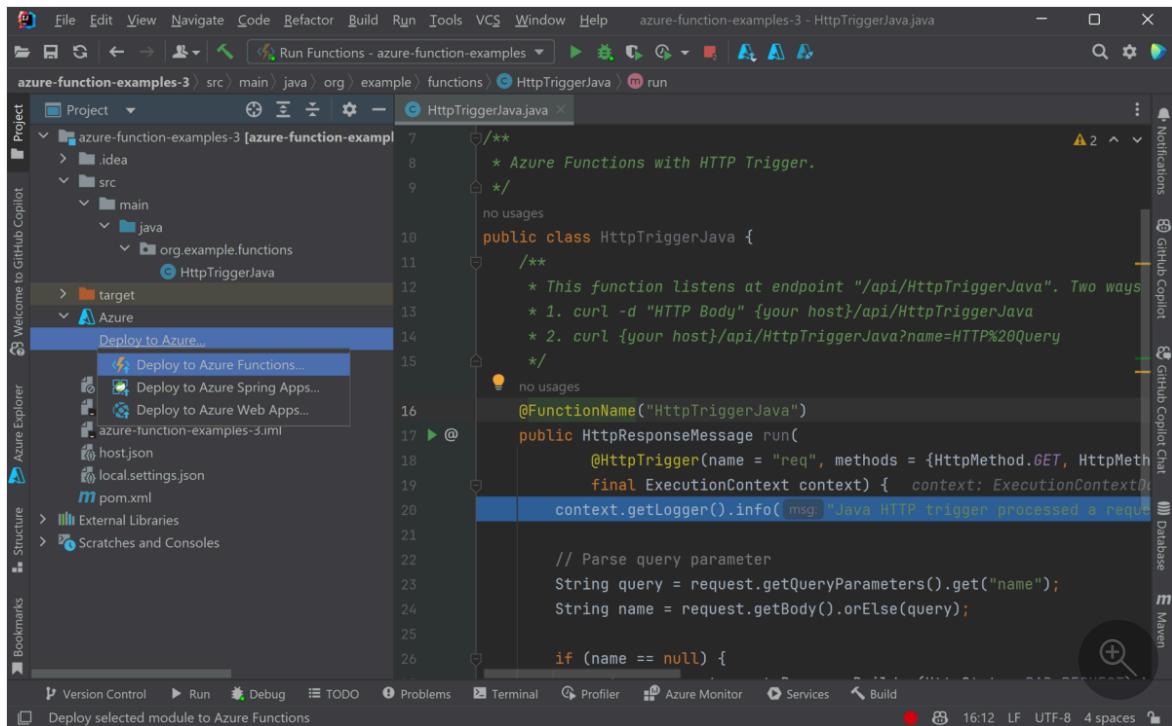
To create your app in the Flex Consumption plan, select **Flex Consumption**.
The [Flex Consumption plan](#) is currently in preview.

3. Select **OK**. A notification is displayed after your function app is created.

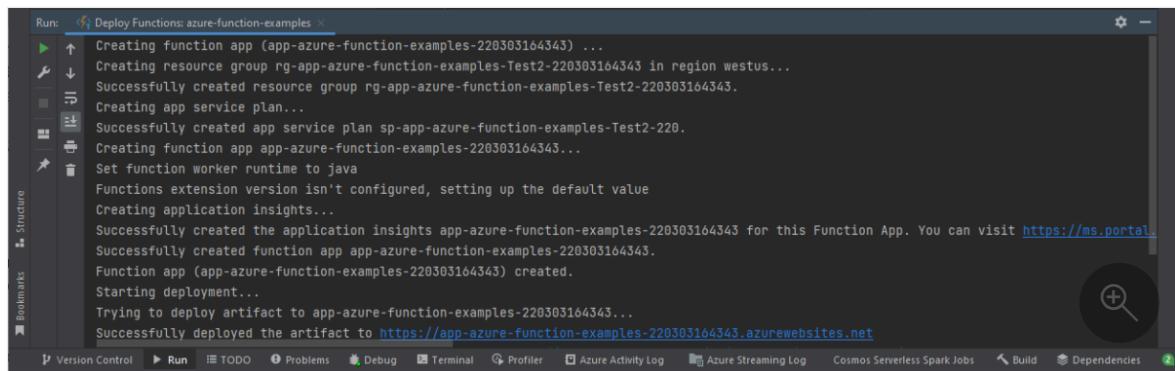
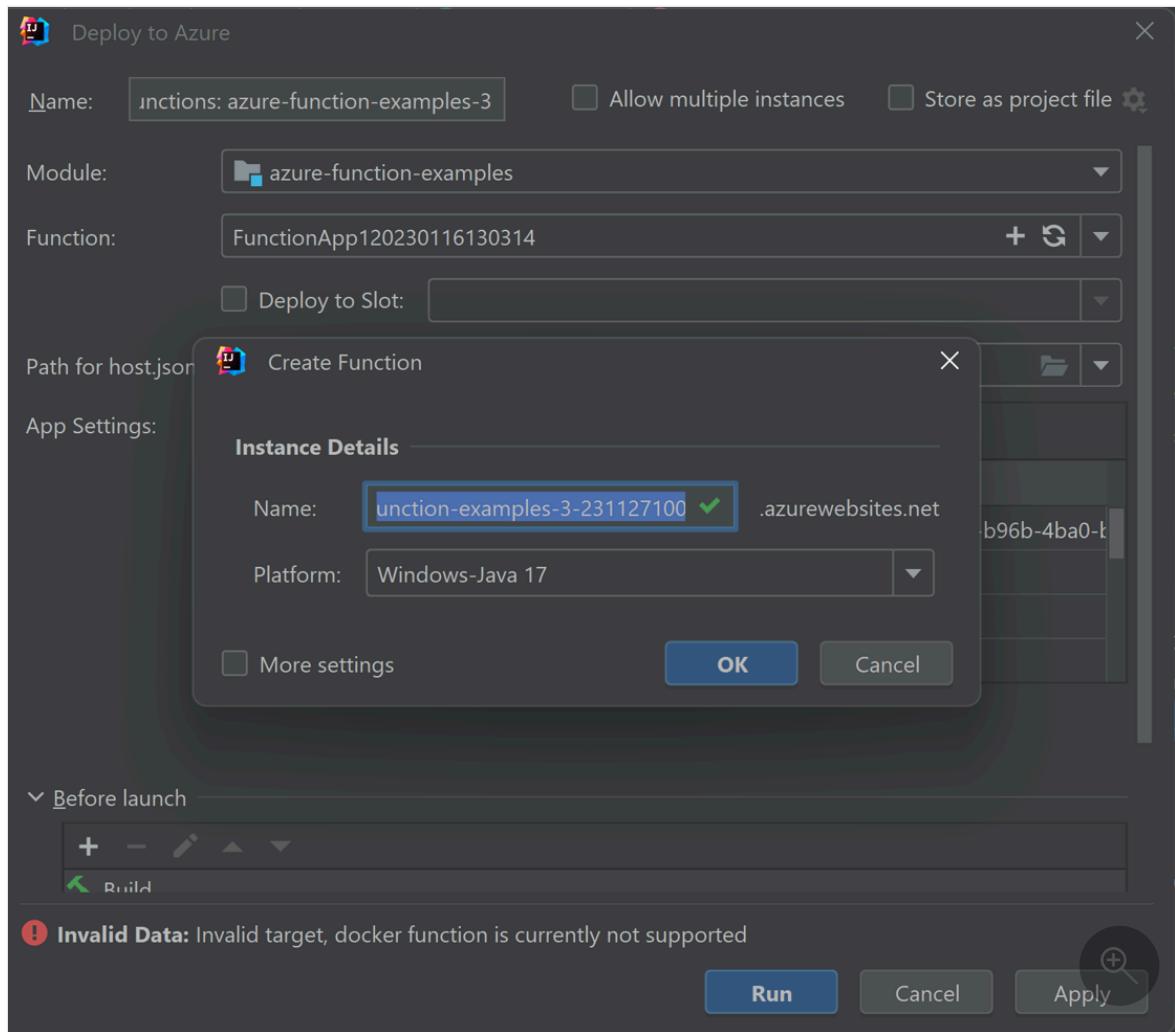
Deploy your project to Azure

To deploy your project to Azure, follow these steps:

1. Select and expand the Azure icon in IntelliJ Project explorer, then select **Deploy to Azure -> Deploy to Azure Functions**.



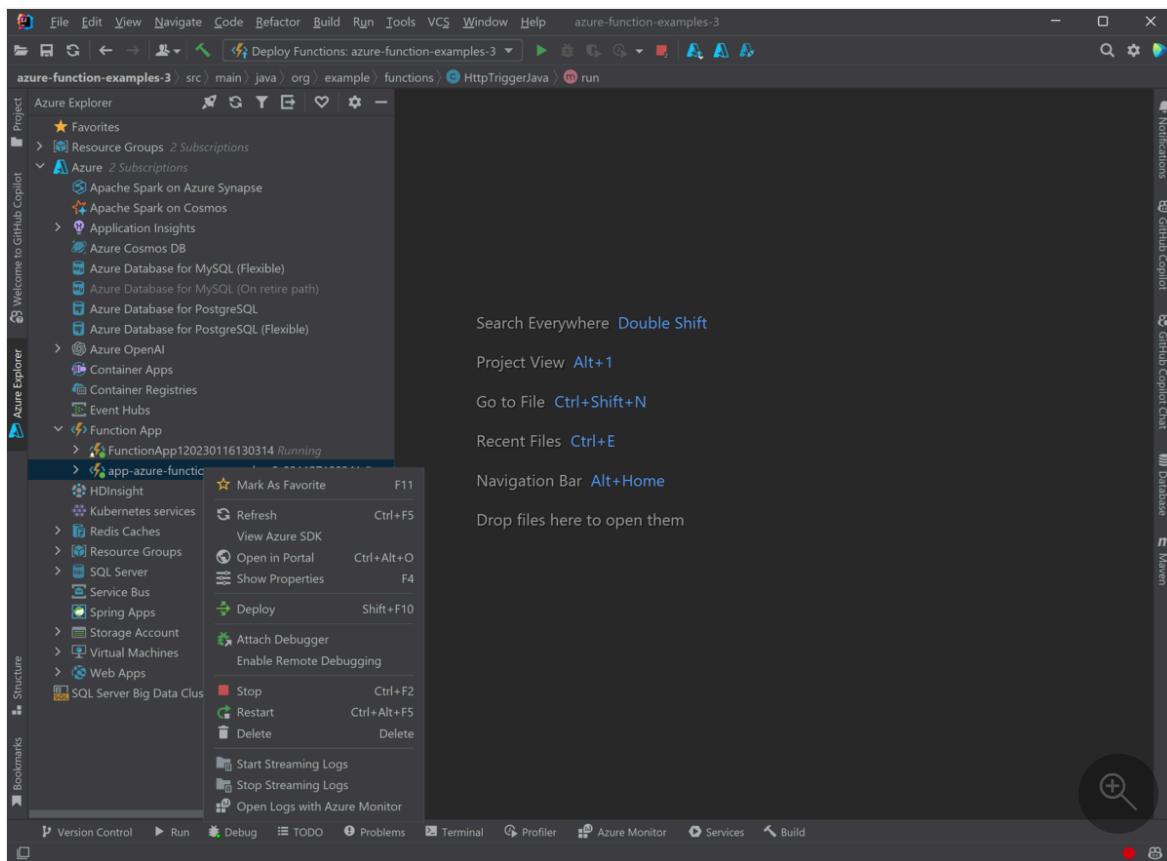
2. You can select the function app from the previous section. To create a new one, select **+** on the **Function** line. Type in the function app name and choose the proper platform. Here, you can accept the default value. Select **OK** and the new function app you created is automatically selected. Select **Run** to deploy your functions.



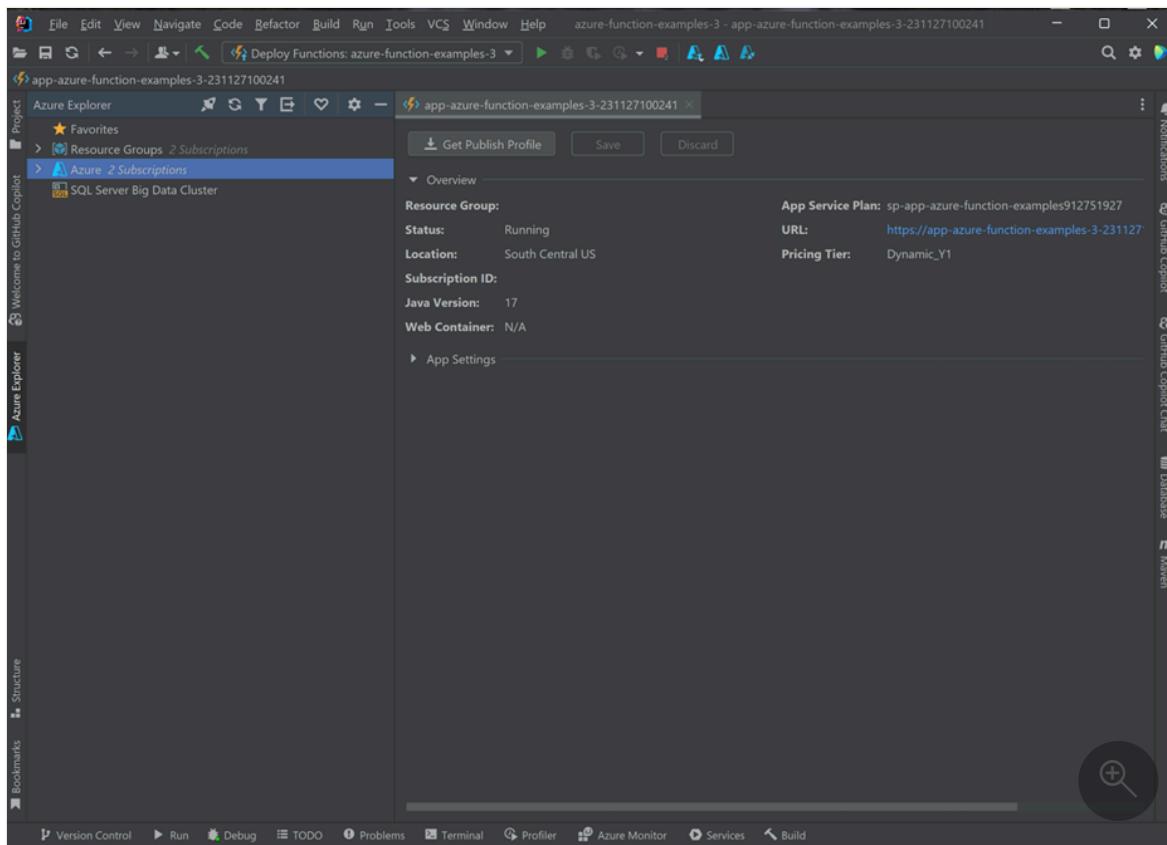
Manage function apps from IDEA

To manage your function apps with **Azure Explorer** in your IDEA, follow these steps:

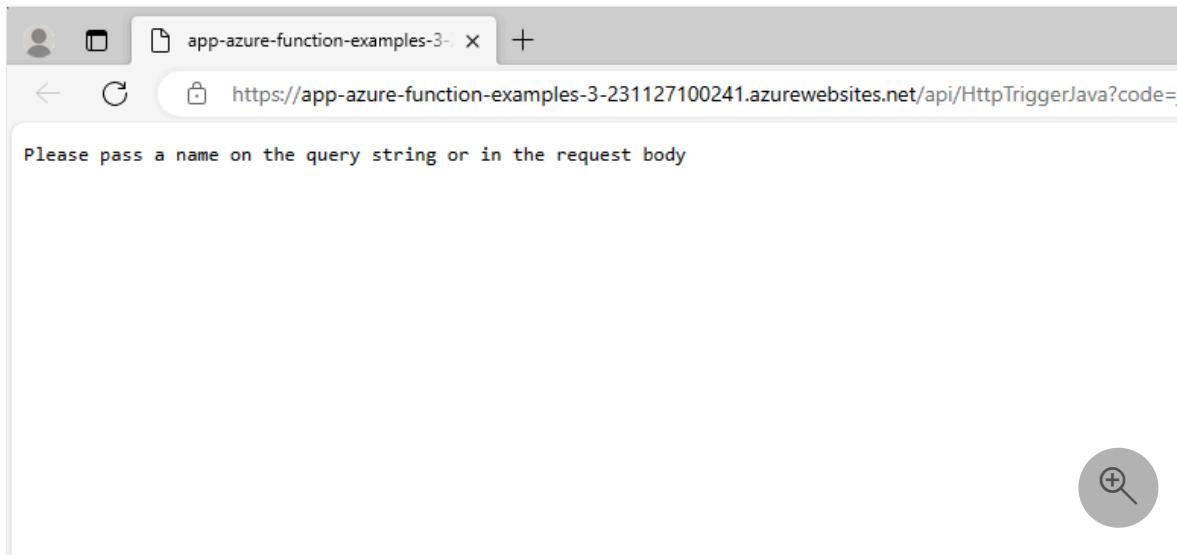
1. Select **Function App** to see all your function apps listed.



2. Select one of your function apps, then right-click and select **Show Properties** to open the detail page.



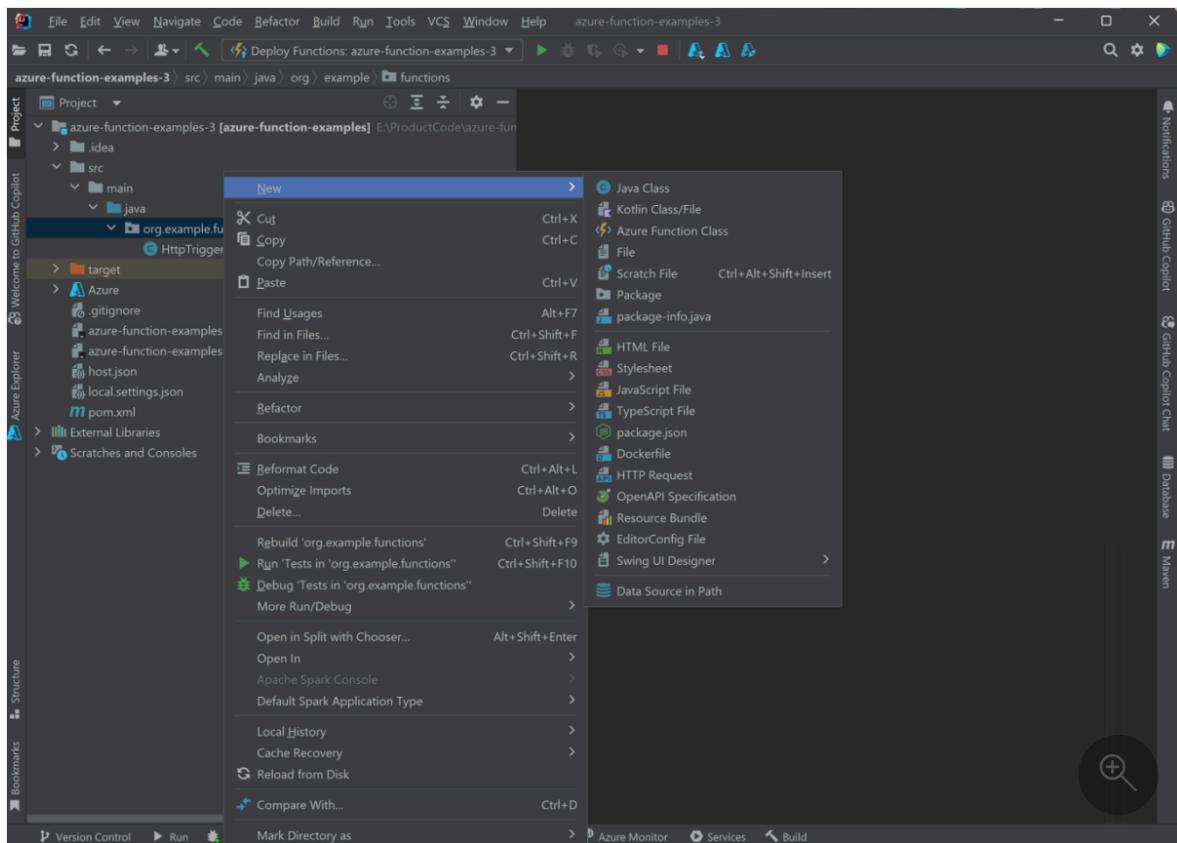
3. Right-click your **HttpTrigger-Java** function app, then select **Trigger Function in Browser**. You should see that the browser is opened with the trigger URL.



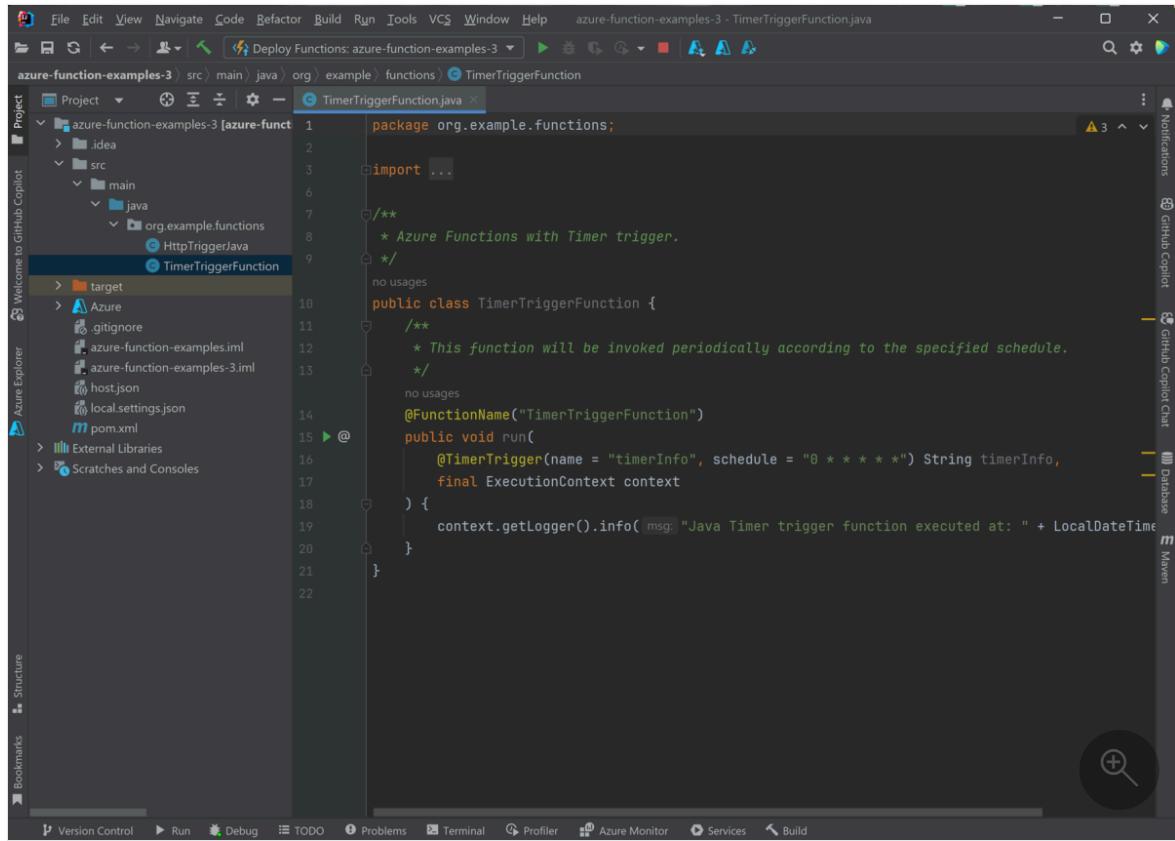
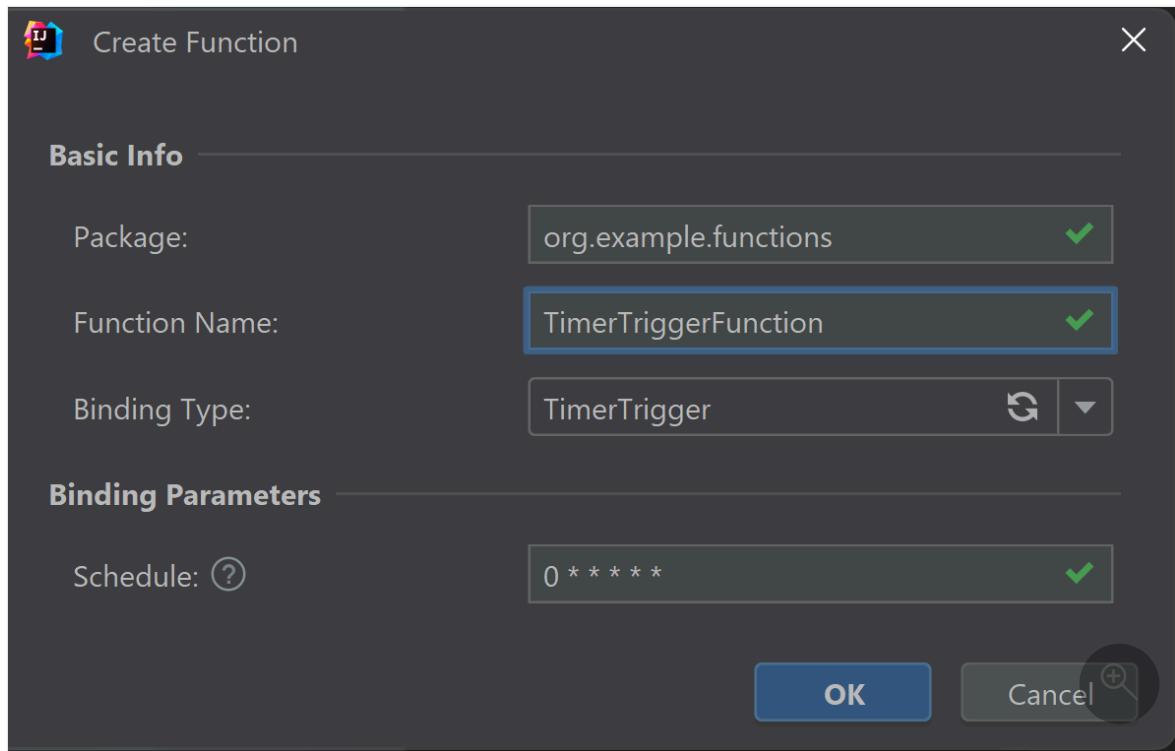
Add more functions to the project

To add more functions to your project, follow these steps:

1. Right-click the package `org.example.functions` and select **New -> Azure Function Class**.

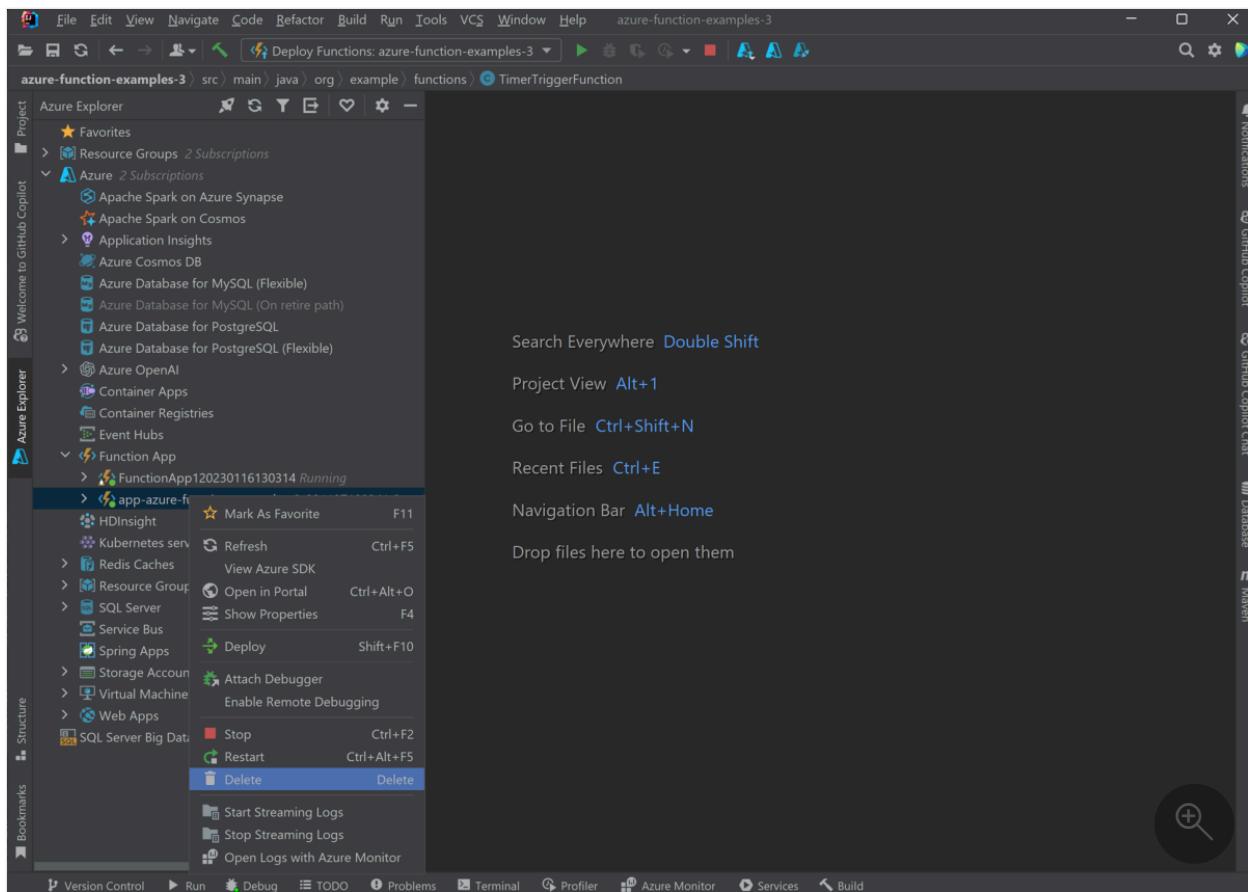


2. Fill in the class name **HttpTest** and select **HttpTrigger** in the create function class wizard, then select **OK** to create. In this way, you can create new functions as you want.



Cleaning up functions

Select one of your function apps using **Azure Explorer** in your IDEA, then right-click and select **Delete**. This command might take several minutes to run. When it's done, the status refreshes in **Azure Explorer**.



Next steps

You've created a Java project with an HTTP triggered function, run it on your local machine, and deployed it to Azure. Now, extend your function by continuing to the following article:

[Adding an Azure Storage queue output binding](#)

Quickstart: Create a Java function in Azure from the command line

Article • 08/24/2023

In this article, you use command-line tools to create a Java function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

If Maven isn't your preferred development tool, check out our similar tutorials for Java developers:

- [Gradle](#)
- [IntelliJ IDEA](#)
- [Visual Studio Code](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Configure your local environment

Before you begin, you must have the following:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- The [Azure CLI](#) version 2.4 or later.
- The [Java Developer Kit](#), version 8 or 11. The `JAVA_HOME` environment variable must be set to the install location of the correct version of the JDK.
- [Apache Maven ↗](#), version 3.0 or above.

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme ↗](#).

Download and run the Core Tools installer, based on your version of Windows:

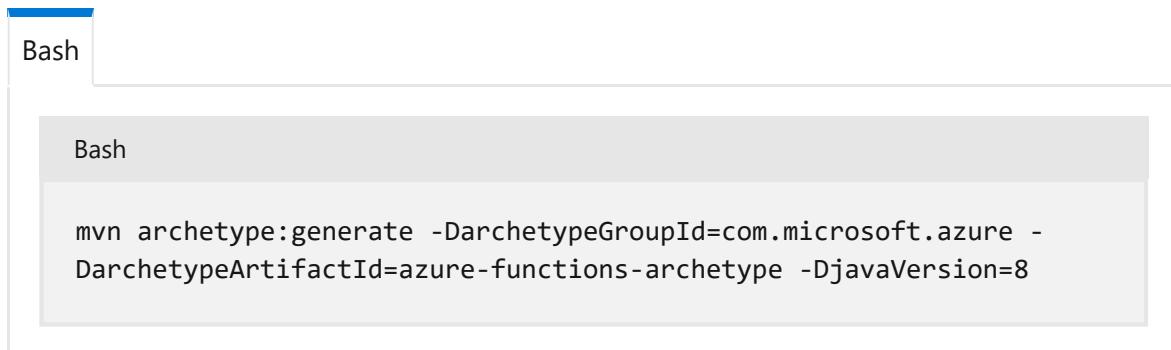
- [v4.x - Windows 64-bit ↗](#) (Recommended. Visual Studio Code debugging requires 64-bit.)
- [v4.x - Windows 32-bit ↗](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

Create a local function project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single function.

1. In an empty folder, run the following command to generate the Functions project from a [Maven archetype ↗](#).



A screenshot of a terminal window titled "Bash". The command `mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure -DarchetypeArtifactId=azure-functions-archetype -DjavaVersion=8` is displayed in the terminal window.

ⓘ Important

- Use `-DjavaVersion=11` if you want your functions to run on Java 11. To learn more, see [Java versions](#).
- The `JAVA_HOME` environment variable must be set to the install location of the correct version of the JDK to complete this article.

2. Maven asks you for values needed to finish generating the project on deployment. Provide the following values when prompted:

Prompt	Value	Description
groupId	com.fabrikam	A value that uniquely identifies your project across all projects, following the package naming rules for Java.
artifactId	fabrikam-functions	A value that is the name of the jar, without a version number.
version	1.0-SNAPSHOT	Choose the default value.
package	com.fabrikam	A value that is the Java package for the generated function code. Use the default.

3. Type `Y` or press Enter to confirm.

Maven creates the project files in a new folder with a name of *artifactId*, which in this example is `fabrikam-functions`.

4. Navigate into the project folder:

```
Console
cd fabrikam-functions
```

This folder contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file.

(Optional) Examine the file contents

If desired, you can skip to [Run the function locally](#) and examine the file contents later.

Function.java

Function.java contains a `run` method that receives request data in the `request` variable is an `HttpRequestMessage` that's decorated with the `HttpTrigger` annotation, which defines the trigger behavior.

```
Java
/**
 * Copyright (c) Microsoft Corporation. All rights reserved.
 * Licensed under the MIT License. See License.txt in the project root for
 * license information.
```

```

*/
package com.functions;

import com.microsoft.azure.functions.ExecutionContext;
import com.microsoft.azure.functions.HttpMethod;
import com.microsoft.azure.functions.HttpRequestMessage;
import com.microsoft.azure.functions HttpResponseMessage;
import com.microsoft.azure.functions.HttpStatus;
import com.microsoft.azure.functions.annotation.AuthorizationLevel;
import com.microsoft.azure.functions.annotation.FixedDelayRetry;
import com.microsoft.azure.functions.annotation.FunctionName;
import com.microsoft.azure.functions.annotation.HttpTrigger;

import java.io.File;
import java.nio.file.Files;
import java.util.Optional;

/**
 * Azure Functions with HTTP Trigger.
 */
public class Function {
    /**
     * This function listens at endpoint "/api/HttpExample". Two ways to
     * invoke it using "curl" command in bash:
     * 1. curl -d "HTTP Body" {your host}/api/HttpExample
     * 2. curl "{your host}/api/HttpExample?name=HTTP%20Query"
     */
    @FunctionName("HttpExample")
    public HttpResponseMessage run(
        @HttpTrigger(
            name = "req",
            methods = {HttpMethod.GET, HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<String>> request,
        final ExecutionContext context) {
        context.getLogger().info("Java HTTP trigger processed a request.");

        // Parse query parameter
        final String query = request.getQueryParameters().get("name");
        final String name = request.getBody().orElse(query);

        if (name == null) {
            return
request.createResponseBuilder(HttpStatus.BAD_REQUEST).body("Please pass a
name on the query string or in the request body").build();
        } else {
            return request.createResponseBuilder(HttpStatus.OK).body("Hello,
" + name).build();
        }
    }

    public static int count = 1;

    /**

```

```
* This function listens at endpoint "/api/HttpExampleRetry". The
function is re-executed in case of errors until the maximum number of
retries occur.
 * Retry policies: https://docs.microsoft.com/en-us/azure/azure-
functions/functions-bindings-error-pages?tabs=java
 */
@FunctionName("HttpExampleRetry")
@FixedDelayRetry(maxRetryCount = 3, delayInterval = "00:00:05")
public HttpResponseMessage HttpExampleRetry(
    @HttpTrigger(
        name = "req",
        methods = {HttpMethod.GET, HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context) throws Exception {
    context.getLogger().info("Java HTTP trigger processed a request.");
}

if(count<3) {
    count++;
    throw new Exception("error");
}

// Parse query parameter
final String query = request.getQueryParameters().get("name");
final String name = request.getBody().orElse(query);

if (name == null) {
    return
request.createResponseBuilder(HttpStatus.BAD_REQUEST).body("Please pass a
name on the query string or in the request body").build();
} else {
    return
request.createResponseBuilder(HttpStatus.OK).body(name).build();
}

/***
 * This function listens at endpoint "/api/HttpTriggerJavaVersion".
 * It can be used to verify the Java home and java version currently in
use in your Azure function
 */
@FunctionName("HttpTriggerJavaVersion")
public static HttpResponseMessage HttpTriggerJavaVersion(
    @HttpTrigger(
        name = "req",
        methods = {HttpMethod.GET, HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context
) {
    context.getLogger().info("Java HTTP trigger processed a request.");
    final String javaVersion = getJavaVersion();
    context.getLogger().info("Function - HttpTriggerJavaVersion" +
javaVersion);
    return
```

```

        request.createResponseBuilder(HttpStatus.OK).body("HttpTriggerJavaVersion").
        build();
    }

    public static String getJavaVersion() {
        return String.join(" - ", System.getProperty("java.home"),
System.getProperty("java.version"));
    }

    /**
     * This function listens at endpoint "/api/StaticWebPage".
     * It can be used to read and serve a static web page.
     * Note: Read the file from the right location for local machine and
     * azure portal usage.
     */
    @FunctionName("StaticWebPage")
    public HttpResponseMessage getStaticWebPage(
        @HttpTrigger(
            name = "req",
            methods = {HttpMethod.GET, HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<String>> request,
        final ExecutionContext context) {
        context.getLogger().info("Java HTTP trigger processed a request.");

        File htmlFile = new File("index.html");
        try{
            byte[] fileContent = Files.readAllBytes(htmlFile.toPath());
            return
request.createResponseBuilder(HttpStatus.OK).body(fileContent).build();
        }catch (Exception e){
            context.getLogger().info("Error reading file.");
            return
request.createResponseBuilder(HttpStatus.INTERNAL_SERVER_ERROR).build();
        }
    }
}

```

The response message is generated by the [HttpResponseMessage.Builder](#) API.

pom.xml

Settings for the Azure resources created to host your app are defined in the **configuration** element of the plugin with a **groupId** of `com.microsoft.azure` in the generated pom.xml file. For example, the configuration element below instructs a Maven-based deployment to create a function app in the `java-functions-group` resource group in the `westus` region. The function app itself runs on Windows hosted in the `java-functions-app-service-plan` plan, which by default is a serverless Consumption plan.

Java

```
        <encoding>${project.build.sourceEncoding}</encoding>
    </configuration>
</plugin>
<plugin>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-functions-maven-plugin</artifactId>
    <version>${azure.functions.maven.plugin.version}</version>
    <configuration>
        <!-- function app name -->
        <appName>${functionAppName}</appName>
        <!-- function app resource group -->
        <resourceGroup>${functionResourceGroup}</resourceGroup>
        <!-- function app service plan name -->
        <appServicePlanName>${functionServicePlan}</appServicePlanName>
        <!-- function app region-->
        <!-- refers https://github.com/microsoft/azure-maven-
plugins/wiki/Azure-Functions:-Configuration-Details#supported-regions for
all valid values -->
        <region>${functionAppRegion}</region>
        <!-- function pricingTier, default to be consumption if not
specified -->
        <!-- refers https://github.com/microsoft/azure-maven-
plugins/wiki/Azure-Functions:-Configuration-Details#supported-pricing-tiers
for all valid values -->
        <pricingTier>${functionPricingTier}</pricingTier>
        <!-- Whether to disable application insights, default is false -->
        <!-- refers https://github.com/microsoft/azure-maven-
plugins/wiki/Azure-Functions:-Configuration-Details for all valid
configurations for application insights-->
        <!-- <disableAppInsights></disableAppInsights> -->

        <runtime>
            <!-- runtime os, could be windows, linux or docker-->
            <os>windows</os>
            <javaVersion>8</javaVersion>
            <!-- for docker function, please set the following parameters -->
        >
            <!-- <image>[hub-user/]repo-name[:tag]</image> -->
            <!-- <serverId></serverId> -->
            <!-- <registryUrl></registryUrl> -->
        </runtime>
        <appSettings>
            <property>
                <name>FUNCTIONS_EXTENSION_VERSION</name>
                <value>~4</value>
            </property>
        </appSettings>
    </configuration>
    <executions>
        <execution>
            <id>package-functions</id>
            <goals>
```

```
<goal>package</goal>
</goals>
```

You can change these settings to control how resources are created in Azure, such as by changing `runtime.os` from `windows` to `linux` before initial deployment. For a complete list of settings supported by the Maven plug-in, see the [configuration details](#).

FunctionTest.java

The archetype also generates a unit test for your function. When you change your function to add bindings or add new functions to the project, you'll also need to modify the tests in the `FunctionTest.java` file.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the `LocalFunctionProj` folder:

```
Console

mvn clean package
mvn azure-functions:run
```

Toward the end of the output, the following lines should appear:

```
...
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
...
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the

host, navigate to the project's root folder, and run the previous command again.

2. Copy the URL of your `HttpExample` function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, use `Ctrl+C` and choose `y` to stop the functions host.

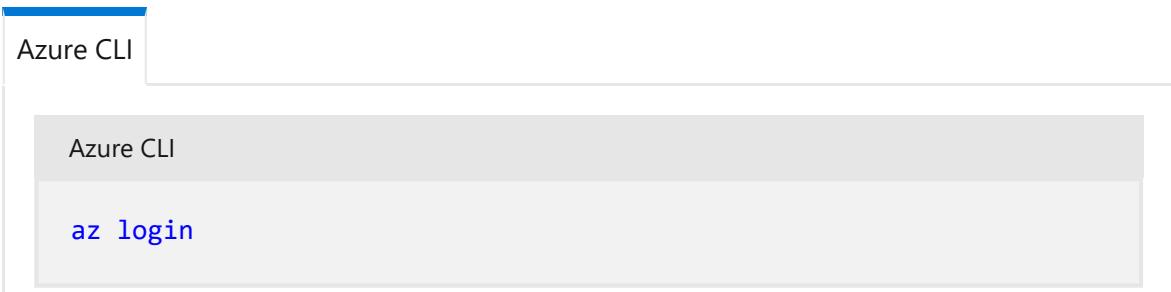
Deploy the function project to Azure

A function app and related resources are created in Azure when you first deploy your functions project. Settings for the Azure resources created to host your app are defined in the [pom.xml file](#). In this article, you'll accept the defaults.

Tip

To create a function app running on Linux instead of Windows, change the `runtime.os` element in the `pom.xml` file from `windows` to `linux`. Running Linux in a consumption plan is supported in [these regions](#). You can't have apps that run on Linux and apps that run on Windows in the same resource group.

1. Before you can deploy, sign in to your Azure subscription using either Azure CLI or Azure PowerShell.



The screenshot shows the Azure CLI interface. A dropdown menu is open, with 'Azure CLI' selected. Below it, the command `az login` is typed into the input field.

The `az login` command signs you into your Azure account.

2. Use the following command to deploy your project to a new function app.



The screenshot shows a Maven console window. The title bar says 'Console'. In the main area, the command `mvn azure-functions:deploy` is visible.

This creates the following resources in Azure:

- Resource group. Named as *java-functions-group*.
- Storage account. Required by Functions. The name is generated randomly based on Storage account name requirements.
- Hosting plan. Serverless hosting for your function app in the *westus* region. The name is *java-functions-app-service-plan*.
- Function app. A function app is the deployment and execution unit for your functions. The name is randomly generated based on your *artifactId*, appended with a randomly generated number.

The deployment packages the project files and deploys them to the new function app using [zip deployment](#). The code runs from the deployment package in Azure.

Important

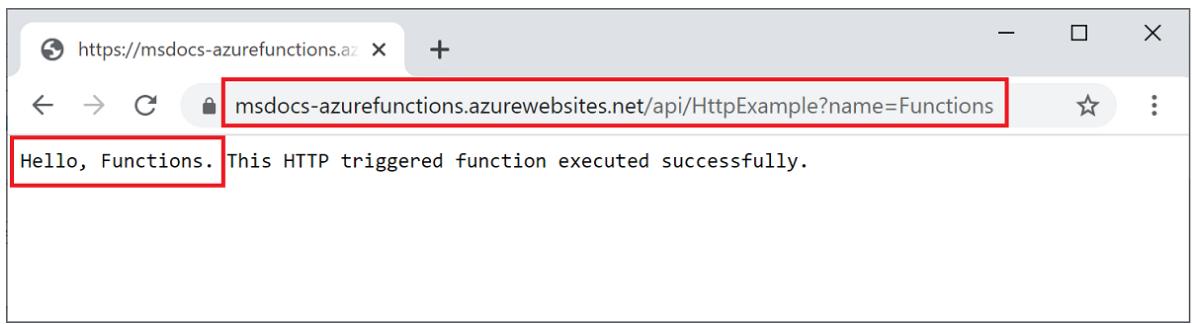
The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.



Hello, Functions. This HTTP triggered function executed successfully.

Run the following command to view near real-time streaming logs:

Console

```
func azure functionapp logstream <APP_NAME>
```

In a separate terminal window or in the browser, call the remote function again. A verbose log of the function execution in Azure is shown in the terminal.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

Azure CLI

```
az group delete --name java-functions-group
```

Next steps

[Connect to an Azure Storage queue](#)

Deploy serverless Java apps with Quarkus on Azure Functions

Article • 03/17/2023

In this article, you'll develop, build, and deploy a serverless Java app to Azure Functions by using [Quarkus ↗](#). This article uses Quarkus Funqy and its built-in support for the Azure Functions HTTP trigger for Java. Using Quarkus with Azure Functions gives you the power of the Quarkus programming model with the scale and flexibility of Azure Functions. When you finish, you'll run serverless Quarkus applications on Azure Functions and continue to monitor your app on Azure.

Prerequisites

- The [Azure CLI](#) installed on your own computer.
- An [Azure account ↗](#). If you don't have an [Azure subscription](#), create an [Azure free account ↗](#) before you begin.
- [Java JDK 17](#) with `JAVA_HOME` configured appropriately. This article was written with Java 17 in mind, but Azure Functions and Quarkus also support older versions of Java.
- [Apache Maven 3.8.1+ ↗](#).

Create the app project

Use the following command to clone the sample Java project for this article. The sample is on [GitHub ↗](#).

Bash

```
git clone https://github.com/Azure-Samples/quarkus-azure
```

Explore the sample function. Open the *functions-quarkus/src/main/java/io/quarkus/GreetingFunction.java* file.

Run the following command. The `@Funq` annotation makes your method (in this case, `funqyHello`) a serverless function.

Java

```
@Funq  
public String funqyHello() {
```

```
        return "hello funqy";
    }
```

Azure Functions Java has its own set of Azure-specific annotations, but these annotations aren't necessary when you're using Quarkus on Azure Functions in a simple capacity as we're doing here. For more information about Azure Functions Java annotations, see the [Azure Functions Java developer guide](#).

Unless you specify otherwise, the function's name is the same as the method name. You can also use the following command to define the function name with a parameter to the annotation:

Java

```
@Funq("alternateName")
public String funqHello() {
    return "hello funqy";
}
```

The name is important. It becomes a part of the REST URI to invoke the function, as shown later in the article.

Test the function locally

Use `mvn` to run Quarkus dev mode on your local terminal. Running Quarkus in this way enables live reload with background compilation. When you modify your Java files and/or your resource files and refresh your browser, these changes will automatically take effect.

A browser refresh triggers a scan of the workspace. If the scan detects any changes, the Java files are recompiled and the application is redeployed. Your redeployed application services the request. If there are any problems with compilation or deployment, an error page will let you know.

In the following procedure, replace `yourResourceGroupName` with a resource group name. Function app names must be globally unique across all of Azure. Resource group names must be globally unique within a subscription. This article achieves the necessary uniqueness by prepending the resource group name to the function name. Consider prepending a unique identifier to any names you create that must be unique. A useful technique is to use your initials followed by today's date in `mmdd` format.

The resource group is not necessary for this part of the instructions, but it's required later. For simplicity, the Maven project requires you to define the property.

1. Invoke Quarkus dev mode:

Bash

```
cd functions-azure  
mvn -DskipTests -DresourceGroup=<yourResourceGroupName> quarkus:dev
```

The output should look like this:

Output

```
...  
--/ _ \ / / / _ | / _ \ / / / / / _/  
-/ /_ / / / / _ | / , _/ , < / /_ / \ \ /  
--\_\_\_\_\_\_\_/_|/_/_/_/_/_|\_\_\_\_\_\_/_/  
INFO [io.quarkus] (Quarkus Main Thread) quarkus-azure-function 1.0-  
SNAPSHOT on JVM (powered by Quarkus xx.xx.xx.) started in 1.290s.  
Listening on: http://localhost:8080  
  
INFO [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live  
Coding activated.  
INFO [io.quarkus] (Quarkus Main Thread) Installed features: [cdi,  
funqy-http, smallrye-context-propagation, vertx]  
  
--  
Tests paused  
Press [r] to resume testing, [o] Toggle test output, [:] for the  
terminal, [h] for more options>
```

2. Access the function by using the `CURL` command on your local terminal:

Bash

```
curl localhost:8080/api/funqyHello
```

The output should look like this:

Output

```
"hello funqy"
```

Add dependency injection to the function

The open-standard technology Jakarta EE Contexts and Dependency Injection (CDI) provides dependency injection in Quarkus. For a high-level overview of injection in general, and CDI specifically, see the [Jakarta EE tutorial ↗](#).

1. Add a new function that uses dependency injection.

Create a `GreetingService.java` file in the `functions-quarkus/src/main/java/io/quarkus` directory. Use the following code as the source code of the file:

Java

```
package io.quarkus;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class GreetingService {

    public String greeting(String name) {
        return "Welcome to build Serverless Java with Quarkus on Azure
Functions, " + name;
    }

}
```

Save the file.

`GreetingService` is an injectable bean that implements a `greeting()` method. The method returns a `Welcome...` string message with a `name` parameter.

2. Open the existing `functions-`

`quarkus/src/main/java/io/quarkus/GreetingFunction.java` file. Replace the class with the following code to add a new `gService` field and the `greeting` method:

Java

```
package io.quarkus;

import javax.inject.Inject;
import io.quarkus.funqy.Funq;

public class GreetingFunction {

    @Inject
    GreetingService gService;

    @Funq
    public String greeting(String name) {
        return gService.greeting(name);
    }

    @Funq
    public String funqyHello() {
        return "hello funqy";
    }
}
```

```
}
```

```
}
```

Save the file.

3. Access the new `greeting` function by using the `curl` command on your local terminal:

Bash

```
curl -d '"Dan"' -X POST localhost:8080/api/greeting
```

The output should look like this:

Output

```
"Welcome to build Serverless Java with Quarkus on Azure Functions, Dan"
```

ⓘ Important

Live Coding (also called dev mode) allows you to run the app and make changes on the fly. Quarkus will automatically recompile and reload the app when changes are made. This is a powerful and efficient style of developing that you'll use throughout this article.

Before you move forward to the next step, stop Quarkus dev mode by selecting **Ctrl+C**.

Deploy the app to Azure

1. If you haven't already, sign in to your Azure subscription by using the following [az login](#) command and follow the on-screen directions:

Azure CLI

```
az login
```

ⓘ Note

If multiple Azure tenants are associated with your Azure credentials, you must specify which tenant you want to sign in to. You can do this by using the `--tenant` option. For example: `az login --tenant contoso.onmicrosoft.com`.

Continue the process in the web browser. If no web browser is available or if the web browser fails to open, use device code flow with `az login --use-device-code`.

After you sign in successfully, the output on your local terminal should look similar to the following:

Output

```
xxxxxxxx-xxxxx-xxxx-xxxx-xxxxxxxxx 'Microsoft'  
[  
  {  
    "cloudName": "AzureCloud",  
    "homeTenantId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxx",  
    "id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxx",  
    "isDefault": true,  
    "managedByTenants": [],  
    "name": "Contoso account services",  
    "state": "Enabled",  
    "tenantId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx",  
    "user": {  
      "name": "user@contoso.com",  
      "type": "user"  
    }  
  }  
]
```

2. Build and deploy the functions to Azure.

The `pom.xml` file that you generated in the previous step uses `azure-functions-maven-plugin`. Running `mvn install` generates configuration files and a staging directory that `azure-functions-maven-plugin` requires. For `yourResourceGroupName`, use the value that you used previously.

Bash

```
mvn clean install -DskipTests -DtenantId=<your tenantId from shown  
previously> -DresourceGroup=<yourResourceGroupName> azure-  
functions:deploy
```

3. During deployment, sign in to Azure. The `azure-functions-maven-plugin` plug-in is configured to prompt for Azure sign-in each time the project is deployed. During

the build, output similar to the following appears:

Output

```
[INFO] Auth type: DEVICE_CODE  
To sign in, use a web browser to open the page  
https://microsoft.com/devicelogin and enter the code AXCWTLGMP to  
authenticate.
```

Do as the output says and authenticate to Azure by using the browser and the provided device code. Many other authentication and configuration options are available. The complete reference documentation for [azure-functions-maven-plugin](#) is available at [Azure Functions: Configuration Details ↗](#).

4. After authentication, the build should continue and finish. Make sure that output includes `BUILD SUCCESS` near the end.

Output

```
Successfully deployed the artifact to https://quarkus-demo-  
123451234.azurewebsites.net
```

You can also find the URL to trigger your function on Azure in the output log:

Output

```
[INFO] HTTP TriggerUrls:  
[INFO] quarkus : https://quarkus-azure-functions-http-archetype-  
20220629204040017.azurewebsites.net/api/{*path}
```

It will take a while for the deployment to finish. In the meantime, let's explore Azure Functions in the Azure portal.

Access and monitor the serverless function on Azure

Sign in to [the portal ↗](#) and ensure that you've selected the same tenant and subscription that you used in the Azure CLI.

1. Type **function app** on the search bar at the top of the Azure portal and select the Enter key. Your function app should be deployed and show up with the name `<yourResourceGroupName>-function-quarkus`.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons like Home, Dashboard, All services, Favorites, Resource groups, App Services, Function App, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, and Virtual networks. The main area is titled 'Function App' and shows a single item: 'redhat.com'. Below it is a search bar and filter options: 'Subscription equals all', 'Resource group equals all', and 'Add filter'. There are also buttons for 'Create', 'Manage view', 'Refresh', 'Export to CSV', 'Open query', 'Assign tags', 'Start', 'Restart', and more. A table lists the function app details: Name (ejb040110q-functions-quarkus), Status (Running), Location (Central US), Pricing Tier (Dynamic), and App Service Plan (asp-ejb040110q-functions-quarkus). The URL 'https://ejb040110q-functions-quarkus.azurewebsites.net' is shown at the bottom right of the main content area.

2. Select the function app to show detailed information, such as **Location**, **Subscription**, **URL**, **Metrics**, and **App Service Plan**. Then, select the **URL** value.

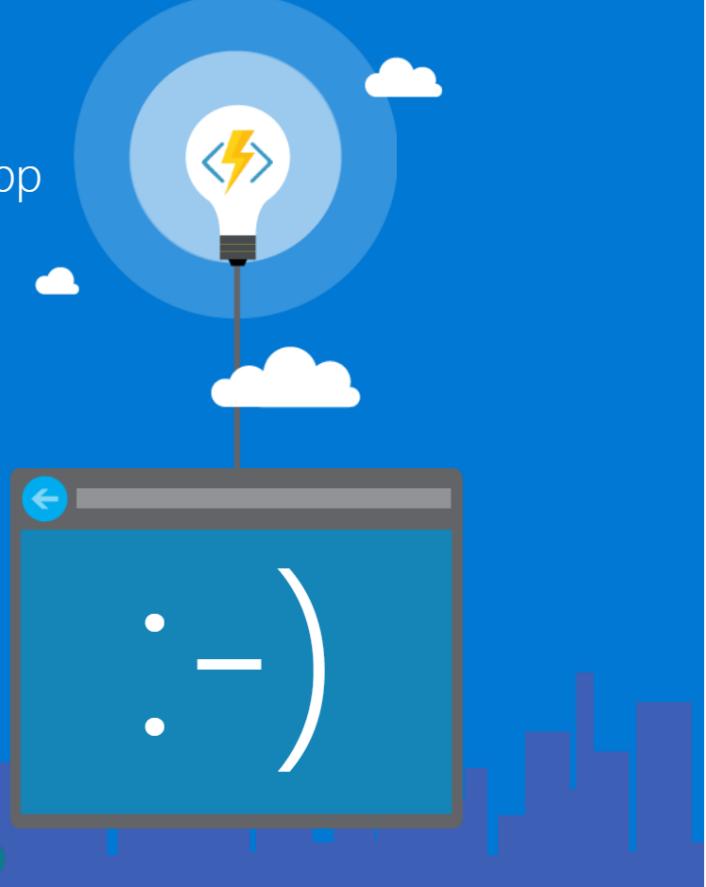
The screenshot shows the 'ejb040110q-functions-quarkus' Function App overview page. On the left, there's a sidebar with links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Functions (Functions, App keys, App files, Proxies), Deployment (Deployment slots, Deployment Center), and Settings. The main content area has tabs for Overview, Browse, Refresh, Stop, Restart, Swap, Get publish profile, Reset publish profile, Download app content, and more. It displays basic app details: Resource group (ejb040110q), Status (Running), Location (Central US), Subscription (Team (EA Subscription 1)), App Service Plan (asp-ejb040110q-functions-quarkus (Y1: 0)), and Properties (See More). It also shows Runtime version (4.14.0.19631). Below this, there are sections for Metrics (Memory working set: 70MB, 60MB) and Function Execution Count (100, 90, 80). At the bottom, there are links for Tags (edit) and Click here to add tags.

3. Confirm that the welcome page says your function app is "up and running."

Your Functions 4.0 app is up and running

Azure Functions is an event-based serverless compute experience to accelerate your development.

Learn more [\(opens in new tab\)](#)



4. Invoke the `greeting` function by using the following `curl` command on your local terminal.

ⓘ Important

Replace `YOUR_HTTP_TRIGGER_URL` with your own function URL that you find in the Azure portal or output.

Bash

```
curl -d '"Dan on Azure"' -X POST
https://YOUR_HTTP_TRIGGER_URL/api/greeting
```

The output should look similar to the following:

Output

```
"Welcome to build Serverless Java with Quarkus on Azure Functions, Dan
on Azure"
```

You can also access the other function (`funqyHello`) by using the following `curl` command:

Bash

```
curl https://YOUR_HTTP_TRIGGER_URL/api/funqyHello
```

The output should be the same as what you observed earlier:

Output

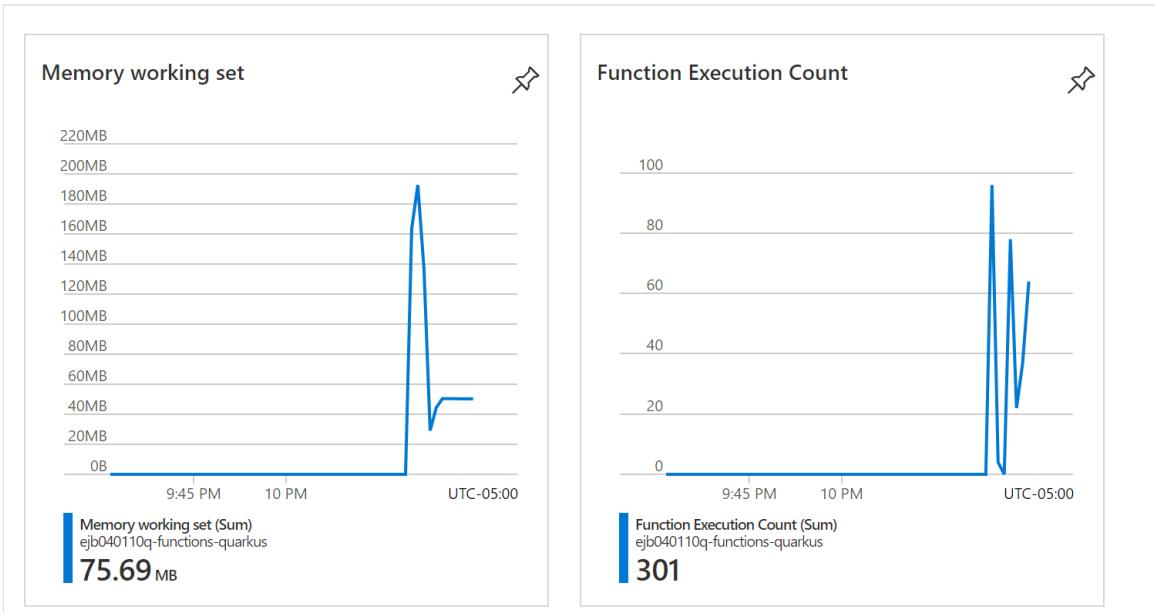
```
"hello funqy"
```

If you want to exercise the basic metrics capability in the Azure portal, try invoking the function within a shell `for` loop:

Bash

```
for i in {1..100}; do curl -d '"Dan on Azure"' -X POST  
https://YOUR_HTTP_TRIGGER_URL/api/greeting; done
```

After a while, you'll see some metrics data in the portal.



Now that you've opened your Azure function in the portal, here are more features that you can access from the portal:

- Monitor the performance of your Azure function. For more information, see [Monitoring Azure Functions](#).
- Explore telemetry. For more information, see [Analyze Azure Functions telemetry in Application Insights](#).
- Set up logging. For more information, see [Enable streaming execution logs in Azure Functions](#).

Clean up resources

If you don't need these resources, you can delete them by running the following command in Azure Cloud Shell or on your local terminal:

Azure CLI

```
az group delete --name <yourResourceGroupName> --yes
```

Next steps

In this article, you learned how to:

- ✓ Run Quarkus dev mode.
- ✓ Deploy a Funqy app to Azure functions by using `azure-functions-maven-plugin`.
- ✓ Examine the performance of the function in the portal.

To learn more about Azure Functions and Quarkus, see the following articles and references:

- [Azure Functions Java developer guide](#)
- [Quickstart: Create a Java function in Azure using Visual Studio Code](#)
- [Azure Functions documentation](#)
- [Quarkus guide to deploying on Azure ↗](#)

Spring Cloud Function in Azure

Article • 12/06/2023

This article guides you through using [Spring Cloud Functions](#) to develop a Java function and publish it to Azure Functions. When you're done, your function code runs on the [Consumption Plan](#) in Azure and can be triggered using an HTTP request.

Prerequisites

- An Azure subscription. If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

To develop functions using Java, you must have the following installed:

- [Java Developer Kit](#), version 11
- [Apache Maven](#), version 3.0 or higher
- [Azure CLI](#)
- [Azure Functions Core Tools](#) version 4

ⓘ Important

1. You must set the `JAVA_HOME` environment variable to the install location of the JDK to complete this quickstart.
2. Make sure your core tools version is at least 4.0.5455.

What we're going to build

We're going to build a classical "Hello, World" function that runs on Azure Functions and is configured with Spring Cloud Function.

The function receives a `User` JSON object, which contains a user name, and sends back a `Greeting` object, which contains the welcome message to that user.

The project is available in the [Spring Cloud Function in Azure](#) sample of the [azure-function-java-worker](#) repository on GitHub. You can use that sample directly if you want to see the final work described in this quickstart.

Create a new Maven project

We're going to create an empty Maven project, and configure it with Spring Cloud Function and Azure Functions.

In an empty folder, create a new *pom.xml* file and copy/paste the content from the sample project's [pom.xml](#) file.

ⓘ Note

This file uses Maven dependencies from both Spring Boot and Spring Cloud Function, and it configures the Spring Boot and Azure Functions Maven plugins.

You need to customize a few properties for your application:

- `<functionAppName>` is the name of your Azure Function
- `<functionAppRegion>` is the name of the Azure region where your Function is deployed
- `<functionResourceGroup>` is the name of the Azure resource group you're using

Change those properties directly near the top of the *pom.xml* file, as shown in the following example:

XML

```
<properties>
    <java.version>11</java.version>

    <!-- Spring Boot start class. WARNING: correct class must be set -->
    <start-class>com.example.DemoApplication</start-class>

    <!-- customize those properties. WARNING: the functionAppName should
be unique across Azure -->

<azure.functions.maven.plugin.version>1.29.0</azure.functions.maven.plugin.v
ersion>
    <functionResourceGroup>my-spring-function-resource-
group</functionResourceGroup>
    <functionAppServicePlanName>my-spring-function-service-
plan</functionAppServicePlanName>
    <functionAppName>my-spring-function</functionAppName>
    <functionPricingTier>Y1</functionPricingTier>
    <functionAppRegion>eastus</functionAppRegion>
</properties>
```

Create Azure configuration files

Create a `src/main/resources` folder and add the following Azure Functions configuration files to it.

`host.json`:

```
JSON

{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.2.0)"
  },
  "functionTimeout": "00:10:00"
}
```

`local.settings.json`:

```
JSON

{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "java",
    "FUNCTIONS_EXTENSION_VERSION": "~4",
    "AzureWebJobsDashboard": ""
  }
}
```

Create domain objects

Azure Functions can receive and send objects in JSON format. We're now going to create our `User` and `Greeting` objects, which represent our domain model. You can create more complex objects, with more properties, if you want to customize this quickstart and make it more interesting for you.

Create a `src/main/java/com/example/model` folder and add the following two files:

`User.java`:

```
Java

package com.example.model;

public class User {
```

```
private String name;

public User() {
}

public User(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}
```

Greeting.java:

Java

```
package com.example.model;

public class Greeting {

    private String message;

    public Greeting() {
    }

    public Greeting(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Create the Spring Boot application

This application manages all business logic, and has access to the full Spring Boot ecosystem. This capability gives you two main benefits over a standard Azure Function:

- It doesn't rely on the Azure Functions APIs, so you can easily port it to other systems. For example, you can reuse it in a normal Spring Boot application.
- You can use all the `@Enable` annotations from Spring Boot to add new features.

In the `src/main/java/com/example` folder, create the following file, which is a normal Spring Boot application:

DemoApplication.java:

Java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Now create the following file in the `src/main/java/com/example/hello` folder. This code contains a Spring Boot component that represents the Function we want to run:

Hello.java:

Java

```
package com.example.hello;

import com.example.model.*;
import org.springframework.stereotype.Component;
import java.util.function.Function;

@Component
public class Hello implements Function<User, Greeting> {

    @Override
    public Greeting apply(User user) {
        return new Greeting("Hello, " + user.getName() + "!\n");
    }
}
```

① Note

The `Hello` function is quite specific:

- It is a `java.util.function.Function`. It contains the business logic, and it uses a standard Java API to transform one object into another.
- Because it has the `@Component` annotation, it's a Spring Bean, and by default its name is the same as the class, but starting with a lowercase character: `hello`. Following this naming convention is important if you want to create other functions in your application. The name must match the Azure Functions name we'll create in the next section.

Create the Azure Function

To benefit from the full Azure Functions API, we now code an Azure Function that delegates its execution to the Spring Cloud Function created in the previous step.

In the `src/main/java/com/example/hello` folder, create the following Azure Function class file:

`HelloHandler.java`:

Java

```
package com.example.hello;

import com.microsoft.azure.functions.*;
import com.microsoft.azure.functions.annotation.AuthorizationLevel;
import com.microsoft.azure.functions.annotation.FunctionName;
import com.microsoft.azure.functions.annotation.HttpTrigger;
import com.example.model.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Optional;

@Component
public class HelloHandler {

    @Autowired
    private Hello hello;

    @FunctionName("hello")
    public HttpResponseMessage execute(
        @HttpTrigger(name = "request", methods = { HttpMethod.GET,
        HttpMethod.POST }, authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<User>> request, ExecutionContext context) {
        User user = request.getBody()
            .filter(u -> u.getName() != null)
            .orElseGet(() -> new
User(request.getQueryParameters().getOrDefault("name", "world")));
    }
}
```

```
        context.getLogger().info("Greeting user name: " + user.getName());
        return request.createResponseBuilder(HttpStatus.OK)
            .body(hello.apply(user))
            .header("Content-Type", "application/json")
            .build();
    }
}
```

This Java class is an Azure Function, with the following interesting features:

- The class has the `@Component` annotation, so it's a Spring Bean.
- The name of the function, as defined by the `@FunctionName("hello")` annotation, is `hello`.
- The class implements a real Azure Function, so you can use the full Azure Functions API here.

Add unit tests

This step is optional but recommended to validate that the application works correctly.

Create a `src/test/java/com/example` folder and add the following JUnit tests:

HelloTest.java:

```
Java

package com.example;

import com.example.hello.Hello;
import com.example.model.Greeting;
import com.example.model.User;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class HelloTest {

    @Test
    public void test() {
        Greeting result = new Hello().apply(new User("foo"));
        assertThat(result.getMessage()).isEqualTo("Hello, foo!\n");
    }
}
```

You can now test your Azure Function using Maven:

```
Bash
```

```
mvn clean test
```

Run the Function locally

Before you deploy your application to Azure Function, let's first test it locally.

First you need to package your application into a Jar file:

Bash

```
mvn package
```

Now that the application is packaged, you can run it using the `azure-functions` Maven plugin:

Bash

```
mvn azure-functions:run
```

The Azure Function should now be available on your localhost, using port 7071. You can test the function by sending it a POST request, with a `User` object in JSON format. For example, using cURL:

Bash

```
curl -X POST http://localhost:7071/api/hello -d "{\"name\":\"Azure\"}"
```

The Function should answer you with a `Greeting` object, still in JSON format:

Output

```
{
  "message": "Hello, Azure!\n"
}
```

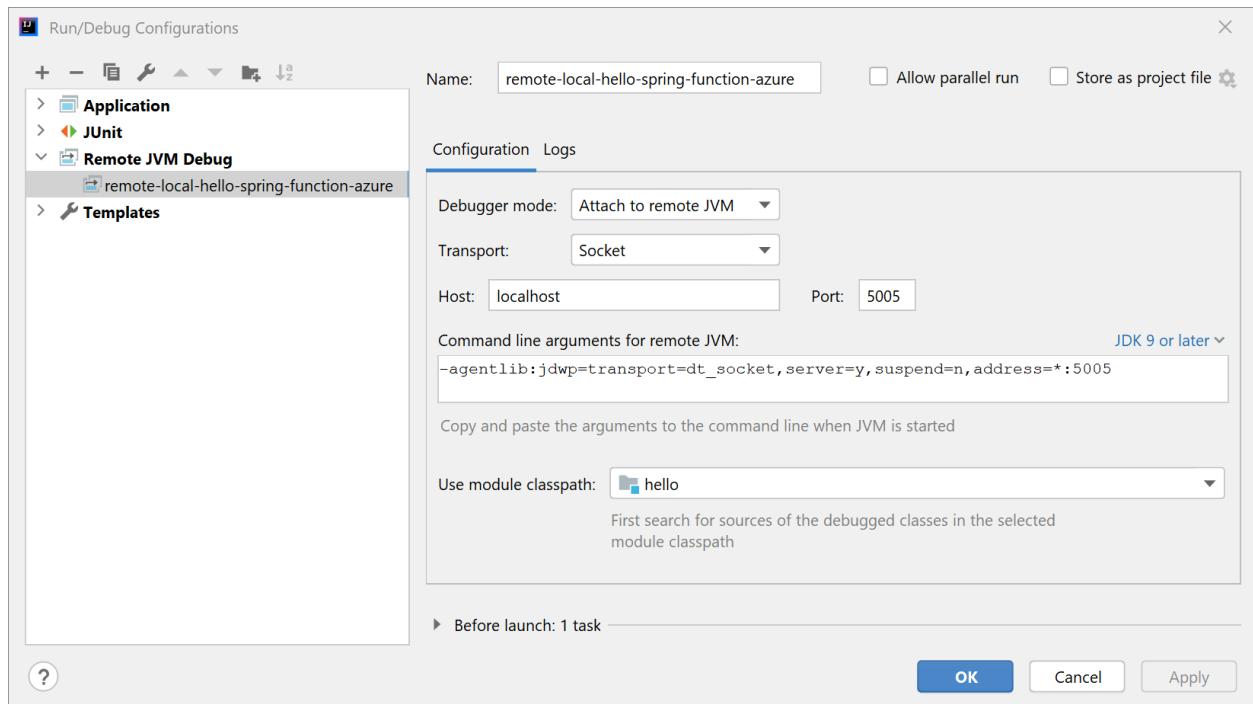
Here's a screenshot of the cURL request on the top of the screen, and the local Azure Function at the bottom:

Debug the Function locally

The following sections describe how to debug the function.

Debug using IntelliJ IDEA

Open the project in IntelliJ IDEA, then create a **Remote JVM Debug** run configuration to attach. For more information, see [Tutorial: Remote debug ↗](#).



Run the application with the following command:

```
Bash
mvn azure-functions:run -DenableDebug
```

When the application starts, you see the following output:

```
Output
Worker process started and initialized.
Listening for transport dt_socket at address: 5005
```

Start project debugging in IntelliJ IDEA. You see the following output:

```
Output
Connected to the target VM, address: 'localhost:5005', transport: 'socket'
```

Mark the breakpoints you want to debug. The IntelliJ IDEA will enter debugging mode after sending a request.

Debug using Visual Studio Code

Open the project in Visual Studio Code, then configure the following *launch.json* file content:

```
JSON
```

```
{  
    "version": "0.2.0",  
    "configurations": [  
        {  
            "type": "java",  
            "name": "Attach to Remote Program",  
            "request": "attach",  
            "hostName": "127.0.0.1",  
            "port": 5005  
        }  
    ]  
}
```

Run the application with the following command:

Bash

```
mvn azure-functions:run -DenableDebug
```

When the application starts, you see the following output:

Output

```
Worker process started and initialized.  
Listening for transport dt_socket at address: 5005
```

Start project debugging in Visual Studio Code, then mark the breakpoints you want to debug. Visual Studio Code will enter debugging mode after sending a request. For more information, see [Running and debugging Java](#).

Deploy the Function to Azure Functions

Now, you're going to publish the Azure Function to production. Remember that the `<functionAppName>`, `<functionAppRegion>`, and `<functionResourceGroup>` properties you've defined in your `pom.xml` file are used to configure your function.

ⓘ Note

The Maven plugin needs to authenticate with Azure. If you have Azure CLI installed, use `az login` before continuing. For more authentication options, see [Authentication](#) in the [azure-maven-plugins](#) repository.

Run Maven to deploy your function automatically:

Bash

```
mvn azure-functions:deploy
```

Now go to the [Azure portal](#) to find the `Function App` that has been created.

Select the function:

- In the function overview, note the function's URL.
- To check your running function, select **Log streaming** on the navigation menu.

Now, as you did in the previous section, use cURL to access the running function, as shown in the following example. Be sure to replace `your-function-name` by your real function name.

Bash

```
curl https://your-function-name.azurewebsites.net/api/hello -d ""  
{"name": "Azure"}
```

Like in the previous section, the Function should answer you with a `Greeting` object, still in JSON format:

Output

```
{  
  "message": "Hello, Azure!\n"  
}
```

Congratulations, you have a Spring Cloud Function running on Azure Functions! For more information and samples of Spring Cloud functions, see the following resources:

- [Spring cloud function blog](#)
- [Spring cloud function reference documents](#)
- [Spring cloud function samples](#)

Next steps

To learn more about Spring and Azure, continue to the Spring on Azure documentation center.

[Spring on Azure](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Quickstart: Create a Java function in Azure using Visual Studio Code

Article • 07/18/2024

In this article, you use Visual Studio Code to create a Java function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

If Visual Studio Code isn't your preferred development tool, check out our similar tutorials for Java developers:

- [Gradle](#)
- [IntelliJ IDEA](#)
- [Maven](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Configure your environment

Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- The [Java Development Kit](#), version 8, 11, 17 or 21(Linux).
- [Apache Maven](#), version 3.0 or above.
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [Java extension pack](#)
- The [Azure Functions extension](#) for Visual Studio Code.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in Java. Later in this article, you'll publish your function code to Azure.

1. In Visual Studio Code, press **F1** to open the command palette and search for and run the command **Azure Functions: Create New Project...**.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[] [Expand table](#)

Prompt	Selection
Select a language	Choose <code>Java</code> .
Select a version of Java	Choose <code>Java 8</code> , <code>Java 11</code> , <code>Java 17</code> or <code>Java 21</code> , the Java version on which your functions run in Azure. Choose a Java version that you've verified locally.
Provide a group ID	Choose <code>com.function</code> .
Provide an artifact ID	Choose <code>myFunction</code> .
Provide a version	Choose <code>1.0-SNAPSHOT</code> .
Provide a package name	Choose <code>com.function</code> .
Provide an app name	Choose <code>myFunction-12345</code> .
Select a template for your project's first function	Choose <code>HTTP trigger</code> .
Select the build tool for Java project	Choose <code>Maven</code> .

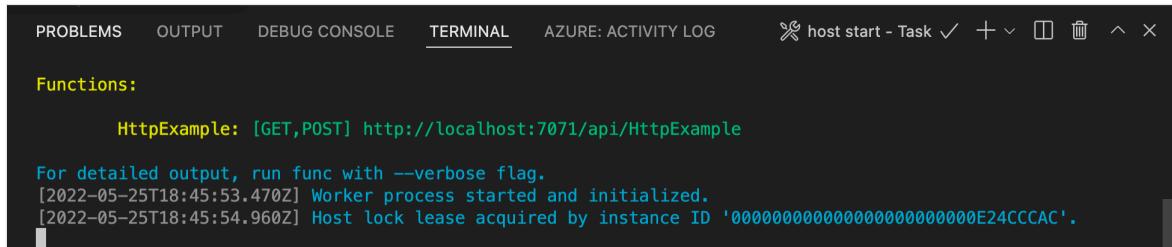
Prompt	Selection
Provide a function name	Enter <code>HttpExample</code> .
Authorization level	Choose <code>Anonymous</code> , which lets anyone call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Choose <code>Open in current window</code> .

4. Visual Studio Code uses the provided information and generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. For more information about the files that are created, see [Generated project files](#).

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

- To start the function locally, press `F5` or the Run and Debug icon in the left-hand side Activity bar. The Terminal panel displays the Output from Core Tools. Your app starts in the Terminal panel. You can see the URL endpoint of your HTTP-triggered function running locally.



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG
✖ host start - Task ✓ + ▾ ⌫ ⌄ ×

Functions:

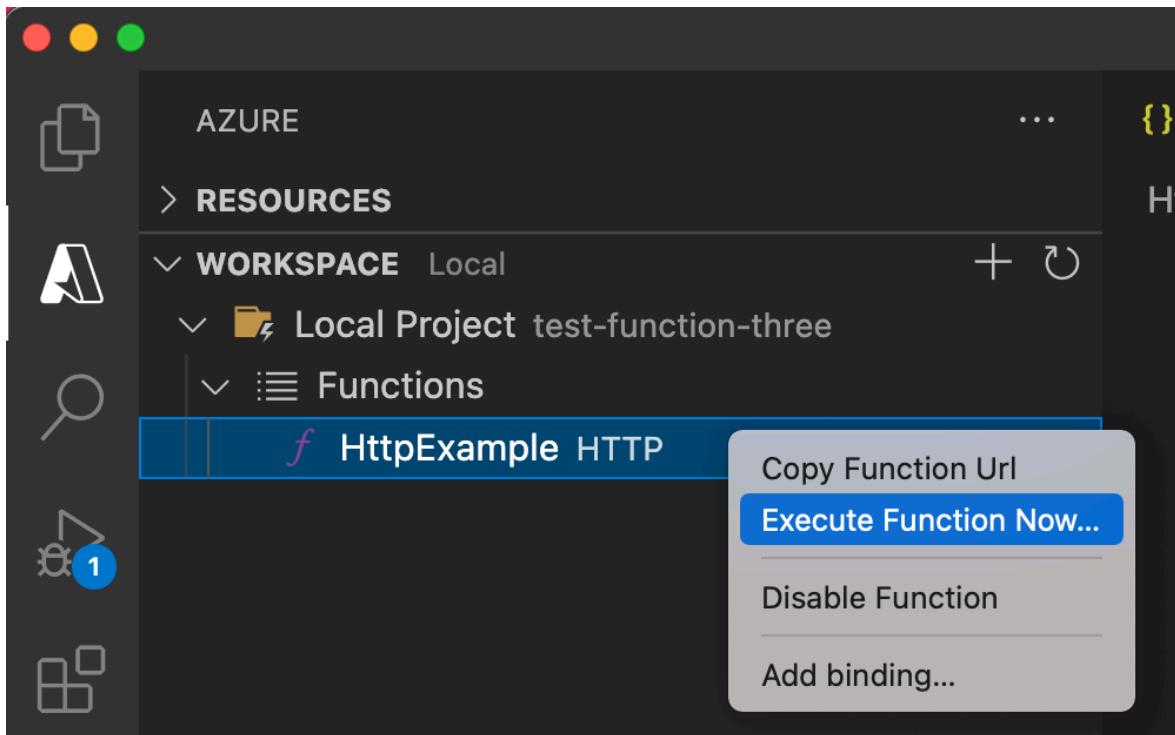
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.

```

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

- With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or `ctrl -` click (macOS) the new function and choose **Execute Function Now....**



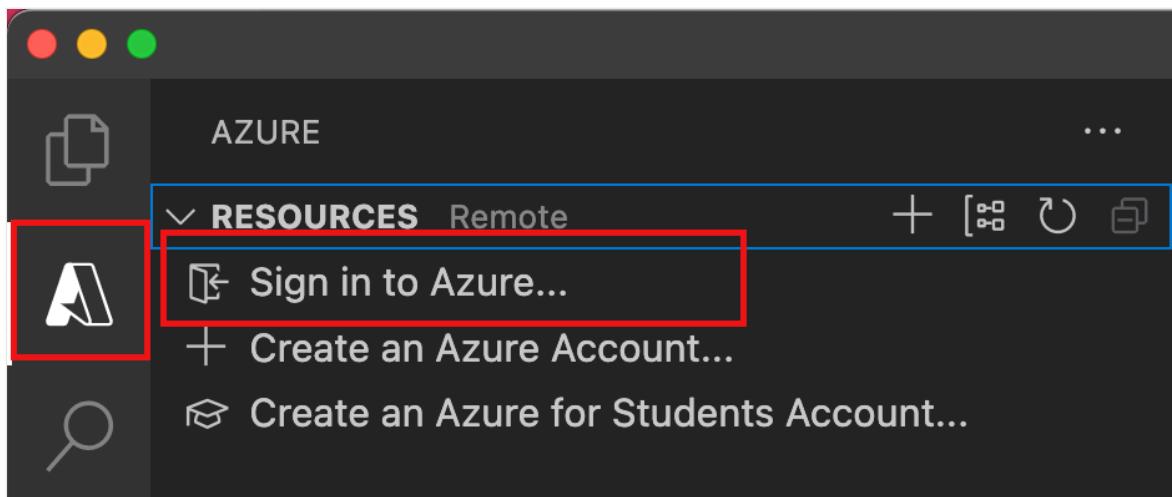
3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.
5. With the **Terminal** panel focused, press `Ctrl + C` to stop Core Tools and disconnect the debugger.

After you've verified that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

2. When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors.

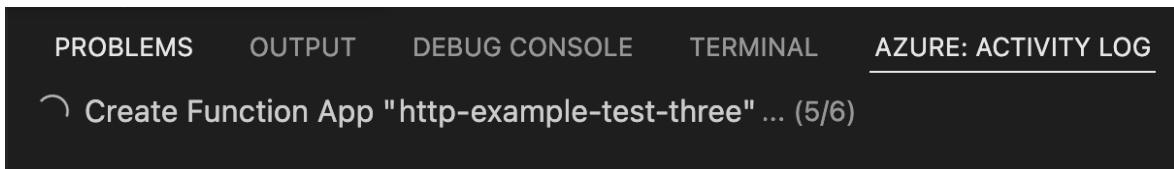
1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
2. At the prompts, provide the following information:

[] [Expand table](#)

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.

Prompt	Action
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region near you .

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

💡 Tip

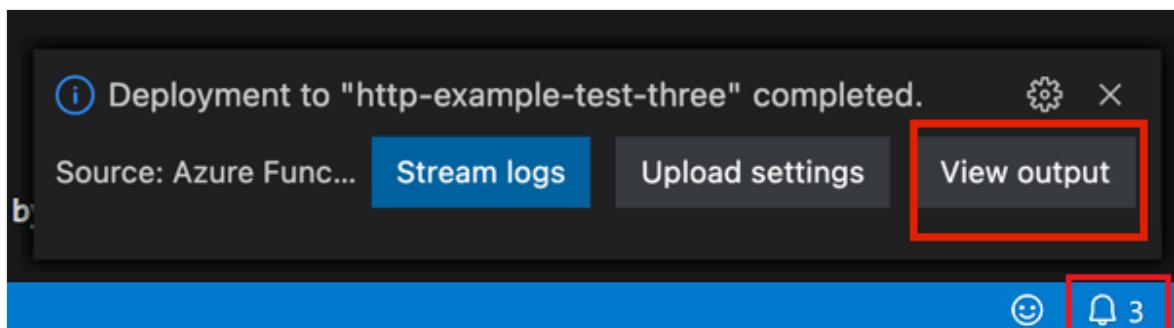
By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

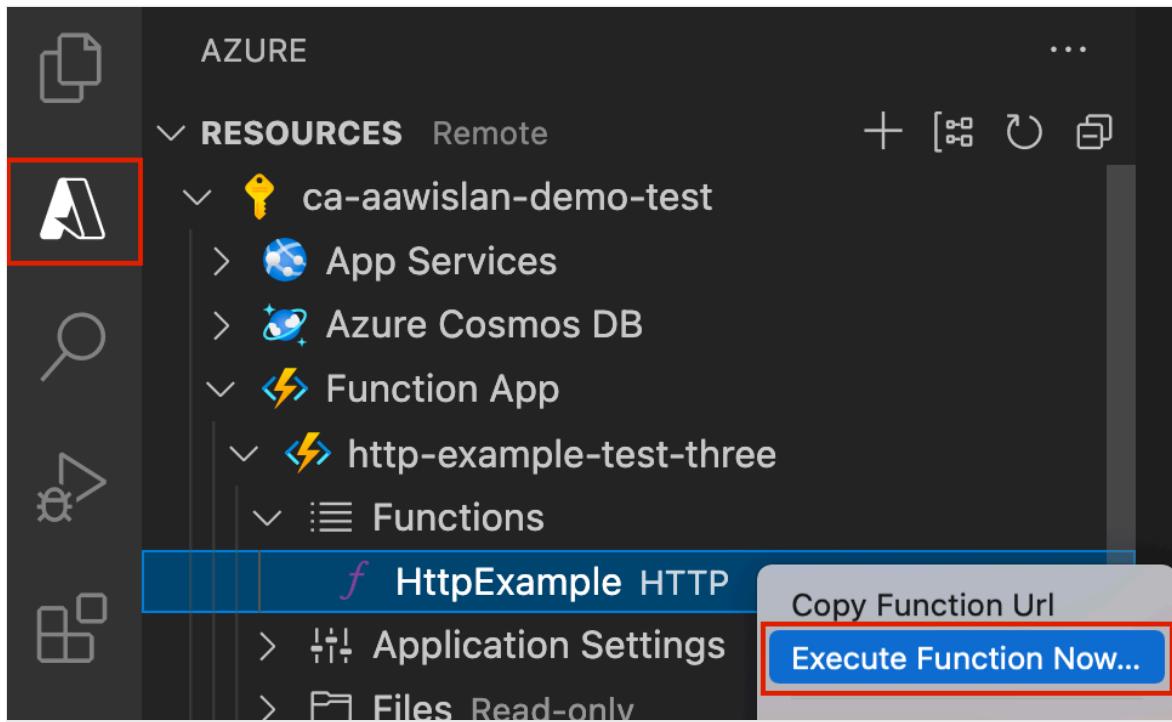
Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Back in the **Resources** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click (Windows) or **Ctrl** - click (macOS) the `HttpExample` function and choose **Execute Function Now....**



2. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
3. When the function executes in Azure and returns a response, a notification is raised in Visual Studio Code.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal's 'Overview' page for a resource group. The left sidebar lists various options: Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (selected), Functions, App keys, App files, and Proxies. The main area shows a card for 'Resource group (change) myResourceGroup'. The card includes fields for Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), and Tags (with a link to 'Click here to add tags'). Below the card, there are navigation tabs: Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You have used [Visual Studio Code](#) to create a function app with a simple HTTP-triggered function. In the next article, you expand that function by connecting to Azure Storage. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to an Azure Storage queue](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | Get help at Microsoft Q&A

Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI

Article • 09/09/2024

In this Quickstart, you use Azure Developer command-line tools to create functions that respond to HTTP requests. After testing the code locally, you deploy it to a new serverless function app you create running in a Flex Consumption plan in Azure Functions.

The project source uses the Azure Developer CLI (azd) to simplify deploying your code to Azure. This deployment follows current best practices for secure and scalable Azure Functions deployments.

Important

The [Flex Consumption plan](#) is currently in preview.

By default, the Flex Consumption plan follows a *pay-for-what-you-use* billing model, which means to complete this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Azure Developer CLI](#).
- [Azure Functions Core Tools](#).
- [Java 17 Developer Kit](#)
 - If you use another [supported version of Java](#), you must update the project's pom.xml file.
 - The `JAVA_HOME` environment variable must be set to the install location of the correct version of the JDK.
- [Apache Maven 3.8.x ↗](#)
- A [secure HTTP test tool](#) for sending requests with JSON payloads to your function endpoints. This article uses `curl`.

Initialize the project

You can use the `azd init` command to create a local Azure Functions code project from a template.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

```
Console
```

```
azd init --template azure-functions-java-flex-consumption-azd -e flexquickstart-java
```

This command pulls the project files from the [template repository](#) and initializes the project in the current folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Run this command to navigate to the `http` app folder:

```
Console
```

```
cd http
```

3. Create a file named `local.settings.json` in the `http` folder that contains this JSON data:

```
JSON
```

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "java"  
    }  
}
```

This file is required when running locally.

Run in your local environment

1. Run this command from your app folder in a terminal or command prompt:

Console

```
mvn clean package  
mvn azure-functions:run
```

When the Functions host starts in your local project folder, it writes the URL endpoints of your HTTP triggered functions to the terminal output.

2. In your browser, navigate to the `httpget` endpoint, which should look like this URL:

<http://localhost:7071/api/httpget>

3. From a new terminal or command prompt window, run this `curl` command to send a POST request with a JSON payload to the `httppost` endpoint:

Console

```
curl -i http://localhost:7071/api/httppost -H "Content-Type: text/json"  
-d @testdata.json
```

This command reads JSON payload data from the `testdata.json` project file. You can find examples of both HTTP requests in the `test.http` project file.

4. When you're done, press **Ctrl+C** in the terminal window to stop the `func.exe` host process.

Review the code (optional)

You can review the code that defines the two HTTP trigger function endpoints:

httpget

Java

```
@FunctionName("httpget")  
public HttpResponseMessage run(  
    @HttpTrigger(  
        name = "req",  
        methods = {HttpMethod.GET},  
        authLevel = AuthorizationLevel.FUNCTION)  
    HttpRequestMessage<Optional<String>> request,  
    final ExecutionContext context) {  
    context.getLogger().info("Java HTTP trigger processed a request.");  
  
    // Parse query parameter
```

```
String name =  
Optional.ofNullable(request.getQueryParameters().get("name")).orElse("Wo  
rld");  
  
return request.createResponseBuilder(HttpStatus.OK).body("Hello, " +  
name).build();  
}
```

You can review the complete template project [here](#).

After you verify your functions locally, it's time to publish them to Azure.

Create Azure resources

This project is configured to use the `azd provision` command to create a function app in a Flex Consumption plan, along with other required Azure resources.

ⓘ Note

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

The `azd up` and `azd deploy` commands aren't currently supported for Java apps.

1. In the root folder of the project, run this command to create the required Azure resources:

Console

```
azd provision
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

[+] [Expand table](#)

Parameter	Description
<i>Azure subscription</i>	Subscription in which your resources are created.
<i>Azure location</i>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd provision` command uses your response to these prompts with the Bicep configuration files to create and configure these required Azure resources:

- Flex Consumption plan and function app
- Azure Storage (required) and Application Insights (recommended)
- Access policies and roles for your account
- Service-to-service connections using managed identities (instead of stored connection strings)
- Virtual network to securely run both the function app and the other Azure resources

After the command completes successfully, you can deploy your project code to this new function app in Azure.

Deploy to Azure

You can use Core Tools to package your code and deploy it to Azure from the `target` output folder.

1. Navigate to the app folder equivalent in the `target` output folder:

```
Console
cd http/target/azure-functions/contoso-functions
```

This folder should have a `host.json` file, which indicates that it's the root of your compiled Java function app.

2. Run these commands to deploy your compiled Java code project to the new function app resource in Azure using Core Tools:

```
bash
```

```
Bash
```

```
APP_NAME=$(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp publish $APP_NAME
```

The `azd env get-value` command gets your function app name from the local environment, which is required for deployment using `func azure functionapp publish`. After publishing completes successfully, you see links to the HTTP trigger endpoints in Azure.

Invoke the function on Azure

You can now invoke your function endpoints in Azure by making HTTP requests to their URLs using your HTTP test tool or from the browser (for GET requests). When your functions run in Azure, access key authorization is enforced, and you must provide a function access key with your request.

You can use the Core Tools to obtain the URL endpoints of your functions running in Azure.

1. In your local terminal or command prompt, run these commands to get the URL endpoint values:

```
bash
```

```
Bash
```

```
SET APP_NAME=(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp list-functions $APP_NAME --show-keys
```

The `azd env get-value` command gets your function app name from the local environment. Using the `--show-keys` option with `func azure functionapp list-functions` means that the returned **Invoke URL:** value for each endpoint includes a function-level access key.

2. As before, use your HTTP test tool to validate these URLs in your function app running in Azure.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs:

```
Console  
azd down --no-prompt
```

ⓘ Note

The `--no-prompt` option instructs `azd` to delete your resource group without a confirmation from you.

This command doesn't affect your local code project.

Related content

- [Flex Consumption plan](#)
- [Azure Developer CLI \(azd\)](#)
- [azd reference](#)
- [Azure Functions Core Tools reference](#)
- [Code and test Azure Functions locally](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a JavaScript function in Azure using Visual Studio Code

Article • 07/18/2024

Use Visual Studio Code to create a JavaScript function that responds to HTTP requests. Test the code locally, then deploy it to the serverless environment of Azure Functions.

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of the page. The v4 model is generally available and is designed to have a more flexible and intuitive experience for JavaScript and TypeScript developers. Learn more about the differences between v3 and v4 in the [migration guide](#).

Completion of this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [CLI-based version](#) of this article.

Configure your environment

Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js 18.x](#) or above. Use the `node --version` command to check your version.
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [Azure Functions extension v1.10.4](#) or above for Visual Studio Code. This extension installs [Azure Functions Core Tools](#) for you the first time you locally run your functions. Node.js v4 requires version 4.0.5382, or a later version of Core Tools.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code

using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in JavaScript. Later in this article, you publish your function code to Azure.

1. In Visual Studio Code, press **F1** to open the command palette and search for and run the command **Azure Functions: Create New Project....**
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[+] Expand table

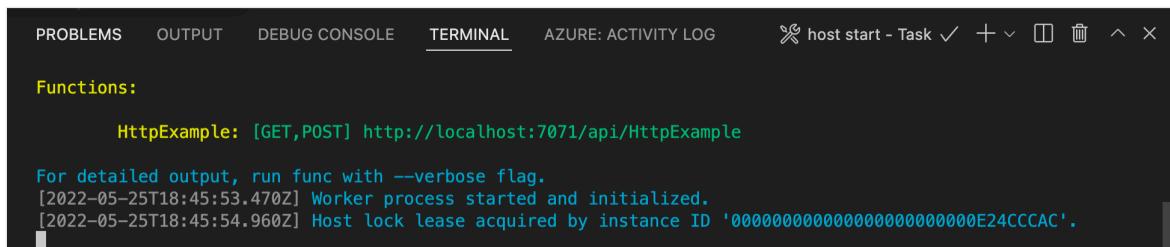
Prompt	Selection
Select a language for your function project	Choose JavaScript .
Select a JavaScript programming model	Choose Model V4
Select a template for your project's first function	Choose HTTP trigger .
Provide a function name	Type HttpExample .
Select how you would like to open your project	Choose Open in current window

Using this information, Visual Studio Code generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. To learn more about files that are created, see [Azure Functions JavaScript developer guide](#).

Run the function locally

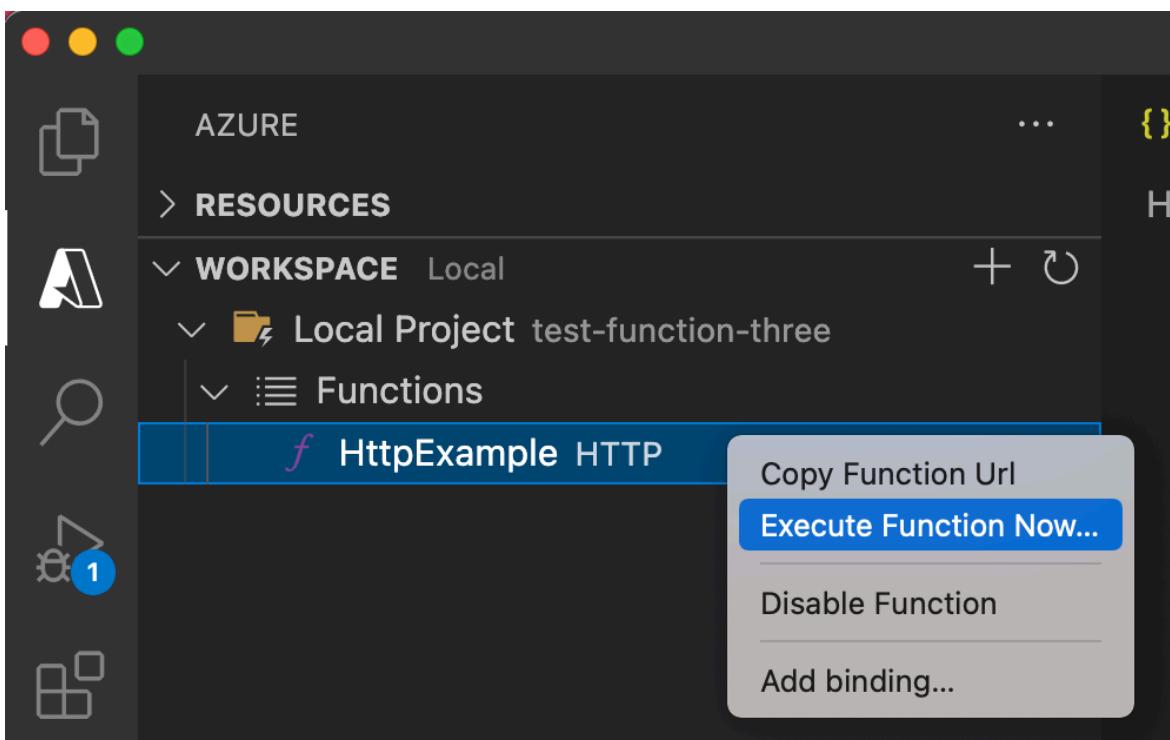
Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

1. To start the function locally, press **F5** or the **Run and Debug** icon in the left-hand side Activity bar. The **Terminal** panel displays the Output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

2. With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or **Ctrl + click** (macOS) the new function and choose **Execute Function Now...**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
 4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.

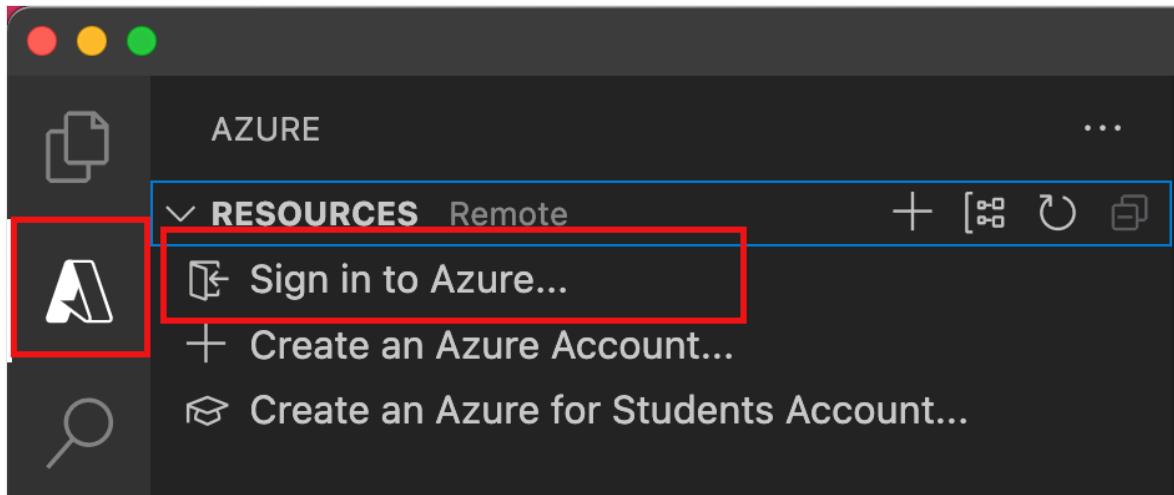
- With the **Terminal** panel focused, press `Ctrl + C` to stop Core Tools and disconnect the debugger.

After you've verified that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

- If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

- When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
- After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors.

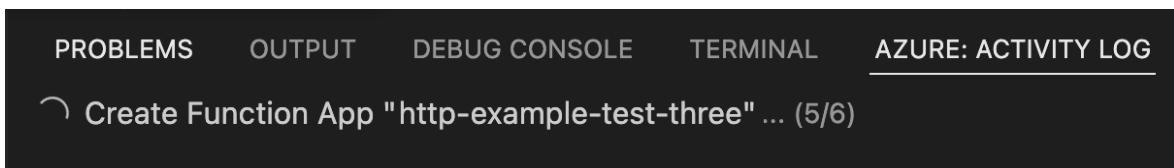
1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.

2. At the prompts, provide the following information:

[+] [Expand table](#)

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region ↗ near you.

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

💡 Tip

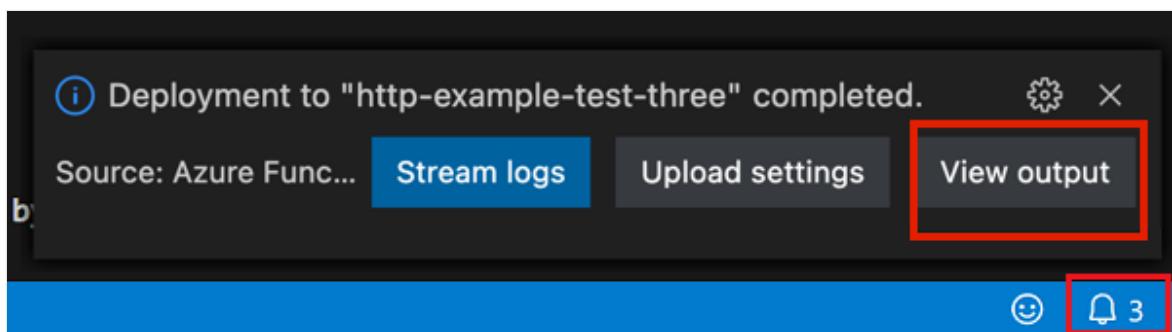
By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

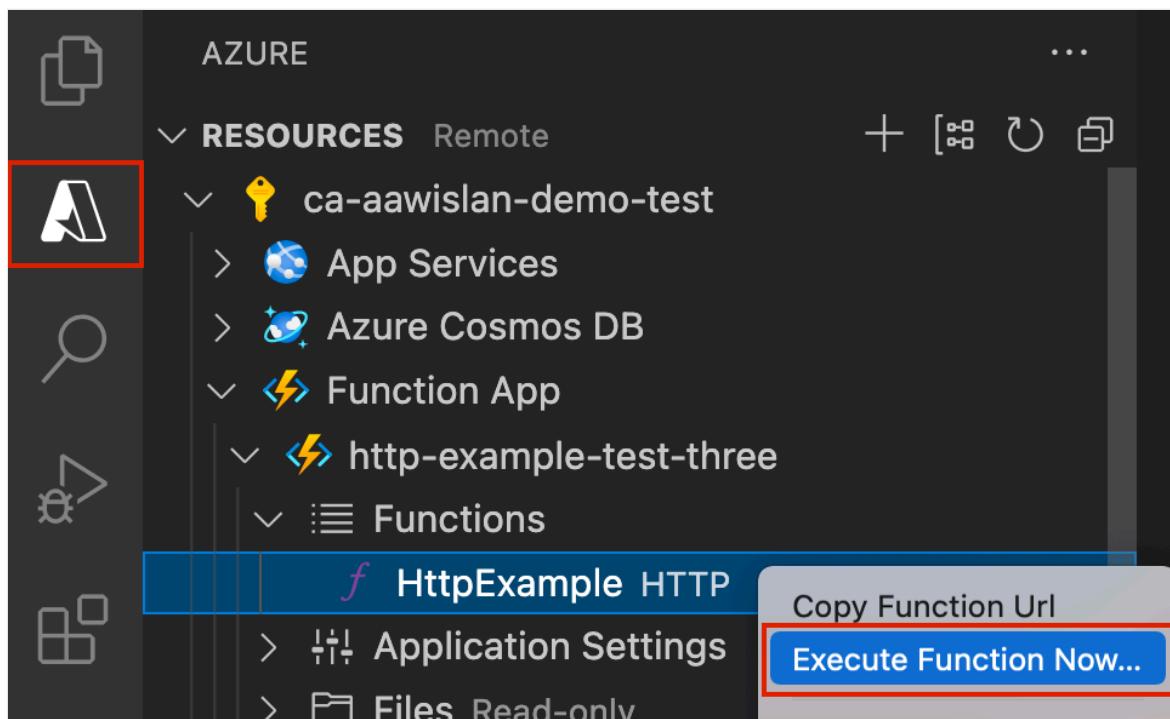
Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Back in the **Resources** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click (Windows) or **Ctrl - click** (macOS) the `HttpExample` function and choose **Execute Function Now....**



2. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
3. When the function executes in Azure and returns a response, a notification is raised in Visual Studio Code.

Troubleshooting

Use the following table to resolve the most common issues encountered when using this quickstart.

[] [Expand table](#)

Problem	Solution
Can't create a local function project?	Make sure you have the Azure Functions extension installed.
Can't run the function locally?	Make sure you have the latest version of Azure Functions Core Tools installed . When running on Windows, make sure that the default terminal shell for Visual Studio Code isn't set to WSL Bash.

Problem	Solution
Can't deploy function to Azure?	Review the Output for error information. The bell icon in the lower right corner is another way to view the output. Did you publish to an existing function app? That action overwrites the content of that app in Azure.
Couldn't run the cloud-based Function app?	Remember to use the query string to send in parameters.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, select the Azure icon to open the Azure explorer.
2. In the Resource Groups section, find your resource group.
3. Right-click the resource group and select **Delete**.

To learn more about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You have used [Visual Studio Code](#) to create a function app with a simple HTTP-triggered function.

[Learn more about JavaScript functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a JavaScript function in Azure from the command line

Article • 02/26/2024

In this article, you use command-line tools to create a JavaScript function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of the page. The v4 model is generally available and is designed to have a more flexible and intuitive experience for JavaScript and TypeScript developers. Learn more about the differences between v3 and v4 in the [migration guide](#).

Completion of this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [Visual Studio Code-based version](#) of this article.

Configure your local environment

Before you begin, you must have the following prerequisites:

- An Azure account with an active subscription. [Create an account for free](#).
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - The Azure [Az PowerShell module](#) version 5.9.0 or later.
- [Node.js](#) version 18 or above.

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. Visual Studio Code debugging requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

- Make sure you install version v4.0.5382 of the Core Tools, or a later version.

Create a local function project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single function.

1. In a suitable folder, run the `func init` command, as follows, to create a JavaScript Node.js v4 project in the current folder:

```
Console  
func init --javascript
```

This folder now contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file. Required npm packages are also installed in `node_modules`.

1. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (`HttpExample`) and the `--template` argument specifies the function's trigger (HTTP).

```
Console
```

```
func new --name HttpExample --template "HTTP trigger" --authlevel  
"anonymous"

[`func new`](functions-core-tools-reference.md#func-new) creates a file  
named *HttpExample.js* in the *src/functions* directory, which contains  
your function's code.
```

2. Add Azure Storage connection information in *local.settings.json*.

JSON

```
{  
  "Values": {  
    "AzureWebJobsStorage": "<Azure Storage connection  
information>",  
    "FUNCTIONS_WORKER_RUNTIME": "node"  
  }  
}
```

3. (Optional) If you want to learn more about a particular function, say HTTP trigger, you can run the following command:

Console

```
func help httptrigger
```

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

Console

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:    4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the following commands to create these items. Both Azure CLI and PowerShell are supported.

1. If you haven't done so already, sign in to Azure:

Azure CLI

Azure CLI

```
az login
```

The `az login` command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in your chosen region:

Azure CLI

Azure CLI

```
az group create --name AzureFunctionsQuickstart-rg --location  
<REGION>
```

The `az group create` command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the `az account list-locations` command.

3. Create a general-purpose storage account in your resource group and region:

Azure CLI

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location <REGION>  
--resource-group AzureFunctionsQuickstart-rg --sku Standard_LRS --  
allow-blob-public-access false
```

The `az storage account create` command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#).

ⓘ Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the

storage account.

4. Create the function app in Azure:

```
Azure CLI
```

```
Azure CLI
```

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg  
--consumption-plan-location <REGION> --runtime node --runtime-  
version 18 --functions-version 4 --name <APP_NAME> --storage-  
account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure. It's recommended that you use the latest LTS version of Node.js, which is currently 18. You can specify the version by setting `--runtime-version` to `18`.

In the previous example, replace `<STORAGE_NAME>` with the name of the account you used in the previous step, and replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default DNS domain for the function app.

This command creates a function app running in your specified language runtime under the [Azure Functions Consumption Plan](#), which is free for the amount of usage you incur here. The command also creates an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

Deploy the function project to Azure

After you've successfully created your function app in Azure, you're now ready to deploy your local functions project by using the `func azure functionapp publish` command.

In the following example, replace `<APP_NAME>` with the name of your app.

```
Console
```

```
func azure functionapp publish <APP_NAME>
```

The publish command shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

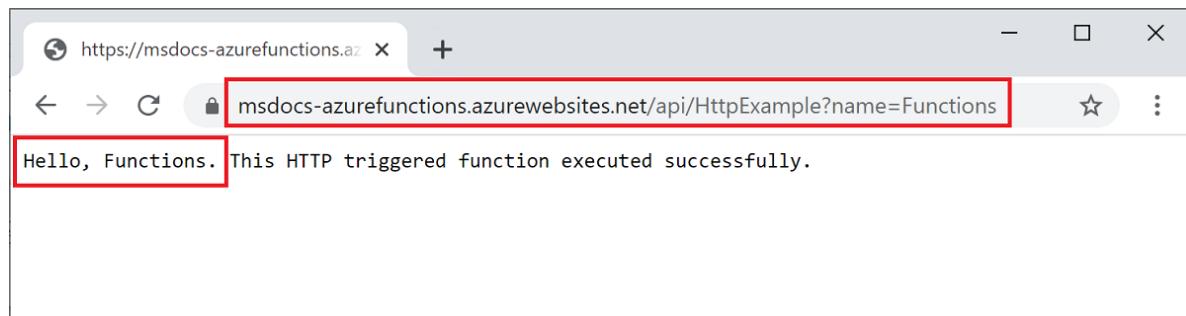
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
  HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-qs.azurewebsites.net/api/httpexample
```

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.



Run the following command to view near real-time streaming logs:

Console

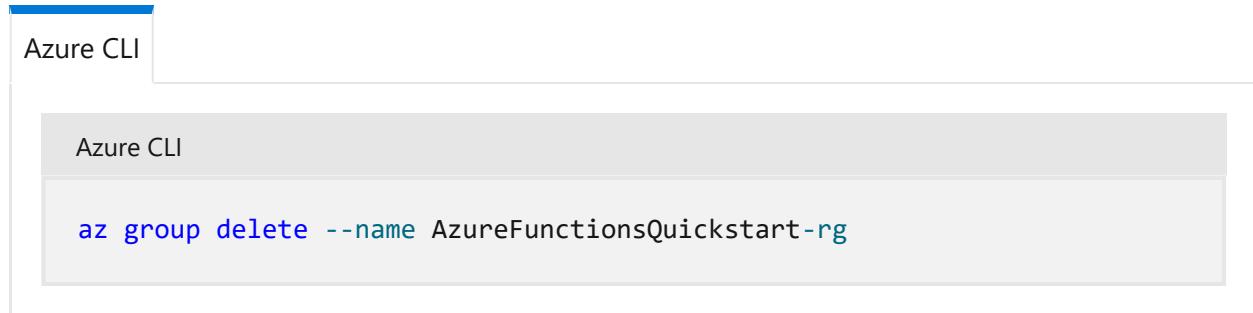
```
func azure functionapp logstream <APP_NAME>
```

In a separate terminal window or in the browser, call the remote function again. A verbose log of the function execution in Azure is shown in the terminal.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.



Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

[Learn more about JavaScript functions](#)

Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI

Article • 09/09/2024

In this Quickstart, you use Azure Developer command-line tools to create functions that respond to HTTP requests. After testing the code locally, you deploy it to a new serverless function app you create running in a Flex Consumption plan in Azure Functions.

The project source uses the Azure Developer CLI (azd) to simplify deploying your code to Azure. This deployment follows current best practices for secure and scalable Azure Functions deployments.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

By default, the Flex Consumption plan follows a *pay-for-what-you-use* billing model, which means to complete this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Azure Developer CLI](#).
- [Azure Functions Core Tools](#).
- [Node.js 20 ↗](#)
- A [secure HTTP test tool](#) for sending requests with JSON payloads to your function endpoints. This article uses `curl`.

Initialize the project

You can use the `azd init` command to create a local Azure Functions code project from a template.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

```
Console
```

```
azd init --template functions-quickstart-javascript-azd -e flexquickstart-js
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Create a file named `local.settings.json` in the root folder that contains this JSON data:

```
JSON
```

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "node"  
    }  
}
```

This file is required when running locally.

Run in your local environment

1. Run this command from your app folder in a terminal or command prompt:

```
Console
```

```
func start
```

When the Functions host starts in your local project folder, it writes the URL endpoints of your HTTP triggered functions to the terminal output.

2. In your browser, navigate to the `httpget` endpoint, which should look like this URL:

<http://localhost:7071/api/httpget>

3. From a new terminal or command prompt window, run this `curl` command to send a POST request with a JSON payload to the `httppost` endpoint:

```
Console
```

```
curl -i http://localhost:7071/api/httppost -H "Content-Type: text/json"  
-d @testdata.json
```

This command reads JSON payload data from the `testdata.json` project file. You can find examples of both HTTP requests in the `test.http` project file.

4. When you're done, press `Ctrl+C` in the terminal window to stop the `func.exe` host process.

Review the code (optional)

You can review the code that defines the two HTTP trigger function endpoints:

```
httpget
```

```
JavaScript
```

```
const { app } = require('@azure/functions');

app.http('httpGetFunction', {
    methods: ['GET'],
    authLevel: 'function',
    handler: async (request, context) => {
        context.log(`Http function processed request for url
        "${request.url}"`);

        const name = request.query.get('name') || await request.text()
        || 'world';

        return { body: `Hello, ${name}!` };
    }
});
```

You can review the complete template project [here ↗](#).

After you verify your functions locally, it's time to publish them to Azure.

Deploy to Azure

This project is configured to use the `azd up` command to deploy this project to a new function app in a Flex Consumption plan in Azure.

💡 Tip

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

1. Run this command to have `azd` create the required Azure resources in Azure and deploy your code project to the new function app:

Console

```
azd up
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

[\[+\] Expand table](#)

Parameter	Description
<code>Azure subscription</code>	Subscription in which your resources are created.
<code>Azure location</code>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd up` command uses your response to these prompts with the Bicep configuration files to complete these deployment tasks:

- Create and configure these required Azure resources (equivalent to `azd provision`):
 - Flex Consumption plan and function app
 - Azure Storage (required) and Application Insights (recommended)
 - Access policies and roles for your account
 - Service-to-service connections using managed identities (instead of stored connection strings)

- Virtual network to securely run both the function app and the other Azure resources
- Package and deploy your code to the deployment container (equivalent to `azd deploy`). The app is then started and runs in the deployed package.

After the command completes successfully, you see links to the resources you created.

Invoke the function on Azure

You can now invoke your function endpoints in Azure by making HTTP requests to their URLs using your HTTP test tool or from the browser (for GET requests). When your functions run in Azure, access key authorization is enforced, and you must provide a function access key with your request.

You can use the Core Tools to obtain the URL endpoints of your functions running in Azure.

1. In your local terminal or command prompt, run these commands to get the URL endpoint values:

```
bash
Bash
SET APP_NAME=(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp list-functions $APP_NAME --show-keys
```

The `azd env get-value` command gets your function app name from the local environment. Using the `--show-keys` option with `func azure functionapp list-functions` means that the returned **Invoke URL:** value for each endpoint includes a function-level access key.

2. As before, use your HTTP test tool to validate these URLs in your function app running in Azure.

Redeploy your code

You can run the `azd up` command as many times as you need to both provision your Azure resources and deploy code updates to your function app.

ⓘ Note

Deployed code files are always overwritten by the latest deployment package.

Your initial responses to `azd` prompts and any environment variables generated by `azd` are stored locally in your named environment. Use the `azd env get-values` command to review all of the variables in your environment that were used when creating Azure resources.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs:

Console

```
azd down --no-prompt
```

ⓘ Note

The `--no-prompt` option instructs `azd` to delete your resource group without a confirmation from you.

This command doesn't affect your local code project.

Related content

- [Flex Consumption plan](#)
- [Azure Developer CLI \(azd\)](#)
- [azd reference](#)
- [Azure Functions Core Tools reference](#)
- [Code and test Azure Functions locally](#)

Feedback

Was this page helpful?

Yes

No

Quickstart: Create a PowerShell function in Azure using Visual Studio Code

Article • 07/18/2024

In this article, you use Visual Studio Code to create a PowerShell function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [CLI-based version](#) of this article.

Configure your environment

Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [PowerShell 7.2](#)
- [.NET 6.0 runtime](#)
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [PowerShell extension for Visual Studio Code](#).
- The [Azure Functions extension](#) for Visual Studio Code.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or

If Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in PowerShell. Later in this article, you'll publish your function code to Azure.

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create New Project...`.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[+] [Expand table](#)

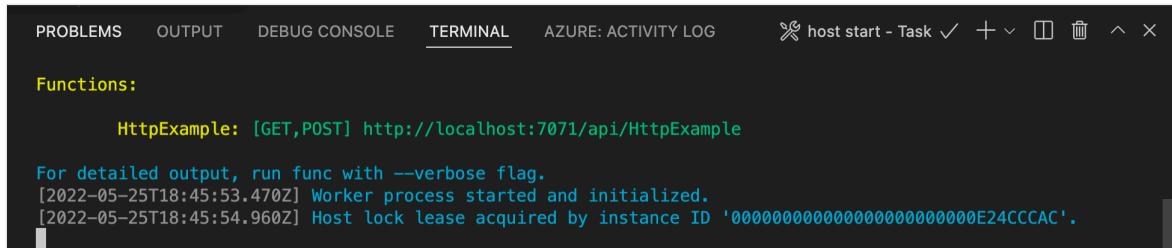
Prompt	Selection
Select a language for your function project	Choose <code>PowerShell</code> .
Select a template for your project's first function	Choose <code>HTTP trigger</code> .
Provide a function name	Type <code>HttpExample</code> .
Authorization level	Choose <code>Anonymous</code> , which enables anyone to call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Choose <code>Open in current window</code> .

Using this information, Visual Studio Code generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. To learn more about files that are created, see [Generated project files](#).

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

1. To start the function locally, press `F5` or the **Run and Debug** icon in the left-hand side Activity bar. The **Terminal** panel displays the Output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The output window displays the following text:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG
✖ host start - Task ✓ + □ ⌂ ⌂ ⌂ ×

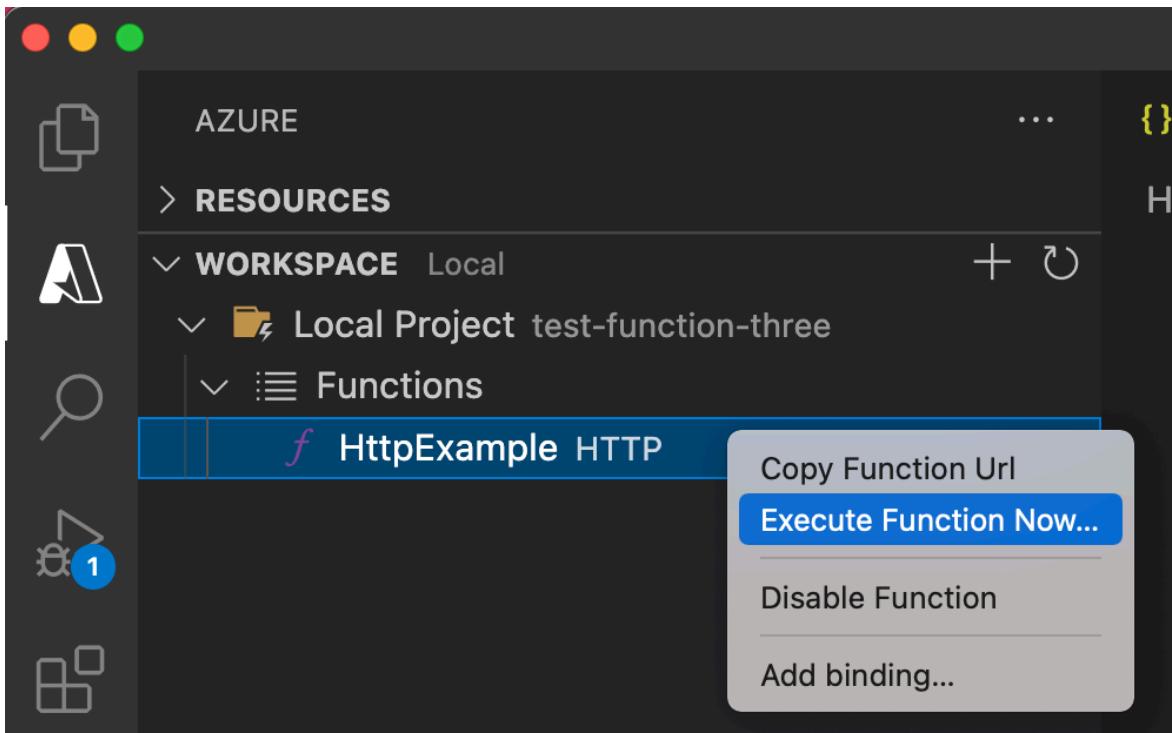
Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.
```

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

2. With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or `Ctrl + Click` (macOS) the new function and choose **Execute Function Now....**



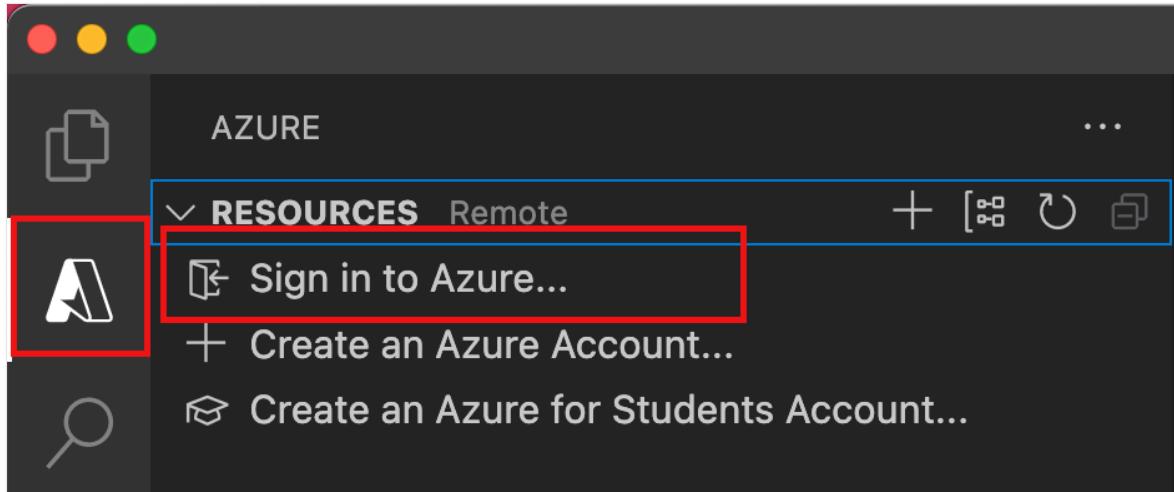
3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.
5. With the **Terminal** panel focused, press `Ctrl + C` to stop Core Tools and disconnect the debugger.

After you've verified that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

2. When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

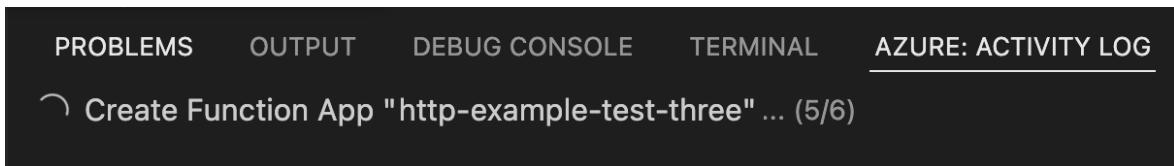
Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors.

1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
2. At the prompts, provide the following information:

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region ↗ near you.

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.



Tip

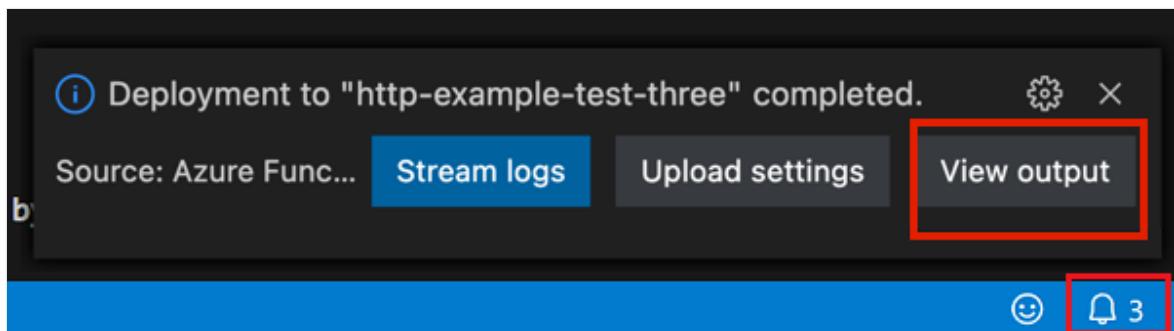
By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

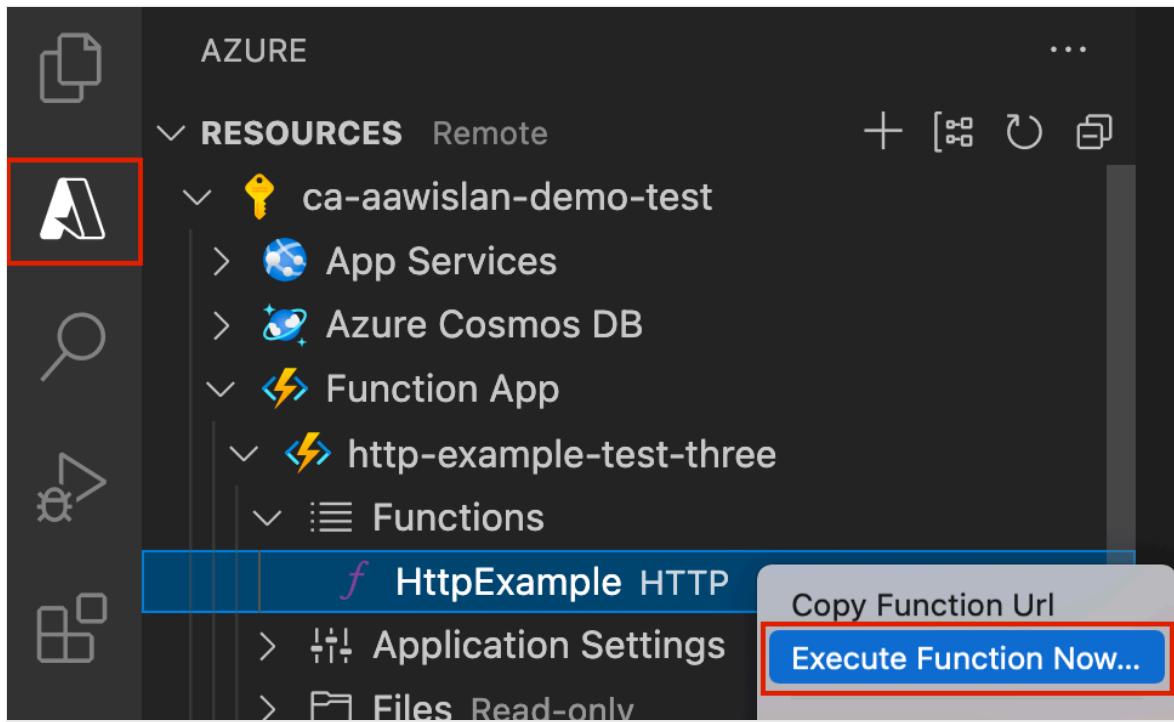
Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Back in the **Resources** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click (Windows) or **Ctrl -** click (macOS) the `HttpExample` function and choose **Execute Function Now....**



2. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
3. When the function executes in Azure and returns a response, a notification is raised in Visual Studio Code.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal's 'Overview' page for a function app named 'myfunctionapp'. The 'Resource group (change)' section is highlighted with a red box. Key details shown include:

- Status: Running
- Location: Central US
- Subscription: Visual Studio Enterprise
- Runtime version: 3.0.13139.0

The left sidebar lists various navigation options, and the bottom of the page has tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You have used [Visual Studio Code](#) to create a function app with a simple HTTP-triggered function. In the next article, you expand that function by connecting to Azure Storage. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to an Azure Storage queue](#)

Feedback

Was this page helpful?

[Yes](#)

[No](#)

[Provide product feedback](#) | Get help at Microsoft Q&A

Quickstart: Create a PowerShell function in Azure from the command line

Article • 08/24/2023

In this article, you use command-line tools to create a PowerShell function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There is also a [Visual Studio Code-based version](#) of this article.

Configure your local environment

Before you begin, you must have the following:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- One of the following tools for creating Azure resources:
 - The Azure [Az PowerShell module](#) version 9.4.0 or later.
 - [Azure CLI](#) version 2.4 or later.
- The [.NET 6.0 SDK ↗](#)
- [PowerShell 7.2](#)

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme ↗](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. [Visual Studio Code debugging](#) requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

Create a local function project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single function.

1. Run the `func init` command, as follows, to create a functions project in a folder named *LocalFunctionProj* with the specified runtime:

```
Console  
func init LocalFunctionProj --powershell
```

2. Navigate into the project folder:

```
Console  
cd LocalFunctionProj
```

This folder contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file.

3. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (`HttpExample`) and the `--template` argument specifies the function's trigger (HTTP).

```
Console  
func new --name HttpExample --template "HTTP trigger" --authlevel  
"anonymous"
```

`func new` creates a subfolder matching the function name that contains a code file appropriate to the project's chosen language and a configuration file named *function.json*.

(Optional) Examine the file contents

If desired, you can skip to [Run the function locally](#) and examine the file contents later.

run.ps1

run.ps1 defines a function script that's triggered according to the configuration in *function.json*.

```
PowerShell

using namespace System.Net

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Write to the Azure Functions log stream.
Write-Host "PowerShell HTTP trigger function processed a request."

# Interact with query parameters or the body of the request.
$name = $Request.Query.Name
if (-not $name) {
    $name = $Request.Body.Name
}

$body = "This HTTP triggered function executed successfully. Pass a name in
the query string or in the request body for a personalized response.

if ($name) {
    $body = "Hello, $name. This HTTP triggered function executed
successfully."
}

# Associate values to output bindings by calling 'Push-OutputBinding'.
Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = [HttpStatusCode]::OK
    Body = $body
})
```

For an HTTP trigger, the function receives request data passed to the `$Request` param defined in *function.json*. The return object, defined as `Response` in *function.json*, is passed to the `Push-OutputBinding` cmdlet as the response.

function.json

function.json is a configuration file that defines the input and output `bindings` for the function, including the trigger type.

```
JSON

{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "Request",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "Response"
    }
  ]
}
```

Each binding requires a direction, a type, and a unique name. The HTTP trigger has an input binding of type `httpTrigger` and output binding of type `http`.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

```
Console
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:    4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the following commands to create these items. Both Azure CLI and PowerShell are supported.

1. If you haven't done so already, sign in to Azure:

Azure CLI

Azure CLI

```
az login
```

The [az login](#) command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in your chosen region:

Azure CLI

Azure CLI

```
az group create --name AzureFunctionsQuickstart-rg --location  
<REGION>
```

The [az group create](#) command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the [az account list-locations](#) command.

3. Create a general-purpose storage account in your resource group and region:

Azure CLI

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location <REGION>  
--resource-group AzureFunctionsQuickstart-rg --sku Standard_LRS --  
allow-blob-public-access false
```

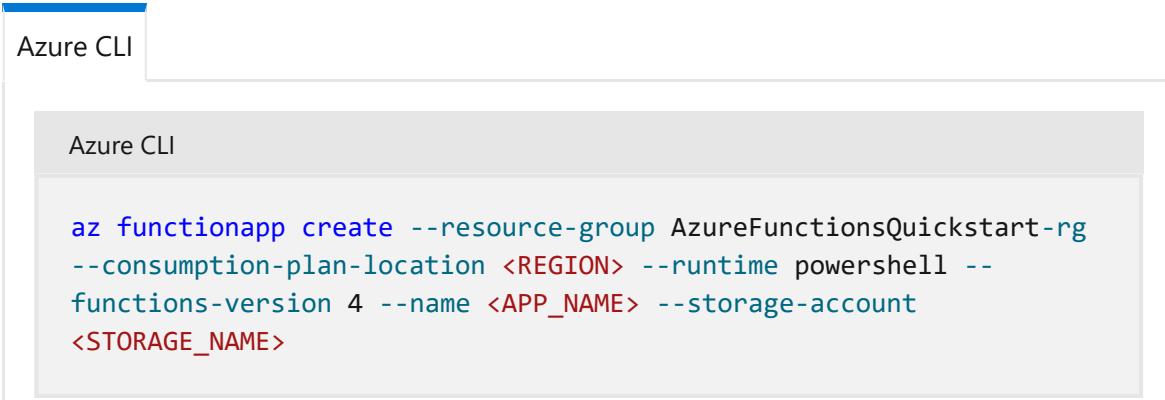
The [az storage account create](#) command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#).

 **Important**

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

4. Create the function app in Azure:



Azure CLI

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location <REGION> --runtime powershell --functions-version 4 --name <APP_NAME> --storage-account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure.

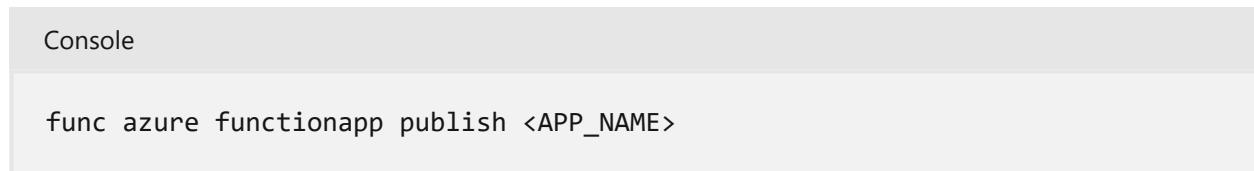
In the previous example, replace `<STORAGE_NAME>` with the name of the account you used in the previous step, and replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default DNS domain for the function app.

This command creates a function app running in your specified language runtime under the [Azure Functions Consumption Plan](#), which is free for the amount of usage you incur here. The command also provisions an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

Deploy the function project to Azure

After you've successfully created your function app in Azure, you're now ready to deploy your local functions project by using the `func azure functionapp publish` command.

In the following example, replace `<APP_NAME>` with the name of your app.



Console

```
func azure functionapp publish <APP_NAME>
```

The publish command shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

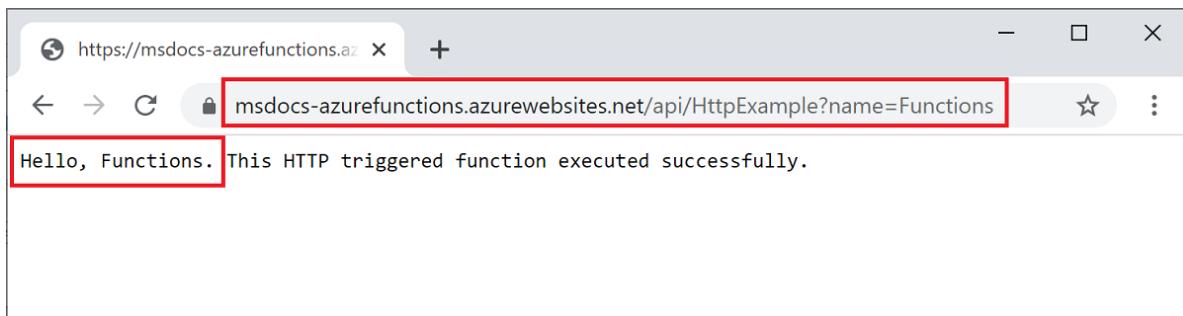
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
  HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-
qs.azurewebsites.net/api/httpexample
```

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.



Run the following command to view near real-time streaming logs:

Console

```
func azure functionapp logstream <APP_NAME>
```

In a separate terminal window or in the browser, call the remote function again. A verbose log of the function execution in Azure is shown in the terminal.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

```
Azure CLI
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

[Connect to an Azure Storage queue](#)

Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI

Article • 09/09/2024

In this Quickstart, you use Azure Developer command-line tools to create functions that respond to HTTP requests. After testing the code locally, you deploy it to a new serverless function app you create running in a Flex Consumption plan in Azure Functions.

The project source uses the Azure Developer CLI (azd) to simplify deploying your code to Azure. This deployment follows current best practices for secure and scalable Azure Functions deployments.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

By default, the Flex Consumption plan follows a *pay-for-what-you-use* billing model, which means to complete this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Azure Developer CLI](#).
- [Azure Functions Core Tools](#).
- [PowerShell 7.2](#)
- [.NET 6.0 SDK ↗](#)
- A [secure HTTP test tool](#) for sending requests with JSON payloads to your function endpoints. This article uses `curl`.

Initialize the project

You can use the `azd init` command to create a local Azure Functions code project from a template.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

```
Console
```

```
azd init --template functions-quickstart-powershell-azd -e flexquickstart-ps
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Run this command to navigate to the `src` app folder:

```
Console
```

```
cd src
```

3. Create a file named `local.settings.json` in the `src` folder that contains this JSON data:

```
JSON
```

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "powershell",  
        "FUNCTIONS_WORKER_RUNTIME_VERSION": "7.2"  
    }  
}
```

This file is required when running locally.

Run in your local environment

1. Run this command from your app folder in a terminal or command prompt:

```
Console
```

```
func start
```

When the Functions host starts in your local project folder, it writes the URL endpoints of your HTTP triggered functions to the terminal output.

2. In your browser, navigate to the `httpget` endpoint, which should look like this URL:

<http://localhost:7071/api/httpget>

3. From a new terminal or command prompt window, run this `curl` command to send a POST request with a JSON payload to the `httppost` endpoint:

Console

```
curl -i http://localhost:7071/api/httppost -H "Content-Type: text/json"  
-d @testdata.json
```

This command reads JSON payload data from the `testdata.json` project file. You can find examples of both HTTP requests in the `test.http` project file.

4. When you're done, press **Ctrl+C** in the terminal window to stop the `func.exe` host process.

Review the code (optional)

You can review the code that defines the two HTTP trigger function endpoints:

httpget

This `function.json` file defines the `httpget` function:

JSON

```
{  
  "bindings": [  
    {  
      "authLevel": "function",  
      "type": "httpTrigger",  
      "direction": "in",  
      "name": "Request",  
      "methods": [  
        "get"  
      ]  
    },  
  ],  
  "disabled": false  
}
```

```
{  
    "type": "http",  
    "direction": "out",  
    "name": "Response"  
}  
]  
}
```

This `run.ps1` file implements the function code:

PowerShell

```
using namespace System.Net  
  
# Input bindings are passed in via param block.  
param($Request, $TriggerMetadata)  
  
# Write to the Azure Functions log stream.  
Write-Host "PowerShell HTTP trigger function processed a request."  
  
# Interact with query parameters  
$name = $Request.Query.name  
  
$body = "This HTTP triggered function executed successfully. Pass a name  
in the query string for a personalized response."  
  
if ($name) {  
    $body = "Hello, $name. This HTTP triggered function executed  
successfully."  
}  
  
# Associate values to output bindings by calling 'Push-OutputBinding'.  
Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{  
    StatusCode = [ HttpStatusCode ]::OK  
    Body = $body  
})
```

You can review the complete template project [here ↗](#).

After you verify your functions locally, it's time to publish them to Azure.

Deploy to Azure

This project is configured to use the `azd up` command to deploy this project to a new function app in a Flex Consumption plan in Azure.

 Tip

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

1. Run this command to have `azd` create the required Azure resources in Azure and deploy your code project to the new function app:

```
Console
```

```
azd up
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

[\[+\] Expand table](#)

Parameter	Description
<code>Azure subscription</code>	Subscription in which your resources are created.
<code>Azure location</code>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd up` command uses your response to these prompts with the Bicep configuration files to complete these deployment tasks:

- Create and configure these required Azure resources (equivalent to `azd provision`):
 - Flex Consumption plan and function app
 - Azure Storage (required) and Application Insights (recommended)
 - Access policies and roles for your account
 - Service-to-service connections using managed identities (instead of stored connection strings)
 - Virtual network to securely run both the function app and the other Azure resources
- Package and deploy your code to the deployment container (equivalent to `azd deploy`). The app is then started and runs in the deployed package.

After the command completes successfully, you see links to the resources you created.

Invoke the function on Azure

You can now invoke your function endpoints in Azure by making HTTP requests to their URLs using your HTTP test tool or from the browser (for GET requests). When your functions run in Azure, access key authorization is enforced, and you must provide a function access key with your request.

You can use the Core Tools to obtain the URL endpoints of your functions running in Azure.

1. In your local terminal or command prompt, run these commands to get the URL endpoint values:

PowerShell

PowerShell

```
$APP_NAME = azd env get-value AZURE_FUNCTION_NAME  
func azure functionapp list-functions $APP_NAME --show-keys
```

The `azd env get-value` command gets your function app name from the local environment. Using the `--show-keys` option with `func azure functionapp list-functions` means that the returned **Invoke URL:** value for each endpoint includes a function-level access key.

2. As before, use your HTTP test tool to validate these URLs in your function app running in Azure.

Redeploy your code

You can run the `azd up` command as many times as you need to both provision your Azure resources and deploy code updates to your function app.

 **Note**

Deployed code files are always overwritten by the latest deployment package.

Your initial responses to `azd` prompts and any environment variables generated by `azd` are stored locally in your named environment. Use the `azd env get-values` command to review all of the variables in your environment that were used when creating Azure resources.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs:

Console

```
azd down --no-prompt
```

Note

The `--no-prompt` option instructs `azd` to delete your resource group without a confirmation from you.

This command doesn't affect your local code project.

Related content

- [Flex Consumption plan](#)
- [Azure Developer CLI \(azd\)](#)
- [azd reference](#)
- [Azure Functions Core Tools reference](#)
- [Code and test Azure Functions locally](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a function in Azure with Python using Visual Studio Code

Article • 09/10/2024

In this article, you use Visual Studio Code to create a Python function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

This article uses the Python v2 programming model for Azure Functions, which provides a decorator-based approach for creating functions. To learn more about the Python v2 programming model, see the [Developer Reference Guide](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [CLI-based version](#) of this article.

This video shows you how to create a Python function in Azure using Visual Studio Code.

[https://learn-video.azurefd.net/vod/player?id=a1e10f96-2940-489c-bc53-da2b915c8fc2&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-python ↗](https://learn-video.azurefd.net/vod/player?id=a1e10f96-2940-489c-bc53-da2b915c8fc2&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-python)

The steps in the video are also described in the following sections.

Configure your environment

Before you begin, make sure that you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- A Python version that is [supported by Azure Functions](#). For more information, see [How to install Python ↗](#).
- [Visual Studio Code ↗](#) on one of the [supported platforms ↗](#).
- The [Python extension ↗](#) for Visual Studio Code.
- The [Azure Functions extension ↗](#) for Visual Studio Code, version 1.8.1 or later.
- The [Azurite V3 extension ↗](#) local storage emulator. While you can also use an actual Azure storage account, this article assumes you're using the Azurite emulator.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in Python. Later in this article, you publish your function code to Azure.

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create New Project...`.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[] [Expand table](#)

Prompt	Selection
Select a language	Choose <code>Python (Programming Model V2)</code> .
Select a Python interpreter to create a virtual environment	Choose your preferred Python interpreter. If an option isn't shown, type in the full path to your Python binary.
Select a template for your project's first function	Choose <code>HTTP trigger</code> .
Name of the function you want to create	Enter <code>HttpExample</code> .

Prompt	Selection
Authorization level	Choose <code>ANONYMOUS</code> , which lets anyone call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Choose <code>Open in current window</code> .

- Visual Studio Code uses the provided information and generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. The generated `function_app.py` project file contains your functions.
- In the `local.settings.json` file, update the `AzureWebJobsStorage` setting as in the following example:

JSON

```
"AzureWebJobsStorage": "UseDevelopmentStorage=true",
```

This tells the local Functions host to use the storage emulator for the storage connection required by the Python v2 model. When you publish your project to Azure, this setting uses the default storage account instead. If you're using an Azure Storage account during local development, set your storage account connection string here.

Start the emulator

- In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azurite: Start`.
- Check the bottom bar and verify that Azurite emulation services are running. If so, you can now run your function locally.

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

- To start the function locally, press `F5` or the Run and Debug icon in the left-hand side Activity bar. The Terminal panel displays the Output from Core Tools. Your app starts in the Terminal panel. You can see the URL endpoint of your HTTP-triggered function running locally.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The output window displays the following text:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG host start - Task ✓ + × ↻ ↺ ↺ ×

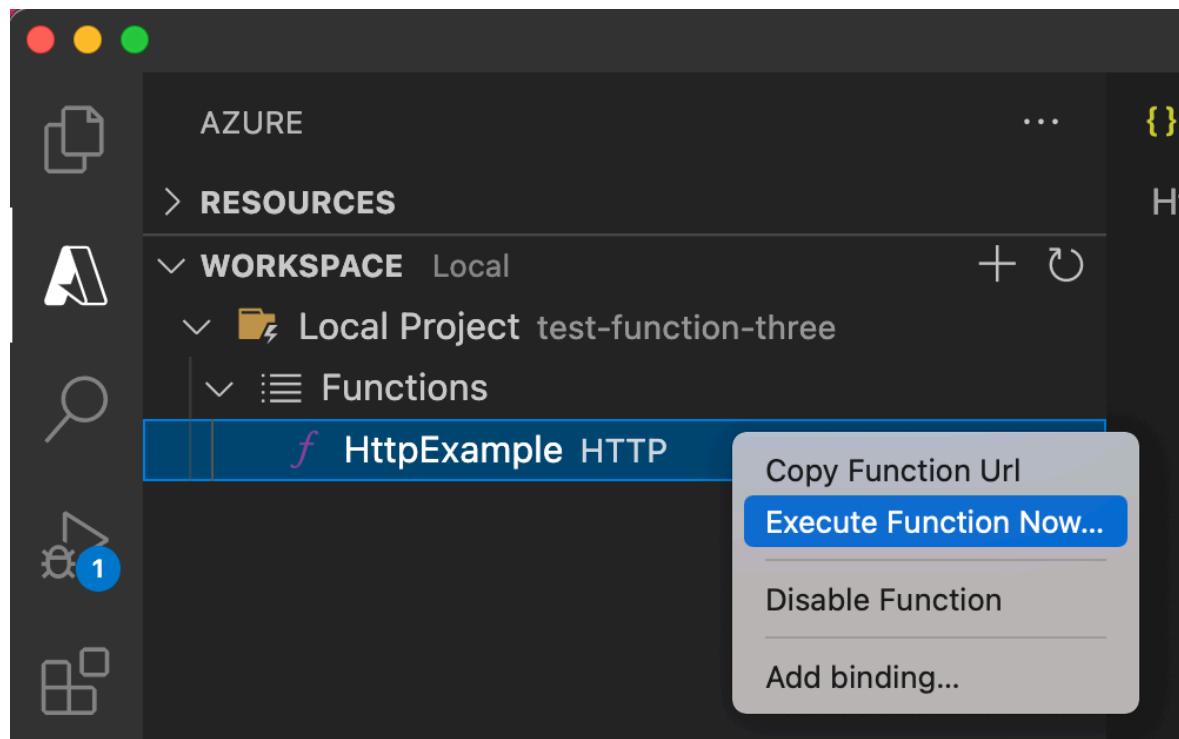
Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.
```

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

- With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or **Ctrl + click** (macOS) the new function and choose **Execute Function Now....**



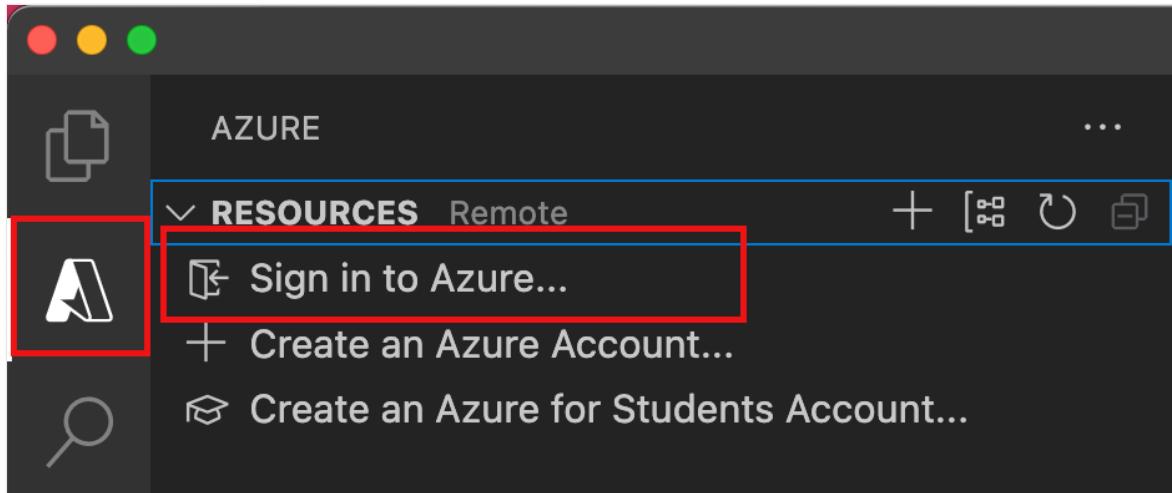
- In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
- When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.
- With the **Terminal** panel focused, press **Ctrl + C** to stop Core Tools and disconnect the debugger.

After you verify that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

2. When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

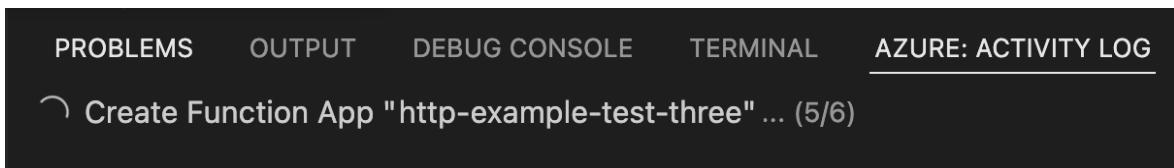
In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors. For more control over the created resources, you must instead [create your function app with advanced options](#).

1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
2. At the prompts, provide the following information:

[] [Expand table](#)

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region near you.

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



- When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

Tip

By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources

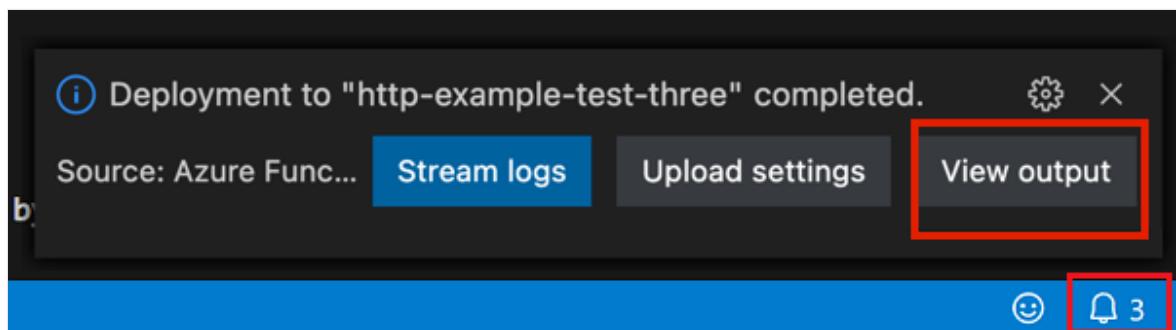
are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Press **F1** to display the command palette, then search for and run the command **Azure Functions:Execute Function Now....**. If prompted, select your subscription.
2. Select your new function app resource and **HttpExample** as your function.
3. In **Enter request body** type `{ "name": "Azure" }`, then press Enter to send this request message to your function.

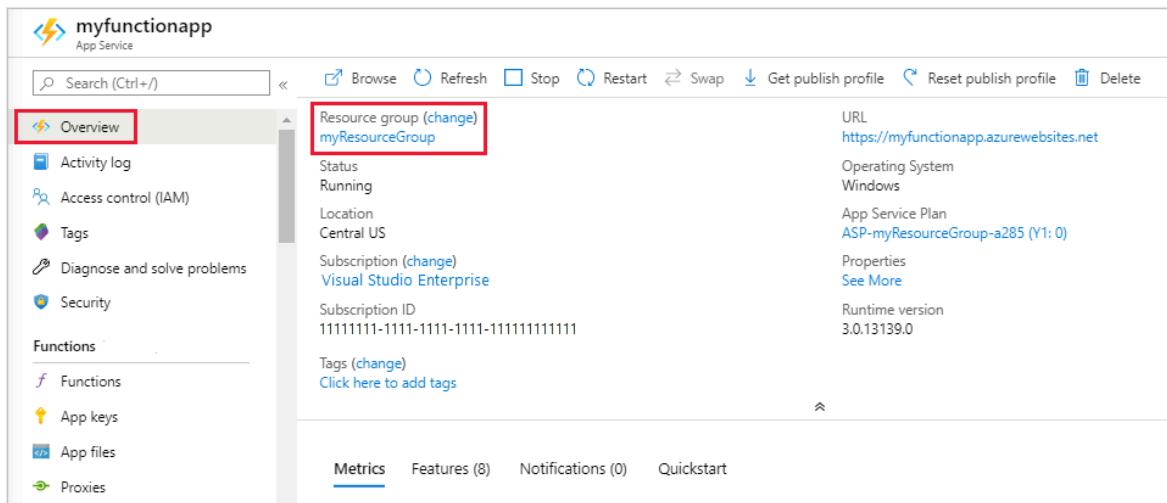
- When the function executes in Azure, the response is displayed in the notification area. Expand the notification to review the full response.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

- In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
- Choose your function app and press `Enter`. The function app page opens in the Azure portal.
- In the **Overview** tab, select the named link next to **Resource group**.



- On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
- Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You created and deployed a function app with a simple HTTP-triggered function. In the next articles, you expand that function by connecting to a storage service in Azure. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to Azure Cosmos DB](#)

[Connect to an Azure Storage queue](#)

Having issues? Let us know. ↗

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a Python function in Azure from the command line

Article • 03/05/2024

In this article, you use command-line tools to create a Python function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

This article uses the Python v2 programming model for Azure Functions, which provides a decorator-based approach for creating functions. To learn more about the Python v2 programming model, see the [Developer Reference Guide](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [Visual Studio Code-based version](#) of this article.

Configure your local environment

Before you begin, you must have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - The Azure [Az PowerShell module](#) version 5.9.0 or later.
- A Python version supported by Azure Functions.
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. Visual Studio Code debugging requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

Use the `func --version` command to make sure your version of Core Tools is at least [4.0.5530](#).

Create and activate a virtual environment

In a suitable folder, run the following commands to create and activate a virtual environment named `.venv`. Make sure that you're using a [version of Python supported by Azure Functions](#).

```
bash
```

```
Bash
```

```
python -m venv .venv
```

```
Bash
```

```
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

```
Bash
```

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment.

Create a local function

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations.

In this section, you create a function project and add an HTTP triggered function.

1. Run the `func init` command as follows to create a Python v2 functions project in the virtual environment.

```
Console  
func init --python
```

The environment now contains various files for the project, including configuration files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file.

2. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (`HttpExample`) and the `--template` argument specifies the function's trigger (HTTP).

```
Console  
func new --name HttpExample --template "HTTP trigger" --authlevel  
"anonymous"
```

If prompted, choose the **ANONYMOUS** option. `func new` adds an HTTP trigger endpoint named `HttpExample` to the `function_app.py` file, which is accessible without authentication.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the `LocalFunctionProj` folder.

```
Console
```

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:        4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create three resources:

- A resource group, which is a logical container for related resources.
- A storage account, which maintains the state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the following commands to create these items. Both Azure CLI and PowerShell are supported.

1. If needed, sign in to Azure.

```
Azure CLI
```

```
Azure CLI
```

```
az login
```

The `az login` command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in your chosen region.

```
Azure CLI
```

```
Azure CLI
```

```
az group create --name AzureFunctionsQuickstart-rg --location  
<REGION>
```

The `az group create` command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the [az account list-locations](#) command.

ⓘ Note

You can't host Linux and Windows apps in the same resource group. If you have an existing resource group named `AzureFunctionsQuickstart-rg` with a Windows function app or web app, you must use a different resource group.

3. Create a general-purpose storage account in your resource group and region.

```
Azure CLI
```

```
Azure CLI
```

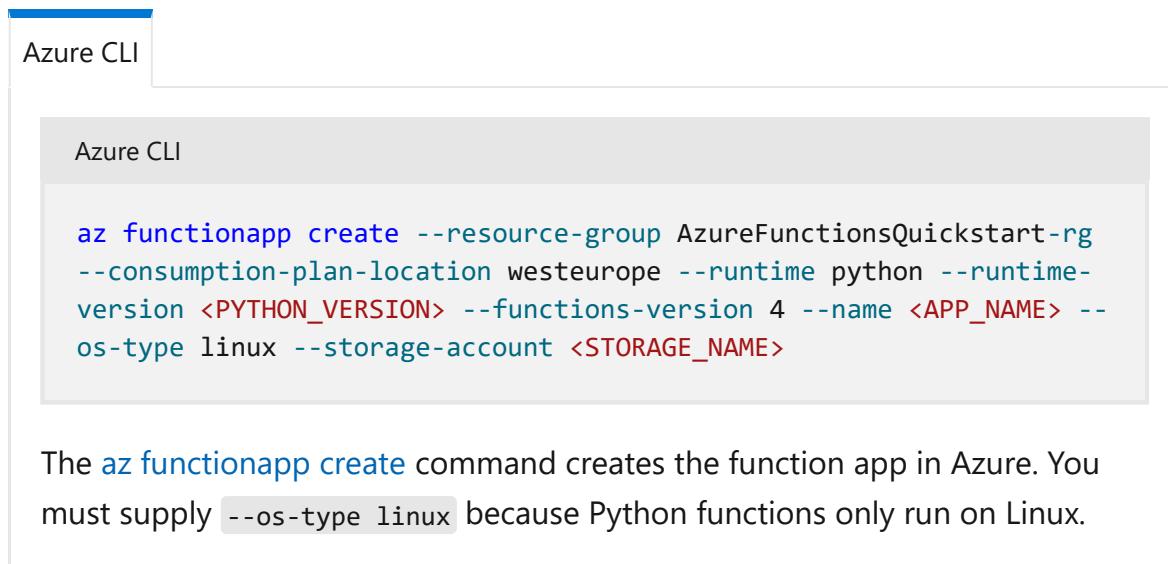
```
az storage account create --name <STORAGE_NAME> --location <REGION>  
--resource-group AzureFunctionsQuickstart-rg --sku Standard_LRS
```

The `az storage account create` command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that's appropriate to you and unique in Azure Storage. Names must contain 3 to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account supported by Functions.

The storage account incurs only a few cents (USD) for this quickstart.

4. Create the function app in Azure.



Azure CLI

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location westeurope --runtime python --runtime-version <PYTHON_VERSION> --functions-version 4 --name <APP_NAME> --os-type linux --storage-account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure. You must supply `--os-type linux` because Python functions only run on Linux.

In the previous example, replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default subdomain for the function app. Make sure that the value you set for `<PYTHON_VERSION>` is a version supported by Functions and is the same version you used during local development.

This command creates a function app running in your specified language runtime under the [Azure Functions Consumption Plan](#), which is free for the amount of usage you incur here. The command also creates an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

Deploy the function project to Azure

After you've successfully created your function app in Azure, you're now ready to deploy your local functions project by using the `func azure functionapp publish` command.

In the following example, replace `<APP_NAME>` with the name of your app.

Console

```
func azure functionapp publish <APP_NAME>
```

The publish command shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
  HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-
qs.azurewebsites.net/api/httpexample
```

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the `publish` command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

[Connect to Azure Cosmos DB](#)

[Connect to an Azure Storage queue](#)

Having issues with this article?

- [Troubleshoot Python function apps in Azure Functions](#)
- [Let us know ↗](#)

Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI

Article • 09/09/2024

In this Quickstart, you use Azure Developer command-line tools to create functions that respond to HTTP requests. After testing the code locally, you deploy it to a new serverless function app you create running in a Flex Consumption plan in Azure Functions.

The project source uses the Azure Developer CLI (azd) to simplify deploying your code to Azure. This deployment follows current best practices for secure and scalable Azure Functions deployments.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

By default, the Flex Consumption plan follows a *pay-for-what-you-use* billing model, which means to complete this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Azure Developer CLI](#).
- [Azure Functions Core Tools](#).
- [Python 3.11 ↗](#).
- A [secure HTTP test tool](#) for sending requests with JSON payloads to your function endpoints. This article uses `curl`.

Initialize the project

You can use the `azd init` command to create a local Azure Functions code project from a template.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

```
Console
```

```
azd init --template functions-quickstart-python-http-azd -e flexquickstart-py
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Create a file named `local.settings.json` in the root folder that contains this JSON data:

```
JSON
```

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "python"  
    }  
}
```

This file is required when running locally.

Create and activate a virtual environment

In the root folder, run these commands to create and activate a virtual environment named `.venv`:

Linux/macOS

Bash

```
python3 -m venv .venv  
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

Bash

```
sudo apt-get install python3-venv
```

Run in your local environment

1. Run this command from your app folder in a terminal or command prompt:

Console

```
func start
```

When the Functions host starts in your local project folder, it writes the URL endpoints of your HTTP triggered functions to the terminal output.

2. In your browser, navigate to the `httpget` endpoint, which should look like this URL:

<http://localhost:7071/api/httpget>

3. From a new terminal or command prompt window, run this `curl` command to send a POST request with a JSON payload to the `httppost` endpoint:

Console

```
curl -i http://localhost:7071/api/httppost -H "Content-Type: text/json"  
-d @testdata.json
```

This command reads JSON payload data from the `testdata.json` project file. You can find examples of both HTTP requests in the `test.http` project file.

4. When you're done, press **Ctrl+C** in the terminal window to stop the `func.exe` host process.

5. Run `deactivate` to shut down the virtual environment.

Review the code (optional)

You can review the code that defines the two HTTP trigger function endpoints:

httpget

Python

```
@app.route(route="httpget", methods=["GET"])
def http_get(req: func.HttpRequest) -> func.HttpResponse:
    name = req.params.get("name", "World")

    logging.info(f"Processing GET request. Name: {name}")

    return func.HttpResponse(f"Hello, {name}!")
```

You can review the complete template project [here ↗](#).

After you verify your functions locally, it's time to publish them to Azure.

Deploy to Azure

This project is configured to use the `azd up` command to deploy this project to a new function app in a Flex Consumption plan in Azure.

💡 Tip

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

1. Run this command to have `azd` create the required Azure resources in Azure and deploy your code project to the new function app:

Console

```
azd up
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

[+] Expand table

Parameter	Description
<i>Azure subscription</i>	Subscription in which your resources are created.
<i>Azure location</i>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd up` command uses your response to these prompts with the Bicep configuration files to complete these deployment tasks:

- Create and configure these required Azure resources (equivalent to `azd provision`):
 - Flex Consumption plan and function app
 - Azure Storage (required) and Application Insights (recommended)
 - Access policies and roles for your account
 - Service-to-service connections using managed identities (instead of stored connection strings)
 - Virtual network to securely run both the function app and the other Azure resources
- Package and deploy your code to the deployment container (equivalent to `azd deploy`). The app is then started and runs in the deployed package.

After the command completes successfully, you see links to the resources you created.

Invoke the function on Azure

You can now invoke your function endpoints in Azure by making HTTP requests to their URLs using your HTTP test tool or from the browser (for GET requests). When your functions run in Azure, access key authorization is enforced, and you must provide a function access key with your request.

You can use the Core Tools to obtain the URL endpoints of your functions running in Azure.

1. In your local terminal or command prompt, run these commands to get the URL endpoint values:

```
bash
```

Bash

```
SET APP_NAME=(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp list-functions $APP_NAME --show-keys
```

The `azd env get-value` command gets your function app name from the local environment. Using the `--show-keys` option with `func azure functionapp list-functions` means that the returned **Invoke URL:** value for each endpoint includes a function-level access key.

2. As before, use your HTTP test tool to validate these URLs in your function app running in Azure.

Redeploy your code

You can run the `azd up` command as many times as you need to both provision your Azure resources and deploy code updates to your function app.

 Note

Deployed code files are always overwritten by the latest deployment package.

Your initial responses to `azd` prompts and any environment variables generated by `azd` are stored locally in your named environment. Use the `azd env get-values` command to review all of the variables in your environment that were used when creating Azure resources.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs:

Console

```
azd down --no-prompt
```

 Note

The `--no-prompt` option instructs `azd` to delete your resource group without a confirmation from you.

This command doesn't affect your local code project.

Related content

- [Flex Consumption plan](#)
- [Azure Developer CLI \(azd\)](#)
- [azd reference](#)
- [Azure Functions Core Tools reference](#)
- [Code and test Azure Functions locally](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a function in Azure with TypeScript using Visual Studio Code

Article • 07/18/2024

In this article, you use Visual Studio Code to create a TypeScript function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of the page. The v4 model is generally available and is designed to have a more flexible and intuitive experience for JavaScript and TypeScript developers. Learn more about the differences between v3 and v4 in the [migration guide](#).

Completion of this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [CLI-based version](#) of this article.

Configure your environment

Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js 18.x](#) or above. Use the `node --version` command to check your version.
- [TypeScript 4.x](#). Use the `tsc -v` command to check your version.
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [Azure Functions extension v1.10.4](#) or above for Visual Studio Code.
- [Azure Functions Core Tools v4.0.5382](#) or above.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in TypeScript. Later in this article, you publish your function code to Azure.

1. In Visual Studio Code, press **F1** to open the command palette and search for and run the command **Azure Functions: Create New Project...**.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[+] Expand table

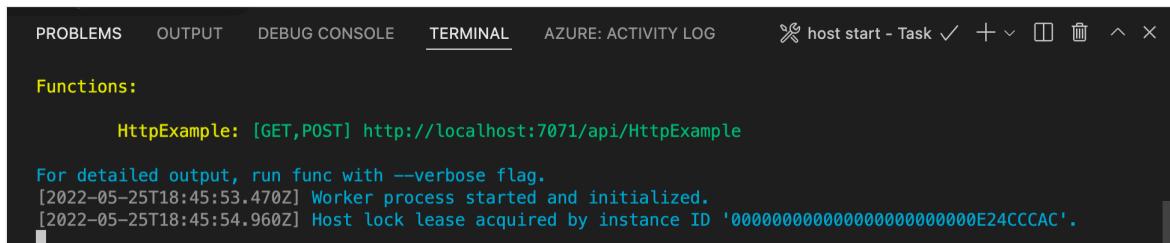
Prompt	Selection
Select a language for your function project	Choose TypeScript .
Select a TypeScript programming model	Choose Model V4
Select a template for your project's first function	Choose HTTP trigger .
Provide a function name	Type HttpExample .
Select how you would like to open your project	Choose Open in current window

Using this information, Visual Studio Code generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. To learn more about files that are created, see [Azure Functions TypeScript developer guide](#).

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

1. To start the function locally, press **F5** or the **Run and Debug** icon in the left-hand side Activity bar. The **Terminal** panel displays the Output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The output window displays the following text:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG
✖ host start - Task ✓ + ×

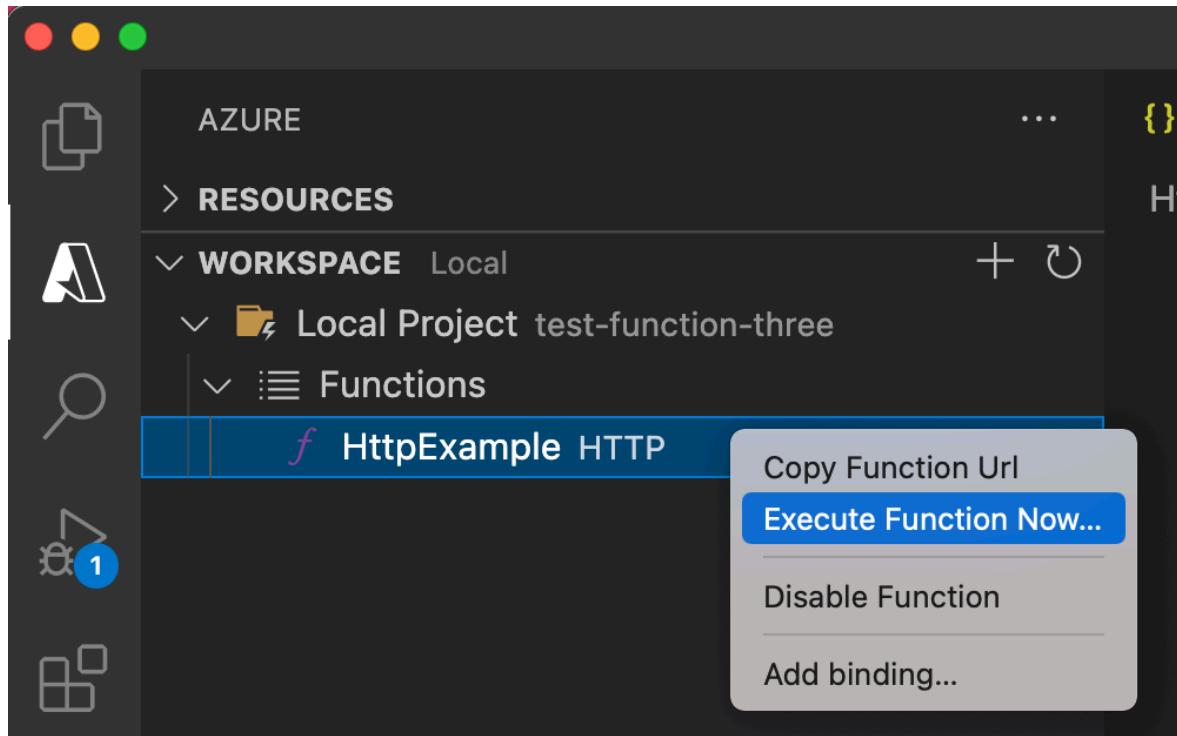
Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.
```

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

2. With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or **Ctrl + click** (macOS) the new function and choose **Execute Function Now...**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.

4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.

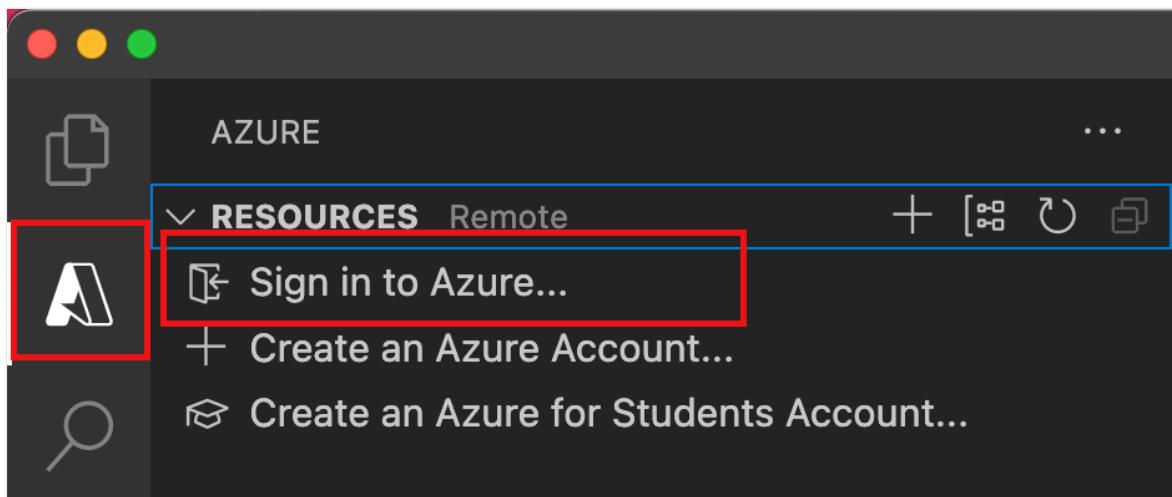
5. With the **Terminal** panel focused, press **Ctrl + c** to stop Core Tools and disconnect the debugger.

After you've verified that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

2. When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors.

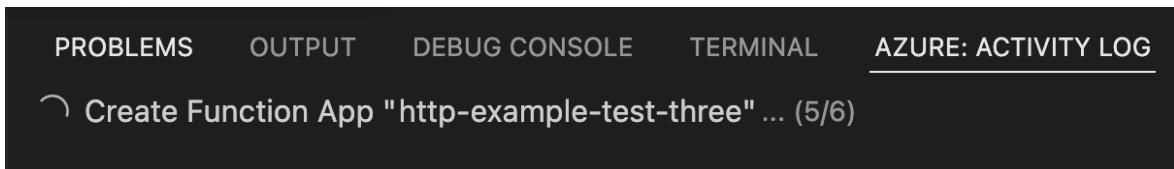
1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
2. At the prompts, provide the following information:

[] [Expand table](#)

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.

Prompt	Action
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region near you .

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

💡 Tip

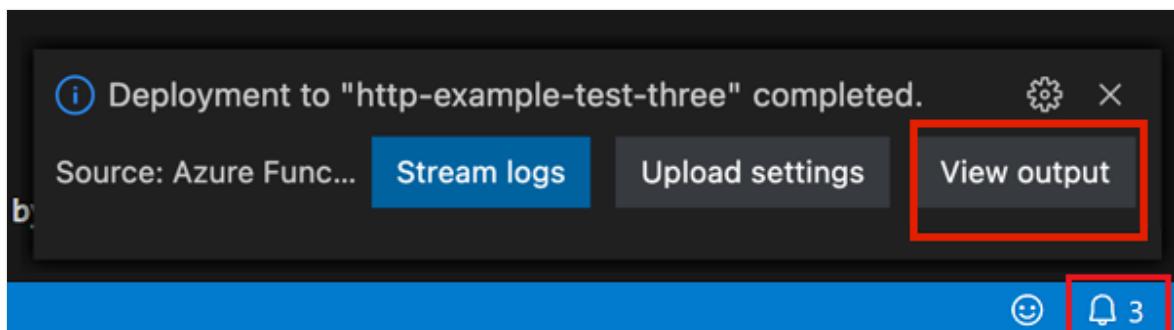
By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

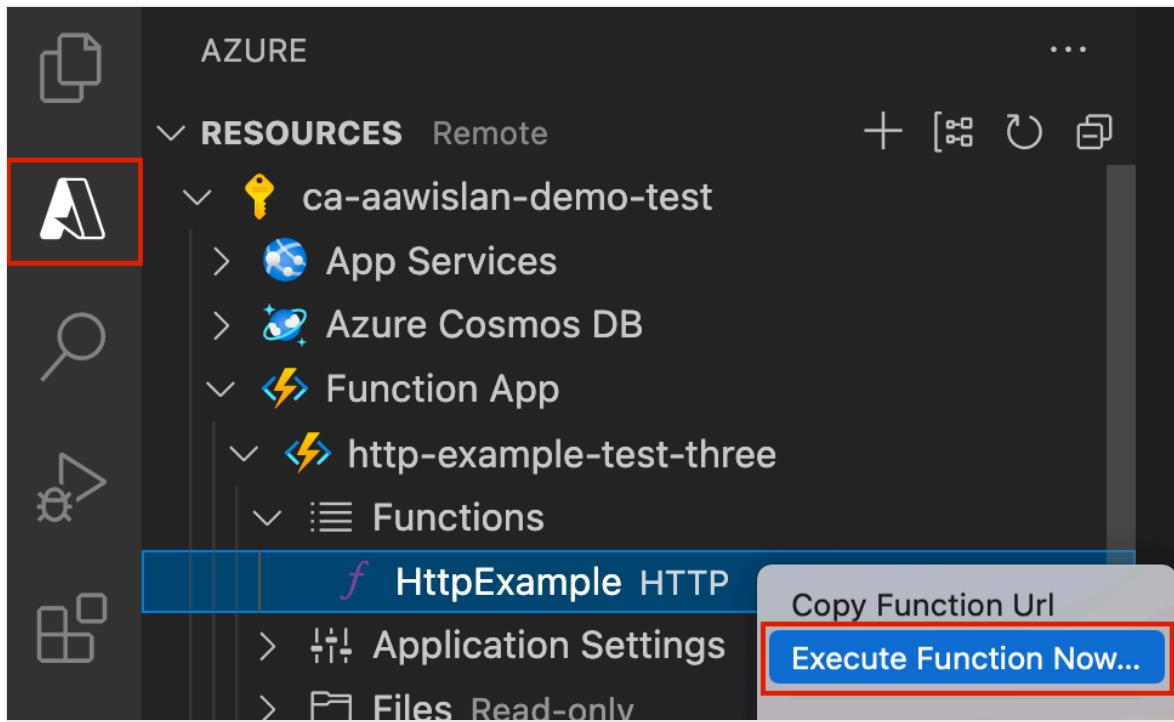
Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Back in the **Resources** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click (Windows) or **Ctrl -** click (macOS) the `HttpExample` function and choose **Execute Function Now....**



2. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
3. When the function executes in Azure and returns a response, a notification is raised in Visual Studio Code.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal's 'Overview' page for a function app named 'myfunctionapp'. The 'Resource group (change)' section is highlighted with a red box. Key details shown include:

- Status: Running
- Location: Central US
- Subscription: Visual Studio Enterprise
- Runtime version: 3.0.13139.0

The left sidebar lists various navigation options, and the bottom of the page has tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You have used [Visual Studio Code](#) to create a function app with a simple HTTP-triggered function. In the next article, you expand that function by connecting to Azure Storage. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to an Azure Storage queue](#)

Feedback

Was this page helpful?

[Yes](#)

[No](#)

[Provide product feedback](#) | Get help at Microsoft Q&A

Quickstart: Create a TypeScript function in Azure from the command line

Article • 12/19/2023

In this article, you use command-line tools to create a TypeScript function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of the page. The v4 model is generally available and is designed to have a more flexible and intuitive experience for JavaScript and TypeScript developers. Learn more about the differences between v3 and v4 in the [migration guide](#).

Completion of this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [Visual Studio Code-based version](#) of this article.

Configure your local environment

Before you begin, you must have the following prerequisites:

- An Azure account with an active subscription. [Create an account for free](#).
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - The Azure [Az PowerShell module](#) version 5.9.0 or later.
- [Node.js](#) version 18 or above.
- [TypeScript](#) version 4+.

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. Visual Studio Code debugging requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

- Make sure you install version v4.0.5382 of the Core Tools, or a later version.

Create a local function project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single function.

1. In a suitable folder, run the `func init` command, as follows, to create a TypeScript Node.js v4 project in the current folder:

```
Console  
func init --typescript
```

This folder now contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file. Required npm packages are also installed in `node_modules`.

2. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (HttpExample) and the `--template` argument specifies the function's trigger (HTTP).

```
Console
```

```
func new --name HttpExample --template "HTTP trigger" --authlevel  
"anonymous"
```

`func new` creates a file named *HttpExample.ts* in the *src/functions* directory, which contains your function's code.

3. Add Azure Storage connection information in *local.settings.json*.

JSON

```
{  
  "Values": {  
    "AzureWebJobsStorage": "<Azure Storage connection  
information>",  
    "FUNCTIONS_WORKER_RUNTIME": "node"  
  }  
}
```

4. (Optional) If you want to learn more about a particular function, say HTTP trigger, you can run the following command:

Console

```
func help httptrigger
```

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder:

Console

```
npm start
```

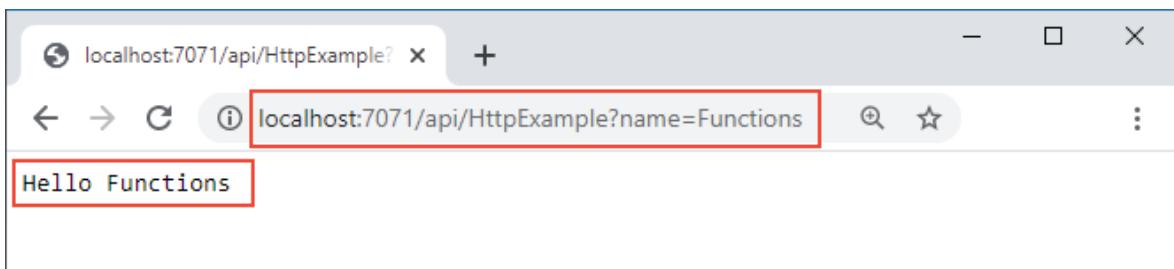
Toward the end of the output, the following logs should appear:

```
Azure Functions Core Tools  
Core Tools Version: 4.0.5049 Commit hash: N/A (64-bit)  
Function Runtime Version: 4.15.2.20177  
  
[2023-03-17T03:27:12.372Z] Worker process started and initialized.  
  
Functions:  
  
  HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

(!) Note

If `HttpExample` doesn't appear as shown in the logs, you likely started the host from outside the root folder of the project. In that case, use `Ctrl + C` to stop the host, navigate to the project's root folder, and run the previous command again.

2. Copy the URL of your `HttpExample` function from this output to a browser and append the query string `?name=<your-name>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a message like `Hello Functions`:



The terminal in which you started your project also shows log output as you make requests.

3. When you're ready, use `Ctrl + C` and choose `y` to stop the functions host.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the following commands to create these items. Both Azure CLI and PowerShell are supported.

1. If you haven't done so already, sign in to Azure:

Azure CLI

Azure CLI

```
az login
```

The `az login` command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in your chosen region:

Azure CLI

Azure CLI

```
az group create --name AzureFunctionsQuickstart-rg --location  
<REGION>
```

The `az group create` command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the `az account list-locations` command.

3. Create a general-purpose storage account in your resource group and region:

Azure CLI

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location <REGION>  
--resource-group AzureFunctionsQuickstart-rg --sku Standard_LRS --  
allow-blob-public-access false
```

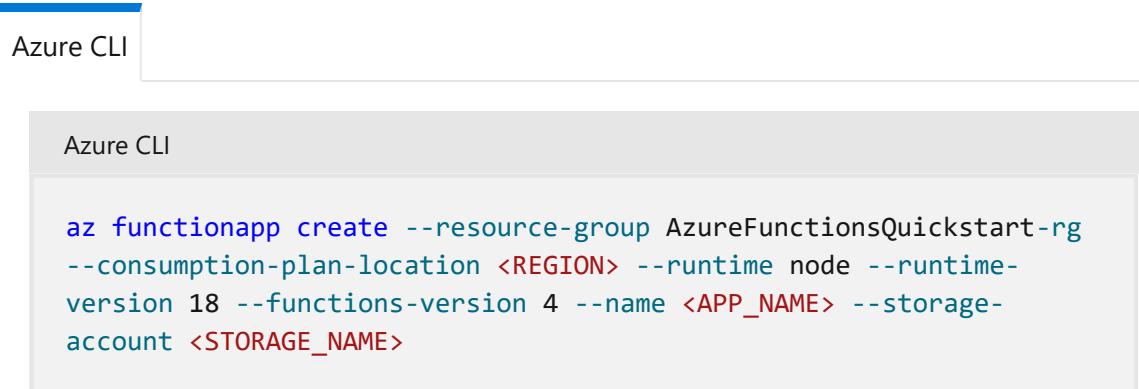
The `az storage account create` command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#).

 **Important**

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

4. Create the function app in Azure:



```
Azure CLI
az functionapp create --resource-group AzureFunctionsQuickstart-rg
--consumption-plan-location <REGION> --runtime node --runtime-
version 18 --functions-version 4 --name <APP_NAME> --storage-
account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure. It's recommended that you use the latest version of Node.js, which is currently 18. You can specify the version by setting `--runtime-version` to 18.

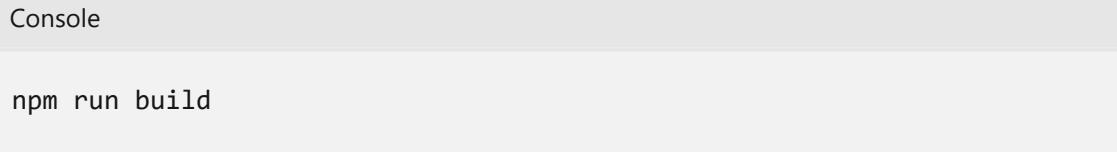
In the previous example, replace `<STORAGE_NAME>` with the name of the account you used in the previous step, and replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default DNS domain for the function app.

This command creates a function app running in your specified language runtime under the [Azure Functions Consumption Plan](#), which is free for the amount of usage you incur here. The command also creates an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

Deploy the function project to Azure

Before you use Core Tools to deploy your project to Azure, you create a production-ready build of JavaScript files from the TypeScript source files.

1. Use the following command to prepare your TypeScript project for deployment:



```
Console
npm run build
```

2. With the necessary resources in place, you're now ready to deploy your local functions project to the function app in Azure by using the [publish](#) command. In the following example, replace `<APP_NAME>` with the name of your app.

Console

```
func azure functionapp publish <APP_NAME>
```

If you see the error, "Can't find app with name ...", wait a few seconds and try again, as Azure may not have fully initialized the app after the previous `az functionapp create` command.

The publish command shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

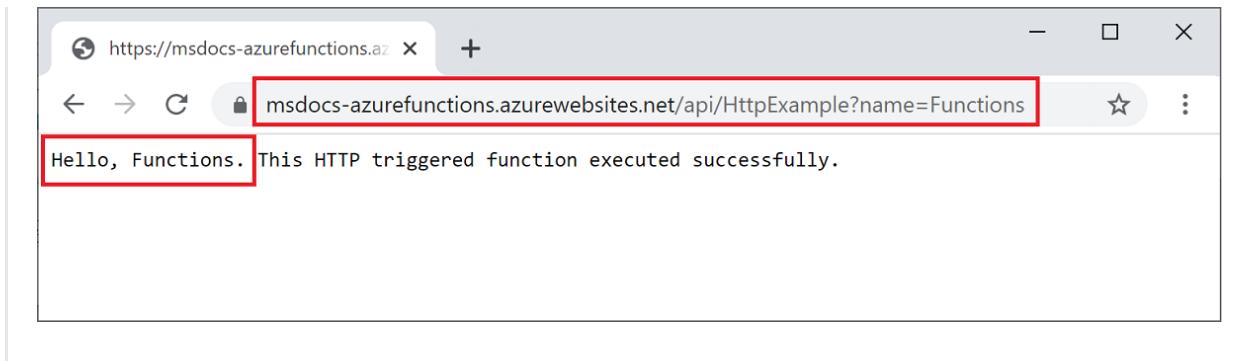
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
    HttpExample - [httpTrigger]
        Invoke url: https://msdocs-azurefunctions-
qs.azurewebsites.net/api/httpexample?
code=KYHrydo4GFe9y0000000qRgRJ8NdLFKpkakGJQfC3izYVidzzDN4gQ==
```

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.



Run the following command to view near real-time streaming logs:

```
Console  
  
func azure functionapp logstream <APP_NAME>
```

In a separate terminal window or in the browser, call the remote function again. A verbose log of the function execution in Azure is shown in the terminal.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

```
Azure CLI  
  
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

[Connect to an Azure Storage queue](#)

Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI

Article • 09/09/2024

In this Quickstart, you use Azure Developer command-line tools to create functions that respond to HTTP requests. After testing the code locally, you deploy it to a new serverless function app you create running in a Flex Consumption plan in Azure Functions.

The project source uses the Azure Developer CLI (azd) to simplify deploying your code to Azure. This deployment follows current best practices for secure and scalable Azure Functions deployments.

Important

The [Flex Consumption plan](#) is currently in preview.

By default, the Flex Consumption plan follows a *pay-for-what-you-use* billing model, which means to complete this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Azure Developer CLI](#).
- [Azure Functions Core Tools](#).
- [.NET 8.0 SDK](#)
- [Azurite storage emulator](#)
- [Java 17 Developer Kit](#)
 - If you use another [supported version of Java](#), you must update the project's pom.xml file.
 - The `JAVA_HOME` environment variable must be set to the install location of the correct version of the JDK.
- [Apache Maven 3.8.x](#)

- [Node.js 20](#)
- [PowerShell 7.2](#)
- [.NET 6.0 SDK](#)
- [Python 3.11](#).
- A [secure HTTP test tool](#) for sending requests with JSON payloads to your function endpoints. This article uses `curl`.

Initialize the project

You can use the `azd init` command to create a local Azure Functions code project from a template.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

```
Console  
  
azd init --template functions-quickstart-dotnet-azd -e flexquickstart-dotnet
```

This command pulls the project files from the [template repository](#) and initializes the project in the current folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Run this command to navigate to the `http` app folder:

```
Console  
  
cd http
```

3. Create a file named `local.settings.json` in the `http` folder that contains this JSON data:

```
JSON  
  
{  
    "IsEncrypted": false,  
    "Values": {
```

```
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
    }
}
```

This file is required when running locally.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

Console

```
azd init --template azure-functions-java-flex-consumption-azd -e
flexquickstart-java
```

This command pulls the project files from the [template repository](#) and initializes the project in the current folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Run this command to navigate to the `http` app folder:

Console

```
cd http
```

3. Create a file named `local.settings.json` in the `http` folder that contains this JSON data:

JSON

```
{
    "IsEncrypted": false,
    "Values": {
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",
        "FUNCTIONS_WORKER_RUNTIME": "java"
    }
}
```

This file is required when running locally.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

Console

```
azd init --template functions-quickstart-javascript-azd -e flexquickstart-js
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Create a file named `local.settings.json` in the root folder that contains this JSON data:

JSON

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "node"  
    }  
}
```

This file is required when running locally.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

Console

```
azd init --template functions-quickstart-powershell-azd -e flexquickstart-ps
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Run this command to navigate to the `src` app folder:

Console

```
cd src
```

3. Create a file named `local.settings.json` in the `src` folder that contains this JSON data:

JSON

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "powershell",  
        "FUNCTIONS_WORKER_RUNTIME_VERSION": "7.2"  
    }  
}
```

This file is required when running locally.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

Console

```
azd init --template functions-quickstart-typescript-azd -e  
flexquickstart-ts
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Create a file named `local.settings.json` in the root folder that contains this JSON data:

JSON

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "FUNCTIONS_WORKER_RUNTIME": "node"  
    }  
}
```

This file is required when running locally.

1. In your local terminal or command prompt, run this `azd init` command in an empty folder:

Console

```
azd init --template functions-quickstart-python-http-azd -e flexquickstart-py
```

This command pulls the project files from the [template repository](#) and initializes the project in the root folder. The `-e` flag sets a name for the current environment. In `azd`, the environment is used to maintain a unique deployment context for your app, and you can define more than one. It's also used in the name of the resource group you create in Azure.

2. Create a file named `local.settings.json` in the root folder that contains this JSON data:

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "python"
  }
}
```

This file is required when running locally.

Create and activate a virtual environment

In the root folder, run these commands to create and activate a virtual environment named `.venv`:

Linux/macOS

Bash

```
python3 -m venv .venv
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

```
Bash
```

```
sudo apt-get install python3-venv
```

Run in your local environment

1. Run this command from your app folder in a terminal or command prompt:

```
Console
```

```
func start
```

```
Console
```

```
mvn clean package  
mvn azure-functions:run
```

```
Console
```

```
npm start
```

When the Functions host starts in your local project folder, it writes the URL endpoints of your HTTP triggered functions to the terminal output.

2. In your browser, navigate to the `httpget` endpoint, which should look like this URL:

<http://localhost:7071/api/httpget>

3. From a new terminal or command prompt window, run this `curl` command to send a POST request with a JSON payload to the `httppost` endpoint:

```
Console
```

```
curl -i http://localhost:7071/api/httppost -H "Content-Type: text/json"  
-d @testdata.json
```

This command reads JSON payload data from the `testdata.json` project file. You can find examples of both HTTP requests in the `test.http` project file.

4. When you're done, press Ctrl+C in the terminal window to stop the `func.exe` host process.

5. Run `deactivate` to shut down the virtual environment.

Review the code (optional)

You can review the code that defines the two HTTP trigger function endpoints:

httpget

C#

```
[Function("httpget")]
    public IActionResult
Run([HttpTrigger(AuthorizationLevel.Function, "get")]
    HttpRequest req,
    string name)
{
    var returnValue = string.IsNullOrEmpty(name)
        ? "Hello, World."
        : $"Hello, {name}.";

    _logger.LogInformation($"C# HTTP trigger function processed
a request for {returnValue}.");

    return new OkObjectResult(returnValue);
}
```

Java

```
@FunctionName("httpget")
public HttpResponseMessage run(
    @HttpTrigger(
        name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.FUNCTION)
    HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context) {
    context.getLogger().info("Java HTTP trigger processed a request.");

    // Parse query parameter
    String name =
Optional.ofNullable(request.getQueryParameters().get("name")).orElse("Wo
rld");

    return request.createResponseBuilder(HttpStatus.OK).body("Hello, " +
```

```
name).build();
}
```

JavaScript

```
const { app } = require('@azure/functions');

app.http('httpGetFunction', {
    methods: ['GET'],
    authLevel: 'function',
    handler: async (request, context) => {
        context.log(`Http function processed request for url
"${request.url}"`);

        const name = request.query.get('name') || await request.text()
        || 'world';

        return { body: `Hello, ${name}!` };
    }
});
```

TypeScript

```
import { app, HttpRequest, HttpResponseInit, InvocationContext } from
"@azure/functions";

export async function httpGetFunction(request: HttpRequest, context:
InvocationContext): Promise<HttpResponseInit> {
    context.log(`Http function processed request for url
"${request.url}"`);

    const name = request.query.get('name') || await request.text() ||
    'world';

    return { body: `Hello, ${name}!` };
}

app.http('httpget', {
    methods: ['GET'],
    authLevel: 'function',
    handler: httpGetFunction
});
```

This `function.json` file defines the `httpget` function:

JSON

```
{
    "bindings": [
        {
```

```
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
        "get"
    ],
},
{
    "type": "http",
    "direction": "out",
    "name": "Response"
}
]
```

This `run.ps1` file implements the function code:

PowerShell

```
using namespace System.Net

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Write to the Azure Functions log stream.
Write-Host "PowerShell HTTP trigger function processed a request."

# Interact with query parameters
$name = $Request.Query.name

$body = "This HTTP triggered function executed successfully. Pass a name
in the query string for a personalized response."

if ($name) {
    $body = "Hello, $name. This HTTP triggered function executed
successfully."
}

# Associate values to output bindings by calling 'Push-OutputBinding'.
Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = [HttpStatusCode]::OK
    Body = $body
})
```

Python

```
@app.route(route="httpget", methods=["GET"])
def http_get(req: func.HttpRequest) -> func.HttpResponse:
    name = req.params.get("name", "World")

    logging.info(f"Processing GET request. Name: {name}")
```

```
return func.HttpResponse(f"Hello, {name}!")
```

You can review the complete template project [here ↗](#).

You can review the complete template project [here ↗](#).

You can review the complete template project [here ↗](#).

You can review the complete template project [here ↗](#).

You can review the complete template project [here ↗](#).

You can review the complete template project [here ↗](#).

After you verify your functions locally, it's time to publish them to Azure.

Create Azure resources

This project is configured to use the `azd provision` command to create a function app in a Flex Consumption plan, along with other required Azure resources.

ⓘ Note

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

The `azd up` and `azd deploy` commands aren't currently supported for Java apps.

1. In the root folder of the project, run this command to create the required Azure resources:

Console

```
azd provision
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

Parameter	Description
<i>Azure subscription</i>	Subscription in which your resources are created.
<i>Azure location</i>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd provision` command uses your response to these prompts with the Bicep configuration files to create and configure these required Azure resources:

- Flex Consumption plan and function app
- Azure Storage (required) and Application Insights (recommended)
- Access policies and roles for your account
- Service-to-service connections using managed identities (instead of stored connection strings)
- Virtual network to securely run both the function app and the other Azure resources

After the command completes successfully, you can deploy your project code to this new function app in Azure.

Deploy to Azure

You can use Core Tools to package your code and deploy it to Azure from the `target` output folder.

1. Navigate to the app folder equivalent in the `target` output folder:

```
Console
cd http/target/azure-functions/contoso-functions
```

This folder should have a `host.json` file, which indicates that it's the root of your compiled Java function app.

2. Run these commands to deploy your compiled Java code project to the new function app resource in Azure using Core Tools:

```
bash
```

Bash

```
APP_NAME=$(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp publish $APP_NAME
```

The `azd env get-value` command gets your function app name from the local environment, which is required for deployment using `func azure functionapp publish`. After publishing completes successfully, you see links to the HTTP trigger endpoints in Azure.

Deploy to Azure

This project is configured to use the `azd up` command to deploy this project to a new function app in a Flex Consumption plan in Azure.

Tip

This project includes a set of Bicep files that `azd` uses to create a secure deployment to a Flex consumption plan that follows best practices.

1. Run this command to have `azd` create the required Azure resources in Azure and deploy your code project to the new function app:

Console

```
azd up
```

The root folder contains the `azure.yaml` definition file required by `azd`.

If you aren't already signed-in, you're asked to authenticate with your Azure account.

2. When prompted, provide these required deployment parameters:

[\[+\] Expand table](#)

Parameter	Description
<code>Azure subscription</code>	Subscription in which your resources are created.

Parameter	Description
<i>Azure location</i>	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

The `azd up` command uses your response to these prompts with the Bicep configuration files to complete these deployment tasks:

- Create and configure these required Azure resources (equivalent to `azd provision`):
 - Flex Consumption plan and function app
 - Azure Storage (required) and Application Insights (recommended)
 - Access policies and roles for your account
 - Service-to-service connections using managed identities (instead of stored connection strings)
 - Virtual network to securely run both the function app and the other Azure resources
- Package and deploy your code to the deployment container (equivalent to `azd deploy`). The app is then started and runs in the deployed package.

After the command completes successfully, you see links to the resources you created.

Invoke the function on Azure

You can now invoke your function endpoints in Azure by making HTTP requests to their URLs using your HTTP test tool or from the browser (for GET requests). When your functions run in Azure, access key authorization is enforced, and you must provide a function access key with your request.

You can use the Core Tools to obtain the URL endpoints of your functions running in Azure.

1. In your local terminal or command prompt, run these commands to get the URL endpoint values:

```
bash
```

Bash

```
SET APP_NAME=(azd env get-value AZURE_FUNCTION_NAME)
func azure functionapp list-functions $APP_NAME --show-keys
```

PowerShell

```
$APP_NAME = azd env get-value AZURE_FUNCTION_NAME
func azure functionapp list-functions $APP_NAME --show-keys
```

The `azd env get-value` command gets your function app name from the local environment. Using the `--show-keys` option with `func azure functionapp list-functions` means that the returned **Invoke URL:** value for each endpoint includes a function-level access key.

2. As before, use your HTTP test tool to validate these URLs in your function app running in Azure.

Redeploy your code

You can run the `azd up` command as many times as you need to both provision your Azure resources and deploy code updates to your function app.

 **Note**

Deployed code files are always overwritten by the latest deployment package.

Your initial responses to `azd` prompts and any environment variables generated by `azd` are stored locally in your named environment. Use the `azd env get-values` command to review all of the variables in your environment that were used when creating Azure resources.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs:

Console

```
azd down --no-prompt
```

Note

The `--no-prompt` option instructs `azd` to delete your resource group without a confirmation from you.

This command doesn't affect your local code project.

Related content

- [Flex Consumption plan](#)
- [Azure Developer CLI \(azd\)](#)
- [azd reference](#)
- [Azure Functions Core Tools reference](#)
- [Code and test Azure Functions locally](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a Go or Rust function in Azure using Visual Studio Code

Article • 07/18/2024

In this article, you use Visual Studio Code to create a [custom handler](#) function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

Custom handlers can be used to create functions in any language or runtime by running an HTTP server process. This article supports both [Go](#) and [Rust](#).

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Configure your environment

Before you get started, make sure you have the following requirements in place:

Go

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Visual Studio Code ↗](#) on one of the [supported platforms ↗](#).
- The [Azure Functions extension ↗](#) for Visual Studio Code.
- [Go ↗](#), latest version recommended. Use the `go version` command to check your version.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions custom handlers project. Later in this article, you'll publish your function code to Azure.

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create New Project...`.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[+] Expand table

Prompt	Selection
Select a language for your function project	Choose <code>Custom Handler</code> .
Select a template for your project's first function	Choose <code>HTTP trigger</code> .
Provide a function name	Type <code>HttpExample</code> .
Authorization level	Choose <code>Anonymous</code> , which enables anyone to call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Choose <code>Open in current window</code> .

Using this information, Visual Studio Code generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer.

Create and build your function

The `function.json` file in the `HttpExample` folder declares an HTTP trigger function. You complete the function by adding a handler and compiling it into an executable.

Go

1. Press **Ctrl + N** (**Cmd + N** on macOS) to create a new file. Save it as *handler.go* in the function app root (in the same folder as *host.json*).
2. In *handler.go*, add the following code and save the file. This is your Go custom handler.

Go

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    message := "This HTTP triggered function executed successfully.
    Pass a name in the query string for a personalized response.\n"
    name := r.URL.Query().Get("name")
    if name != "" {
        message = fmt.Sprintf("Hello, %s. This HTTP triggered
        function executed successfully.\n", name)
    }
    fmt.Fprint(w, message)
}

func main() {
    listenAddr := ":8080"
    if val, ok := os.LookupEnv("FUNCTIONS_CUSTOMHANDLER_PORT"); ok
    {
        listenAddr = ":" + val
    }
    http.HandleFunc("/api/HttpExample", helloHandler)
    log.Printf("About to listen on %s. Go to https://127.0.0.1%s/",
    listenAddr, listenAddr)
    log.Fatal(http.ListenAndServe(listenAddr, nil))
}
```

3. Press **Ctrl + Shift + `** or select *New Terminal* from the *Terminal* menu to open a new integrated terminal in VS Code.
4. Compile your custom handler using the following command. An executable file named `handler` (`handler.exe` on Windows) is output in the function app root folder.

The screenshot shows a VS Code interface with the following details:

- Explorer View:** Shows a folder named "20201122-TEST-GO" containing ".vscode", "HttpExample", "handler.go", "host.json", "local.settings.json", and "proxies.json".
- Editor View:** Displays the file "handler.go" with the following code:

```

1 package main
2
3 import (
4     "fmt"
5     "log"
6     "net/http"
7     "os"
8 )
9
10 func helloHandler(w http.ResponseWriter, r *http.Request) {
11     name := r.URL.Query().Get("name")
12     if name == "" {
13         w.Write([]byte("This HTTP triggered function executed successfully. Pass a query parameter named 'name'"))
14     } else {
15         w.Write([]byte("Hello, " + name + ". This HTTP triggered function executed"))
16     }

```
- Terminal View:** Shows the command "go build handler.go" being run in the terminal, with the output "antchu-macbook:20201122-test-go antchu\$ go build handler.go" and "antchu-macbook:20201122-test-go antchu\$".

Configure your function app

The function host needs to be configured to run your custom handler binary when it starts.

1. Open `host.json`.
2. In the `customHandler.description` section, set the value of `defaultExecutablePath` to `handler` (on Windows, set it to `handler.exe`).
3. In the `customHandler` section, add a property named `enableForwardingHttpRequest` and set its value to `true`. For functions consisting of only an HTTP trigger, this setting simplifies programming by allow you to work with a typical HTTP request instead of the custom handler `request payload`.
4. Confirm the `customHandler` section looks like this example. Save the file.

```

"customHandler": {
    "description": {
        "defaultExecutablePath": "handler",
        "workingDirectory": "",
        "arguments": []
    },
}

```

```
        "enableForwardingHttpRequest": true  
    }
```

The function app is configured to start your custom handler executable.

Run the function locally

You can run this project on your local development computer before you publish to Azure.

1. In the integrated terminal, start the function app using Azure Functions Core Tools.

```
Bash
```

```
func start
```

2. With Core Tools running, navigate to the following URL to execute a GET request, which includes `?name=Functions` query string.

```
http://localhost:7071/api/HttpExample?name=Functions
```

3. A response is returned, which looks like the following in a browser:



4. Information about the request is shown in **Terminal** panel.

```
[1/30/2020 7:26:15 PM] Executing HTTP request: {
[1/30/2020 7:26:15 PM]   "requestId": "6660fd29-2b0d-41fc-9a17-a4f700415a84",
[1/30/2020 7:26:15 PM]   "method": "GET",
[1/30/2020 7:26:15 PM]   "uri": "/api/HttpExample"
[1/30/2020 7:26:15 PM] }
[1/30/2020 7:26:15 PM] Executing 'Functions.HttpExample' (Reason='This function was programmatically called via
a the host APIs.', Id=65d05c7f-5192-4ff2-a1c6-d8b3a78385d3)
[1/30/2020 7:26:15 PM] JavaScript HTTP trigger function processed a request.
[1/30/2020 7:26:15 PM] Executed 'Functions.HttpExample' (Succeeded, Id=65d05c7f-5192-4ff2-a1c6-d8b3a78385d3)
[1/30/2020 7:26:15 PM] Executed HTTP request: {
[1/30/2020 7:26:15 PM]   "requestId": "6660fd29-2b0d-41fc-9a17-a4f700415a84",
[1/30/2020 7:26:15 PM]   "method": "GET",
[1/30/2020 7:26:15 PM]   "uri": "/api/HttpExample",
[1/30/2020 7:26:15 PM]   "identities": [
[1/30/2020 7:26:15 PM]     {
[1/30/2020 7:26:15 PM]       "type": "WebJobsAuthLevel",
[1/30/2020 7:26:15 PM]       "level": "Admin"
[1/30/2020 7:26:15 PM]     }
[1/30/2020 7:26:15 PM]   ],
[1/30/2020 7:26:15 PM]   "status": 200,
[1/30/2020 7:26:15 PM]   "duration": 39
[1/30/2020 7:26:15 PM] }
```

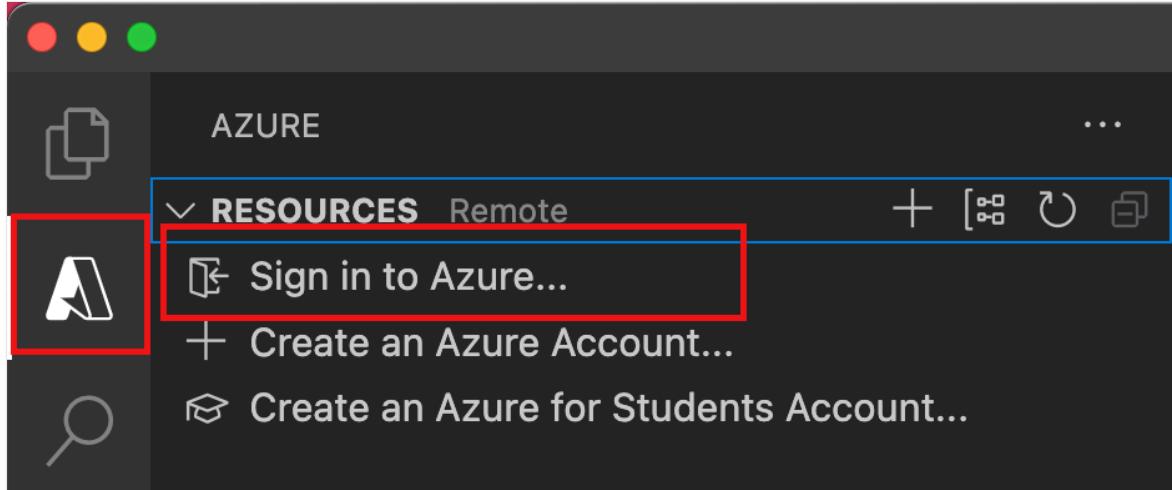
5. Press `ctrl + c` to stop Core Tools.

After you've verified that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



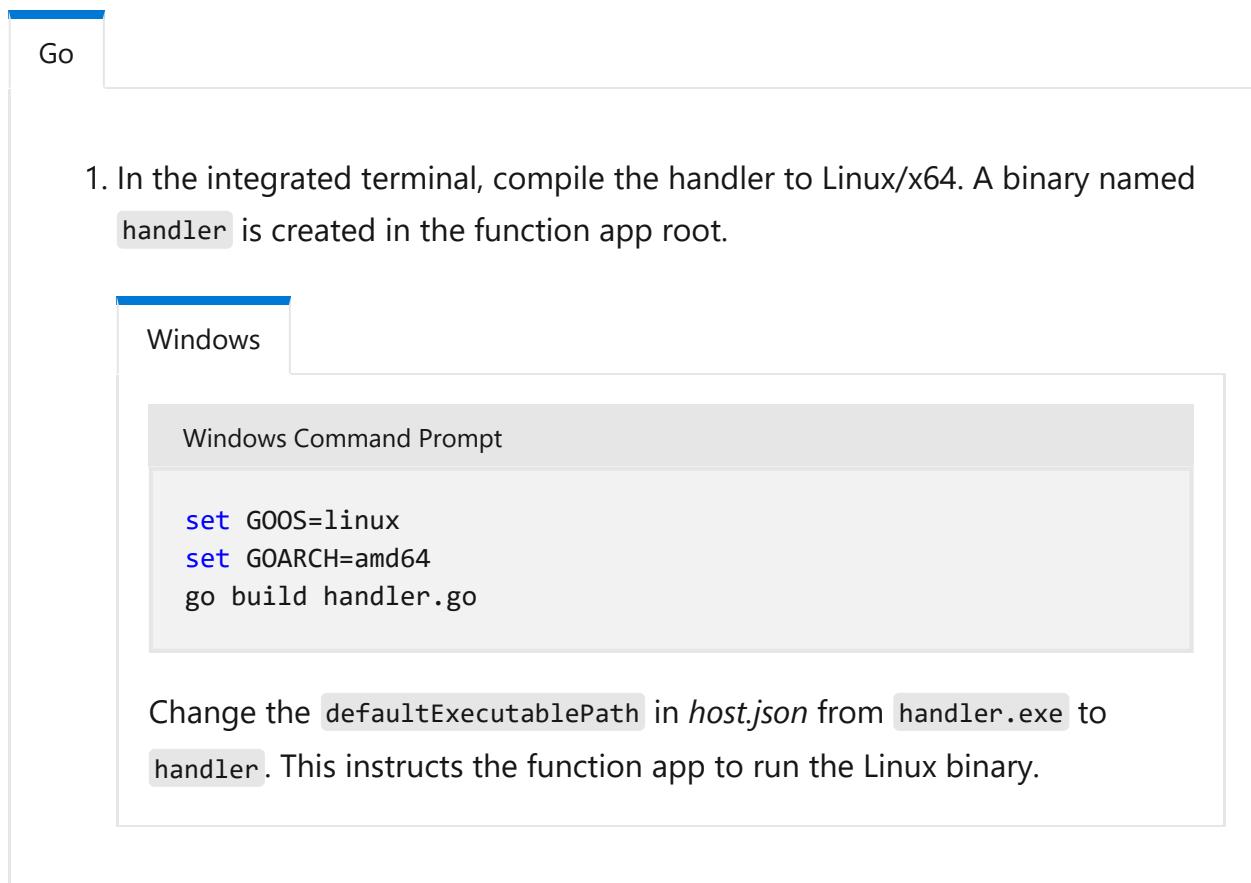
If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

- When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.

3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Compile the custom handler for Azure

In this section, you publish your project to Azure in a function app running Linux. In most cases, you must recompile your binary and adjust your configuration to match the target platform before publishing it to Azure.



The screenshot shows the Azure Functions developer tools interface. At the top, there's a navigation bar with a blue bar on the left and a 'Go' button on the right. Below this is a main area with a 'Windows' tab selected. A 'Windows Command Prompt' window is open, showing the following command sequence:

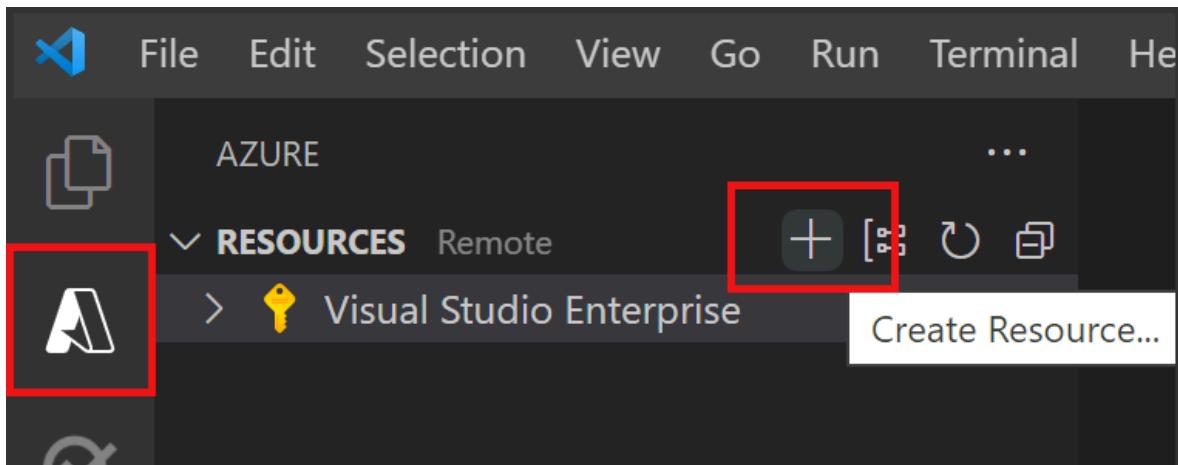
```
set GOOS=linux
set GOARCH=amd64
go build handler.go
```

Below the terminal window, there's a note: "Change the `defaultExecutablePath` in `host.json` from `handler.exe` to `handler`. This instructs the function app to run the Linux binary."

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription.

1. Choose the Azure icon in the Activity bar. Then in the **Resources** area, select the + icon and choose the **Create Function App in Azure** option.

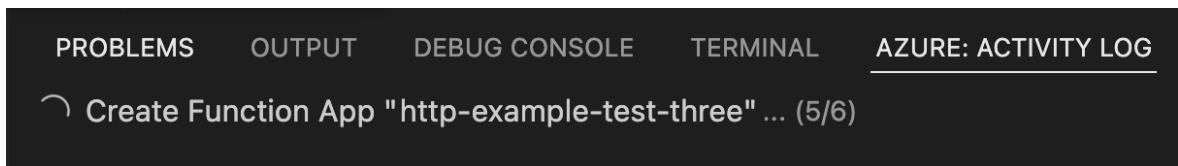


2. Provide the following information at the prompts:

[+] Expand table

Prompt	Selection
Select subscription	Choose the subscription to use. You won't see this when you have only one subscription visible under Resources.
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Choose Custom Handler.
Select a location for new resources	For better performance, choose a region near you.

The extension shows the status of individual resources as they are being created in Azure in the **Azure: Activity Log** panel.



3. When the creation is complete, the following Azure resources are created in your subscription. The resources are named based on your function app name:

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier

management, deployment, and sharing of resources within the same hosting plan.

- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

 **Tip**

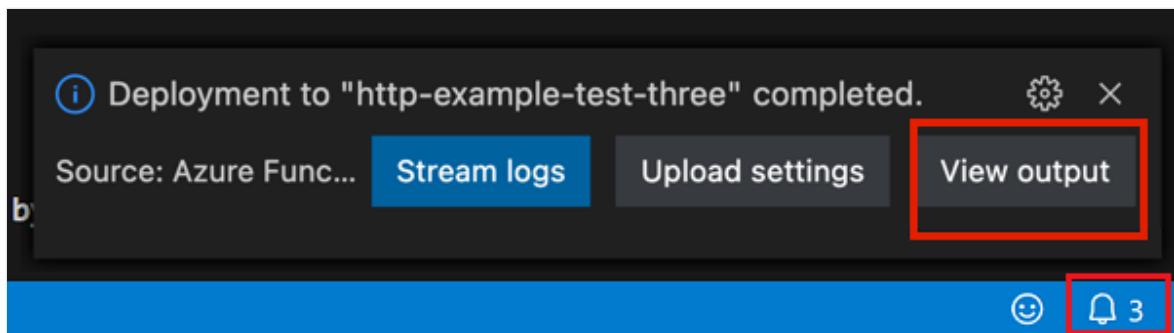
By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

 **Important**

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Press `F1` to display the command palette, then search for and run the command `Azure Functions:Execute Function Now....`. If prompted, select your subscription.
2. Select your new function app resource and `HttpExample` as your function.
3. In `Enter request body` type `{ "name": "Azure" }`, then press Enter to send this request message to your function.
4. When the function executes in Azure, the response is displayed in the notification area. Expand the notification to review the full response.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

myfunctionapp
App Service

Search (Ctrl+ /)

Overview (highlighted)
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Security

Functions

Metrics (highlighted)
Features (8)
Notifications (0)
Quickstart

Resource group (change)
myResourceGroup

Status
Running
Location
Central US
Subscription (change)
Visual Studio Enterprise
Subscription ID
11111111-1111-1111-1111-111111111111
Tags (change)
Click here to add tags

URL
<https://myfunctionapp.azurewebsites.net>
Operating System
Windows
App Service Plan
[ASP-myResourceGroup-a285 \(Y1: 0\)](#)
Properties
[See More](#)
Runtime version
3.0.13139.0

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

[Learn about Azure Functions custom handlers](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | Get help at Microsoft Q&A

Quickstart: Create and deploy Azure Functions resources using Bicep

Article • 04/05/2023

In this article, you use Azure Functions with Bicep to create a function app and related resources in Azure. The function app provides an execution context for your function code executions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Bicep is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It provides concise syntax, reliable type safety, and support for code reuse. Bicep offers the best authoring experience for your infrastructure-as-code solutions in Azure.

After you create the function app, you can deploy Azure Functions project code to that app.

Prerequisites

Azure account

Before you begin, you must have an Azure account with an active subscription. [Create an account for free](#).

Review the Bicep file

The Bicep file used in this quickstart is from [Azure Quickstart Templates](#).

Bicep

```
@description('The name of the function app that you wish to create.')
param appName string = 'fnapp${uniqueString(resourceGroup().id)}'

@description('Storage Account type')
@allowed([
    'Standard_LRS'
    'Standard_GRS'
    'Standard_RAGRS'
])
param storageAccountType string = 'Standard_LRS'
```

```

@description('Location for all resources.')
param location string = resourceGroup().location

@description('Location for Application Insights')
param appInsightsLocation string

@description('The language worker runtime to load in the function app.')
@allowed([
    'node'
    'dotnet'
    'java'
])
param runtime string = 'node'

var functionAppName = appName
var hostingPlanName = appName
var applicationInsightsName = appName
var storageAccountName = '${uniqueString(resourceGroup().id)}azfunctions'
var functionWorkerRuntime = runtime

resource storageAccount 'Microsoft.Storage/storageAccounts@2022-05-01' = {
    name: storageAccountName
    location: location
    sku: {
        name: storageAccountType
    }
    kind: 'Storage'
    properties: {
        supportsHttpsTrafficOnly: true
        defaultToOAuthAuthentication: true
    }
}

resource hostingPlan 'Microsoft.Web/serverfarms@2021-03-01' = {
    name: hostingPlanName
    location: location
    sku: {
        name: 'Y1'
        tier: 'Dynamic'
    }
    properties: {}
}

resource functionApp 'Microsoft.Web/sites@2021-03-01' = {
    name: functionAppName
    location: location
    kind: 'functionapp'
    identity: {
        type: 'SystemAssigned'
    }
    properties: {
        serverFarmId: hostingPlan.id
        siteConfig: {
            appSettings: [

```

```

    {
        name: 'AzureWebJobsStorage'
        value:
'DefaultEndpointsProtocol=https;AccountName=${storageAccountName};EndpointS
ffix=${environment().suffixes.storage};AccountKey=${storageAccount.listKeys(
).keys[0].value}'
    }
    {
        name: 'WEBSITE_CONTENTAZUREFILECONNECTIONSTRING'
        value:
'DefaultEndpointsProtocol=https;AccountName=${storageAccountName};EndpointS
ffix=${environment().suffixes.storage};AccountKey=${storageAccount.listKeys(
).keys[0].value}'
    }
    {
        name: 'WEBSITE_CONTENTSHARE'
        value: toLower(functionAppName)
    }
    {
        name: 'FUNCTIONS_EXTENSION_VERSION'
        value: '~4'
    }
    {
        name: 'WEBSITE_NODE_DEFAULT_VERSION'
        value: '~14'
    }
    {
        name: 'APPINSIGHTS_INSTRUMENTATIONKEY'
        value: applicationInsights.properties.InstrumentationKey
    }
    {
        name: 'FUNCTIONS_WORKER_RUNTIME'
        value: functionWorkerRuntime
    }
]
ftpsState: 'FtpsOnly'
minTlsVersion: '1.2'
}
httpsOnly: true
}

resource applicationInsights 'Microsoft.Insights/components@2020-02-02' = {
    name: applicationInsightsName
    location: appInsightsLocation
    kind: 'web'
    properties: {
        Application_Type: 'web'
        Request_Source: 'rest'
    }
}

```

The following four Azure resources are created by this Bicep file:

- **Microsoft.Storage/storageAccounts**: create an Azure Storage account, which is required by Functions.
- **Microsoft.Web/serverfarms**: create a serverless Consumption hosting plan for the function app.
- **Microsoft.Web/sites**: create a function app.
- **microsoft.insights/components**: create an Application Insights instance for monitoring.

ⓘ Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

Deploy the Bicep file

1. Save the Bicep file as **main.bicep** to your local computer.
2. Deploy the Bicep file using either Azure CLI or Azure PowerShell.

CLI

Azure CLI

```
az group create --name exampleRG --location eastus
az deployment group create --resource-group exampleRG --template-file main.bicep --parameters appInsightsLocation=<app-location>
```

ⓘ Note

Replace **<app-location>** with the region for Application Insights, which is usually the same as the resource group.

When the deployment finishes, you should see a message indicating the deployment succeeded.

Validate the deployment

Use Azure CLI or Azure PowerShell to validate the deployment.

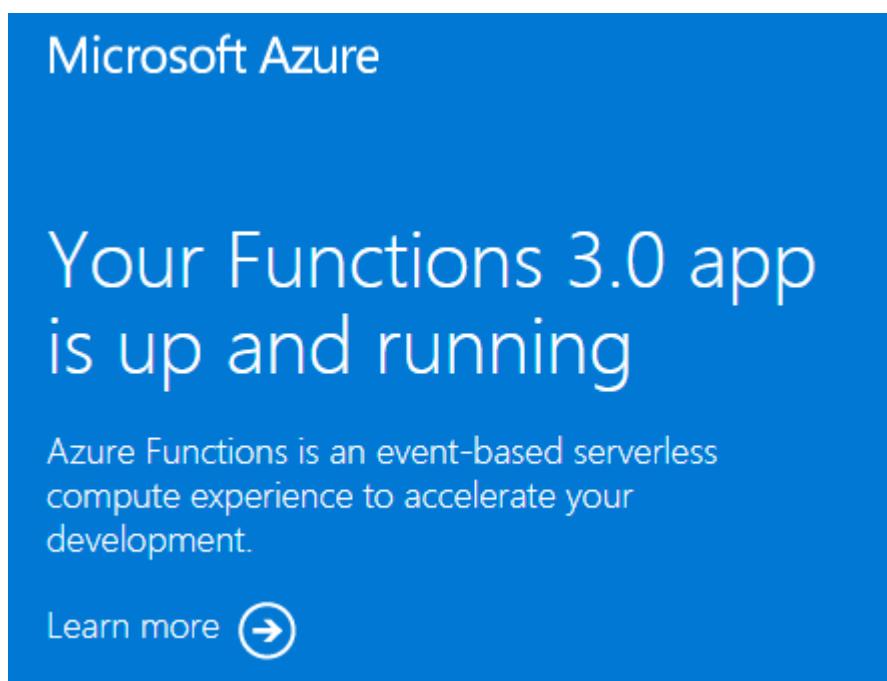
Azure CLI

```
az resource list --resource-group exampleRG
```

Visit function app welcome page

1. Use the output from the previous validation step to retrieve the unique name created for your function app.
2. Open a browser and enter the following URL:
`<https://<appName.azurewebsites.net>`. Make sure to replace `<\appName>` with the unique name created for your function app.

When you visit the URL, you should see a page like this:



Clean up resources

If you continue to the next step and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, if you no longer need the resources, use Azure CLI, PowerShell, or Azure portal to delete the resource group and its resources.

Azure CLI

```
az group delete --name exampleRG
```

Next steps

Now that you've created your function app resources in Azure, you can deploy your code to the existing app by using one of the following tools:

- [Visual Studio Code](#)
- [Visual Studio](#)
- [Azure Functions Core Tools](#)

Quickstart: Create and deploy Azure Functions resources from an ARM template

Article • 04/05/2023

In this article, you use Azure Functions with an Azure Resource Manager template (ARM template) to create a function app and related resources in Azure. The function app provides an execution context for your function code executions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

A [resource manager template](#) is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

If your environment meets the prerequisites and you're familiar with using ARM templates, select the **Deploy to Azure** button. The template will open in the Azure portal.



After you create the function app, you can deploy Azure Functions project code to that app.

Prerequisites

Azure account

Before you begin, you must have an Azure account with an active subscription. [Create an account for free](#).

Review the template

The template used in this quickstart is from [Azure Quickstart Templates](#).

JSON

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "metadata": {
        "_generator": {
            "name": "bicep",
            "version": "0.15.31.15270",
            "templateHash": "11861629922040246994"
        }
    },
    "parameters": {
        "appName": {
            "type": "string",
            "defaultValue": "[format('fnapp{0}', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "The name of the function app that you wish to create."
            }
        },
        "storageAccountType": {
            "type": "string",
            "defaultValue": "Standard_LRS",
            "allowedValues": [
                "Standard_LRS",
                "Standard_GRS",
                "Standard_RAGRS"
            ],
            "metadata": {
                "description": "Storage Account type"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for all resources."
            }
        },
        "appInsightsLocation": {
            "type": "string",
            "metadata": {
                "description": "Location for Application Insights"
            }
        },
        "runtime": {
            "type": "string",
            "defaultValue": "node",
            "allowedValues": [
                "node",
                "dotnet",
                "java"
            ],
            "metadata": {
                "description": "Runtime for the function app"
            }
        }
    }
}
```

```
    "metadata": {
        "description": "The language worker runtime to load in the function
app."
    }
},
{
    "variables": {
        "functionAppName": "[parameters('appName')]",
        "hostingPlanName": "[parameters('appName')]",
        "applicationInsightsName": "[parameters('appName')]",
        "storageAccountName": "[format('{0}azfunctions',
uniqueString(resourceGroup().id))]",
        "functionWorkerRuntime": "[parameters('runtime')]"
    },
    "resources": [
        {
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2022-05-01",
            "name": "[variables('storageAccountName')]",
            "location": "[parameters('location')]",
            "sku": {
                "name": "[parameters('storageAccountType')]"
            },
            "kind": "Storage",
            "properties": {
                "supportsHttpsTrafficOnly": true,
                "defaultToOAuthAuthentication": true
            }
        },
        {
            "type": "Microsoft.Web/serverfarms",
            "apiVersion": "2021-03-01",
            "name": "[variables('hostingPlanName')]",
            "location": "[parameters('location')]",
            "sku": {
                "name": "Y1",
                "tier": "Dynamic"
            },
            "properties": {}
        },
        {
            "type": "Microsoft.Web/sites",
            "apiVersion": "2021-03-01",
            "name": "[variables('functionAppName')]",
            "location": "[parameters('location')]",
            "kind": "functionapp",
            "identity": {
                "type": "SystemAssigned"
            },
            "properties": {
                "serverFarmId": "[resourceId('Microsoft.Web/serverfarms',
variables('hostingPlanName'))]",
                "siteConfig": {
                    "appSettings": [
                        {
                            "name": "FUNCTIONS_EXTENSION_VERSION",
                            "value": "3.x"
                        }
                    ],
                    "bindings": [
                        {
                            "type": "httpTrigger",
                            "name": "req",
                            "method": "get",
                            "route": "hello-world"
                        }
                    ]
                }
            }
        }
    ]
}
```

```

        "name": "AzureWebJobsStorage",
        "value": "[format('DefaultEndpointsProtocol=https;AccountName={0};EndpointSuffix={1};AccountKey={2}', variables('storageAccountName'), environment().suffixes.storage, listKeys(resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName')), '2022-05-01').keys[0].value)]"
    },
    {
        "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",
        "value": "[format('DefaultEndpointsProtocol=https;AccountName={0};EndpointSuffix={1};AccountKey={2}', variables('storageAccountName'), environment().suffixes.storage, listKeys(resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName')), '2022-05-01').keys[0].value)]"
    },
    {
        "name": "WEBSITE_CONTENTSHARE",
        "value": "[toLowerCase(variables('functionAppName'))]"
    },
    {
        "name": "FUNCTIONS_EXTENSION_VERSION",
        "value": "~4"
    },
    {
        "name": "WEBSITE_NODE_DEFAULT_VERSION",
        "value": "~14"
    },
    {
        "name": "APPINSIGHTS_INSTRUMENTATIONKEY",
        "value": "
[reference(resourceId('Microsoft.Insights/components', variables('applicationInsightsName')), '2020-02-02').InstrumentationKey]
",
        {
            "name": "FUNCTIONS_WORKER_RUNTIME",
            "value": "[variables('functionWorkerRuntime')]"
        }
    ],
    "ftpsState": "FtpsOnly",
    "minTlsVersion": "1.2"
},
{
    "httpsOnly": true
},
"dependsOn": [
    "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]",
    "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
]
},
{
    "type": "Microsoft.Insights/components",
    "apiVersion": "2020-02-02",
    "properties": {
        "name": "MyAppInsights"
    }
}

```

```
        "name": "[variables('applicationInsightsName')]",
        "location": "[parameters('appInsightsLocation')]",
        "kind": "web",
        "properties": {
            "Application_Type": "web",
            "Request_Source": "rest"
        }
    }
]
```

The following four Azure resources are created by this template:

- **Microsoft.Storage/storageAccounts**: create an Azure Storage account, which is required by Functions.
- **Microsoft.Web/serverfarms**: create a serverless Consumption hosting plan for the function app.
- **Microsoft.Web/sites**: create a function app.
- **microsoft.insights/components**: create an Application Insights instance for monitoring.

ⓘ Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

Deploy the template

The following scripts are designed for and tested in [Azure Cloud Shell](#). Choose Try It to open a Cloud Shell instance right in your browser.

Azure CLI

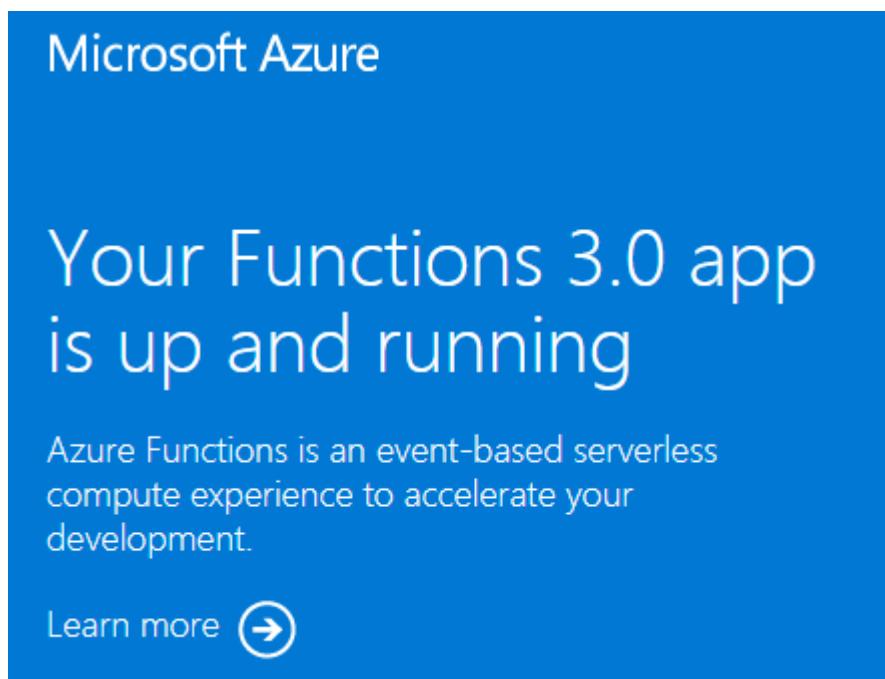
```
read -p "Enter a resource group name that is used for generating
resource names:" resourceGroupName &&
read -p "Enter the location (like 'eastus' or 'northeurope'):" location
&&
templateUri="https://raw.githubusercontent.com/Azure/azure-quickstart-
templates/master/quickstarts/microsoft.web/function-app-create-
dynamic/azuredeploy.json" &&
az group create --name $resourceGroupName --location "$location" &&
az deployment group create --resource-group $resourceGroupName --
```

```
template-uri $templateUri &&
echo "Press [ENTER] to continue ..." &&
read
```

Visit function app welcome page

1. Use the output from the previous validation step to retrieve the unique name created for your function app.
2. Open a browser and enter the following URL:
`<https://<appName.azurewebsites.net>`. Make sure to replace `<\appName>` with the unique name created for your function app.

When you visit the URL, you should see a page like this:



Clean up resources

If you continue to the next step and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete -n myResourceGroup --yes
```

```
az group delete --name <RESOURCE_GROUP_NAME>
```

Replace `<RESOURCE_GROUP_NAME>` with the name of your resource group.

Next steps

Now that you've created your function app resources in Azure, you can deploy your code to the existing app by using one of the following tools:

- [Visual Studio Code](#)
- [Visual Studio](#)
- [Azure Functions Core Tools](#)

Create your first function on Azure Arc (preview)

Article • 08/24/2023

In this quickstart, you create an Azure Functions project and deploy it to a function app running on an [Azure Arc-enabled Kubernetes cluster](#). To learn more, see [App Service, Functions, and Logic Apps on Azure Arc](#). This scenario only supports function apps running on Linux.

ⓘ Note

Support for running functions on an Azure Arc-enabled Kubernetes cluster is currently in preview.

Publishing PowerShell function projects to Azure Arc-enabled Kubernetes clusters isn't currently supported. If you need to deploy PowerShell functions to Azure Arc-enabled Kubernetes clusters, [create your function app in a container](#).

If you need to customize the container in which your function app runs, instead see [Create your first containerized functions on Azure Arc \(preview\)](#).

Prerequisites

On your local computer:

C#

- [.NET 6.0 SDK](#)
- [Azure CLI](#) version 2.4 or later

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. Visual Studio Code debugging requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

Create an App Service Kubernetes environment

Before you begin, you must [create an App Service Kubernetes environment](#) for an Azure Arc-enabled Kubernetes cluster.

Note

When you create the environment, make sure to make note of both the custom location name and the name of the resource group that contains the custom location. You can use these to find the custom location ID, which you'll need when creating your function app in the environment.

If you didn't create the environment, check with your cluster administrator.

Add Azure CLI extensions

Launch the Bash environment in [Azure Cloud Shell](#).



[Launch Cloud Shell](#)



Because these CLI commands are not yet part of the core CLI set, add them with the following commands:

Azure CLI

```
az extension add --upgrade --yes --name customlocation  
az extension remove --name appservice-kube
```

```
az extension add --upgrade --yes --name appservice-kube
```

Create the local function project

In Azure Functions, a function project is the unit of deployment and execution for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single function.

1. Run the `func init` command, as follows, to create a functions project in a folder named *LocalFunctionProj* with the specified runtime:

The screenshot shows a terminal window with a blue header bar containing the text "C#". Below the header, there are two tabs: "Console" and "Output". The "Console" tab is active and displays the command "func init LocalFunctionProj --dotnet" in white text on a dark background. The "Output" tab is visible below it but is currently empty.

2. Navigate into the project folder:

The screenshot shows a terminal window with a blue header bar containing the text "Console". Below the header, there is one tab labeled "Console". The console area displays the command "cd LocalFunctionProj" in white text on a dark background.

This folder contains various files for the project, including configurations files named `local.settings.json` and `host.json`. By default, the `local.settings.json` file is excluded from source control in the `.gitignore` file. This exclusion is because the file can contain secrets that are downloaded from Azure.

3. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (HttpExample) and the `--template` argument specifies the function's trigger (HTTP).

The screenshot shows a terminal window with a blue header bar containing the text "Console". Below the header, there is one tab labeled "Console". The console area displays the command "func new --name HttpExample --template \"HTTP trigger\" --authlevel \"anonymous\"" in white text on a dark background.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

```
Console
```

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:        4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

 **Note**

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like

`http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.

3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

Get the custom location

To be able to create a function app in a custom location, you'll need to get information about the environment.

Get the following information about the custom location from your cluster administrator (see [Create a custom location](#)).

```
Azure CLI
```

```
customLocationGroup="<resource-group-containing-custom-location>"
```

```
customLocationName="<name-of-custom-location>"
```

Get the custom location ID for the next step.

Azure CLI

```
customLocationId=$(az customlocation show \  
    --resource-group $customLocationGroup \  
    --name $customLocationName \  
    --query id \  
    --output tsv)
```

Create Azure resources

Before you can deploy your function code to your new App Service Kubernetes environment, you need to create two more resources:

- A [Storage account](#). While this article creates a storage account, in some cases a storage account may not be required. For more information, see [Azure Arc-enabled clusters](#) in the storage considerations article.
- A function app, which provides the context for executing your function code. The function app runs in the App Service Kubernetes environment and maps to your local function project. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

 **Note**

Function apps run in an App Service Kubernetes environment on a Dedicated (App Service) plan. When you create your function app without an existing plan, the correct plan is created for you.

Create Storage account

Use the [az storage account create](#) command to create a general-purpose storage account in your resource group and region:

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location westeurope --  
resource-group myResourceGroup --sku Standard_LRS
```

ⓘ Note

In some cases, a storage account may not be required. For more information, see [Azure Arc-enabled clusters in the storage considerations article](#).

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#). The `--location` value is a standard Azure region.

Create the function app

Run the `az functionapp create` command to create a new function app in the environment.

C#

Azure CLI

```
az functionapp create --resource-group MyResourceGroup --name <APP_NAME>
--custom-location <CUSTOM_LOCATION_ID> --storage-account <STORAGE_NAME>
--functions-version 4 --runtime dotnet
```

In this example, replace `<CUSTOM_LOCATION_ID>` with the ID of the custom location you determined for the App Service Kubernetes environment. Also, replace `<STORAGE_NAME>` with the name of the account you used in the previous step, and replace `<APP_NAME>` with a globally unique name appropriate to you.

Deploy the function project to Azure

After you've successfully created your function app in Azure, you're now ready to deploy your local functions project by using the `func azure functionapp publish` command.

In the following example, replace `<APP_NAME>` with the name of your app.

Console

```
func azure functionapp publish <APP_NAME>
```

The publish command shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
  HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-
qs.azurewebsites.net/api/httpexample
```

Because it can take some time for a full deployment to complete on an Azure Arc-enabled Kubernetes cluster, you may want to rerun the following command to verify your published functions:

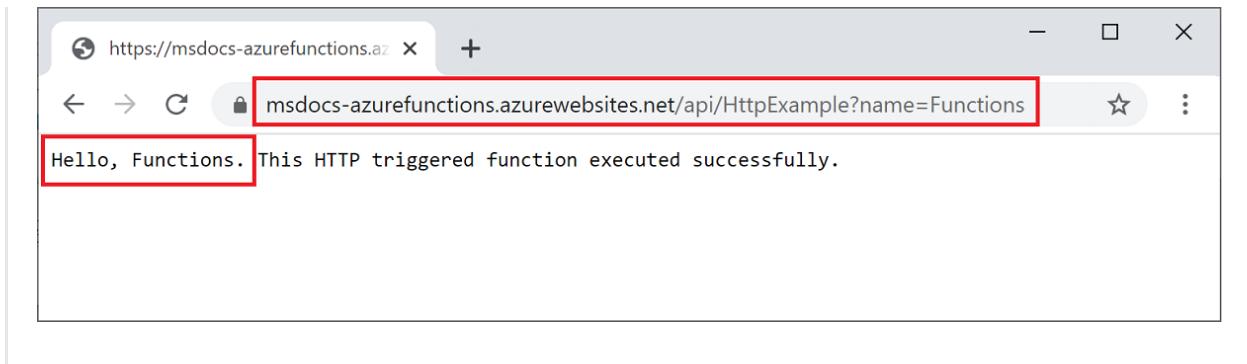
```
command
func azure functionapp list-functions
```

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.



Next steps

Now that you have your function app running in a container in an Azure Arc-enabled App Service Kubernetes environment, you can connect it to Azure Storage by adding a Queue Storage output binding.

C#

[Connect to an Azure Storage queue](#)

Create your first containerized Azure Functions on Azure Arc (preview)

Article • 06/05/2023

In this article, you create a function app running in a Linux container and deploy it to an [Azure Arc-enabled Kubernetes cluster](#) from a container registry. When you create your own container, you can customize the execution environment for your function app. To learn more, see [App Service, Functions, and Logic Apps on Azure Arc](#).

ⓘ Note

Support for deploying a custom container to an Azure Arc-enabled Kubernetes cluster is currently in preview.

You can also publish your functions to an Azure Arc-enabled Kubernetes cluster without first creating a container. To learn more, see [Create your first function on Azure Arc \(preview\)](#)

Choose your development language

First, you use Azure Functions tools to create your project code as a function app in a Docker container using a language-specific Linux base image. Make sure to select your language of choice at the top of the article.

Core Tools automatically generates a Dockerfile for your project that uses the most up-to-date version of the correct base image for your functions language. You should regularly update your container from the latest base image and redeploy from the updated version of your container. For more information, see [Creating containerized function apps](#).

Prerequisites

Before you begin, you must have the following requirements in place:

- Install the [.NET 6 SDK](#).
- Install [Azure Functions Core Tools](#) version 4.0.5198, or a later version.
- [Azure CLI](#) version 2.4 or a later version.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

To publish the containerized function app image you create to a container registry, you need a Docker ID and [Docker](#) running on your local computer. If you don't have a Docker ID, you can [create a Docker account](#).

Azure Container Registry

You also need to complete the [Create a container registry](#) section of the Container Registry quickstart to create a registry instance. Make a note of your fully qualified login server name.

Create and test the local functions project

In a terminal or command prompt, run the following command for your chosen language to create a function app project in the current folder:

Console

```
func init --worker-runtime dotnet-isolated --docker
```

The `--docker` option generates a *Dockerfile* for the project, which defines a suitable container for use with Azure Functions and the selected runtime.

Use the following command to add a function to your project, where the `--name` argument is the unique name of your function and the `--template` argument specifies the function's trigger. `func new` creates a C# code file in your project.

Console

```
func new --name HttpExample --template "HTTP trigger" --authlevel anonymous
```

To test the function locally, start the local Azure Functions runtime host in the root of the project folder.

Console

```
func start
```

After you see the `HttpExample` endpoint written to the output, navigate to that endpoint. You should see a welcome message in the response output.

Press **Ctrl+C** (**Command+C** on macOS) to stop the host.

Build the container image and verify locally

(Optional) Examine the *Dockerfile* in the root of the project folder. The *Dockerfile* describes the required environment to run the function app on Linux. The complete list of supported base images for Azure Functions can be found in the [Azure Functions base image page](#).

In the root project folder, run the [docker build](#) command, provide a name as `azurefunctionsimage`, and tag as `v1.0.0`. Replace `<DOCKER_ID>` with your Docker Hub account ID. This command builds the Docker image for the container.

Console

```
docker build --tag <DOCKER_ID>/azurefunctionsimage:v1.0.0 .
```

When the command completes, you can run the new container locally.

To verify the build, run the image in a local container using the [docker run](#) command, replace `<DOCKER_ID>` again with your Docker Hub account ID, and add the ports argument as `-p 8080:80`:

Console

```
docker run -p 8080:80 -it <DOCKER_ID>/azurefunctionsimage:v1.0.0
```

After the image starts in the local container, browse to `http://localhost:8080/api/HttpExample`, which must display the same greeting message as before. Because the HTTP triggered function you created uses anonymous authorization, you can call the function running in the container without having to obtain an access key. For more information, see [authorization keys](#).

After verifying the function app in the container, press **Ctrl+C** (**Command+C** on macOS) to stop execution.

Publish the container image to a registry

To make your container image available for deployment to a hosting environment, you must push it to a container registry.

Azure Container Registry

Azure Container Apps is a private registry service for building, storing, and managing container images and related artifacts. You should use a private registry service for publishing your containers to Azure services.

1. Use the following command to sign in to your registry instance:

Azure CLI

```
az acr login --name <REGISTRY_NAME>
```

In the previous command, replace `<REGISTRY_NAME>` with the name of your Container Registry instance.

2. Use the following command to tag your image with the fully qualified name of your registry login server:

docker

```
docker tag <DOCKER_ID>/azurefunctionsimage:v1.0.0  
<LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

Replace `<LOGIN_SERVER>` with the fully qualified name of your registry login server and `<DOCKER_ID>` with your Docker ID.

3. Use the following command to push the container to your registry instance:

docker

```
docker push <LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

4. Use the following command to enable the built-in admin account so that Functions can connect to the registry with a username and password:

Azure CLI

```
az acr update -n <REGISTRY_NAME> --admin-enabled true
```

 **Important**

The admin account is designed for a single user to access the registry, mainly for testing purposes and for specific Azure services. In a production scenario, you should instead [add a user-assigned managed identity](#) to which you can grant access to the registry.

5. Use the following command to retrieve the admin username and password, which Functions needs to connect to the registry:

Azure CLI

```
az acr credential show -n <REGISTRY_NAME> --query "[username, passwords[0].value]" -o tsv
```

 **Important**

The admin account username and password are important credentials. Make sure to store them securely and never in an accessible location like a public repository.

Create an App Service Kubernetes environment

Before you begin, you must [create an App Service Kubernetes environment](#) for an Azure Arc-enabled Kubernetes cluster.

 **Note**

When you create the environment, make sure to make note of both the custom location name and the name of the resource group that contains the custom location. You can use these to find the custom location ID, which you'll need when creating your function app in the environment.

If you didn't create the environment, check with your cluster administrator.

Add Azure CLI extensions

Launch the Bash environment in [Azure Cloud Shell](#).

 [Launch Cloud Shell](#) 

Because these CLI commands are not yet part of the core CLI set, add them with the following commands:

Azure CLI

```
az extension add --upgrade --yes --name customlocation  
az extension remove --name appservice-kube  
az extension add --upgrade --yes --name appservice-kube
```

Create Azure resources

Before you can deploy your container to your new App Service Kubernetes environment, you need to create two more resources:

- A [Storage account](#). While this article creates a storage account, in some cases a storage account may not be required. For more information, see [Azure Arc-enabled clusters](#) in the storage considerations article.
- A function app, which provides the context for running your container. The function app runs in the App Service Kubernetes environment and maps to your local function project. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

ⓘ Note

Function apps run in an App Service Kubernetes environment on a Dedicated (App Service) plan. When you create your function app without an existing plan, a plan is created for you.

Create Storage account

Use the [az storage account create](#) command to create a general-purpose storage account in your resource group and region:

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location westeurope --  
resource-group myResourceGroup --sku Standard_LRS
```

ⓘ Note

In some cases, a storage account may not be required. For more information, see [Azure Arc-enabled clusters](#) in the storage considerations article.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#). The `--location` value is a standard Azure region.

Create the function app

Run the `az functionapp create` command to create a new function app in the environment.

```
Azure Container Registry

Azure CLI

az functionapp create --name <APP_NAME> --custom-location
<CUSTOM_LOCATION_ID> --storage-account <STORAGE_NAME> --resource-group
AzureFunctionsContainers-rg --image
<LOGIN_SERVER>/azurefunctionsimage:v1.0.0 --registry-username <USERNAME>
--registry-password <SECURE_PASSWORD>
```

In this example, replace `<CUSTOM_LOCATION_ID>` with the ID of the custom location you determined for the App Service Kubernetes environment. Also, replace `<STORAGE_NAME>` with the name of the account you used in the previous step, `<APP_NAME>` with a globally unique name, and `<DOCKER_ID>` or `<LOGIN_SERVER>` with your Docker Hub account ID or Container Registry server, respectively. When you're deploying from a custom container registry, the image name indicates the URL of the registry.

When you first create the function app, it pulls the initial image from your Docker Hub.

Set required app settings

Run the following commands to create an app setting for the storage account connection string:

```
Azure CLI

storageConnectionString=$(az storage account show-connection-string --
resource-group AzureFunctionsContainers-rg --name <STORAGE_NAME> --query
connectionString --output tsv)
```

```
az functionapp config appsettings set --name <app_name> --resource-group  
AzureFunctionsContainers-rg --settings  
AzureWebJobsStorage=$storageConnectionString
```

This code must be run either in Cloud Shell or in Bash on your local computer. Replace `<STORAGE_NAME>` with the name of the storage account and `<APP_NAME>` with the function app name.

Invoke the function on Azure

Because your function uses an HTTP trigger, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `?name=Functions`. The browser should display similar output as when you ran the function locally.



Clean up resources

If you want to continue working with Azure Function using the resources you created in this article, you can leave all those resources in place.

When you are done working with this function app deployment, delete the `AzureFunctionsContainers-rg` resource group to clean up all the resources in that group:

Azure CLI

```
az group delete --name AzureFunctionsContainers-rg
```

This only removes the resources created in this article. The underlying Azure Arc environment remains in place.

Next steps

[Working with custom containers and Azure Functions](#)

Create your first containerized functions on Azure Container Apps

Article • 07/20/2024

In this article, you create a function app running in a Linux container and deploy it to an Azure Container Apps environment from a container registry. By deploying to Container Apps, you're able to integrate your function apps into cloud-native microservices. For more information, see [Azure Container Apps hosting of Azure Functions](#).

This article shows you how to create functions running in a Linux container and deploy the container to a Container Apps environment.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account, which you can minimize by [cleaning-up resources](#) when you're done.

Choose your development language

First, you use Azure Functions tools to create your project code as a function app in a Docker container using a language-specific Linux base image. Make sure to select your language of choice at the top of the article.

Core Tools automatically generates a Dockerfile for your project that uses the most up-to-date version of the correct base image for your functions language. You should regularly update your container from the latest base image and redeploy from the updated version of your container. For more information, see [Creating containerized function apps](#).

Prerequisites

Before you begin, you must have the following requirements in place:

- Install the [.NET 8.0 SDK](#).
- Install [Azure Functions Core Tools](#) version 4.0.5198, or a later version.
- [Azure CLI](#) version 2.4 or a later version.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

To publish the containerized function app image you create to a container registry, you need a Docker ID and [Docker](#) running on your local computer. If you don't have a Docker ID, you can [create a Docker account](#).

Azure Container Registry

You also need to complete the [Create a container registry](#) section of the Container Registry quickstart to create a registry instance. Make a note of your fully qualified login server name.

Create and test the local functions project

In a terminal or command prompt, run the following command for your chosen language to create a function app project in the current folder:

Console

```
func init --worker-runtime dotnet-isolated --docker
```

The `--docker` option generates a *Dockerfile* for the project, which defines a suitable container for use with Azure Functions and the selected runtime.

Use the following command to add a function to your project, where the `--name` argument is the unique name of your function and the `--template` argument specifies the function's trigger. `func new` creates a C# code file in your project.

Console

```
func new --name HttpExample --template "HTTP trigger"
```

To test the function locally, start the local Azure Functions runtime host in the root of the project folder.

Console

```
func start
```

After you see the `HttpExample` endpoint written to the output, navigate to that endpoint. You should see a welcome message in the response output.

Press **Ctrl+C** (**Command+C** on macOS) to stop the host.

Build the container image and verify locally

(Optional) Examine the *Dockerfile* in the root of the project folder. The *Dockerfile* describes the required environment to run the function app on Linux. The complete list of supported base images for Azure Functions can be found in the [Azure Functions base image page](#).

In the root project folder, run the [docker build](#) command, provide a name as `azurefunctionsimage`, and tag as `v1.0.0`. Replace `<DOCKER_ID>` with your Docker Hub account ID. This command builds the Docker image for the container.

Console

```
docker build --tag <DOCKER_ID>/azurefunctionsimage:v1.0.0 .
```

When the command completes, you can run the new container locally.

To verify the build, run the image in a local container using the [docker run](#) command, replace `<DOCKER_ID>` again with your Docker Hub account ID, and add the ports argument as `-p 8080:80`:

Console

```
docker run -p 8080:80 -it <DOCKER_ID>/azurefunctionsimage:v1.0.0
```

After the image starts in the local container, browse to

`http://localhost:8080/api/HttpExample`, which must display the same greeting message as before. Because the HTTP triggered function you created uses anonymous authorization, you can call the function running in the container without having to obtain an access key. For more information, see [authorization keys](#).

After verifying the function app in the container, press **Ctrl+C** (**Command+C** on macOS) to stop execution.

Publish the container image to a registry

To make your container image available for deployment to a hosting environment, you must push it to a container registry.

Azure Container Registry

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts. You should use a private registry service for publishing your containers to Azure services.

1. Use this command to sign in to your registry instance using your current Azure credentials:

```
Azure CLI
```

```
az acr login --name <REGISTRY_NAME>
```

In the previous command, replace `<REGISTRY_NAME>` with the name of your Container Registry instance.

2. Use this command to tag your image with the fully qualified name of your registry login server:

```
docker
```

```
docker tag <DOCKER_ID>/azurefunctionsimage:v1.0.0  
<LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

Replace `<LOGIN_SERVER>` with the fully qualified name of your registry login server and `<DOCKER_ID>` with your Docker ID.

3. Use this command to push the container to your registry instance:

```
docker
```

```
docker push <LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

Create supporting Azure resources for your function

Before you can deploy your container to Azure, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- An Azure Container Apps environment with a Log Analytics workspace.

Use the following commands to create these items.

1. If you haven't done so already, sign in to Azure.

The `az login` command signs you into your Azure account. Use `az account set` when you have more than one subscription associated with your account.

2. Run the following command to update the Azure CLI to the latest version:

```
Azure CLI
```

```
az upgrade
```

If your version of Azure CLI isn't the latest version, an installation begins. The manner of upgrade depends on your operating system. You can proceed after the upgrade is complete.

3. Run the following commands that upgrade the Azure Container Apps extension and register namespaces required by Container Apps:

```
Azure CLI
```

```
az extension add --name containerapp --upgrade -y  
az provider register --namespace Microsoft.Web  
az provider register --namespace Microsoft.App  
az provider register --namespace Microsoft.OperationalInsights
```

4. Create a resource group named `AzureFunctionsContainers-rg`.

```
Azure CLI
```

```
az group create --name AzureFunctionsContainers-rg --location eastus
```

This `az group create` command creates a resource group in the East US region. If you instead want to use a region near you, using an available region code returned from the `az account list-locations` command. You must modify subsequent commands to use your custom region instead of `eastus`.

5. Create Azure Container App environment with workload profiles enabled.

```
Azure CLI
```

```
az containerapp env create --name MyContainerappEnvironment --enable-workload-profiles --resource-group AzureFunctionsContainers-rg --location eastus
```

This command can take a few minutes to complete.

6. Create a general-purpose storage account in your resource group and region.

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location eastus --resource-group AzureFunctionsContainers-rg --sku Standard_LRS
```

The [az storage account create](#) command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Storage names must contain 3 to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account [supported by Functions](#).

Create and configure a function app on Azure with the image

A function app on Azure manages the execution of your functions in your Azure Container Apps environment. In this section, you use the Azure resources from the previous section to create a function app from an image in a container registry in a Container Apps environment. You also configure the new environment with a connection string to the required Azure Storage account.

Use the [az functionapp create](#) command to create a function app in the new managed environment backed by Azure Container Apps:

Azure Container Registry

Azure CLI

```
az functionapp create --name <APP_NAME> --storage-account <STORAGE_NAME> --environment MyContainerappEnvironment --workload-profile-name "Consumption" --resource-group AzureFunctionsContainers-rg --functions-version 4 --runtime dotnet-isolated --image <LOGIN_SERVER>/azurefunctionsimage:v1.0.0 --assign-identity
```

In the [az functionapp create](#) command, the `--environment` parameter specifies the Container Apps environment and the `--image` parameter specifies the image to use for the function app. In this example, replace `<STORAGE_NAME>` with the name you used in the previous section for the storage account. Also, replace `<APP_NAME>` with

a globally unique name appropriate to you and <LOGIN_SERVER> with your fully qualified Container Registry server.

To use a system-assigned managed identity to access the container registry, you need to enable managed identities in your app and grant the system-assigned managed identity access to the container registry. This example uses `az functionapp identity assign` and `az role assignment create` command to enable managed identities in the app and assign the system-assigned identity to the `ACRPull` role in the container registry:

Azure CLI

```
FUNCTION_APP_ID=$(az functionapp identity assign --name <APP_NAME> --resource-group AzureFunctionsContainers-rg --query principalId --output tsv)
ACR_ID=$(az acr show --name <REGISTRY_NAME> --query id --output tsv)
az role assignment create --assignee $FUNCTION_APP_ID --role AcrPull --scope $ACR_ID
```

In this example, replace `<APP_NAME>` and `<REGISTRY_NAME>` with the name of your function app and container registry, respectively.

Specifying `--workload-profile-name "Consumption"` creates your app in an environment using the default `Consumption` workload profile, which costs the same as running in a Container Apps Consumption plan. When you first create the function app, it pulls the initial image from your registry.

At this point, your functions are running in a Container Apps environment, with the required application settings already added. When needed, you can add other settings in your functions app in the standard way for Functions. For more information, see [Use application settings](#).

Tip

When you make subsequent changes to your function code, you need to rebuild the container, republish the image to the registry, and update the function app with the new image version. For more information, see [Update an image in the registry](#)

Verify your functions on Azure

With the image deployed to your function app in Azure, you can now invoke the function through HTTP requests.

1. Run the following `az functionapp function show` command to get the URL of your new function:

```
Azure CLI
```

```
az functionapp function show --resource-group AzureFunctionsContainers-rg --name <APP_NAME> --function-name HttpExample --query invokeUrlTemplate
```

Replace `<APP_NAME>` with the name of your function app.

2. Use the URL you just obtained to call the `HttpExample` function endpoint.

When you navigate to this URL, the browser must display similar output as when you ran the function locally.

The request URL should look something like this:

```
https://myfunctionapp.kindtree-  
796af82b.eastus.azurecontainerapps.io/api/httpexample
```

Clean up resources

If you want to continue working with Azure Function using the resources you created in this article, you can leave all those resources in place.

When you're done working with this function app deployment, delete the `AzureFunctionsContainers-rg` resource group to clean up all the resources in that group:

```
Azure CLI
```

```
az group delete --name AzureFunctionsContainers-rg
```

Next steps

[Azure Container Apps hosting of Azure Functions](#)

[Working with containers and Azure Functions](#)

Help make the experience better

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Connect functions to Azure Storage using Visual Studio

Article • 03/31/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio to connect the function you created in the [previous quickstart article](#) to Azure Storage. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the Storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Prerequisites

Before you start this article, you must:

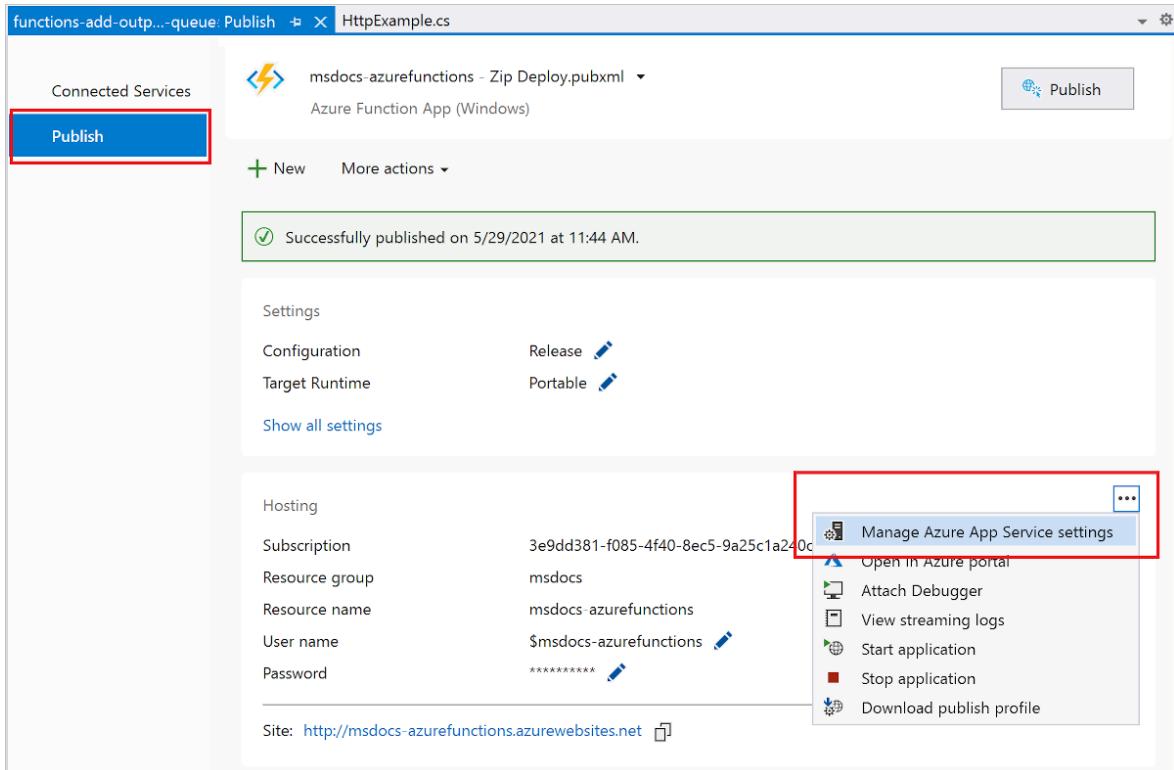
- Complete [part 1 of the Visual Studio quickstart](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Sign in to your Azure subscription from Visual Studio.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required Storage account. The connection string for this account is stored securely in app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your Storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. In **Solution Explorer**, right-click the project and select **Publish**.

2. In the Publish tab under **Hosting**, expand the three dots (...) and select **Manage Azure App Service settings**.



3. Under **AzureWebJobsStorage**, copy the **Remote** string value to **Local**, and then select **OK**.

The storage binding, which uses the `AzureWebJobsStorage` setting for the connection, can now connect to your Queue storage when running locally.

Register binding extensions

Because you're using a Queue storage output binding, you need the Storage bindings extension installed before you run the project. Except for HTTP and timer triggers, bindings are implemented as extension packages.

1. From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.
2. In the console, run the following `Install-Package` command to install the Storage extensions:

Isolated worker model

Bash

```
Install-Package  
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues
```

Now, you can add the storage output binding to your project.

Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

Isolated worker model

Open the *HttpExample.cs* project file and add the following `MultiResponse` class:

C#

```
public class MultiResponse  
{  
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]  
    public string[] Messages { get; set; }  
    public HttpResponseMessage HttpResponseMessage { get; set; }  
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpExample");
    logger.LogInformation("C# HTTP trigger function processed a
request.");

    var message = "Welcome to Azure Functions!";

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
    response.WriteString(message);

    // Return a response to both HTTP trigger and storage output
binding.
    return new MultiResponse()
    {
        // Write a single message.
        Messages = new string[] { message },
        HttpResponseMessage = response
    };
}
```

Run the function locally

1. To run your function, press `F5` in Visual Studio. You might need to enable a firewall exception so that the tools can handle HTTP requests. Authorization levels are never enforced when you run a function locally.
2. Copy the URL of your function from the Azure Functions runtime output.

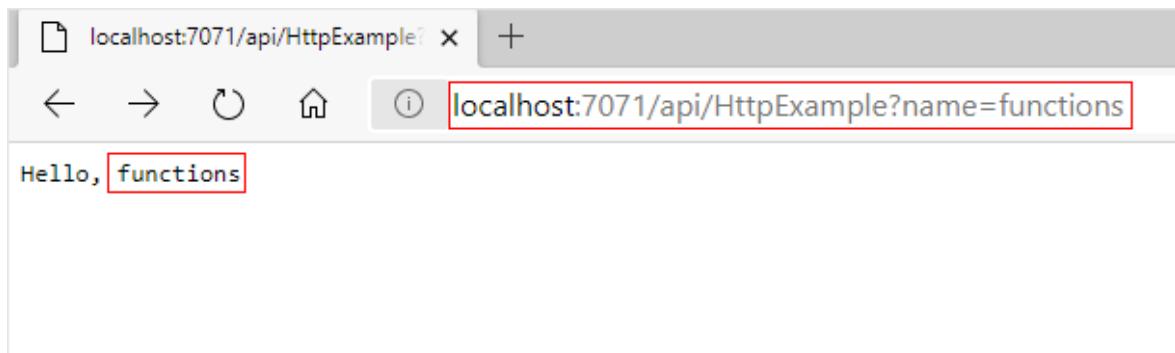
```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.49.0\cli_x64\func.exe
[5/27/2020 7:53:39 AM] Loading functions metadata
[5/27/2020 7:53:39 AM] 1 functions loaded
[5/27/2020 7:53:39 AM] Generating 1 job function(s)
[5/27/2020 7:53:40 AM] Found the following functions:
[5/27/2020 7:53:40 AM] FunctionApp2.HttpExample.Run
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Initializing function HTTP routes
[5/27/2020 7:53:40 AM] Mapped function route 'api/HttpExample' [get,post] to 'HttpExample'
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Host initialized (691ms)
[5/27/2020 7:53:40 AM] Host started (712ms)
[5/27/2020 7:53:40 AM] Job host started
Hosting environment: Development
Content root path: C:\source\repos\FunctionApp\FunctionApp\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[5/27/2020 7:53:47 AM] Host lock lease acquired by instance ID '00000000000000000000000000000000FB2CECE'.
```

- Paste the URL for the HTTP request into your browser's address bar and run the request. The following image shows the response in the browser to the local GET request returned by the function:



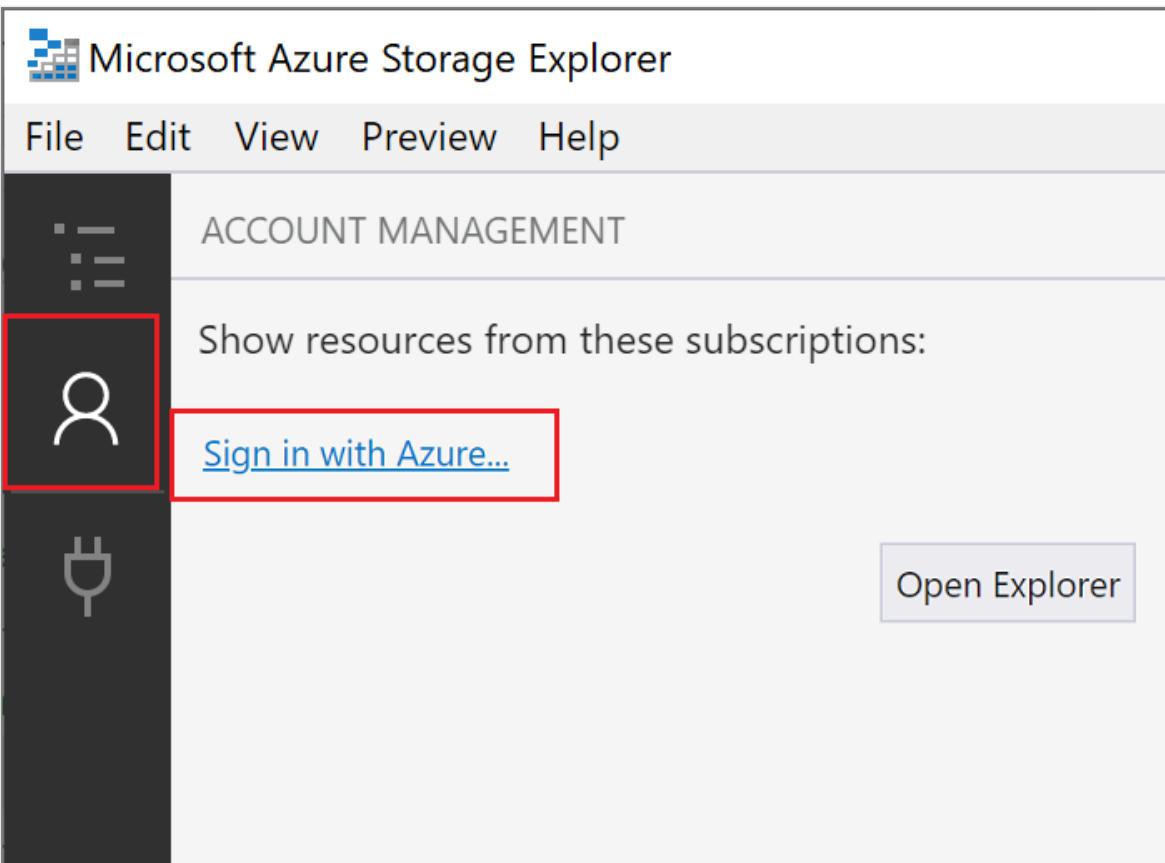
- To stop debugging, press **Shift + F5** in Visual Studio.

A new queue named `outqueue` is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

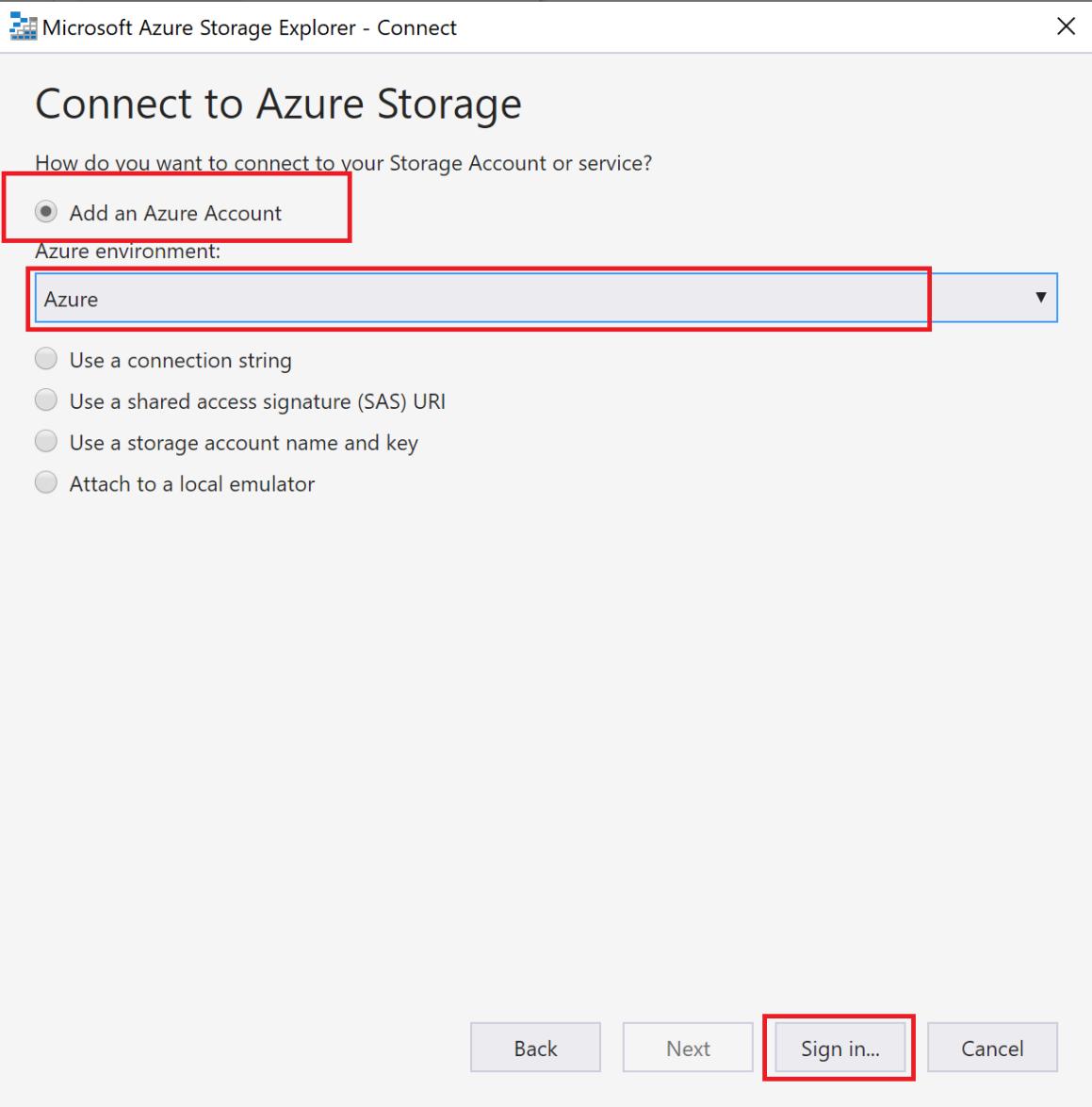
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

- Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

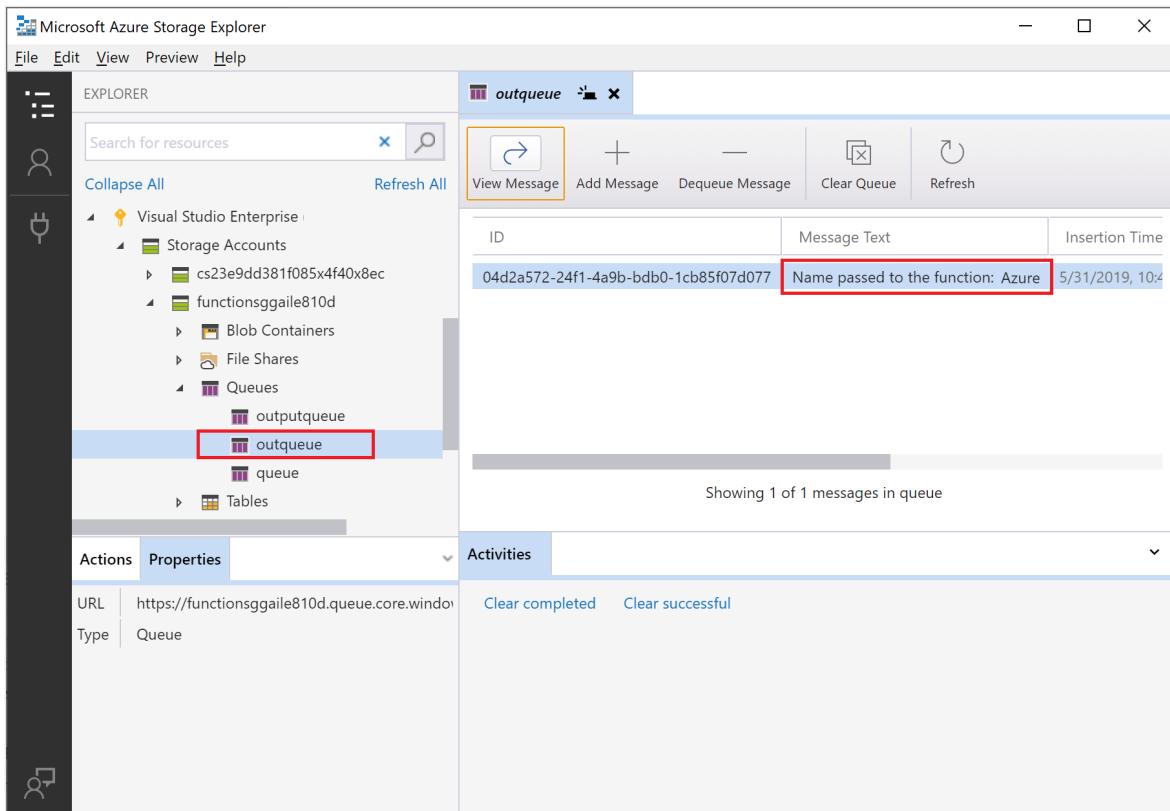


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Storage Explorer, expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default `name` value of *Azure*, the queue message is *Name passed to the function: Azure*.



- Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

- In **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
- After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL.
- Again **view the message in the storage queue** to verify that the output binding again generates a new message in the queue.

Clean up resources

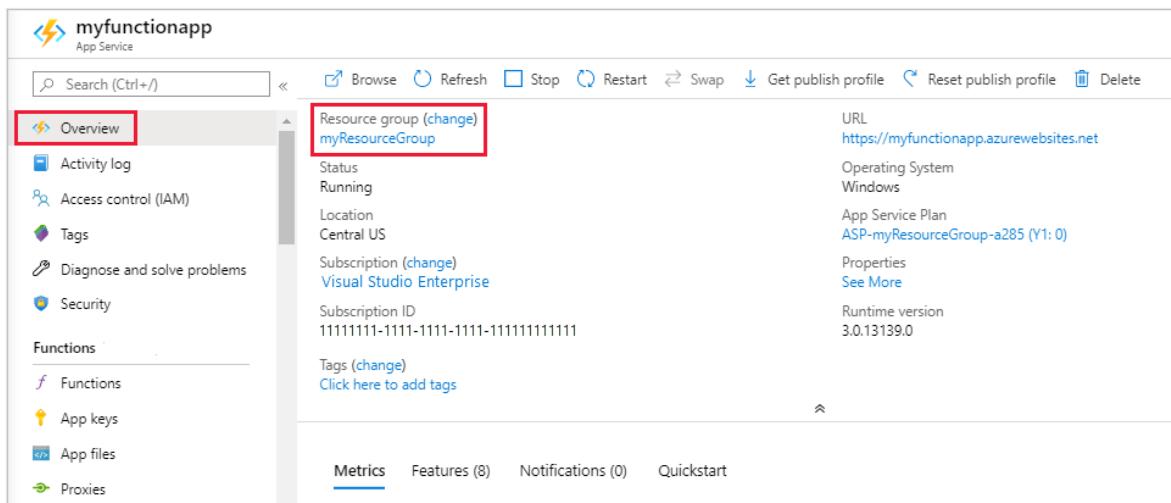
Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has a "Functions" section with "myfunctionapp" selected. The main content area is the "Overview" tab, which displays the following details:

Setting	Value
URL	https://myfunctionapp.azurewebsites.net
Operating System	Windows
App Service Plan	ASP-myResourceGroup-a285 (Y1: 0)
Properties	See More
Runtime version	3.0.13139.0

Under "Resource group (change)", it shows "myResourceGroup" with the following details:

- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 1111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. To learn more about developing Functions, see [Develop Azure Functions using Visual Studio](#).

Next, you should enable Application Insights monitoring for your function app:

[Enable Application Insights integration](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Install [.NET Core CLI tools](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings...`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Except for HTTP and timer triggers, bindings are implemented as extension packages. Run the following [dotnet add package](#) command in the Terminal window to add the Storage extension package to your project.

```
Isolated process

Bash

dotnet add package
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues --prerelease
```

Now, you can add the storage output binding to your project.

Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

Isolated worker model

Open the `HttpExample.cs` project file and add the following `MultiResponse` class:

C#

```
public class MultiResponse
{
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Isolated worker model

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
```

```

    {
        var logger = ExecutionContext.GetLogger("HttpExample");
        logger.LogInformation("C# HTTP trigger function processed a
request.");
    }

        var message = "Welcome to Azure Functions!";

        var response = req.CreateResponse(HttpStatusCode.OK);
        response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
        response.WriteString(message);

        // Return a response to both HTTP trigger and storage output
binding.
        return new MultiResponse()
    {
        // Write a single message.
        Messages = new string[] { message },
        HttpResponseMessage = response
    };
}
}

```

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure. If you don't already have Core Tools installed locally, you are prompted to install it the first time you run your project.

1. To call your function, press **F5** to start the function app project. The **Terminal** panel displays the output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG
✖ host start - Task ✓ + × ☰ 🗑 ⌂ ⌂ ×

Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

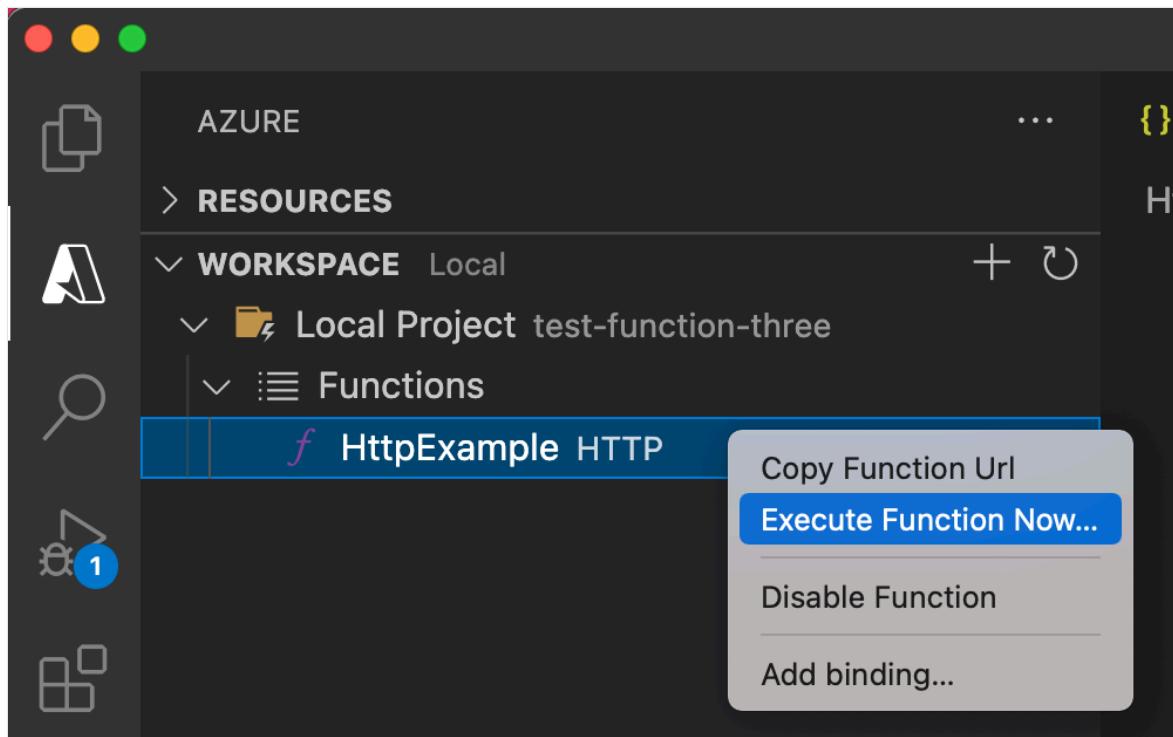
For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.

```

If you don't already have Core Tools installed, select **Install** to install Core Tools when prompted to do so.

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

- With the Core Tools running, go to the Azure: Functions area. Under Functions, expand Local Project > Functions. Right-click (Windows) or `ctrl -` click (macOS) the `HttpExample` function and choose Execute Function Now....

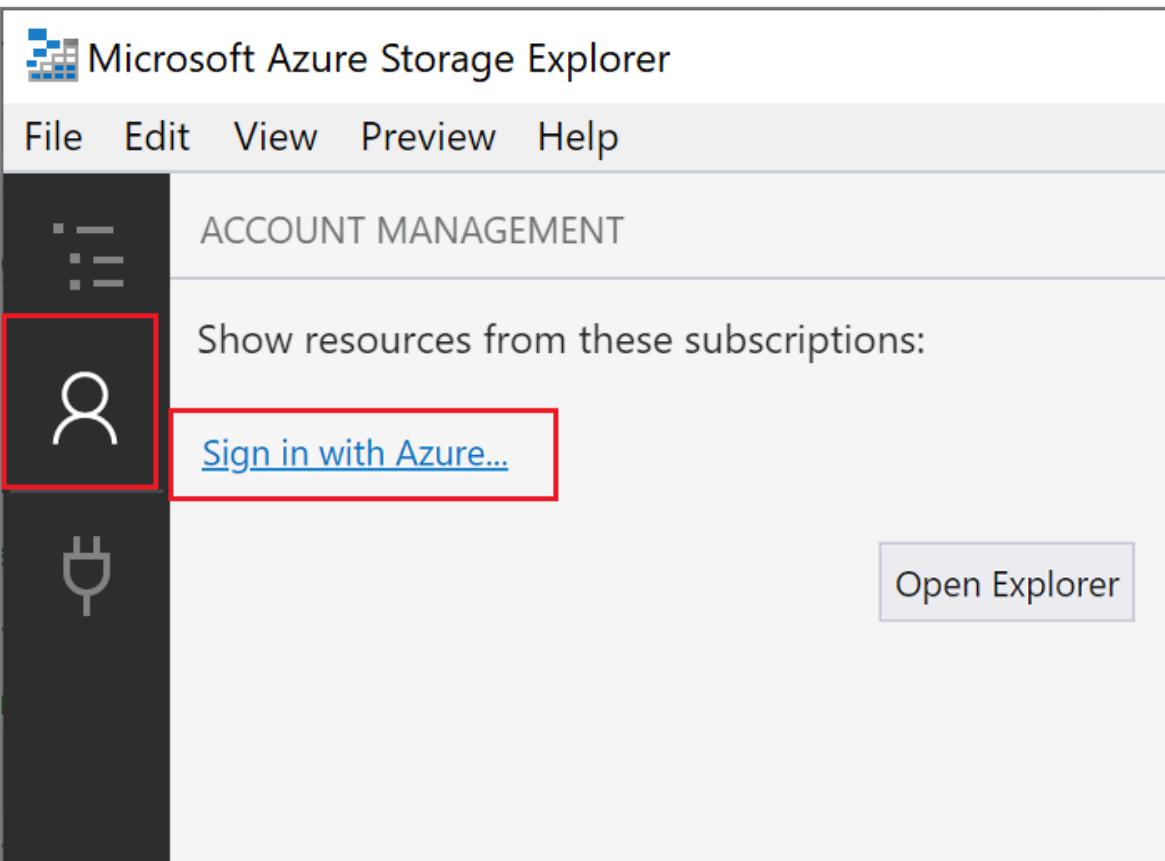


- In the Enter request body, press `Enter` to send a request message to your function.
- When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in the Terminal panel.
- Press `Ctrl + C` to stop Core Tools and disconnect the debugger.

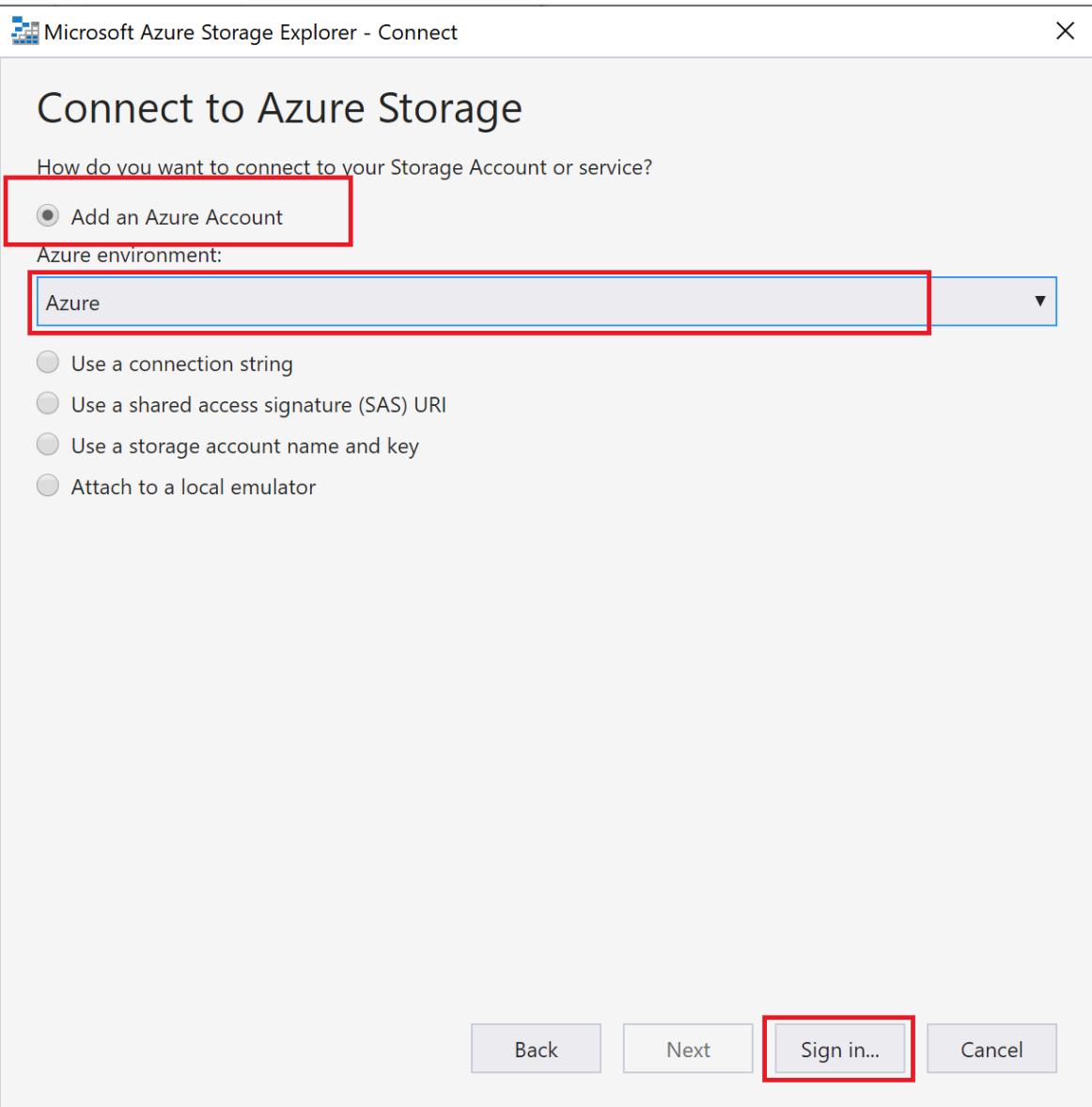
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

- Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select Add an account.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

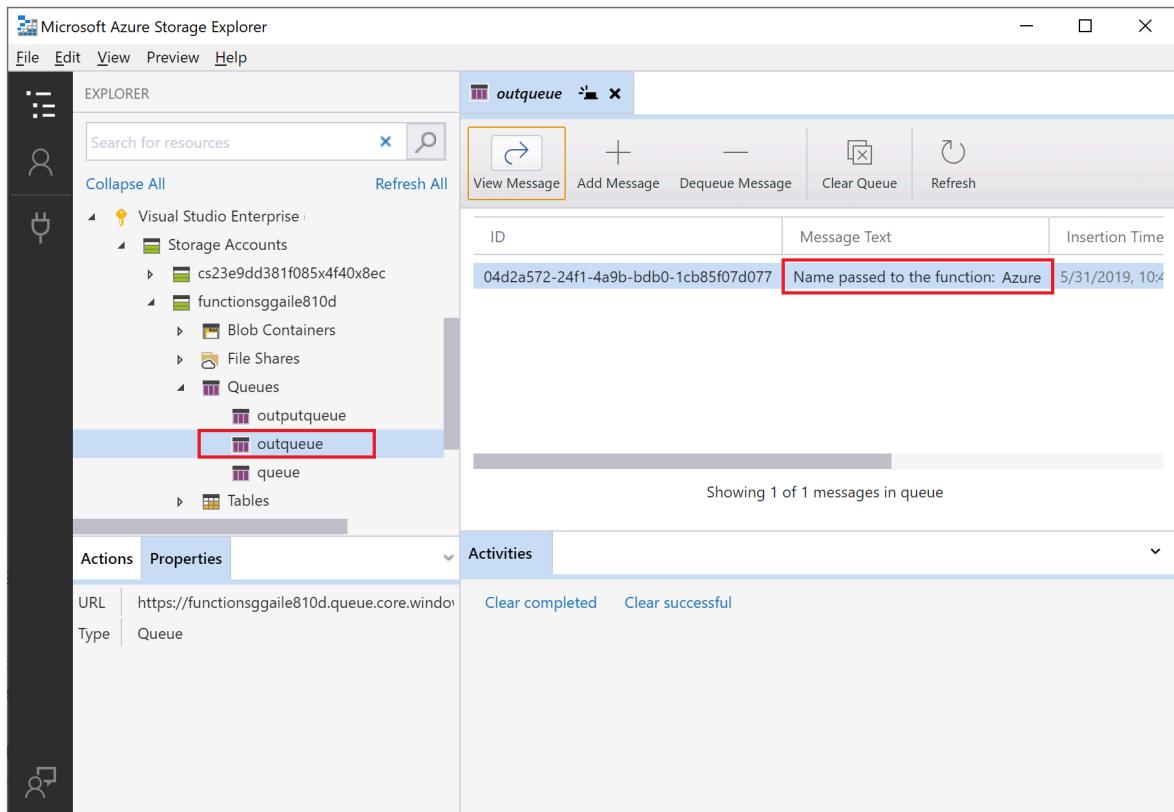


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

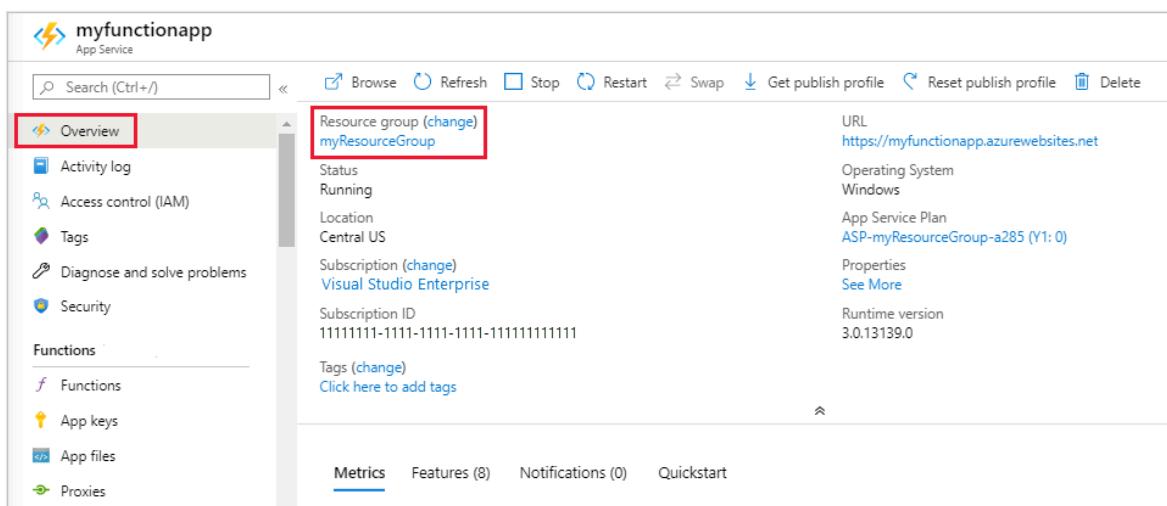
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the "Overview" tab, which displays details about the app: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (with a link to add tags). A red box highlights the "Resource group (change)" link, which points to "myResourceGroup". To the right of the main content, there are sections for URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties (with a "See More" link), and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in C#.](#)
- [Azure Functions C# developer reference](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Register binding extensions

Except for HTTP and timer triggers, bindings are implemented as extension packages. Run the following `dotnet add package` command in the Terminal window to add the Storage extension package to your project.

```
Isolated process

Bash

dotnet add package
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues --prerelease
```

Now, you can add the storage output binding to your project.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

```
Isolated worker model

C#

public class MultiResponse
{
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
```

```
    public HttpResponseMessage HttpResponse { get; set; }  
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Isolated worker model

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpExample");
    logger.LogInformation("C# HTTP trigger function processed a
request.");

    var message = "Welcome to Azure Functions!";

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
    response.WriteString(message);

    // Return a response to both HTTP trigger and storage output
binding.
    return new MultiResponse()
{
```

```
// Write a single message.  
Messages = new string[] { message },  
HttpServletResponse = response  
};  
}  
}
```

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

```
Console  
  
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools  
Core Tools Version: 4.0.5049 Commit hash: N/A (64-bit)  
Function Runtime Version: 4.15.2.20177  
  
[2023-03-17T03:27:12.372Z] Worker process started and initialized.  
  
Functions:  
  
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.

3. When you're done, press `ctrl + c` and type `y` to stop the functions host.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's `local.setting.json` file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

```
bash
Bash
export AZURE_STORAGE_CONNECTION_STRING=""
```

2. (Optional) Use the `az storage queue list` command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
Azure CLI
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

```
bash
Azure CLI
echo `echo $(az storage message get --queue-name outqueue -o tsv --
query '[].{Message:content}') | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the `LocalFunctionsProj` folder, use the [func azure functionapp publish](#) command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`. The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

```
Azure CLI
```

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in C#.](#)
- [Azure Functions C# developer reference](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the

app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles is already enabled in the `host.json` file at the root of the project, which should look like the following example:

```
JSON

{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[3.*, 4.0.0)"
  }
}
```

Now, you can add the storage output binding to your project.

Add an output binding

In a Java project, the bindings are defined as binding annotations on the function method. The *function.json* file is then autogenerated based on these annotations.

Browse to the location of your function code under *src/main/java*, open the *Function.java* project file, and add the following parameter to the `run` method definition:

Java

```
@QueueOutput(name = "msg", queueName = "outqueue",
connection = "AzureWebJobsStorage") OutputBinding<String> msg,
```

The `msg` parameter is an `OutputBinding<T>` type, which represents a collection of strings that are written as messages to an output binding when the function completes. In this case, the output is a storage queue named `outqueue`. The connection string for the Storage account is set by the `connection` method. Rather than the connection string itself, you pass the application setting that contains the Storage account connection string.

The `run` method definition should now look like the following example:

Java

```
@FunctionName("HttpExample")
public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET,
HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @QueueOutput(name = "msg", queueName = "outqueue",
connection = "AzureWebJobsStorage") OutputBinding<String> msg,
    final ExecutionContext context) {
```

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Now, you can use the new `msg` parameter to write to the output binding from your function code. Add the following line of code before the success response to add the value of `name` to the `msg` output binding.

Java

```
msg.setValue(name);
```

When you use an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Your `run` method should now look like the following example:

Java

```
@FunctionName("HttpExample")
public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET,
HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @QueueOutput(name = "msg", queueName = "outqueue",
        connection = "AzureWebJobsStorage") OutputBinding<String> msg,
    final ExecutionContext context) {
    context.getLogger().info("Java HTTP trigger processed a request.");

    // Parse query parameter
    String query = request.getQueryParameters().get("name");
    String name = request.getBody().orElse(query);

    if (name == null) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Please pass a name on the query string or in the request
body").build();
    } else {
        // Write the name to the message queue.
        msg.setValue(name);

        return request.createResponseBuilder(HttpStatus.OK).body("Hello, " +
name).build();
    }
}
```

Update the tests

Because the archetype also creates a set of tests, you need to update these tests to handle the new `msg` parameter in the `run` method signature.

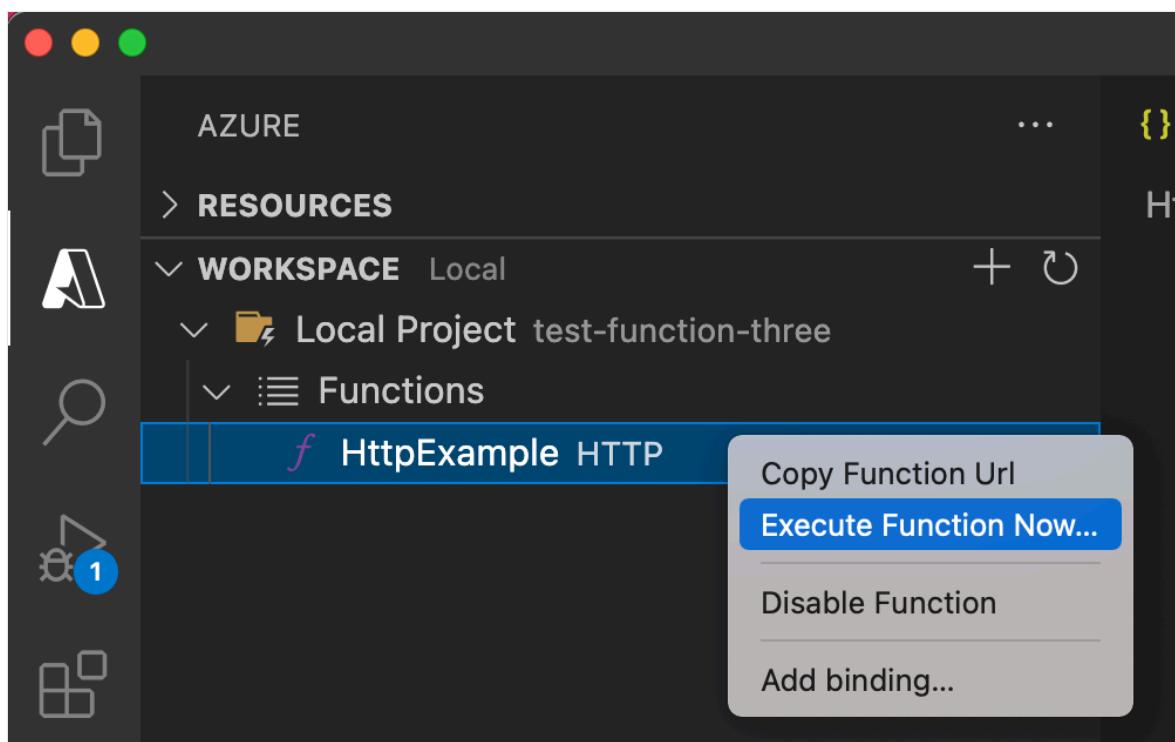
Browse to the location of your test code under `src/test/java`, open the `Function.java` project file, and replace the line of code under `//Invoke` with the following code.

Java

```
@SuppressWarnings("unchecked")
final OutputBinding<String> msg =
(OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);
```

Run the function locally

1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and select **Execute Function Now....**



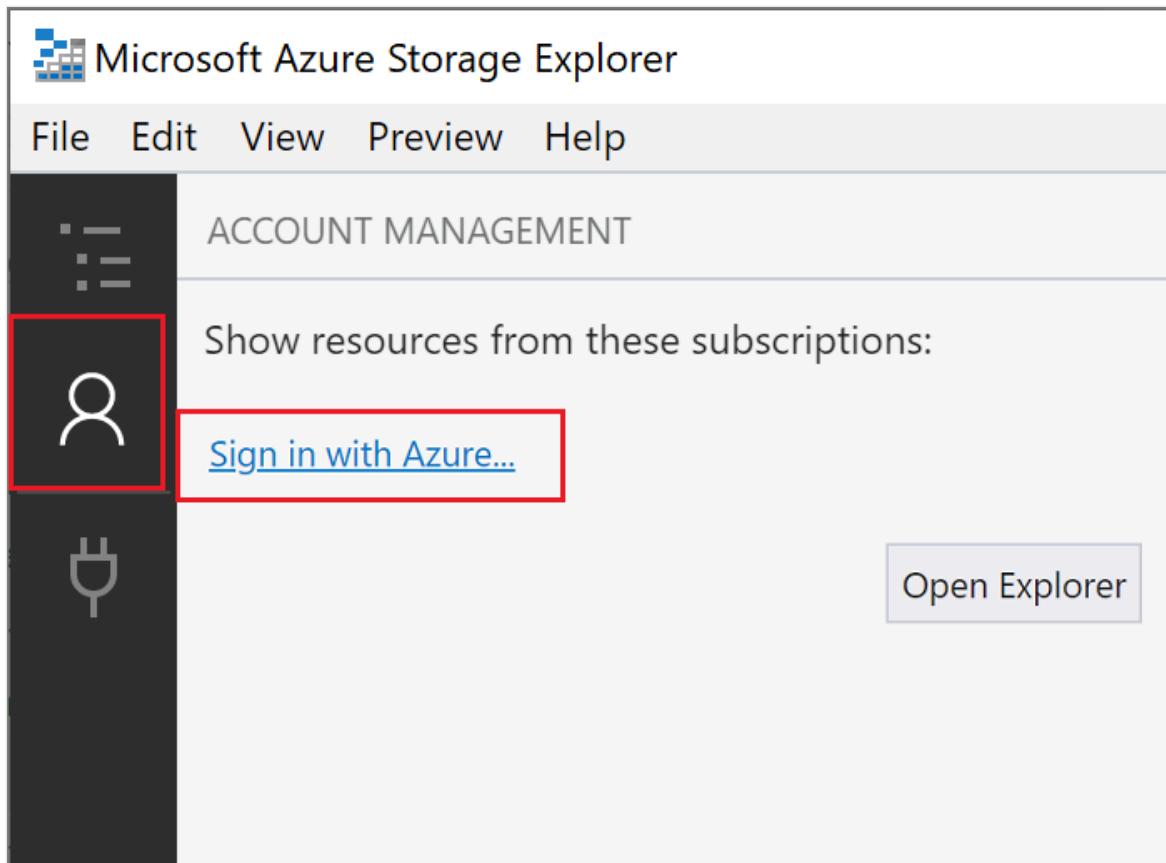
3. In the **Enter request body**, you see the request message body value of `{ "name": "Azure" }`. Press `Enter` to send this request message to your function.
4. After a response is returned, press `Ctrl + C` to stop Core Tools.

Because you're using the storage connection string, your function connects to the Azure storage account when running locally. A new queue named `outqueue` is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

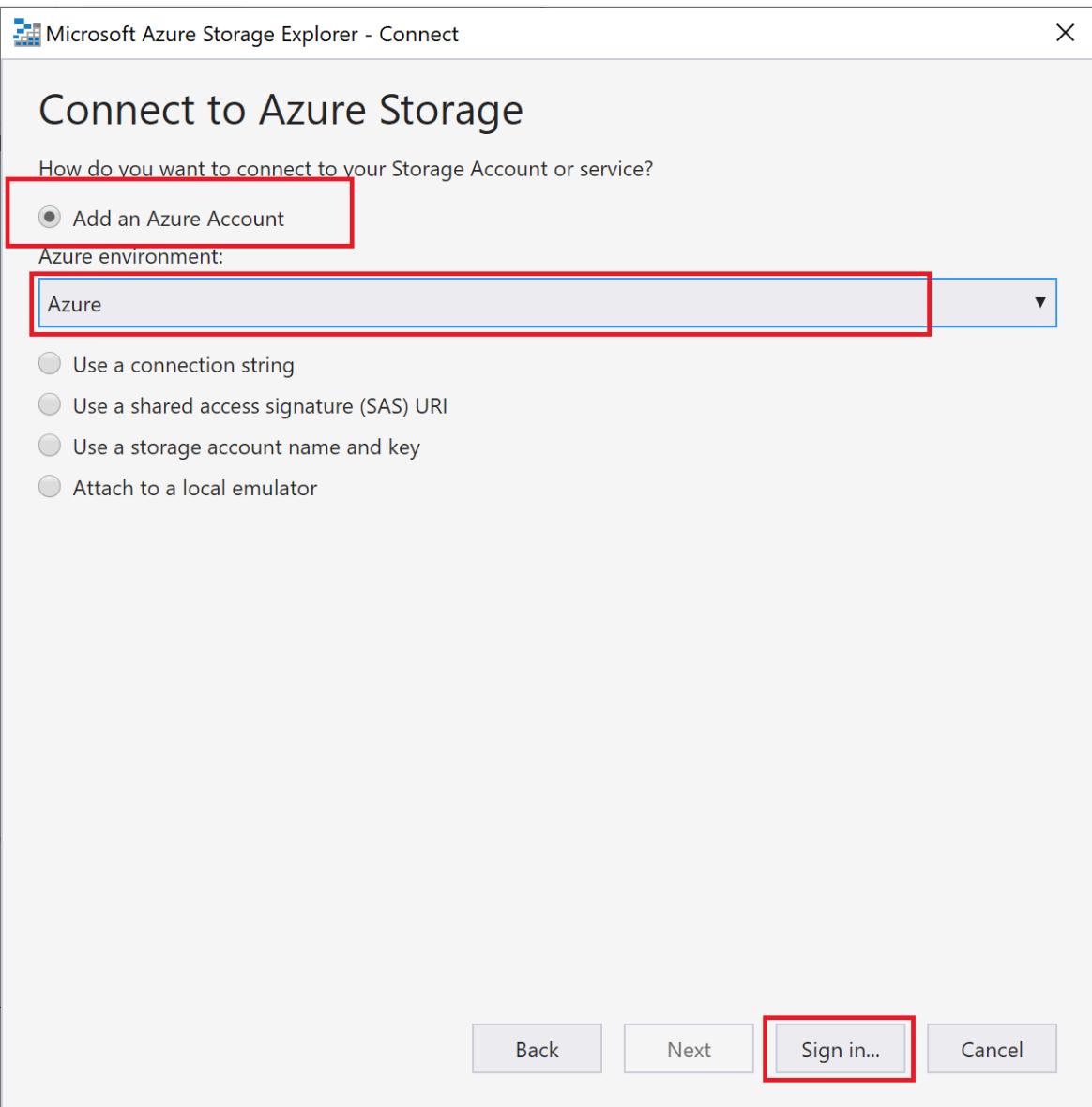
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

1. Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select Add an account.



2. In the Connect dialog, choose Add an Azure account, choose your Azure environment, and then select Sign in....

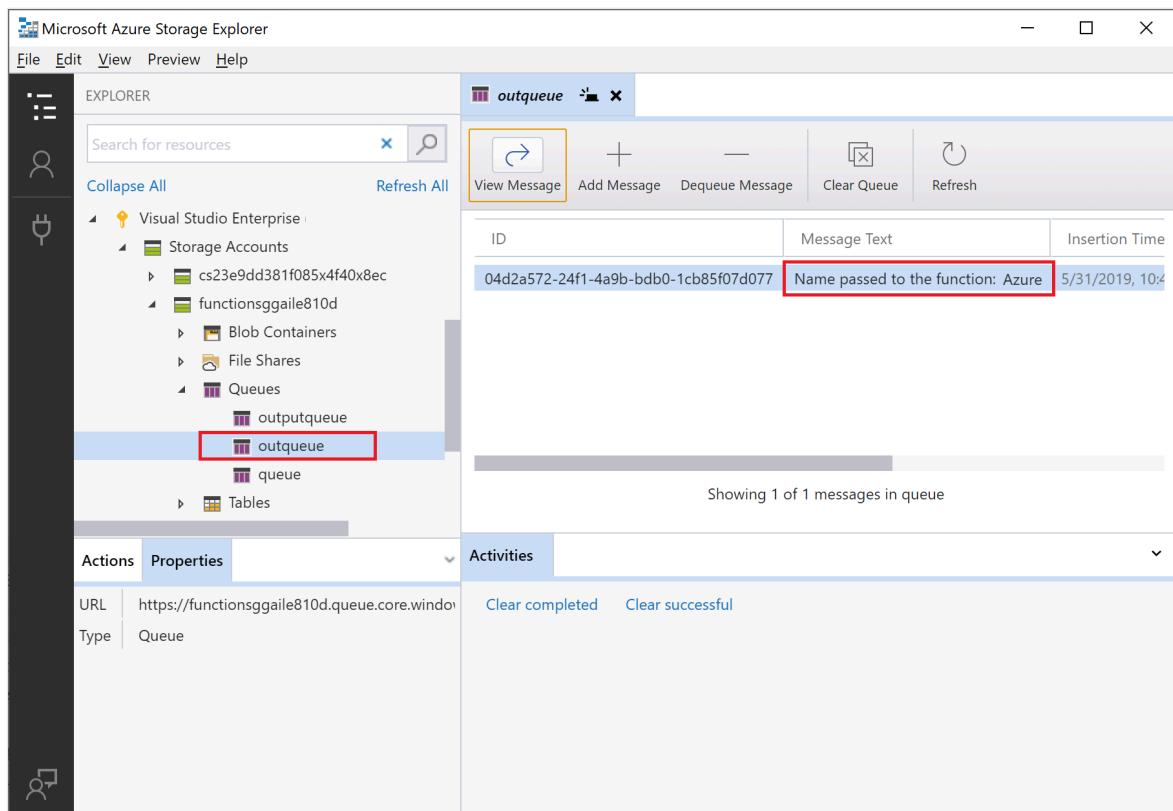


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



- Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

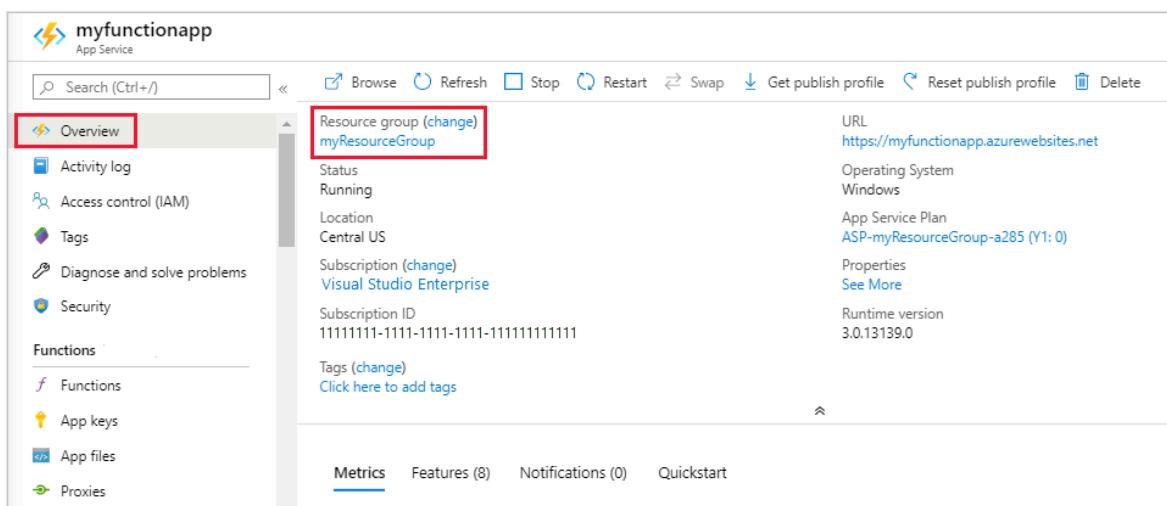
- In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
- Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
- After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
- Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the "Overview" tab, which displays details about the app: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (with a link to add tags). A red box highlights the "Resource group (change)" link, which points to "myResourceGroup". To the right of the main content, there are sections for URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties (with a "See More" link), and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in Java](#).
- [Azure Functions Java developer guide](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

In a Java project, the bindings are defined as binding annotations on the function method. The `function.json` file is then autogenerated based on these annotations.

Browse to the location of your function code under `src/main/java`, open the `Function.java` project file, and add the following parameter to the `run` method definition:

Java

```
@QueueOutput(name = "msg", queueName = "outqueue", connection =
"AzureWebJobsStorage") OutputBinding<String> msg
```

The `msg` parameter is an `OutputBinding<T>` type, which represents a collection of strings. These strings are written as messages to an output binding when the function completes. In this case, the output is a storage queue named `outqueue`. The connection string for the Storage account is set by the `connection` method. You pass the application setting that contains the Storage account connection string, rather than passing the connection string itself.

The `run` method definition must now look like the following example:

Java

```
@FunctionName("HttpTrigger-Java")
public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET,
    HttpMethod.POST}, authLevel = AuthorizationLevel.FUNCTION)
    HttpRequestMessage<Optional<String>> request,
    @QueueOutput(name = "msg", queueName = "outqueue", connection =
"AzureWebJobsStorage")
    OutputBinding<String> msg, final ExecutionContext context) {
```

```
}
```

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Now, you can use the new `msg` parameter to write to the output binding from your function code. Add the following line of code before the success response to add the value of `name` to the `msg` output binding.

Java

```
msg.setValue(name);
```

When you use an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Your `run` method must now look like the following example:

Java

```
public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET,
    HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @QueueOutput(name = "msg", queueName = "outqueue",
    connection = "AzureWebJobsStorage") OutputBinding<String> msg,
    final ExecutionContext context) {
    context.getLogger().info("Java HTTP trigger processed a request.");

    // Parse query parameter
    String query = request.getQueryParameters().get("name");
    String name = request.getBody().orElse(query);

    if (name == null) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Please pass a name on the query string or in the request
body").build();
    } else {
        // Write the name to the message queue.
        msg.setValue(name);
```

```
        return request.createResponseBuilder(HttpStatus.OK).body("Hello, " +  
name).build();  
    }  
}
```

Update the tests

Because the archetype also creates a set of tests, you need to update these tests to handle the new `msg` parameter in the `run` method signature.

Browse to the location of your test code under `src/test/java`, open the `Function.java` project file, and replace the line of code under `//Invoke` with the following code:

Java

```
@SuppressWarnings("unchecked")  
final OutputBinding<String> msg =  
(OutputBinding<String>)mock(OutputBinding.class);  
final HttpResponseMessage ret = new Function().run(req, msg, context);
```

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the `LocalFunctionProj` folder.

Console

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools  
Core Tools Version: 4.0.5049 Commit hash: N/A (64-bit)  
Function Runtime Version: 4.15.2.20177  
  
[2023-03-17T03:27:12.372Z] Worker process started and initialized.  
  
Functions:  
  
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

(!) Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `ctrl + c` and type `y` to stop the functions host.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's `local.setting.json` file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

```
bash
Bash
export AZURE_STORAGE_CONNECTION_STRING=""
```

2. (Optional) Use the [az storage queue list](#) command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
Azure CLI
```

```
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

bash

Azure CLI

```
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}') | base64 --decode`
```

Because the message body is stored `base64 encoded`, the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the local project folder, use the following Maven command to republish your project:

```
mvn azure-functions:deploy
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`.

The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

ⓘ Note

This article currently supports [Node.js v4 for Functions](#).

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`

2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

i Important

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles is already enabled in the `host.json` file at the root of the project, which should look like the following example:

JSON

```
{  
  "version": "2.0",  
  "logging": {  
    "applicationInsights": {  
      "samplingSettings": {  
        "isEnabled": true,  
        "excludedTypes": "Request"  
      }  
    }  
  }  
}
```

```
        },
    },
    "extensionBundle": {
        "id": "Microsoft.Azure.Functions.ExtensionBundle",
        "version": "[4.*, 5.0.0)"
    }
}
```

Now, you can add the storage output binding to your project.

Add an output binding

To write to an Azure Storage queue:

- Add an `extraOutputs` property to the binding configuration

JavaScript

```
{
    methods: ['GET', 'POST'],
    extraOutputs: [sendToQueue], // add output binding to HTTP trigger
    authLevel: 'anonymous',
    handler: () => {}
}
```

- Add a `output.storageQueue` function above the `app.http` call

JavaScript

```
const sendToQueue = output.storageQueue({
    queueName: 'outqueue',
    connection: 'AzureWebJobsStorage',
});
```

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Add code that uses the output binding object on `context.extraOutputs` to create a queue message. Add this code before the return statement.

JavaScript

```
context.extraOutputs.set(sendToQueue, [msg]);
```

At this point, your function could look as follows:

JavaScript

```
const { app, output } = require('@azure/functions');

const sendToQueue = output.storageQueue({
    queueName: 'outqueue',
    connection: 'AzureWebJobsStorage',
});

app.http('HttpExample', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    extraOutputs: [sendToQueue],
    handler: async (request, context) => {
        try {
            context.log(`Http function processed request for url
"${{request.url}}`);

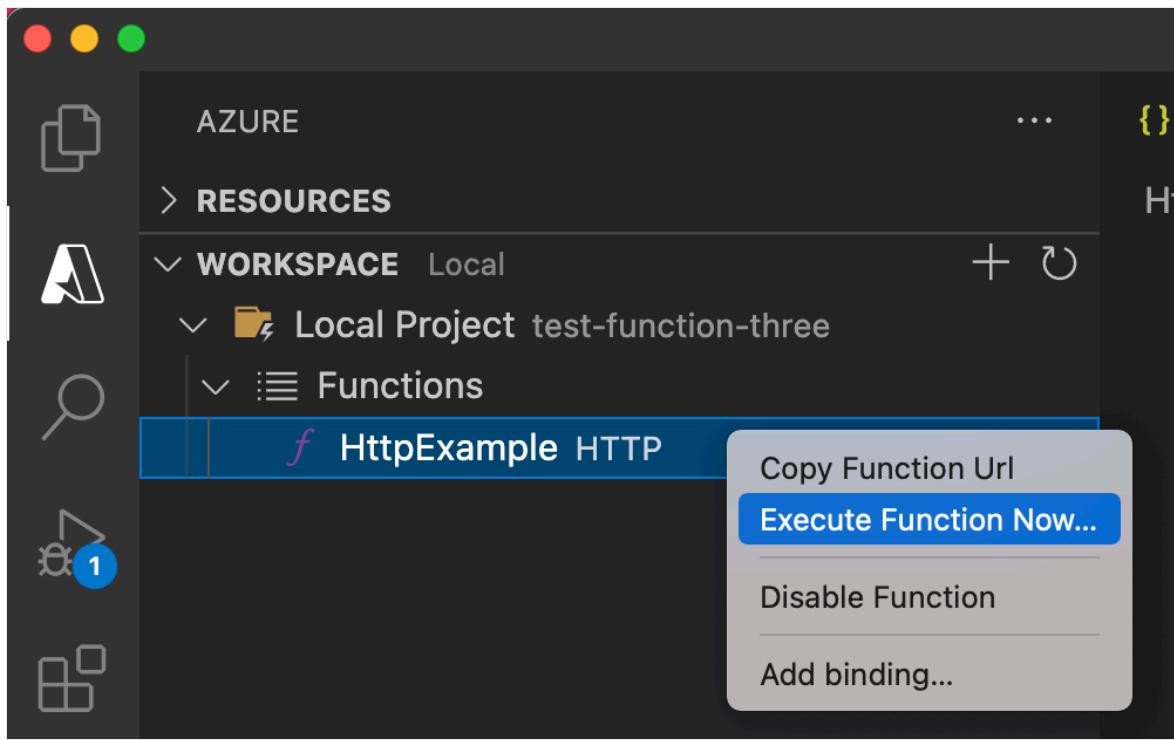
            const name = request.query.get('name') || (await request.text());
            context.log(`Name: ${name}`);

            if (name) {
                const msg = `Name passed to the function ${name}`;
                context.extraOutputs.set(sendToQueue, [msg]);
                return { body: msg };
            } else {
                context.log('Missing required data');
                return { status: 404, body: 'Missing required data' };
            }
        } catch (error) {
            context.log(`Error: ${error}`);
            return { status: 500, body: 'Internal Server Error' };
        }
    },
});
```

Run the function locally

1. As in the previous article, press **F5** to start the function app project and Core Tools.
2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the **HttpExample**

function and select **Execute Function Now....**



3. In the **Enter request body**, you see the request message body value of `{ "name": "Azure" }`. Press **Enter** to send this request message to your function.

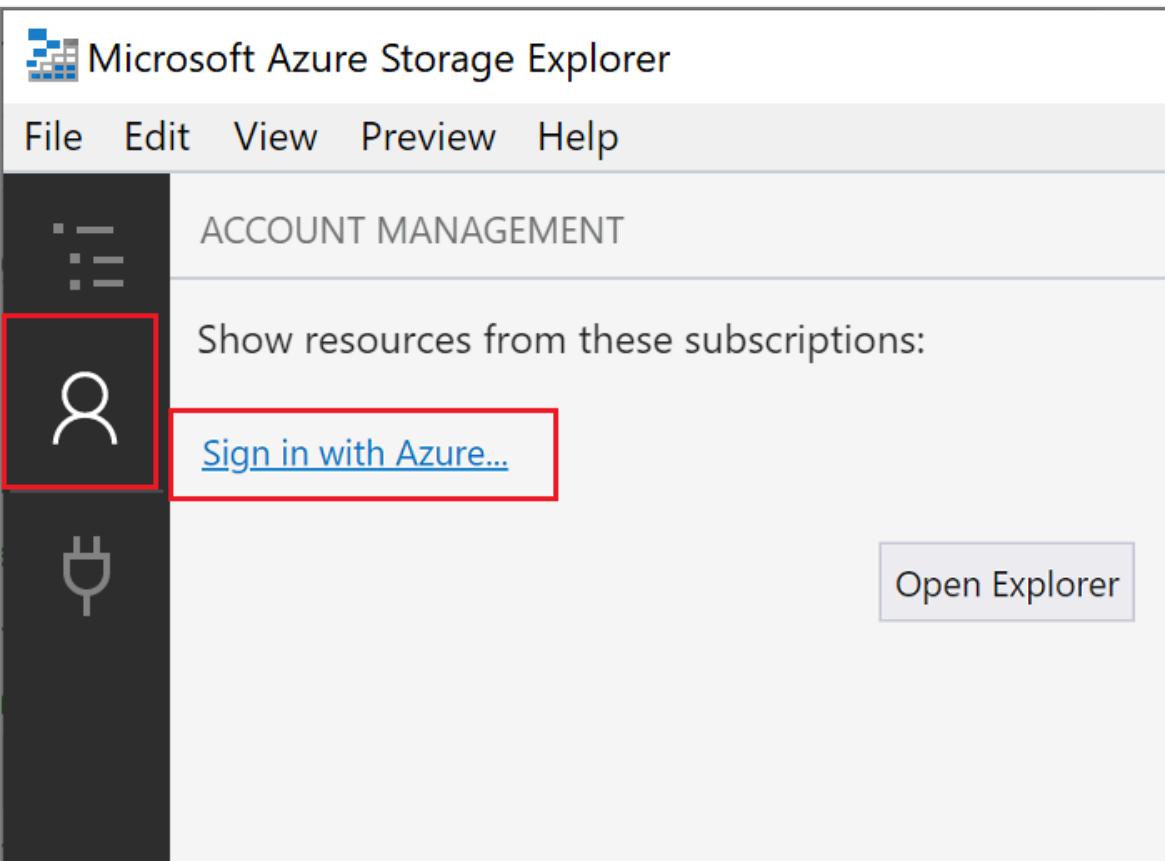
4. After a response is returned, press **Ctrl + c** to stop Core Tools.

Because you're using the storage connection string, your function connects to the Azure storage account when running locally. A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

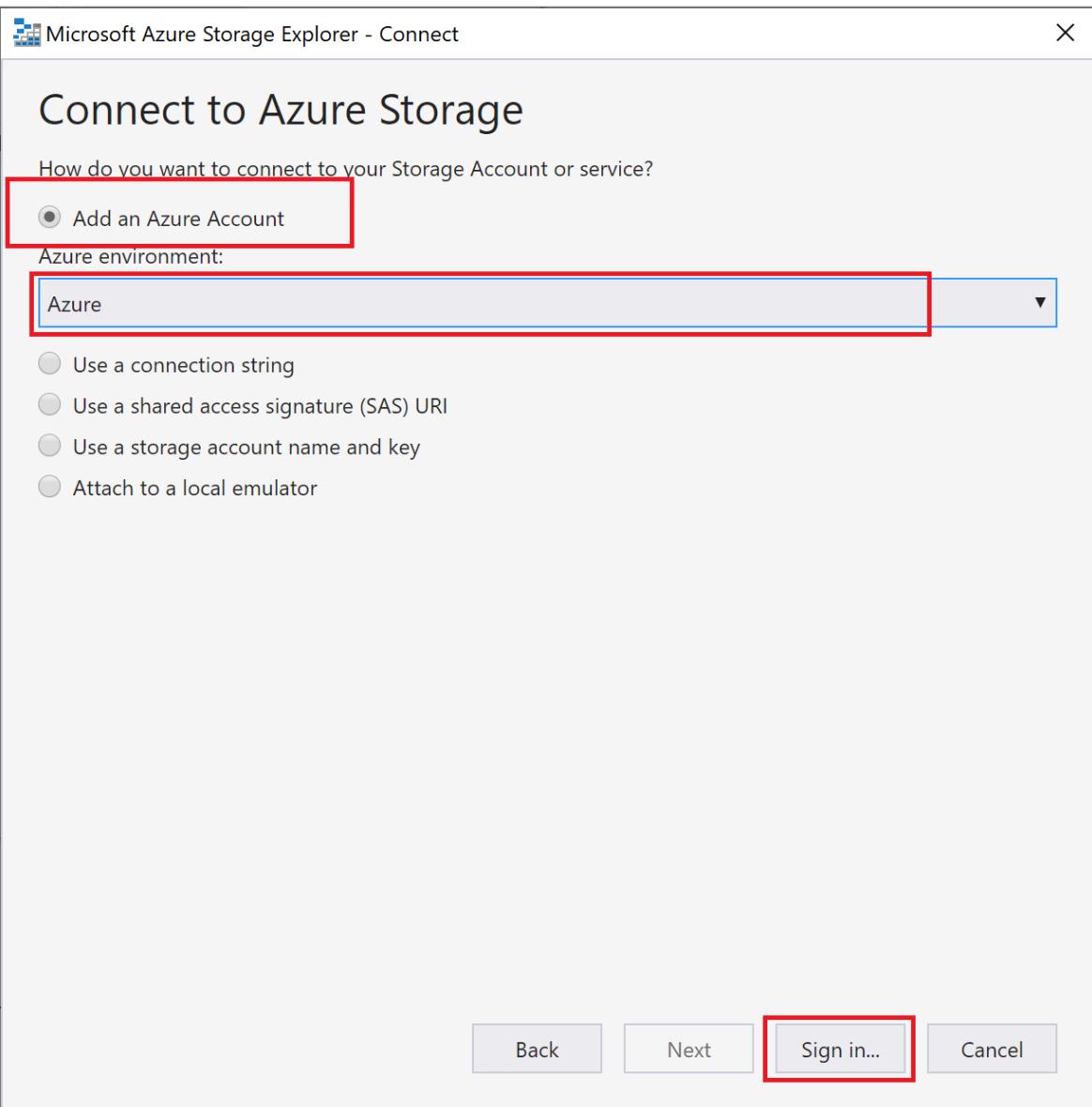
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

1. Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

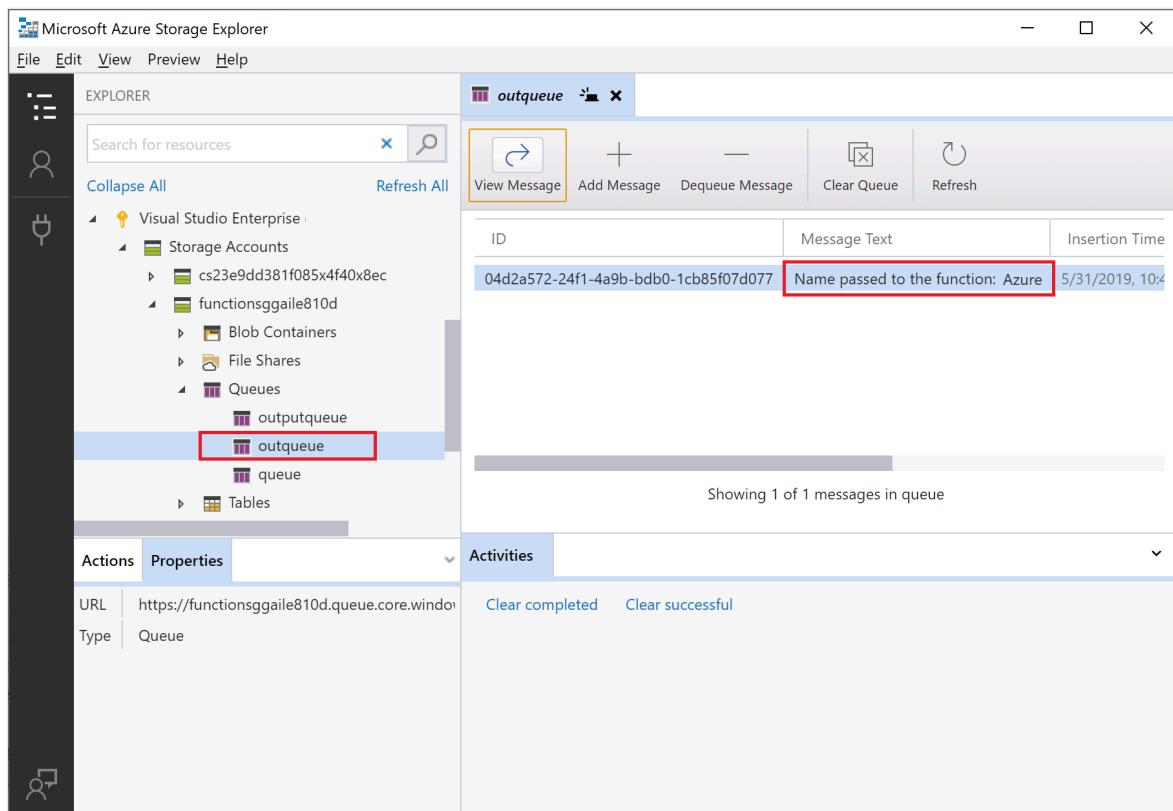


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

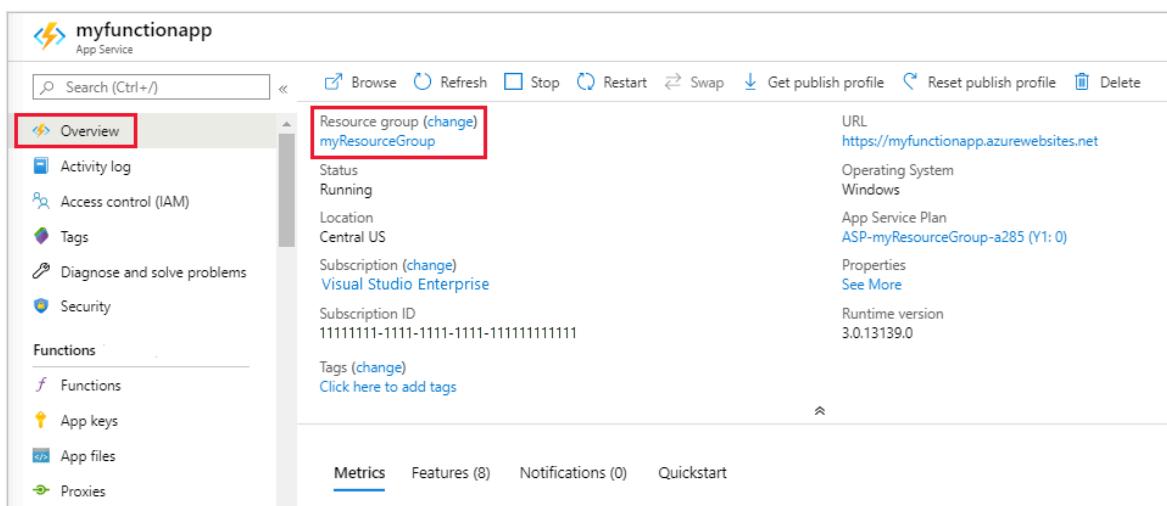
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named 'myfunctionapp'. The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the 'Overview' tab, which displays details about the app: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (with a link to add tags). A red box highlights the 'Resource group (change)' link, which points to 'myResourceGroup'. To the right of the resource group details, there are links for URL (https://myfunctionapp.azurewebsites.net), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties, See More, and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in JavaScript](#).
- [Azure Functions JavaScript developer guide](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

When using the [Node.js v4 programming model](#), binding attributes are defined directly in the `./src/functions/HttpExample.js` file. From the previous quickstart, your file already contains an HTTP binding defined by the `app.http` method.

JavaScript

```
const { app } = require('@azure/functions');

app.http('httpTrigger', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    handler: async (request, context) => {
        try {
            context.log(`Http function processed request for url
"${request.url}"`);

            const name = request.query.get('name') || (await request.text());
            context.log(`Name: ${name}`);

            if (!name) {
                return { status: 404, body: 'Not Found' };
            }

            return { body: `Hello, ${name}!` };
        } catch (error) {
            context.log(`Error: ${error}`);
            return { status: 500, body: 'Internal Server Error' };
        }
    },
});
```

To write to an Azure Storage queue:

- Add an `extraOutputs` property to the binding configuration

TypeScript

```
{  
  methods: ['GET', 'POST'],  
  extraOutputs: [sendToQueue], // add output binding to HTTP trigger  
  authLevel: 'anonymous',  
  handler: () => {}  
}
```

- Add a `output.storageQueue` function above the `app.http` call

TypeScript

```
const sendToQueue: StorageQueueOutput = output.storageQueue({  
  queueName: 'outqueue',  
  connection: 'AzureWebJobsStorage',  
});
```

For a `queue` type, you must specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from `local.settings.json` file) in `connection`.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Add code that uses the output binding object on `context.extraOutputs` to create a queue message. Add this code before the return statement.

JavaScript

```
context.extraOutputs.set(sendToQueue, [msg]);
```

At this point, your function could look as follows:

JavaScript

```
const { app, output } = require('@azure/functions');  
  
const sendToQueue = output.storageQueue({  
  queueName: 'outqueue',  
  connection: 'AzureWebJobsStorage',  
});
```

```

app.http('HttpExample', {
  methods: ['GET', 'POST'],
  authLevel: 'anonymous',
  extraOutputs: [sendToQueue],
  handler: async (request, context) => {
    try {
      context.log(`Http function processed request for url
      "${request.url}"`);

      const name = request.query.get('name') || (await request.text());
      context.log(`Name: ${name}`);

      if (name) {
        const msg = `Name passed to the function ${name}`;
        context.extraOutputs.set(sendToQueue, [msg]);
        return { body: msg };
      } else {
        context.log('Missing required data');
        return { status: 404, body: 'Missing required data' };
      }
    } catch (error) {
      context.log(`Error: ${error}`);
      return { status: 500, body: 'Internal Server Error' };
    }
  },
});

```

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

Console

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:    4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

💡 Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in `host.json`.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#).

You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's `local.setting.json` file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

```
bash
Bash
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>
```

2. (Optional) Use the `az storage queue list` command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
Azure CLI
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

```
bash
Azure CLI
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}') | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`,

you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the *LocalFunctionsProj* folder, use the `func azure functionapp publish` command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`. The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in JavaScript.](#)
- [Azure Functions JavaScript developer guide](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the

app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles is already enabled in the `host.json` file at the root of the project, which should look like the following example:

```
JSON

{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[3.*, 4.0.0)"
  }
}
```

Now, you can add the storage output binding to your project.

Add an output binding

In Functions, each type of binding requires a `direction`, `type`, and unique `name`. The way you define these attributes depends on the language of your function app.

Binding attributes are defined in the `function.json` file for a given function. Depending on the binding type, additional properties may be required. The [queue output configuration](#) describes the fields required for an Azure Storage queue binding. The extension makes it easy to add bindings to the `function.json` file.

To create a binding, right-click (Ctrl+click on macOS) the `function.json` file in your `HttpTrigger` folder and choose **Add binding....** Follow the prompts to define the following binding properties for the new binding:

[\[+\] Expand table](#)

Prompt	Value	Description
Select binding direction	<code>out</code>	The binding is an output binding.
Select binding with direction...	<code>Azure Queue Storage</code>	The binding is an Azure Storage queue binding.
The name used to identify this binding in your code	<code>msg</code>	Name that identifies the binding parameter referenced in your code.
The queue to which the message will be sent	<code>outqueue</code>	The name of the queue that the binding writes to. When the <code>queueName</code> doesn't exist, the binding creates it on first use.
Select setting from "local.setting.json"	<code>AzureWebJobsStorage</code>	The name of an application setting that contains the connection string for the Storage account. The <code>AzureWebJobsStorage</code> setting contains the connection string for the Storage account you created with the function app.

A binding is added to the `bindings` array in your `function.json`, which should look like the following:

```
JSON
{
  "bindings": [
    {
      "name": "msg",
      "queueName": "outqueue",
      "connection": "AzureWebJobsStorage"
    }
  ]
}
```

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Add code that uses the `Push-OutputBinding` cmdlet to write text to the queue using the `msg` output binding. Add this code before you set the OK status in the `if` statement.

PowerShell

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function must look as follows:

PowerShell

```
using namespace System.Net

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Write to the Azure Functions log stream.
Write-Host "PowerShell HTTP trigger function processed a request."

# Interact with query parameters or the body of the request.
$name = $Request.Query.Name
if (-not $name) {
    $name = $Request.Body.Name
}

if ($name) {
    # Write the $name value to the queue,
    # which is the name passed to the function.
    $outputMsg = $name
    Push-OutputBinding -name msg -Value $outputMsg

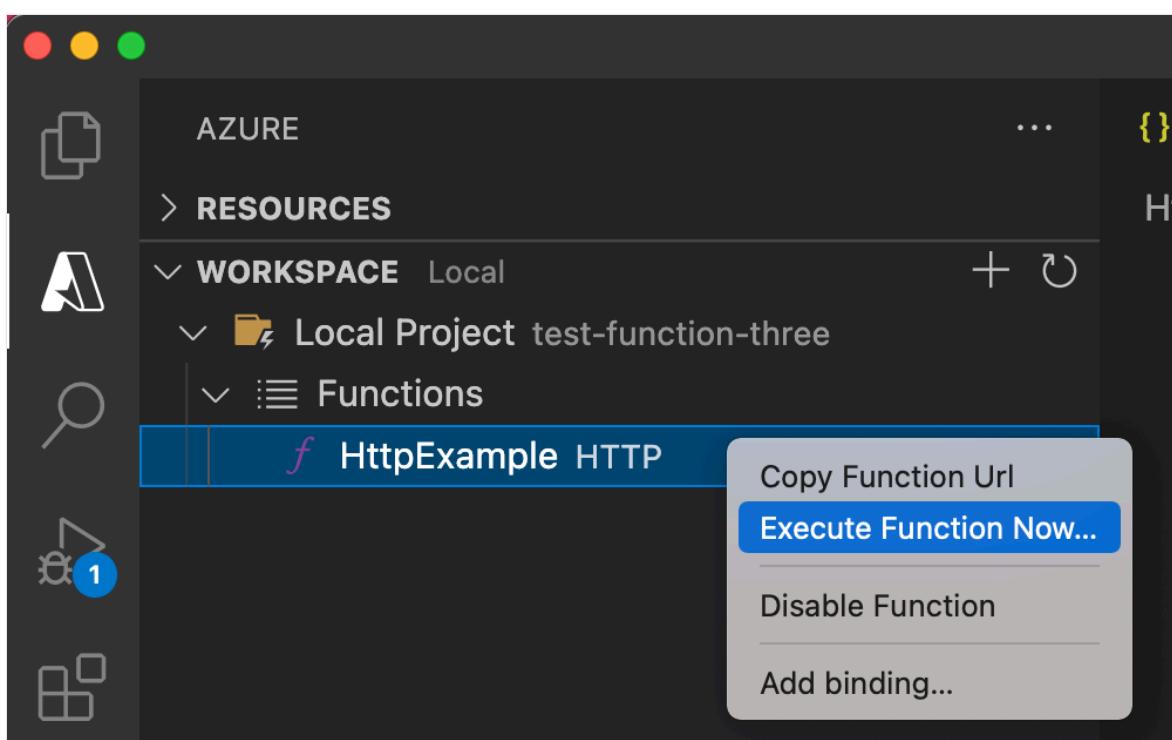
    $status = [HttpStatusCode]::OK
    $body = "Hello $name"
}
else {
    $status = [HttpStatusCode]::BadRequest
    $body = "Please pass a name on the query string or in the request body."
}

# Associate values to output bindings by calling 'Push-OutputBinding'.
Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = $status
})
```

```
    Body = $body  
})
```

Run the function locally

1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and select **Execute Function Now....**



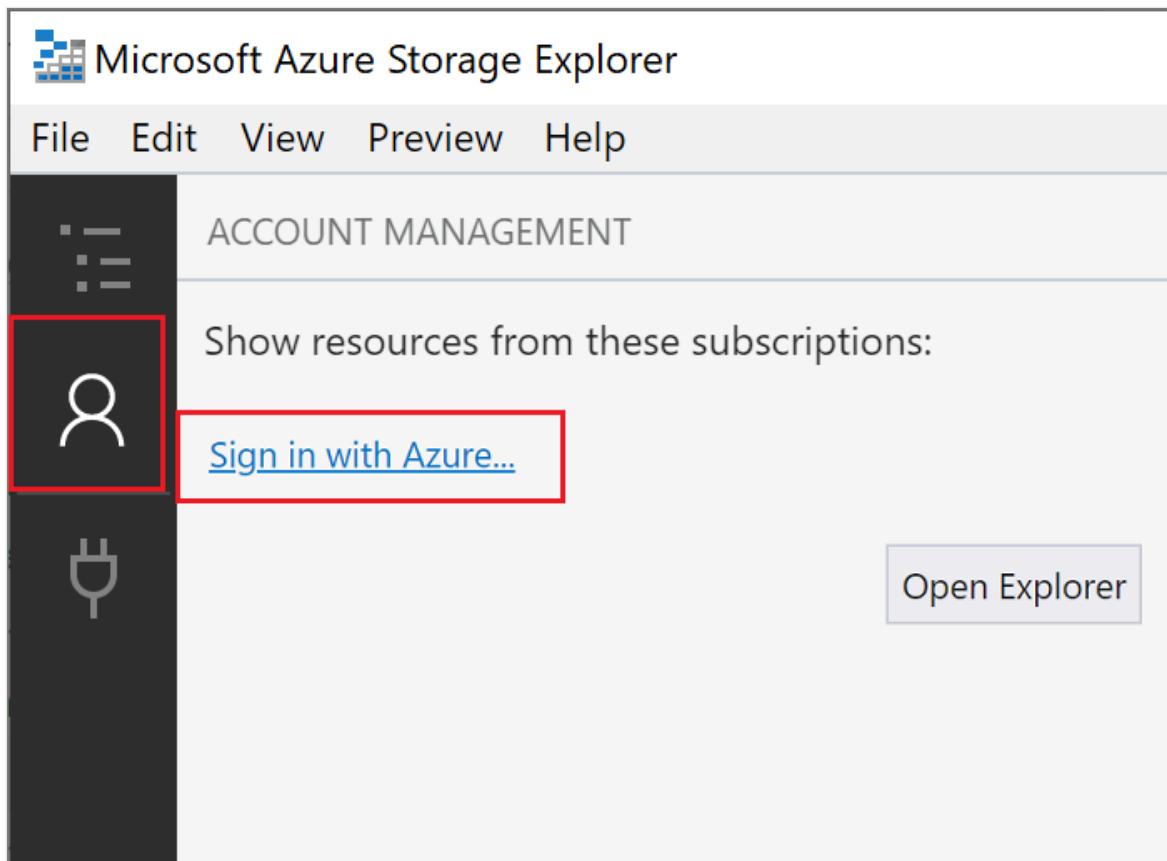
3. In the **Enter request body**, you see the request message body value of `{ "name": "Azure" }`. Press `Enter` to send this request message to your function.
4. After a response is returned, press `Ctrl + C` to stop Core Tools.

Because you're using the storage connection string, your function connects to the Azure storage account when running locally. A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

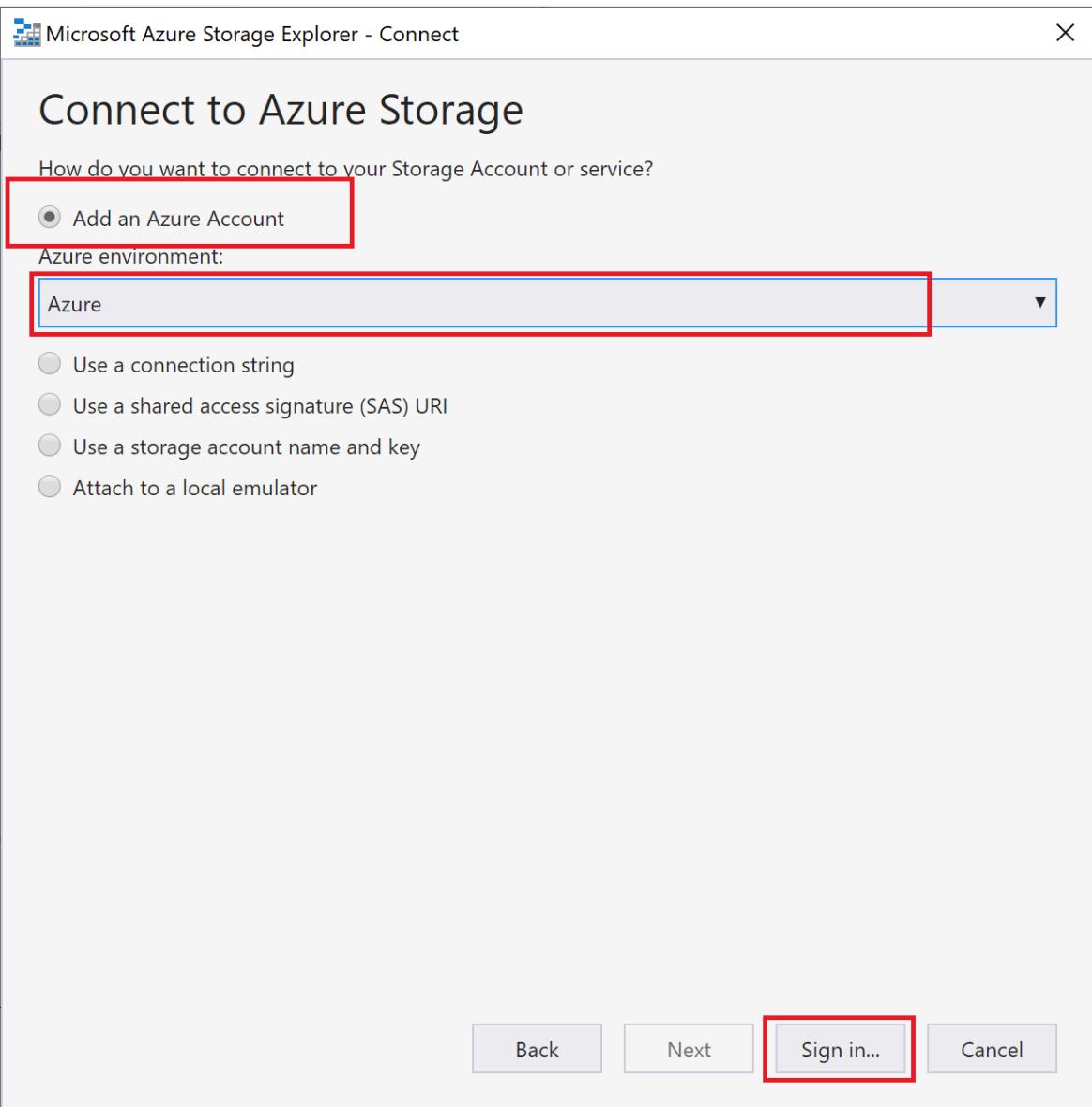
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

1. Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

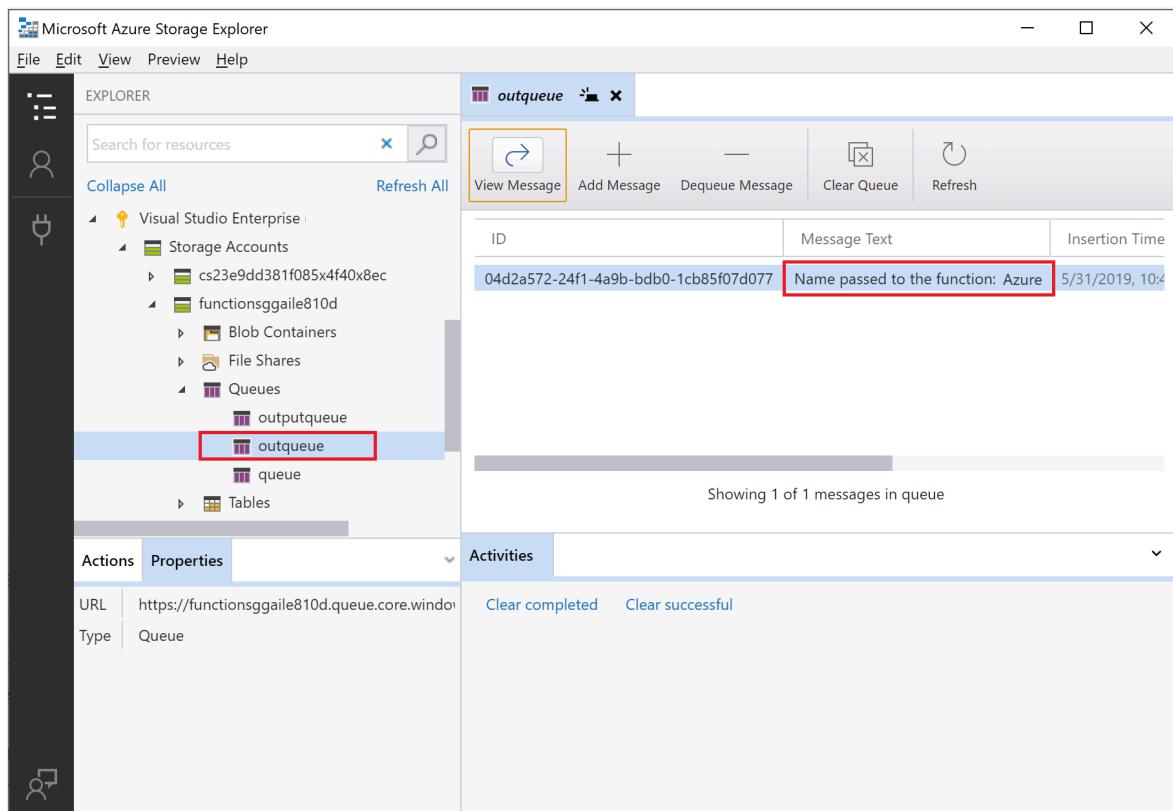


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

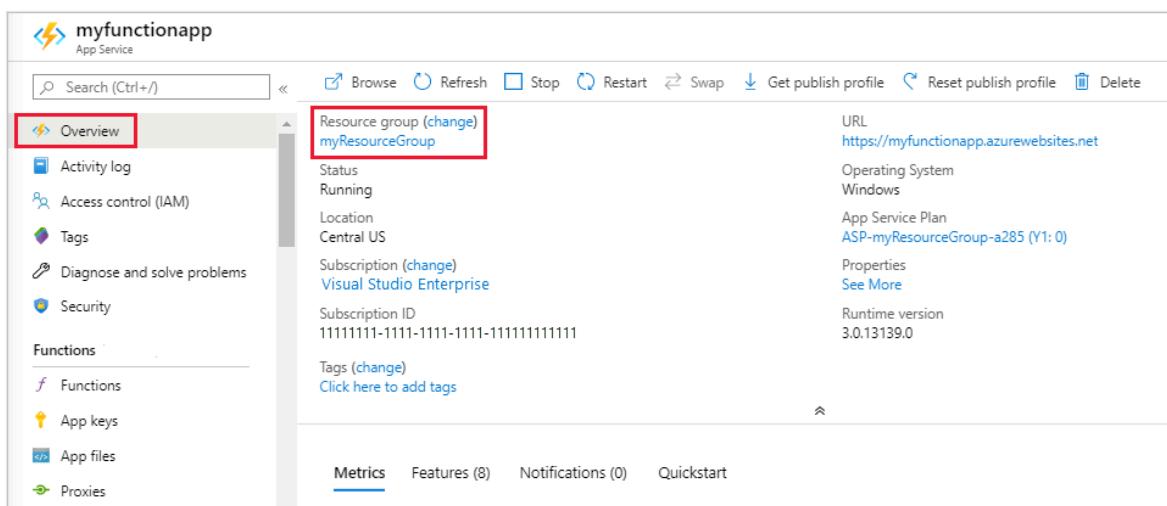
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named 'myfunctionapp'. The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the 'Overview' tab, which displays the following details:

- Resource group (change)**: myResourceGroup
- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags
- URL: https://myfunctionapp.azurewebsites.net
- Operating System: Windows
- App Service Plan: ASP-myResourceGroup-a285 (Y1: 0)
- Properties: See More
- Runtime version: 3.0.13139.0

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in PowerShell](#).
- [Azure Functions PowerShell developer guide](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

You declare these bindings in the `function.json` file in your function folder. From the previous quickstart, your `function.json` file in the `HttpExample` folder contains two bindings in the `bindings` collection:

JSON

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "Response"
  }
]
```

The second binding in the collection is named `res`. This `http` binding is an output binding (`out`) that is used to write the HTTP response.

To write to an Azure Storage queue from this function, add an `out` binding of type `queue` with the name `msg`, as shown in the code below:

JSON

```
{
  "authLevel": "function",
```

```
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
        "get",
        "post"
    ],
},
{
    "type": "http",
    "direction": "out",
    "name": "Response"
},
{
    "type": "queue",
    "direction": "out",
    "name": "msg",
    "queueName": "outqueue",
    "connection": "AzureWebJobsStorage"
}
]
```

For a `queue` type, you must specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from `local.settings.json` file) in `connection`.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Add code that uses the `Push-OutputBinding` cmdlet to write text to the queue using the `msg` output binding. Add this code before you set the OK status in the `if` statement.

PowerShell

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function must look as follows:

PowerShell

```
using namespace System.Net
```

```

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Write to the Azure Functions log stream.
Write-Host "PowerShell HTTP trigger function processed a request."

# Interact with query parameters or the body of the request.
$name = $Request.Query.Name
if (-not $name) {
    $name = $Request.Body.Name
}

if ($name) {
    # Write the $name value to the queue,
    # which is the name passed to the function.
    $outputMsg = $name
    Push-OutputBinding -name msg -Value $outputMsg

    $status = [HttpStatusCode]::OK
    $body = "Hello $name"
}
else {
    $status = [HttpStatusCode]::BadRequest
    $body = "Please pass a name on the query string or in the request body."
}

# Associate values to output bindings by calling 'Push-OutputBinding'.
Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = $status
    Body = $body
})

```

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

Console

func start

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:    4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use `Ctrl+C` to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

💡 Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in `host.json`.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#).

You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's `local.setting.json` file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

```
bash
Bash
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>
```

2. (Optional) Use the `az storage queue list` command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
Azure CLI
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

```
bash
Azure CLI
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}') | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`,

you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the *LocalFunctionsProj* folder, use the `func azure functionapp publish` command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`. The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in PowerShell.](#)
- [Azure Functions PowerShell developer guide](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the

app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles is already enabled in the `host.json` file at the root of the project, which should look like the following example:

```
JSON

{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[3.*, 4.0.0)"
  }
}
```

Now, you can add the storage output binding to your project.

Add an output binding

Binding attributes are defined by decorating specific function code in the `function_app.py` file. You use the `queue_output` decorator to add an [Azure Queue storage output binding](#).

By using the `queue_output` decorator, the binding direction is implicitly 'out' and type is Azure Storage Queue. Add the following decorator to your function code in `HttpExample\function_app.py`:

Python

```
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
```

In this code, `arg_name` identifies the binding parameter referenced in your code, `queue_name` is name of the queue that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Storage account. In quickstarts you use the same storage account as the function app, which is in the `AzureWebJobsStorage` setting. When the `queue_name` doesn't exist, the binding creates it on first use.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Update `HttpExample\function_app.py` to match the following code, add the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="HttpExample")
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
def HttpExample(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) ->
func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    name = req.params.get('name')
```

```

if not name:
    try:
        req_body = req.get_json()
    except ValueError:
        pass
    else:
        name = req_body.get('name')

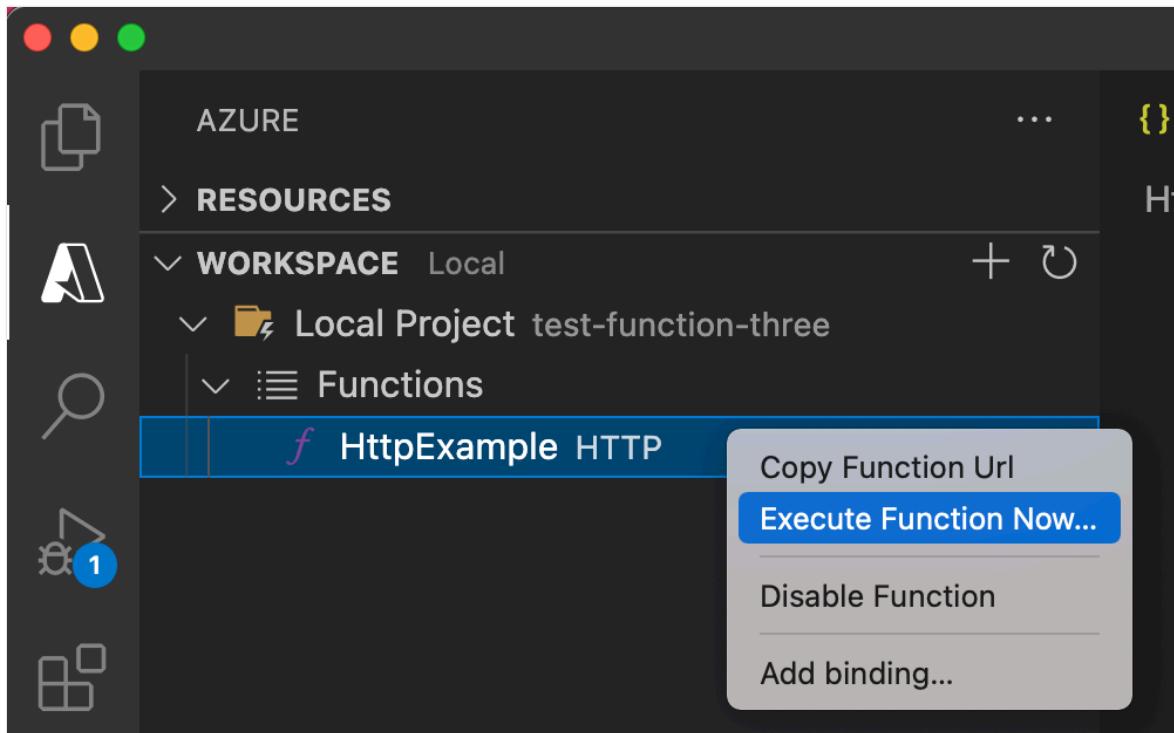
if name:
    msg.set(name)
    return func.HttpResponse(f"Hello, {name}. This HTTP triggered
function executed successfully.")
else:
    return func.HttpResponse(
        "This HTTP triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
        status_code=200
)

```

The `msg` parameter is an instance of the `azure.functions.Out class`. The `set` method writes a string message to the queue. In this case, it's the `name` passed to the function in the URL query string.

Run the function locally

1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and select **Execute Function Now....**



3. In the **Enter request body**, you see the request message body value of `{ "name": "Azure" }`. Press **Enter** to send this request message to your function.

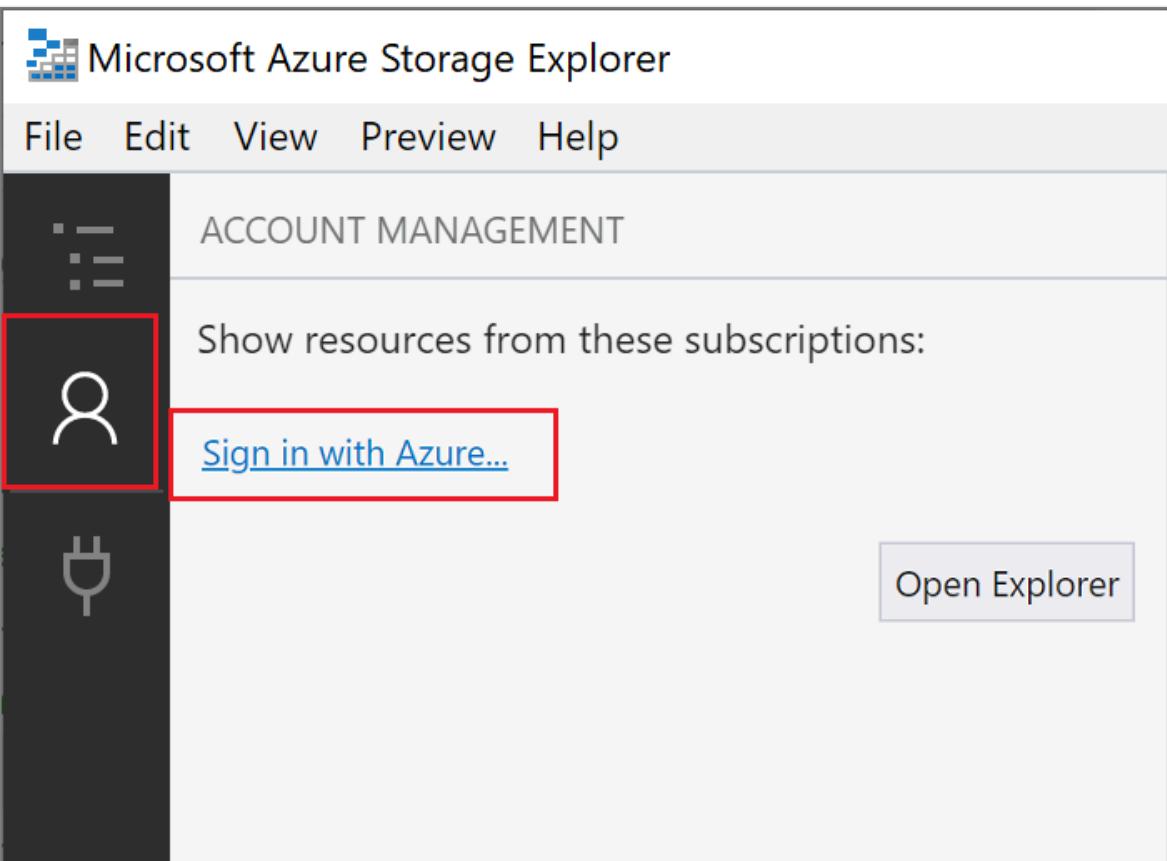
4. After a response is returned, press **Ctrl + C** to stop Core Tools.

Because you're using the storage connection string, your function connects to the Azure storage account when running locally. A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

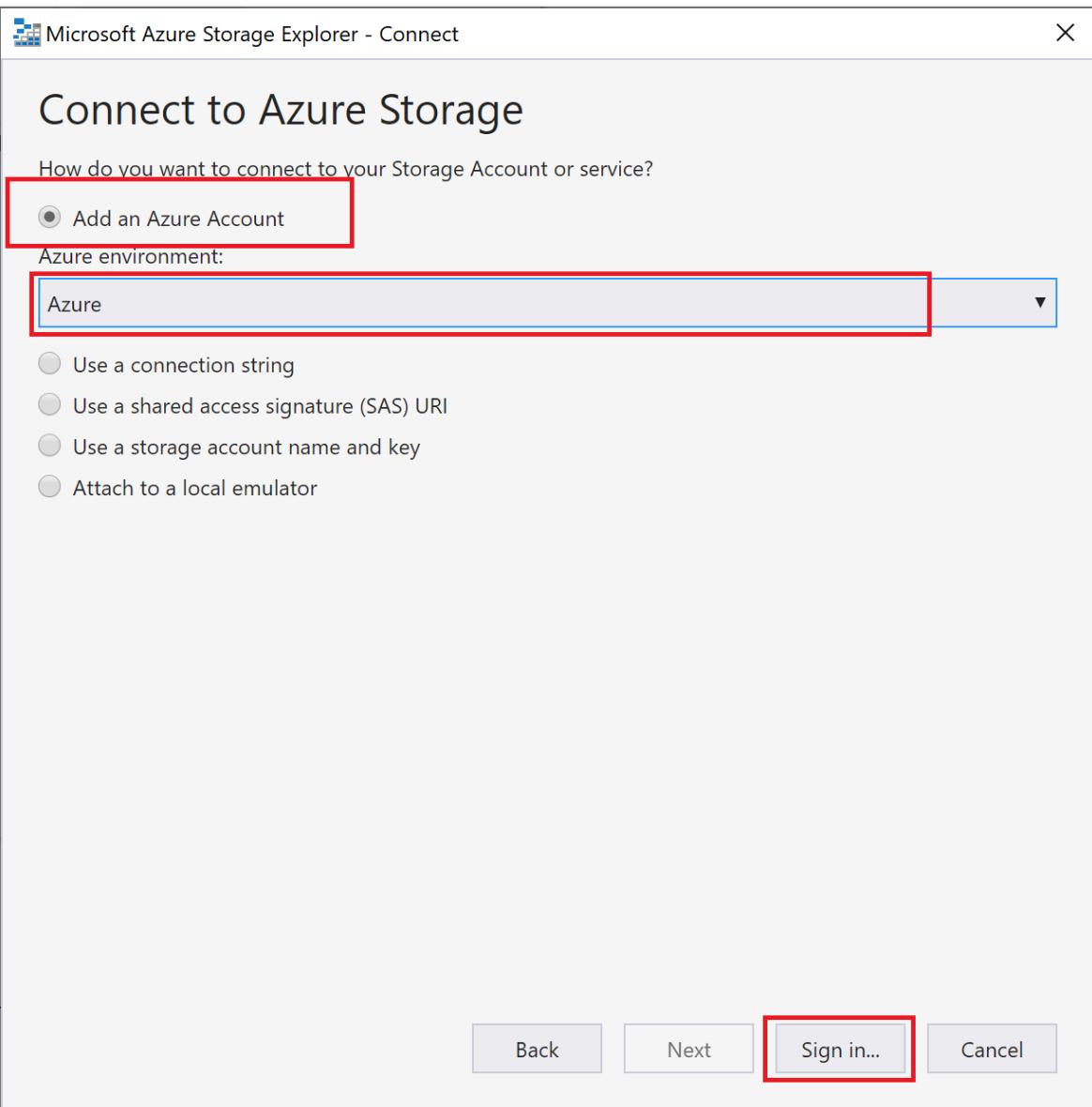
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

1. Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

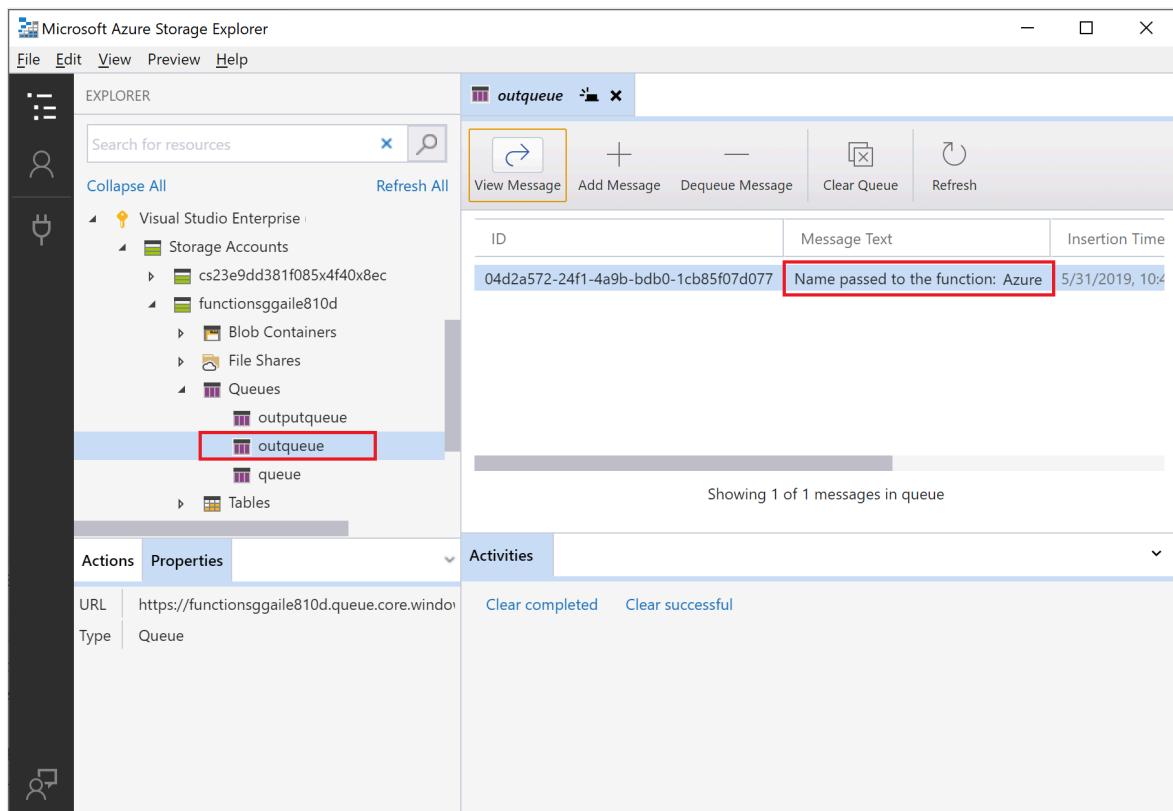


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

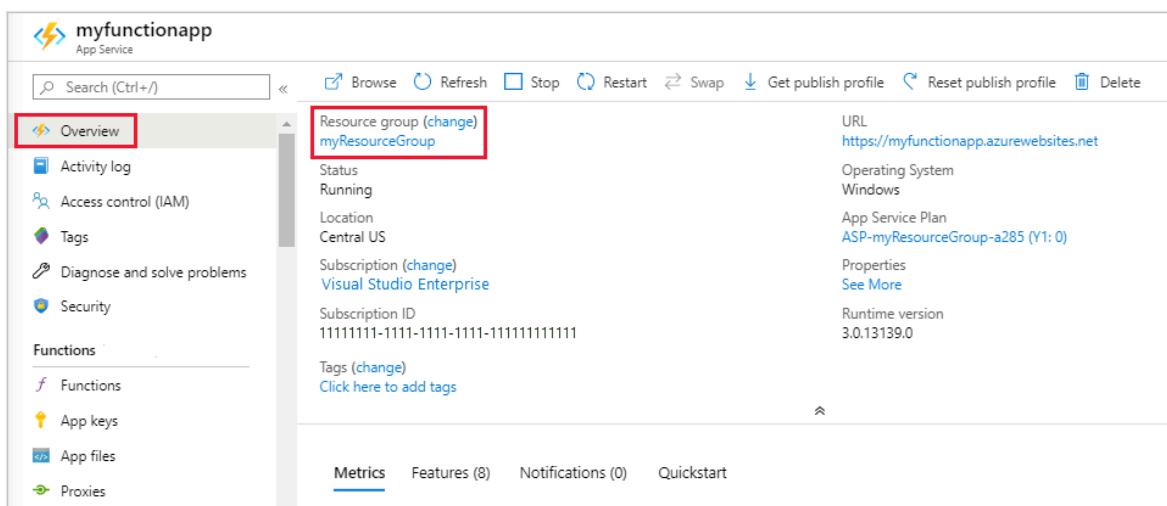
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named 'myfunctionapp'. The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the 'Overview' tab, which displays the following details:

- Resource group (change)**: myResourceGroup
- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags
- URL: https://myfunctionapp.azurewebsites.net
- Operating System: Windows
- App Service Plan: ASP-myResourceGroup-a285 (Y1: 0)
- Properties: See More
- Runtime version: 3.0.13139.0

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in Python](#).
- [Azure Functions Python developer guide](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

When using the [Python v2 programming model](#), binding attributes are defined directly in the `function_app.py` file as decorators. From the previous quickstart, your `function_app.py` file already contains one decorator-based binding:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="hello", auth_level=func.AuthLevel.ANONYMOUS)
```

The `route` decorator adds `HttpTrigger` and `HttpOutput` binding to the function, which enables your function be triggered when http requests hit the specified route.

To write to an Azure Storage queue from this function, add the `queue_output` decorator to your function code:

Python

```
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
```

In the decorator, `arg_name` identifies the binding parameter referenced in your code, `queue_name` is name of the queue that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Storage account. In quickstarts you use the same storage account as the function app, which is in the `AzureWebJobsStorage` setting (from `local.settings.json` file). When the `queue_name` doesn't exist, the binding creates it on first use.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Update `HttpExample\function_app.py` to match the following code, add the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="HttpExample")
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
def HttpExample(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) ->
func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        msg.set(name)
        return func.HttpResponse(f"Hello, {name}. This HTTP triggered
function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
            status_code=200
        )
```

The `msg` parameter is an instance of the `azure.functions.Out class`. The `set` method writes a string message to the queue. In this case, it's the `name` passed to the function in the URL query string.

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

```
Console  
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools  
Core Tools Version: 4.0.5049 Commit hash: N/A (64-bit)  
Function Runtime Version: 4.15.2.20177  
  
[2023-03-17T03:27:12.372Z] Worker process started and initialized.  
  
Functions:  
  
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

💡 Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[1.*, 2.0.0)"  
    }  
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in *host.json*.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's *local.setting.json* file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

bash

Bash

```
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>"
```

2. (Optional) Use the [az storage queue list](#) command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

Azure CLI

```
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

bash

Azure CLI

```
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}')` | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the `LocalFunctionsProj` folder, use the `func azure functionapp publish` command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`.

The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the

app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles is already enabled in the `host.json` file at the root of the project, which should look like the following example:

```
JSON

{
  "version": "2.0",
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      }
    }
  },
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.0.0)"
  }
}
```

```
    }  
}
```

Now, you can add the storage output binding to your project.

Add an output binding

To write to an Azure Storage queue:

- Add an `extraOutputs` property to the binding configuration

TypeScript

```
{  
  methods: ['GET', 'POST'],  
  extraOutputs: [sendToQueue], // add output binding to HTTP trigger  
  authLevel: 'anonymous',  
  handler: () => {}  
}
```

- Add a `output.storageQueue` function above the `app.http` call

TypeScript

```
const sendToQueue: StorageQueueOutput = output.storageQueue({  
  queueName: 'outqueue',  
  connection: 'AzureWebJobsStorage',  
});
```

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Add code that uses the output binding object on `context.extraOutputs` to create a queue message. Add this code before the return statement.

TypeScript

```
context.extraOutputs.set(sendToQueue, [msg]);
```

At this point, your function could look as follows:

TypeScript

```
import {
  app,
  output,
  HttpRequest,
  HttpResponseMessage,
  InvocationContext,
  StorageQueueOutput,
} from '@azure/functions';

const sendToQueue: StorageQueueOutput = output.storageQueue({
  queueName: 'outqueue',
  connection: 'AzureWebJobsStorage',
});

export async function HttpExample(
  request: HttpRequest,
  context: InvocationContext,
): Promise<HttpResponseInit> {
  try {
    context.log(`Http function processed request for url "${request.url}"`);

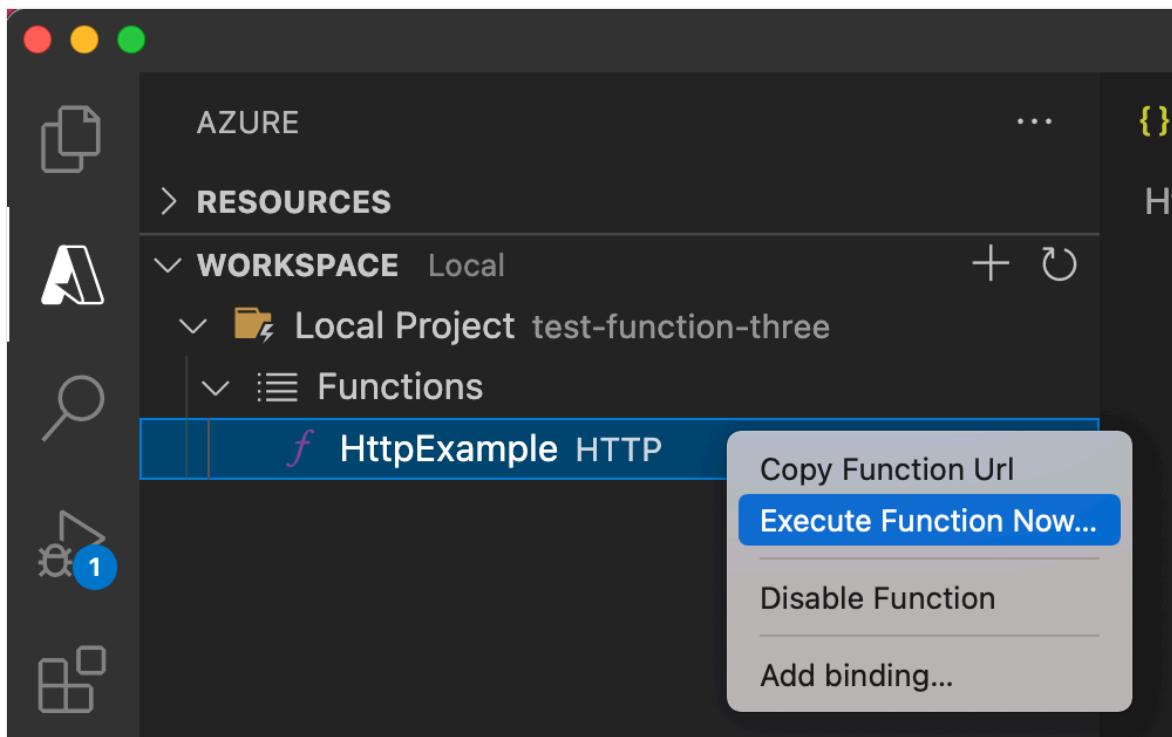
    const name = request.query.get('name') || (await request.text());
    context.log(`Name: ${name}`);

    if (name) {
      const msg = `Name passed to the function ${name}`;
      context.extraOutputs.set(sendToQueue, [msg]);
      return { body: msg };
    } else {
      context.log('Missing required data');
      return { status: 404, body: 'Missing required data' };
    }
  } catch (error) {
    context.log(`Error: ${error}`);
    return { status: 500, body: 'Internal Server Error' };
  }
}

app.http('HttpExample', {
  methods: ['GET', 'POST'],
  authLevel: 'anonymous',
  handler: HttpExample,
});
```

Run the function locally

1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and select **Execute Function Now....**



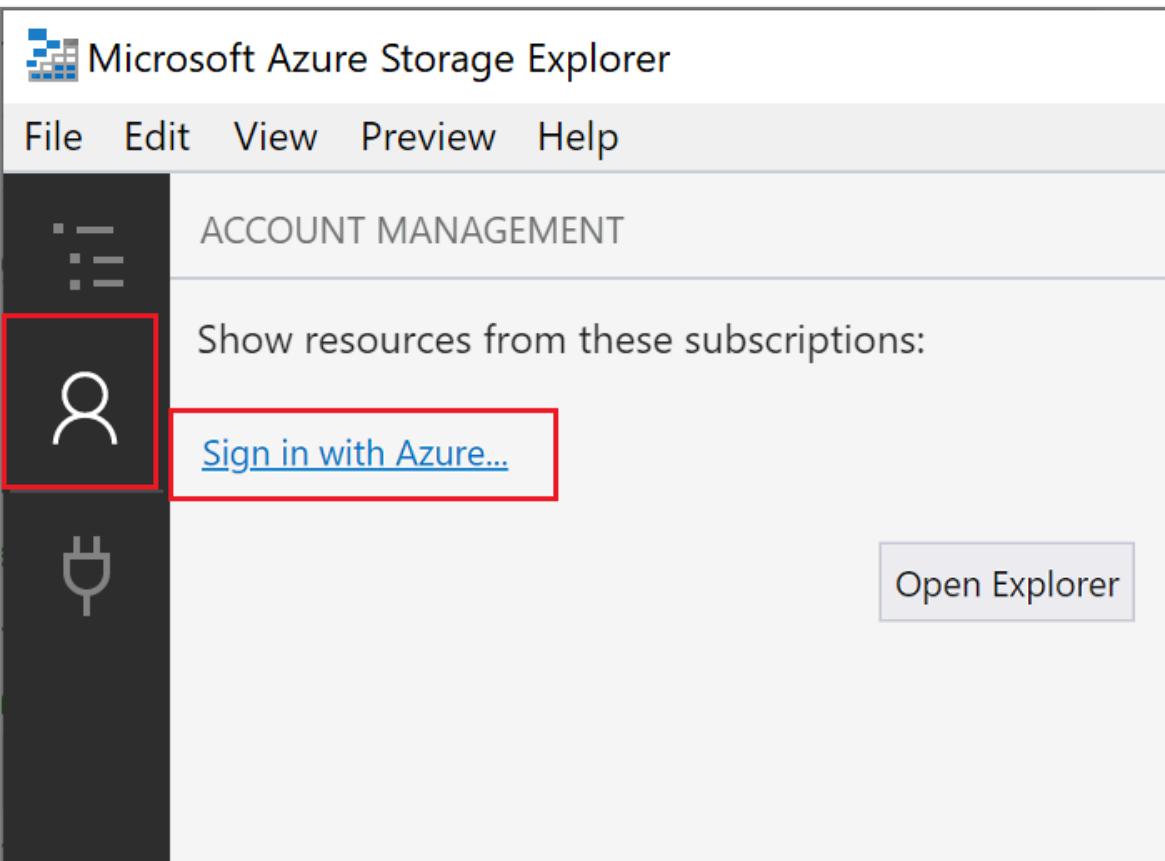
3. In the **Enter request body**, you see the request message body value of `{ "name": "Azure" }`. Press `Enter` to send this request message to your function.
4. After a response is returned, press `Ctrl + C` to stop Core Tools.

Because you're using the storage connection string, your function connects to the Azure storage account when running locally. A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

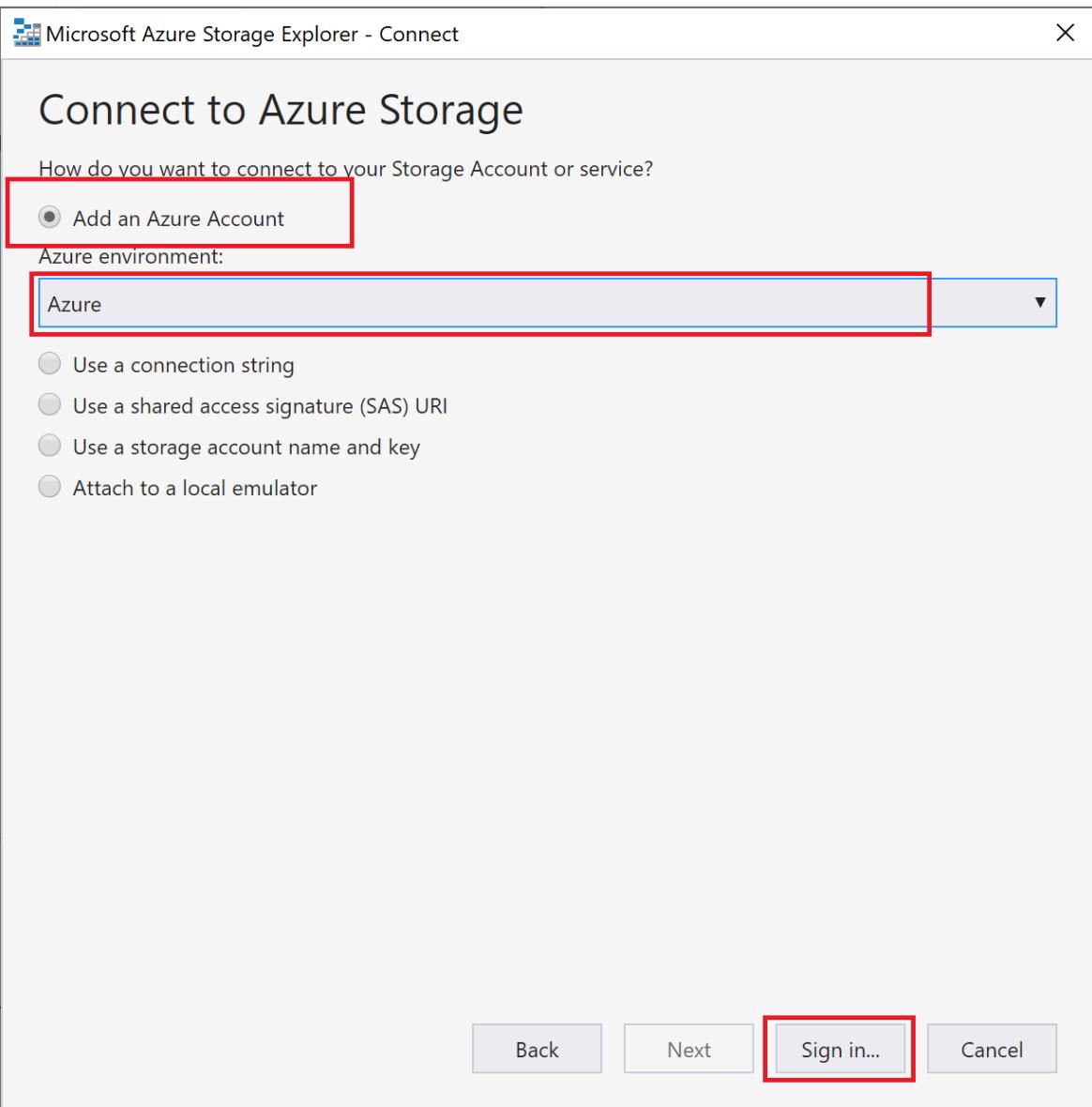
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

1. Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

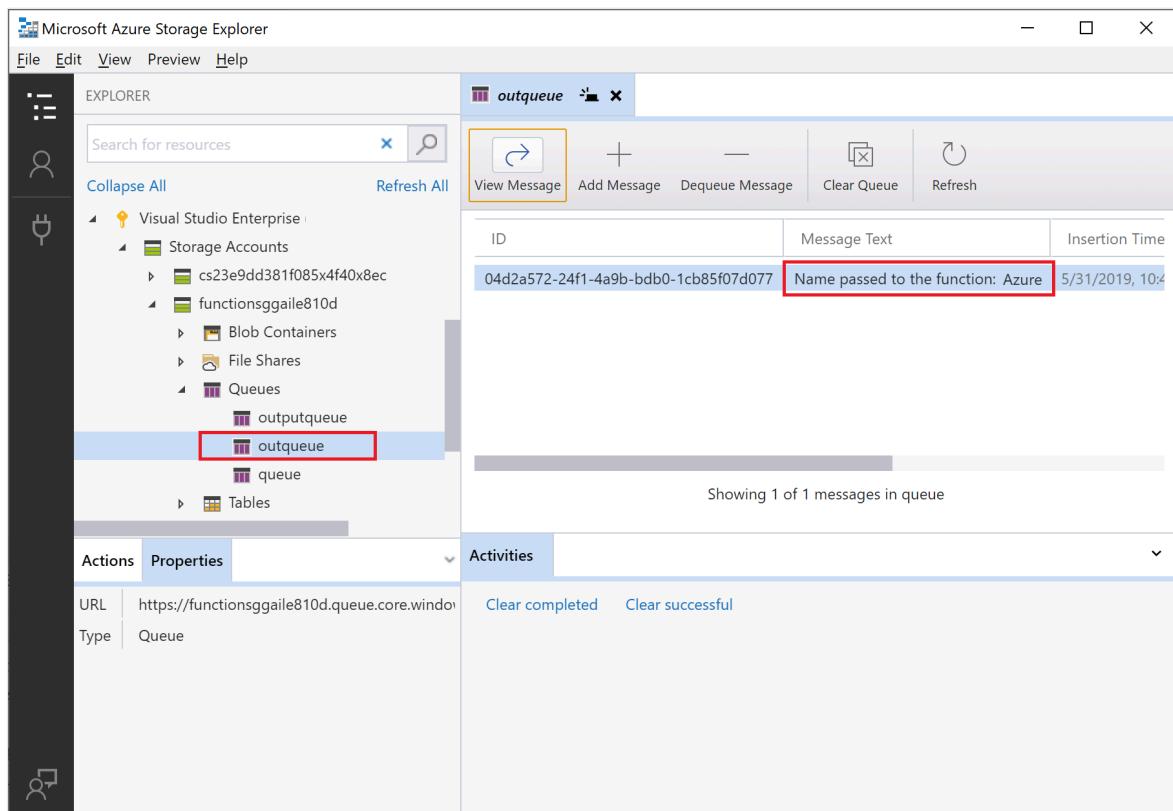


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

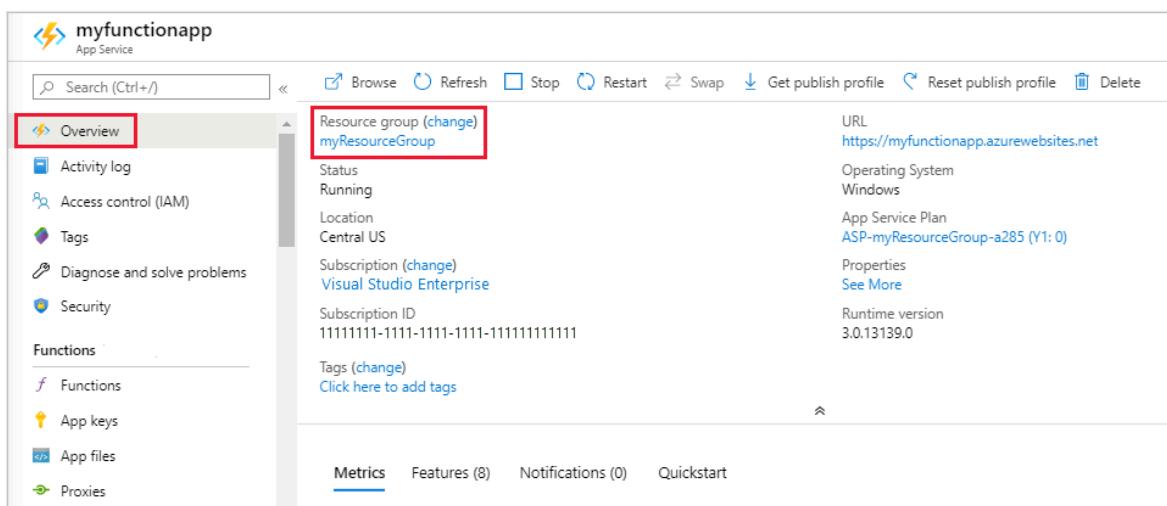
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the "Overview" tab, which displays details about the app: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (with a link to add tags). A red box highlights the "Resource group (change)" link, which points to "myResourceGroup". On the right, there are sections for URL (https://myfunctionapp.azurewebsites.net), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties (See More), and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in TypeScript](#).
- [Azure Functions TypeScript developer guide](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

When using the [Node.js v4 programming model](#), binding attributes are defined directly in the `./src/functions/HttpExample.js` file. From the previous quickstart, your file already contains an HTTP binding defined by the `app.http` method.

TypeScript

```
import {
  app,
  HttpRequest,
  HttpResponseMessage,
  InvocationContext,
} from '@azure/functions';

export async function httpTrigger1(
  request: HttpRequest,
  context: InvocationContext,
): Promise<HttpResponseMessage> {
  context.log(`Http function processed request for url "${request.url}"`);

  const name = request.query.get('name') || (await request.text()) ||
  'world';

  return { body: `Hello, ${name}!` };
}

app.http('httpTrigger1', {
  methods: ['GET', 'POST'],
  authLevel: 'anonymous',
  handler: httpTrigger1,
});
```

To write to an Azure Storage queue:

- Add an `extraOutputs` property to the binding configuration

TypeScript

```
{  
  methods: ['GET', 'POST'],  
  extraOutputs: [sendToQueue], // add output binding to HTTP trigger  
  authLevel: 'anonymous',  
  handler: () => {}  
}
```

- Add a `output.storageQueue` function above the `app.http` call

TypeScript

```
const sendToQueue: StorageQueueOutput = output.storageQueue({  
  queueName: 'outqueue',  
  connection: 'AzureWebJobsStorage',  
});
```

For a `queue` type, you must specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from `local.settings.json` file) in `connection`.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Add code that uses the output binding object on `context.extraOutputs` to create a queue message. Add this code before the return statement.

TypeScript

```
context.extraOutputs.set(sendToQueue, [msg]);
```

At this point, your function could look as follows:

TypeScript

```
import {  
  app,  
  output,  
  HttpRequest,  
  HttpResponseMessage,  
  InvocationContext,
```

```

StorageQueueOutput,
} from '@azure/functions';

const sendToQueue: StorageQueueOutput = output.storageQueue({
  queueName: 'outqueue',
  connection: 'AzureWebJobsStorage',
});

export async function HttpExample(
  request: HttpRequest,
  context: InvocationContext,
): Promise<HttpResponseInit> {
  try {
    context.log(`Http function processed request for url "${request.url}"`);

    const name = request.query.get('name') || (await request.text());
    context.log(`Name: ${name}`);

    if (name) {
      const msg = `Name passed to the function ${name}`;
      context.extraOutputs.set(sendToQueue, [msg]);
      return { body: msg };
    } else {
      context.log('Missing required data');
      return { status: 404, body: 'Missing required data' };
    }
  } catch (error) {
    context.log(`Error: ${error}`);
    return { status: 500, body: 'Internal Server Error' };
  }
}

app.http('HttpExample', {
  methods: ['GET', 'POST'],
  authLevel: 'anonymous',
  handler: HttpExample,
});

```

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

Console

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:      4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press **Ctrl + C** and type **y** to stop the functions host.

💡 Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in `host.json`.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's `local.setting.json` file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

```
bash
Bash
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>
```

2. (Optional) Use the `az storage queue list` command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
Azure CLI
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

```
bash
Azure CLI
```

```
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}') | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the `LocalFunctionsProj` folder, use the [func azure functionapp publish](#) command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`. The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in TypeScript.](#)
- [Azure Functions TypeScript developer guide](#)

Connect Azure Functions to Azure Cosmos DB using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio Code to connect [Azure Cosmos DB](#) to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a JSON document stored in an Azure Cosmos DB container.

Before you begin, you must complete the [quickstart: Create a C# function in Azure using Visual Studio Code](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Configure your environment

Before you get started, make sure to install the [Azure Databases extension](#) for Visual Studio Code.

Create your Azure Cosmos DB account

Now, you create an Azure Cosmos DB account as a [serverless account type](#). This consumption-based mode makes Azure Cosmos DB a strong option for serverless workloads.

1. In Visual Studio Code, select **View > Command Palette...** then in the command palette search for `Azure Databases: Create Server...`
2. Provide the following information at the prompts:

[] [Expand table](#)

Prompt	Selection
Select an Azure Database Server	Choose Core (NoSQL) to create a document database that you can query by using a SQL syntax or a Query Copilot (Preview) converting natural language prompts to queries. Learn more about the Azure Cosmos DB .
Account name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
Select a capacity model	Select Serverless to create an account in serverless mode .
Select a resource group for new resources	Choose the resource group where you created your function app in the previous article .
Select a location for new resources	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to you or your users to get the fastest access to your data.

After your new account is provisioned, a message is displayed in notification area.

Create an Azure Cosmos DB database and container

1. Select the Azure icon in the Activity bar, expand **Resources > Azure Cosmos DB**, right-click (Ctrl+select on macOS) your account, and select **Create database...**
2. Provide the following information at the prompts:

[\[+\] Expand table](#)

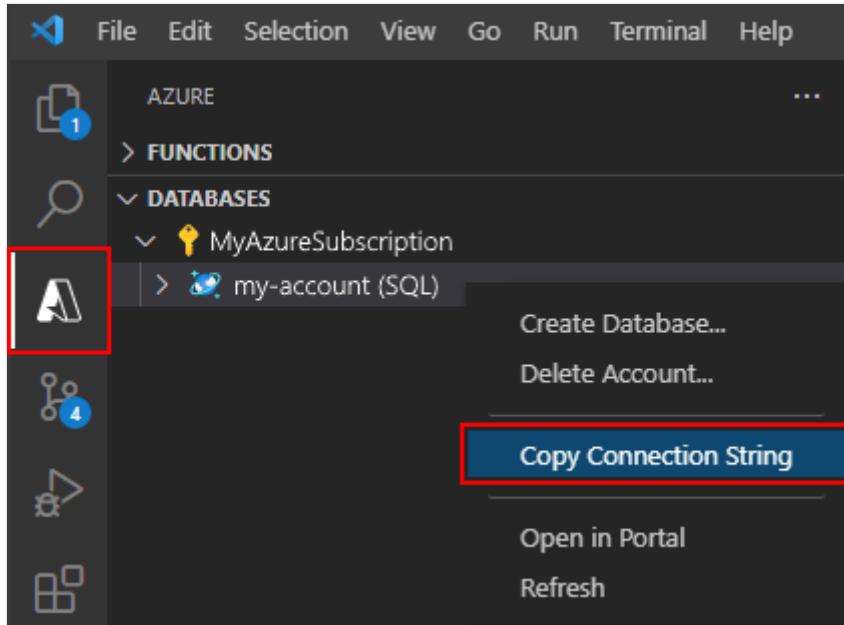
Prompt	Selection
Database name	Type <code>my-database</code> .
Enter and ID for your collection	Type <code>my-container</code> .
Enter the partition key for the collection	Type <code>/id</code> as the partition key .

3. Select **OK** to create the container and database.

Update your function app settings

In the [previous quickstart article](#), you created a function app in Azure. In this article, you update your app to write JSON documents to the Azure Cosmos DB container you've created. To connect to your Azure Cosmos DB account, you must add its connection string to your app settings. You then download the new setting to your local.settings.json file so you can connect to your Azure Cosmos DB account when running locally.

1. In Visual Studio Code, right-click (Ctrl+select on macOS) on your new Azure Cosmos DB account, and select **Copy Connection String**.



2. Press **F1** to open the command palette, then search for and run the command `Azure Functions: Add New Setting...`

3. Choose the function app you created in the previous article. Provide the following information at the prompts:

[+] Expand table

Prompt	Selection
Enter new app setting name	Type <code>CosmosDbConnectionSetting</code> .
Enter value for "CosmosDbConnectionSetting"	Paste the connection string of your Azure Cosmos DB account you copied. You can also configure Microsoft Entra identity as an alternative.

This creates an application setting named connection `CosmosDbConnectionSetting` in your function app in Azure. Now, you can download this setting to your local.settings.json file.

4. Press **F1** again to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`.
5. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

This downloads all of the setting from Azure to your local project, including the new connection string setting. Most of the downloaded settings aren't used when running locally.

Register binding extensions

Because you're using an Azure Cosmos DB output binding, you must have the corresponding bindings extension installed before you run the project.

Except for HTTP and timer triggers, bindings are implemented as extension packages. Run the following `dotnet add package` command in the Terminal window to add the Azure Cosmos DB extension package to your project.

```
command
```

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.CosmosDB
```

Now, you can add the Azure Cosmos DB output binding to your project.

Add an output binding

In a C# class library project, the bindings are defined as binding attributes on the function method.

Open the `HttpExample.cs` project file and add the following classes:

```
C#
```

```
public class MultiResponse
{
    [CosmosDBOutput("my-database", "my-container",
        Connection = "CosmosDbConnectionString", CreateIfNotExists = true)]
    public MyDocument Document { get; set; }
    public HttpResponseMessage HttpResponseMessage { get; set; }
}
public class MyDocument {
    public string id { get; set; }
```

```
    public string message { get; set; }  
}
```

The `MyDocument` class defines an object that gets written to the database. The connection string for the Storage account is set by the `Connection` property. In this case, you could omit `Connection` because you're already using the default storage account.

The `MultiResponse` class allows you to both write to the specified collection in the Azure Cosmos DB and return an HTTP success message. Because you need to return a `MultiResponse` object, you need to also update the method signature.

Specific attributes specify the name of the container and the name of its parent database. The connection string for your Azure Cosmos DB account is set by the `CosmosDbConnectionStringSetting`.

Add code that uses the output binding

Replace the existing Run method with the following code:

C#

```
[Function("HttpExample")]  
public static MultiResponse Run([HttpTrigger(AuthorizationLevel.Anonymous,  
"get", "post")] HttpRequestData req,  
    FunctionContext executionContext)  
{  
    var logger = executionContext.GetLogger("HttpExample");  
    logger.LogInformation("C# HTTP trigger function processed a request.");  
  
    var message = "Welcome to Azure Functions!";  
  
    var response = req.CreateResponse(HttpStatusCode.OK);  
    response.Headers.Add("Content-Type", "text/plain; charset=utf-8");  
    response.WriteString(message);  
  
    // Return a response to both HTTP trigger and Azure Cosmos DB output  
    // binding.  
    return new MultiResponse()  
    {  
        Document = new MyDocument  
        {  
            id = System.Guid.NewGuid().ToString(),  
            message = message  
        },  
        HttpResponseMessage = response  
    };  
}
```

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure. If you don't already have Core Tools installed locally, you are prompted to install it the first time you run your project.

1. To call your function, press **F5** to start the function app project. The **Terminal** panel displays the output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    AZURE: ACTIVITY LOG    ⚙ host start - Task ✓    +    □    🗑    ^    ×

Functions:

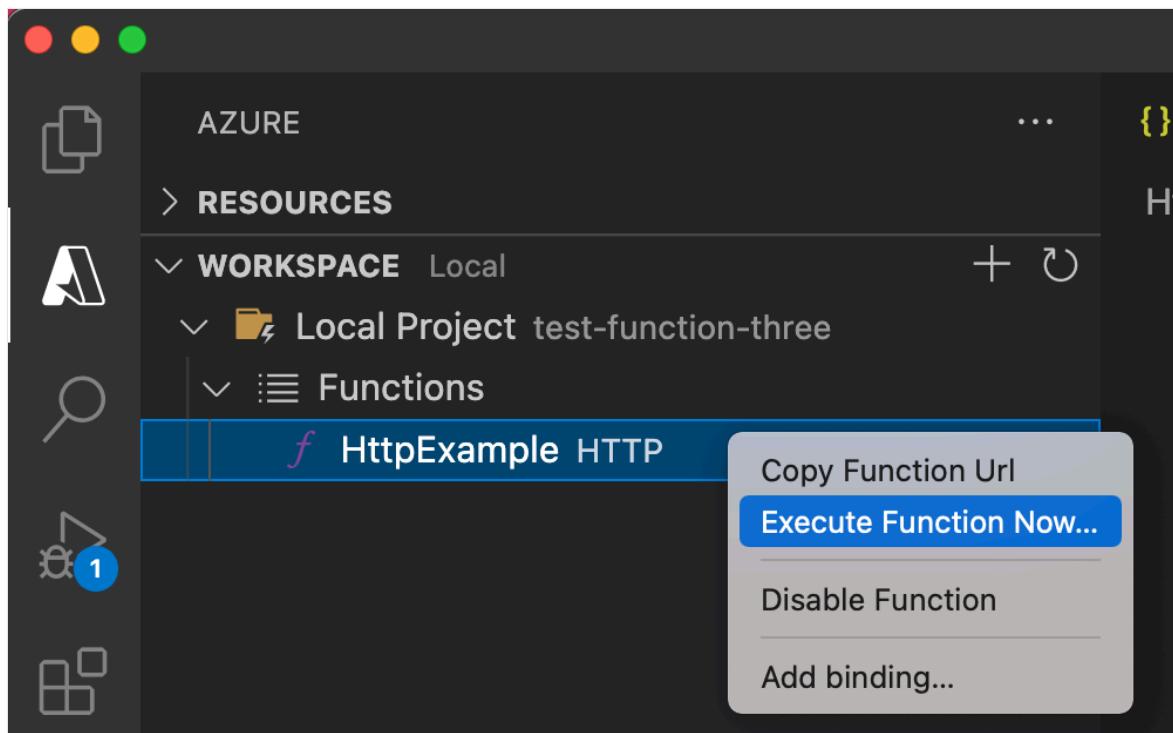
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '00000000000000000000E24CCCAC'.
```

If you don't already have Core Tools installed, select **Install** to install Core Tools when prompted to do so.

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

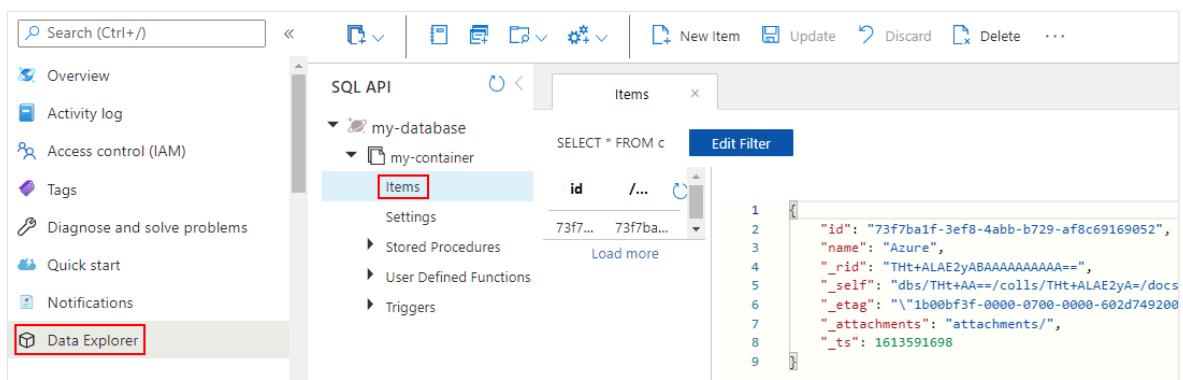
- With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Windows) or `ctrl -` click (macOS) the **HttpExample** function and choose **Execute Function Now....**



3. In the **Enter request body**, press **Enter** to send a request message to your function.
4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in the **Terminal** panel.
5. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

Verify that a JSON document has been created

1. On the Azure portal, go back to your Azure Cosmos DB account and select **Data Explorer**.
2. Expand your database and container, and select **Items** to list the documents created in your container.
3. Verify that a new JSON document has been created by the output binding.



The screenshot shows the Azure portal's Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is highlighted with a red box. The main area shows a hierarchical tree structure under SQL API > my-database > my-container. The 'Items' node is selected and highlighted with a red box. To the right, there's a table with columns 'id' and '_...'. Below the table, there's a code editor window displaying a JSON document:

```

1 "id": "73f7ba1f-3ef8-4abb-b729-af8c69169052",
2 "name": "Azure",
3 "_rid": "THT+ALAE2yABAAAAAAA==",
4 "_self": "dbs/THT+AA==/colls/THT+ALAE2yA=/docs",
5 "_etag": "\\"1b00bf3f-0000-0700-0000-602d749200
6
7 "_attachments": "attachments/",
8
9 "_ts": 1613591698

```

Redeploy and verify the updated app

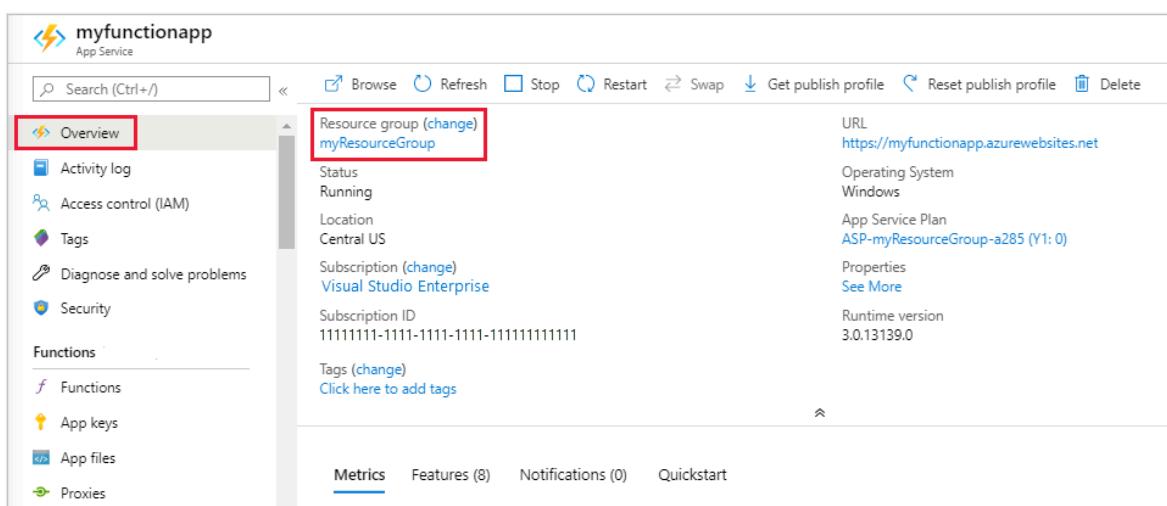
1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [check the documents created in your Azure Cosmos DB container](#) to verify that the output binding again generates a new JSON document.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (Functions, App keys, App files, Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The Overview tab is selected. On the right, there is a summary card with details: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (Click here to add tags). A red box highlights the "Resource group (change)" link under the "myResourceGroup" section. Below the card, there are sections for URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties (See More), and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write JSON documents to an Azure Cosmos DB container. Now you can learn more about developing Functions using Visual Studio Code:

- Develop Azure Functions using Visual Studio Code
- Azure Functions triggers and bindings.
- Examples of complete Function projects in C#.
- Azure Functions C# developer reference

Connect Azure Functions to Azure SQL Database using Visual Studio Code

Article • 04/25/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio Code to connect [Azure SQL Database](#) to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a table in Azure SQL Database.

Before you begin, you must complete the [quickstart: Create a C# function in Azure using Visual Studio Code](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

More details on the settings for [Azure SQL bindings and trigger for Azure Functions](#) are available in the Azure Functions documentation.

Create your Azure SQL Database

1. Follow the [Azure SQL Database create quickstart](#) to create a serverless Azure SQL Database. The database can be empty or created from the sample dataset AdventureWorksLT.
2. Provide the following information at the prompts:

[] Expand table

Prompt	Selection
Resource group	Choose the resource group where you created your function app in the previous article .
Database name	Enter <code>mySampleDatabase</code> .
Server name	Enter a unique name for your server. We can't provide an exact server name to use because server names must be globally

Prompt	Selection
	unique for all servers in Azure, not just unique within a subscription.
Authentication method	Select SQL Server authentication .
Server admin login	Enter <code>azureuser</code> .
Password	Enter a password that meets the complexity requirements.
Allow Azure services and resources to access this server	Select Yes .

3. Once the creation has completed, navigate to the database blade in the Azure portal, and, under **Settings**, select **Connection strings**. Copy the **ADO.NET** connection string for **SQL authentication**. Paste the connection string into a temporary document for later use.

The screenshot shows the 'mySampleDatabase - Connection strings' blade in the Azure portal. On the left, there's a sidebar with various options like Overview, Activity log, Tags, etc., and a 'Connection strings' option which is highlighted with a red box. The main area shows three connection strings under the ADO.NET tab:

- ADO.NET (SQL authentication)**: Contains the connection string: `Server=tcp:myserver99.database.windows.net,1433;Initial Catalog=mySampleDatabase;Persist Security Info=False;User ID=(your_username);Password=(your_password);MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;`
- ADO.NET (Active Directory password authentication)**: Contains the connection string: `Server=tcp:myserver99.database.windows.net,1433;Initial Catalog=mySampleDatabase;Persist Security Info=False;User ID=(your_username);Password=(your_password);MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Authentication="Active Directory Password";`
- ADO.NET (Active Directory integrated authentication)**: Contains the connection string: `Server=tcp:myserver99.database.windows.net,1433;Initial Catalog=mySampleDatabase;Persist Security Info=False;User ID=(your_username);MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Authentication="Active Directory Integrated";`

4. Create a table to store the data from the HTTP request. In the Azure portal, navigate to the database blade and select **Query editor**. Enter the following query to create a table named `dbo.ToDo`:

```
CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

5. Verify that your Azure Function will be able to access the Azure SQL Database by checking the [server's firewall settings](#). Navigate to the **server blade** on the Azure portal, and under **Security**, select **Networking**. The exception for **Allow Azure services and resources to access this server** should be checked.

The screenshot shows the Azure portal interface for managing a SQL server named 'mydocsamplesqlserver'. The left sidebar lists various security features: Microsoft Defender for Cloud, Transparent data encryption, Identity, Auditing, Intelligent Performance, Automatic tuning, and Recommendations. The 'Networking' option is selected and highlighted with a red box. The main content area shows the 'Networking' blade with tabs for Public access, Private access, and Connectivity. Under the 'Public access' tab, there is a section for 'Firewall rules' with options to add client IPv4 addresses or firewall rules. Below this is an 'Exceptions' section, also highlighted with a red box, containing a checkbox labeled 'Allow Azure services and resources to access this server'.

Update your function app settings

In the [previous quickstart article](#), you created a function app in Azure. In this article, you update your app to write data to the Azure SQL Database you've just created. To connect to your Azure SQL Database, you must add its connection string to your app settings. You then download the new setting to your local.settings.json file so you can connect to your Azure SQL Database when running locally.

1. Edit the connection string in the temporary document you created earlier. Replace the value of `Password` with the password you used when creating the Azure SQL Database. Copy the updated connection string.
2. Press `Ctrl/Cmd+shift+P` to open the command palette, then search for and run the command `Azure Functions: Add New Setting...`.
3. Choose the function app you created in the previous article. Provide the following information at the prompts:

[\[+\] Expand table](#)

Prompt	Selection
Enter new app setting name	Type <code>SqlConnectionString</code> .
Enter value for "SqlConnectionString"	Paste the connection string of your Azure SQL Database you just copied.

This creates an application setting named connection `SqlConnectionString` in your function app in Azure. Now, you can download this setting to your `local.settings.json` file.

4. Press `Ctrl/Cmd+shift+P` again to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
5. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

This downloads all of the setting from Azure to your local project, including the new connection string setting. Most of the downloaded settings aren't used when running locally.

Register binding extensions

Because you're using an Azure SQL output binding, you must have the corresponding bindings extension installed before you run the project.

With the exception of HTTP and timer triggers, bindings are implemented as extension packages. Run the following `dotnet add package` command in the Terminal window to add the Azure SQL extension package to your project.

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Sql
```

Now, you can add the Azure SQL output binding to your project.

Add an output binding

In Functions, each type of binding requires a `direction`, `type`, and a unique `name` to be defined in the `function.json` file. The way you define these attributes depends on the language of your function app.

Open the *HttpExample.cs* project file and add the following `ToDoItem` class, which defines the object that is written to the database:

```
C#  
  
namespace AzureSQL.ToDo  
{  
    public class ToDoItem  
    {  
        public Guid Id { get; set; }  
        public int? order { get; set; }  
        public string title { get; set; }  
        public string url { get; set; }  
        public bool? completed { get; set; }  
    }  
}
```

In a C# class library project, the bindings are defined as binding attributes on the function method. The *function.json* file required by Functions is then auto-generated based on these attributes.

Open the *HttpExample.cs* project file and add the following output type class, which defines the combined objects that will be output from our function for both the HTTP response and the SQL output:

```
cs  
  
public static class OutputType  
{  
    [SqlOutput("dbo.ToDo", connectionStringSetting: "SqlConnectionString")]  
    public ToDoItem ToDoItem { get; set; }  
    public HttpResponseMessage HttpResponseMessage { get; set; }  
}
```

Add a using statement to the `Microsoft.Azure.Functions.Worker.Extensions.Sql` library to the top of the file:

```
cs  
  
using Microsoft.Azure.Functions.Worker.Extensions.Sql;
```

Add code that uses the output binding

Replace the existing Run method with the following code:

CS

```
[Function("HttpExample")]
public static OutputType Run([HttpTrigger(AuthorizationLevel.Anonymous,
"get", "post")] HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpExample");
    logger.LogInformation("C# HTTP trigger function processed a request.");

    var message = "Welcome to Azure Functions!";

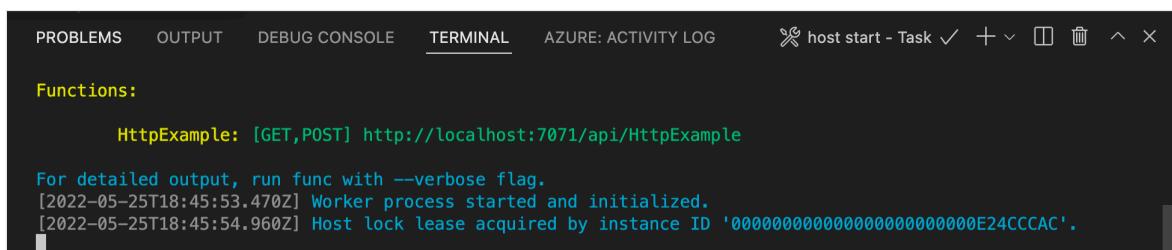
    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-8");
    response.WriteString(message);

    // Return a response to both HTTP trigger and Azure SQL output binding.
    return new OutputType()
    {
        ToDoItem = new ToDoItem
        {
            id = System.Guid.NewGuid().ToString(),
            title = message,
            completed = false,
            url = ""
        },
        HttpResponseMessage = response
    };
}
```

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure. If you don't already have Core Tools installed locally, you are prompted to install it the first time you run your project.

1. To call your function, press **F5** to start the function app project. The **Terminal** panel displays the output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The terminal window displays the following output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG
✖ host start - Task ✓ + □ ⌂ ⌂ ×

Functions:

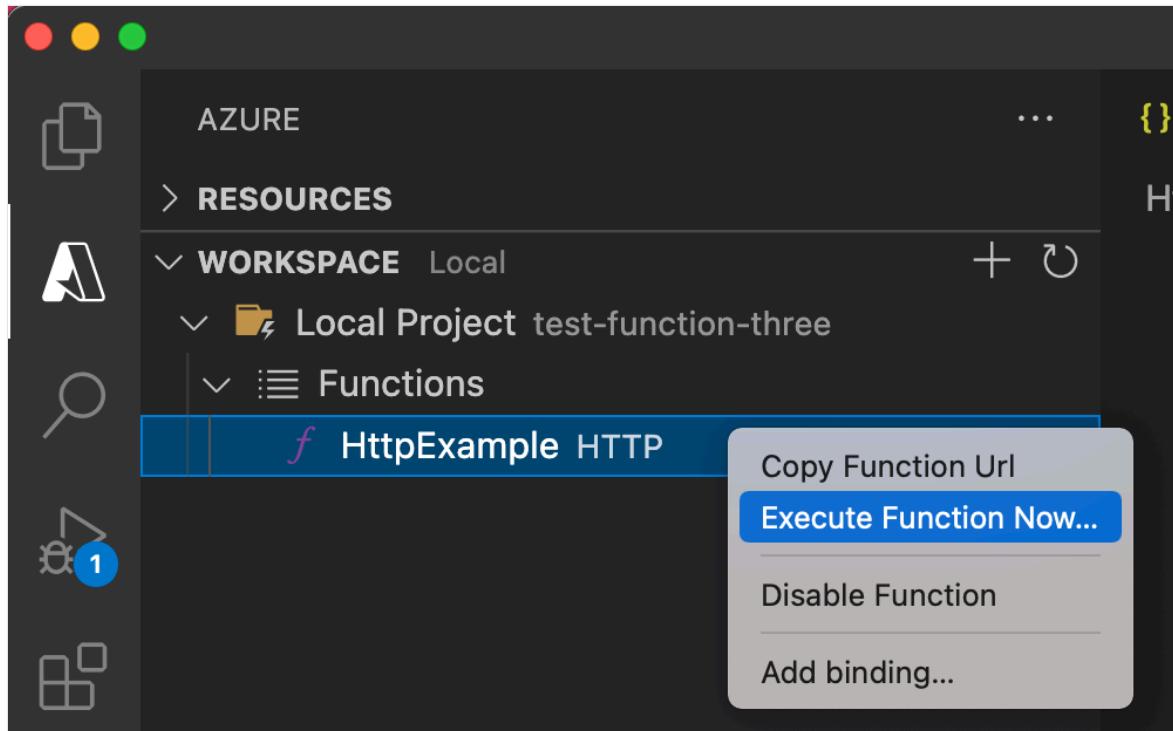
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '00000000000000000000E24CCCAC'.
```

If you don't already have Core Tools installed, select **Install** to install Core Tools when prompted to do so.

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

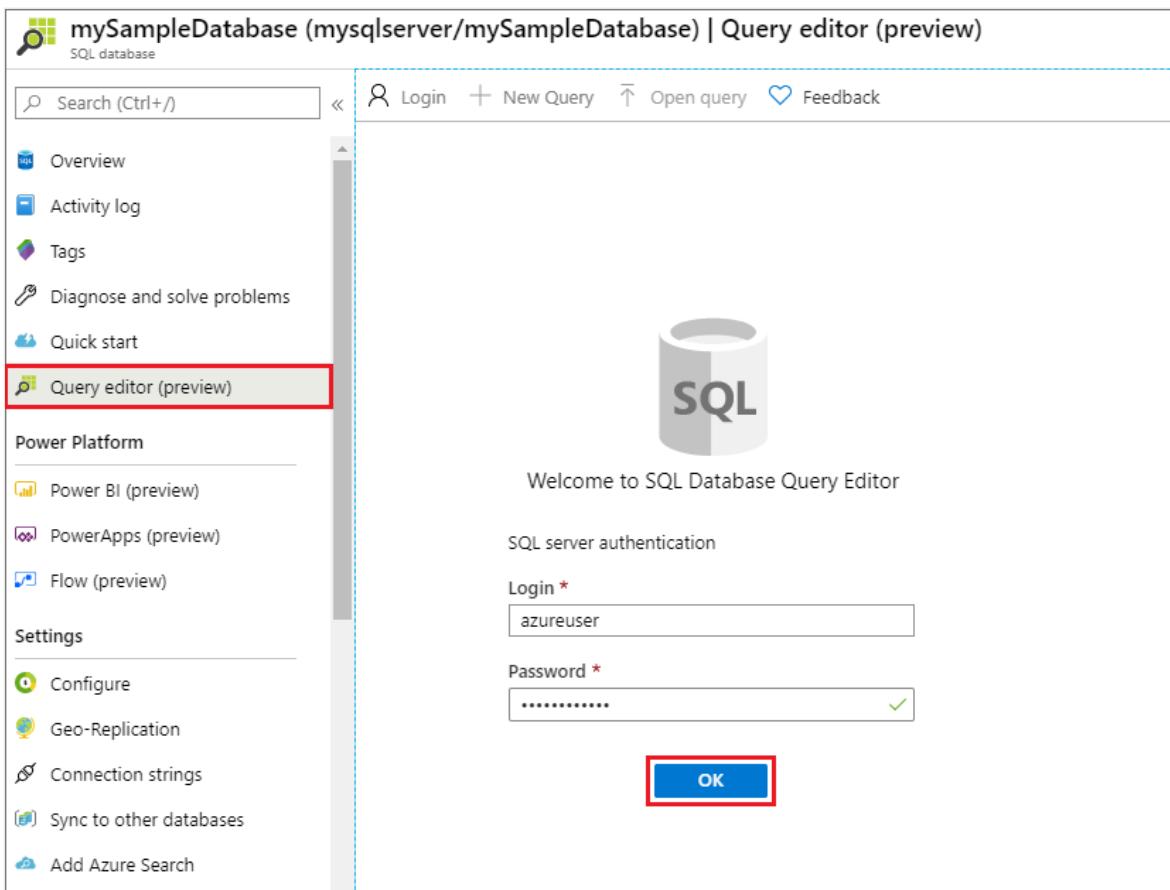
- With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Windows) or `ctrl -` click (macOS) the `HttpExample` function and choose **Execute Function Now....**



- In the **Enter request body**, press `Enter` to send a request message to your function.
- When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in the **Terminal** panel.
- Press `Ctrl + C` to stop Core Tools and disconnect the debugger.

Verify that information has been written to the database

- On the Azure portal, go back to your Azure SQL Database and select **Query editor**.



2. Connect to your database and expand the **Tables** node in object explorer on the left. Right-click on the `dbo.ToDo` table and select **Select Top 1000 Rows**.
3. Verify that the new information has been written to the database by the output binding.

Redeploy and verify the updated app

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Deploy to function app....`.
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [check the data written to your Azure SQL Database](#) to verify that the output binding again generates a new JSON document.

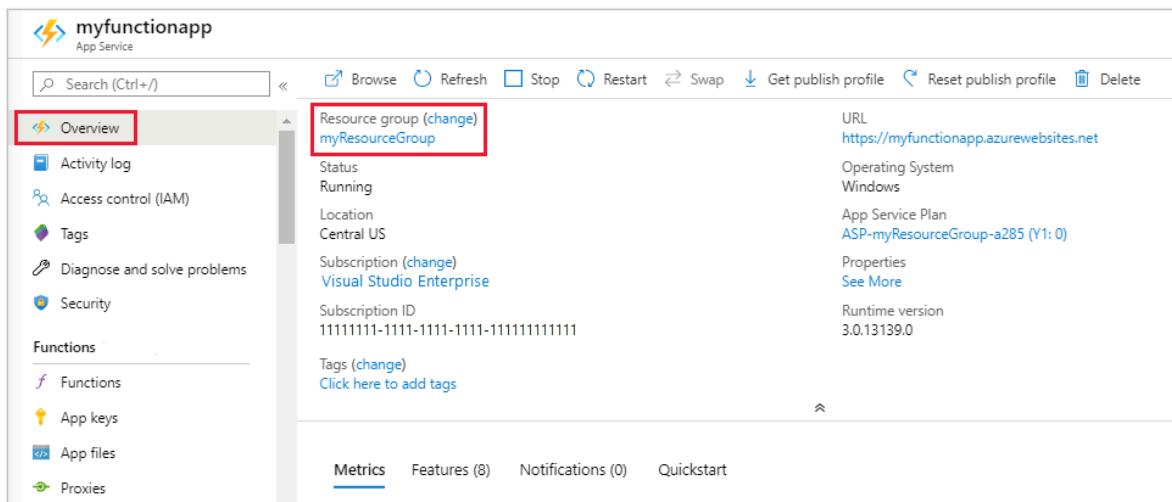
Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth.

They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to Azure SQL Database. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure SQL bindings and trigger for Azure Functions](#)

- Azure Functions triggers and bindings.
- Examples of complete Function projects in C#.
- Azure Functions C# developer reference

Connect Azure Functions to Azure Cosmos DB using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio Code to connect [Azure Cosmos DB](#) to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a JSON document stored in an Azure Cosmos DB container.

Before you begin, you must complete the [quickstart: Create a JavaScript function in Azure using Visual Studio Code](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

ⓘ Note

This article currently only supports [Node.js v3 for Functions](#).

Configure your environment

Before you get started, make sure to install the [Azure Databases extension](#) for Visual Studio Code.

Create your Azure Cosmos DB account

Now, you create an Azure Cosmos DB account as a [serverless account type](#). This consumption-based mode makes Azure Cosmos DB a strong option for serverless workloads.

1. In Visual Studio Code, select **View > Command Palette...** then in the command palette search for `Azure Databases: Create Server...`

- Provide the following information at the prompts:

[+] Expand table

Prompt	Selection
Select an Azure Database Server	Choose Core (NoSQL) to create a document database that you can query by using a SQL syntax or a Query Copilot (Preview) converting natural language prompts to queries. Learn more about the Azure Cosmos DB.
Account name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
Select a capacity model	Select Serverless to create an account in serverless mode.
Select a resource group for new resources	Choose the resource group where you created your function app in the previous article .
Select a location for new resources	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to you or your users to get the fastest access to your data.

After your new account is provisioned, a message is displayed in notification area.

Create an Azure Cosmos DB database and container

- Select the Azure icon in the Activity bar, expand **Resources > Azure Cosmos DB**, right-click (Ctrl+select on macOS) your account, and select **Create database...**
- Provide the following information at the prompts:

[+] Expand table

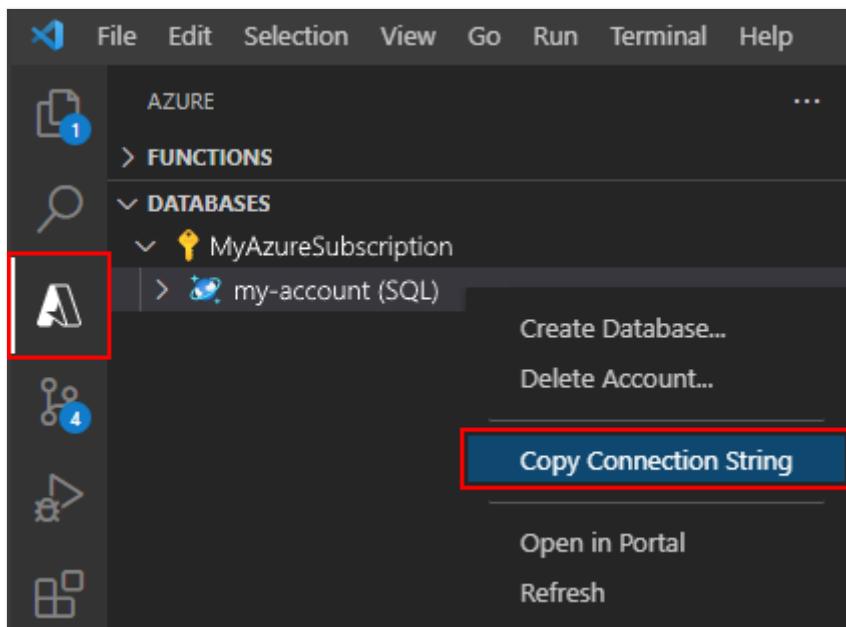
Prompt	Selection
Database name	Type <code>my-database</code> .
Enter and ID for your collection	Type <code>my-container</code> .
Enter the partition key for the collection	Type <code>/id</code> as the partition key .

- Select **OK** to create the container and database.

Update your function app settings

In the [previous quickstart article](#), you created a function app in Azure. In this article, you update your app to write JSON documents to the Azure Cosmos DB container you've created. To connect to your Azure Cosmos DB account, you must add its connection string to your app settings. You then download the new setting to your local.settings.json file so you can connect to your Azure Cosmos DB account when running locally.

1. In Visual Studio Code, right-click (Ctrl+select on macOS) on your new Azure Cosmos DB account, and select **Copy Connection String**.



2. Press **F1** to open the command palette, then search for and run the command `Azure Functions: Add New Setting....`
3. Choose the function app you created in the previous article. Provide the following information at the prompts:

 Expand table

Prompt	Selection
Enter new app setting name	Type <code>CosmosDbConnectionSetting</code> .
Enter value for "CosmosDbConnectionSetting"	Paste the connection string of your Azure Cosmos DB account you copied. You can also configure Microsoft Entra identity as an alternative.

This creates an application setting named connection `CosmosDbConnectionSetting` in your function app in Azure. Now, you can download this setting to your

`local.settings.json` file.

4. Press **F1** again to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
5. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

This downloads all of the setting from Azure to your local project, including the new connection string setting. Most of the downloaded settings aren't used when running locally.

Register binding extensions

Because you're using an Azure Cosmos DB output binding, you must have the corresponding bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles usage is enabled in the `host.json` file at the root of the project, which appears as follows:

```
JSON

{
  "version": "2.0",
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      }
    }
  },
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.0.0)"
  },
  "concurrency": {
    "dynamicConcurrencyEnabled": true,
    "snapshotPersistenceEnabled": true
  },
  "extensions": {
    "cosmosDB": {
      "connectionMode": "Gateway"
    }
  }
}
```

```
}
```

Now, you can add the Azure Cosmos DB output binding to your project.

Add an output binding

Binding attributes are defined directly in your function code. The [Azure Cosmos DB output configuration](#) describes the fields required for an Azure Cosmos DB output binding.

For this `MultiResponse` scenario, you need to add an `extraOutputs` output binding to the function.

JavaScript

```
app.http('HttpExample', {
  methods: ['GET', 'POST'],
  extraOutputs: [sendToCosmosDb],
  handler: async (request, context) => {
```

Add the following properties to the binding configuration:

JavaScript

```
const sendToCosmosDb = output.cosmosDB({
  databaseName: 'my-database',
  containerName: 'my-container',
  createIfNotExists: false,
  connection: 'CosmosDBConnectionString',
});
```

Add code that uses the output binding

Add code that uses the `extraInputs` output binding object on `context` to send a JSON document to the named output binding function, `sendToCosmosDb`. Add this code before the `return` statement.

JavaScript

```
context.extraOutputs.set(sendToCosmosDb, {
  // create a random ID
  id:
    new Date().toISOString() + Math.random().toString().substring(2, 10),
```

```
    name: name,  
});
```

At this point, your function should look as follows:

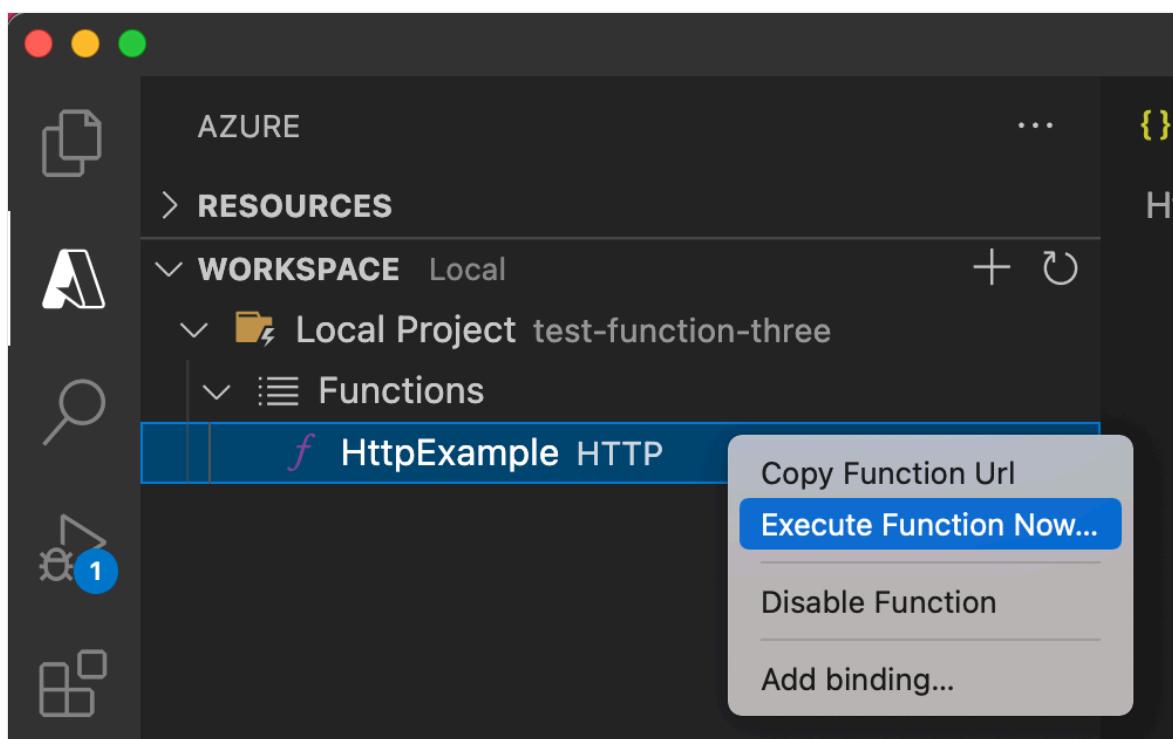
```
JavaScript  
  
const { app, output } = require('@azure/functions');  
  
const sendToCosmosDb = output.cosmosDB({  
  databaseName: 'my-database',  
  containerName: 'my-container',  
  createIfNotExists: false,  
  connection: 'CosmosDBConnectionString',  
});  
  
app.http('HttpExampleToCosmosDB', {  
  methods: ['GET', 'POST'],  
  extraOutputs: [sendToCosmosDb],  
  handler: async (request, context) => {  
    try {  
      context.log(`Http function processed request for url  
"${request.url}"`);  
  
      const name = request.query.get('name') || (await request.text());  
  
      if (!name) {  
        return { status: 404, body: 'Missing required data' };  
      }  
  
      // Output to Database  
      context.extraOutputs.set(sendToCosmosDb, {  
        // create a random ID  
        id:  
          new Date().toISOString() + Math.random().toString().substring(2,  
10),  
        name: name,  
      });  
  
      const responseMessage = name  
        ? `Hello, ${name}  
        . This HTTP triggered function executed successfully.  
        : 'This HTTP triggered function executed successfully. Pass a name  
        in the query string or in the request body for a personalized response.';  
  
      // Return to HTTP client  
      return { body: responseMessage };  
    } catch (error) {  
      context.log(`Error: ${error}`);  
      return { status: 500, body: 'Internal Server Error' };  
    }  
  }  
});
```

```
 },  
});
```

This code now returns a `MultiResponse` object that contains both a document and an HTTP response.

Run the function locally

1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and choose **Execute Function Now...**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. After a response is returned, press `Ctrl + C` to stop Core Tools.

Verify that a JSON document has been created

1. On the Azure portal, go back to your Azure Cosmos DB account and select **Data Explorer**.

2. Expand your database and container, and select **Items** to list the documents created in your container.

3. Verify that a new JSON document has been created by the output binding.

The screenshot shows the Azure portal interface with the 'Data Explorer' blade open. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The 'Data Explorer' option is highlighted with a red box. The main area shows the 'SQL API' for a database named 'my-database' and a container named 'my-container'. The 'Items' tab is selected, and a single document is listed with the ID '73f7ba1f-3ef8-4abb-b729-af8c69169052'. The JSON content of the document is displayed in a code editor-like pane:

```
1 "id": "73f7ba1f-3ef8-4abb-b729-af8c69169052",
2 "name": "Azure",
3 "_rid": "THT+ALAE2yABAAAAAA==",
4 "_self": "dbs/THT+AA==/colls/THT+ALAE2yA=/docs",
5 "_etag": "\\"1b00bf3f-0000-0700-0000-602d749200",
6 "_attachments": "attachments/",
7 "_ts": 1613591698
```

Redeploy and verify the updated app

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app...**.

2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.

3. After deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.

4. Again [check the documents created in your Azure Cosmos DB container](#) to verify that the output binding again generates a new JSON document.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select **Azure: Open in portal**.

2. Choose your function app and press **Enter**. The function app page opens in the Azure portal.

3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal's Overview page for an App Service named 'myfunctionapp'. The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area shows the following details:

Setting	Value
Status	Running
Location	Central US
Subscription (change)	Visual Studio Enterprise
Subscription ID	11111111-1111-1111-1111-111111111111
Tags (change)	Click here to add tags
URL	https://myfunctionapp.azurewebsites.net
Operating System	Windows
App Service Plan	ASP-myResourceGroup-a285 (Y1: 0)
Properties	See More
Runtime version	3.0.13139.0

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write JSON documents to an Azure Cosmos DB container. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in JavaScript](#).
- [Azure Functions JavaScript developer guide](#)

Connect Azure Functions to Azure SQL Database using Visual Studio Code

Article • 04/25/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio Code to connect [Azure SQL Database](#) to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a table in Azure SQL Database.

Before you begin, you must complete the [quickstart: Create a JavaScript function in Azure using Visual Studio Code](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

More details on the settings for [Azure SQL bindings and trigger for Azure Functions](#) are available in the Azure Functions documentation.

Create your Azure SQL Database

1. Follow the [Azure SQL Database create quickstart](#) to create a serverless Azure SQL Database. The database can be empty or created from the sample dataset AdventureWorksLT.
2. Provide the following information at the prompts:

[+] [Expand table](#)

Prompt	Selection
Resource group	Choose the resource group where you created your function app in the previous article .
Database name	Enter <code>mySampleDatabase</code> .
Server name	Enter a unique name for your server. We can't provide an exact server name to use because server names must be globally

Prompt	Selection
	unique for all servers in Azure, not just unique within a subscription.
Authentication method	Select SQL Server authentication .
Server admin login	Enter <code>azureuser</code> .
Password	Enter a password that meets the complexity requirements.
Allow Azure services and resources to access this server	Select Yes .

3. Once the creation has completed, navigate to the database blade in the Azure portal, and, under **Settings**, select **Connection strings**. Copy the **ADO.NET** connection string for **SQL authentication**. Paste the connection string into a temporary document for later use.

The screenshot shows the 'mySampleDatabase - Connection strings' blade in the Azure portal. The left sidebar includes links for Overview, Activity log, Tags, Diagnose and solve problems, Quick start, Query editor (preview), Settings (Configure, Geo-Replication), and Connection strings. The 'Connection strings' link is highlighted with a red box. The main area shows tabs for ADO.NET, JDBC, ODBC, and PHP. The ADO.NET tab is active, displaying the 'ADO.NET (SQL authentication)' section with a connection string. This string is also highlighted with a red box. Below it are sections for 'ADO.NET (Active Directory password authentication)' and 'ADO.NET (Active Directory integrated authentication)', each with its own connection string. At the bottom, there's a link to 'Download ADO.NET driver for SQL server'.

4. Create a table to store the data from the HTTP request. In the Azure portal, navigate to the database blade and select **Query editor**. Enter the following query to create a table named `dbo.ToDo`:

```
CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

5. Verify that your Azure Function will be able to access the Azure SQL Database by checking the [server's firewall settings](#). Navigate to the **server blade** on the Azure portal, and under **Security**, select **Networking**. The exception for **Allow Azure services and resources to access this server** should be checked.

The screenshot shows the Azure portal interface for managing a SQL server named 'mydocsamplesqlserver'. The left sidebar lists various security features: Microsoft Defender for Cloud, Transparent data encryption, Identity, Auditing, Intelligent Performance, Automatic tuning, and Recommendations. The 'Networking' option is selected and highlighted with a red box. The main content area shows the 'Networking' blade with tabs for Public access, Private access, and Connectivity. Under the 'Public access' tab, there is a section for 'Firewall rules' with options to add client IPv4 addresses or firewall rules. Below this is an 'Exceptions' section, also highlighted with a red box, containing a checkbox labeled 'Allow Azure services and resources to access this server'.

Update your function app settings

In the [previous quickstart article](#), you created a function app in Azure. In this article, you update your app to write data to the Azure SQL Database you've just created. To connect to your Azure SQL Database, you must add its connection string to your app settings. You then download the new setting to your local.settings.json file so you can connect to your Azure SQL Database when running locally.

1. Edit the connection string in the temporary document you created earlier. Replace the value of `Password` with the password you used when creating the Azure SQL Database. Copy the updated connection string.
2. Press `Ctrl/Cmd+shift+P` to open the command palette, then search for and run the command `Azure Functions: Add New Setting...`.
3. Choose the function app you created in the previous article. Provide the following information at the prompts:

[\[+\] Expand table](#)

Prompt	Selection
Enter new app setting name	Type <code>SqlConnectionString</code> .
Enter value for "SqlConnectionString"	Paste the connection string of your Azure SQL Database you just copied.

This creates an application setting named connection `SqlConnectionString` in your function app in Azure. Now, you can download this setting to your `local.settings.json` file.

4. Press `Ctrl/Cmd+shift+P` again to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
5. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

This downloads all of the setting from Azure to your local project, including the new connection string setting. Most of the downloaded settings aren't used when running locally.

Register binding extensions

Because you're using an Azure SQL output binding, you must have the corresponding bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles usage is enabled in the `host.json` file at the root of the project, which appears as follows:

```
JSON

{
  "version": "2.0",
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      }
    }
  },
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.0.0)"
  }
}
```

```
  },
  "concurrency": {
    "dynamicConcurrencyEnabled": true,
    "snapshotPersistenceEnabled": true
  }
}
```

...

Now, you can add the Azure SQL output binding to your project.

Add an output binding

In Functions, each type of binding requires a `direction`, `type`, and a unique `name` to be defined in the `function.json` file. The way you define these attributes depends on the language of your function app.

Binding attributes are defined directly in your code. The [Azure SQL output configuration](#) describes the fields required for an Azure SQL output binding.

For this `MultiResponse` scenario, you need to add an `extraOutputs` output binding to the function.

JavaScript

```
app.http('HttpExample', {
  methods: ['GET', 'POST'],
  extraOutputs: [sendToSql],
  handler: async (request, context) => {
```

Add the following properties to the binding configuration:

JavaScript

```
const sendToSql = output.sql({
  commandText: 'dboToDo',
  connectionStringSetting: 'SqlConnectionString',
});
```

Add code that uses the output binding

Add code that uses the `extraInputs` output binding object on `context` to send a JSON document to the named output binding function, `sendToSql`. Add this code before the `return` statement.

```
JavaScript
```

```
const data = JSON.stringify([
  {
    // create a random ID
    Id: crypto.randomUUID(),
    title: name,
    completed: false,
    url: '',
  },
]);

// Output to Database
context.extraOutputs.set(sendToSql, data);
```

To utilize the `crypto` module, add the following line to the top of the file:

```
JavaScript
```

```
const crypto = require("crypto");
```

At this point, your function should look as follows:

```
JavaScript
```

```
const { app, output } = require('@azure/functions');
const crypto = require('crypto');

const sendToSql = output.sql({
  commandText: 'dbo.ToDo',
  connectionStringSetting: 'SqlConnectionString',
});

app.http('HttpExample', {
  methods: ['GET', 'POST'],
  extraOutputs: [sendToSql],
  handler: async (request, context) => {
    try {
      context.log(`Http function processed request for url
      "${request.url}"`);

      const name = request.query.get('name') || (await request.text());

      if (!name) {
        return { status: 404, body: 'Missing required data' };
      }

      // Stringified array of objects to be inserted into the database
      const data = JSON.stringify([
        {
          // create a random ID
```

```
    Id: crypto.randomUUID(),
    title: name,
    completed: false,
    url: '',
  },
]);

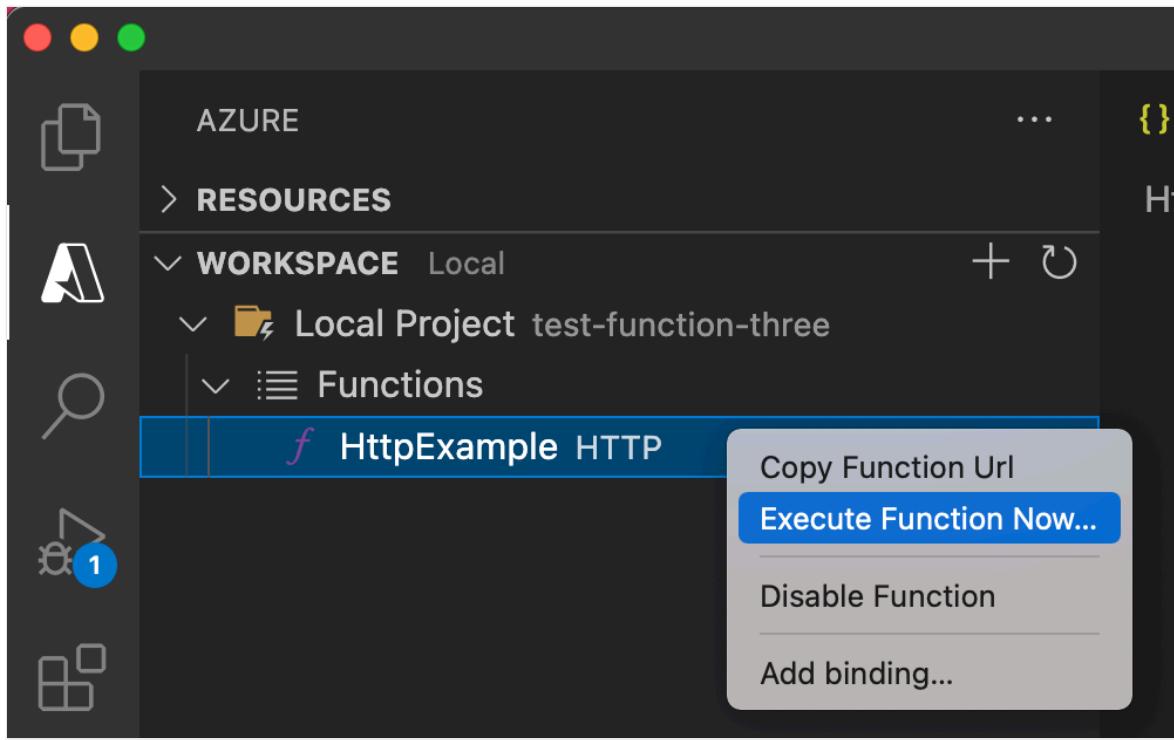
// Output to Database
context.extraOutputs.set(sendToSql, data);

const responseMessage = name
? 'Hello, ' +
  name +
  '. This HTTP triggered function executed successfully.'
: 'This HTTP triggered function executed successfully. Pass a name
in the query string or in the request body for a personalized response.';

// Return to HTTP client
return { body: responseMessage };
} catch (error) {
  context.log(`Error: ${error}`);
  return { status: 500, body: 'Internal Server Error' };
}
},
));
}
```

Run the function locally

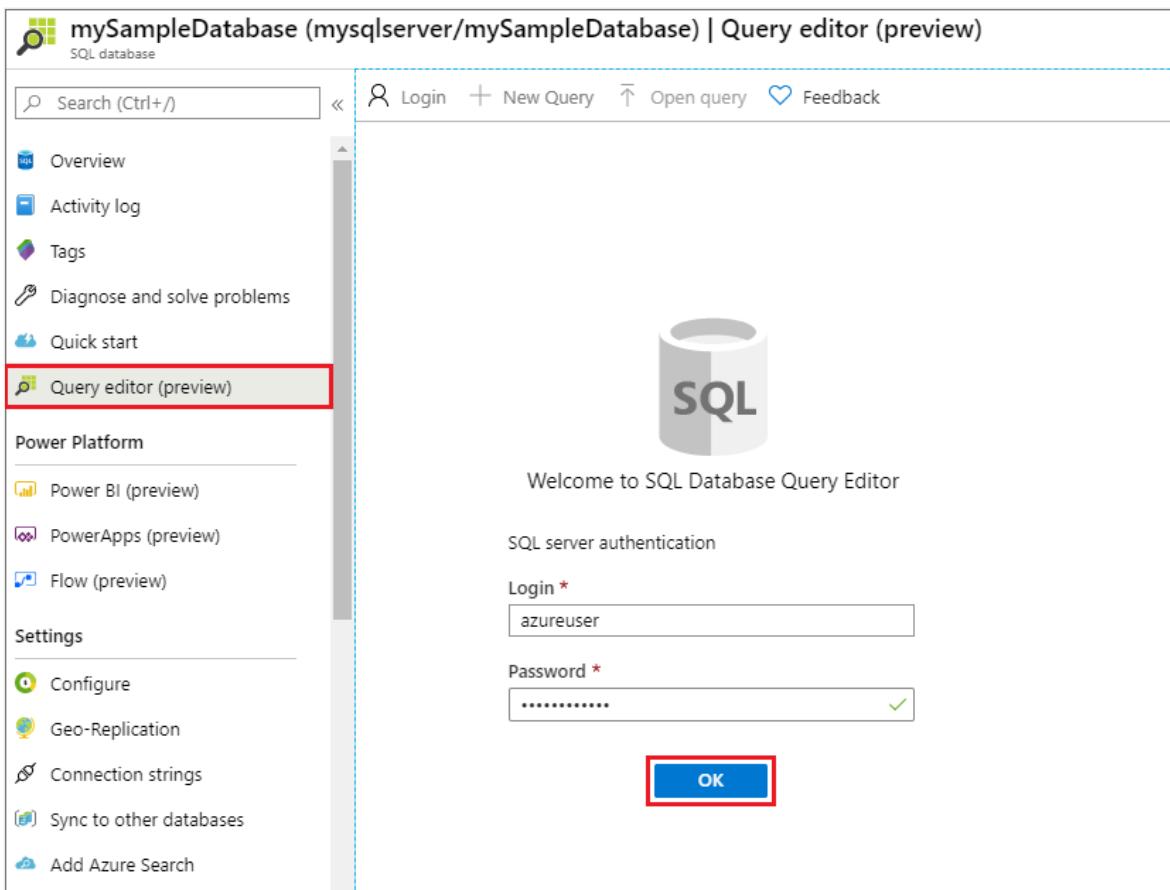
1. As in the previous article, press **F5** to start the function app project and Core Tools.
2. With Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the **HttpExample** function and choose **Execute Function Now...**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. After a response is returned, press `Ctrl + C` to stop Core Tools.

Verify that information has been written to the database

1. On the Azure portal, go back to your Azure SQL Database and select **Query editor**.



2. Connect to your database and expand the **Tables** node in object explorer on the left. Right-click on the `dbo.ToDo` table and select **Select Top 1000 Rows**.
3. Verify that the new information has been written to the database by the output binding.

Redeploy and verify the updated app

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Deploy to function app....`.
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [check the data written to your Azure SQL Database](#) to verify that the output binding again generates a new JSON document.

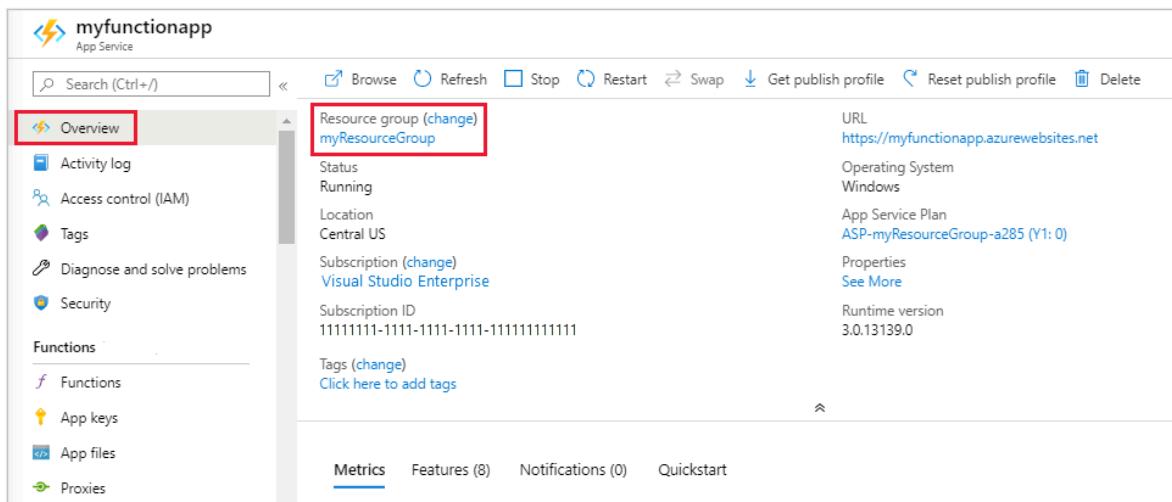
Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth.

They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to Azure SQL Database. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure SQL bindings and trigger for Azure Functions](#)

- Azure Functions triggers and bindings.
- Examples of complete Function projects in JavaScript.
- Azure Functions JavaScript developer guide

Connect Azure Functions to Azure Cosmos DB using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio Code to connect [Azure Cosmos DB](#) to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a JSON document stored in an Azure Cosmos DB container.

Before you begin, you must complete the [quickstart: Create a Python function in Azure using Visual Studio Code](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Configure your environment

Before you get started, make sure to install the [Azure Databases extension](#) for Visual Studio Code.

Create your Azure Cosmos DB account

Now, you create an Azure Cosmos DB account as a [serverless account type](#). This consumption-based mode makes Azure Cosmos DB a strong option for serverless workloads.

1. In Visual Studio Code, select **View > Command Palette...** then in the command palette search for `Azure Databases: Create Server...`
2. Provide the following information at the prompts:

[] [Expand table](#)

Prompt	Selection
Select an Azure Database Server	Choose Core (NoSQL) to create a document database that you can query by using a SQL syntax or a Query Copilot (Preview) converting natural language prompts to queries. Learn more about the Azure Cosmos DB .
Account name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
Select a capacity model	Select Serverless to create an account in serverless mode.
Select a resource group for new resources	Choose the resource group where you created your function app in the previous article .
Select a location for new resources	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to you or your users to get the fastest access to your data.

After your new account is provisioned, a message is displayed in notification area.

Create an Azure Cosmos DB database and container

1. Select the Azure icon in the Activity bar, expand **Resources > Azure Cosmos DB**, right-click (Ctrl+select on macOS) your account, and select **Create database...**
2. Provide the following information at the prompts:

[\[+\] Expand table](#)

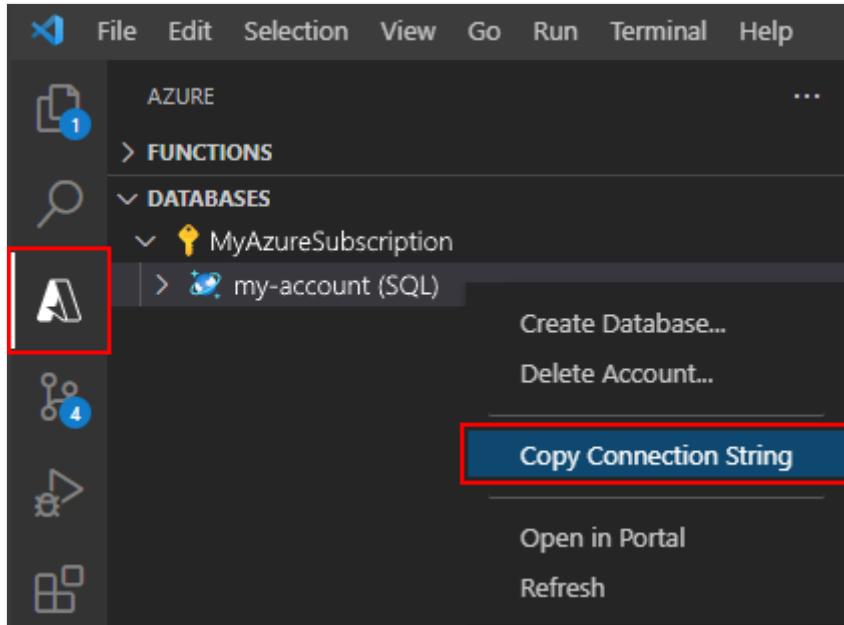
Prompt	Selection
Database name	Type <code>my-database</code> .
Enter and ID for your collection	Type <code>my-container</code> .
Enter the partition key for the collection	Type <code>/id</code> as the partition key .

3. Select **OK** to create the container and database.

Update your function app settings

In the [previous quickstart article](#), you created a function app in Azure. In this article, you update your app to write JSON documents to the Azure Cosmos DB container you've created. To connect to your Azure Cosmos DB account, you must add its connection string to your app settings. You then download the new setting to your local.settings.json file so you can connect to your Azure Cosmos DB account when running locally.

1. In Visual Studio Code, right-click (Ctrl+select on macOS) on your new Azure Cosmos DB account, and select **Copy Connection String**.



2. Press **F1** to open the command palette, then search for and run the command `Azure Functions: Add New Setting...`

3. Choose the function app you created in the previous article. Provide the following information at the prompts:

[+] [Expand table](#)

Prompt	Selection
Enter new app setting name	Type <code>CosmosDbConnectionSetting</code> .
Enter value for "CosmosDbConnectionSetting"	Paste the connection string of your Azure Cosmos DB account you copied. You can also configure Microsoft Entra identity as an alternative.

This creates an application setting named connection `CosmosDbConnectionSetting` in your function app in Azure. Now, you can download this setting to your local.settings.json file.

4. Press **F1** again to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`.
5. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

This downloads all of the setting from Azure to your local project, including the new connection string setting. Most of the downloaded settings aren't used when running locally.

Register binding extensions

Because you're using an Azure Cosmos DB output binding, you must have the corresponding bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles usage is enabled in the `host.json` file at the root of the project, which appears as follows:

```
JSON

{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[3.*, 4.0.0)"
  }
}
```

Now, you can add the Azure Cosmos DB output binding to your project.

Add an output binding

Binding attributes are defined directly in the `function_app.py` file. You use the `cosmos_db_output` decorator to add an [Azure Cosmos DB output binding](#):

```
Python

@app.cosmos_db_output(arg_name="outputDocument", database_name="my-
database",
                      container_name="my-container", connection="CosmosDbConnectionString")
```

In this code, `arg_name` identifies the binding parameter referenced in your code, `database_name` and `container_name` are the database and collection names that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Azure Cosmos DB account, which is in the `CosmosDbConnectionSetting` setting in the `local.settings.json` file.

Add code that uses the output binding

Update `HttpExample\function_app.py` to match the following code. Add the `outputDocument` parameter to the function definition and `outputDocument.set()` under the `if name:` statement:

```
Python

import azure.functions as func
import logging

app = func.FunctionApp()

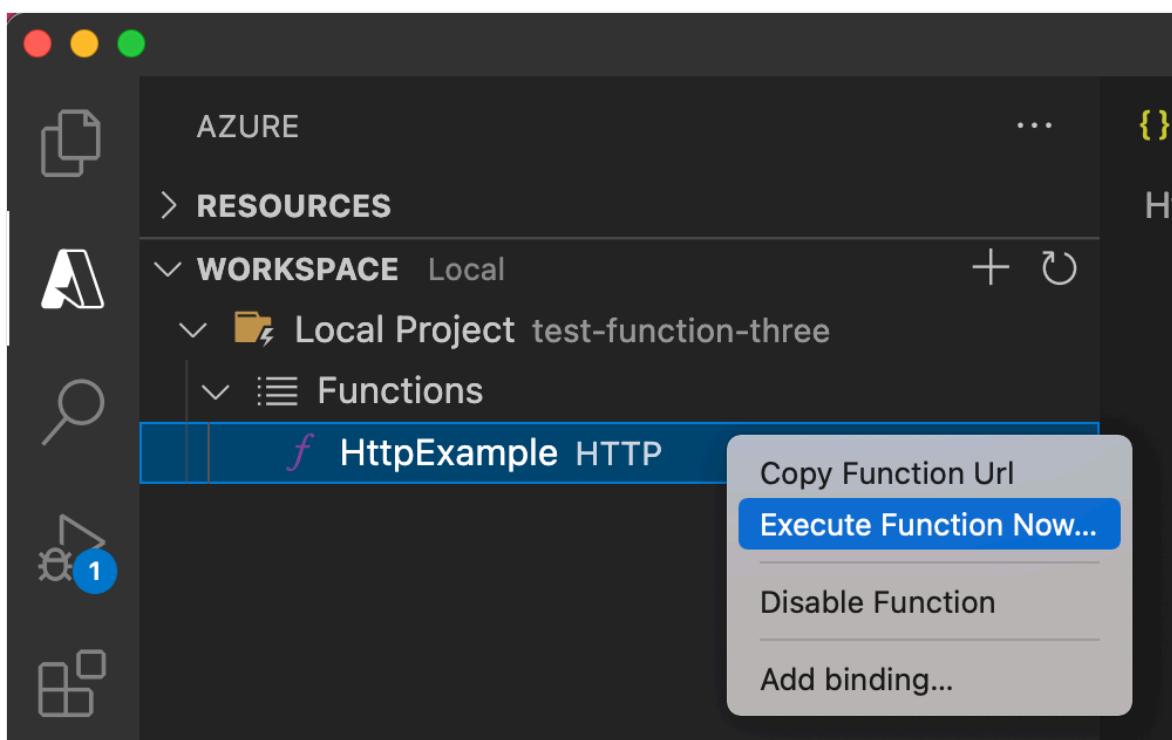
@app.function_name(name="HttpTrigger1")
@app.route(route="hello", auth_level=func.AuthLevel.ANONYMOUS)
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
@app.cosmos_db_output(arg_name="outputDocument", database_name="my-
database", container_name="my-container",
connection="CosmosDbConnectionSetting")
def test_function(req: func.HttpRequest, msg: func.Out[func.QueueMessage],
outputDocument: func.Out[func.Document]) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    logging.info('Python Cosmos DB trigger function processed a request.')
    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        outputDocument.set(func.Document.from_dict({"id": name}))
        msg.set(name)
        return func.HttpResponse(f"Hello {name}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the
request body",
            status_code=400
        )
```

The document `{"id": "name"}` is created in the database collection specified in the binding.

Run the function locally

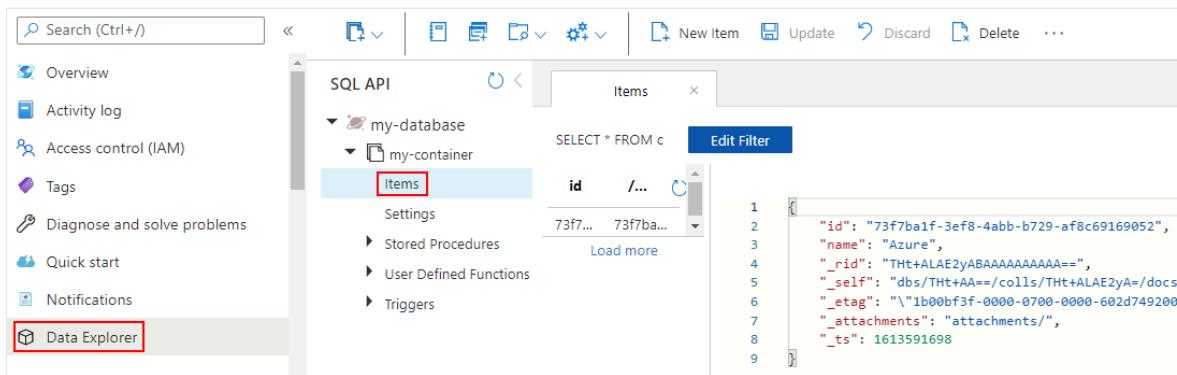
1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and choose **Execute Function Now....**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. After a response is returned, press `ctrl + c` to stop Core Tools.

Verify that a JSON document has been created

1. On the Azure portal, go back to your Azure Cosmos DB account and select **Data Explorer**.
2. Expand your database and container, and select **Items** to list the documents created in your container.
3. Verify that a new JSON document has been created by the output binding.



Redeploy and verify the updated app

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app...**.
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [check the documents created in your Azure Cosmos DB container](#) to verify that the output binding again generates a new JSON document.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure: Open in portal**.
2. Choose your function app and press **Enter**. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal's 'Overview' page for a resource group. The left sidebar has a red box around the 'Overview' link. The main area shows a card for 'myResourceGroup' with a red box around its title. The card contains details like 'Status: Running', 'Location: Central US', and 'Subscription: Visual Studio Enterprise'. At the bottom of the card, there's a link 'Click here to add tags'. To the right, there's a detailed view of properties: URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties (See More), and Runtime version (3.0.13139.0). Below the card, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write JSON documents to an Azure Cosmos DB container. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings.](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)

Connect Azure Functions to Azure SQL Database using Visual Studio Code

Article • 04/25/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio Code to connect [Azure SQL Database](#) to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a table in Azure SQL Database.

Before you begin, you must complete the [quickstart: Create a Python function in Azure using Visual Studio Code](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

More details on the settings for [Azure SQL bindings and trigger for Azure Functions](#) are available in the Azure Functions documentation.

Create your Azure SQL Database

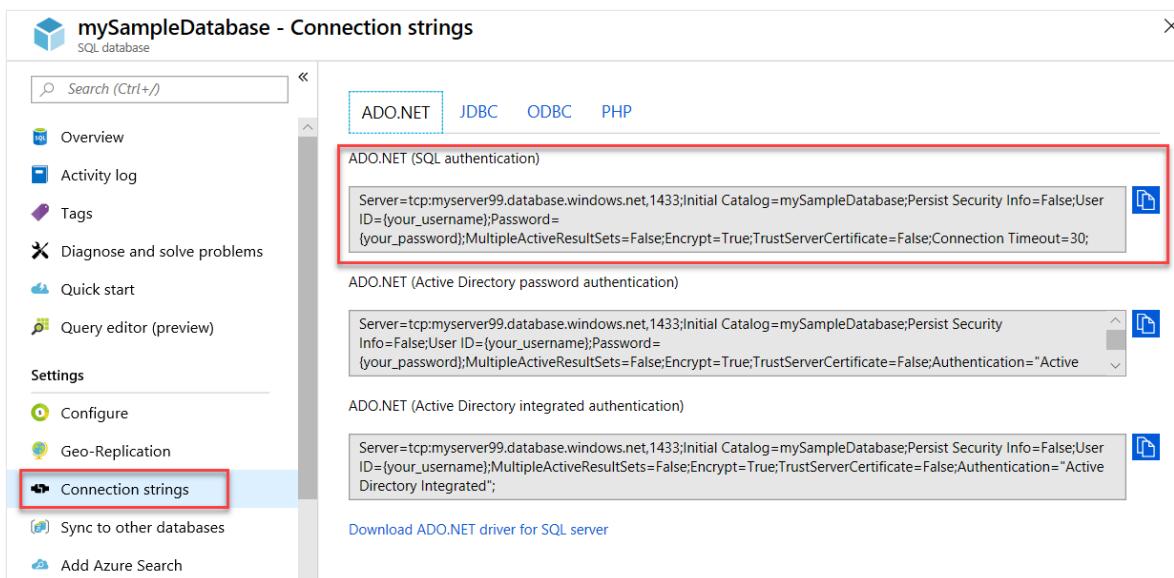
1. Follow the [Azure SQL Database create quickstart](#) to create a serverless Azure SQL Database. The database can be empty or created from the sample dataset AdventureWorksLT.
2. Provide the following information at the prompts:

[] Expand table

Prompt	Selection
Resource group	Choose the resource group where you created your function app in the previous article .
Database name	Enter <code>mySampleDatabase</code> .
Server name	Enter a unique name for your server. We can't provide an exact server name to use because server names must be globally

Prompt	Selection
	unique for all servers in Azure, not just unique within a subscription.
Authentication method	Select SQL Server authentication .
Server admin login	Enter <code>azureuser</code> .
Password	Enter a password that meets the complexity requirements.
Allow Azure services and resources to access this server	Select Yes .

3. Once the creation has completed, navigate to the database blade in the Azure portal, and, under **Settings**, select **Connection strings**. Copy the **ADO.NET** connection string for **SQL authentication**. Paste the connection string into a temporary document for later use.



4. Create a table to store the data from the HTTP request. In the Azure portal, navigate to the database blade and select **Query editor**. Enter the following query to create a table named `dbo.ToDo`:

```
SQL

CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

5. Verify that your Azure Function will be able to access the Azure SQL Database by checking the [server's firewall settings](#). Navigate to the **server blade** on the Azure portal, and under **Security**, select **Networking**. The exception for **Allow Azure services and resources to access this server** should be checked.

The screenshot shows the Azure portal interface for managing a SQL server named 'mydocsamplesqlserver'. The left sidebar lists various security features: Microsoft Defender for Cloud, Transparent data encryption, Identity, Auditing, Intelligent Performance, Automatic tuning, and Recommendations. The 'Networking' option is selected and highlighted with a red box. On the right, the 'Networking' blade is displayed with tabs for Public access, Private access, and Connectivity. Under the Public access tab, there is a section for Firewall rules with options to add client IPv4 addresses or firewall rules. Below this is an 'Exceptions' section, which is also highlighted with a red box, containing a checkbox labeled 'Allow Azure services and resources to access this server'.

Update your function app settings

In the [previous quickstart article](#), you created a function app in Azure. In this article, you update your app to write data to the Azure SQL Database you've just created. To connect to your Azure SQL Database, you must add its connection string to your app settings. You then download the new setting to your local.settings.json file so you can connect to your Azure SQL Database when running locally.

1. Edit the connection string in the temporary document you created earlier. Replace the value of `Password` with the password you used when creating the Azure SQL Database. Copy the updated connection string.
2. Press `Ctrl/Cmd+shift+P` to open the command palette, then search for and run the command `Azure Functions: Add New Setting...`.
3. Choose the function app you created in the previous article. Provide the following information at the prompts:

[] [Expand table](#)

Prompt	Selection
Enter new app setting name	Type <code>SqlConnectionString</code> .
Enter value for "SqlConnectionString"	Paste the connection string of your Azure SQL Database you just copied.

This creates an application setting named connection `SqlConnectionString` in your function app in Azure. Now, you can download this setting to your `local.settings.json` file.

4. Press `Ctrl/Cmd+shift+P` again to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
5. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

This downloads all of the setting from Azure to your local project, including the new connection string setting. Most of the downloaded settings aren't used when running locally.

Register binding extensions

Because you're using an Azure SQL output binding, you must have the corresponding bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles usage is enabled in the `host.json` file at the root of the project, which appears as follows:

```
JSON

{
  "version": "2.0",
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      }
    }
  },
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.0.0)"
  }
}
```

```
        },
        "concurrency": {
            "dynamicConcurrencyEnabled": true,
            "snapshotPersistenceEnabled": true
        }
    }
}
```

:::

Now, you can add the Azure SQL output binding to your project.

Add an output binding

In Functions, each type of binding requires a `direction`, `type`, and a unique `name` to be defined in the `function.json` file. The way you define these attributes depends on the language of your function app.

Binding attributes are defined directly in the `function_app.py` file. You use the `generic_output_binding` decorator to add an [Azure SQL output binding](#):

Python

```
@app.generic_output_binding(arg_name="ToDoItems", type="sql",
    CommandText="dbo.ToDo", ConnectionStringSetting="SqlConnectionString"
    data_type=DataType.STRING)
```

In this code, `arg_name` identifies the binding parameter referenced in your code, `type` denotes the output binding is a SQL output binding, `CommandText` is the table that the binding writes to, and `ConnectionStringSetting` is the name of an application setting that contains the Azure SQL connection string. The connection string is in the `SqlConnectionString` setting in the `local.settings.json` file.

Add code that uses the output binding

Update `HttpExample\function_app.py` to match the following code. Add the `ToDoItems` parameter to the function definition and `ToDoItems.set()` under the `if name: __main__` statement:

Python

```
import azure.functions as func
import logging
from azure.functions.decorators.core import DataType
import uuid
```

```

app = func.FunctionApp()

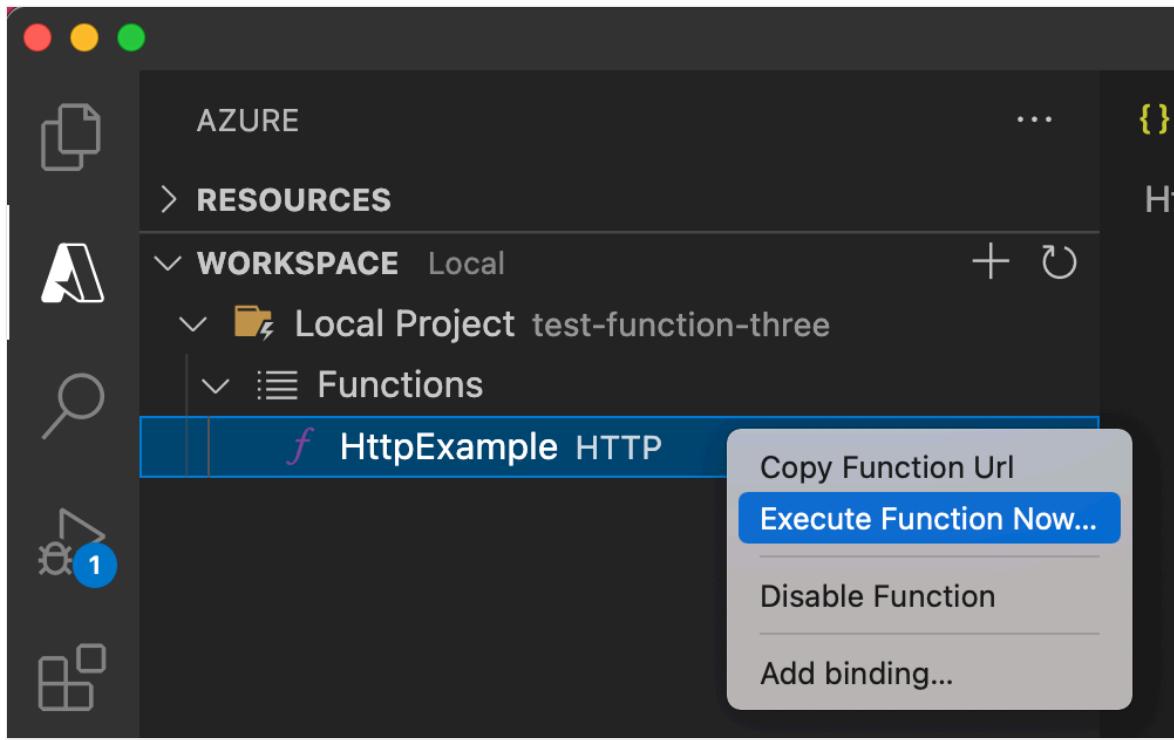
@app.function_name(name="HttpTrigger1")
@app.route(route="hello", auth_level=func.AuthLevel.ANONYMOUS)
@app.generic_output_binding(arg_name="ToDoItems", type="sql",
CommandText="dbo.ToDo",
ConnectionStringSetting="SqlConnectionString", data_type=DataType.STRING)
def test_function(req: func.HttpRequest, ToDoItems: func.Out[func.SqlRow]) -
> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    name = req.get_json().get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
    else:
        name = req_body.get('name')

    if name:
        ToDoItems.set(func.SqlRow({"Id": str(uuid.uuid4()), "title": name,
"completed": False, "url": ""}))
        return func.HttpResponse(f"Hello {name}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the
request body",
            status_code=400
        )

```

Run the function locally

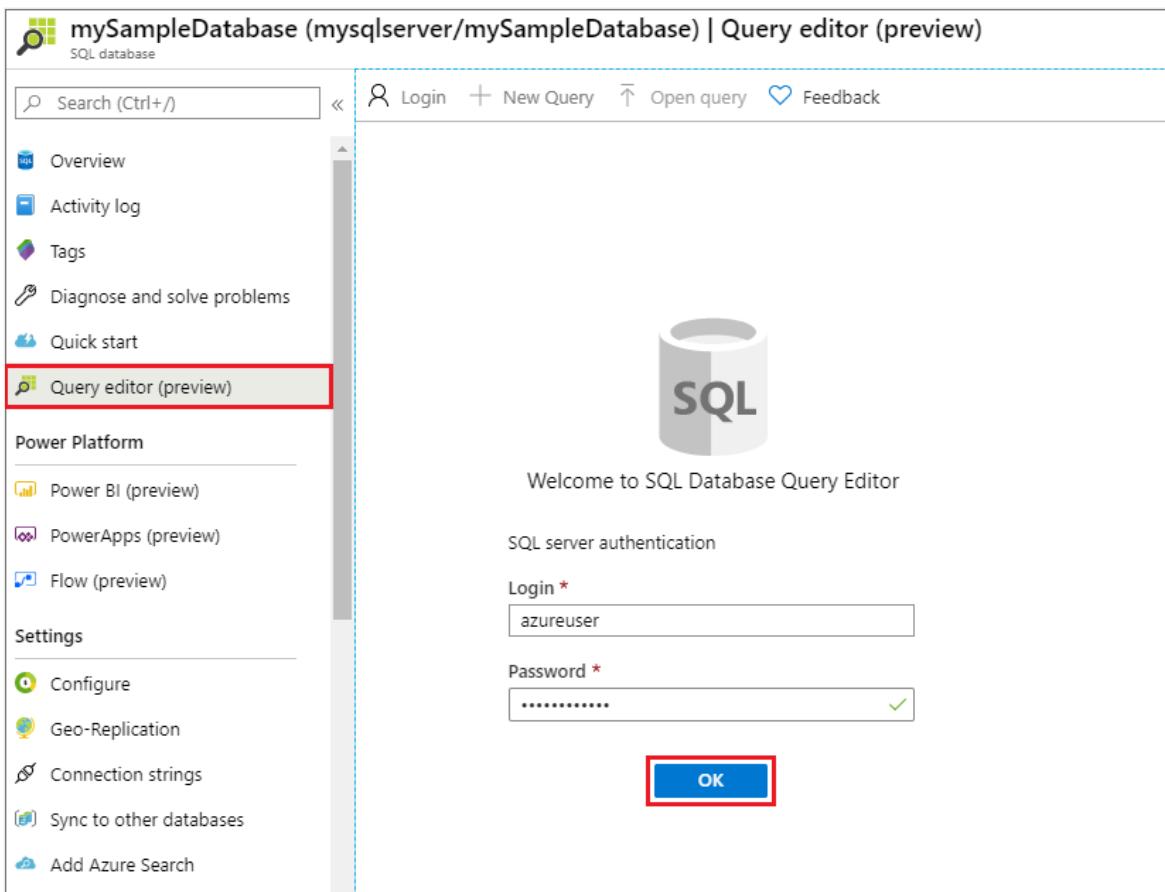
1. As in the previous article, press **F5** to start the function app project and Core Tools.
2. With Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the **HttpExample** function and choose **Execute Function Now...**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. After a response is returned, press `Ctrl + C` to stop Core Tools.

Verify that information has been written to the database

1. On the Azure portal, go back to your Azure SQL Database and select **Query editor**.



2. Connect to your database and expand the **Tables** node in object explorer on the left. Right-click on the `dbo.ToDo` table and select **Select Top 1000 Rows**.
3. Verify that the new information has been written to the database by the output binding.

Redeploy and verify the updated app

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Deploy to function app....`.
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [check the data written to your Azure SQL Database](#) to verify that the output binding again generates a new JSON document.

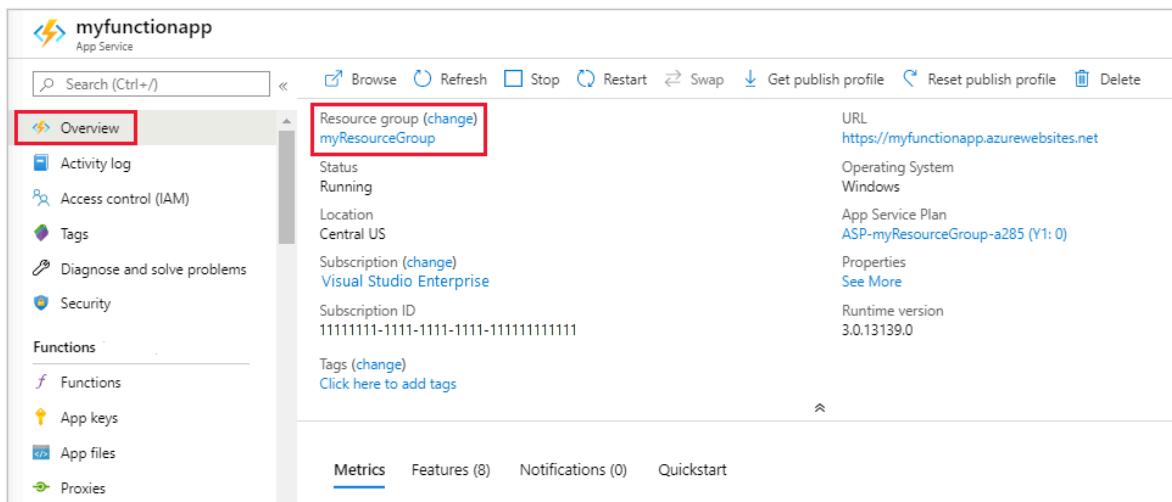
Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth.

They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to Azure SQL Database. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure SQL bindings and trigger for Azure Functions](#)

- Azure Functions triggers and bindings.
- Examples of complete Function projects in Python.
- Azure Functions Python developer guide

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- Install [.NET Core CLI tools](#).
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.

3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[] [Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <i>mySampleOpenAI</i> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page ↗

Create Azure OpenAI

...

1 Basics

2 Network

3 Tags

4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

5. Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

1. In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
2. In the local.settings.json file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in *local.settings.json* is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
3. In the local.settings.json file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
4. Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Register binding extensions

Because you're using an Azure OpenAI output binding, you must have the corresponding bindings extension installed before you run the project.

Except for HTTP and timer triggers, bindings are implemented as extension packages. To add the Azure OpenAI extension package to your project, run this [dotnet add package](#) command in the **Terminal** window:

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.OpenAI --  
prerelease
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. In your existing `HttpExample` class file, add this `using` statement:

C#

```
using  
Microsoft.Azure.Functions.Worker.Extensions.OpenAI.TextCompletion;
```

2. In the same file, add this code that defines a new HTTP trigger endpoint named `whois`:

C#

```
[Function(nameof(WhoIs))]  
public IActionResult WhoIs([HttpTrigger(AuthorizationLevel.Function,  
Route = "whois/{name}")] HttpRequest req,  
[TextCompletionInput("Who is {name}?", Model =  
"%CHAT_MODEL_DEPLOYMENT_NAME%")] TextCompletionResponse response)  
{  
    if(!String.IsNullOrEmpty(response.Content))  
    {  
        return new OkObjectResult(response.Content);  
    }  
    else  
    {  
        return new NotFoundObjectResult("Something went wrong.");  
    }  
}
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.
2. Press `F5` to start the function app project and Core Tools in debug mode.
3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

```
http://localhost:7071/api/whois/<NAME>
```

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', and 'Security'. Below that is another 'Functions' section with 'Functions', 'App keys', 'App files', and 'Proxies'. The main pane shows the 'Overview' tab for a resource group named 'myResourceGroup'. The resource group details include: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (Click here to add tags). At the bottom of the main pane are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples ↗](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.
3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[+] [Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <i>mySampleOpenAI</i> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page ↗

Create Azure OpenAI

...

1 Basics

2 Network

3 Tags

4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

- Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

- In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
- In the `local.settings.json` file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in `local.settings.json` is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
- In the `local.settings.json` file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
- Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Update the extension bundle

To access the preview Azure OpenAI bindings, you must use a preview version of the extension bundle that contains this extension.

Replace the `extensionBundle` setting in your current `host.json` file with this JSON:

JSON

```
"extensionBundle": {  
    "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
    "version": "[4.*, 5.0.0)"  
}
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. Update the `pom.xml` project file to add this reference to the `properties` collection:

XML

```
<azure-functions-java-library-openai>0.3.0-preview</azure-functions-  
java-library-openai>
```

2. In the same file, add this dependency to the `dependencies` collection:

XML

```
<dependency>  
    <groupId>com.microsoft.azure.functions</groupId>  
    <artifactId>azure-functions-java-library-openai</artifactId>  
    <version>${azure-functions-java-library-openai}</version>  
</dependency>
```

3. In the existing `Function.java` project file, add these `import` statements:

Java

```
import  
com.microsoft.azure.functions.openai.annotation.textcompletion.TextComp  
letion;  
import  
com.microsoft.azure.functions.openai.annotation.textcompletion.TextComp  
letionResponse;
```

4. In the same file, add this code that defines a new HTTP trigger endpoint named

`whois`:

```
Java

@FunctionName("WhoIs")
public HttpResponseMessage whoIs(
    @HttpTrigger(
        name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.ANONYMOUS,
        route = "whois/{name}")
    HttpRequestMessage<Optional<String>> request,
    @BindingName("name") String name,
    @TextCompletion(prompt = "Who is {name}?", model =
"%CHAT_MODEL_DEPLOYMENT_NAME%", name = "response")
    TextCompletionResponse response,
    final ExecutionContext context) {
    return request.createResponseBuilder(HttpStatus.OK)
        .header("Content-Type", "application/json")
        .body(response.getContent())
        .build();
}
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.

2. Press `F5` to start the function app project and Core Tools in debug mode.

3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

`http://localhost:7071/api/whois/<NAME>`

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

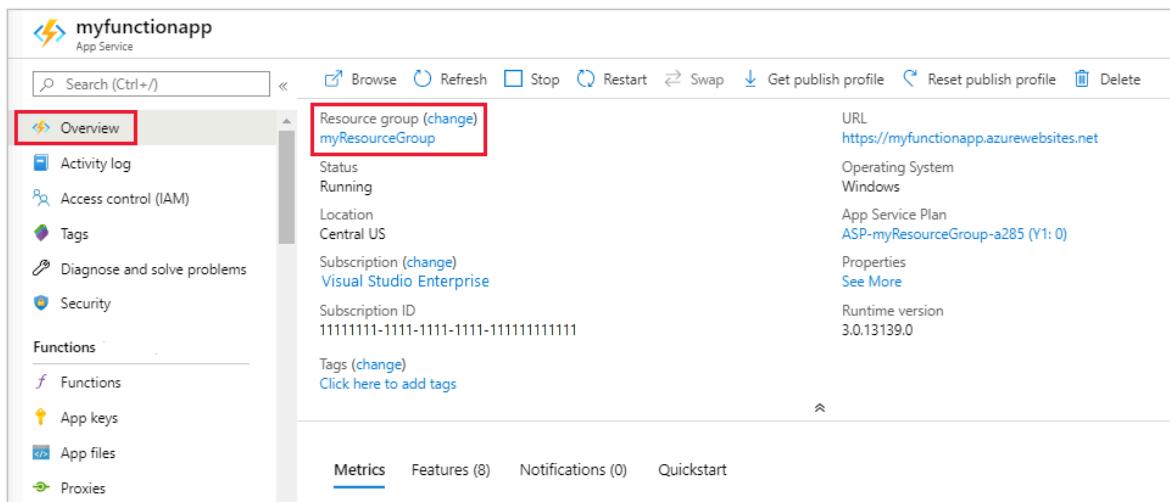
8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth.

They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.
3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[+] [Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <i>mySampleOpenAI</i> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page ↗

Create Azure OpenAI

...

Basics

Network

Tags

Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

- Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

- In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
- In the `local.settings.json` file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in `local.settings.json` is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
- In the `local.settings.json` file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
- Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Update the extension bundle

To access the preview Azure OpenAI bindings, you must use a preview version of the extension bundle that contains this extension.

Replace the `extensionBundle` setting in your current `host.json` file with this JSON:

JSON

```
"extensionBundle": {  
  "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
  "version": "[4.*, 5.0.0)"  
}
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. In Visual Studio Code, Press F1 and in the command palette type `Azure Functions: Create Function...`, select **HTTP trigger**, type the function name `whois`, and press Enter.
2. In the new `whois.js` code file, replace the contents of the file with this code:

```
JavaScript  
  
const { app, input } = require("@azure/functions");  
  
// This OpenAI completion input requires a {name} binding value.  
const openAICompletionInput = input.generic({  
  prompt: 'Who is {name}?',  
  maxTokens: '100',  
  type: 'textCompletion',  
  model: '%CHAT_MODEL_DEPLOYMENT_NAME%'  
})  
  
app.http('whois', {  
  methods: ['GET'],  
  route: 'whois/{name}',  
  authLevel: 'function',  
  extraInputs: [openAICompletionInput],  
  handler: async (_request, context) => {  
    var response = context.extraInputs.get(openAICompletionInput)  
    return { body: response.content.trim() }  
  }  
});
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.
2. Press `F5` to start the function app project and Core Tools in debug mode.
3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

```
http://localhost:7071/api/whois/<NAME>
```

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal's Overview page for a resource group. The left sidebar has a red box around the 'Overview' tab. The main content area has a red box around the 'myResourceGroup' resource card. Resource details shown include status, location, subscription, tags, and various service properties.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples ↗](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.
3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[+] [Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <i>mySampleOpenAI</i> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page ↗

Create Azure OpenAI

...

1 Basics

2 Network

3 Tags

4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

- Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

- In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
- In the `local.settings.json` file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in `local.settings.json` is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
- In the `local.settings.json` file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
- Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Update the extension bundle

To access the preview Azure OpenAI bindings, you must use a preview version of the extension bundle that contains this extension.

Replace the `extensionBundle` setting in your current `host.json` file with this JSON:

JSON

```
"extensionBundle": {  
  "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
  "version": "[4.*, 5.0.0)"  
}
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. In Visual Studio Code, Press F1 and in the command palette type `Azure Functions: Create Function...`, select **HTTP trigger**, type the function name `whois`, select **Anonymous**, and press Enter.
2. Open the new `whois/function.json` code file and replace its contents with this code, which adds a definition for the `TextCompletionResponse` input binding:

```
JSON  
  
{  
  "bindings": [  
    {  
      "authLevel": "function",  
      "type": "httpTrigger",  
      "direction": "in",  
      "name": "Request",  
      "route": "whois/{name}",  
      "methods": [  
        "get"  
      ]  
    },  
    {  
      "type": "http",  
      "direction": "out",  
      "name": "Response"  
    },  
    {  
      "type": "textCompletion",  
      "direction": "in",  
      "name": "TextCompletionResponse",  
      "prompt": "Who is {name}?",  
      "maxTokens": "100",  
      "model": "text-davinci-003"  
    }  
  ]  
}
```

```
        "model": "%CHAT_MODEL_DEPLOYMENT_NAME%"  
    }  
}  
}
```

3. Replace the content of the `whois/run.ps1` code file with this code, which returns the input binding response:

PowerShell

```
using namespace System.Net  
  
param($Request, $TriggerMetadata, $TextCompletionResponse)  
  
Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{  
    StatusCode = [HttpStatusCode]::OK  
    Body      = $TextCompletionResponse.Content  
})
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.
2. Press `F5` to start the function app project and Core Tools in debug mode.
3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

`http://localhost:7071/api/whois/<NAME>`

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

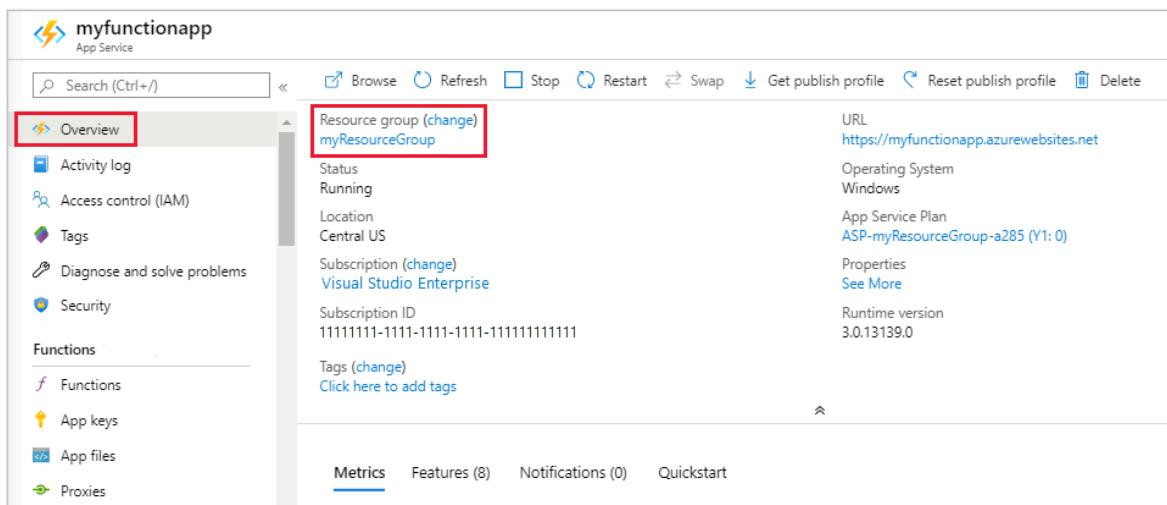
8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by

deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.
3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[+] [Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <i>mySampleOpenAI</i> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page ↗

Create Azure OpenAI

...

Basics

Network

Tags

Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

- Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

- In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
- In the `local.settings.json` file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in `local.settings.json` is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
- In the `local.settings.json` file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
- Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Update the extension bundle

To access the preview Azure OpenAI bindings, you must use a preview version of the extension bundle that contains this extension.

Replace the `extensionBundle` setting in your current `host.json` file with this JSON:

JSON

```
"extensionBundle": {  
    "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
    "version": "[4.*, 5.0.0)"  
}
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. In the existing `function_app.py` project file, add this `import` statement:

Python

```
import json
```

2. In the same file, add this code that defines a new HTTP trigger endpoint named

`whois`:

Python

```
@app.route(route="whois/{name}", methods=["GET"])  
@app.text_completion_input(arg_name="response", prompt="Who is  
{name}?", max_tokens="100", model = "%CHAT_MODEL_DEPLOYMENT_NAME%")  
def whois(req: func.HttpRequest, response: str) -> func.HttpResponse:  
    response_json = json.loads(response)  
    return func.HttpResponse(response_json["content"], status_code=200)  
  
@app.route(route="genericcompletion", methods=["POST"])  
@app.text_completion_input(arg_name="response", prompt="{Prompt}",  
model = "%CHAT_MODEL_DEPLOYMENT_NAME%")  
def genericcompletion(req: func.HttpRequest, response: str) ->  
func.HttpResponse:  
    response_json = json.loads(response)  
    return func.HttpResponse(response_json["content"], status_code=200)
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.
2. Press `F5` to start the function app project and Core Tools in debug mode.
3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

```
http://localhost:7071/api/whois/<NAME>
```

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', and 'Security'. Below that is another 'Functions' section with 'Functions', 'App keys', 'App files', and 'Proxies'. The main pane shows the 'Overview' tab for a resource group named 'myResourceGroup'. The resource group details include:

- Status: Running
- Location: Central US
- Subscription: Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags: Click here to add tags
- URL: <https://myfunctionapp.azurewebsites.net>
- Operating System: Windows
- App Service Plan: ASP-myResourceGroup-a285 (Y1: 0)
- Properties: See More
- Runtime version: 3.0.13139.0

At the bottom of the main pane are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples ↗](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.
3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[+] [Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <i>mySampleOpenAI</i> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page ↗

Create Azure OpenAI

...

1 Basics

2 Network

3 Tags

4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

- Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

- In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
- In the local.settings.json file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in `local.settings.json` is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
- In the local.settings.json file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
- Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Update the extension bundle

To access the preview Azure OpenAI bindings, you must use a preview version of the extension bundle that contains this extension.

Replace the `extensionBundle` setting in your current `host.json` file with this JSON:

JSON

```
"extensionBundle": {  
  "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
  "version": "[4.*, 5.0.0)"  
}
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. In Visual Studio Code, Press F1 and in the command palette type `Azure Functions: Create Function...`, select **HTTP trigger**, type the function name `whois`, and press Enter.
2. In the new `whois.ts` code file, replace the contents of the file with this code:

TypeScript

```
import { app, input } from "@azure/functions";  
  
// This OpenAI completion input requires a {name} binding value.  
const openAICompletionInput = input.generic({  
  prompt: 'Who is {name}?',  
  maxTokens: '100',  
  type: 'textCompletion',  
  model: '%CHAT_MODEL_DEPLOYMENT_NAME%'  
})  
  
app.http('whois', {  
  methods: ['GET'],  
  route: 'whois/{name}',  
  authLevel: 'function',  
  extraInputs: [openAICompletionInput],  
  handler: async (_request, context) => {  
    var response: any =  
    context.extraInputs.get(openAICompletionInput)  
    return { body: response.content.trim() }  
  }  
});
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.
2. Press `F5` to start the function app project and Core Tools in debug mode.
3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

```
http://localhost:7071/api/whois/<NAME>
```

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', and 'Security'. Below that is another 'Functions' section with 'Functions', 'App keys', 'App files', and 'Proxies'. The main pane shows the 'Overview' tab for a resource group named 'myResourceGroup'. The details include:

- Status: Running
- Location: Central US
- Subscription: Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags: Click here to add tags
- URL: <https://myfunctionapp.azurewebsites.net>
- Operating System: Windows
- App Service Plan: ASP-myResourceGroup-a285 (Y1: 0)
- Properties: See More
- Runtime version: 3.0.13139.0

At the bottom, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples ↗](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Create a function to integrate with Azure Logic Apps

Article • 08/09/2024

Azure Functions integrates with Azure Logic Apps in the Logic Apps Designer. This integration allows you use the computing power of Functions in orchestrations with other Azure and third-party services.

This tutorial shows you how to create a workflow to analyze X activity. As tweets are evaluated, the workflow sends notifications when positive sentiments are detected.

In this tutorial, you learn to:

- ✓ Create an Azure AI services API Resource.
- ✓ Create a function that categorizes tweet sentiment.
- ✓ Create a logic app that connects to X.
- ✓ Add sentiment detection to the logic app.
- ✓ Connect the logic app to the function.
- ✓ Send an email based on the response from the function.

Prerequisites

- An active [X](#) account.
- An [Outlook.com](#) account (for sending notifications).

ⓘ Note

If you want to use the Gmail connector, only G-Suite business accounts can use this connector without restrictions in logic apps. If you have a Gmail consumer account, you can use the Gmail connector with only specific Google-approved apps and services, or you can [create a Google client app to use for authentication in your Gmail connector](#).

For more information, see [Data security and privacy policies for Google connectors in Azure Logic Apps](#).

Create Text Analytics resource

The Azure AI services APIs are available in Azure as individual resources. Use the Text Analytics API to detect the sentiment of posted tweets.

1. Sign in to the [Azure portal](#).
2. Select **Create a resource** in the upper left-hand corner of the Azure portal.
3. Under *Categories*, select **AI + Machine Learning**
4. Under *Text Analytics*, select **Create**.
5. Enter the following values in the *Create Text Analytics* screen.

[\[+\] Expand table](#)

Setting	Value	Remarks
Subscription	Your Azure subscription name	
Resource group	Create a new resource group named tweet-sentiment-tutorial	Later, you delete this resource group to remove all the resources created during this tutorial.
Region	Select the region closest to you	
Name	TweetSentimentApp	
Pricing tier	Select Free F0	

6. Select **Review + create**.
7. Select **Create**.
8. Once the deployment is complete, select **Go to Resource**.

Get Text Analytics settings

With the Text Analytics resource created, you'll copy a few settings and set them aside for later use.

1. Select **Keys and Endpoint**.
2. Copy **Key 1** by clicking on the icon at the end of the input box.
3. Paste the value into a text editor.
4. Copy the **Endpoint** by clicking on the icon at the end of the input box.

5. Paste the value into a text editor.

Create the function app

1. From the top search box, search for and select **Function app**.

2. Select **Create**.

3. Enter the following values.

[] [Expand table](#)

Setting	Suggested Value	Remarks
Subscription	Your Azure subscription name	
Resource group	<code>tweet-sentiment-tutorial</code>	Use the same resource group name throughout this tutorial.
Function App name	<code>TweetSentimentAPI + a unique suffix</code>	Function application names are globally unique. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Publish	Code	
Runtime stack	.NET	The function code provided for you is in C#.
Version	Select the latest version number	
Region	Select the region closest to you	

4. Select **Review + create**.

5. Select **Create**.

6. Once the deployment is complete, select **Go to Resource**.

Create an HTTP-triggered function

1. From the left menu of the *Functions* window, select **Functions**.

2. Select **Add** from the top menu and enter the following values.

Setting	Value	Remarks
Development environment	Develop in portal	
Template	HTTP Trigger	
New Function	TweetSentimentFunction	This is the name of your function.
Authorization level	Function	

3. Select the **Add** button.

4. Select the **Code + Test** button.

5. Paste the following code in the code editor window.

```
C#  
  
#r "Newtonsoft.Json"  
  
using System;  
using System.Net;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Extensions.Logging;  
using Microsoft.Extensions.Primitives;  
using Newtonsoft.Json;  
  
public static async Task<IActionResult> Run(HttpContext req, ILogger log)  
{  
  
    string requestBody = String.Empty;  
    using (StreamReader streamReader = new StreamReader(req.Body))  
    {  
        requestBody = await streamReader.ReadToEndAsync();  
    }  
  
    dynamic score = JsonConvert.DeserializeObject(requestBody);  
    string value = "Positive";  
  
    if(score < .3)  
    {  
        value = "Negative";  
    }  
    else if (score < .6)  
    {  
        value = "Neutral";  
    }  
  
    return requestBody != null  
        ? (ActionResult)new OkObjectResult(value)
```

```
: new BadRequestObjectResult("Pass a sentiment score in the  
request body.");  
}
```

A sentiment score is passed into the function, which returns a category name for the value.

6. Select the **Save** button on the toolbar to save your changes.

ⓘ Note

To test the function, select **Test/Run** from the top menu. On the *Input* tab, enter a value of **0.9** in the *Body* input box, and then select **Run**. Verify that a value of *Positive* is returned in the *HTTP response content* box in the *Output* section.

Next, create a logic app that integrates with Azure Functions, X, and the Azure AI services API.

Create a logic app

1. From the top search box, search for and select **Logic Apps**.
2. Select **Add**.
3. Select **Consumption** and enter the following values.

[] Expand table

Setting	Suggested Value
Subscription	Your Azure subscription name
Resource group	tweet-sentiment-tutorial
Logic app name	TweetSentimentApp
Region	Select the region closest to you, preferably the same region you selected in previous steps.

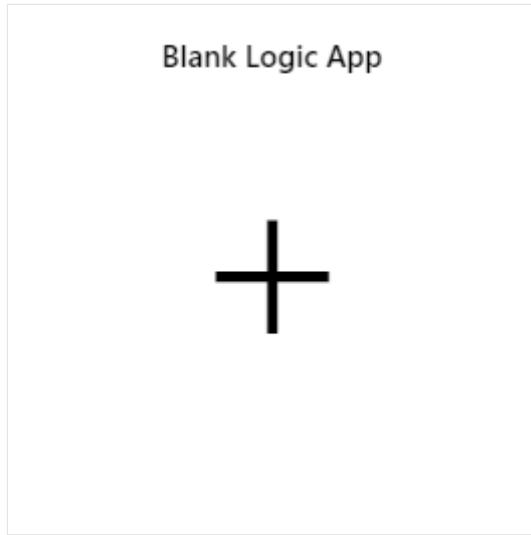
Accept default values for all other settings.

4. Select **Review + create**.

5. Select **Create**.

6. Once the deployment is complete, select **Go to Resource**.

7. Select the **Blank Logic App** button.



8. Select the **Save** button on the toolbar to save your progress.

You can now use the Logic Apps Designer to add services and triggers to your application.

Connect to X

Create a connection to X so your app can poll for new tweets.

1. Search for **X** in the top search box.

2. Select the **X** icon.

3. Select the **When a new tweet is posted** trigger.

4. Enter the following values to set up the connection.

[] Expand table

Setting	Value
Connection name	MyXConnection
Authentication Type	Use default shared application

5. Select **Sign in**.

6. Follow the prompts in the pop-up window to complete signing in to X.

7. Next, enter the following values in the *When a new tweet is posted* box.

[+] Expand table

Setting	Value
Search text	#my-x-tutorial
How often do you want to check for items?	1 in the textbox, and Hour in the dropdown. You may enter different values but be sure to review the current limitations of the X connector.

8. Select the **Save** button on the toolbar to save your progress.

Next, connect to text analytics to detect the sentiment of collected tweets.

Add Text Analytics sentiment detection

1. Select **New step**.
2. Search for **Text Analytics** in the search box.
3. Select the **Text Analytics** icon.
4. Select **Detect Sentiment** and enter the following values.

[+] Expand table

Setting	Value
Connection name	TextAnalyticsConnection
Account Key	Paste in the Text Analytics account key you set aside earlier.
Site URL	Paste in the Text Analytics endpoint you set aside earlier.

5. Select **Create**.
6. Click inside the *Add new parameter* box, and check the box next to **documents** that appears in the pop-up.
7. Click inside the *documents Id - 1* textbox to open the dynamic content pop-up.
8. In the *dynamic content* search box, search for **id**, and click on **Tweet id**.
9. Click inside the *documents Text - 1* textbox to open the dynamic content pop-up.

10. In the *dynamic content* search box, search for **text**, and click on **Tweet text**.
11. In **Choose an action**, type **Text Analytics**, and then click the **Detect sentiment** action.
12. Select the **Save** button on the toolbar to save your progress.

The *Detect Sentiment* box should look like the following screenshot.

The screenshot shows the 'Parameters' tab of the 'Sentiment (V4)' logic app step. It includes fields for 'Resource Subdomain Or Region' (set to 'westus'), 'Documents' (containing 'Id - 1' with 'Tweet id' and 'Text - 1' with 'Tweet text' inputs), 'Language - 1' (set to 'en'), and an 'Advanced parameters' section with 'Showing 1 of 5' items. A note at the bottom indicates a connection to 'Detect Sentiment'.

Resource Subdomain Or Region *

westus

Documents *

Id - 1 *

Tweet id

Text - 1 *

Tweet text

Language - 1

en

+ Add new item

Advanced parameters

Showing 1 of 5

Show all

Clear all

StringIndexType

TextElement_v8

Connected to Detect Sentiment. [Change connection](#)

Connect sentiment output to function endpoint

1. Select **New step**.

2. Search for **Azure Functions** in the search box.
3. Select the **Azure Functions** icon.
4. Search for your function name in the search box. If you followed the guidance above, your function name begins with **TweetSentimentAPI**.
5. Select the function icon.
6. Select the **TweetSentimentFunction** item.
7. Click inside the *Request Body* box, and select the **Detect Sentiment score** item from the pop-up window.
8. Select the **Save** button on the toolbar to save your progress.

Add conditional step

1. Select the **Add an action** button.
2. Click inside the *Control* box, and search for and select **Control** in the pop-up window.
3. Select **Condition**.
4. Click inside the *Choose a value* box, and select the **TweetSentimentFunction Body** item from the pop-up window.
5. Enter **Positive** in the *Choose a value* box.
6. Select the **Save** button on the toolbar to save your progress.

Add email notifications

1. Under the *True* box, select the **Add an action** button.
2. Search for and select **Office 365 Outlook** in the text box.
3. Search for **send** and select **Send an email** in the text box.
4. Select the **Sign in** button.
5. Follow the prompts in the pop-up window to complete signing in to Office 365 Outlook.
6. Enter your email address in the *To* box.

7. Click inside the *Subject* box and click on the **Body** item under *TweetSentimentFunction*. If the *Body* item isn't shown in the list, click the **See more** link to expand the options list.
8. After the *Body* item in the *Subject*, enter the text **Tweet from:**
9. After the *Tweet from:* text, click on the box again and select **User name** from the *When a new tweet is posted* options list.
10. Click inside the *Body* box and select **Tweet text** under the *When a new tweet is posted* options list. If the *Tweet text* item isn't shown in the list, click the **See more** link to expand the options list.
11. Select the **Save** button on the toolbar to save your progress.

The email box should now look like this screenshot.

To *

someone@contoso.com

Subject *

Body *

Normal Arial 15px

B I U A ~~A~~ ∞

X Tweet text x

Advanced parameters

Showing 1 of 7

Importance

Normal

Connected Change connection

Run the workflow

1. From your X account, tweet the following text: I'm enjoying #my-x-tutorial.
2. Return to the Logic Apps Designer and select the **Run** button.
3. Check your email for a message from the workflow.

Clean up resources

To clean up all the Azure services and accounts created during this tutorial, delete the resource group.

1. Search for **Resource groups** in the top search box.

2. Select the **tweet-sentiment-tutorial**.
3. Select **Delete resource group**
4. Enter **tweet-sentiment-tutorial** in the text box.
5. Select the **Delete** button.

Optionally, you may want to return to your X account and delete any test tweets from your feed.

Next steps

[Create a serverless API using Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create a function in Azure with Python using Visual Studio Code

Article • 09/10/2024

In this article, you use Visual Studio Code to create a Python function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

This article uses the Python v2 programming model for Azure Functions, which provides a decorator-based approach for creating functions. To learn more about the Python v2 programming model, see the [Developer Reference Guide](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [CLI-based version](#) of this article.

This video shows you how to create a Python function in Azure using Visual Studio Code.

[https://learn-video.azurefd.net/vod/player?id=a1e10f96-2940-489c-bc53-da2b915c8fc2&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-python ↗](https://learn-video.azurefd.net/vod/player?id=a1e10f96-2940-489c-bc53-da2b915c8fc2&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-python)

The steps in the video are also described in the following sections.

Configure your environment

Before you begin, make sure that you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- A Python version that is [supported by Azure Functions](#). For more information, see [How to install Python ↗](#).
- [Visual Studio Code ↗](#) on one of the [supported platforms ↗](#).
- The [Python extension ↗](#) for Visual Studio Code.
- The [Azure Functions extension ↗](#) for Visual Studio Code, version 1.8.1 or later.
- The [Azurite V3 extension ↗](#) local storage emulator. While you can also use an actual Azure storage account, this article assumes you're using the Azurite emulator.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in Python. Later in this article, you publish your function code to Azure.

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create New Project...`.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

[] [Expand table](#)

Prompt	Selection
Select a language	Choose <code>Python (Programming Model V2)</code> .
Select a Python interpreter to create a virtual environment	Choose your preferred Python interpreter. If an option isn't shown, type in the full path to your Python binary.
Select a template for your project's first function	Choose <code>HTTP trigger</code> .
Name of the function you want to create	Enter <code>HttpExample</code> .

Prompt	Selection
Authorization level	Choose <code>ANONYMOUS</code> , which lets anyone call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Choose <code>Open in current window</code> .

- Visual Studio Code uses the provided information and generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. The generated `function_app.py` project file contains your functions.
- In the `local.settings.json` file, update the `AzureWebJobsStorage` setting as in the following example:

JSON

```
"AzureWebJobsStorage": "UseDevelopmentStorage=true",
```

This tells the local Functions host to use the storage emulator for the storage connection required by the Python v2 model. When you publish your project to Azure, this setting uses the default storage account instead. If you're using an Azure Storage account during local development, set your storage account connection string here.

Start the emulator

- In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azurite: Start`.
- Check the bottom bar and verify that Azurite emulation services are running. If so, you can now run your function locally.

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

- To start the function locally, press `F5` or the Run and Debug icon in the left-hand side Activity bar. The Terminal panel displays the Output from Core Tools. Your app starts in the Terminal panel. You can see the URL endpoint of your HTTP-triggered function running locally.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The output in the terminal window is as follows:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG host start - Task ✓ + × ↻ ↺ ↺ ×

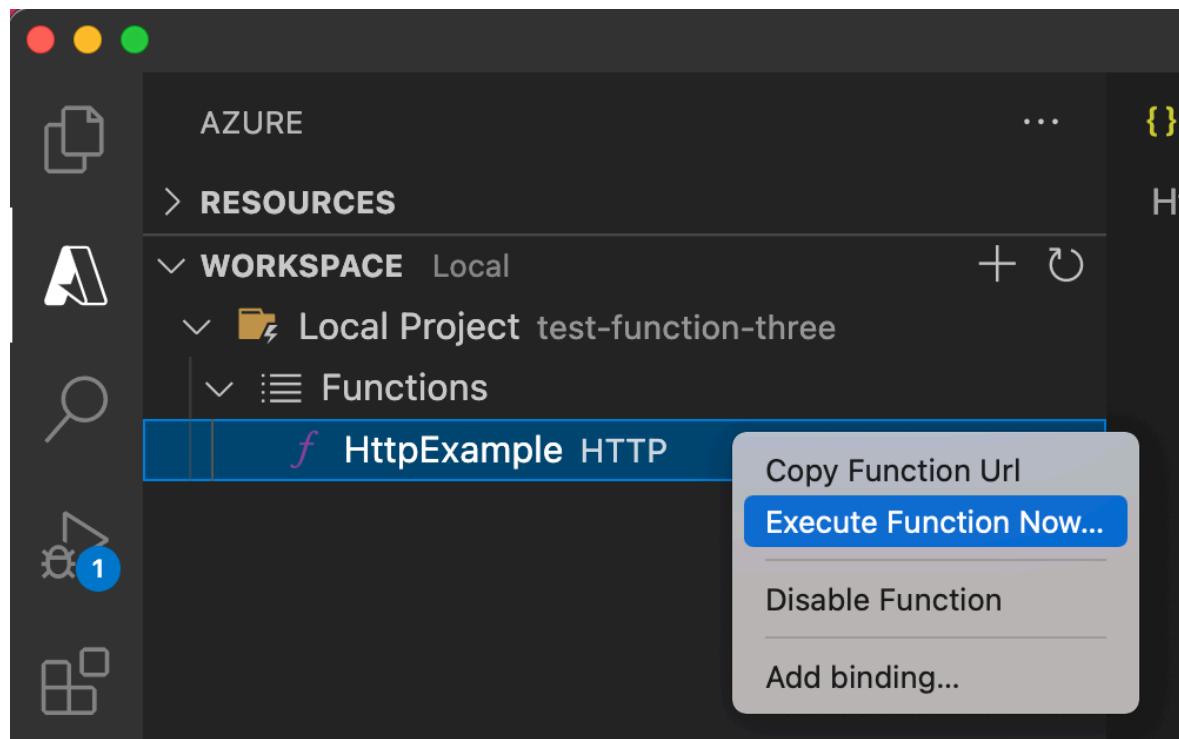
Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.
```

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

- With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or **Ctrl** click (macOS) the new function and choose **Execute Function Now....**



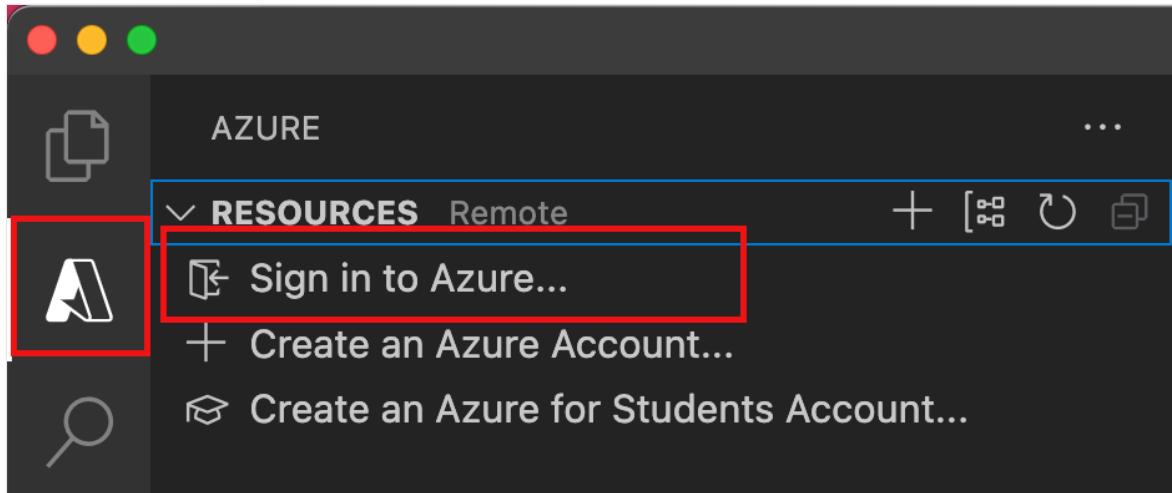
- In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
- When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.
- With the **Terminal** panel focused, press **Ctrl + C** to stop Core Tools and disconnect the debugger.

After you verify that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

2. When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

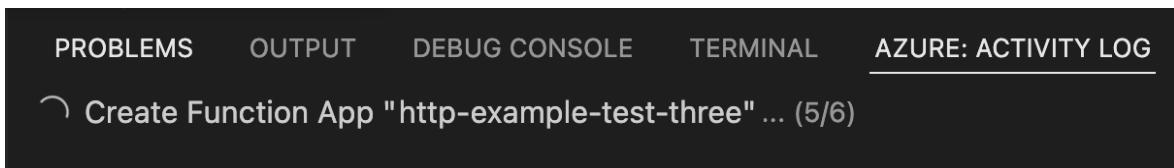
In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors. For more control over the created resources, you must instead [create your function app with advanced options](#).

1. In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
2. At the prompts, provide the following information:

[] [Expand table](#)

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Select the language version you currently run locally.
Select a location for new resources	Select an Azure region. For better performance, select a region near you.

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



- When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

💡 Tip

By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources

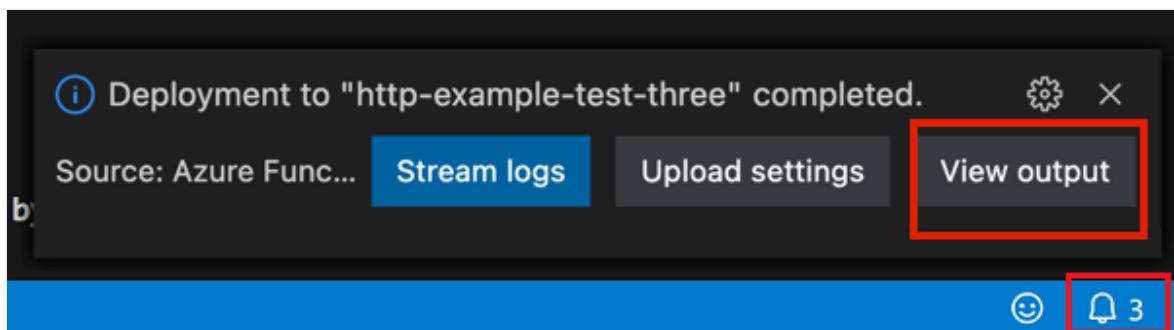
are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Deploy the project to Azure

ⓘ Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Press **F1** to display the command palette, then search for and run the command **Azure Functions:Execute Function Now....**. If prompted, select your subscription.
2. Select your new function app resource and **HttpExample** as your function.
3. In **Enter request body** type `{ "name": "Azure" }`, then press Enter to send this request message to your function.

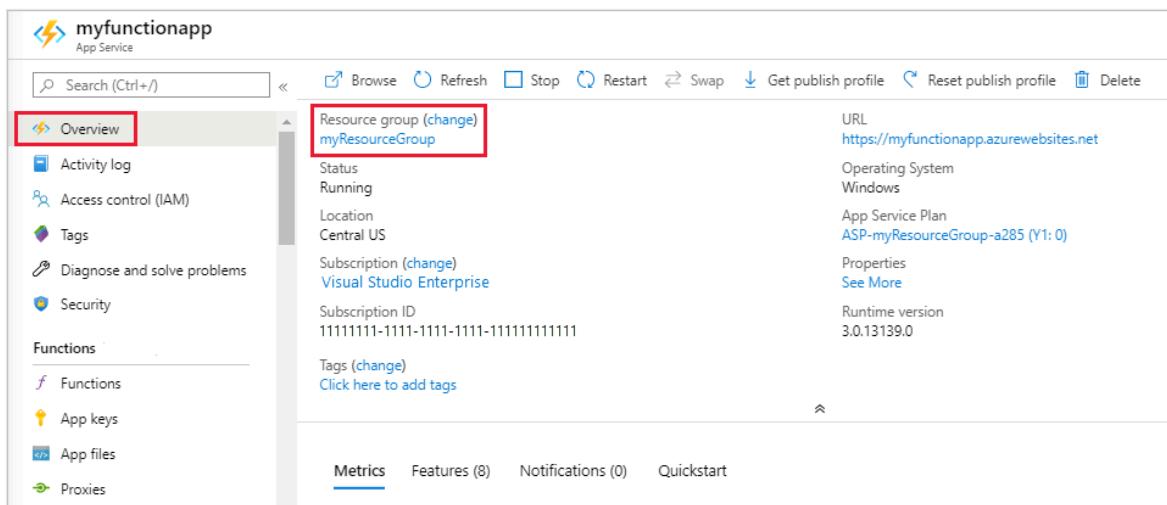
4. When the function executes in Azure, the response is displayed in the notification area. Expand the notification to review the full response.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You created and deployed a function app with a simple HTTP-triggered function. In the next articles, you expand that function by connecting to a storage service in Azure. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to Azure Cosmos DB](#)

[Connect to an Azure Storage queue](#)

Having issues? Let us know. ↗

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create serverless APIs in Visual Studio using Azure Functions and API Management integration

Article • 08/07/2024

REST APIs are often described using an OpenAPI definition (formerly known as Swagger) file. This file contains information about operations in an API and how the request and response data for the API should be structured.

In this tutorial, you learn how to:

- ✓ Create the code project in Visual Studio
- ✓ Install the OpenAPI extension
- ✓ Add an HTTP trigger endpoint, which includes OpenAPI definitions
- ✓ Test function APIs locally using built-in OpenAPI functionality
- ✓ Publish project to a function app in Azure
- ✓ Enable API Management integration
- ✓ Download the OpenAPI definition file

The serverless function you create provides an API that lets you determine whether an emergency repair on a wind turbine is cost-effective. Since you create both the function app and API Management instance in a consumption tier, your cost for completing this tutorial is minimal.

Prerequisites

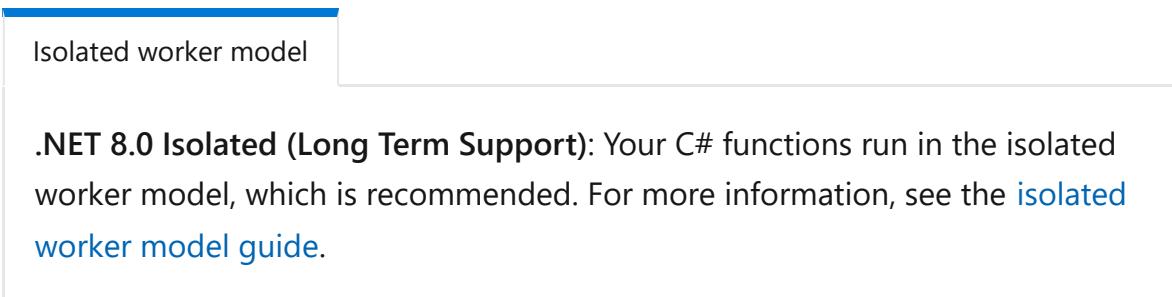
- [Visual Studio 2022](#). Make sure you select the **Azure development** workload during installation.
- An active [Azure subscription](#), [create a free account](#) before you begin.

Create the code project

The Azure Functions project template in Visual Studio creates a project that you can publish to a function app in Azure. You'll also create an HTTP triggered function from a template that supports OpenAPI definition file (formerly Swagger file) generation.

1. From the Visual Studio menu, select **File > New > Project**.

2. In **Create a new project**, enter *functions* in the search box, choose the **Azure Functions** template, and then select **Next**.
3. In **Configure your new project**, enter a **Project name** for your project like `TurbineRepair`, and then select **Create**.
4. For the **Create a new Azure Functions application** settings, select one of these options for **Functions worker**, where the option you choose depends on your chosen process model:



5. For the rest of the options, use the values in the following table:

[] [Expand table](#)

Setting	Value	Description
Function template	Empty	This creates a project without a trigger, which gives you more control over the name of the HTTP triggered function when you add it later.
Use Azurite for runtime storage account (AzureWebJobsStorage)	Selected	You can use the emulator for local development of HTTP trigger functions. Because a function app in Azure requires a storage account, one is assigned or created when you publish your project to Azure.
Authorization level	Function	When running in Azure, clients must provide a key when accessing the endpoint. For more information, see Authorization level .

6. Select **Create** to create the function project.

Next, you update the project by installing the OpenAPI extension for Azure Functions, which enables the discoverability of API endpoints in your app.

Install the OpenAPI extension

To install the OpenAPI extension:

1. From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.
2. In the console, run the following [Install-Package](#) command to install the OpenAPI extension:

The screenshot shows a command-line interface for the Azure Functions Package Manager. At the top, it says "Isolated worker model". Below that, under the heading "command", is the command: "NuGet\Install-Package Microsoft.Azure.Functions.Worker.Extensions.OpenApi -Version 1.5.1". A note below the command says: "You might need to update the [specific version](#), based on your version of .NET."

Now, you can add your HTTP endpoint function.

Add an HTTP endpoint function

In a C# class library, the bindings used by the function are defined by applying attributes in the code. To create a function with an HTTP trigger:

1. In **Solution Explorer**, right-click your project node and select **Add > New Azure Function**.
2. Enter **Turbine.cs** for the class, and then select **Add**.
3. Choose the **Http trigger** template, set **Authorization level** to **Function**, and then select **Add**. A Turbine.cs code file is added to your project that defines a new function endpoint with an HTTP trigger.

Now you can replace the HTTP trigger template code with code that implements the Turbine function endpoint, along with attributes that use OpenAPI to define endpoint.

Update the function code

The function uses an HTTP trigger that takes two parameters:

[\[+\] Expand table](#)

Parameter name	Description
<i>hours</i>	The estimated time to make a turbine repair, up to the nearest whole hour.
<i>capacity</i>	The capacity of the turbine, in kilowatts.

The function then calculates how much a repair costs, and how much revenue the turbine could make in a 24-hour period. Parameters are supplied either in the query string or in the payload of a POST request.

In the `Turbine.cs` project file, replace the contents of the class generated from the HTTP trigger template with the following code, which depends on your process model:

Isolated worker model

```
C#
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.WebJobs.Extensions.OpenApi.Core.Attributes;
using Microsoft.Azure.WebJobs.Extensions.OpenApi.Core.Enums;
using Microsoft.Extensions.Logging;
using Microsoft.OpenApi.Models;
using Newtonsoft.Json;
using System.Net;

namespace TurbineRepair
{
    public class Turbine
    {
        const double revenuePerkW = 0.12;
        const double technicianCost = 250;
        const double turbineCost = 100;

        private readonly ILogger<Turbine> _logger;

        public Turbine(ILogger<Turbine> logger)
        {
            _logger = logger;
        }

        [Function("TurbineRepair")]
        [OpenApiOperation(operationId: "Run")]
        [OpenApiSecurity("function_key", SecuritySchemeType.ApiKey, Name = "code", In = OpenApiSecurityLocationType.Query)]
        [OpenApiRequestBody("application/json", typeof(RequestBodyModel),
            Description = "JSON request body containing { hours, capacity}")]
        [OpenApiResponseWithBody(statusCode: HttpStatusCode.OK,
```

```

    contentType: "application/json", bodyType: typeof(string),
        Description = "The OK response message containing a JSON
result.")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "post", Route =
null)] HttpRequest req,
        ILogger log)
    {
        // Get request body data.
        string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
        dynamic? data = JsonConvert.DeserializeObject(requestBody);
        int? capacity = data?.capacity;
        int? hours = data?.hours;

        // Return bad request if capacity or hours are not passed in
        if (capacity == null || hours == null)
        {
            return new BadRequestObjectResult("Please pass capacity
and hours in the request body");
        }
        // Formulas to calculate revenue and cost
        double? revenueOpportunity = capacity * revenuePerkW * 24;
        double? costToFix = hours * technicianCost + turbineCost;
        string repairTurbine;

        if (revenueOpportunity > costToFix)
        {
            repairTurbine = "Yes";
        }
        else
        {
            repairTurbine = "No";
        };

        return new OkObjectResult(new
        {
            message = repairTurbine,
            revenueOpportunity = "$" + revenueOpportunity,
            costToFix = "$" + costToFix
        });
    }
    public class RequestBodyModel
    {
        public int Hours { get; set; }
        public int Capacity { get; set; }
    }
}
}

```

This function code returns a message of `Yes` or `No` to indicate whether an emergency repair is cost-effective. It also returns the revenue opportunity that the turbine

represents and the cost to fix the turbine.

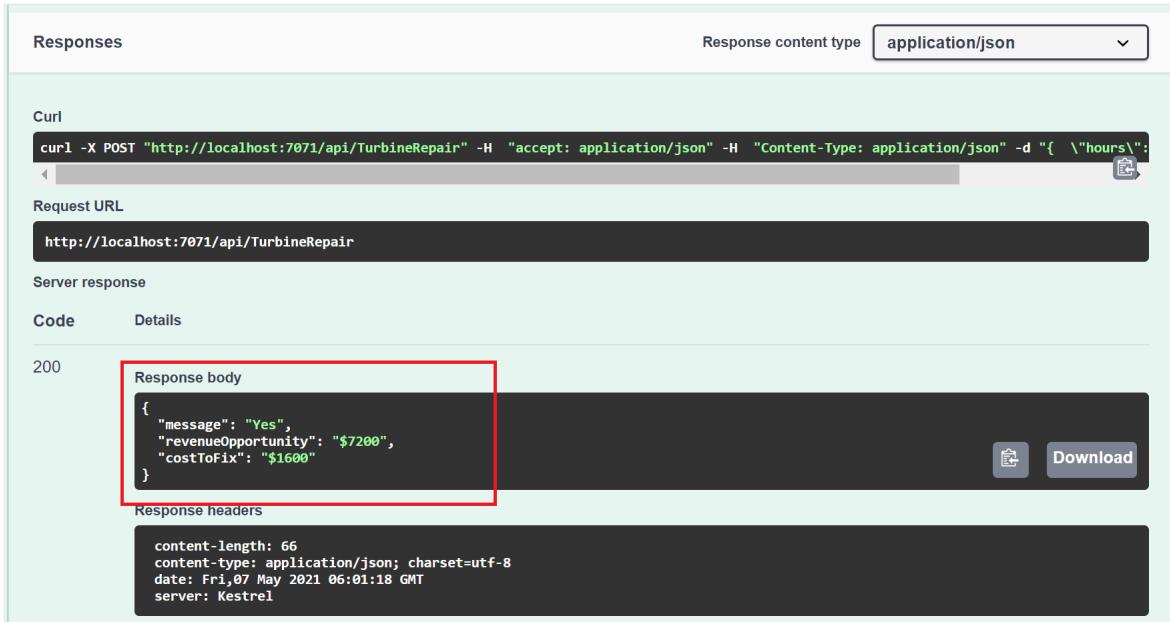
Run and verify the API locally

When you run the function, the OpenAPI endpoints make it easy to try out the function locally using a generated page. You don't need to provide function access keys when running locally.

1. Press F5 to start the project. When Functions runtime starts locally, a set of OpenAPI and Swagger endpoints are shown in the output, along with the function endpoint.
2. In your browser, open the RenderSwaggerUI endpoint, which should look like `http://localhost:7071/api/swagger/ui`. A page is rendered, based on your OpenAPI definitions.
3. Select **POST > Try it out**, enter values for `hours` and `capacity` either as query parameters or in the JSON request body, and select **Execute**.

The screenshot shows the 'Try it out' dialog for a POST request to the '/TurbineRepair' endpoint. The dialog has a green header bar with 'POST' and the endpoint path. Below the header is a 'Parameters' section with a 'Cancel' button. The main body shows a table with 'Name' and 'Description' columns. A row for 'body object (body)' has an 'Edit Value | Model' link and a JSON input field containing '{ "hours": 6, "capacity": 2500 }'. A red box highlights this JSON input field. At the bottom, there's another 'Cancel' button, a 'Parameter content type' dropdown set to 'application/json', and a large blue 'Execute' button highlighted with a red box.

4. When you enter integer values like 6 for `hours` and 2500 for `capacity`, you get a JSON response that looks like the following example:



```
curl -X POST "http://localhost:7071/api/TurbineRepair" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"hours\":6, \"capacity\":2500}"
```

Request URL
`http://localhost:7071/api/TurbineRepair`

Server response

Code Details

200 Response body

```
{  
  "message": "Yes",  
  "revenueOpportunity": "$7200",  
  "costToFix": "$1600"  
}
```

Response headers

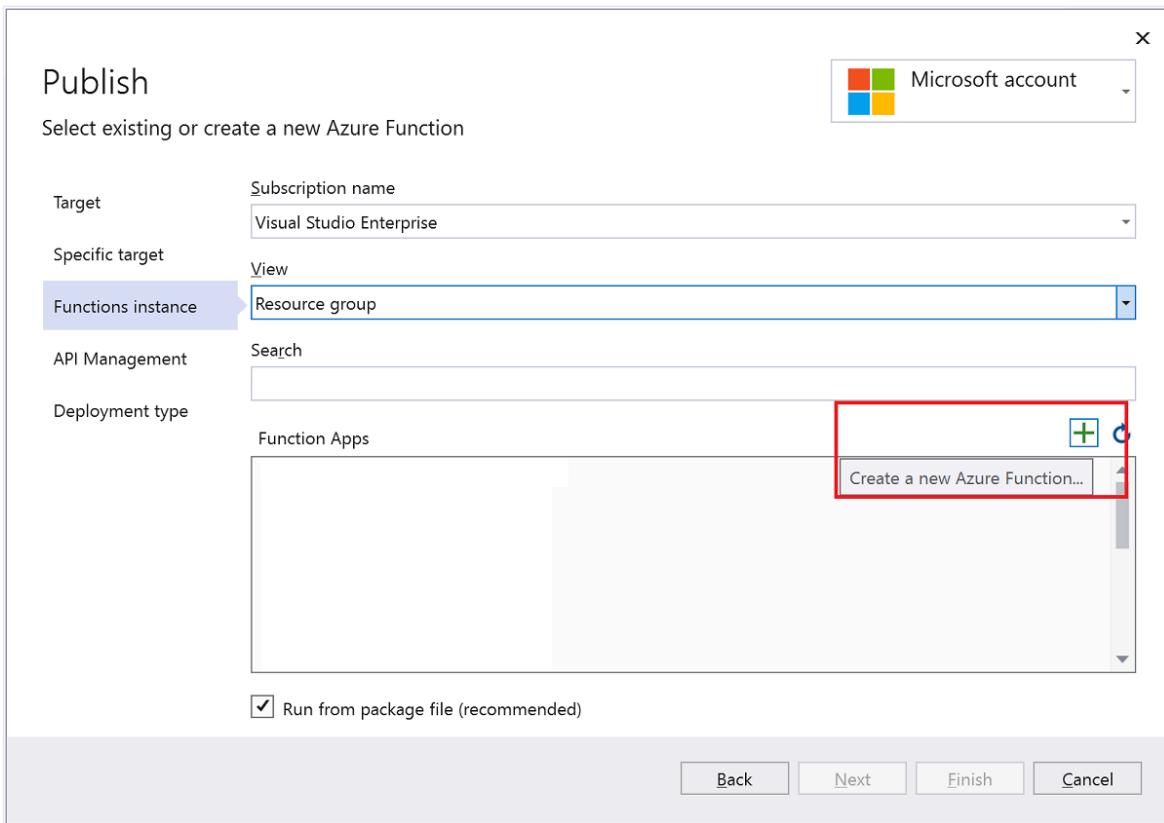
```
content-length: 66  
content-type: application/json; charset=utf-8  
date: Fri, 07 May 2021 06:01:18 GMT  
server: Kestrel
```

Now you have a function that determines the cost-effectiveness of emergency repairs. Next, you publish your project and API definitions to Azure.

Publish the project to Azure

Before you can publish your project, you must have a function app in your Azure subscription. Visual Studio publishing creates a function app the first time you publish your project. It can also create an API Management instance that integrates with your function app to expose the TurbineRepair API.

1. In **Solution Explorer**, right-click the project and select **Publish** and in **Target**, select **Azure** then **Next**.
2. For the **Specific target**, choose **Azure Function App (Windows)** to create a function app that runs on Windows, then select **Next**.
3. In **Function Instance**, choose **+ Create a new Azure Function....**



4. Create a new instance using the values specified in the following table:

[\[...\] Expand table](#)

Setting	Value	Description
Name	Globally unique name	Name that uniquely identifies your new function app. Accept this name or enter a new name. Valid characters are: a-z, 0-9, and -.
Subscription	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the drop-down list.
Resource group	Name of your resource group	The resource group in which to create your function app. Select an existing resource group from the drop-down list or choose New to create a new resource group.
Plan Type	Consumption	When you publish your project to a function app that runs in a Consumption plan , you pay only for executions of your functions app. Other hosting plans incur higher costs.
Location	Location of the service	Choose a Location in a region near you or other services your functions access.
Azure Storage	General-purpose storage account	An Azure Storage account is required by the Functions runtime. Select New to configure a general-purpose

Setting	Value	Description
		storage account. You can also choose an existing account that meets the storage account requirements .

Name: TurbineRepair20210506232328

Subscription name: Visual Studio Enterprise

Resource group: TurbineRepairGroup* [New...](#)

Plan Type: Consumption

Location: South Central US

Azure Storage: turbinerepair20210506232* (East US) [New...](#)

Buttons: Export..., Create, Cancel

5. Select **Create** to create a function app and its related resources in Azure. Status of resource creation is shown in the lower left of the window.
6. Back in **Functions instance**, make sure that **Run from package file** is checked. Your function app is deployed using [Zip Deploy](#) with [Run-From-Package](#) mode enabled. This deployment method is recommended for your functions project, since it results in better performance.
7. Select **Next**, and in **API Management** page, also choose **+ Create an API Management API**.
8. Create an **API in API Management** by using values in the following table:

[Expand table](#)

Setting	Value	Description
API name	TurbineRepair	Name for the API.
Subscription name	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the drop-down list.
Resource group	Name of your resource group	Select the same resource group as your function app from the drop-down list.
API Management service	New instance	Select New to create a new API Management instance in the same location in the serverless tier. Select OK to create the instance.

API in API Management

Create new

Microsoft account

X

API name

TurbineRepair

Subscription name

Visual Studio Enterprise

Resource group

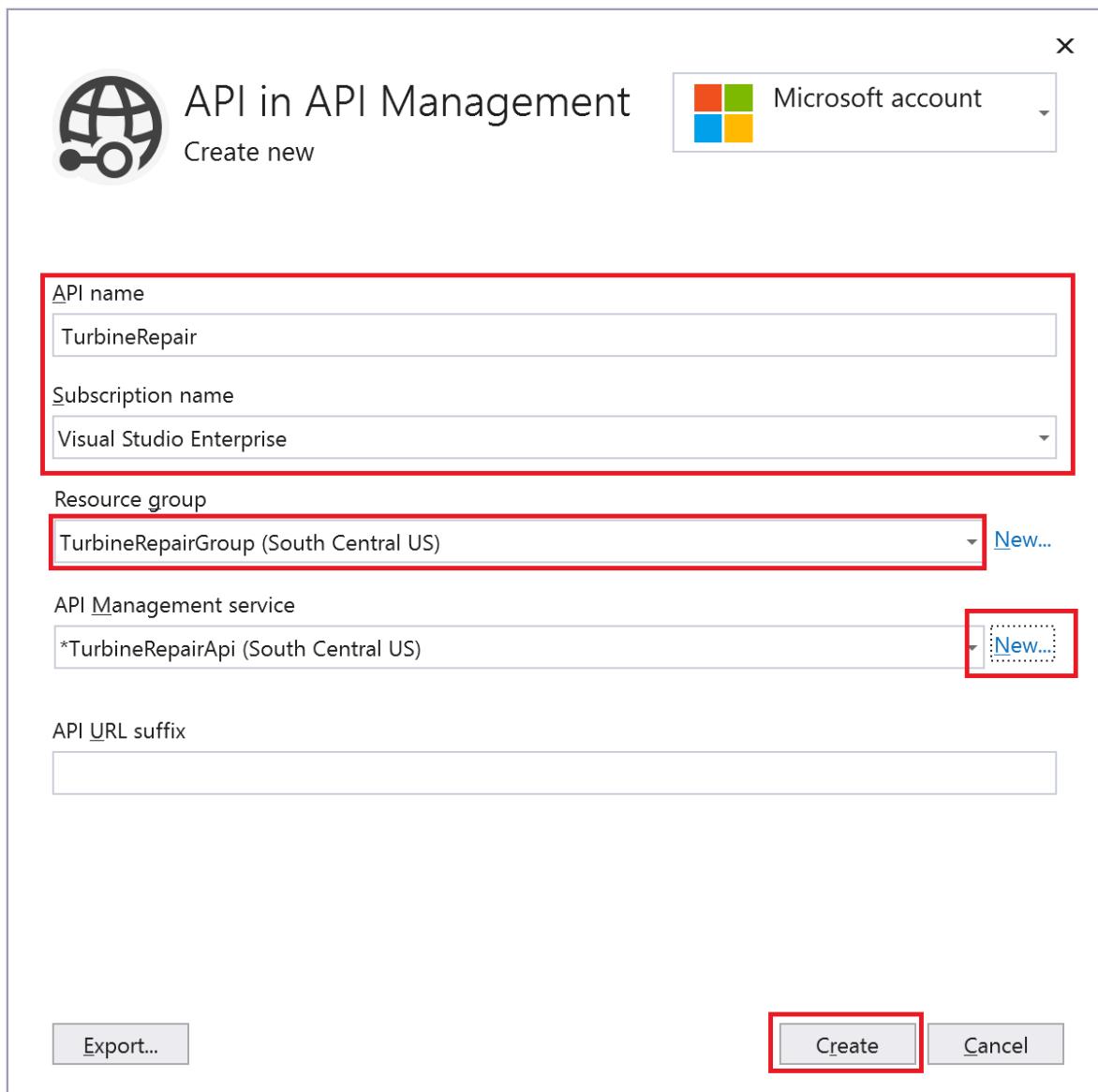
TurbineRepairGroup (South Central US)

API Management service

*TurbineRepairApi (South Central US)

API URL suffix

Export... Create Cancel



9. Select **Create** to create the API Management instance with the TurbineRepair API from the function integration.

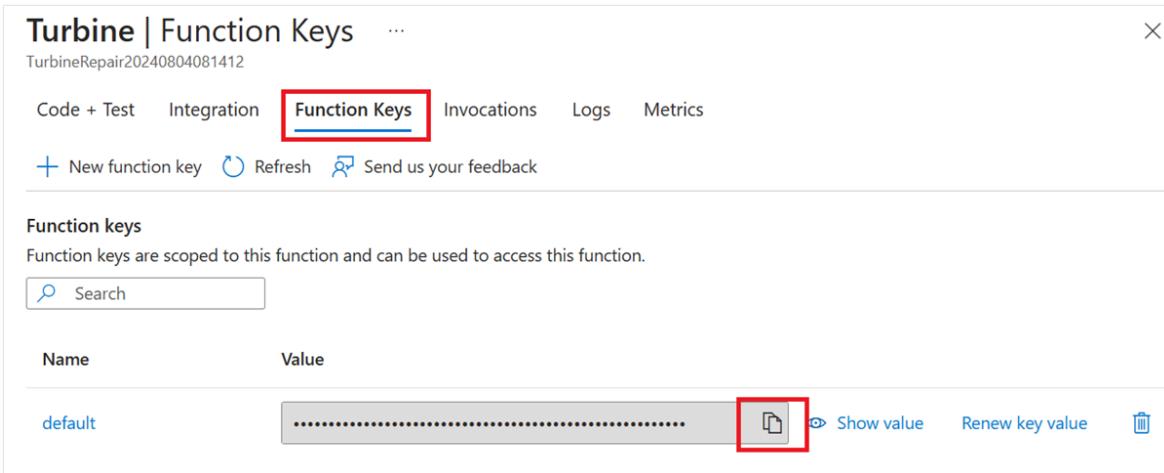
10. Select **Finish** and after the publish profile creation process completes, select **Close**.

11. Verify the Publish page now says **Ready to publish**, and then select **Publish** to deploy the package containing your project files to your new function app in Azure.

After the deployment completes, the root URL of the function app in Azure is shown in the **Publish** tab.

Get the function access key

1. In the **Publish** tab, select the ellipses (...) next to **Hosting** and select **Open in Azure portal**. The function app you created is opened in the Azure portal in your default browser.
2. Under **Functions** on the **Overview page**, select > **Turbine** then select **Function keys**.



The screenshot shows the 'Turbine | Function Keys' blade in the Azure portal. At the top, there's a navigation bar with tabs: 'Code + Test', 'Integration', 'Function Keys' (which is highlighted with a red box), 'Invocations', 'Logs', and 'Metrics'. Below the tabs, there are buttons for '+ New function key', 'Refresh', and 'Send us your feedback'. The main area is titled 'Function keys' and contains the sub-instruction: 'Function keys are scoped to this function and can be used to access this function.' Below this, there's a search bar labeled 'Search'. A table lists function keys, with one row shown: 'Name' (default) and 'Value' (redacted). To the right of the value column are icons: a copy-to-clipboard icon (highlighted with a red box), 'Show value', 'Renew key value', and a delete icon.

3. Under **Function keys**, select the *copy to clipboard* icon next to the **default** key. You can now set this key you copied in API Management so that it can access the function endpoint.

Configure API Management

1. In the function app page, expand **API** and select **API Management**.
2. If the function app isn't already connected to the new API Management instance, select it under **API Management**, select **API > OpenAPI Document on Azure Functions**, make sure **Import functions** is checked, and select **Link API**. Make sure that only **TurbineRepair** is selected for import and then **Select**.

3. Select **Go to API Management** at the top of the page, and in the API Management instance, expand **APIs**.

4. Under **APIs > All APIs**, select **OpenAPI Document on Azure Functions > POST Run**, then under **Inbound processing** select **Add policy > Set query parameters**.

5. Below **Inbound processing**, in **Set query parameters**, type `code` for **Name**, select **+Value**, paste in the copied function key, and select **Save**. API Management includes the function key when it passes calls through to the function endpoint.

The screenshot shows the 'Inbound processing' section of the API Management configuration. On the left, there's a sidebar with 'Design', 'Settings', 'Test', 'Revisions', and 'Change log'. The main area shows the path 'OpenAPI Document on Azure Fu... > Run > Policies'. The 'Inbound processing' section has a sub-section titled 'Set query parameters' with the instruction 'Modify the request before it is sent to the backend service.' It contains a table:

NAME	VALUE	ACTION	DELETE
code	(redacted)	override	

At the bottom of the table, there's a '+ Add parameter' link. At the very bottom of the page, there are 'Operations' and 'Definitions' tabs, and a 'Save' button which is highlighted with a red box, and a 'Discard' button.

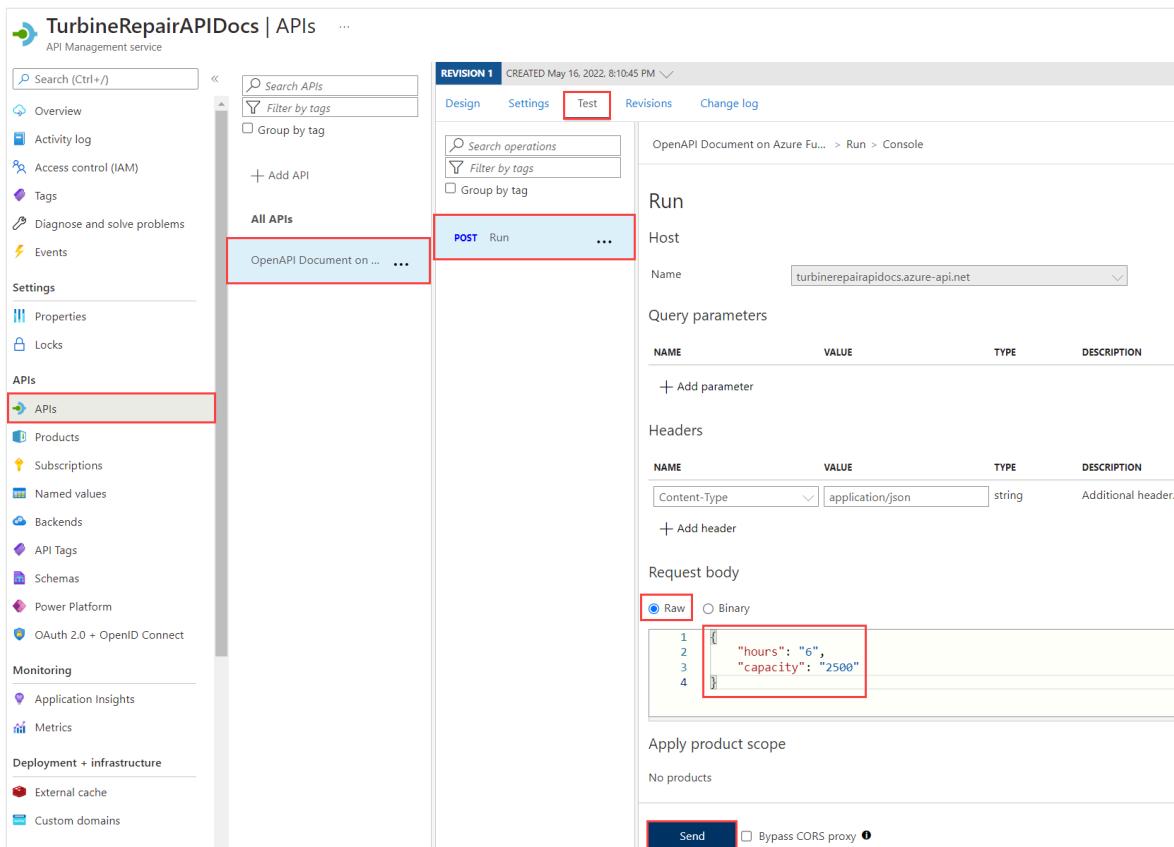
Now that the function key is set, you can call the `turbine` API endpoint to verify that it works when hosted in Azure.

Verify the API in Azure

1. In the API, select the **Test** tab and then **POST Run**, enter the following code in the **Request body > Raw**, and select **Send**:

The screenshot shows the 'Test' tab of the API configuration. The 'Request body > Raw' section contains the following JSON:

```
{  
  "hours": "6",  
  "capacity": "2500"  
}
```



As before, you can also provide the same values as query parameters.

2. Select **Send**, and then view the **HTTP response** to verify the same results are returned from the API.

Download the OpenAPI definition

If your API works as expected, you can download the OpenAPI definition for the new hosted APIs from API Management.

1. a. Under **APIs**, select **OpenAPI Document on Azure Functions**, select the ellipses (...), and select **Export**.

The screenshot shows the Azure API Management service interface. On the left, there's a sidebar with various navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Properties and Locks), APIs (Products, Subscriptions, Named values, Backends, API Tags), and a main APIs section. The 'APIs' link under 'APIs' is highlighted with a red box. In the main content area, there's a search bar for APIs, a 'Filter by tags' button, and a 'Group by tag' checkbox. Below that is a '+ Add API' button and a 'All APIs' section. An API entry is selected, with its details shown: 'OpenAPI Document on ...' and a three-dot ellipsis button. To the right of the API entry is a context menu with several options: Clone (with a copy icon), Add revision (with a pencil icon), Add version (with a list icon), Import (with an upward arrow icon), Export (highlighted with a red box and a downward arrow icon), Create Power Connector (with a diamond icon), and Delete (with a trash bin icon).

2. Choose the means of API export, including OpenAPI files in various formats. You can also [export APIs from Azure API Management to the Power Platform](#).

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select the group you created.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type the name of your group in the text box to confirm, and then select **Delete**.

Next steps

You've used Visual Studio 2022 to create a function that's self-documenting because of the [OpenAPI Extension](#) and integrated with API Management. You can now refine the definition in API Management in the portal. You can also [learn more about API Management](#).

[Edit the OpenAPI definition in API Management](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Tutorial: Integrate Azure Functions with an Azure virtual network by using private endpoints

Article • 03/30/2023

This tutorial shows you how to use Azure Functions to connect to resources in an Azure virtual network by using private endpoints. You create a new function app using a new storage account that's locked behind a virtual network via the Azure portal. The virtual network uses a Service Bus queue trigger.

In this tutorial, you'll:

- ✓ Create a function app in the Elastic Premium plan with virtual network integration and private endpoints.
- ✓ Create Azure resources, such as the Service Bus
- ✓ Lock down your Service Bus behind a private endpoint.
- ✓ Deploy a function app that uses both the Service Bus and HTTP triggers.
- ✓ Test to see that your function app is secure inside the virtual network.
- ✓ Clean up resources.

Create a function app in a Premium plan

You create a C# function app in an [Elastic Premium plan](#), which supports networking capabilities such as virtual network integration on create along with serverless scale. This tutorial uses C# and Windows. Other languages and Linux are also supported.

1. On the Azure portal menu or the [Home](#) page, select [Create a resource](#).
2. On the [New](#) page, select [Compute > Function App](#).
3. On the [Basics](#) page, use the following table to configure the function app settings.

Setting	Suggested value	Description
Subscription	Your subscription	Subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group where you create your function app.
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z (case insensitive), 0-9, and -.

Setting	Suggested value	Description
Publish	Code	Choose to publish code files or a Docker container.
Runtime stack	.NET	This tutorial uses .NET.
Version	6 (LTS)	This tutorial uses .NET 6.0 running in the same process as the Functions host .
Region	Preferred region	Choose a region near you or near other services that your functions access.
Operating system	Windows	This tutorial uses Windows but also works for Linux.
Plan	Functions Premium	<p>Hosting plan that defines how resources are allocated to your function app. By default, when you select Premium, a new App Service plan is created. The default Sku and size is EP1, where <i>EP</i> stands for <i>elastic premium</i>. For more information, see the list of Premium SKUs.</p> <p>When you run JavaScript functions on a Premium plan, choose an instance that has fewer vCPUs. For more information, see Choose single-core Premium plans.</p>

4. Select **Next: Storage**. On the **Storage** page, enter the following settings.

Setting	Suggested value	Description
Storage account	Globally unique name	Create a storage account used by your function app. Storage account names must be between 3 and 24 characters long. They may contain numbers and lowercase letters only. You can also use an existing account that isn't restricted by firewall rules and meets the storage account requirements . When using Functions with a locked down storage account, a v2 storage account is needed. This is the default storage version created when creating a function app with networking capabilities through the create blade.

5. Select **Next: Networking**. On the **Networking** page, enter the following settings.

! **Note**

Some of these settings aren't visible until other options are selected.

Setting	Suggested value	Description
Enable public access	Off	Deny public network access will block all incoming traffic except that comes from private endpoints.
Enable network injection	On	The ability to configure your application with VNet integration at creation appears in the portal window after this option is switched to On .
Virtual Network	Create New	Select the Create New field. In the pop-out screen, provide a name for your virtual network and select Ok . Options to restrict inbound and outbound access to your function app on create are displayed. You must explicitly enable VNet integration in the Outbound access portion of the window to restrict outbound access.

Enter the following settings for the **Inbound access** section. This step creates a private endpoint on your function app.

💡 Tip

To continue interacting with your function app from portal, you'll need to add your local computer to the virtual network. If you don't wish to restrict inbound access, skip this step.

Setting	Suggested value	Description
Enable private endpoints	On	The ability to configure your application with VNet integration at creation appears in the portal after this option is enabled.
Private endpoint name	myInboundPrivateEndpointName	Name that identifies your new function app private endpoint.
Inbound subnet	Create New	This option creates a new subnet for your inbound private endpoint. Multiple private endpoints may be added to a singular subnet. Provide a Subnet Name . The Subnet Address Block may be left at the default value. Select Ok . To learn more about subnet sizing, see Subnets .

Setting	Suggested value	Description
DNS	Azure Private DNS Zone	This value indicates which DNS server your private endpoint uses. In most cases if you're working within Azure, Azure Private DNS Zone is the DNS zone you should use as using Manual for custom DNS zones have increased complexity.

Enter the following settings for the **Outbound access** section. This step integrates your function app with a virtual network on creation. It also exposes options to create private endpoints on your storage account and restrict your storage account from network access on create. When function app is vnet integrated, all outbound traffic by default goes [through the vnet..](#)

Setting	Suggested value	Description
Enable VNet Integration	On	This integrates your function app with a VNet on create and direct all outbound traffic through the VNet.
Outbound subnet	Create new	This creates a new subnet for your function app's VNet integration. A function app can only be VNet integrated with an empty subnet. Provide a Subnet Name . The Subnet Address Block may be left at the default value. If you wish to configure it, please learn more about Subnet sizing here. Select Ok . The option to create Storage private endpoints is displayed. To use your function app with virtual networks, you need to join it to a subnet.

Enter the following settings for the **Storage private endpoint** section. This step creates private endpoints for the blob, queue, file, and table endpoints on your storage account on create. This effectively integrates your storage account with the VNet.

Setting	Suggested value	Description
Add storage private endpoint	On	The ability to configure your application with VNet integration at creation is displayed in the portal after this option is enabled.
Private endpoint name	myInboundPrivateEndpointName	Name that identifies your storage account private endpoint.

Setting	Suggested value	Description
Private endpoint subnet	Create New	This creates a new subnet for your inbound private endpoint on the storage account. Multiple private endpoints may be added to a singular subnet. Provide a Subnet Name . The Subnet Address Block may be left at the default value. If you wish to configure it, please learn more about Subnet sizing here. Select Ok .
DNS	Azure Private DNS Zone	This value indicates which DNS server your private endpoint uses. In most cases if you're working within Azure, Azure Private DNS Zone is the DNS zone you should use as using Manual for custom DNS zones will have increased complexity.

6. Select **Next: Monitoring**. On the **Monitoring** page, enter the following settings.

Setting	Suggested value	Description
Application Insights	Default	Create an Application Insights resource of the same app name in the nearest supported region. Expand this setting if you need to change the New resource name or store your data in a different Location in an Azure geography .

7. Select **Review + create** to review the app configuration selections.

8. On the **Review + create** page, review your settings. Then select **Create** to create and deploy the function app.

9. In the upper-right corner of the portal, select the **Notifications** icon and watch for the **Deployment succeeded** message.

10. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

Congratulations! You've successfully created your premium function app.

Note

Some deployments may occasionally fail to create the private endpoints in the storage account with the error 'StorageAccountOperationInProgress'. This failure occurs even though the function app itself gets created successfully. When you

encounter such an error, delete the function app and retry the operation. You can instead create the private endpoints on the storage account manually.

Create a Service Bus

Next, you create a Service Bus instance that is used to test the functionality of your function app's network capabilities in this tutorial.

1. On the Azure portal menu or the **Home** page, select **Create a resource**.
2. On the **New** page, search for *Service Bus*. Then select **Create**.
3. On the **Basics** tab, use the following table to configure the Service Bus settings. All other settings can use the default values.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription in which your resources are created.
Resource group	myResourceGroup	The resource group you created with your function app.
Namespace name	myServiceBus	The name of the Service Bus instance for which the private endpoint is enabled.
Location	myFunctionRegion	The region where you created your function app.
Pricing tier	Premium	Choose this tier to use private endpoints with Azure Service Bus.

4. Select **Review + create**. After validation finishes, select **Create**.

Lock down your Service Bus

Create the private endpoint to lock down your Service Bus:

1. In your new Service Bus, in the menu on the left, select **Networking**.
2. On the **Private endpoint connections** tab, select **Private endpoint**.

The screenshot shows the Azure portal interface for a Service Bus Namespace named 'myServiceBus-tutorial'. The left sidebar has a 'Networking' section with several options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Shared access policies, Scale, Geo-Recovery), Networking (which is highlighted with a red box), and Encryption.

The main content area is titled 'Firewalls and virtual networks' and shows a table for 'Private endpoint connections'. The table includes columns for Connection name, Connection state, and actions (Approve, Reject, Remove, Refresh). A red box highlights the 'Private endpoint' button in the top-left of the table header.

3. On the **Basics** tab, use the private endpoint settings shown in the following table.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription in which your resources are created.
Resource group	myResourceGroup	The resource group you created with your function app.
Name	sb-endpoint	The name of the private endpoint for the service bus.
Region	myFunctionRegion	The region where you created your storage account.

4. On the **Resource** tab, use the private endpoint settings shown in the following table.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Resource type	Microsoft.ServiceBus/namespaces	The resource type for the Service Bus.
Resource	myServiceBus	The Service Bus you created earlier in the tutorial.

Setting	Suggested value	Description
Target subresource	namespace	The private endpoint that is used for the namespace from the Service Bus.

5. On the **Virtual Network** tab, for the **Subnet** setting, choose **default**.
6. Select **Review + create**. After validation finishes, select **Create**.
7. After the private endpoint is created, return to the **Networking** section of your Service Bus namespace and check the **Public Access** tab.
8. Ensure **Selected networks** is selected.
9. Select **+ Add existing virtual network** to add the recently created virtual network.
10. On the **Add networks** tab, use the network settings from the following table:

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Virtual networks	myVirtualNet	The name of the virtual network to which your function app connects.
Subnets	functions	The name of the subnet to which your function app connects.

11. Select **Add your client IP address** to give your current client IP access to the namespace.

 **Note**

Allowing your client IP address is necessary to enable the Azure portal to publish messages to the queue later in this tutorial.

12. Select **Enable** to enable the service endpoint.
13. Select **Add** to add the selected virtual network and subnet to the firewall rules for the Service Bus.
14. Select **Save** to save the updated firewall rules.

Resources in the virtual network can now communicate with the Service Bus using the private endpoint.

Create a queue

Create the queue where your Azure Functions Service Bus trigger gets events:

1. In your Service Bus, in the menu on the left, select **Queues**.
2. Select **Queue**. For the purposes of this tutorial, provide the name *queue* as the name of the new queue.

The screenshot shows the Azure portal interface for creating a new queue. On the left, the navigation menu is open, with the 'Queues' option under the 'Entities' section highlighted by a red box. In the center, there is a list of queues with the message 'No results.' Below this is a search bar. To the right, a 'Create queue' dialog box is open. The 'Name' field contains 'queue', which is also highlighted by a red box. Other settings in the dialog include 'Max queue size: 1 GB', 'Max delivery count: 10', 'Message time to live: 14 Days, 0 Hours, 0 Minutes, 0 Seconds', and 'Lock duration: 0 Days, 0 Hours, 0 Minutes, 30 Seconds'. There are several optional checkboxes at the bottom, none of which are checked. A large blue 'Create' button is at the bottom right of the dialog.

3. Select **Create**.

Get a Service Bus connection string

1. In your Service Bus, in the menu on the left, select **Shared access policies**.
2. Select **RootManageSharedAccessKey**. Copy and save the **Primary Connection String**. You need this connection string when you configure the app settings.

Configure your function app settings

1. In your function app, in the menu on the left, select **Configuration**.
2. To use your function app with virtual networks and service bus, update the app settings shown in the following table. To add or edit a setting, select **+ New application setting** or the **Edit** icon in the rightmost column of the app settings table. When you finish, select **Save**.

Setting	Suggested value	Description
SERVICEBUS_CONNECTION	myServiceBusConnectionString	Create this app setting for the connection string of your Service Bus. This storage connection string is from the Get a Service Bus connection string section.
WEBSITE_CONTENTOVERVNET	1	Create this app setting. A value of 1 enables your function app to scale when your storage account is restricted to a virtual network.

3. Since you're using an Elastic Premium hosting plan, In the **Configuration** view, select the **Function runtime settings** tab. Set **Runtime Scale Monitoring** to **On**. Then select **Save**. Runtime-driven scaling allows you to connect non-HTTP trigger functions to services that run inside your virtual network.

The screenshot shows the Azure Function App Configuration page for a 'scaling-sample' function app. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, and Events (preview). The main area has tabs for Application settings, Function runtime settings (which is selected and highlighted in blue), and General settings. Under Function runtime settings, there's a 'Runtime version' dropdown set to '~4'. Below it is a section titled 'Runtime Scale Monitor...' with a radio button set to 'On'. A note below says: 'Enable this setting to allow your app to be scaled based on runtime metrics.'

① Note

Runtime scaling isn't needed for function apps hosted in a Dedicated App Service plan.

Deploy a Service Bus trigger and HTTP trigger

① Note

Enabling private endpoints on a function app also makes the Source Control Manager (SCM) site publicly inaccessible. The following instructions give deployment directions using the Deployment Center within the function app. Alternatively, use **zip deploy** or **self-hosted agents** that are deployed into a subnet on the virtual network.

1. In GitHub, go to the following sample repository. It contains a function app and two functions, an HTTP trigger, and a Service Bus queue trigger.
<https://github.com/Azure-Samples/functions-vnet-tutorial>
2. At the top of the page, select **Fork** to create a fork of this repository in your own GitHub account or organization.
3. In your function app, in the menu on the left, select **Deployment Center**. Then select **Settings**.
4. On the **Settings** tab, use the deployment settings shown in the following table.

Setting	Suggested value	Description
---------	-----------------	-------------

Setting	Suggested value	Description
Source	GitHub	You should have created a GitHub repository for the sample code in step 2.
Organization	myOrganization	The organization your repo is checked into. It's usually your account.
Repository	functions-vnet-tutorial	The repository forked from https://github.com/Azure-Samples/functions-vnet-tutorial .
Branch	main	The main branch of the repository you created.
Runtime stack	.NET	The sample code is in C#.
Version	.NET Core 3.1	The runtime version.

5. Select Save.

vnet-app-tutorial | Deployment Center

Save Discard Browse Manage publish profile Redeploy/Sync Leave Feedback

Logs Settings * FTPS credentials

You're now in the production slot, which is not recommended for setting up CI/CD. Learn more

Deploy and build code from your preferred source and build provider. Learn more

Source* GitHub

Building with GitHub Actions. Change provider.

GitHub

If you can't find an organization or repository, you may need to enable additional permissions on GitHub. Learn more

Signed in as cachai2 Change Account

Organization* cachai2

Repository* functions-vnet-tutorial

Branch* master

Workflow Option* Add a workflow: Add a new workflow file 'master_vnet-app-tutorial.yml' in the selected repository and branch. Use available workflow: Use one of the workflow files available in the selected repository and branch.

Build

Runtime stack* .NET

Version* .NET Core 3.1

Deployment slots Deployment Center (Classic) Deployment Center

Settings Configuration Authentication / Authorization Authentication (preview) Application Insights Identity

6. Your initial deployment might take a few minutes. When your app is successfully deployed, on the Logs tab, you see a Success (Active) status message. If necessary, refresh the page.

Congratulations! You've successfully deployed your sample function app.

Test your locked-down function app

1. In your function app, in the menu on the left, select **Functions**.

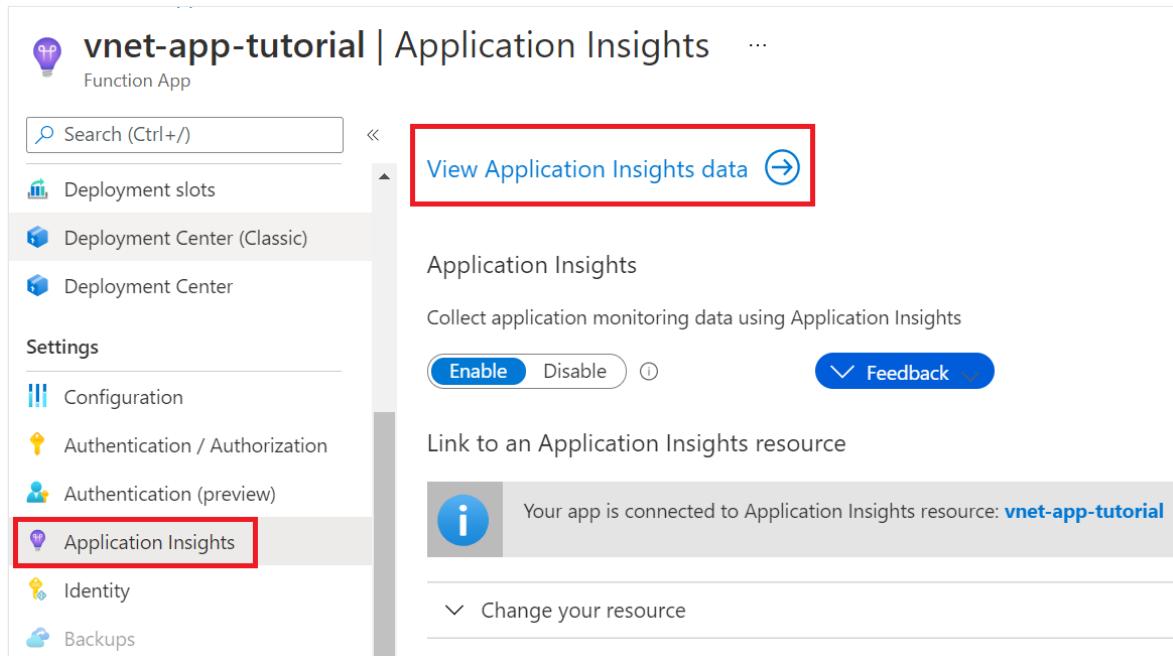
2. Select **ServiceBusQueueTrigger**.

3. In the menu on the left, select **Monitor**.

You see that you can't monitor your app. Your browser doesn't have access to the virtual network, so it can't directly access resources within the virtual network.

Here's an alternative way to monitor your function by using Application Insights:

1. In your function app, in the menu on the left, select **Application Insights**. Then select **View Application Insights data**.



2. In the menu on the left, select **Live metrics**.

3. Open a new tab. In your Service Bus, in the menu on the left, select **Queues**.

4. Select your queue.

5. In the menu on the left, select **Service Bus Explorer**. Under **Send**, for **Content Type**, choose **Text/Plain**. Then enter a message.

6. Select **Send** to send the message.

The screenshot shows the Service Bus Explorer interface for a queue named 'queue' under the namespace 'myServiceBus-tutorial'. The left sidebar contains navigation links for Overview, Access control (IAM), Diagnose and solve problems, Settings (with Shared access policies and Service Bus Explorer (preview) selected), Properties, Locks, Automation (Tasks (preview) and Export template), and Support + troubleshooting (New support request). The main area has tabs for Send, Receive, and Peek, with 'Send' selected. A form titled 'Send Message to Queue **queue**' is displayed, with 'Content Type *' set to 'Text/Plain' and the message body containing 'Hello World'. A red box highlights the message body input field. Below the message body is a checkbox for 'Expand Advanced Properties' and a section for 'Custom Properties' with two empty text fields for Name and Value. At the bottom is a large red-bordered 'Send' button.

- On the Live metrics tab, you should see that your Service Bus queue trigger has fired. If it hasn't, resend the message from Service Bus Explorer.

The screenshot shows the Application Insights Live metrics page for the 'vnet-app-tutorial' application. The left sidebar includes Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Investigate (Application map, Smart Detection, and Live metrics selected), Transaction search, and Availability. The main area displays 'Incoming Requests' and 'Outgoing Requests' metrics. On the right, the 'Sample telemetry' section shows a list of trace logs. One log entry is highlighted with a red box: '45502 AM | Trace @c221a34ce7ffe37...1462144ac5e60b2 C# ServiceBus queue trigger function processed message: Hello World'. This indicates the function was triggered by a message from the Service Bus queue.

Congratulations! You've successfully tested your function app setup with private endpoints.

Understand private DNS zones

You've used a private endpoint to connect to Azure resources. You're connecting to a private IP address instead of the public endpoint. Existing Azure services are configured to use an existing DNS to connect to the public endpoint. You must override the DNS configuration to connect to the private endpoint.

A private DNS zone is created for each Azure resource that was configured with a private endpoint. A DNS record is created for each private IP address associated with the

private endpoint.

The following DNS zones were created in this tutorial:

- privatelink.file.core.windows.net
- privatelink.blob.core.windows.net
- privatelink.servicebus.windows.net
- privatelink.azurewebsites.net

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

In this tutorial, you created a Premium function app, storage account, and Service Bus. You secured all of these resources behind private endpoints.

Use the following links to learn more Azure Functions networking options and private endpoints:

- [How to configure Azure Functions with a virtual network](#)
- [Networking options in Azure Functions](#)
- [Azure Functions Premium plan](#)
- [Service Bus private endpoints](#)
- [Azure Storage private endpoints](#)

Tutorial: Establish Azure Functions private site access

Article • 06/28/2022

This tutorial shows you how to enable [private site access](#) with Azure Functions. By using private site access, you can require that your function code is only triggered from a specific virtual network.

Private site access is useful in scenarios when access to the function app needs to be limited to a specific virtual network. For example, the function app may be applicable to only employees of a specific organization, or services which are within the specified virtual network (such as another Azure Function, Azure Virtual Machine, or an AKS cluster).

If a Functions app needs to access Azure resources within the virtual network, or connected via [service endpoints](#), then [virtual network integration](#) is needed.

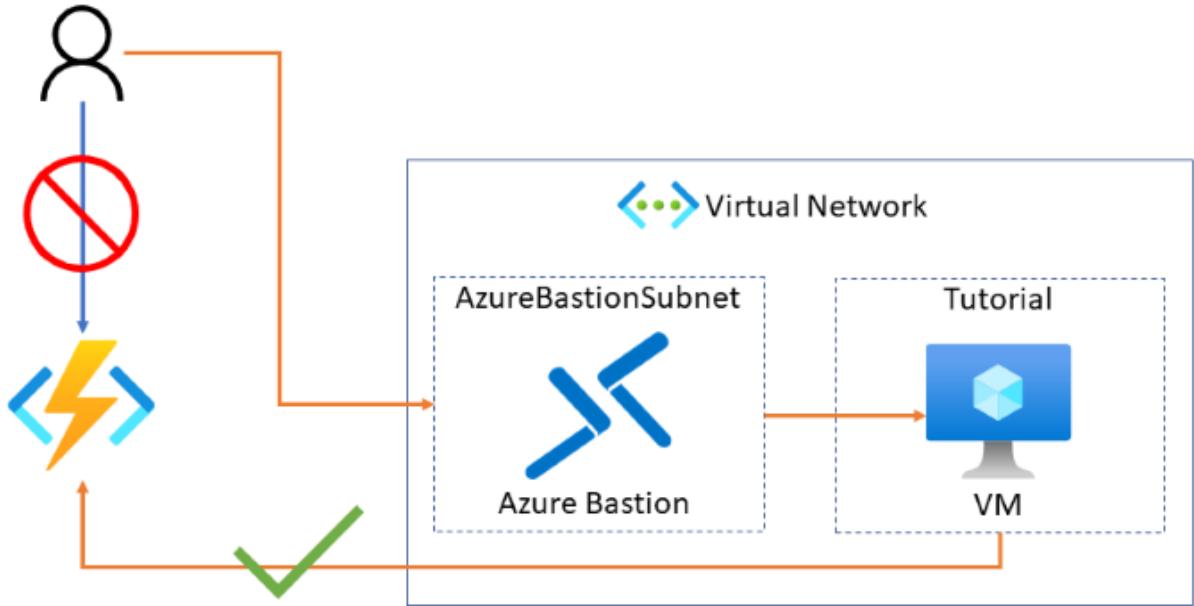
In this tutorial, you learn how to configure private site access for your function app:

- ✓ Create a virtual machine
- ✓ Create an Azure Bastion service
- ✓ Create an Azure Functions app
- ✓ Configure a virtual network service endpoint
- ✓ Create and deploy an Azure Function
- ✓ Invoke the function from outside and within the virtual network

If you don't have an Azure subscription, create a [free account](#) before you begin.

Topology

The following diagram shows the architecture of the solution to be created:



Prerequisites

For this tutorial, it's important that you understand IP addressing and subnetting. You can start with [this article that covers the basics of addressing and subnetting ↗](#). Many more articles and videos are available online.

Sign in to Azure portal

Sign in to the [Azure portal ↗](#).

Create a virtual machine

The first step in this tutorial is to create a new virtual machine inside a virtual network. The virtual machine will be used to access your function once you've restricted its access to only be available from within the virtual network.

1. Select the **Create a resource** button.
2. In the search field, type **Windows Server**, and select **Windows Server** in the search results.
3. Select **Windows Server 2019 Datacenter** from the list of Windows Server options, and press the **Create** button.
4. In the *Basics* tab, use the VM settings as specified in the table below the image:

Create a virtual machine

[Basics](#) [Disks](#) [Networking](#) [Management](#) [Advanced](#) [Tags](#) [Review + create](#)

Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image.

Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization.

Looking for classic VMs? [Create VM from Azure Marketplace](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Visual Studio Ultimate

Resource group * ⓘ

(New) myResourceGroup

[Create new](#)

Instance details

Virtual machine name * ⓘ

myVM

Region * ⓘ

(US) North Central US

Availability options ⓘ

No infrastructure redundancy required

Image * ⓘ

Windows Server 2019 Datacenter

[Browse all public and private images](#)

Azure Spot instance ⓘ

Yes No

Size * ⓘ

Standard DS1 v2

1 vcpu, 3.5 GiB memory (\$53.29/month)

[Change size](#)

Administrator account

Username * ⓘ

myusername

Password * ⓘ

Confirm password * ⓘ

Inbound port rules

Select which virtual machine network ports are accessible from the public internet. You can specify more limited or granular network access on the Networking tab.

Public inbound ports * ⓘ

None Allow selected ports

Select inbound ports

Select one or more ports



All traffic from the internet will be blocked by default. You will be able to change inbound port rules in the VM > Networking page.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Resource group	myResourceGroup	Choose the resource group to contain all the resources for this tutorial. Using the same resource group makes it easier to clean up resources when you're done with this tutorial.

Setting	Suggested value	Description
Virtual machine name	myVM	The VM name needs to be unique in the resource group
Region ↗	(US) North Central US	Choose a region near you or near the functions to be accessed.
Public inbound ports	None	Select None to ensure there is no inbound connectivity to the VM from the internet. Remote access to the VM will be configured via the Azure Bastion service.

5. Choose the *Networking* tab and select **Create new** to configure a new virtual network.

Create a virtual machine

Basics Disks **Networking** Management Advanced Tags Review + create

Define network connectivity for your virtual machine by configuring network interface card (NIC) settings. You can control ports, inbound and outbound connectivity with security group rules, or place behind an existing load balancing solution. [Learn more](#)

Network interface

When creating a virtual machine, a network interface will be created for you.

Virtual network * ⓘ (new) myResourceGroup-vnet [Create new](#)

Subnet * ⓘ (new) Tutorial (10.10.1.0/24)

Public IP ⓘ None [Create new](#)

NIC network security group ⓘ None Basic Advanced

Public inbound ports * ⓘ None Allow selected ports

Select inbound ports Select one or more ports

Accelerated networking ⓘ On Off
The selected VM size does not support accelerated networking.

Load balancing
You can place this virtual machine in the backend pool of an existing Azure load balancing solution. [Learn more](#)

Place this virtual machine behind an existing load balancing solution? Yes No

Review + create < Previous Next : Management >

6. In *Create virtual network*, use the settings in the table below the image:

Create virtual network

The Microsoft Azure Virtual Network service enables Azure resources to securely communicate with each other in a virtual network which is a logical isolation of the Azure cloud dedicated to your subscription. You can connect virtual networks to other virtual networks, or your on-premises network. [Learn more](#)

Name * myResourceGroup-vnet

Address space
The virtual network's address space, specified as one or more address prefixes in CIDR notation (e.g. 192.168.1.0/24).

<input type="checkbox"/> Address range	Addresses	Overlap
<input type="checkbox"/> 10.10.0.0/16	10.10.0.0 - 10.10.255.255 (65536 addresses)	None
	(0 Addresses)	

Subnets
The subnet's address range in CIDR notation. It must be contained by the address space of the virtual network.

<input type="checkbox"/> Subnet name	Address range	Addresses
<input type="checkbox"/> Tutorial	10.10.1.0/24	10.10.1.0 - 10.10.1.255 (256 addresses)
		(0 Addresses)

OK **Discard**

Setting	Suggested value	Description
Name	myResourceGroup-vnet	You can use the default name generated for your virtual network.
Address range	10.10.0.0/16	Use a single address range for the virtual network.
Subnet name	Tutorial	Name of the subnet.
Address range (subnet)	10.10.1.0/24	The subnet size defines how many interfaces can be added to the subnet. This subnet is used by the VM. A /24 subnet provides 254 host addresses.

7. Select **OK** to create the virtual network.
8. Back in the *Networking* tab, ensure **None** is selected for *Public IP*.
9. Choose the *Management* tab, then in *Diagnostic storage account*, choose **Create new** to create a new Storage account.
10. Leave the default values for the *Identity*, *Auto-shutdown*, and *Backup* sections.
11. Select *Review + create*. After validation completes, select **Create**. The VM create process takes a few minutes.

Configure Azure Bastion

Azure Bastion [↗](#) is a fully managed Azure service which provides secure RDP and SSH access to virtual machines directly from the Azure portal. Using the Azure Bastion service removes the need to configure network settings related to RDP access.

1. In the portal, choose **Add** at the top of the resource group view.
2. In the search field, type **Bastion**.
3. Select **Bastion** in the search results.
4. Select **Create** to begin the process of creating a new Azure Bastion resource. You will notice an error message in the *Virtual network* section as there is not yet an AzureBastionSubnet subnet. The subnet is created in the following steps. Use the settings in the table below the image:

Create a Bastion

Basics Tags Review + create

Bastion allows web based RDP access to your vnet VM. [Learn more.](#)

Project details

Subscription * Visual Studio Ultimate

Resource group * myResourceGroup [Create new](#)

Instance details

Name * myBastion

Region * North Central US

Configure virtual networks

Virtual network * myResourceGroup-vnet [Create new](#)

To associate a virtual network with a Bastion, it must contain a subnet with name AzureBastionSubnet with prefix of at least /27.

Subnet * Filter subnets [Manage subnet configuration](#)

Public IP address

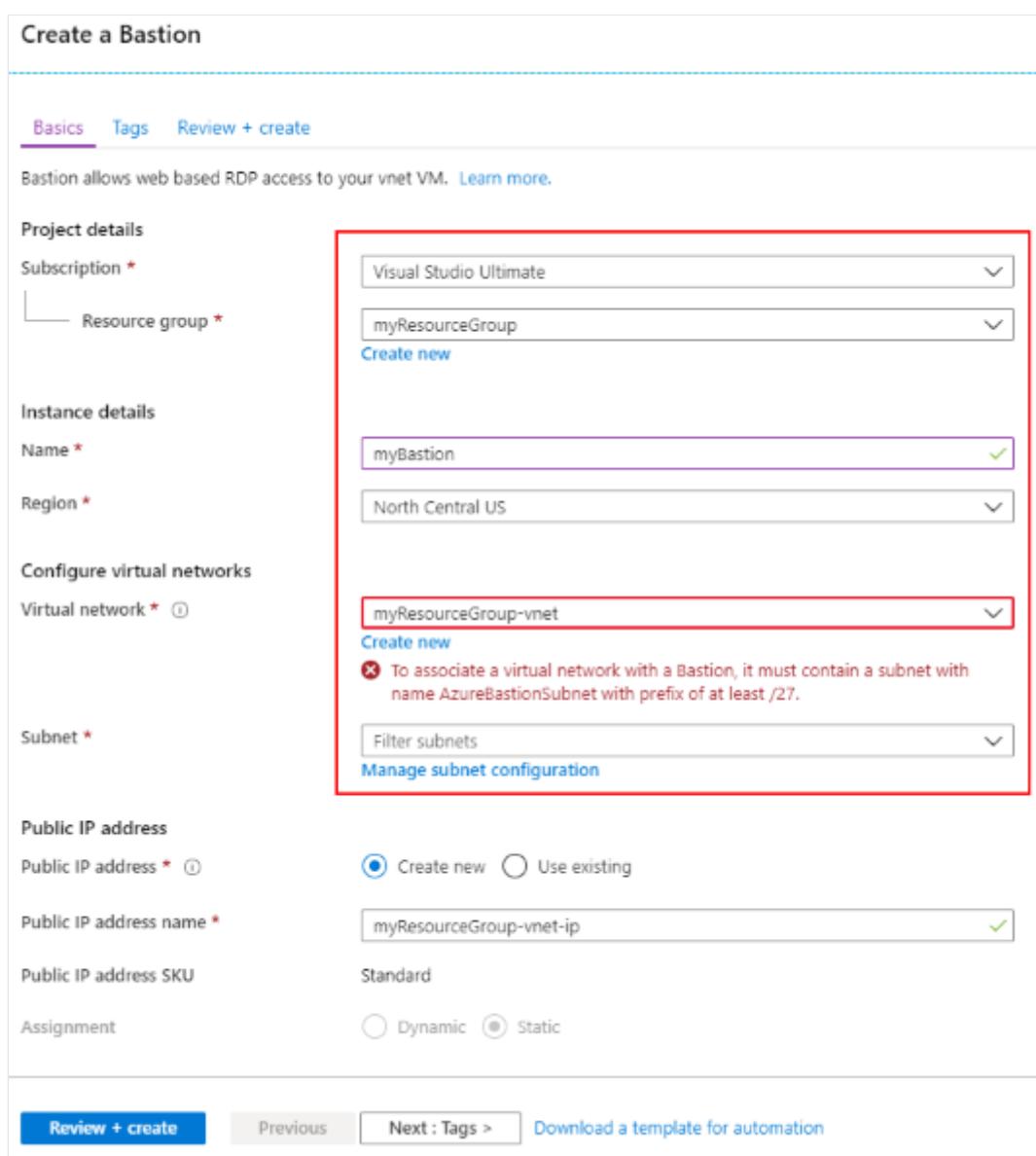
Public IP address * Create new Use existing

Public IP address name * myResourceGroup-vnet-ip

Public IP address SKU Standard

Assignment Dynamic Static

Review + create Previous Next : Tags > Download a template for automation



Setting	Suggested value	Description
Name	myBastion	The name of the new Bastion resource

Setting	Suggested value	Description
<i>Region</i>	North Central US	Choose a region near you or near other services your functions access.
<i>Virtual network</i>	myResourceGroup-vnet	The virtual network in which the Bastion resource will be created in
<i>Subnet</i>	AzureBastionSubnet	The subnet in your virtual network to which the new Bastion host resource will be deployed. You must create a subnet using the name value AzureBastionSubnet . This value lets Azure know which subnet to deploy the Bastion resources to. You must use a subnet of at least /27 or larger (/27, /26, and so on).

 **Note**

For a detailed, step-by-step guide to creating an Azure Bastion resource, refer to the [Create an Azure Bastion host](#) tutorial.

5. Create a subnet in which Azure can provision the Azure Bastion host. Choosing **Manage subnet configuration** opens a new pane where you can define a new subnet. Choose **+ Subnet** to create a new subnet.
6. The subnet must be of the name **AzureBastionSubnet** and the subnet prefix must be at least **/27**. Select **OK** to create the subnet.

Add subnet

myResourceGroup-vnet

Name * ✓

Address range (CIDR block) * ⓘ ✓
10.10.0.0 - 10.10.0.31 (27 + 5 Azure reserved addresses)

NAT gateway ⓘ

Add IPv6 address space

Network security group

Route table

Service endpoints

Services ⓘ

Subnet delegation

Delegate subnet to a service ⓘ

OK

7. On the *Create a Bastion* page, select the newly created **AzureBastionSubnet** from the list of available subnets.

Home > Resource groups > myResourceGroup > New > Bastion > Create a Bastion

Create a Bastion

[Basics](#) [Tags](#) [Review + create](#)

Bastion allows web based RDP access to your vnet VM. [Learn more.](#)

Project details

Subscription * Visual Studio Ultimate

Resource group * myResourceGroup [Create new](#)

Instance details

Name * myBastion

Region * North Central US

Configure virtual networks

Virtual network * myResourceGroup-vnet [Create new](#)

Subnet * AzureBastionSubnet (10.10.0.0/27) [Manage subnet configuration](#)

Public IP address

Public IP address * Create new Use existing

Public IP address name * myResourceGroup-vnet-ip

Public IP address SKU Standard

Assignment Dynamic Static

[Review + create](#) [Previous](#) [Next : Tags >](#) [Download a template for automation](#)

8. Select **Review & Create**. Once validation completes, select **Create**. It will take a few minutes for the Azure Bastion resource to be created.

Create an Azure Functions app

The next step is to create a function app in Azure using the [Consumption plan](#). You deploy your function code to this resource later in the tutorial.

1. In the portal, choose **Add** at the top of the resource group view.
2. Select **Compute > Function App**
3. On the *Basics* section, use the function app settings as specified in the table below.

Setting	Suggested value	Description
---------	-----------------	-------------

Setting	Suggested value	Description
<i>Resource Group</i>	myResourceGroup	Choose the resource group to contain all the resources for this tutorial. Using the same resource group for the function app and VM makes it easier to clean up resources when you're done with this tutorial.
<i>Function App name</i>	Globally unique name	Name that identifies your new function app. Valid characters are a-z (case insensitive), 0-9, and -.
<i>Publish</i>	Code	Option to publish code files or a Docker container.
<i>Runtime stack</i>	Preferred language	Choose a runtime that supports your favorite function programming language.
<i>Region</i>	North Central US	Choose a region near you or near other services your functions access.

Select the **Next: Hosting >** button.

4. For the *Hosting* section, select the proper *Storage account*, *Operating system*, and *Plan* as described in the following table.

Setting	Suggested value	Description
<i>Storage account</i>	Globally unique name	Create a storage account used by your function app. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only. You can also use an existing account, which must meet the storage account requirements .
<i>Operating system</i>	Preferred operating system	An operating system is pre-selected for you based on your runtime stack selection, but you can change the setting if necessary.
<i>Plan</i>	Consumption	The hosting plan dictates how the function app is scaled and resources available to each instance.

5. Select **Review + Create** to review the app configuration selections.

6. Select **Create** to provision and deploy the function app.

Configure access restrictions

The next step is to configure [access restrictions](#) to ensure only resources on the virtual network can invoke the function.

Private site access is enabled by creating an Azure Virtual Network service endpoint between the function app and the specified virtual network. Access restrictions are implemented via service endpoints. Service endpoints ensure only traffic originating from within the specified virtual network can access the designated resource. In this case, the designated resource is the Azure Function.

1. Within the function app, select the **Networking** link under the *Settings* section header.
2. The *Networking* page is the starting point to configure Azure Front Door, the Azure CDN, and also Access Restrictions.
3. Select **Configure Access Restrictions** to configure private site access.
4. On the *Access Restrictions* page, you see only the default restriction in place. The default doesn't place any restrictions on access to the function app. Select **Add rule** to create a private site access restriction configuration.
5. In the *Add Access Restriction* pane, provide a *Name*, *Priority*, and *Description* for the new rule.
6. Select **Virtual Network** from the *Type* drop-down box, then select the previously created virtual network, and then select the **Tutorial** subnet.

ⓘ Note

It may take several minutes to enable the service endpoint.

7. The *Access Restrictions* page now shows that there is a new restriction. It may take a few seconds for the *Endpoint status* to change from Disabled through Provisioning to Enabled.

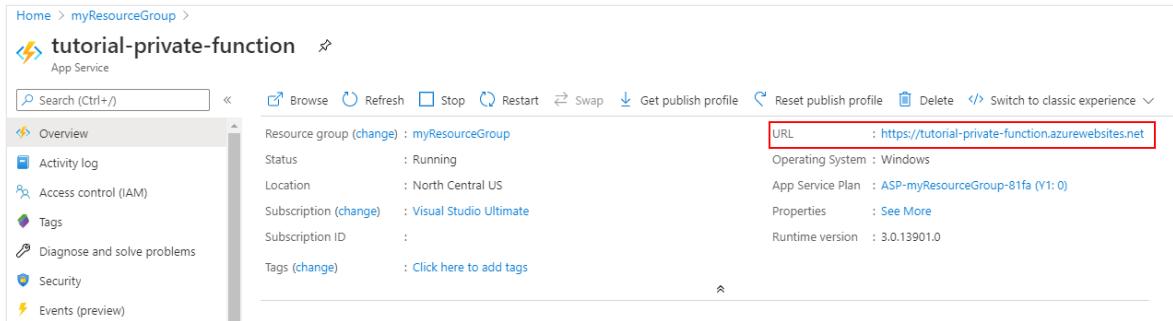
ⓘ Important

Each function app has an **Advanced Tool (Kudu) site** that is used to manage function app deployments. This site is accessed from a URL like:

`<FUNCTION_APP_NAME>.scm.azurewebsites.net`. Enabling access restrictions on the Kudu site prevents the deployment of the project code from a local developer workstation, and then an agent is needed within the virtual network to perform the deployment.

Access the functions app

1. Return to the previously created function app. In the *Overview* section, copy the URL.



The screenshot shows the Azure portal's overview page for an app service named 'tutorial-private-function'. The URL is highlighted with a red box: `https://tutorial-private-function.azurewebsites.net`. Other details shown include the resource group 'myResourceGroup', status 'Running', location 'North Central US', subscription 'Visual Studio Ultimate', and runtime version '3.0.13901.0'.

If you try to access the function app now from your computer outside of your virtual network, you'll receive an HTTP 403 page indicating that access is forbidden.

2. Return to the resource group and select the previously created virtual machine. In order to access the site from the VM, you need to connect to the VM via the Azure Bastion service.
3. Select **Connect** and then choose **Bastion**.
4. Provide the required username and password to log into the virtual machine.
5. Select **Connect**. A new browser window will pop up to allow you to interact with the virtual machine. It's possible to access the site from the web browser on the VM because the VM is accessing the site through the virtual network. While the site is only accessible from within the designated virtual network, a public DNS entry remains.

Create a function

The next step in this tutorial is to create an HTTP-triggered Azure Function. Invoking the function via an HTTP GET or POST should result in a response of "Hello, {name}".

1. Follow one of the following quickstarts to create and deploy your Azure Functions app.
 - [Visual Studio Code](#)
 - [Visual Studio](#)
 - [Command line](#)
 - [Maven \(Java\)](#)

- When publishing your Azure Functions project, choose the function app resource that you created earlier in this tutorial.
- Verify the function is deployed.

The screenshot shows the Azure portal's Functions blade. On the left, there's a sidebar with links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events (preview), Functions (which is selected and highlighted with a red box), App keys, App files, and Proxies. The main area lists functions: 'HttpTriggerCSharp1' (highlighted with a red box). It shows the trigger as 'HTTP' and the status as 'Enabled'. There are buttons for '+ Add', 'Develop Locally', 'Refresh', 'Enable', 'Disable', and 'Delete'.

Invoke the function directly

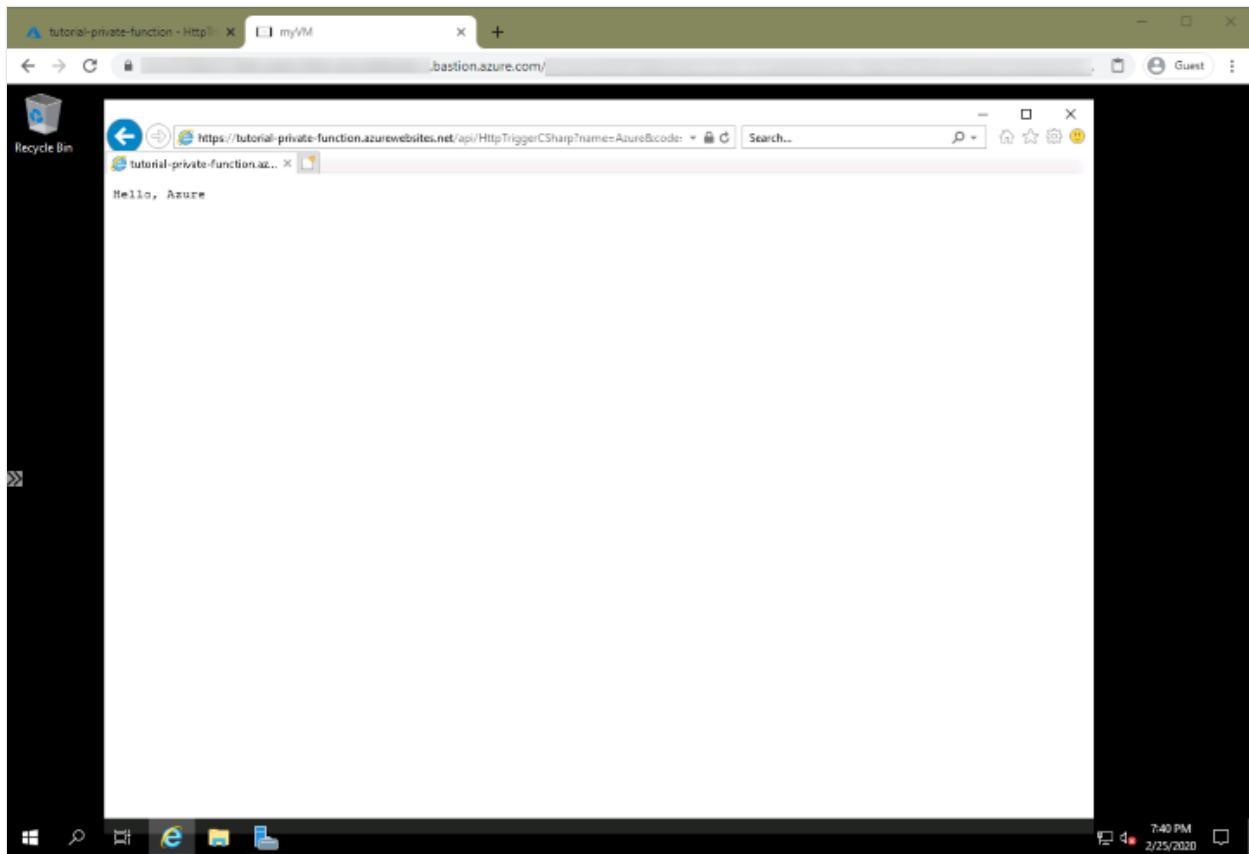
- In order to test access to the function, you need to copy the function URL. Select the deployed function, and then select **Get Function Url**. Then click the **Copy** button to copy the URL to your clipboard.

The screenshot shows the 'HttpTriggerCSharp1' function details page. In the top right, there's a 'Get Function Url' button (highlighted with a red box). Below it, a 'Get Function Url' dialog box is open, showing a dropdown menu set to 'default (function key)' and a long URL in the text input field. A red box highlights the URL in the input field. At the bottom of the dialog is an 'OK' button.

- Paste the URL into a web browser. When you now try to access the function app from a computer outside of your virtual network, you receive an HTTP 403 response indicating access to the app is forbidden.

Invoke the function from the virtual network

Accessing the function via a web browser (by using the Azure Bastion service) on the configured VM on the virtual network results in success!



Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

[Learn more about the networking options in Functions](#)

Tutorial: Control Azure Functions outbound IP with an Azure virtual network NAT gateway

Article • 01/27/2023

Virtual network address translation (NAT) simplifies outbound-only internet connectivity for virtual networks. When configured on a subnet, all outbound connectivity uses your specified static public IP addresses. An NAT can be useful for apps that need to consume a third-party service that uses an allowlist of IP address as a security measure. To learn more, see [What is Azure NAT Gateway?](#).

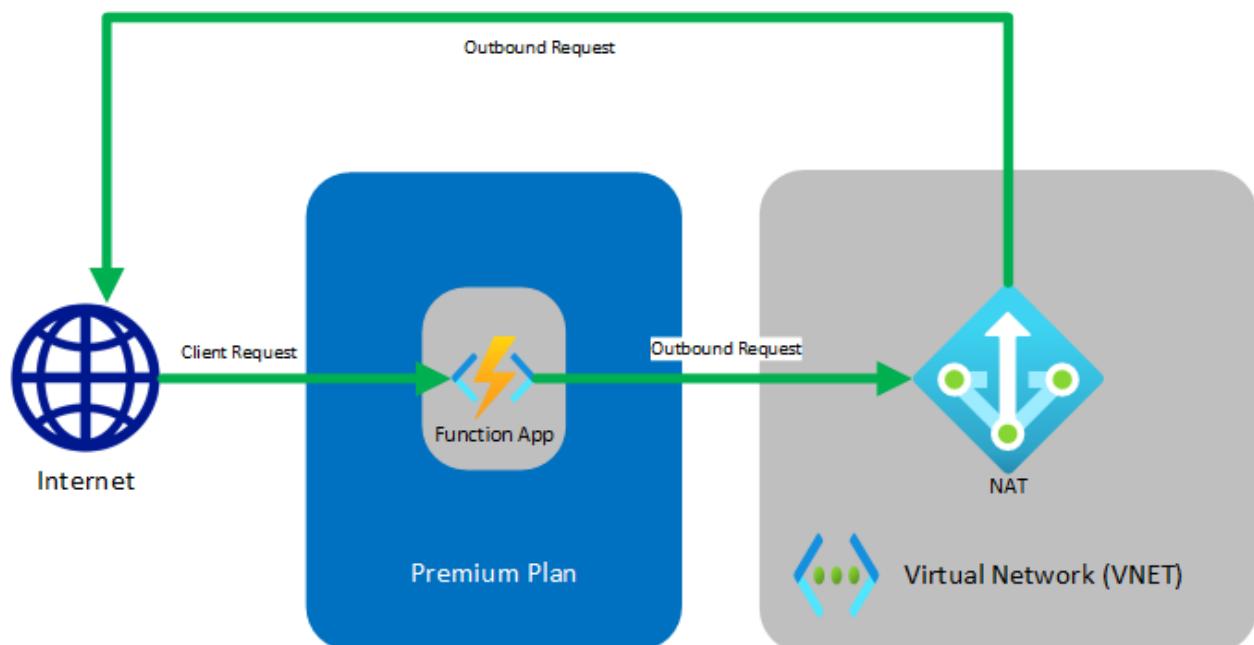
This tutorial shows you how to use NAT gateways to route outbound traffic from an HTTP triggered function. This function lets you check its own outbound IP address.

During this tutorial, you'll:

- ✓ Create a virtual network
- ✓ Create a Premium plan function app
- ✓ Create a public IP address
- ✓ Create a NAT gateway
- ✓ Configure function app to route outbound traffic through the NAT gateway

Topology

The following diagram shows the architecture of the solution that you create:



Functions running in the Premium plan have the same hosting capabilities as web apps in Azure App Service, which includes the VNet Integration feature. To learn more about VNet Integration, including troubleshooting and advanced configuration, see [Integrate your app with an Azure virtual network](#).

Prerequisites

For this tutorial, it's important that you understand IP addressing and subnetting. You can start with [this article that covers the basics of addressing and subnetting](#). Many more articles and videos are available online.

If you don't have an Azure subscription, create a [free account](#) before you begin.

If you've already completed the [integrate Functions with an Azure virtual network](#) tutorial, you can skip to [Create an HTTP trigger function](#).

Create a virtual network

1. From the Azure portal menu, select **Create a resource**. From the Azure Marketplace, select **Networking > Virtual network**.
2. In **Create virtual network**, enter or select the settings specified as shown in the following table:

Setting	Value
Subscription	Select your subscription.
Resource group	Select Create new , enter <i>myResourceGroup</i> , then select OK .
Name	Enter <i>myResourceGroup-vnet</i> .
Location	Select East US .

3. Select **Next: IP Addresses**, and for **IPv4 address space**, enter **10.10.0.0/16**.
4. Select **Add subnet**, then enter *Tutorial-Net* for **Subnet name** and **10.10.1.0/24** for **Subnet address range**.

Home > New > Virtual Network >

Create virtual network

Basics IP Addresses Security Tags Review + create

The virtual network's address space, specified as one or more address prefixes in CIDR notation (e.g. 192.168.1.0/24).

IPv4 address space

✓ Delete

Add IPv6 address space ⓘ

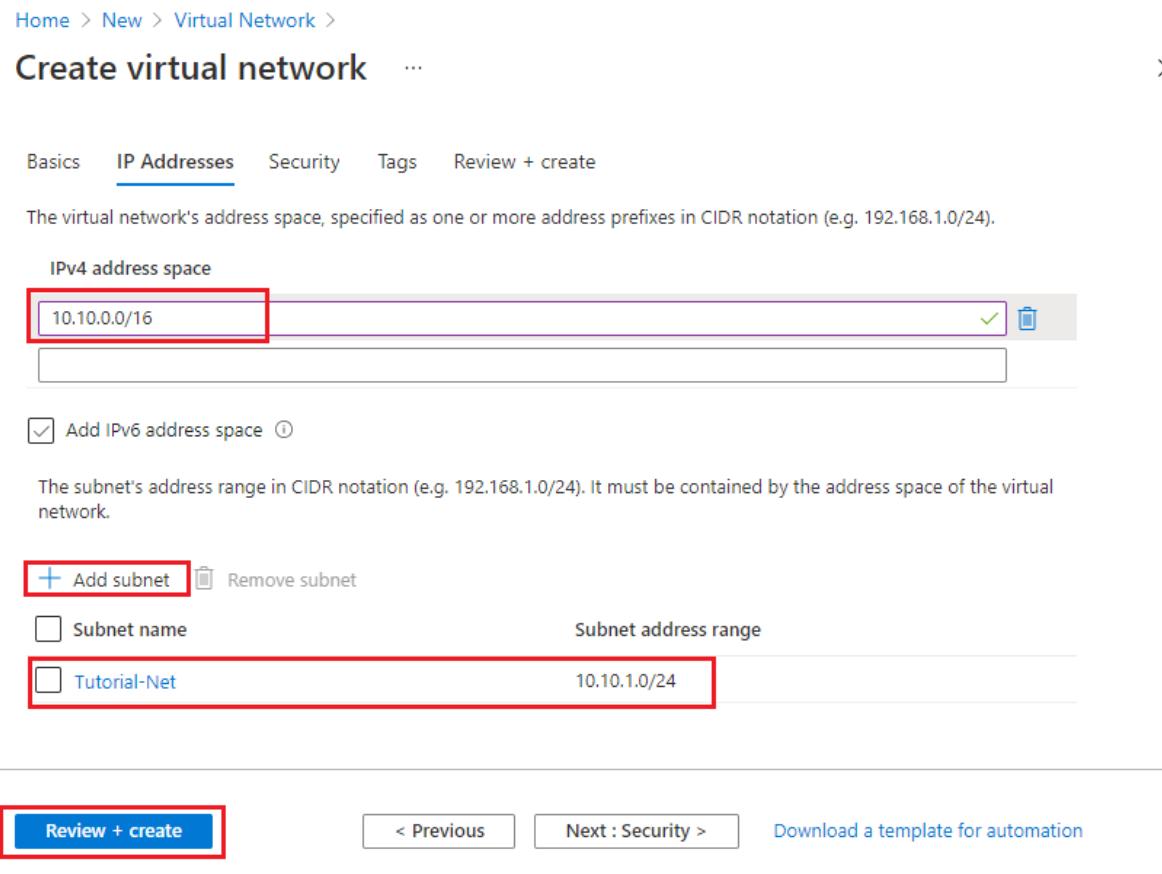
The subnet's address range in CIDR notation (e.g. 192.168.1.0/24). It must be contained by the address space of the virtual network.

+ Add subnet Remove subnet

Subnet name Subnet address range

10.10.1.0/24

Review + create < Previous Next : Security > Download a template for automation



5. Select **Add**, then select **Review + create**. Leave the rest as default and select **Create**.

6. In **Create virtual network**, select **Create**.

Next, you create a function app in the [Premium plan](#). This plan provides serverless scale while supporting virtual network integration.

Create a function app in a Premium plan

This tutorial shows you how to create your function app in a [Premium plan](#). The same functionality is also available when using a [Dedicated \(App Service\) plan](#).

Note

For the best experience in this tutorial, choose .NET for runtime stack and choose Windows for operating system. Also, create your function app in the same region as your virtual network.

1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.

3. On the **Basics** page, use the function app settings as specified in the following table:

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which this new function app is created.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which to create your function app.
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Publish	Code	Option to publish code files or a Docker container.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. In-portal editing isn't currently supported for Python development .
Region	Preferred region	Choose a region near you or near other services your functions access.

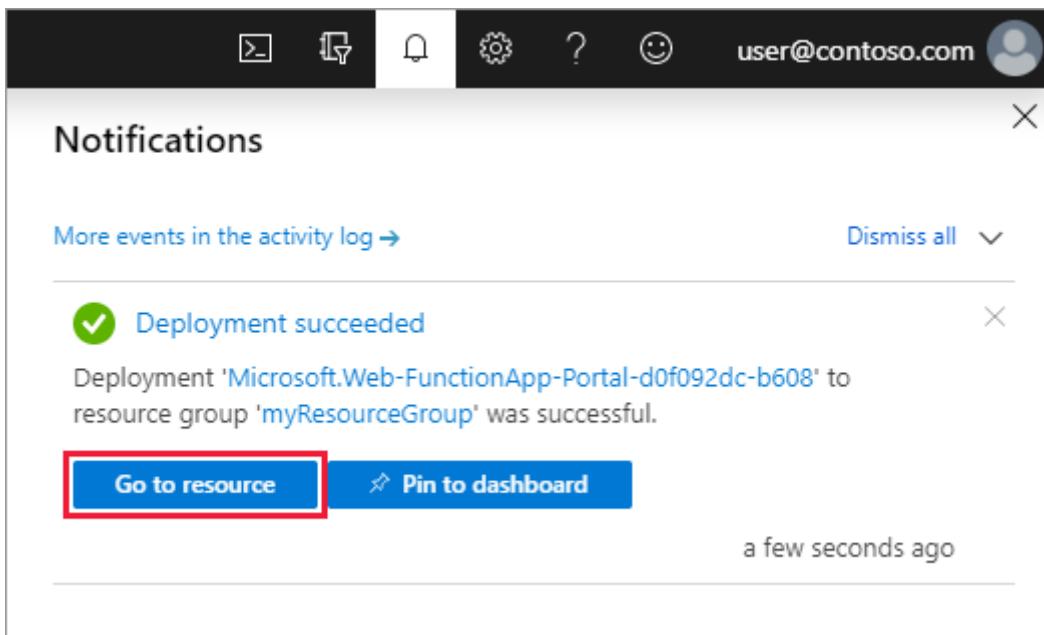
4. Select **Next: Hosting**. On the **Hosting** page, enter the following settings:

Setting	Suggested value	Description
Storage account	Globally unique name	Create a storage account used by your function app. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only. You can also use an existing account, which must meet the storage account requirements .
Operating system	Preferred operating system	An operating system is pre-selected for you based on your runtime stack selection, but you can change the setting if necessary. Python is only supported on Linux. In-portal editing is only supported on Windows.
Plan	Premium	Hosting plan that defines how resources are allocated to your function app. Select Premium . By default, a new App Service plan is created. The default Sku and size is EP1 , where EP stands for <i>elastic premium</i> . To learn more, see the list of Premium SKUs . When running JavaScript functions on a Premium plan, you should choose an instance that has fewer vCPUs. For more information, see Choose single-core Premium plans .

5. Select **Next: Monitoring**. On the **Monitoring** page, enter the following settings:

Setting	Suggested value	Description
Application Insights	Default	Creates an Application Insights resource of the same <i>App name</i> in the nearest supported region. By expanding this setting, you can change the New resource name or choose a different Location in an Azure geography to store your data.

6. Select **Review + create** to review the app configuration selections.
7. On the **Review + create** page, review your settings, and then select **Create** to provision and deploy the function app.
8. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
9. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.



Connect your function app to the virtual network

You can now connect your function app to the virtual network.

1. In your function app, select **Networking** in the left menu, then under **VNet Integration**, select **Click here to configure**.

function-vnet | Networking

Function App

Search (Ctrl+ /)

Settings

- Configuration
- Authentication / Authorization
- Authentication (preview)
- Application Insights
- Identity
- Backups
- Custom domains
- TLS/SSL settings
- Networking**
- Scale up (App Service plan)

VNet Integration

Securely access resources available in or through your Azure VNet.
[Learn More](#)

Click here to configure

Private Endpoint connections

Private access to services hosted on the Azure platform, keeping your data secure.
[Learn More](#)

[Configure your private endpoint connections](#)

Hybrid connections

Securely access applications in private networks
[Learn More](#)

2. On the **VNET Integration** page, select **Add VNet**.

3. In **Network Feature Status**, use the settings in the table below the image:

Network Feature Status

X

vnet-function1

i This VNet Integration experience is in preview.

Virtual Network

myResourceGroup-vnet (northeurope)

Subnet

Create New Subnet Select Existing

* Subnet Name

Function-Net

Virtual Network Address Block

10.10.0.0/16

* Subnet Address Block

10.10.2.0/24

SUBNET NAME

ADDRESS RANGE

Tutorial-Net

10.10.1.0 - 10.10.1.255

OK

Setting	Suggested value	Description
Virtual Network	MyResourceGroup-vnet	This virtual network is the one you created earlier.
Subnet	Create New Subnet	Create a subnet in the virtual network for your function app to use. VNet Integration must be configured to use an empty subnet.
Subnet name	Function-Net	Name of the new subnet.
Virtual network address block	10.10.0.0/16	You should only have one address block defined.
Subnet Address Block	10.10.2.0/24	The subnet size restricts the total number of instances that your Premium plan function app can scale out to. This example uses a /24 subnet with 254 available host addresses. This subnet is over-provisioned, but easy to calculate.

4. Select **OK** to add the subnet. Close the **VNet Integration** and **Network Feature Status** pages to return to your function app page.

The function app can now access the virtual network. When connectivity is enabled, the `vnetrouteallenabled` site setting is set to `1`. You must have either this site setting or the legacy `WEBSITE_VNET_ROUTE_ALL` application setting set to `1`.

Next, you'll add an HTTP-triggered function to the function app.

Create an HTTP trigger function

1. From the left menu of the **Functions** window, select **Functions**, then select **Add** from the top menu.
2. From the **New Function** window, select **Http trigger** and accept the default name for **New Function**, or enter a new name.
3. In **Code + Test**, replace the template-generated C# script (.csx) code with the following code:

```
C#  
  
#r "Newtonsoft.Json"  
  
using System.Net;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Extensions.Primitives;  
using Newtonsoft.Json;  
  
public static async Task<IActionResult> Run(HttpContext req, ILogger log)  
{  
    log.LogInformation("C# HTTP trigger function processed a  
request.");  
  
    var client = new HttpClient();  
    var response = await client.GetAsync("https://ifconfig.me");  
    var responseMessage = await response.Content.ReadAsStringAsync();  
  
    return new OkObjectResult(responseMessage);  
}
```

This code calls an external website that returns the IP address of the caller, which in this case is this function. This method lets you easily determine the outbound IP address being used by your function app.

Now you're ready to run the function and check the current outbound IPs.

Verify current outbound IPs

Now, you can run the function. But first, check in the portal and see what outbound IPs are being used by the function app.

1. In your function app, select **Properties** and review the **Outbound IP Addresses** field.

The screenshot shows the 'natdemo' Function App Properties page. The 'Properties' section is highlighted with a red box. Under 'Outbound IP addresses', a list of IP addresses is shown: 52.191.224.90, 52.191.224.96, 52.191.224.105, 52.191.224.124, 52.191.224.131, and 52.191.224.176.

2. Now, return to your HTTP trigger function, select **Code + Test** and then **Test/Run**.

The screenshot shows the 'IpCheck' function's 'Code + Test' page. The 'Code + Test' section is highlighted with a red box. The 'Test/Run' button is also highlighted with a red box. The code editor shows a C# file named 'run.csx'.

```
1 #r "Newtonsoft.Json"
2
3 using System.Net;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Extensions.Primitives;
6 using Newtonsoft.Json;
7
8 public static async Task<IActionResult> Run(
```

3. Select **Run** to execute the function, then switch to the **Output**.

Input	Output
HTTP response code	200 OK
HTTP response content	52.191.224.124

4. Verify that IP address in the HTTP response body is one of the values from the outbound IP addresses you viewed earlier.

Now, you can create a public IP and use a NAT gateway to modify this outbound IP address.

Create public IP

1. From your resource group, select **Add**, search the Azure Marketplace for **Public IP address**, and select **Create**. Use the settings in the table below the image:

Create public IP address

IPv4 IPv6 Both

SKU * ⓘ

Standard Basic

Tier

Regional Global

IPv4 IP Address Configuration

Name *

Outbound-IP 

IP address assignment

Dynamic Static

Routing preference ⓘ

Microsoft network Internet

Idle timeout (minutes) * ⓘ

4

DNS name label ⓘ

.eastus.cloudapp.azure.com

Subscription *

[REDACTED]

Resource group *

myResourceGroup

[Create new](#)

Location *

(US) East US

Availability zone * ⓘ

No Zone

Create

[Automation options](#)

Setting	Suggested value
IP Version	IPv4
SKU	Standard
Tier	Regional

Setting	Suggested value
Name	Outbound-IP
Subscription	ensure your subscription is displayed
Resource group	myResourceGroup (or name you assigned to your resource group)
Location	East US (or location you assigned to your other resources)
Availability Zone	No Zone

2. Select **Create** to submit the deployment.
3. Once the deployment completes, navigate to your newly created Public IP Address resource and view the IP Address in the **Overview**.

The screenshot shows the Azure portal interface for a Public IP address named "Outbound-IP". The "Overview" tab is selected, indicated by a red box. The "Essentials" section displays the following details:

- Resource group (change) : myResourceGroup
- Location : East US
- Subscription (change) : kbmicrosoft vs ent dev
- Subscription ID : [redacted]
- SKU : Standard
- Tier : Regional
- IP address : 52.170.234.218 (highlighted with a red box)
- DNS name : -
- Associated to : -

On the left sidebar, other tabs like "Activity log", "Access control (IAM)", and "Tags" are visible, but "Overview" is the active tab.

Create NAT gateway

Now, let's create the NAT gateway. When you start with the [previous virtual networking tutorial](#), `Function-Net` was the suggested subnet name and `MyResourceGroup-vnet` was the suggested virtual network name in that tutorial.

1. From your resource group, select **Add**, search the Azure Marketplace for **NAT gateway**, and select **Create**. Use the settings in the table below the image to populate the **Basics** tab:

Create network address translation (NAT) gateway

[Basics](#) [Outbound IP](#) [Subnet](#) [Tags](#) [Review + create](#)

Azure NAT gateway can be used to translate outbound flows from a virtual network to the public internet.
[Learn more about NAT gateways.](#)

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource group *

[Create new](#)

Instance details

NAT gateway name *

myNatGateway



Region *

(US) East US



Availability zone ⓘ

None



Idle timeout (minutes) * ⓘ

4

4-120

Setting	Suggested value
Subscription	Your subscription
Resource group	myResourceGroup (or name you assigned to your resource group)
NAT gateway name	myNatGateway
Region	East US (or location you assigned to your other resources)
Availability Zone	None

2. Select Next: Outbound IP. In the Public IP addresses field, select the previously created public IP address. Leave Public IP Prefixes unselected.
3. Select Next: Subnet. Select the *myResourceGroup-vnet* resource in the Virtual network field and *Function-Net* subnet.

Create network address translation (NAT) gateway ...

Basics Outbound IP **Subnet** Tags Review + create

Configure which subnets of a virtual network should use this NAT gateway. Subnets with Basic load balancers or virtual machines that are using a Basic public IP are not compatible and cannot be used.

Note: While you do not have to complete this step to create a NAT gateway, the NAT gateway will not be functional until you have added at least one subnet. You can also add and reconfigure which subnets are included after creating the NAT gateway.

Virtual network ⓘ

myResourceGroup-vnet

Create new

Subnets having IPv6 address spaces or associated to other NAT gateways are not included.

Subnet name	Subnet address range
Tutorial-Net	10.10.1.0/24
<input checked="" type="checkbox"/> Function-Net	10.10.2.0/24

Manage subnets >

4. Select **Review + Create** then **Create** to submit the deployment.

Once the deployment completes, the NAT gateway is ready to route traffic from your function app subnet to the Internet.

Verify new outbound IPs

Repeat [the steps earlier](#) to run the function again. You should now see the outbound IP address that you configured in the NAT shown in the function output.

Clean up resources

You created resources to complete this tutorial. You'll be billed for these resources, depending on your [account status](#) and [service pricing](#). To avoid incurring extra costs, delete the resources when you no longer need them.

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.

The screenshot shows the Azure portal interface for an App Service application named "myfunctionapp". The left sidebar has a "Functions" section with options like "Functions", "App keys", "App files", and "Proxies". The main content area is titled "Overview" and shows details about the resource group "myResourceGroup". The "Resource group (change)" section is highlighted with a red box. Key details shown include:

- Status: Running
- Location: Central US
- Subscription: Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags: Click here to add tags
- Metrics (selected)
- Features (8)
- Notifications (0)
- Quickstart

The URL listed is <https://myfunctionapp.azurewebsites.net>.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

[Azure Functions networking options](#)

Tutorial: Create a function app that connects to Azure services using identities instead of secrets

Article • 06/27/2024

This tutorial shows you how to configure a function app using Microsoft Entra identities instead of secrets or connection strings, where possible. Using identities helps you avoid accidentally leaking sensitive secrets and can provide better visibility into how data is accessed. To learn more about identity-based connections, see [configure an identity-based connection](#).

While the procedures shown work generally for all languages, this tutorial currently supports C# class library functions on Windows specifically.

In this tutorial, you learn how to:

- ✓ Create a function app in Azure using an ARM template
- ✓ Enable both system-assigned and user-assigned managed identities on the function app
- ✓ Create role assignments that give permissions to other resources
- ✓ Move secrets that can't be replaced with identities into Azure Key Vault
- ✓ Configure an app to connect to the default host storage using its managed identity

After you complete this tutorial, you should complete the follow-on tutorial that shows how to [use identity-based connections instead of secrets with triggers and bindings].

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- The [.NET 6.0 SDK](#)
- The [Azure Functions Core Tools](#) version 4.x.

Why use identity?

Managing secrets and credentials is a common challenge for teams of all sizes. Secrets need to be secured against theft or accidental disclosure, and they might need to be periodically rotated. Many Azure services allow you to instead use an identity in [Microsoft Entra ID](#) to authenticate clients and check against permissions, which can be

modified and revoked quickly. Doing so allows for greater control over application security with less operational overhead. An identity could be a human user, such as the developer of an application, or a running application in Azure with a [managed identity](#).

Because some services don't support Microsoft Entra authentication, your applications might still require secrets in certain cases. However, these secrets can be stored in [Azure Key Vault](#), which helps simplify the management lifecycle for your secrets. Access to a key vault is also controlled with identities.

By understanding how to use identities instead of secrets when you can, and to use Key Vault when you can't, you reduce risk, decrease operational overhead, and generally improve the security posture for your applications.

Create a function app that uses Key Vault for necessary secrets

Azure Files is an example of a service that doesn't yet support Microsoft Entra authentication for Server Message Block (SMB) file shares. Azure Files is the default file system for Windows deployments on Premium and Consumption plans. While we could [remove Azure Files entirely](#), doing so introduces limitations you might not want. Instead, you move the Azure Files connection string into Azure Key Vault. That way it's centrally managed, with access controlled by the identity.

Create an Azure Key Vault

First you need a key vault to store secrets in. You configure it to use [Azure role-based access control \(RBAC\)](#) for determining who can read secrets from the vault.

1. In the [Azure portal](#), choose **Create a resource (+)**.
2. On the **Create a resource** page, select **Security > Key Vault**.
3. On the **Basics** page, use the following table to configure the key vault.

[] [Expand table](#)

Option	Suggested value	Description
Subscription	Your subscription	Subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group where you create your function app.

Option	Suggested value	Description
Key vault name	Globally unique name	Name that identifies your new key vault. The vault name must only contain alphanumeric characters and dashes and can't start with a number.
Pricing Tier	Standard	Options for billing. Standard is sufficient for this tutorial.
Region	Preferred region	Choose a region near you or near other services that your functions access.

Use the default selections for the "Recovery options" sections.

4. Make a note of the name you used, for use later.
5. Select **Next: Access Policy** to navigate to the **Access Policy** tab.
6. Under **Permission model**, choose **Azure role-based access control**
7. Select **Review + create**. Review the configuration, and then select **Create**.

Set up an identity and permissions for the app

In order to use Azure Key Vault, your app needs to have an identity that can be granted permission to read secrets. This app uses a user-assigned identity so that the permissions can be set up before the app is even created. For more information about managed identities for Azure Functions, see [How to use managed identities in Azure Functions](#).

1. In the [Azure portal](#), choose **Create a resource (+)**.
2. On the **Create a resource** page, select **Identity > User Assigned Managed Identity**.
3. On the **Basics** page, use the following table to configure the identity.

[] Expand table

Option	Suggested value	Description
Subscription	Your subscription	Subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group where you create your function app.

Option	Suggested value	Description
Region	Preferred region	Choose a region near you or near other services that your functions access.
Name	Globally unique name	Name that identifies your new user-assigned identity.

4. Select **Review + create**. Review the configuration, and then select **Create**.
5. When the identity is created, navigate to it in the portal. Select **Properties**, and make note of the **Resource ID** for use later.
6. Select **Azure Role Assignments**, and select **Add role assignment (Preview)**.
7. In the **Add role assignment (Preview)** page, use options as shown in the following table.

[\[+\] Expand table](#)

Option	Suggested value	Description
Scope	Key Vault	Scope is a set of resources that the role assignment applies to. Scope has levels that are inherited at lower levels. For example, if you select a subscription scope, the role assignment applies to all resource groups and resources in the subscription.
Subscription	Your subscription	Subscription under which this new function app is created.
Resource	Your key vault	The key vault you created earlier.
Role	Key Vault Secrets User	A role is a collection of permissions that are being granted. Key Vault Secrets User gives permission for the identity to read secret values from the vault.

8. Select **Save**. It might take a minute or two for the role to show up when you refresh the role assignments list for the identity.

The identity is now able to read secrets stored in the key vault. Later in the tutorial, you add additional role assignments for different purposes.

Generate a template for creating a function app

Because the portal experience for creating a function app doesn't interact with Azure Key Vault, you need to generate and edit an Azure Resource Manager template. You can then use this template to create your function app referencing the Azure Files connection string from your key vault.

ⓘ Important

Don't create the function app until after you edit the ARM template. The Azure Files configuration needs to be set up at app creation time.

1. In the [Azure portal](#), choose **Create a resource (+)**.
2. On the **Create a resource** page, select **Compute > Function App**.
3. On the **Basics** page, use the following table to configure the function app.

[\[+\] Expand table](#)

Option	Suggested value	Description
Subscription	Your subscription	Subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group where you create your function app.
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Publish	Code	Choose to publish code files or a Docker container.
Runtime stack	.NET	This tutorial uses .NET.
Region	Preferred region	Choose a region near you or near other services that your functions access.

4. Select **Review + create**. Your app uses the default values on the **Hosting** and **Monitoring** page. Review the default options, which are included in the ARM template that you generate.
5. Instead of creating your function app here, choose **Download a template for automation**, which is to the right of the **Next** button.
6. In the template page, select **Deploy**, then in the Custom deployment page, select **Edit template**.

The screenshot shows the Azure Resource Manager template editor interface. At the top, there are buttons for 'Download', 'Add to library (preview)', 'Deploy' (which is highlighted with a red box), and 'Visualize template'. Below this, a note says 'Automate deploying resources with Azure Resource Manager templates in a single, coordinated operation. Define resources and configurable input parameters and deploy with script or code. Learn more about template deployment.' There is a checked checkbox for 'Include parameters'. The main area shows a JSON template with sections for 'Template', 'Parameters', and 'Scripts'. The 'Template' section contains code starting with '1 {', '2 "schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#"', '3 "contentVersion": "1.0.0.0"', '4 "parameters": {', '5 "subscriptionId": {', '6 "type": "string"'.

Edit the template

You now edit the template to store the Azure Files connection string in Key Vault and allow your function app to reference it. Make sure that you have the following values from the earlier sections before proceeding:

- The resource ID of the user-assigned identity
- The name of your key vault

ⓘ Note

If you were to create a full template for automation, you would want to include definitions for the identity and role assignment resources, with the appropriate `dependsOn` clauses. This would replace the earlier steps which used the portal.

Consult the [Azure Resource Manager guidance](#) and the documentation for each service.

1. In the editor, find where the `resources` array begins. Before the function app definition, add the following section, which puts the Azure Files connection string into Key Vault. Substitute "VAULT_NAME" with the name of your key vault.

JSON

```
{  
  "type": "Microsoft.KeyVault/vaults/secrets",  
  "apiVersion": "2016-10-01",  
  "name": "VAULT_NAME/azurefilesconnectionstring",  
  "properties": {  
    "value": "  
[concat('DefaultEndpointsProtocol=https;AccountName=',parameters('storageAccountName'),';AccountKey=',listKeys(resourceId('Microsoft.Storage/storageAccounts', parameters('storageAccountName')),'2019-06-01').keys[0].value,';EndpointSuffix=','core.windows.net')]"  
  },  
  "dependsOn": [  
    "[concat('Microsoft.Storage/storageAccounts/',  
    parameters('storageAccountName'))]"
```

```
    ],  
},
```

2. In the definition for the function app resource (which has `type` set to `Microsoft.Web/sites`), add `Microsoft.KeyVault/vaults/VAULT_NAME/secrets/azurefilesconnectionstring` to the `dependsOn` array. Again, substitute "VAULT_NAME" with the name of your key vault. Doing so prevents your app from being created before the secret is defined. The `dependsOn` array should look like the following example:

JSON

```
{  
  "type": "Microsoft.Web/sites",  
  "apiVersion": "2018-11-01",  
  "name": "[parameters('name')]",  
  "location": "[parameters('location')]",  
  "tags": null,  
  "dependsOn": [  
    "microsoft.insights/components/idxxntut",  
  
    "Microsoft.KeyVault/vaults/VAULT_NAME/secrets/azurefilesconnectionstring",  
    "[concat('Microsoft.Web/serverfarms/',  
    parameters('hostingPlanName'))]",  
    "[concat('Microsoft.Storage/storageAccounts/',  
    parameters('storageAccountName'))]"  
  ],  
  // ...  
}
```

3. Add the `identity` block from the following example into the definition for your function app resource. Substitute "IDENTITY_RESOURCE_ID" for the resource ID of your user-assigned identity.

JSON

```
{  
  "apiVersion": "2018-11-01",  
  "name": "[parameters('name')]",  
  "type": "Microsoft.Web/sites",  
  "kind": "functionapp",  
  "location": "[parameters('location')]",  
  "identity": {  
    "type": "SystemAssigned,UserAssigned",  
    "userAssignedIdentities": {  
      "IDENTITY_RESOURCE_ID": {}  
    }  
  },  
},
```

```
    "tags": null,  
    // ...  
}
```

This `identity` block also sets up a system-assigned identity, which you use later in this tutorial.

4. Add the `keyVaultReferenceIdentity` property to the `properties` object for the function app, as in the following example. Substitute "IDENTITY_RESOURCE_ID" for the resource ID of your user-assigned identity.

JSON

```
{  
    // ...  
    "properties": {  
        "name": "[parameters('name')]",  
        "keyVaultReferenceIdentity": "IDENTITY_RESOURCE_ID",  
        // ...  
    }  
}
```

You need this configuration because an app could have multiple user-assigned identities configured. Whenever you want to use a user-assigned identity, you must specify it with an ID. System-assigned identities don't need to be specified this way, because an app can only ever have one. Many features that use managed identity assume they should use the system-assigned one by default.

5. Find the JSON objects that define the `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` application setting, which should look like the following example:

JSON

```
{  
    "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",  
    "value": "  
[concat('DefaultEndpointsProtocol=https;AccountName=',parameters('storageAccountName'),';AccountKey=',listKeys(resourceId('Microsoft.Storage/storageAccounts', parameters('storageAccountName'))), '2019-06-01').keys[0].value,';EndpointSuffix=' , 'core.windows.net')]"  
},
```

6. Replace the `value` field with a reference to the secret as shown in the following example. Substitute "VAULT_NAME" with the name of your key vault.

JSON

```
{  
    "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",  
    "value": "[concat('@Microsoft.KeyVault(SecretUri=',  
reference(resourceId('Microsoft.KeyVault/vaults/secrets', 'VAULT_NAME',  
'azurefilesconnectionstring')).secretUri, ')')]"  
},
```

7. Select **Save** to save the updated ARM template.

Deploy the modified template

1. Make sure that your create options, including **Resource Group**, are still correct and select **Review + create**.
2. After your template validates, make a note of your **Storage Account Name**, since you'll use this account later. Finally, select **Create** to create your Azure resources and deploy your code to the function app.
3. After deployment completes, select **Go to resource group** and then select the new function app.

Congratulations! You've successfully created your function app to reference the Azure Files connection string from Azure Key Vault.

Whenever your app would need to add a reference to a secret, you would just need to define a new application setting pointing to the value stored in Key Vault. For more information, see [Key Vault references for Azure Functions](#).

💡 Tip

The [Application Insights connection string](#) and its included instrumentation key are not considered secrets and can be retrieved from App Insights using [Reader](#) permissions. You do not need to move them into Key Vault, although you certainly can.

Use managed identity for AzureWebJobsStorage

Next, you use the system-assigned identity you configured in the previous steps for the `AzureWebJobsStorage` connection. `AzureWebJobsStorage` is used by the Functions runtime and by several triggers and bindings to coordinate between multiple running instances.

It's required for your function app to operate, and like Azure Files, is configured with a connection string by default when you create a new function app.

Grant the system-assigned identity access to the storage account

Similar to the steps you previously followed with the user-assigned identity and your key vault, you now create a role assignment granting the system-assigned identity access to your storage account.

1. In the [Azure portal](#), navigate to the storage account that was created with your function app earlier.
2. Select **Access Control (IAM)**. This page is where you can view and configure who has access to the resource.
3. Select **Add** and select **add role assignment**.
4. Search for **Storage Blob Data Owner**, select it, and select **Next**
5. On the **Members** tab, under **Assign access to**, choose **Managed Identity**
6. Select **Select members** to open the **Select managed identities** panel.
7. Confirm that the **Subscription** is the one in which you created the resources earlier.
8. In the **Managed identity** selector, choose **Function App** from the **System-assigned managed identity** category. The **Function App** label might have a number in parentheses next to it, indicating the number of apps in the subscription with system-assigned identities.
9. Your app should appear in a list below the input fields. If you don't see it, you can use the **Select** box to filter the results with your app's name.
10. Select your application. It should move down into the **Selected members** section. Choose **Select**.
11. On the **Add role assignment** screen, select **Review + assign**. Review the configuration, and then select **Review + assign**.

Tip

If you intend to use the function app for a blob-triggered function, you will need to repeat these steps for the **Storage Account Contributor** and **Storage Queue Data**

Contributor roles over the account used by AzureWebJobsStorage. To learn more, see [Blob trigger identity-based connections](#).

Edit the AzureWebJobsStorage configuration

Next you update your function app to use its system-assigned identity when it uses the blob service for host storage.

Important

The `AzureWebJobsStorage` configuration is used by some triggers and bindings, and those extensions must be able to use identity-based connections, too. Apps that use blob triggers or event hub triggers may need to update those extensions. Because no functions have been defined for this app, there isn't a concern yet. To learn more about this requirement, see [Connecting to host storage with an identity](#).

Similarly, `AzureWebJobsStorage` is used for deployment artifacts when using server-side build in Linux Consumption. When you enable identity-based connections for `AzureWebJobsStorage` in Linux Consumption, you will need to deploy via [an external deployment package](#).

1. In the [Azure portal](#), navigate to your function app.
2. In your function app, expand **Settings**, and then select **Environment variables**.
3. In the **App settings** tab, select the **AzureWebJobsStorage** app setting, and edit it according to the following table:

 Expand table

Option	Suggested value	Description
Name	<code>AzureWebJobsStorage__accountName</code>	Change the name from <code>AzureWebJobsStorage</code> to the exact name <code>AzureWebJobsStorage__accountName</code> . This setting instructs the host to use the identity instead of searching for a stored secret. The new setting uses a double underscore (<code>__</code>), which is a special character in application settings.

Option	Suggested value	Description
Value	Your account name	Update the name from the connection string to just your StorageAccountName.

This configuration tells the system to use an identity to connect to the resource.

4. Select **Apply**, and then select **Apply** and **Confirm** to save your changes and restart the app function.

You've now removed the storage connection string requirement for AzureWebJobsStorage by configuring your app to instead connect to blobs using managed identities.

 **Note**

The `__accountName` syntax is unique to the AzureWebJobsStorage connection and cannot be used for other storage connections. To learn to define other connections, check the reference for each trigger and binding your app uses.

Next steps

This tutorial showed how to create a function app without storing secrets in its configuration.

Advance to the next tutorial to learn how to use identities in trigger and binding connections.

[Use identity-based connections with triggers and bindings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗

Tutorial: Use identity-based connections instead of secrets with triggers and bindings

Article • 07/02/2024

This tutorial shows you how to configure Azure Functions to connect to Azure Service Bus queues by using managed identities, instead of secrets stored in the function app settings. The tutorial is a continuation of the [Create a function app without default storage secrets in its definition](#) tutorial. To learn more about identity-based connections, see [Configure an identity-based connection..](#)

While the procedures shown work generally for all languages, this tutorial currently supports C# class library functions on Windows specifically.

In this tutorial, you learn how to:

- ✓ Create a Service Bus namespace and queue.
- ✓ Configure your function app with a managed identity.
- ✓ Create a role assignment granting that identity permission to read from the Service Bus queue.
- ✓ Create and deploy a function app with a Service Bus trigger.
- ✓ Verify your identity-based connection to the Service Bus.

Prerequisite

Complete the previous tutorial: [Create a function app with identity-based connections](#).

Create a Service Bus namespace and queue

1. In the [Azure portal](#), choose **Create a resource (+)**.
2. On the **Create a resource** page, search for and select **Service Bus**, and then select **Create**.
3. On the **Basics** page, use the following table to configure the Service Bus namespace settings. Use the default values for the remaining options.

 Expand table

Option	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Resource group	myResourceGroup	The resource group you created with your function app.
Namespace name	Globally unique name	The namespace of your instance from which to trigger your function. Because the namespace is publicly accessible, you must use a name that is globally unique across Azure. The name must also be between 6 and 50 characters in length, contain only alphanumeric characters and dashes, and can't start with a number.
Location	myFunctionRegion	The region where you created your function app.
Pricing tier	Basic	The basic Service Bus tier.

4. Select **Review + create**. After validation finishes, select **Create**.
5. After deployment completes, select **Go to resource**.
6. In your new Service Bus namespace, select **+ Queue** to add a queue.
7. Enter **myinputqueue** as the new queue's name and select **Create**.

Now that you have a queue, you can add a role assignment to the managed identity of your function app.

Configure your Service Bus trigger with a managed identity

To use Service Bus triggers with identity-based connections, you need to add the **Azure Service Bus Data Receiver** role assignment to the managed identity in your function app. This role is required when using managed identities to trigger off of your Service Bus namespace. You can also add your own account to this role, which makes it possible to connect to the Service Bus namespace during local testing.

① Note

Role requirements for using identity-based connections vary depending on the service and how you are connecting to it. Needs vary across triggers, input bindings, and output bindings. For more information about specific role requirements, see the trigger and binding documentation for the service.

1. In your Service Bus namespace that you created, select **Access control (IAM)**. This page is where you can view and configure who has access to the resource.
2. Select **+ Add** and select **Add role assignment**.
3. Search for **Azure Service Bus Data Receiver**, select it, and then select **Next**.
4. On the **Members** tab, under **Assign access to**, choose **Managed Identity**
5. Select **Select members** to open the **Select managed identities** panel.
6. Confirm that the **Subscription** is the one in which you created the resources earlier.
7. In the **Managed identity** selector, choose **Function App** from the **System-assigned managed identity** category. The **Function App** label might have a number in parentheses next to it, indicating the number of apps in the subscription with system-assigned identities.
8. Your app should appear in a list below the input fields. If you don't see it, you can use the **Select** box to filter the results with your app's name.
9. Select your application. It should move down into the **Selected members** section. Select **Select**.
10. Back on the **Add role assignment** screen, select **Review + assign**. Review the configuration, and then select **Review + assign**.

You've granted your function app access to the Service Bus namespace using managed identities.

Connect to the Service Bus in your function app

1. In the portal, search for the function app you created in the [previous tutorial](#), or browse to it in the [Function App](#) page.
2. In your function app, expand **Settings**, and then select **Environment variables**.
3. In the **App settings** tab, select **+ Add** to create a setting. Use the information in the following table to enter the **Name** and **Value** for the new setting:

 Expand table

Name	Value	Description
ServiceBusConnection__fullyQualifiedNamespace	<SERVICE_BUS_NAMESPACE>.servicebus.windows.net	This setting connects your function app to the Service Bus using an identity-based connection instead of secrets.

4. Select **Apply**, and then select **Apply** and **Confirm** to save your changes and restart the app function.

Note

When you use [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator, such as `:` or `/`, in place of the `_` to ensure names are resolved correctly.

For example, `ServiceBusConnection:fullyQualifiedNamespace`.

Now that you've prepared the function app to connect to the Service Bus namespace using a managed identity, you can add a new function that uses a Service Bus trigger to your local project.

Add a Service Bus triggered function

1. Run the `func init` command, as follows, to create a functions project in a folder named `LocalFunctionProj` with the specified runtime:

```
C#  
func init LocalFunctionProj --dotnet
```

2. Navigate to the project folder:

```
Console
```

```
cd LocalFunctionProj
```

3. In the root project folder, run the following command:

```
command
```

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.ServiceBus --version 5.2.0
```

This command replaces the default version of the Service Bus extension package with a version that supports managed identities.

4. Run the following command to add a Service Bus triggered function to the project:

```
C#
```

```
func new --name ServiceBusTrigger --template ServiceBusQueueTrigger
```

This command adds the code for a new Service Bus trigger and a reference to the extension package. You need to add a Service Bus namespace connection setting for this trigger.

5. Open the new *ServiceBusTrigger.cs* project file and replace the `ServiceBusTrigger` class with the following code:

```
C#
```

```
public static class ServiceBusTrigger
{
    [FunctionName("ServiceBusTrigger")]
    public static void Run([ServiceBusTrigger("myinputqueue",
        Connection = "ServiceBusConnection")]string myQueueItem, ILogger log)
    {
        log.LogInformation($"C# ServiceBus queue trigger function processed message:
{myQueueItem}");
    }
}
```

This code sample updates the queue name to `myinputqueue`, which is the same name as you queue you created earlier. It also sets the name of the Service Bus connection to `ServiceBusConnection`. This name is the Service Bus namespace used by the identity-based connection `ServiceBusConnection_fullyQualifiedNamespace` you configured in the portal.

① Note

If you try to run your functions now using `func start`, you'll receive an error. This is because you don't have an identity-based connection defined locally. If you want to run your function locally, set the app setting `ServiceBusConnection_fullyQualifiedNamespace` in `local.settings.json` as you did in [the previous section](#connect-to-the-service-bus-in-your-function-app). In addition, you need to assign the role to your developer identity. For more information, see [local development with identity-based connections](#).

Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `ServiceBusConnection:fullyQualifiedNamespace`.

Publish the updated project

1. Run the following command to locally generate the files needed for the deployment package:

Console

```
dotnet publish --configuration Release
```

2. Browse to the `\bin\Release\netcoreapp3.1\publish` subfolder and create a .zip file from its contents.

3. Publish the .zip file by running the following command, replacing the `FUNCTION_APP_NAME`, `RESOURCE_GROUP_NAME`, and `PATH_TO_ZIP` parameters as appropriate:

Azure CLI

```
az functionapp deploy -n FUNCTION_APP_NAME -g RESOURCE_GROUP_NAME --src-path PATH_TO_ZIP
```

Now that you've updated the function app with the new trigger, you can verify that it works using the identity.

Validate your changes

1. In the portal, search for [Application Insights](#) and select **Application Insights** under Services.
2. In **Application Insights**, browse or search for your named instance.
3. In your instance, select **Live Metrics** under **Investigate**.
4. Keep the previous tab open, and open the Azure portal in a new tab. In your new tab, navigate to your Service Bus namespace, select **Queues** from the left menu.
5. Select your queue named `myinputqueue`.
6. Select **Service Bus Explorer** from the left menu.
7. Send a test message.
8. Select your open **Live Metrics** tab and see the Service Bus queue execution.

Congratulations! You have successfully set up your Service Bus queue trigger with a managed identity.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

In this tutorial, you created a function app with identity-based connections.

Advance to the next article to learn how to manage identity.

[Managed identity in Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) 

Tutorial: Connect a function app to Azure SQL with managed identity and SQL bindings

Article • 10/12/2023

Azure Functions provides a [managed identity](#), which is a turn-key solution for securing access to [Azure SQL Database](#) and other Azure services. Managed identities make your app more secure by eliminating secrets from your app, such as credentials in the connection strings. In this tutorial, you'll add managed identity to an Azure Function that utilizes [Azure SQL bindings](#). A sample Azure Function project with SQL bindings is available in the [ToDo backend example](#).

When you're finished with this tutorial, your Azure Function will connect to Azure SQL database without the need of username and password.

An overview of the steps you'll take:

- ✓ Enable Microsoft Entra authentication to the SQL database
- ✓ Enable Azure Function managed identity
- ✓ Grant SQL Database access to the managed identity
- ✓ Configure Azure Function SQL connection string

Grant database access to Microsoft Entra user

First enable Microsoft Entra authentication to SQL database by assigning a Microsoft Entra user as the Active Directory admin of the server. This user is different from the Microsoft account you used to sign up for your Azure subscription. It must be a user that you created, imported, synced, or invited into Microsoft Entra ID. For more information on allowed Microsoft Entra users, see [Microsoft Entra features and limitations in SQL database](#).

Enabling Microsoft Entra authentication can be completed via the Azure portal, PowerShell, or Azure CLI. Directions for Azure CLI are below and information completing this via Azure portal and PowerShell is available in the [Azure SQL documentation on Microsoft Entra authentication](#).

1. If your Microsoft Entra tenant doesn't have a user yet, create one by following the steps at [Add or delete users using Microsoft Entra ID](#).

2. Find the object ID of the Microsoft Entra user using the `az ad user list` and replace `<user-principal-name>`. The result is saved to a variable.

For Azure CLI 2.37.0 and newer:

Azure CLI

```
azureaduser=$(az ad user list --filter "userPrincipalName eq '<user-principal-name>'" --query [].id --output tsv)
```

For older versions of Azure CLI:

Azure CLI

```
azureaduser=$(az ad user list --filter "userPrincipalName eq '<user-principal-name>'" --query [].objectId --output tsv)
```

💡 Tip

To see the list of all user principal names in Microsoft Entra ID, run `az ad user list --query [].userPrincipalName`.

3. Add this Microsoft Entra user as an Active Directory admin using `az sql server ad-admin create` command in the Cloud Shell. In the following command, replace `<server-name>` with the server name (without the `.database.windows.net` suffix).

Azure CLI

```
az sql server ad-admin create --resource-group myResourceGroup --server-name <server-name> --display-name ADMIN --object-id $azureaduser
```

For more information on adding an Active Directory admin, see [Provision a Microsoft Entra administrator for your server](#)

Enable system-assigned managed identity on Azure Function

In this step we'll add a system-assigned identity to the Azure Function. In later steps, this identity will be given access to the SQL database.

To enable system-assigned managed identity in the Azure portal:

1. Create an Azure Function in the portal as you normally would. Navigate to it in the portal.
2. Scroll down to the Settings group in the left navigation.
3. Select Identity.
4. Within the System assigned tab, switch Status to On. Click Save.

The screenshot shows the Azure Functions portal interface. On the left, there's a sidebar with options like Deployment slots, Deployment Center, Configuration, Authentication, Application Insights, and Identity. The Identity option is highlighted with a red box. The main area shows the 'sqlbindingsmanagedidentity' function app. Under the 'Identity' section, the 'System assigned' tab is selected. It displays a note about system-assigned managed identities being restricted to one per resource and authenticated with Azure AD. Below this, there's a 'Status' button with 'Off' and 'On' options, where 'On' is highlighted with a red box. At the bottom of the identity section, there are 'Object (principal) ID' and 'Permissions' sections, with 'Azure role assignments' being the current tab. At the very bottom of the identity section, there are 'Save', 'Discard', 'Refresh', and 'Got feedback?' buttons, with 'Save' also highlighted with a red box.

For information on enabling system-assigned managed identity through Azure CLI or PowerShell, check out more information on [using managed identities with Azure Functions](#).

💡 Tip

For user-assigned managed identity, switch to the User Assigned tab. Click Add and select a Managed Identity. For more information on creating user-assigned managed identity, see the [Manage user-assigned managed identities](#).

Grant SQL database access to the managed identity

In this step we'll connect to the SQL database with a Microsoft Entra user account and grant the managed identity access to the database.

1. Open your preferred SQL tool and login with a Microsoft Entra user account (such as the Microsoft Entra user we assigned as administrator). This can be accomplished in Cloud Shell with the SQLCMD command.

Bash

```
sqlcmd -S <server-name>.database.windows.net -d <db-name> -U <aad-user-name> -P "<aad-password>" -G -l 30
```

2. In the SQL prompt for the database you want, run the following commands to grant permissions to your function. For example,

SQL

```
CREATE USER [<identity-name>] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [<identity-name>];
ALTER ROLE db_datawriter ADD MEMBER [<identity-name>];
GO
```

<*identity-name*> is the name of the managed identity in Microsoft Entra ID. If the identity is system-assigned, the name is always the same as the name of your Function app.

Configure Azure Function SQL connection string

In the final step we'll configure the Azure Function SQL connection string to use Microsoft Entra managed identity authentication.

The connection string setting name is identified in our Functions code as the binding attribute "ConnectionStringSetting", as seen in the SQL input binding [attributes and annotations](#).

In the application settings of our Function App the SQL connection string setting should be updated to follow this format:

```
Server=demo.database.windows.net; Authentication=Active Directory Managed Identity;
Database=testdb
```

testdb is the name of the database we're connecting to and *demo.database.windows.net* is the name of the server we're connecting to.

Tip

For user-assigned managed identity, use `Server=demo.database.windows.net;`
`Authentication=Active Directory Managed Identity;` User

```
Id=ClientIdOfManagedIdentity; Database=testdb.
```

Next steps

- Read data from a database (Input binding)
- Save data to a database (Output binding)
- Review ToDo API sample with Azure SQL bindings

Tutorial: Trigger Azure Functions on blob containers using an event subscription

Article • 05/21/2024

Previous versions of the Azure Functions Blob Storage trigger poll your storage container for changes. More recent version of the Blob Storage extension (5.x+) instead use an Event Grid event subscription on the container. This event subscription reduces latency by triggering your function instantly as changes occur in the subscribed container.

This article shows how to create a function that runs based on events raised when a blob is added to a container. You use Visual Studio Code for local development and to validate your code before deploying your project to Azure.

- ✓ Create an event-based Blob Storage triggered function in a new project.
- ✓ Validate locally within Visual Studio Code using the Azurite emulator.
- ✓ Create a blob storage container in a new storage account in Azure.
- ✓ Create a function app in the Flex Consumption plan (preview).
- ✓ Create an event subscription to the new blob container.
- ✓ Deploy and validate your function code in Azure.

This article creates a C# app that runs in isolated worker mode, which supports .NET 8.0.

ⓘ Important

This tutorial has you use the [Flex Consumption plan](#), which is currently in preview. The Flex Consumption plan only supports the event-based version of the Blob Storage trigger. You can complete this tutorial using any other [hosting plan](#) for your function app.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [.NET 8.0 SDK](#).
- [Visual Studio Code](#) on one of the [supported platforms](#).

- [C# extension](#) for Visual Studio Code.
- [Azure Functions extension](#) for Visual Studio Code.
- [Azure Storage extension](#) for Visual Studio Code.

ⓘ Note

The Azure Storage extension for Visual Studio Code is currently in preview.

Create a Blob triggered function

When you create a Blob Storage trigger function using Visual Studio Code, you also create a new project. You need to edit the function to consume an event subscription as the source, rather than use the regular polled container.

1. In Visual Studio Code, open your function app.
2. Press F1 to open the command palette, enter `Azure Functions: Create Function...`, and select `Create new project`.
3. For your project workspace, select the directory location. Make sure that you either create a new folder or choose an empty folder for the project workspace.
Don't choose a project folder that's already part of a workspace.
4. At the prompts, provide the following information:

 Expand table

Prompt	Action
Select a language	Select <code>C#</code> .
Select a .NET runtime	Select <code>.NET 8.0 Isolated LTS</code> .
Select a template for your project's first function	Select <code>Azure Blob Storage trigger (using Event Grid)</code> .
Provide a function name	Enter <code>BlobTriggerEventGrid</code> .
Provide a namespace	Enter <code>My.Functions</code> .
Select setting from "local.settings.json"	Select <code>create new local app setting</code> .
Select subscription	Select your subscription.

Prompt	Action
Select a storage account	Use Azurite emulator for local storage.
This is the path within your storage account that the trigger will monitor	Accept the default value <code>samples-workitems</code> .
Select how you would like to open your project	Select <code>Open in current window</code> .

Upgrade the Storage extension

To use the Event Grid-based Blob Storage trigger, you must have at least version 5.x of the Azure Functions Storage extension.

To upgrade your project with the required extension version, in the Terminal window, run this [dotnet add package](#) command:

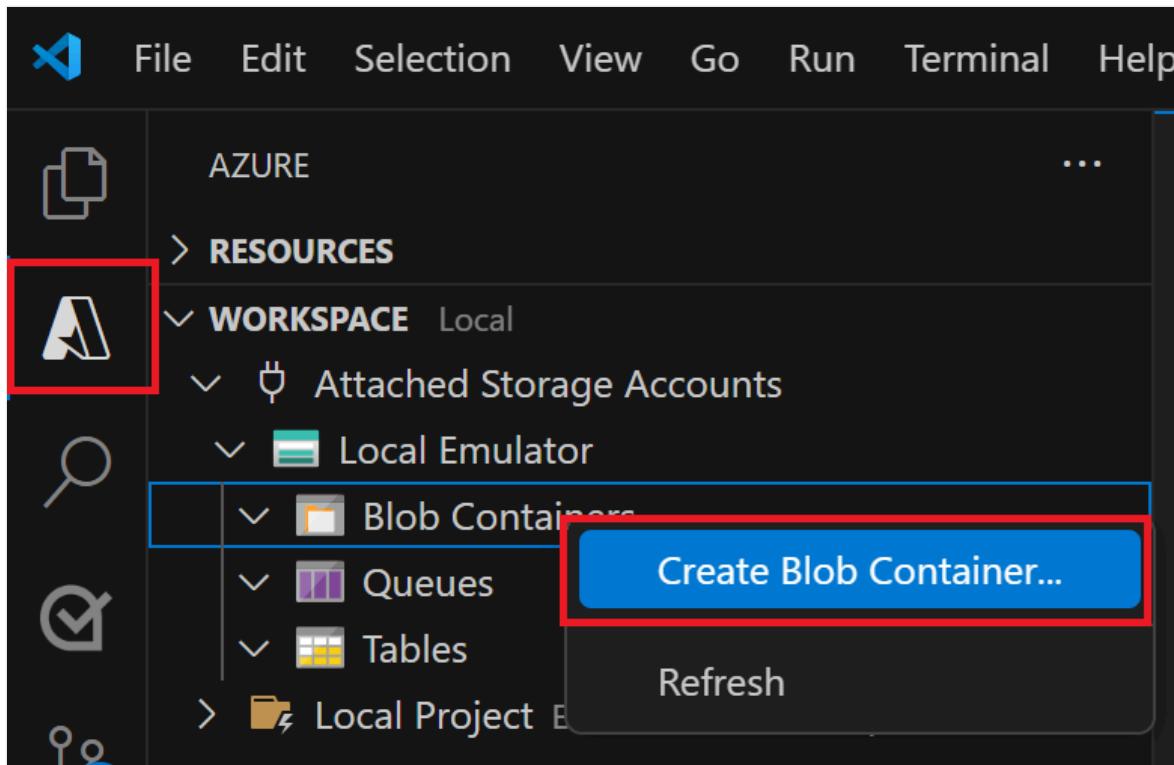
Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs
```

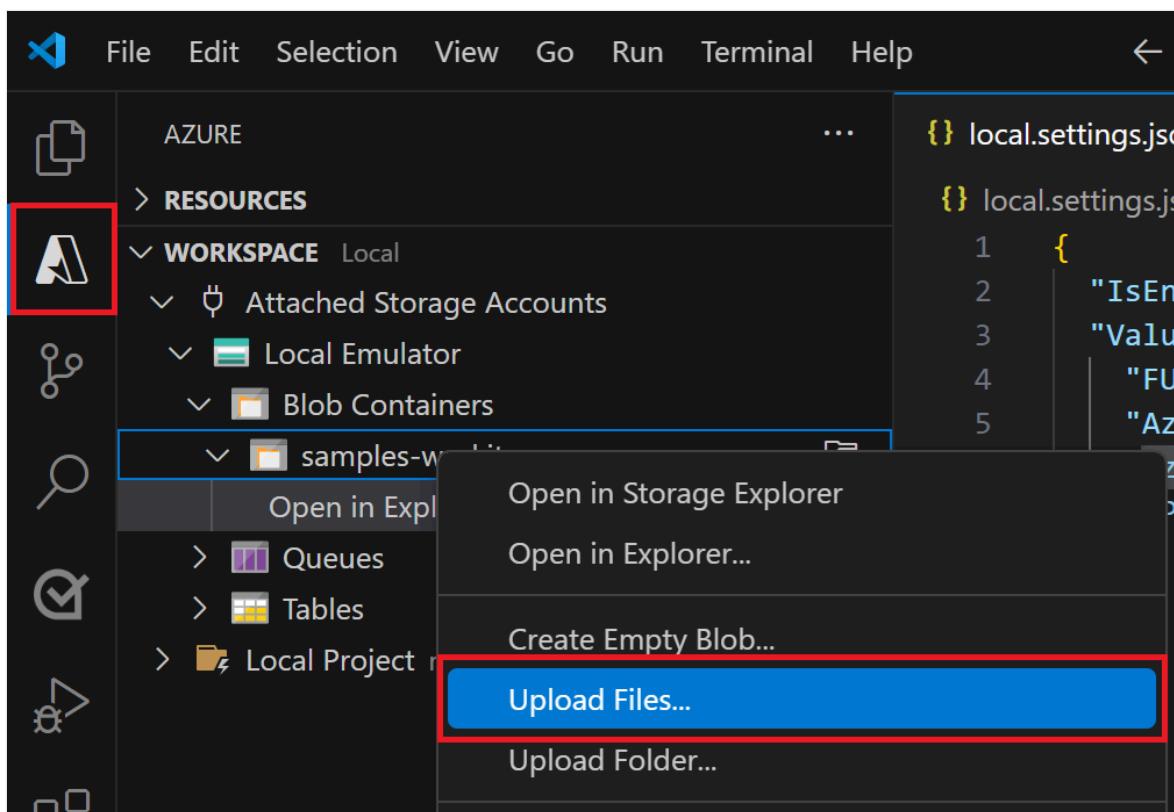
Prepare local storage emulation

Visual Studio Code uses Azurite to emulate Azure Storage services when running locally. You use Azurite to emulate the Azure Blob Storage service during local development and testing.

1. If haven't already done so, install the [Azurite v3 extension for Visual Studio Code](#).
2. Verify that the `local.settings.json` file has `"UseDevelopmentStorage=true"` set for `AzureWebJobsStorage`, which tells Core Tools to use Azurite instead of a real storage account connection when running locally.
3. Press F1 to open the command palette, type `Azurite: Start Blob Service`, and press enter, which starts the Azurite Blob Storage service emulator.
4. Select the Azure icon in the Activity bar, expand **Workspace > Attached Storage Accounts > Local Emulator**, right-click **Blob Containers**, select **Create Blob Container...**, enter the name `samples-workitems`, and press Enter.



5. Expand **Blob Containers** > **samples-workitems** and select **Upload files....**

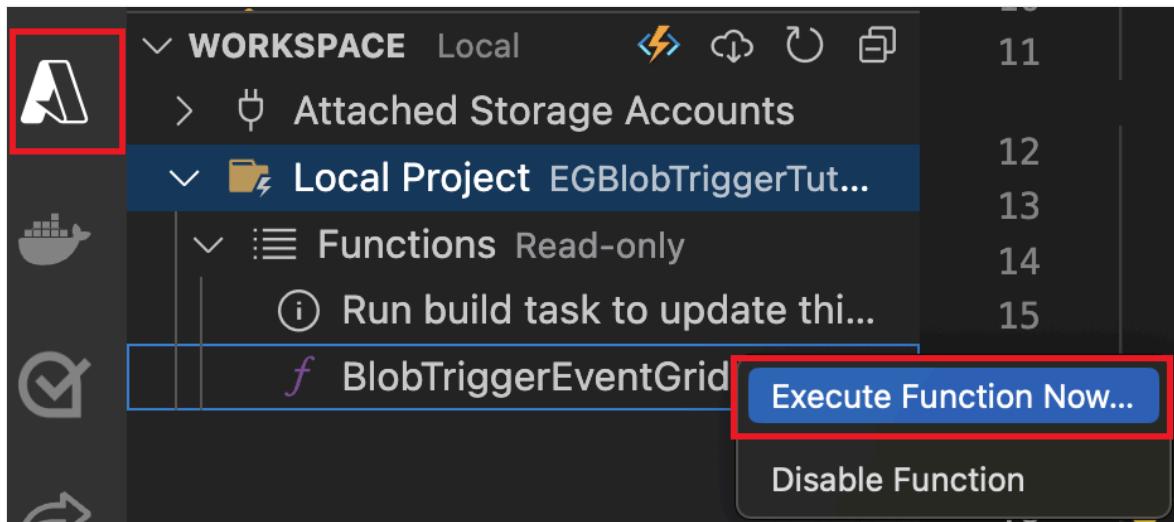


6. Choose a file to upload to the locally emulated container. This file gets processed later by your function to verify and debug your function code. A text file might work best with the Blob trigger template code.

Run the function locally

With a file in emulated storage, you can run your function to simulate an event raised by an Event Grid subscription. The event info passed to your trigger depends on the file you added to the local container.

1. Set any breakpoints and press F5 to start your project for local debugging. Azure Functions Core Tools should be running in your Terminal window.
2. Back in the Azure area, expand **Workspace > Local Project > Functions**, right-click the function, and select **Execute Function Now....**



3. In the request body dialog, type `samples-workitems/<TEST_FILE_NAME>`, replacing `<TEST_FILE_NAME>` with the name of the file you uploaded in the local storage emulator.
4. Press Enter to run the function. The value you provided is the path to your blob in the local emulator. This string gets passed to your trigger in the request payload, which simulates the payload when an event subscription calls your function to report a blob being added to the container.
5. Review the output of this function execution. You should see in the output the name of the file and its contents logged. If you set any breakpoints, you might need to continue the execution.

Now that you've successfully validated your function code locally, it's time to publish the project to a new function app in Azure.

Prepare the Azure Storage account

Event subscriptions to Azure Storage require a general-purpose v2 storage account. You can use the Azure Storage extension for Visual Studio Code to create this storage account.

1. In Visual Studio Code, press F1 again to open the command palette and enter `Azure Storage: Create Storage Account...`. Provide this information when prompted:

[+] Expand table

Prompt	Action
Enter the name of the new storage account	Provide a globally unique name. Storage account names must have 3 to 24 characters in length with only lowercase letters and numbers. For easier identification, we use the same name for the resource group and the function app name.
Select a location for new resources	For better performance, choose a region near you ↗ .

The extension creates a general-purpose v2 storage account with the name you provided. The same name is also used for the resource group that contains the storage account. The Event Grid-based Blob Storage trigger requires a general-purpose v2 storage account.

2. Press F1 again and in the command palette enter `Azure Storage: Create Blob Container...`. Provide this information when prompted:

[+] Expand table

Prompt	Action
Select a resource	Select the general-purpose v2 storage account that you created.
Enter a name for the new blob container	Enter <code>samples-workitems</code> , which is the container name referenced in your code project.

Your function app also needs a storage account to run. For simplicity, this tutorial uses the same storage account for your blob trigger and your function app. However, in production, you might want to use a separate storage account with your function app. For more information, see [Storage considerations for Azure Functions](#).

Create the function app

Use these steps to create a function app in the Flex Consumption plan. When your app is hosted in a Flex Consumption plan, Blob Storage triggers must use event subscriptions.

1. In the command pallet, enter **Azure Functions: Create function app in Azure... (Advanced)**.

2. Following the prompts, provide this information:

 Expand table

Prompt	Selection
Enter a globally unique name for the new function app.	Type a globally unique name that identifies your new function app and then select Enter. Valid characters for a function app name are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Select a hosting plan.	Choose Flex Consumption (Preview) .
Select a runtime stack.	Choose the language stack and version on which you've been running locally.
Select a resource group for new resources.	Choose the existing resource group in which you created the storage account.
Select a location for new resources.	Select a location in a supported region near you or near other services that your functions access. Unsupported regions aren't displayed. For more information, see View currently supported regions .
Select a storage account.	Choose the name of the storage account you created.
Select an Application Insights resource for your app.	Choose Create new Application Insights resource and at the prompt provide the name for the instance used to store runtime data from your functions.

A notification appears after your function app is created. Select **View Output** in this notification to view the creation results, including the Azure resources that you created.

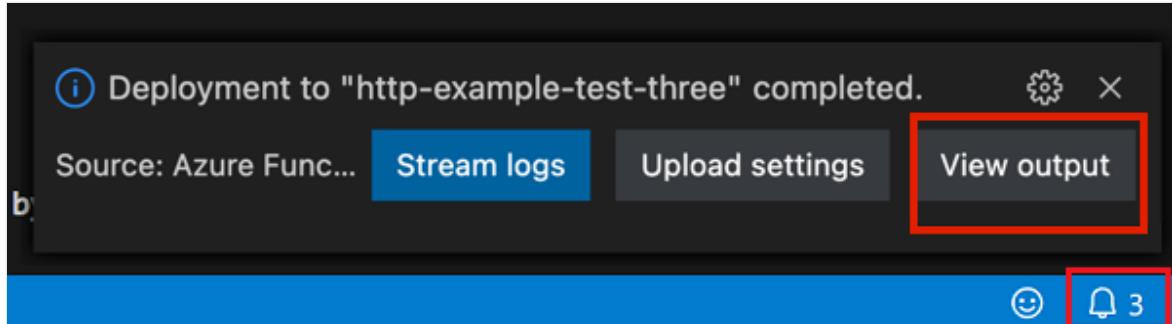
Deploy your function code

Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the **Resources** area of the Azure activity, locate the function app resource you just created, right-click the resource, and select **Deploy to function app....**

- When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
- After deployment completes, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower right corner to see it again.



Update application settings

Because required application settings from the `local.settings.json` file aren't automatically published, you must upload them to your function app so that your function runs correctly in Azure.

- In the command pallet, enter `Azure Functions: Download Remote Settings...`, and in the **Select a resource** prompt choose the name of your function app.
- When prompted that the `AzureWebJobsStorage` setting already exists, select **Yes** to overwrite the local emulator setting with the actual storage account connection string from Azure.
- In the `local.settings.json` file, replace the local emulator setting with same connection string used for `AzureWebJobsStorage`.
- Remove the `FUNCTIONS_WORKER_RUNTIME` entry, which isn't supported in a Flex Consumption plan.
- In the command pallet, enter `Azure Functions: Upload Local Settings...`, and in the **Select a resource** prompt choose the name of your function app.

Now both the Functions host and the trigger are sharing the same storage account.

Build the endpoint URL

To create an event subscription, you need to provide Event Grid with the URL of the specific endpoint to report Blob Storage events. This *blob extension* URL is composed of

these parts:

[+] Expand table

Part	Example
Base function app URL	<code>https://<FUNCTION_APP_NAME>.azurewebsites.net</code>
Blob-specific path	<code>/runtime/webhooks/blobs</code>
Function query string	<code>?functionName=Host.Functions.BlobTriggerEventGrid</code>
Blob extension access key	<code>&code=<BLOB_EXTENSION_KEY></code>

The blob extension access key is designed to make it more difficult for others to access your blob extension endpoint. To determine your blob extension access key:

1. In Visual Studio Code, choose the Azure icon in the Activity bar. In **Resources**, expand your subscription, expand **Function App**, right-click the function app you created, and select **Open in portal**.
2. Under **Functions** in the left menu, select **App keys**.
3. Under **System keys** select the key named **blobs_extension**, and copy the key **Value**. You include this value in the query string of new endpoint URL.
4. Create a new endpoint URL for the Blob Storage trigger based on the following example:

```
HTTP  
  
https://<FUNCTION_APP_NAME>.azurewebsites.net/runtime/webhooks/blobs?  
functionName=Host.Functions.BlobTriggerEventGrid&code=<BLOB_EXTENSION_KEY>
```

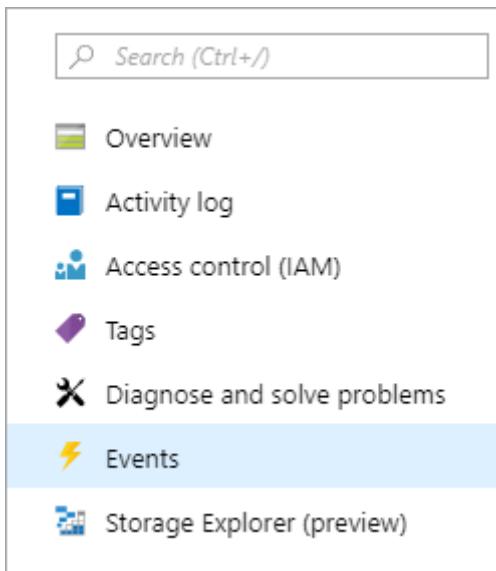
In this example, replace `<FUNCTION_APP_NAME>` with the name of your function app and replace `<BLOB_EXTENSION_KEY>` with the value you got from the portal. If you used a different name for your function, you'll also need to change the `functionName` query string value to your function name.

You can now use this endpoint URL to create an event subscription.

Create the event subscription

An event subscription, powered by Azure Event Grid, raises events based on changes in the subscribed blob container. This event is then sent to the blob extension endpoint for your function. After you create an event subscription, you can't update the endpoint URL.

1. In Visual Studio Code, choose the Azure icon in the Activity bar. In **Resources**, expand your subscription, expand **Storage accounts**, right-click the storage account you created earlier, and select **Open in portal**.
2. Sign in to the [Azure portal](#) and make a note of the **Resource group** for your storage account. You create your other resources in the same group to make it easier to clean up resources when you're done.
3. select the **Events** option from the left menu.



4. In the **Events** window, select the **+ Event Subscription** button, and provide values from the following table into the **Basic** tab:

[\[+\] Expand table](#)

Setting	Suggested value	Description
Name	<code>myBlobEventSub</code>	Name that identifies the event subscription. You can use the name to quickly find the event subscription.
Event Schema	Event Grid Schema	Use the default schema for events.
System Topic Name	<code>samples-workitems-blobs</code>	Name for the topic, which represents the container. The topic is created with the first subscription, and you'll use it for future event subscriptions.

Setting	Suggested value	Description
Filter to Event Types	Blob Created	
Endpoint Type	Web Hook	The blob storage trigger uses a web hook endpoint.
Endpoint	Your Azure-based URL endpoint	Use the URL endpoint that you built, which includes the key value.

5. Select **Confirm selection** to validate the endpoint URL.

6. Select **Create** to create the event subscription.

Upload a file to the container

You can upload a file from your computer to your blob storage container using Visual Studio Code.

1. In Visual Studio Code, press F1 to open the command palette and type `Azure Storage: Upload Files....`.
2. In the **Open** dialog box, choose a file, preferably a text file, and select **Upload**.
3. Provide the following information at the prompts:

[] Expand table

Setting	Suggested value	Description
Enter the destination directory of this upload	default	Just accept the default value of <code>/</code> , which is the container root.
Select a resource	Storage account name	Choose the name of the storage account you created in a previous step.
Select a resource type	Blob Containers	You're uploading to a blob container.
Select Blob Container	samples-workitems	This value is the name of the container you created in a previous step.

Browse your local file system to find a file to upload and then select the **Upload** button to upload the file.

Verify the function in Azure

Now that you uploaded a file to the `samples-workitems` container, the function should be triggered. You can verify by checking the following on the Azure portal:

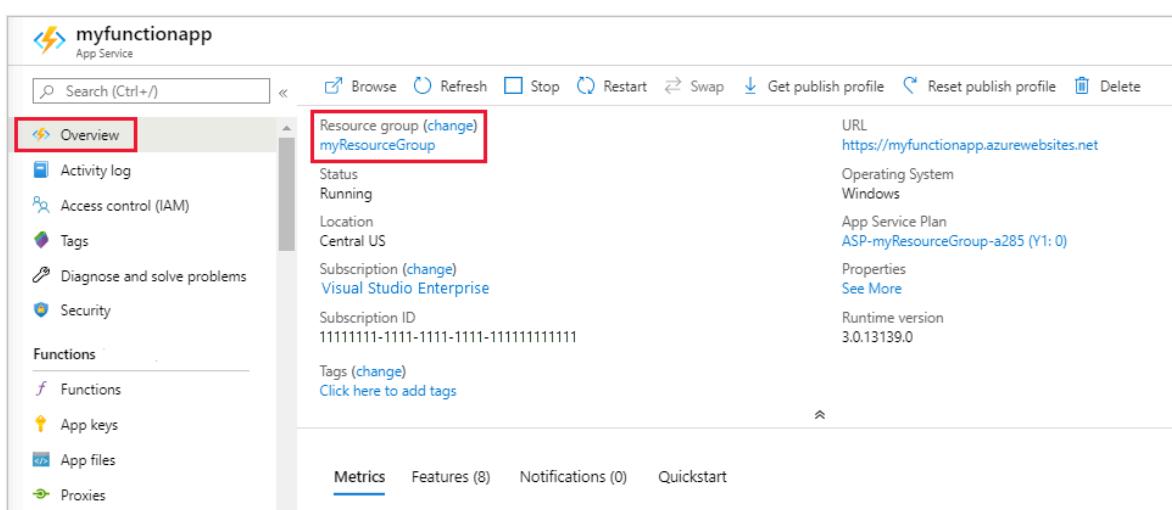
1. In your storage account, go to the **Events** page, select **Event Subscriptions**, and you should see that an event was delivered. There might be up a five-minute delay for the event to show up on the chart.
2. Back in your function app page in the portal, under **Functions** find your function and select **Invocations and more**. You should see traces written from your successful function execution.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

- [Working with blobs](#)
- [Automate resizing uploaded images using Event Grid](#)
- [Event Grid trigger for Azure Functions](#)

Use Azure Event Grid to route Blob storage events to web endpoint (Azure portal)

Article • 11/27/2023

Event Grid is a fully managed service that enables you to easily manage events across many different Azure services and applications. It simplifies building event-driven and serverless applications. For an overview of the service, see [Event Grid overview](#).

In this article, you use the Azure portal to do the following tasks:

1. Create a Blob storage account.
2. Subscribe to events for that blob storage.
3. Trigger an event by uploading a file to the blob storage.
4. View the result in a handler web app. Typically, you send events to an endpoint that processes the event data and takes actions. To keep it simple, you send events to a web app that collects and displays the messages.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

When you're finished, you see that the event data has been sent to the web app.

Create a storage account

1. Sign in to [Azure portal](#).
 2. To create a Blob storage, select **Create a resource**.
 3. In the **Search**, enter **Storage account**, and select **Storage account** from the result list.

Create a resource

Get started

Recently created

Categories

- AI + Machine Learning
- Analytics
- Blockchain
- Compute
- Containers

Storage account

Windows Server 2019 Datacenter

Ubuntu Server 20.04 LTS

Web App

4. On the **Storage account** page, select **Create** to start creating the storage account.

To subscribe to events, create either a general-purpose v2 storage account or a Blob storage account.

5. On the **Create storage account** page, do the following steps:

a. Select your Azure subscription.

b. For **Resource group**, create a new resource group or select an existing one.

c. Enter the name for your storage account.

d. Select the **Region** in which you want the storage account to be created.

e. For **Redundancy**, select **Locally-redundant storage (LRS)** from the drop-down list.

f. Select **Review** at the bottom of the page.

Create a storage account ...

Basics Advanced Networking Data protection Encryption Tags Review

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more about Azure storage accounts](#)

Project details

Select the subscription in which to create the new storage account. Choose a new or existing resource group to organize and manage your storage account together with other resources.

Subscription * Visual Studio Enterprise Subscription

Resource group * (New) spegridrg [Create new](#)

Instance details

If you need to create a legacy storage account type, please click [here](#).

Storage account name ⓘ * spstorageaccount1027

Region ⓘ * (US) East US

Performance ⓘ * Standard: Recommended for most scenarios (general-purpose v2 account)
 Premium: Recommended for scenarios that require low latency.

Redundancy ⓘ * Locally-redundant storage (LRS)

Review < Previous Next : Advanced >

g. On the **Review** page, review the settings, and select **Create**.

① Note

Only storage accounts of kind **StorageV2 (general purpose v2)** and **BlobStorage** support event integration. **Storage (general purpose v1)** does *not* support integration with Event Grid.

6. The deployment takes a few minutes to complete. On the **Deployment** page, select **Go to resource**.

The screenshot shows the 'Overview' tab for a deployment named 'spstorageaccount1027_1666928259897'. The main content area displays a green checkmark icon followed by the text 'Your deployment is complete'. Below this, it lists deployment details: 'Deployment name: spstorageaccount1027_1666928259897', 'Subscription: Visual Studio Enterprise Subscription', and 'Resource group: spegridrg'. There are two expandable sections: 'Deployment details' and 'Next steps'. A prominent blue button labeled 'Go to resource' is centered below the deployment details. At the bottom, there are links for 'Give feedback' and 'Tell us about your experience with deployment'.

7. On the **Storage account** page, select **Events** on the left menu.

The screenshot shows the 'Events' page for a storage account named 'spstorageaccount1027'. The left sidebar has a red box around the 'Events' link under the 'Storage account' section. The main content area features a large heading 'Events, automated.' and a descriptive text about using Event Grid for automation. It includes links for 'Event Subscription', 'Refresh', and 'Feedback'. Below this, there's a section for creating logic apps to respond to events. At the bottom, there's a diagram illustrating the flow of events from a source through a cloud icon to a sink, with binary code (0s and 1s) shown inside the cloud.

8. Keep this page in the web browser open.

Create a message endpoint

Before subscribing to the events for the Blob storage, let's create the endpoint for the event message. Typically, the endpoint takes actions based on the event data. To simplify this quickstart, you deploy a [prebuilt web app](#) that displays the event messages. The deployed solution includes an App Service plan, an App Service web app, and source code from GitHub.

1. Select **Deploy to Azure** to deploy the solution to your subscription.



2. On the **Custom deployment** page, do the following steps:

- a. For **Resource group**, select the resource group that you created when creating the storage account. It will be easier for you to clean up after you're done with the tutorial by deleting the resource group.
- b. For **Site Name**, enter a name for the web app.
- c. For **Hosting plan name**, enter a name for the App Service plan to use for hosting the web app.
- d. Select **Review + create**.

Custom deployment

Deploy from a custom template

Basics Review + create

Template



Customized template ↗

2 resources



Edit template



Edit parameters



Visualize

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Visual Studio Enterprise Subscription

Resource group * ⓘ

contosoegridrg

Create new

Instance details

Region * ⓘ

(US) East US

Site Name * ⓘ

contosoegridviewer

Hosting Plan Name * ⓘ

contosositeplan

Sku ⓘ

F1

Repo URL ⓘ

<https://github.com/Azure-Samples/azure-event-grid-viewer.git>

Branch ⓘ

master

Location ⓘ

[resourceGroup().location]

Previous

Next

Review + create

3. On the **Review + create** page, select **Create**.

4. The deployment takes a few minutes to complete. On the **Deployment** page, select **Go to resource group**.

The screenshot shows the Microsoft Azure Deployment Overview page for a template named "Microsoft.Template-20221027235018". The main message is "Your deployment is complete". Deployment details include the name, subscription, and resource group. There are sections for "Deployment details" and "Next steps", with a prominent blue button labeled "Go to resource group". Below the main content are links to "Give feedback" and "Tell us about your experience with deployment".

5. On the **Resource group** page, in the list of resources, select the web app that you created. You also see the App Service plan and the storage account in this list.

The screenshot shows the Microsoft Azure Resource Group Overview page for a group named "contosoegriddrg". The main area displays a list of resources, including an Event Grid Topic, an App Service, and an App Service plan. The "contosoegridviewer" resource is highlighted with a red box. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Resource visualizer, Events, Settings, Deployments, Security, Deployment stacks, Policies, Properties, Locks, Cost Management, Cost analysis, Cost alerts (preview), Budgets, and Advisor recommendations.

6. On the **App Service** page for your web app, select the URL to navigate to the web site. The URL should be in this format: `https://<your-site-name>.azurewebsites.net`.

The screenshot shows the Azure portal's configuration page for a web app named 'contosoegridviewer'. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Deployment, and Deployment slots. The 'Overview' tab is selected. On the right, under the 'Essentials' section, there are several settings: Resource group (contosoegridrg), Status (Running), Location (East US), Subscription (Visual Studio Enterprise Subscription). To the right of these, there's a 'Default domain' field containing 'contosoegridviewer.azurewebsites.net', which is highlighted with a red rectangular box. Other settings listed include App Service Plan (contosositeplan), Operating System (Windows), and Health Check (Not Configured). A 'JSON View' link is located at the top right of the essentials panel.

7. Confirm that you see the site but no events have been posted to it yet.

The screenshot shows a browser window with the URL 'https://contosoegridviewer.azurewebsites.net'. The page title is 'Azure Event Grid Viewer' and features a blue icon of two overlapping arrows. Below the title, there's a large empty white area where event logs would normally appear. In the bottom-left corner of this area, there's a light blue callout box with the title 'Important' in bold. The text inside the box reads: 'Keep the Azure Event Grid Viewer window open so that you can see events as they are posted.'

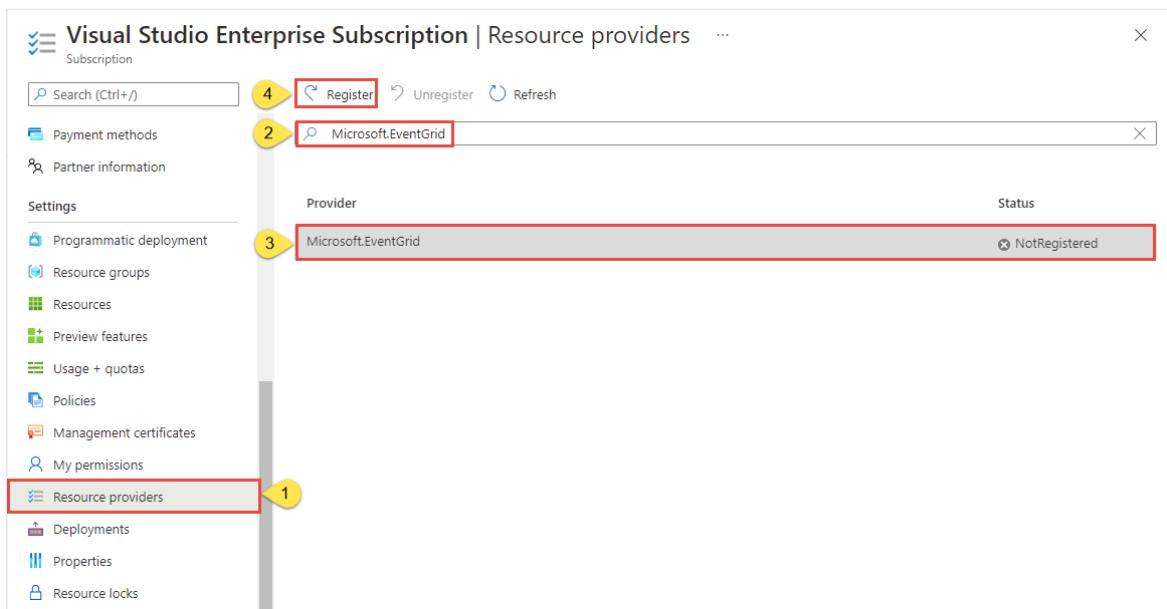
Register the Event Grid resource provider

Unless you've used Event Grid before, you'll need to register the Event Grid resource provider. If you've used Event Grid before, skip to the next section.

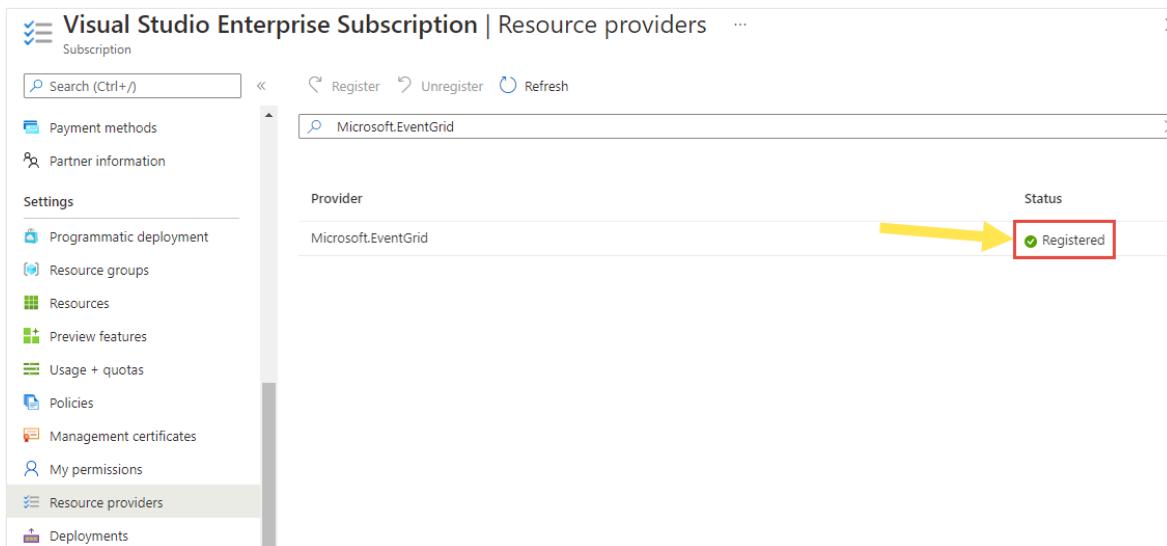
In the Azure portal, do the following steps:

1. On the left menu, select **Subscriptions**.
2. Select the **subscription** you want to use for Event Grid from the subscription list.
3. On the **Subscription** page, select **Resource providers** under **Settings** on the left menu.
4. Search for **Microsoft.EventGrid**, and confirm that the **Status** is **Not Registered**.
5. Select **Microsoft.EventGrid** in the provider list.

6. Select Register on the command bar.



7. Refresh to make sure the status of Microsoft.EventGrid is changed to Registered.



Subscribe to the Blob storage

You subscribe to a topic to tell Event Grid which events you want to track, and where to send the events.

1. If you closed the **Storage account** page, navigate to your Azure Storage account that you created earlier. On the left menu, select **All resources** and select your storage account.
2. On the **Storage account** page, select **Events** on the left menu.
3. Select **More Options**, and **Web Hook**. You're sending events to your viewer app using a web hook for the endpoint.

The screenshot shows the Azure Storage account 'spstorageaccount1027' Events page. The left sidebar contains navigation links: Overview, Activity log, Tags, Diagnose and solve problems, Access Control (IAM), Data migration, **Events** (highlighted with a red box and the number 1), and Storage browser. The main content area features a heading 'Events, automated.' and a brief description of Azure Event Grid. Below this, it says 'Azure Event Grid natively supports these resources as event handlers. Learn more'. It lists four options: Logic Apps, Azure Function, More Options (highlighted with a red box and the number 2), and Web Hook (highlighted with a red box and the number 3). Each option has a description and a small icon.

Event Handler	Description
Logic Apps	Use events as a trigger for executing a Logic App, starting Azure-wide workflows and automation.
Azure Function	Use events as a trigger for executing an Azure Function, which enables serverless custom code execution.
Web Hook	Send events as HTTP pushes to a Web Hook endpoint, both within or outside of Azure.
Storage Queues	Send events as messages to a Storage Queue, providing decoupling and in-motion storage.

4. On the **Create Event Subscription** page, do the following steps:

- Enter a **name** for the event subscription.
- Enter a **name** for the **system topic**. To learn about system topics, see [Overview of system topics](#).

 **Create Event Subscription** ...

Event Grid

Basics Filters Additional Features Delivery Properties Advanced Editor

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

EVENT SUBSCRIPTION DETAILS

Name * spegridsubscription ✓

Event Schema Event Grid Schema ▾

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Type Storage account

Source Resource spstorageaccount1027

System Topic Name * spegridsystopic ✓

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Filter to Event Types * 2 selected ▾

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type * ▾

Create

c. Select **Web Hook** for **Endpoint type**.

 **Create Event Subscription** ...

Event Grid

Basics **Filters** **Additional Features** **Delivery Properties** **Advanced Editor**

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

EVENT SUBSCRIPTION DETAILS

Name * spegridsubscription ✓

Event Schema Event Grid Schema ▾

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Type Storage account

Source Resource

System Topic Name * ⓘ

Web Hook

Azure Function

Storage Queues

Event Hubs

Hybrid Connections

Service Bus Queue

Service Bus Topic

Partner Destination

EVENT TYPES

Pick which event types get pushed to your dest

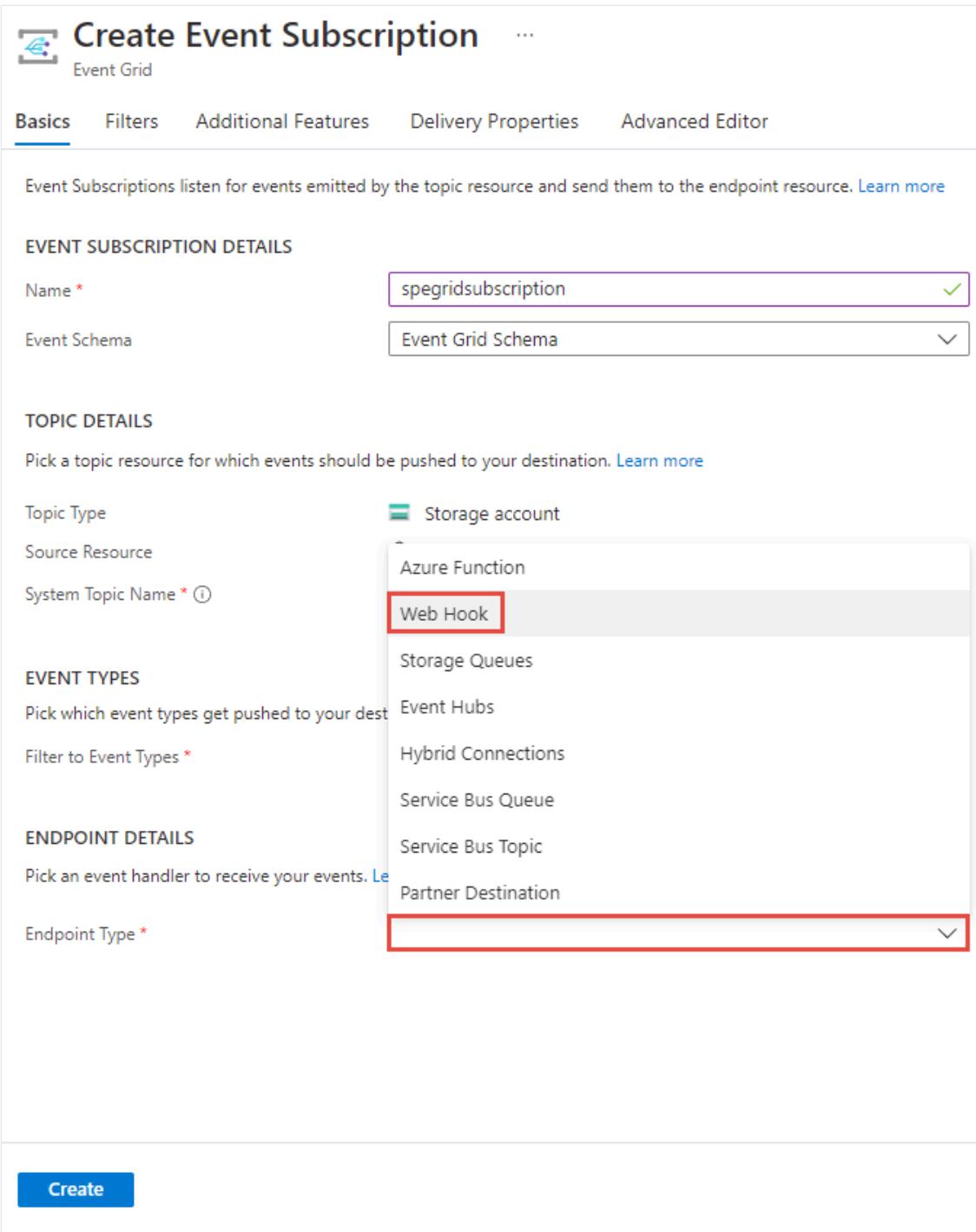
Filter to Event Types *

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type *

Create



5. For **Endpoint**, choose **Select an endpoint**, and enter the URL of your web app and add `api/updates` to the home page URL (for example: `https://spegridsite.azurewebsites.net/api/updates`), and then select **Confirm Selection**.

The screenshot shows the Azure portal interface for creating an event subscription. On the left, the main page displays 'Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource.' Below this are sections for 'EVENT SUBSCRIPTION DETAILS', 'TOPIC DETAILS', 'EVENT TYPES', and 'ENDPOINT DETAILS'. In the 'ENDPOINT DETAILS' section, the 'Endpoint Type' is set to 'Web Hook (change)' and the 'Endpoint' field contains the URL 'https://mygridviewer1027.azurewebsites.net/api/updates'. A red box highlights this URL. To the right, a modal window titled 'Select Web Hook' is open, showing the 'Subscriber Endpoint' field with the same URL. A red box highlights this field. The number '2' is placed near the URL in the main form, and the number '3' is placed near the URL in the modal. A circular button with a plus sign and a magnifying glass icon is also visible.

6. Now, on the **Create Event Subscription** page, select **Create** to create the event subscription.



Create Event Subscription

...

Event Grid

Basics Filters Additional Features Delivery Properties Advanced Editor

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

EVENT SUBSCRIPTION DETAILS

Name *

spegridsubscription



Event Schema

Event Grid Schema



TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Type

Storage account

Source Resource

spstorageaccount1027

System Topic Name *

spegridssystopic



EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Filter to Event Types *

2 selected



ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type *

Web Hook [\(change\)](#)

Endpoint *

<https://myegridviewer1027.azurewebsites.net/api/updates> [\(change\)](#)

Create

7. View your web app again, and notice that a subscription validation event has been sent to it. Select the eye icon to expand the event data. Event Grid sends the validation event so the endpoint can verify that it wants to receive event data. The web app includes code to validate the subscription.



Event Type	Subject
Microsoft.EventGrid.SubscriptionValidationEvent	<pre>[{ "id": "6a7d9e9f-3a33-4da3-aa8e-c1e7bb9764d6", "topic": "/subscriptions/0000000000-0000-0000-0000-000000000000/resourceGroups/spegridrg/providers/Microsoft.Storage/storageAccounts/spstorageaccount1127", "subject": "", "data": { "validationCode": "7F1D940E-3161-4082-A61D-26778CB7484B", "validationUrl": "https://rp-eastus.eventgrid.azure.net:553/events/subscriptions/spegridsubscription/validate?id=7F1D940E-3161-4082-A61D-26778CB7484B&t=2023-11-27T19:40:07.6384543Z&apiVersion=2023-1-15-preview&token=RMzbApTOhQH00c4JYjtzudlWpt8YK%2bGUkIV1m%2bN7rVs%3d" }, "eventType": "Microsoft.EventGrid.SubscriptionValidationEvent", "eventTime": "2023-11-27T19:40:07.6384543Z", "metadataVersion": "1", "dataVersion": "2" }]</pre>

Now, let's trigger an event to see how Event Grid distributes the message to your endpoint.

Send an event to your endpoint

You trigger an event for the Blob storage by uploading a file. The file doesn't need any specific content.

1. In the Azure portal, navigate to your Blob storage account, and select **Containers** on the left menu.
2. Select **+ Container**. Give your container a name, and use any access level, and select **Create**.

The screenshot shows the 'Containers' blade for the storage account 'spstorageaccount1027'. The left sidebar has 'Containers' selected (1). The top navigation bar includes a search bar, a '+ Container' button (2), and other account management links. A modal window titled 'New container' is open, prompting for a name ('eventcontainer') (3). The 'Public access level' dropdown is set to 'Private (no anonymous access)' (4). The main table lists existing containers like '\$logs'.

3. Select your new container.

The screenshot shows the 'Containers' blade for the same storage account. The left sidebar still has 'Containers' selected (1). The main table now lists two containers: '\$logs' and 'eventcontainer' (2). The 'eventcontainer' row is highlighted with a red border.

4. To upload a file, select **Upload**. On the **Upload blob** page, browse and select a file that you want to upload for testing, and then select **Upload** on that page.

The screenshot shows the 'eventcontainer' blade and an open 'Upload blob' dialog. The left sidebar of the container blade has 'Overview' selected (1). The top navigation bar includes a search bar, an 'Upload' button (2), and other container management links. The 'Upload blob' dialog shows a file 'Kitchen Cabinet.png' selected for upload (3). The 'Advanced' section is collapsed (4).

5. Browse to your test file and upload it.
6. You've triggered the event, and Event Grid sent the message to the endpoint you configured when subscribing. The message is in the JSON format and it contains an array with one or more events. In the following example, the JSON message contains an array with one event. View your web app and notice that a **blob created** event was received.

The screenshot shows the Azure Event Grid Viewer interface. At the top, there's a header with the title "Azure Event Grid Viewer". Below the header, there's a table with two columns: "Event Type" and "Subject". Under "Event Type", there's a Microsoft Storage icon followed by the text "Microsoft.Storage.BlobCreated". Under "Subject", there's the URL "/blobServices/default/containers/eventcontainer/blobs/Kitchen Cabinet.png". Below the table, a large JSON object is displayed, representing the event payload. The JSON starts with a brace and includes fields like "topic", "subject", "eventType", "id", "data", and "sequenceNumber". The "data" field is expanded to show detailed information about the blob, including its API ("PutBlob"), client request ID ("37bdc77e-f589-4b32-8d9c-bd55e956f02c"), request ID ("0bbfffb35-b01e-0087-496d-21739b06261d"), eTag ("0x8DBEF84A251A6A8"), content type ("image/png"), content length (777113), blob type ("BlockBlob"), URL ("https://spstorageaccount1127.blob.core.windows.net/eventcontainer/Kitchen Cabinet.png"), and storage diagnostics. The JSON ends with a brace. At the bottom of the viewer, there's another row with a Microsoft EventGrid icon and the text "Microsoft.EventGrid.SubscriptionValidationEvent".

```
{  
    "topic": "/subscriptions/000000000-0000-0000-0000-000000000000/resourceGroups/spegridrg/providers/Microsoft.Storage/storageAccounts/spstorageaccount1127",  
    "subject": "/blobServices/default/containers/eventcontainer/blobs/Kitchen Cabinet.png",  
    "eventType": "Microsoft.Storage.BlobCreated",  
    "id": "0bbfffb35-b01e-0087-496d-21739b06261d",  
    "data": {  
        "api": "PutBlob",  
        "clientRequestId": "37bdc77e-f589-4b32-8d9c-bd55e956f02c",  
        "requestId": "0bbfffb35-b01e-0087-496d-21739b000000",  
        "eTag": "0x8DBEF84A251A6A8",  
        "contentType": "image/png",  
        "contentLength": 777113,  
        "blobType": "BlockBlob",  
        "url": "https://spstorageaccount1127.blob.core.windows.net/eventcontainer/Kitchen Cabinet.png",  
        "sequencer": "00000000000000000000000000000371490000000003d83cd7",  
        "storageDiagnostics": {  
            "batchId": "9b6af860-d006-0081-006d-214024000000"  
        }  
    },  
    "dataVersion": "",  
    "metadataVersion": "1",  
    "eventTime": "2023-11-27T20:08:34.1048753Z"  
}  
Microsoft.EventGrid.SubscriptionValidationEvent
```

Clean up resources

If you plan to continue working with this event, don't clean up the resources created in this article. Otherwise, delete the resources you created in this article.

Select the resource group, and select **Delete resource group**.

Next steps

Now that you know how to create custom topics and event subscriptions, learn more about what Event Grid can help you do:

- [About Event Grid](#)
- [Route Blob storage events to a custom web endpoint](#)

- Monitor virtual machine changes with Azure Event Grid and Logic Apps
 - Stream big data into a data warehouse
-

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback 

Tutorial: Add Azure OpenAI text completion hints to your functions in Visual Studio Code

Article • 07/12/2024

This article shows you how to use Visual Studio Code to add an HTTP endpoint to the function app you created in the previous quickstart article. When triggered, this new HTTP endpoint uses an [Azure OpenAI text completion input binding](#) to get text completion hints from your data model.

During this tutorial, you learn how to accomplish these tasks:

- ✓ Create resources in Azure OpenAI.
- ✓ Deploy a model in OpenAI the resource.
- ✓ Set access permissions to the model resource.
- ✓ Enable your function app to connect to OpenAI.
- ✓ Add OpenAI bindings to your HTTP triggered function.

1. Check prerequisites

- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Obtain access to Azure OpenAI in your Azure subscription. If you haven't already been granted access, complete [this form](#) to request access.
- Install [.NET Core CLI tools](#).
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

2. Create your Azure OpenAI resources

The following steps show how to create an Azure OpenAI data model in the Azure portal.

1. Sign in with your Azure subscription in the [Azure portal](#).
2. Select **Create a resource** and search for the **Azure OpenAI**. When you locate the service, select **Create**.

3. On the **Create Azure OpenAI** page, provide the following information for the fields on the **Basics** tab:

[Expand table](#)

Field	Description
Subscription	Your subscription, which has been onboarded to use Azure OpenAI.
Resource group	The resource group you created for the function app in the previous article. You can find this resource group name by right-clicking the function app in the Azure Resources browser, selecting properties, and then searching for the <code>resourceGroup</code> setting in the returned JSON resource file.
Region	Ideally, the same location as the function app.
Name	A descriptive name for your Azure OpenAI Service resource, such as <code>mySampleOpenAI</code> .
Pricing Tier	The pricing tier for the resource. Currently, only the Standard tier is available for the Azure OpenAI Service. For more info on pricing visit the Azure OpenAI pricing page

Create Azure OpenAI

...

1 Basics

2 Network

3 Tags

4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

OpenAI Test Subscription



Resource group * ⓘ

test-resource-group



[Create new](#)

Instance details

Region * ⓘ

South Central US



Name * ⓘ

azure-openai-test-001



Pricing tier * ⓘ

Standard S0



[View full pricing details](#)

Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the Completions and image generations APIs as well as the content it sends back. If content is flagged by the service's filters, it may be reviewed by a Microsoft full-time employee.

[Learn more about how Microsoft processes, uses, and stores your data](#)

[Apply for modified content filters and abuse monitoring](#)

[Review the Azure OpenAI code of conduct](#)

[Previous](#)

[Next](#)

4. Select **Next** twice to accept the default values for both the **Network** and **Tags** tabs.

The service you create doesn't have any network restrictions, including from the internet.

5. Select **Next** a final time to move to the final stage in the process: **Review + submit**.

6. Confirm your configuration settings, and select **Create**.

The Azure portal displays a notification when the new resource is available. Select **Go to resource** in the notification or search for your new Azure OpenAI resource by name.

7. In the Azure OpenAI resource page for your new resource, select **Click here to view endpoints** under **Essentials > Endpoints**. Copy the **endpoint URL** and the **keys**. Save these values, you need them later.

Now that you have the credentials to connect to your model in Azure OpenAI, you need to set these access credentials in application settings.

3. Deploy a model

Now you can deploy a model. You can select from one of several available models in Azure OpenAI Studio.

To deploy a model, follow these steps:

1. Sign in to [Azure OpenAI Studio](#).
2. Choose the subscription and the Azure OpenAI resource you created, and select **Use resource**.
3. Under **Management** select **Deployments**.
4. Select **Create new deployment** and configure the following fields:

[] [Expand table](#)

Field	Description
Deployment name	Choose a name carefully. The deployment name is used in your code to call the model by using the client libraries and the REST APIs, so you must save for use later on.
Select a model	Model availability varies by region. For a list of available models per region, see Model summary table and region availability .

i Important

When you access the model via the API, you need to refer to the deployment name rather than the underlying model name in API calls, which is one of the key differences between OpenAI and Azure OpenAI. OpenAI only requires the model name. Azure OpenAI always requires deployment name, even when using the model parameter. In our docs, we often have examples where deployment names are represented as identical to model names to help indicate which model works with a particular API endpoint. Ultimately your

deployment names can follow whatever naming convention is best for your use case.

5. Accept the default values for the rest of the setting and select **Create**.

The deployments table shows a new entry that corresponds to your newly created model.

You now have everything you need to add Azure OpenAI-based text completion to your function app.

4. Update application settings

1. In Visual Studio Code, open the local code project you created when you completed the [previous article](#).
2. In the local.settings.json file in the project root folder, update the `AzureWebJobsStorage` setting to `UseDevelopmentStorage=true`. You can skip this step if the `AzureWebJobsStorage` setting in *local.settings.json* is set to the connection string for an existing Azure Storage account instead of `UseDevelopmentStorage=true`.
3. In the local.settings.json file, add these settings values:
 - `AZURE_OPENAI_ENDPOINT`: required by the binding extension. Set this value to the endpoint of the Azure OpenAI resource you created earlier.
 - `AZURE_OPENAI_KEY`: required by the binding extension. Set this value to the key for the Azure OpenAI resource.
 - `CHAT_MODEL_DEPLOYMENT_NAME`: used to define the input binding. Set this value to the name you chose for your model deployment.
4. Save the file. When you deploy to Azure, you must also add these settings to your function app.

5. Register binding extensions

Because you're using an Azure OpenAI output binding, you must have the corresponding bindings extension installed before you run the project.

Except for HTTP and timer triggers, bindings are implemented as extension packages. To add the Azure OpenAI extension package to your project, run this [dotnet add package](#) command in the **Terminal** window:

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.OpenAI --  
prerelease
```

Now, you can use the Azure OpenAI output binding in your project.

6. Return text completion from the model

The code you add creates a `whois` HTTP function endpoint in your existing project. In this function, data passed in a URL `name` parameter of a GET request is used to dynamically create a completion prompt. This dynamic prompt is bound to a text completion input binding, which returns a response from the model based on the prompt. The completion from the model is returned in the HTTP response.

1. In your existing `HttpExample` class file, add this `using` statement:

C#

```
using  
Microsoft.Azure.Functions.Worker.Extensions.OpenAI.TextCompletion;
```

2. In the same file, add this code that defines a new HTTP trigger endpoint named `whois`:

C#

```
[Function(nameof(WhoIs))]  
public IActionResult WhoIs([HttpTrigger(AuthorizationLevel.Function,  
Route = "whois/{name}")] HttpRequest req,  
[TextCompletionInput("Who is {name}?", Model =  
"%CHAT_MODEL_DEPLOYMENT_NAME%")] TextCompletionResponse response)  
{  
    if(!String.IsNullOrEmpty(response.Content))  
    {  
        return new OkObjectResult(response.Content);  
    }  
    else  
    {  
        return new NotFoundObjectResult("Something went wrong.");  
    }  
}
```

7. Run the function

1. In Visual Studio Code, Press F1 and in the command palette type `Azurite: Start` and press Enter to start the Azurite storage emulator.
2. Press `F5` to start the function app project and Core Tools in debug mode.
3. With the Core Tools running, send a GET request to the `whois` endpoint function, with a name in the path, like this URL:

```
http://localhost:7071/api/whois/<NAME>
```

Replace the `<NAME>` string with the value you want passed to the `"Who is {name}?"` prompt. The `<NAME>` must be the URL-encoded name of a public figure, like `Abraham%20Lincoln`.

The response you see is the text completion response from your Azure OpenAI model.

4. After a response is returned, press `Ctrl + C` to stop Core Tools.

8. Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You created resources to complete these quickstarts. You could be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', and 'Security'. Below that is another 'Functions' section with 'Functions', 'App keys', 'App files', and 'Proxies'. The main pane shows the 'Overview' tab for a resource group named 'myResourceGroup'. The resource group details include:

Setting	Value
Status	Running
Location	Central US
Subscription	Visual Studio Enterprise
Subscription ID	11111111-1111-1111-1111-111111111111
Tags	Click here to add tags
URL	https://myfunctionapp.azurewebsites.net
Operating System	Windows
App Service Plan	ASP-myResourceGroup-a285 (Y1: 0)
Properties	See More
Runtime version	3.0.13139.0

At the bottom of the main pane, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Related content

- [Azure OpenAI extension for Azure Functions](#)
- [Azure OpenAI extension samples ↗](#)
- [Machine learning and AI](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Apply machine learning models in Azure Functions with Python and TensorFlow

Article • 03/09/2023

In this article, you learn how to use Python, TensorFlow, and Azure Functions with a machine learning model to classify an image based on its contents. Because you do all work locally and create no Azure resources in the cloud, there is no cost to complete this tutorial.

- ✓ Initialize a local environment for developing Azure Functions in Python.
- ✓ Import a custom TensorFlow machine learning model into a function app.
- ✓ Build a serverless HTTP API for classifying an image as containing a dog or a cat.
- ✓ Consume the API from a web app.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Python 3.7.4](#). (Python 3.7.4 and Python 3.6.x are verified with Azure Functions; Python 3.8 and later versions are not yet supported.)
- The [Azure Functions Core Tools](#)
- A code editor such as [Visual Studio Code](#)

Prerequisite check

1. In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 2.7.1846 or later.
2. Run `python --version` (Linux/MacOS) or `py --version` (Windows) to check your Python version reports 3.7.x.

Clone the tutorial repository

1. In a terminal or command window, clone the following repository using Git:

```
git clone https://github.com/Azure-Samples/functions-python-tensorflow-tutorial.git
```

2. Navigate into the folder and examine its contents.

```
cd functions-python-tensorflow-tutorial
```

- *start* is your working folder for the tutorial.
- *end* is the final result and full implementation for your reference.
- *resources* contains the machine learning model and helper libraries.
- *frontend* is a website that calls the function app.

Create and activate a Python virtual environment

Navigate to the *start* folder and run the following commands to create and activate a virtual environment named `.venv`. Be sure to use Python 3.7, which is supported by Azure Functions.

bash

```
Bash
```

```
cd start
python -m venv .venv
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

Bash

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment. (To exit the virtual environment, run `deactivate`.)

Create a local functions project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local

and hosting configurations. In this section, you create a function project that contains a single boilerplate function named `classify` that provides an HTTP endpoint. You add more specific code in a later section.

1. In the *start* folder, use the Azure Functions Core Tools to initialize a Python function app:

```
func init --worker-runtime python
```

After initialization, the *start* folder contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file.

 **Tip**

Because a function project is tied to a specific runtime, all the functions in the project must be written with the same language.

2. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function and the `--template` argument specifies the function's trigger. `func new` create a subfolder matching the function name that contains a code file appropriate to the project's chosen language and a configuration file named `function.json`.

```
func new --name classify --template "HTTP trigger"
```

This command creates a folder matching the name of the function, `classify`. In that folder are two files: `__init__.py`, which contains the function code, and `function.json`, which describes the function's trigger and its input and output bindings. For details on the contents of these files, see [Examine the file contents](#) in the Python quickstart.

Run the function locally

1. Start the function by starting the local Azure Functions runtime host in the *start* folder:

```
func start
```

2. Once you see the `classify` endpoint appear in the output, navigate to the URL, <http://localhost:7071/api/classify?name=Azure>. The message "Hello Azure!" should appear in the output.
3. Use **Ctrl-C** to stop the host.

Import the TensorFlow model and add helper code

To modify the `classify` function to classify an image based on its contents, you use a pre-built TensorFlow model that was trained with and exported from Azure Custom Vision Service. The model, which is contained in the `resources` folder of the sample you cloned earlier, classifies an image based on whether it contains a dog or a cat. You then add some helper code and dependencies to your project.

To build your own model using the free tier of the Custom Vision Service, follow the instructions in the [sample project repository](#).

Tip

If you want to host your TensorFlow model independent of the function app, you can instead mount a file share containing your model to your Linux function app. To learn more, see [Mount a file share to a Python function app using Azure CLI](#).

1. In the `start` folder, run following command to copy the model files into the `classify` folder. Be sure to include `*` in the command.

```
bash
```

```
Bash
```

```
cp ../resources/model/* classify
```

2. Verify that the `classify` folder contains files named `model.pb` and `labels.txt`. If not, check that you ran the command in the `start` folder.

3. In the *start* folder, run the following command to copy a file with helper code into the *classify* folder:

```
bash
```

```
Bash
```

```
cp ../resources/predict.py classify
```

4. Verify that the *classify* folder now contains a file named *predict.py*.
5. Open *start/requirements.txt* in a text editor and add the following dependencies required by the helper code:

```
txt
```

```
tensorflow==1.14  
Pillow  
requests
```

6. Save *requirements.txt*.
7. Install the dependencies by running the following command in the *start* folder. Installation may take a few minutes, during which time you can proceed with modifying the function in the next section.

```
pip install --no-cache-dir -r requirements.txt
```

On Windows, you may encounter the error, "Could not install packages due to an EnvironmentError: [Errno 2] No such file or directory:" followed by a long pathname to a file like *sharded Mutable Dense Hashtable.cpython-37.pyc*. Typically, this error happens because the depth of the folder path becomes too long. In this case, set the registry key

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem@LongPathsEnabled` to `1` to enable long paths. Alternately, check where your Python interpreter is installed. If that location has a long path, try reinstalling in a folder with a shorter path.

 Tip

When calling upon `predict.py` to make its first prediction, a function named `_initialize` loads the TensorFlow model from disk and caches it in global variables. This caching speeds up subsequent predictions. For more information on using global variables, refer to the [Azure Functions Python developer guide](#).

Update the function to run predictions

1. Open `classify/_init_.py` in a text editor and add the following lines after the existing `import` statements to import the standard JSON library and the `predict` helpers:

Python

```
import logging
import azure.functions as func
import json

# Import helper script
from .predict import predict_image_from_url
```

2. Replace the entire contents of the `main` function with the following code:

Python

```
def main(req: func.HttpRequest) -> func.HttpResponse:
    image_url = req.params.get('img')
    logging.info('Image URL received: ' + image_url)

    results = predict_image_from_url(image_url)

    headers = {
        "Content-type": "application/json",
        "Access-Control-Allow-Origin": "*"
    }

    return func.HttpResponse(json.dumps(results), headers = headers)
```

This function receives an image URL in a query string parameter named `img`. It then calls `predict_image_from_url` from the helper library to download and classify the image using the TensorFlow model. The function then returns an HTTP response with the results.

 **Important**

Because this HTTP endpoint is called by a web page hosted on another domain, the response includes an `Access-Control-Allow-Origin` header to satisfy the browser's Cross-Origin Resource Sharing (CORS) requirements.

In a production application, change `*` to the web page's specific origin for added security.

3. Save your changes, then assuming that dependencies have finished installing, start the local function host again with `func start`. Be sure to run the host in the `start` folder with the virtual environment activated. Otherwise the host will start, but you will see errors when invoking the function.

```
func start
```

4. In a browser, open the following URL to invoke the function with the URL of a cat image and confirm that the returned JSON classifies the image as a cat.

```
http://localhost:7071/api/classify?  
img=https://raw.githubusercontent.com/Azure-Samples/functions-python-  
tensorflow-tutorial/master/resources/assets/samples/cat1.png
```

5. Keep the host running because you use it in the next step.

Run the local web app front end to test the function

To test invoking the function endpoint from another web app, there's a simple app in the repository's `frontend` folder.

1. Open a new terminal or command prompt and activate the virtual environment (as described earlier under [Create and activate a Python virtual environment](#)).
2. Navigate to the repository's `frontend` folder.
3. Start an HTTP server with Python:

```
bash
```

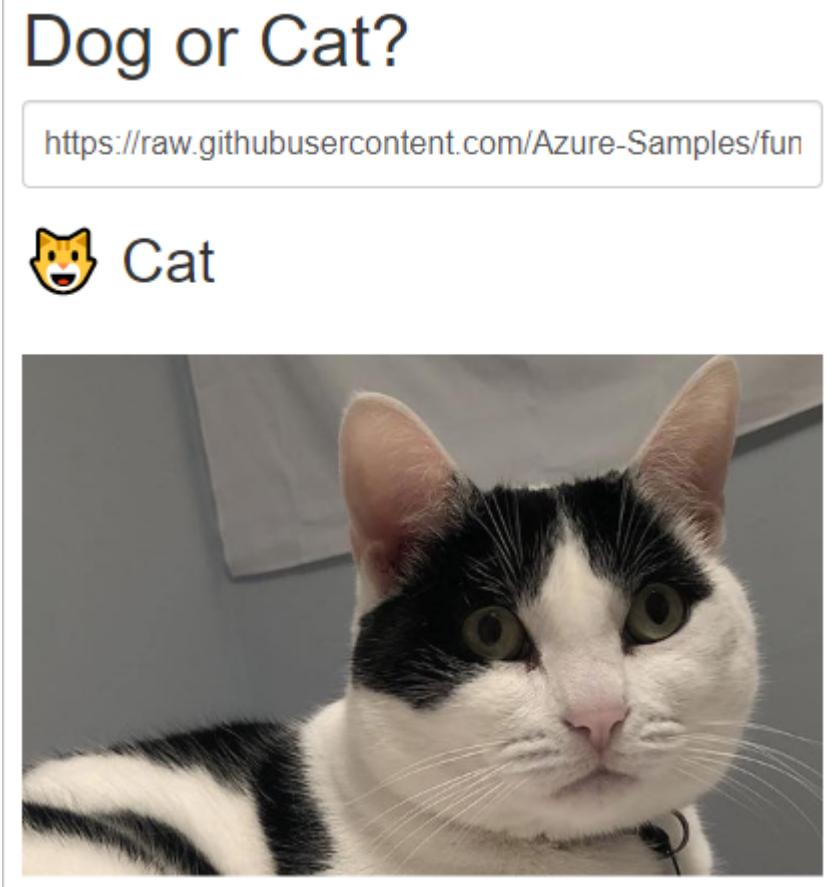
```
Bash
```

```
python -m http.server
```

4. In a browser, navigate to `localhost:8000`, then enter one of the following photo URLs into the textbox, or use the URL of any publicly accessible image.

- `https://raw.githubusercontent.com/Azure-Samples/functions-python-tensorflow-tutorial/master/resources/assets/samples/cat1.png`
- `https://raw.githubusercontent.com/Azure-Samples/functions-python-tensorflow-tutorial/master/resources/assets/samples/cat2.png`
- `https://raw.githubusercontent.com/Azure-Samples/functions-python-tensorflow-tutorial/master/resources/assets/samples/dog1.png`
- `https://raw.githubusercontent.com/Azure-Samples/functions-python-tensorflow-tutorial/master/resources/assets/samples/dog2.png`

5. Select **Submit** to invoke the function endpoint to classify the image.



If the browser reports an error when you submit the image URL, check the terminal in which you're running the function app. If you see an error like "No module found 'PIL'", you may have started the function app in the `start` folder without first activating the virtual environment you created earlier. If you still see errors, run `pip`

```
install -r requirements.txt again with the virtual environment activated and look for errors.
```

ⓘ Note

The model always classifies the content of the image as a cat or a dog, regardless of whether the image contains either, defaulting to dog. Images of tigers and panthers, for example, typically classify as cat, but images of elephants, carrots, or airplanes classify as dog.

Clean up resources

Because the entirety of this tutorial runs locally on your machine, there are no Azure resources or services to clean up.

Next steps

In this tutorial, you learned how to build and customize an HTTP API endpoint with Azure Functions to classify images using a TensorFlow model. You also learned how to call the API from a web app. You can use the techniques in this tutorial to build out APIs of any complexity, all while running on the serverless compute model provided by Azure Functions.

[Deploy the function to Azure Functions using the Azure CLI Guide](#)

See also:

- [Deploy the function to Azure using Visual Studio Code ↗](#).
- [Azure Functions Python Developer Guide](#)
- [Mount a file share to a Python function app using Azure CLI](#)

Tutorial: Deploy a pre-trained image classification model to Azure Functions with PyTorch

Article • 03/09/2023

In this article, you learn how to use Python, PyTorch, and Azure Functions to load a pre-trained model for classifying an image based on its contents. Because you do all work locally and create no Azure resources in the cloud, there's no cost to complete this tutorial.

- ✓ Initialize a local environment for developing Azure Functions in Python.
- ✓ Import a pre-trained PyTorch machine learning model into a function app.
- ✓ Build a serverless HTTP API for classifying an image as one of 1000 ImageNet classes ↗ .
- ✓ Consume the API from a web app.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Python 3.7.4 or above ↗](#). (Python 3.8.x and Python 3.6.x are also verified with Azure Functions.)
- The [Azure Functions Core Tools](#)
- A code editor such as [Visual Studio Code ↗](#)

Prerequisite check

1. In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 2.7.1846 or later.
2. Run `python --version` (Linux/MacOS) or `py --version` (Windows) to check your Python version reports 3.7.x.

Clone the tutorial repository

1. In a terminal or command window, clone the following repository using Git:



```
git clone https://github.com/Azure-Samples/functions-python-pytorch-tutorial.git
```

2. Navigate into the folder and examine its contents.

```
cd functions-python-pytorch-tutorial
```

- *start* is your working folder for the tutorial.
- *end* is the final result and full implementation for your reference.
- *resources* contains the machine learning model and helper libraries.
- *frontend* is a website that calls the function app.

Create and activate a Python virtual environment

Navigate to the *start* folder and run the following commands to create and activate a virtual environment named `.venv`.

bash

Bash

```
cd start
python -m venv .venv
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

Bash

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment. (To exit the virtual environment, run `deactivate`.)

Create a local functions project

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations. In this section, you create a function project that contains a single boilerplate function named `classify` that provides an HTTP endpoint. You add more specific code in a later section.

1. In the `start` folder, use the Azure Functions Core Tools to initialize a Python function app:

```
func init --worker-runtime python
```

After initialization, the `start` folder contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file.

 **Tip**

Because a function project is tied to a specific runtime, all the functions in the project must be written with the same language.

2. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function and the `--template` argument specifies the function's trigger. `func new` create a subfolder matching the function name that contains a code file appropriate to the project's chosen language and a configuration file named `function.json`.

```
func new --name classify --template "HTTP trigger"
```

This command creates a folder matching the name of the function, `classify`. In that folder are two files: `__init__.py`, which contains the function code, and `function.json`, which describes the function's trigger and its input and output bindings. For details on the contents of these files, see [Examine the file contents](#) in the Python quickstart.

Run the function locally

1. Start the function by starting the local Azure Functions runtime host in the *start* folder:

```
func start
```

2. Once you see the `classify` endpoint appear in the output, navigate to the URL, `http://localhost:7071/api/classify?name=Azure`. The message "Hello Azure!" should appear in the output.

3. Use **Ctrl-C** to stop the host.

Import the PyTorch model and add helper code

To modify the `classify` function to classify an image based on its contents, you use a pre-trained [ResNet](#) model. The pre-trained model, which comes from [PyTorch](#), classifies an image into 1 of 1000 [ImageNet classes](#). You then add some helper code and dependencies to your project.

1. In the *start* folder, run the following command to copy the prediction code and labels into the *classify* folder.

```
bash
Bash
cp ../resources/predict.py classify
cp ../resources/labels.txt classify
```

2. Verify that the *classify* folder contains files named *predict.py* and *labels.txt*. If not, check that you ran the command in the *start* folder.
3. Open *start/requirements.txt* in a text editor and add the dependencies required by the helper code, which should look like:

```
txt
azure-functions
requests
-f https://download.pytorch.org/whl/torch_stable.html
torch==1.13.0+cpu
torchvision==0.14.0+cpu
```

💡 Tip

The versions of torch and torchvision must match values listed in the version table of the [PyTorch vision repo](#).

4. Save *requirements.txt*, then run the following command from the *start* folder to install the dependencies.

```
pip install --no-cache-dir -r requirements.txt
```

Installation may take a few minutes, during which time you can proceed with modifying the function in the next section.

💡 Tip

On Windows, you may encounter the error, "Could not install packages due to an EnvironmentError: [Errno 2] No such file or directory:" followed by a long pathname to a file like *sharded Mutable Dense Hashtable.cpython-37.pyc*.

Typically, this error happens because the depth of the folder path becomes too long. In this case, set the registry key

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem@LongPathsEnabled` to 1 to enable long paths. Alternately, check where your Python interpreter is installed. If that location has a long path, try reinstalling in a folder with a shorter path.

Update the function to run predictions

1. Open *classify/_init_.py* in a text editor and add the following lines after the existing `import` statements to import the standard JSON library and the *predict* helpers:

Python

```
import logging
import azure.functions as func
import json
```

```
# Import helper script
from .predict import predict_image_from_url
```

2. Replace the entire contents of the `main` function with the following code:

Python

```
def main(req: func.HttpRequest) -> func.HttpResponse:
    image_url = req.params.get('img')
    logging.info('Image URL received: ' + image_url)

    results = predict_image_from_url(image_url)

    headers = {
        "Content-type": "application/json",
        "Access-Control-Allow-Origin": "*"
    }

    return func.HttpResponse(json.dumps(results), headers = headers)
```

This function receives an image URL in a query string parameter named `img`. It then calls `predict_image_from_url` from the helper library to download and classify the image using the PyTorch model. The function then returns an HTTP response with the results.

 **Important**

Because this HTTP endpoint is called by a web page hosted on another domain, the response includes an `Access-Control-Allow-Origin` header to satisfy the browser's Cross-Origin Resource Sharing (CORS) requirements.

In a production application, change `*` to the web page's specific origin for added security.

3. Save your changes, then assuming that dependencies have finished installing, start the local function host again with `func start`. Be sure to run the host in the `start` folder with the virtual environment activated. Otherwise the host will start, but you'll see errors when invoking the function.

```
func start
```

4. In a browser, open the following URL to invoke the function with the URL of a Bernese Mountain Dog image and confirm that the returned JSON classifies the image as a Bernese Mountain Dog.

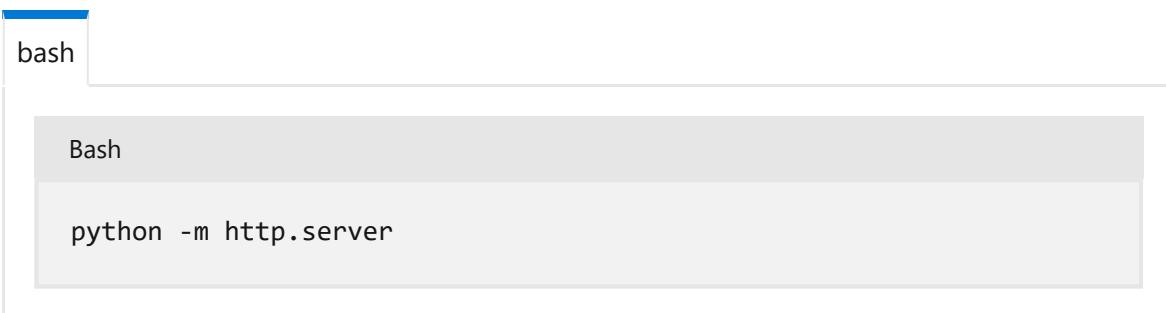
```
http://localhost:7071/api/classify?  
img=https://raw.githubusercontent.com/Azure-Samples/functions-python-  
pytorch-tutorial/master/resources/assets/Bernese-Mountain-Dog-  
Temperament-long.jpg
```

5. Keep the host running because you use it in the next step.

Run the local web app front end to test the function

To test invoking the function endpoint from another web app, there's a simple app in the repository's *frontend* folder.

1. Open a new terminal or command prompt and activate the virtual environment (as described earlier under [Create and activate a Python virtual environment](#)).
2. Navigate to the repository's *frontend* folder.
3. Start an HTTP server with Python:



```
bash  
Bash  
python -m http.server
```

4. In a browser, navigate to `localhost:8000`, then enter one of the following photo URLs into the textbox, or use the URL of any publicly accessible image.

- `https://raw.githubusercontent.com/Azure-Samples/functions-python-
pytorch-tutorial/master/resources/assets/Bernese-Mountain-Dog-
Temperament-long.jpg`
- `https://github.com/Azure-Samples/functions-python-pytorch-
tutorial/blob/master/resources/assets/bald-eagle.jpg?raw=true`
- `https://raw.githubusercontent.com/Azure-Samples/functions-python-
pytorch-tutorial/master/resources/assets/penguin.jpg`

5. Select **Submit** to invoke the function endpoint to classify the image.

Run PyTorch Image Classification

<https://raw.githubusercontent.com/Azure-Samples/functions-image-classification/master/submit.html>

king penguin, *Aptenodytes patagonica*



If the browser reports an error when you submit the image URL, check the terminal in which you're running the function app. If you see an error like "No module found 'PIL'", you may have started the function app in the *start* folder without first activating the virtual environment you created earlier. If you still see errors, run `pip install -r requirements.txt` again with the virtual environment activated and look for errors.

Clean up resources

Because the entirety of this tutorial runs locally on your machine, there are no Azure resources or services to clean up.

Next steps

In this tutorial, you learned how to build and customize an HTTP API endpoint with Azure Functions to classify images using a PyTorch model. You also learned how to call the API from a web app. You can use the techniques in this tutorial to build out APIs of any complexity, all while running on the serverless compute model provided by Azure Functions.

See also:

- [Deploy the function to Azure using Visual Studio Code](#).
- [Azure Functions Python Developer Guide](#)

[Deploy the function to Azure Functions using the Azure CLI Guide](#)

Tutorial: Deploy Azure Functions as IoT Edge modules

Article • 06/10/2024

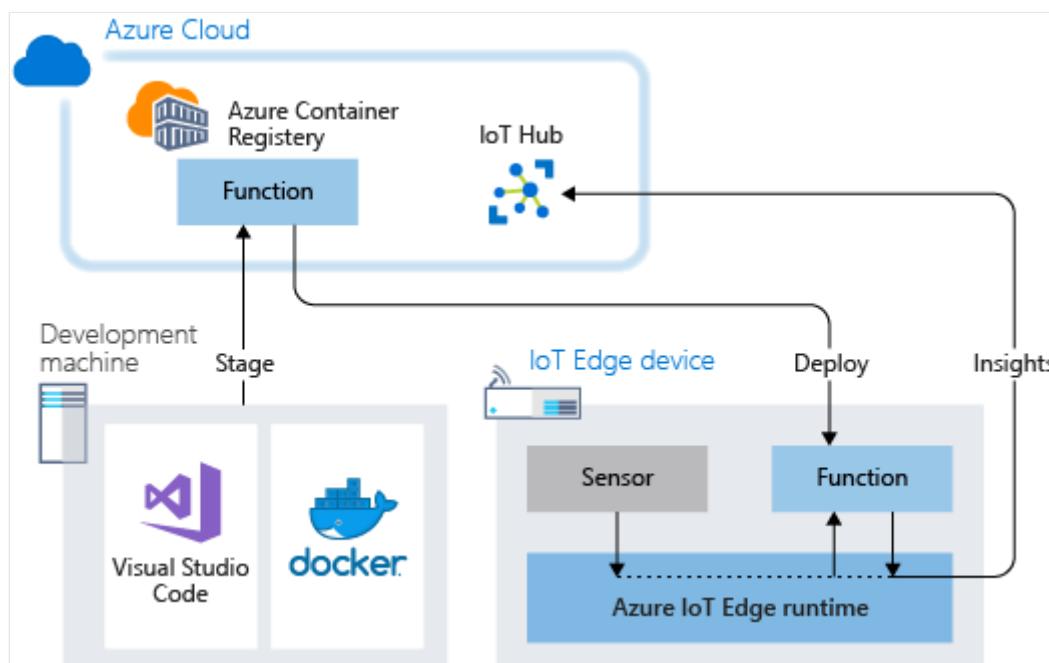
Applies to: ✓ IoT Edge 1.5 ✓ IoT Edge 1.4

ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can use Azure Functions to deploy code that implements your business logic directly to your Azure IoT Edge devices. This tutorial walks you through creating and deploying an Azure Function that filters sensor data on the simulated IoT Edge device. You use the simulated IoT Edge device that you created in the quickstarts. In this tutorial, you learn how to:

- ✓ Use Visual Studio Code to create an Azure Function.
- ✓ Use Visual Studio Code and Docker to create a Docker image and publish it to a container registry.
- ✓ Deploy the module from the container registry to your IoT Edge device.
- ✓ View filtered data.



The Azure Function that you create in this tutorial filters the temperature data that's generated by your device. The Function only sends messages upstream to Azure IoT

Hub when the temperature is above a specified threshold.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

Before beginning this tutorial, do the tutorial to set up your development environment for Linux container development: [Develop Azure IoT Edge modules using Visual Studio Code](#). After completing that tutorial, you should have the following prerequisites in place:

- A free or standard-tier [IoT Hub](#) in Azure.
- An AMD64 device running Azure IoT Edge with Linux containers. You can use the quickstart to set up a [Linux device](#) or [Windows device](#).
- A container registry, like [Azure Container Registry](#).
- [Visual Studio Code](#) configured with the [Azure IoT Edge](#) and [Azure IoT Hub](#) extensions. The *Azure IoT Edge tools for Visual Studio Code* extension is in [maintenance mode](#).
- Download and install a [Docker compatible container management system](#) on your development machine. Configure it to run Linux containers.

To develop an IoT Edge module with Azure Functions, install additional prerequisites on your development machine:

- [C# for Visual Studio Code \(powered by OmniSharp\) extension](#).
- [The .NET Core SDK](#).

Create a function project

The Azure IoT Edge for Visual Studio Code that you installed in the prerequisites provides management capabilities and some code templates. In this section, you use Visual Studio Code to create an IoT Edge solution that contains an Azure Function.

Create a new project

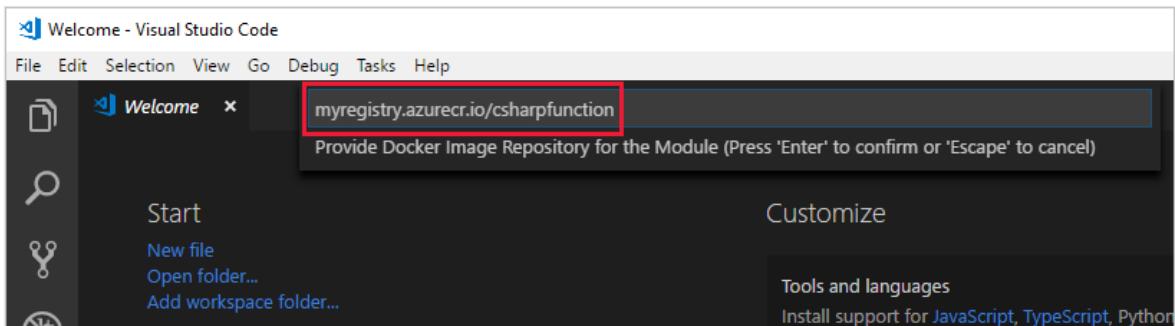
Follow these steps to create a C# Function solution template that's customizable.

1. Open Visual Studio Code on your development machine.
2. Open the Visual Studio Code command palette by selecting **View > Command Palette**.

3. In the command palette, add and run the command **Azure IoT Edge: New IoT**

Edge solution. Follow these prompts in the command palette to create your solution:

- Select a folder: choose the location on your development machine for Visual Studio Code to create the solution files.
- Provide a solution name: add a descriptive name for your solution, like **FunctionSolution**, or accept the default.|
- Select a module template: choose **Azure Functions - C#**.
- Provide a module name | Name your module **CSharpFunction**.
- Provide a Docker image repository for the module. An image repository includes the name of your container registry and the name of your container image. Your container image is pre-populated from the last step. Replace **localhost:5000** with the **Login server** value from your Azure container registry. You can retrieve the **Login server** from the **Overview** page of your container registry in the Azure portal. The final string looks like <registry name>.azurecr.io/csharpfunction.



Add your registry credentials

The environment file of your solution stores the credentials for your container registry and shares them with the IoT Edge runtime. The runtime needs these credentials to pull your private images onto your IoT Edge device.

The IoT Edge extension in Visual Studio Code tries to pull your container registry credentials from Azure and populate them in the environment file. Check to see if your credentials are already in the file. If not, add them now:

1. In the Visual Studio Code explorer, open the `.env` file.
2. Update the fields with the **username** and **password** values that you copied from your Azure container registry. You can find them again by going to your container registry in Azure and looking on the **Settings > Access keys** page.
3. Save this file.

(!) Note

This tutorial uses admin login credentials for Azure Container Registry, which are convenient for development and test scenarios. When you're ready for production scenarios, we recommend a least-privilege authentication option like service principals. For more information, see [Manage access to your container registry](#).

Set target architecture to AMD64

Running Azure Functions modules on IoT Edge is supported only on Linux AMD64 based containers. The default target architecture for Visual Studio Code is Linux AMD64, but we set it explicitly to Linux AMD64 here.

1. Open the command palette and search for **Azure IoT Edge: Set Default Target Platform for Edge Solution**.
2. In the command palette, select the AMD64 target architecture from the list of options.

Update the module with custom code

Let's add some additional code so your **CSharpFunction** module processes the messages at the edge before forwarding them to IoT Hub.

1. In the Visual Studio Code explorer, open **modules > CSharpFunction > CSharpFunction.cs**.
2. Replace the contents of the **CSharpFunction.cs** file with the following code. This code receives telemetry about ambient and machine temperature, and only forwards the message on to IoT Hub if the machine temperature is above a defined threshold.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.EdgeHub;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
```

```

using Newtonsoft.Json;

namespace Functions.Samples
{
    public static class CSharpFunction
    {
        [FunctionName("CSharpFunction")]
        public static async Task FilterMessageAndSendMessage(
            [EdgeHubTrigger("input1")] Message messageReceived,
            [EdgeHub(OutputName = "output1")] IAsyncCollector<Message>
output,
            ILogger logger)
        {
            const int temperatureThreshold = 20;
            byte[] messageBytes = messageReceived.GetBytes();
            var messageString =
                System.Text.Encoding.UTF8.GetString(messageBytes);

            if (!string.IsNullOrEmpty(messageString))
            {
                logger.LogInformation("Info: Received one non-empty
message");
                // Get the body of the message and deserialize it.
                var messageBody =
                    JsonConvert.DeserializeObject<MessageBody>(messageString);

                if (messageBody != null &&
messageBody.machine.temperature > temperatureThreshold)
                {
                    // Send the message to the output as the
temperature value is greater than the threshold.
                    using (var filteredMessage = new
Message(messageBytes))
                    {
                        // Copy the properties of the original message
into the new Message object.
                        foreach (KeyValuePair<string, string> prop in
messageReceived.Properties)
                            {filteredMessage.Properties.Add(prop.Key,
prop.Value);}
                        // Add a new property to the message to
indicate it is an alert.
                        filteredMessage.Properties.Add("MessageType",
"Alert");
                        // Send the message.
                        await output.AddAsync(filteredMessage);
                        logger.LogInformation("Info: Received and
transferred a message with temperature above the threshold");
                    }
                }
            }
        }
    }
    //Define the expected schema for the body of incoming messages.
    class MessageBody

```

```
{  
    public Machine machine {get; set;}  
    public Ambient ambient {get; set;}  
    public string timeCreated {get; set;}  
}  
class Machine  
{  
    public double temperature {get; set;}  
    public double pressure {get; set;}  
}  
class Ambient  
{  
    public double temperature {get; set;}  
    public int humidity {get; set;}  
}  
}
```

3. Save the file.

Build and push your IoT Edge solution

In the previous section, you created an IoT Edge solution and modified the **CSharpFunction** to filter out messages with reported machine temperatures below the acceptable threshold. Now you need to build the solution as a container image and push it to your container registry.

1. Open the Visual Studio Code integrated terminal by selecting **View > Terminal**.
2. Sign in to Docker by entering the following command in the terminal. Sign in with the username, password, and login server from your Azure container registry. You can retrieve these values from the **Access keys** section of your registry in the Azure portal.

Bash

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

You may receive a security warning recommending the use of `--password-stdin`. While that best practice is recommended for production scenarios, it's outside the scope of this tutorial. For more information, see the [docker login](#) reference.

3. In the Visual Studio Code explorer, right-click the **deployment.template.json** file and select **Build and Push IoT Edge Solution**.

The build and push command starts three operations. First, it creates a new folder in the solution called **config** that holds the full deployment manifest, which is built

out of information in the deployment template and other solution files. Second, it runs `docker build` to build the container image based on the appropriate dockerfile for your target architecture. Then, it runs `docker push` to push the image repository to your container registry.

This process may take several minutes the first time, but is faster the next time that you run the commands.

View your container image

Visual Studio Code outputs a success message when your container image is pushed to your container registry. If you want to confirm the successful operation for yourself, you can view the image in the registry.

1. In the Azure portal, browse to your Azure container registry.
2. Select **Services > Repositories**.
3. You should see the **csharpfunction** repository in the list. Select this repository to see more details.
4. In the **Tags** section, you should see the **0.0.1-amd64** tag. This tag indicates the version and platform of the image that you built. These values are set in the `module.json` file in the `CSharpFunction` folder.

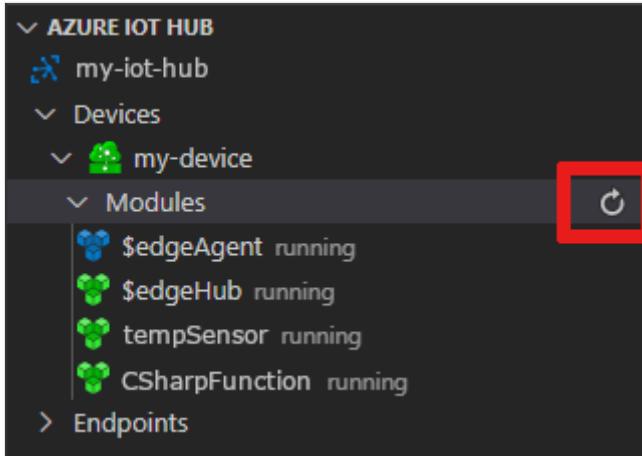
Deploy and run the solution

You can use the Azure portal to deploy your Function module to an IoT Edge device like you did in the quickstart. You can also deploy and monitor modules from within Visual Studio Code. The following sections use the Azure IoT Edge and IoT Hub for Visual Studio Code that was listed in the prerequisites. Install the extensions now, if you haven't already.

1. In the Visual Studio Code explorer, under the **Azure IoT Hub** section, expand **Devices** to see your list of IoT devices.
2. Right-click the name of your IoT Edge device, and then select **Create Deployment for Single Device**.
3. Browse to the solution folder that contains the **CSharpFunction**. Open the config folder, select the `deployment.amd64.json` file, and then choose **Select Edge Deployment Manifest**.
4. Under your device, expand **Modules** to see a list of deployed and running modules. Select the refresh button. You should see the new **CSharpFunction**

running along with the `SimulatedTemperatureSensor` module and the `$edgeAgent` and `$edgeHub`.

It may take a few moments for the new modules to show up. Your IoT Edge device has to retrieve its new deployment information from IoT Hub, start the new containers, and then report the status back to IoT Hub.



View the generated data

You can see all of the messages that arrive at your IoT hub from all your devices by running **Azure IoT Hub: Start Monitoring Built-in Event Endpoint** in the command palette. To stop monitoring messages, run the command **Azure IoT Hub: Stop Monitoring Built-in Event Endpoint** in the command palette.

You can also filter the view to see all of the messages that arrive at your IoT hub from a specific device. Right-click the device in the **Azure IoT Hub > Devices** section of the Visual Studio Code explorer and select **Start Monitoring Built-in Event Endpoint**.

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you created in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub

inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

In this tutorial, you created an Azure Function module with code to filter raw data that's generated by your IoT Edge device.

Continue on to the next tutorials to learn other ways that Azure IoT Edge can help you turn data into business insights at the edge.

[Find averages by using a floating window in Azure Stream Analytics](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Tutorial: Create a function in Java with an Event Hub trigger and an Azure Cosmos DB output binding

Article • 02/14/2024

This tutorial shows you how to use Azure Functions to create a Java function that analyzes a continuous stream of temperature and pressure data. Event hub events that represent sensor readings trigger the function. The function processes the event data, then adds status entries to an Azure Cosmos DB instance.

In this tutorial, you'll:

- ✓ Create and configure Azure resources using the Azure CLI.
- ✓ Create and test Java functions that interact with these resources.
- ✓ Deploy your functions to Azure and monitor them with Application Insights.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

To complete this tutorial, you must have the following installed:

- [Java Developer Kit](#), version 8
- [Apache Maven](#), version 3.0 or above
- [Azure Functions Core Tools](#) version 2.6.666 or above
- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).

 [Launch Cloud Shell](#) 

- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
- Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

ⓘ Important

The `JAVA_HOME` environment variable must be set to the install location of the JDK to complete this tutorial.

If you prefer to use the code for this tutorial directly, see the [java-functions-eventhub-cosmosdb](#) sample repo.

Create Azure resources

In this tutorial, you'll need these resources:

- A resource group to contain the other resources
- An Event Hubs namespace, event hub, and authorization rule
- An Azure Cosmos DB account, database, and collection
- A function app and a storage account to host it

The following sections show you how to create these resources using the Azure CLI.

Set environment variables

Next, create some environment variables for the names and location of the resources you'll create. Use the following commands, replacing the `<value>` placeholders with values of your choosing. Values should conform to the [naming rules and restrictions for Azure resources](#). For the `LOCATION` variable, use one of the values produced by the `az functionapp list-consumption-locations` command.

Bash

Bash

```
RESOURCE_GROUP=<value>
EVENT_HUB_NAMESPACE=<value>
EVENT_HUB_NAME=<value>
EVENT_HUB_AUTHORIZATION_RULE=<value>
COSMOS_DB_ACCOUNT=<value>
```

```
STORAGE_ACCOUNT=<value>
FUNCTION_APP=<value>
LOCATION=<value>
```

The rest of this tutorial uses these variables. Be aware that these variables persist only for the duration of your current Azure CLI or Cloud Shell session. You will need to run these commands again if you use a different local terminal window or your Cloud Shell session times out.

Create a resource group

Azure uses resource groups to collect all related resources in your account. That way, you can view them as a unit and delete them with a single command when you're done with them.

Use the following command to create a resource group:

Bash

Azure CLI

```
az group create \
--name $RESOURCE_GROUP \
--location $LOCATION
```

Create an event hub

Next, create an Azure Event Hubs namespace, event hub, and authorization rule using the following commands:

Bash

Azure CLI

```
az eventhubs namespace create \
--resource-group $RESOURCE_GROUP \
--name $EVENT_HUB_NAMESPACE
az eventhubs eventhub create \
--resource-group $RESOURCE_GROUP \
--name $EVENT_HUB_NAME \
--namespace-name $EVENT_HUB_NAMESPACE \
--message-retention 1
az eventhubs eventhub authorization-rule create \
```

```
--resource-group $RESOURCE_GROUP \
--name $EVENT_HUB_AUTHORIZATION_RULE \
--eventhub-name $EVENT_HUB_NAME \
--namespace-name $EVENT_HUB_NAMESPACE \
--rights Listen Send
```

The Event Hubs namespace contains the actual event hub and its authorization rule. The authorization rule enables your functions to send messages to the hub and listen for the corresponding events. One function sends messages that represent telemetry data. Another function listens for events, analyzes the event data, and stores the results in Azure Cosmos DB.

Create an Azure Cosmos DB

Next, create an Azure Cosmos DB account, database, and collection using the following commands:

Bash

Azure CLI

```
az cosmosdb create \
--resource-group $RESOURCE_GROUP \
--name $COSMOS_DB_ACCOUNT
az cosmosdb sql database create \
--resource-group $RESOURCE_GROUP \
--account-name $COSMOS_DB_ACCOUNT \
--name TelemetryDb
az cosmosdb sql container create \
--resource-group $RESOURCE_GROUP \
--account-name $COSMOS_DB_ACCOUNT \
--database-name TelemetryDb \
--name TelemetryInfo \
--partition-key-path '/temperatureStatus'
```

The `partition-key-path` value partitions your data based on the `temperatureStatus` value of each item. The partition key enables Azure Cosmos DB to increase performance by dividing your data into distinct subsets that it can access independently.

Create a storage account and function app

Next, create an Azure Storage account, which is required by Azure Functions, then create the function app. Use the following commands:

Bash

Azure CLI

```
az storage account create \
    --resource-group $RESOURCE_GROUP \
    --name $STORAGE_ACCOUNT \
    --sku Standard_LRS
az functionapp create \
    --resource-group $RESOURCE_GROUP \
    --name $FUNCTION_APP \
    --storage-account $STORAGE_ACCOUNT \
    --consumption-plan-location $LOCATION \
    --runtime java \
    --functions-version 3
```

When the `az functionapp create` command creates your function app, it also creates an Application Insights resource with the same name. The function app is automatically configured with a setting named `APPINSIGHTS_INSTRUMENTATIONKEY` that connects it to Application Insights. You can view app telemetry after you deploy your functions to Azure, as described later in this tutorial.

Configure your function app

Your function app will need to access the other resources to work correctly. The following sections show you how to configure your function app so that it can run on your local machine.

Retrieve resource connection strings

Use the following commands to retrieve the storage, event hub, and Azure Cosmos DB connection strings and save them in environment variables:

Bash

Azure CLI

```
AZURE_WEB_JOBS_STORAGE=$( \
    az storage account show-connection-string \
        --name $STORAGE_ACCOUNT \
        --query connectionString \
        --output tsv)
echo $AZURE_WEB_JOBS_STORAGE
EVENT_HUB_CONNECTION_STRING=$( \
```

```
az eventhubs eventhub authorization-rule keys list \
    --resource-group $RESOURCE_GROUP \
    --name $EVENT_HUB_AUTHORIZATION_RULE \
    --eventhub-name $EVENT_HUB_NAME \
    --namespace-name $EVENT_HUB_NAMESPACE \
    --query primaryConnectionString \
    --output tsv)
echo $EVENT_HUB_CONNECTION_STRING
COSMOS_DB_CONNECTION_STRING=$( \
    az cosmosdb keys list \
    --resource-group $RESOURCE_GROUP \
    --name $COSMOS_DB_ACCOUNT \
    --type connection-strings \
    --query 'connectionStrings[0].connectionString' \
    --output tsv)
echo $COSMOS_DB_CONNECTION_STRING
```

These variables are set to values retrieved from Azure CLI commands. Each command uses a JMESPath query to extract the connection string from the JSON payload returned. The connection strings are also displayed using `echo` so you can confirm that they've been retrieved successfully.

Update your function app settings

Next, use the following command to transfer the connection string values to app settings in your Azure Functions account:

Bash

Azure CLI

```
az functionapp config appsettings set \
    --resource-group $RESOURCE_GROUP \
    --name $FUNCTION_APP \
    --settings \
        AzureWebJobsStorage=$AZURE_WEB_JOBS_STORAGE \
        EventHubConnectionString=$EVENT_HUB_CONNECTION_STRING \
        CosmosDBConnectionString=$COSMOS_DB_CONNECTION_STRING
```

Your Azure resources have now been created and configured to work properly together.

Create and test your functions

Next, you'll create a project on your local machine, add Java code, and test it. You'll use commands that work with the Azure Functions Plugin for Maven and the Azure Functions Core Tools. Your functions will run locally, but will use the cloud-based resources you've created. After you get the functions working locally, you can use Maven to deploy them to the cloud and watch your data and analytics accumulate.

If you used Cloud Shell to create your resources, then you won't be connected to Azure locally. In this case, use the `az login` command to launch the browser-based login process. Then if necessary, set the default subscription with `az account set --subscription` followed by the subscription ID. Finally, run the following commands to recreate some environment variables on your local machine. Replace the `<value>` placeholders with the same values you used previously.

Bash

Bash

```
RESOURCE_GROUP=<value>
FUNCTION_APP=<value>
```

Create a local functions project

Use the following Maven command to create a functions project and add the required dependencies.

Bash

```
mvn archetype:generate --batch-mode \
-DarchetypeGroupId=com.microsoft.azure \
-DarchetypeArtifactId=azure-functions-archetype \
-DappName=$FUNCTION_APP \
-DresourceGroup=$RESOURCE_GROUP \
-DappRegion=$LOCATION \
-DgroupId=com.example \
-DartifactId=telemetry-functions
```

This command generates several files inside a `telemetry-functions` folder:

- A `pom.xml` file for use with Maven

- A `local.settings.json` file to hold app settings for local testing
- A `host.json` file that enables the Azure Functions Extension Bundle, required for Azure Cosmos DB output binding in your data analysis function
- A `Function.java` file that includes a default function implementation
- A few test files that this tutorial doesn't need

To avoid compilation errors, you'll need to delete the test files. Run the following commands to navigate to the new project folder and delete the test folder:

```
Bash
Bash
cd telemetry-functions
rm -r src/test
```

Retrieve your function app settings for local use

For local testing, your function project will need the connection strings that you added to your function app in Azure earlier in this tutorial. Use the following Azure Functions Core Tools command, which retrieves all the function app settings stored in the cloud and adds them to your `local.settings.json` file:

```
Bash
Bash
func azure functionapp fetch-app-settings $FUNCTION_APP
```

Add Java code

Next, open the `Function.java` file and replace the contents with the following code.

```
Java
package com.example;

import com.example.TelemetryItem.Status;
import com.microsoft.azure.functions.annotation.Cardinality;
import com.microsoft.azure.functions.annotation.CosmosDBOutput;
import com.microsoft.azure.functions.annotation.EventHubOutput;
```

```
import com.microsoft.azure.functions.annotation.EventHubTrigger;
import com.microsoft.azure.functions.annotation.FunctionName;
import com.microsoft.azure.functions.annotation.TimerTrigger;
import com.microsoft.azure.functions.ExecutionContext;
import com.microsoft.azure.functions.OutputBinding;

public class Function {

    @FunctionName("generateSensorData")
    @EventHubOutput(
        name = "event",
        eventHubName = "", // blank because the value is included in the
    connection string
        connection = "EventHubConnectionString")
    public TelemetryItem generateSensorData(
        @TimerTrigger(
            name = "timerInfo",
            schedule = "*/*/*/*/*") // every 10 seconds
            String timerInfo,
        final ExecutionContext context) {

        context.getLogger().info("Java Timer trigger function executed at: "
            + java.time.LocalDateTime.now());
        double temperature = Math.random() * 100;
        double pressure = Math.random() * 50;
        return new TelemetryItem(temperature, pressure);
    }

    @FunctionName("processSensorData")
    public void processSensorData(
        @EventHubTrigger(
            name = "msg",
            eventHubName = "", // blank because the value is included in the
    connection string
            cardinality = Cardinality.ONE,
            connection = "EventHubConnectionString")
            TelemetryItem item,
        @CosmosDBOutput(
            name = "databaseOutput",
            databaseName = "TelemetryDb",
            containerName = "TelemetryInfo",
            connection = "CosmosDBConnectionString")
            OutputBinding<TelemetryItem> document,
        final ExecutionContext context) {

        context.getLogger().info("Event hub message received: " +
item.toString());

        if (item.getPressure() > 30) {
            item.setNormalPressure(false);
        } else {
            item.setNormalPressure(true);
        }

        if (item.getTemperature() < 40) {
```

```

        item.setTemperatureStatus(status.COOL);
    } else if (item.getTemperature() > 90) {
        item.setTemperatureStatus(status.HOT);
    } else {
        item.setTemperatureStatus(status.WARM);
    }

    document.setValue(item);
}
}

```

As you can see, this file contains two functions, `generateSensorData` and `processSensorData`. The `generateSensorData` function simulates a sensor that sends temperature and pressure readings to the event hub. A timer trigger runs the function every 10 seconds, and an event hub output binding sends the return value to the event hub.

When the event hub receives the message, it generates an event. The `processSensorData` function runs when it receives the event. It then processes the event data and uses an Azure Cosmos DB output binding to send the results to Azure Cosmos DB.

The data used by these functions is stored using a class called `TelemetryItem`, which you'll need to implement. Create a new file called `TelemetryItem.java` in the same location as `Function.java` and add the following code:

Java

```

package com.example;

public class TelemetryItem {

    private String id;
    private double temperature;
    private double pressure;
    private boolean isNormalPressure;
    private status temperatureStatus;
    static enum status {
        COOL,
        WARM,
        HOT
    }

    public TelemetryItem(double temperature, double pressure) {
        this.temperature = temperature;
        this.pressure = pressure;
    }

    public String getId() {

```

```
        return id;
    }

    public double getTemperature() {
        return temperature;
    }

    public double getPressure() {
        return pressure;
    }

    @Override
    public String toString() {
        return "TelemetryItem{id=" + id + ",temperature="
            + temperature + ",pressure=" + pressure + "}";
    }

    public boolean isNormalPressure() {
        return isNormalPressure;
    }

    public void setNormalPressure(boolean isNormal) {
        this.isNormalPressure = isNormal;
    }

    public status getTemperatureStatus() {
        return temperatureStatus;
    }

    public void setTemperatureStatus(status temperatureStatus) {
        this.temperatureStatus = temperatureStatus;
    }
}
```

Run locally

You can now build and run the functions locally and see data appear in your Azure Cosmos DB.

Use the following Maven commands to build and run the functions:

Bash

Bash

```
mvn clean package
mvn azure-functions:run
```

After some build and startup messages, you'll see output similar to the following example for each time the functions run:

```
Output

[10/22/19 4:01:30 AM] Executing 'Functions.generateSensorData'
(Reason='Timer fired at 2019-10-21T21:01:30.0016769-07:00', Id=c1927c7f-
4f70-4a78-83eb-bc077d838410)
[10/22/19 4:01:30 AM] Java Timer trigger function executed at: 2019-10-
21T21:01:30.015
[10/22/19 4:01:30 AM] Function "generateSensorData" (Id: c1927c7f-4f70-4a78-
83eb-bc077d838410) invoked by Java Worker
[10/22/19 4:01:30 AM] Executed 'Functions.generateSensorData' (Succeeded,
Id=c1927c7f-4f70-4a78-83eb-bc077d838410)
[10/22/19 4:01:30 AM] Executing 'Functions.processSensorData' (Reason='',
Id=f4c3b4d7-9576-45d0-9c6e-85646bb52122)
[10/22/19 4:01:30 AM] Event hub message received: TelemetryItem=
{id=null,temperature=32.728691307527015,pressure=10.122563042388165}
[10/22/19 4:01:30 AM] Function "processSensorData" (Id: f4c3b4d7-9576-45d0-
9c6e-85646bb52122) invoked by Java Worker
[10/22/19 4:01:38 AM] Executed 'Functions.processSensorData' (Succeeded,
Id=1cf0382b-0c98-4cc8-9240-ee2a2f71800d)
```

You can then go to the [Azure portal](#) and navigate to your Azure Cosmos DB account. Select **Data Explorer**, expand **TelemetryInfo**, then select **Items** to view your data when it arrives.

The screenshot shows the Microsoft Azure Data Explorer interface for the 'weatherdata123 - Data Explorer' account. The left sidebar lists various monitoring and management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The 'Data Explorer' option is highlighted with a red box. The main area shows the 'SQL API' interface for the 'TelemetryDb' database. Under 'TelemetryInfo', the 'Items' collection is selected, also highlighted with a red box. A table titled 'Items' displays several rows of sensor data. One row is expanded to show its JSON structure:

id	/temp...
75964b75...	WARM
d1bbc4f2...	HOT
5b75dad6...	WARM
3d5e7408...	WARM
76dd38f1...	COOL
38da208b...	COOL

Row 12 (highlighted) shows the expanded JSON data:

```
1 "temperature": 47.967774187747615,
2 "pressure": 19.179577577377465,
3 "isNormalPressure": true,
4 "temperatureStatus": "WARM",
5 "id": "3d5e7408-4536-40c1-ab49-5cc2b92
6 "_rid": "3fRaAK+T9gIEAAAAAAA==",
7 "_self": " dbs/3fRaAK+/colls/3fRaAK+T9
8 "_etag": "3002e2ef-0000-0700-0000-5c
9 "_attachments": "attachments/",
10 "_ts": 1572063509
```

Deploy to Azure and view app telemetry

Finally, you can deploy your app to Azure and verify that it continues to work the same way it did locally.

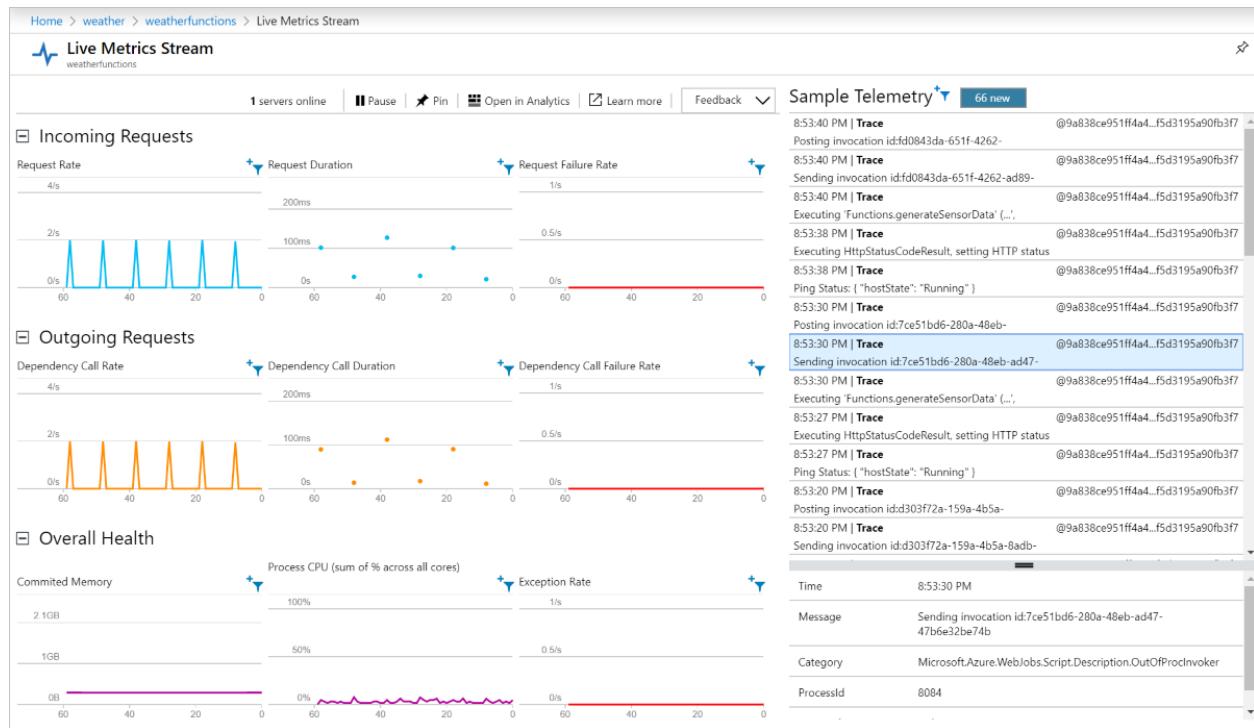
Deploy your project to Azure using the following command:



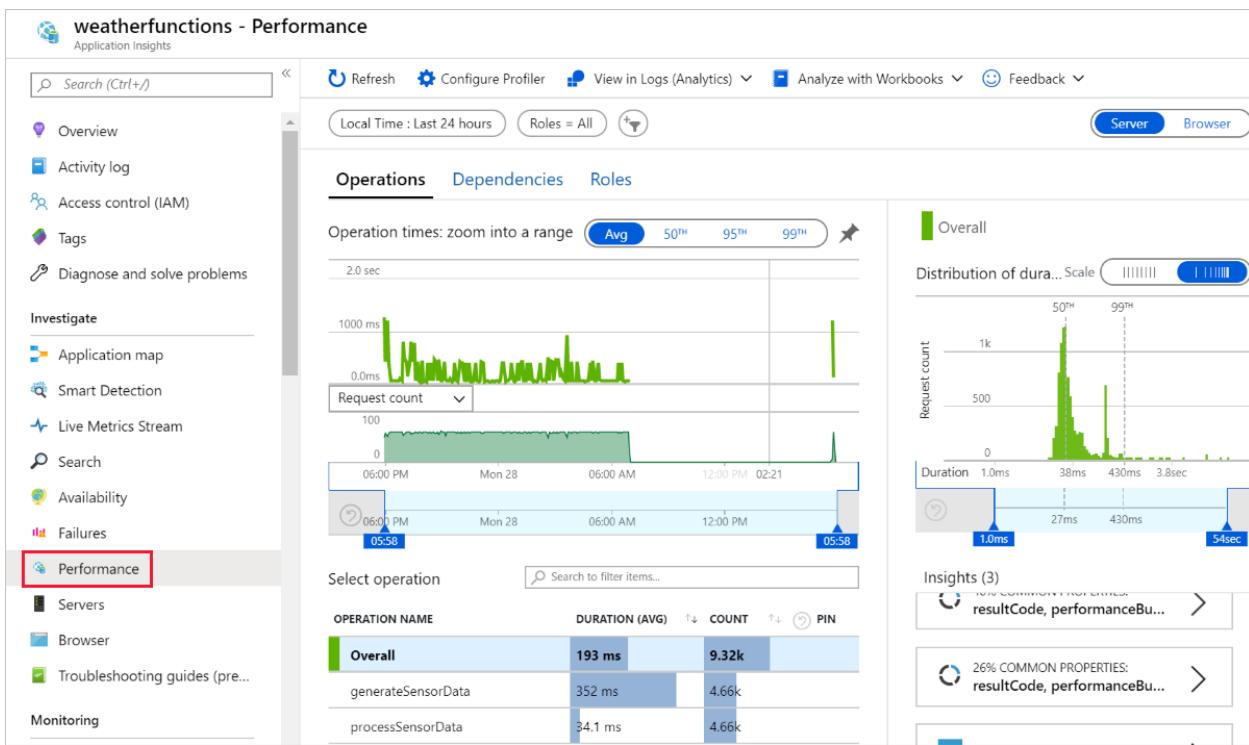
```
Bash
Bash
mvn azure-functions:deploy
```

Your functions now run in Azure, and continue to accumulate data in your Azure Cosmos DB. You can view your deployed function app in the Azure portal, and view app telemetry through the connected Application Insights resource, as shown in the following screenshots:

Live Metrics Stream:



Performance:



Clean up resources

When you're finished with the Azure resources you created in this tutorial, you can delete them using the following command:

```
Bash
```

```
Azure CLI
```

```
az group delete --name $RESOURCE_GROUP
```

Next steps

In this tutorial, you learned how to create an Azure Function that handles Event Hub events and updates an Azure Cosmos DB instance. For more information, see the [Azure Functions Java developer guide](#). For information on the annotations used, see the [com.microsoft.azure.functions.annotation](#) reference.

This tutorial used environment variables and application settings to store secrets such as connection strings. For information on storing these secrets in Azure Key Vault, see [Use Key Vault references for App Service and Azure Functions](#).

Next, learn how to use Azure Pipelines CI/CD for automated deployment:

[Build and deploy Java to Azure Functions](#)

Azure CLI Samples

Article • 04/22/2024

These end-to-end Azure CLI scripts are provided to help you learn how to provision and managing the Azure resources required by Azure Functions. You must use the [Azure Functions Core Tools](#) to create actual Azure Functions code projects from the command line on your local computer and deploy code to these Azure resources. For a complete end-to-end example of developing and deploying from the command line using both Core Tools and the Azure CLI, see one of these language-specific command line quickstarts:

- [C#](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)
- [TypeScript](#)

The following table includes links to bash scripts that you can use to create and manage the Azure resources required by Azure Functions using the Azure CLI.

[+] Expand table

Create app	Description
Create a function app for serverless execution	Create a function app in a Consumption plan.
Create a serverless Python function app	Create a Python function app in a Consumption plan.
Create a function app in a scalable Premium plan	Create a function app in a Premium plan.
Create a function app in a dedicated (App Service) plan	Create a function app in a dedicated App Service plan.

[+] Expand table

Integrate	Description
Create a function app and connect to a storage account	Create a function app and connect it to a storage account.

Integrate	Description
Create a function app and connect to an Azure Cosmos DB	Create a function app and connect it to an Azure Cosmos DB instance.
Create a Python function app and mount an Azure Files share	By mounting a share to your Linux function app, you can leverage existing machine learning models or other data in your functions.

[\[\]](#) Expand table

Continuous deployment	Description
Deploy from GitHub	Create a function app that deploys from a GitHub repository.

Create a function app for serverless code execution

Article • 01/13/2023

This Azure Functions sample script creates a function app, which is a container for your functions. The function app is created using the [Consumption plan](#), which is ideal for event-driven serverless workloads.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon on the left, the text 'Launch Cloud Shell' in the center, and a small blue square with a white arrow icon on the right.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run [`az upgrade`](#).

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.

# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="eastus"
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="create-function-app-consumption"
storage="msdocsaccount$randomIdentifier"
functionApp="msdocs-serverless-function-$randomIdentifier"
skuStorage="Standard_LRS"
functionsVersion="4"

# Create a resource group
echo "Creating $resourceGroup in \"$location\"..."
az group create --name $resourceGroup --location "$location" --tags $tag

# Create an Azure storage account in the resource group.
echo "Creating $storage"
az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage
```

```
# Create a serverless function app in the resource group.  
echo "Creating $functionApp"  
az functionapp create --name $functionApp --storage-account $storage --  
consumption-plan-location "$location" --resource-group $resourceGroup --  
functions-version $functionsVersion
```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the [az group delete](#) command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp create	Creates a function app.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create a serverless Python function app using Azure CLI

Article • 01/13/2023

This Azure Functions sample script creates a function app, which is a container for your functions. This script creates an Azure Function app using the [Consumption plan](#).

ⓘ Note

The function app created runs on Python version 3.9. Python version 3.7 and 3.8 are also supported by Azure Functions.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon on the left and the text 'Launch Cloud Shell' in white. To the right of the text is a small blue square with a white right-pointing arrow.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

Sample script

[Launch Azure Cloud Shell](#)

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select Try it from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI
```

```
subscription=<subscriptionId> # add subscription here  
az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI
```

```
# Function app and storage account names must be unique.  
  
# Variable block  
let "randomIdentifier=$RANDOM*$RANDOM"  
location="eastus"  
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"  
tag="create-function-app-consumption-python"  
storage="msdocsaccount$randomIdentifier"  
functionApp="msdocs-serverless-python-function-$randomIdentifier"  
skuStorage="Standard_LRS"  
functionsVersion="4"  
pythonVersion="3.9" #Allowed values: 3.7, 3.8, and 3.9  
  
# Create a resource group  
echo "Creating $resourceGroup in \"$location\"..."  
az group create --name $resourceGroup --location "$location" --tags $tag
```

```
# Create an Azure storage account in the resource group.  
echo "Creating $storage"  
az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage  
  
# Create a serverless python function app in the resource group.  
echo "Creating $functionApp"  
az functionapp create --name $functionApp --storage-account $storage --consumption-plan-location "$location" --resource-group $resourceGroup --os-type Linux --runtime python --runtime-version $pythonVersion --functions-version $functionsVersion
```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the [az group delete](#) command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp create	Creates a function app.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create a function app in a Premium plan - Azure CLI

Article • 01/13/2023

This Azure Functions sample script creates a function app, which is a container for your functions. The function app that is created uses a [scalable Premium plan](#).

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square icon with a white arrow pointing outwards.
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square icon with a white arrow pointing outwards.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select Try it from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to

<https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment. Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Azure CLI

```
subscription=<subscriptionId> # add subscription here  
az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

Azure CLI

```
# Function app and storage account names must be unique.  
  
# Variable block  
let "randomIdentifier=$RANDOM*$RANDOM"  
location="eastus"  
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"  
tag="create-function-app-premium-plan"  
storage="msdocsaccount$randomIdentifier"  
premiumPlan="msdocs-premium-plan-$randomIdentifier"  
functionApp="msdocs-function-$randomIdentifier"  
skuStorage="Standard_LRS" # Allowed values: Standard_LRS, Standard_GRS,  
Standard_RAGRS, Standard_ZRS, Premium_LRS, Premium_ZRS, Standard_GZRS,  
Standard_RAGZRS  
skuPlan="EP1"  
functionsVersion="4"  
  
# Create a resource group  
echo "Creating $resourceGroup in \"$location\"..."  
az group create --name $resourceGroup --location "$location" --tags $tag  
  
# Create an Azure storage account in the resource group.  
echo "Creating $storage"
```

```

az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage

# Create a Premium plan
echo "Creating $premiumPlan"
az functionapp plan create --name $premiumPlan --resource-group $resourceGroup --location "$location" --sku $skuPlan

# Create a Function App
echo "Creating $functionApp"
az functionapp create --name $functionApp --storage-account $storage --plan $premiumPlan --resource-group $resourceGroup --functions-version $functionsVersion

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the `az group delete` command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp plan create	Creates a Premium plan in a specific SKU .
az functionapp create	Creates a function app in the App Service plan.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create a Function App in an App Service plan

Article • 01/13/2023

This Azure Functions sample script creates a function app, which is a container for your functions. The function app that is created uses a dedicated App Service plan, which means your server resources are always on.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square with a white right-pointing arrow.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run [`az upgrade`](#).

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.

# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="eastus"
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="create-function-app-consumption"
storage="msdocsaccount$randomIdentifier"
appServicePlan="msdocs-app-service-plan-$randomIdentifier"
functionApp="msdocs-serverless-function-$randomIdentifier"
skuStorage="Standard_LRS"
skuPlan="B1"
functionsVersion="4"

# Create a resource group
echo "Creating $resourceGroup in \"$location\"..."
az group create --name $resourceGroup --location "$location" --tags $tag

# Create an Azure storage account in the resource group.
echo "Creating $storage"
```

```

az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage

# Create an App Service plan
echo "Creating $appServicePlan"
az functionapp plan create --name $appServicePlan --resource-group $resourceGroup --location "$location" --sku $skuPlan

# Create a Function App
echo "Creating $functionApp"
az functionapp create --name $functionApp --storage-account $storage --plan $appServicePlan --resource-group $resourceGroup --functions-version $functionsVersion

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the `az group delete` command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp plan create	Creates a Premium plan.
az functionapp create	Creates a function app in the App Service plan.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create a function app with a named Storage account connection

Article • 01/13/2023

This Azure Functions sample script creates a function app and connects the function to an Azure Storage account. The created app setting that contains the storage connection string can be used with a [storage trigger or binding](#).

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square with a white right-pointing arrow.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run [`az upgrade`](#).

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.

# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="eastus"
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="create-function-app-connect-to-storage-account"
storage="msdocsaccount$randomIdentifier"
functionApp="msdocs-serverless-function-$randomIdentifier"
skuStorage="Standard_LRS"
functionsVersion="4"

# Create a resource group
echo "Creating $resourceGroup in \"$location\"..."
az group create --name $resourceGroup --location "$location" --tags $tag

# Create an Azure storage account in the resource group.
echo "Creating $storage"
az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage
```

```

# Create a serverless function app in the resource group.
echo "Creating $functionApp"
az functionapp create --name $functionApp --resource-group $resourceGroup --
storage-account $storage --consumption-plan-location "$location" --
functions-version $functionsVersion

# Get the storage account connection string.
connstr=$(az storage account show-connection-string --name $storage --
resource-group $resourceGroup --query connectionString --output tsv)

# Update function app settings to connect to the storage account.
az functionapp config appsettings set --name $functionApp --resource-group
$resourceGroup --settings StorageConStr=$connstr

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the [az group delete](#) command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

This script uses the following commands. Each command in the table links to command specific documentation.

Command	Notes
az group create	Create a resource group with location.
az storage account create	Create a storage account.
az functionapp create	Creates a function app in the serverless Consumption plan .
az storage account show-connection-string	Gets the connection string for the account.
az functionapp config appsettings set	Sets the connection string as an app setting in the function app.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create an Azure Function that connects to an Azure Cosmos DB

Article • 01/13/2023

This Azure Functions sample script creates a function app and connects the function to an Azure Cosmos DB database. It makes the connection using an Azure Cosmos DB endpoint and access key that it adds to app settings. The created app setting that contains the connection can be used with an [Azure Cosmos DB trigger or binding](#).

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the text is a small blue square with a white right-pointing arrow.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.

# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="eastus"
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="create-function-app-connect-to-cosmos-db"
storage="msdocsaccount$randomIdentifier"
functionApp="msdocs-serverless-function-$randomIdentifier"
skuStorage="Standard_LRS"
functionsVersion="4"

# Create a resource group
echo "Creating $resourceGroup in \"$location\"..."
az group create --name $resourceGroup --location "$location" --tags $tag

# Create a storage account for the function app.
echo "Creating $storage"
az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage
```

```

# Create a serverless function app in the resource group.
echo "Creating $functionApp"
az functionapp create --name $functionApp --resource-group $resourceGroup --
storage-account $storage --consumption-plan-location "$location" --
functions-version $functionsVersion

# Create an Azure Cosmos DB database account using the same function app
# name.
echo "Creating $functionApp"
az cosmosdb create --name $functionApp --resource-group $resourceGroup

# Get the Azure Cosmos DB connection string.
endpoint=$(az cosmosdb show --name $functionApp --resource-group
$resourceGroup --query documentEndpoint --output tsv)
echo $endpoint

key=$(az cosmosdb keys list --name $functionApp --resource-group
$resourceGroup --query primaryMasterKey --output tsv)
echo $key

# Configure function app settings to use the Azure Cosmos DB connection
# string.
az functionapp config appsettings set --name $functionApp --resource-group
$resourceGroup --setting CosmosDB_Endpoint=$endpoint CosmosDB_Key=$key

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the [az group delete](#) command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Command	Notes
az group create	Create a resource group with location
az storage accounts create	Create a storage account
az functionapp create	Creates a function app in the serverless Consumption plan .
az cosmosdb create	Create an Azure Cosmos DB database.

Command	Notes
<code>az cosmosdb show</code>	Gets the database account connection.
<code>az cosmosdb list-keys</code>	Gets the keys for the database.
<code>az functionapp config appsettings set</code>	Sets the connection string as an app setting in the function app.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

More Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Mount a file share to a Python function app using Azure CLI

Article • 01/13/2023

This Azure Functions sample script creates a function app using the [Consumption plan](#) and creates a share in Azure Files. It then mounts the share so that the data can be accessed by your functions.

ⓘ Note

The function app created runs on Python version 3.9. Azure Functions also [supports Python versions 3.7 and 3.8](#).

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
[A Launch Cloud Shell](#)
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment. Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.

# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="eastus"
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="functions-cli-mount-files-storage-linux"
export AZURE_STORAGE_ACCOUNT="msdocsstorage$randomIdentifier"
functionApp="msdocs-serverless-function-$randomIdentifier"
skuStorage="Standard_LRS"
functionsVersion="4"
pythonVersion="3.9" #Allowed values: 3.7, 3.8, and 3.9
share="msdocs-fileshare-$randomIdentifier"
directory="msdocs-directory-$randomIdentifier"
```

```

shareId="msdocs-share-$randomIdentifier"
mountPath="/mounted-$randomIdentifier"

# Create a resource group
echo "Creating $resourceGroup in \"$location\"..."
az group create --name $resourceGroup --location "$location" --tags $tag

# Create an Azure storage account in the resource group.
echo "Creating $AZURE_STORAGE_ACCOUNT"
az storage account create --name $AZURE_STORAGE_ACCOUNT --location
"$location" --resource-group $resourceGroup --sku $skuStorage

# Set the storage account key as an environment variable.
export AZURE_STORAGE_KEY=$(az storage account keys list -g $resourceGroup -n
$AZURE_STORAGE_ACCOUNT --query '[0].value' -o tsv)

# Create a serverless function app in the resource group.
echo "Creating $functionApp"
az functionapp create --name $functionApp --storage-account
$AZURE_STORAGE_ACCOUNT --consumption-plan-location "$location" --resource-
group $resourceGroup --os-type Linux --runtime python --runtime-version
$pythonVersion --functions-version $functionsVersion

# Work with Storage account using the set env variables.
# Create a share in Azure Files.
echo "Creating $share"
az storage share create --name $share

# Create a directory in the share.
echo "Creating $directory in $share"
az storage directory create --share-name $share --name $directory

# Create webapp config storage account
echo "Creating $AZURE_STORAGE_ACCOUNT"
az webapp config storage-account add \
--resource-group $resourceGroup \
--name $functionApp \
--custom-id $shareId \
--storage-type AzureFiles \
--share-name $share \
--account-name $AZURE_STORAGE_ACCOUNT \
--mount-path $mountPath \
--access-key $AZURE_STORAGE_KEY

# List webapp storage account
az webapp config storage-account list --resource-group $resourceGroup --name
$functionApp

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the `az group delete` command - unless you have an ongoing need for these

resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp create	Creates a function app.
az storage share create	Creates an Azure Files share in storage account.
az storage directory create	Creates a directory in the share.
az webapp config storage-account add	Mounts the share to the function app.
az webapp config storage-account list	Shows file shares mounted to the function app.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create a function app in Azure that is deployed from GitHub

Article • 01/13/2023

This Azure Functions sample script creates a function app using the [Consumption plan](#), along with its related resources. The script also configures your function code for continuous deployment from a public GitHub repository. There is also commented out code for using a private GitHub repository.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the text is a small blue square with a white right-pointing arrow.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.
let "randomIdentifier=$RANDOM*$RANDOM"
location=eastus
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="deploy-function-app-with-function-github"
storage="msdocs$randomIdentifier"
skuStorage="Standard_LRS"
functionApp=mygithubfunc$randomIdentifier
functionsVersion="4"
runtime="node"
# Public GitHub repository containing an Azure Functions code project.
gitrepo=https://github.com/Azure-Samples/functions-quickstart-javascript
## Enable authenticated git deployment in your subscription when using a
private repo.
#token=<Replace with a GitHub access token when using a private repo.>
#az functionapp deployment source update-token \
# --git-token $token

# Create a resource group.
```

```

echo "Creating $resourceGroup in """$location""""...
az group create --name $resourceGroup --location "$location" --tags $tag

# Create an Azure storage account in the resource group.
echo "Creating $storage"
az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage

# Create a function app with source files deployed from the specified GitHub repo.
echo "Creating $functionApp"
az functionapp create --name $functionApp --storage-account $storage --consumption-plan-location "$location" --resource-group $resourceGroup --deployment-source-url $gitrepo --deployment-source-branch main --functions-version $functionsVersion --runtime $runtime

# Connect to function application
curl -s "https://$functionApp.azurewebsites.net/api/httpexample?name=Azure"

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the [az group delete](#) command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates the storage account required by the function app.
az functionapp create	Creates a function app in the serverless Consumption plan and associates it with a Git or Mercurial repository.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create function app resources in Azure using PowerShell

Article • 05/02/2023

The Azure PowerShell example scripts in this article create function apps and other resources required to host your functions in Azure. A function app provides an execution context in which your functions are executed. All functions running in a function app share the same resources and connections, and they're all scaled together.

After the resources are created, you can deploy your project files to the new function app. To learn more, see [Deployment methods](#).

Every function app requires your PowerShell scripts to create the following resources:

Resource	cmdlet	Description
Resource group	<code>New-AzResourceGroup</code>	Creates a resource group in which you'll create your function app.
Storage account	<code>New-AzStorageAccount</code>	Creates a storage account used by your function app. Storage account names must be between 3 and 24 characters in length and can contain numbers and lowercase letters only. You can also use an existing account, which must meet the storage account requirements .
App Service plan	<code>New-AzFunctionAppPlan</code>	Explicitly creates a hosting plan, which defines how resources are allocated to your function app. Used only when hosting in a Premium or Dedicated plan. You won't use this cmdlet when hosting in a serverless Consumption plan , since Consumption plans are created when you run <code>New-AzFunctionApp</code> . For more information, see Azure Functions hosting options .
Function app	<code>New-AzFunctionApp</code>	Creates the function app using the required resources. The <code>-Name</code> parameter must be a globally unique name across all of Azure App Service. Valid characters in <code>-Name</code> are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> . Most examples create a function app that supports C# functions. You can change the language by using the <code>-Runtime</code> parameter, with supported values of <code>DotNet</code> , <code>Java</code> , <code>Node</code> , <code>PowerShell</code> , and <code>Python</code> . Use the <code>-RuntimeVersion</code> to choose a specific language version .

This article contains the following examples:

- [Create a serverless function app for C#](#)
- [Create a serverless function app for Python](#)

- Create a scalable function app in a Premium plan
- Create a function app in a Dedicated plan
- Create a function app with a named Storage connection
- Create a function app with an Azure Cosmos DB connection
- Create a function app with continuous deployment
- Create a serverless Python function app and mount file share

Prerequisites

- If you choose to use Azure PowerShell locally:
 - [Install the Az PowerShell module.](#)
 - Connect to your Azure account using the [Connect-AzAccount](#) cmdlet.
- If you choose to use Azure Cloud Shell:
 - See [Overview of Azure Cloud Shell](#) for more information.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Create a serverless function app for C#

The following script creates a serverless C# function app in the default Consumption plan:

```
azurepowershell-interactive

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
$location = "eastus"
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "create-function-app-consumption"}
$storage = "msdocsaccount$randomIdentifier"
$functionApp = "msdocs-serverless-function-$randomIdentifier"
$skuStorage = "Standard_LRS"
$functionsVersion = "4"

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage
```

```
# Create a serverless function app in the resource group.  
Write-Host "Creating $functionApp"  
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -Location  
$location -ResourceGroupName $resourceGroup -Runtime DotNet -  
FunctionsVersion $functionsVersion
```

Create a serverless function app for Python

The following script creates a serverless Python function app in a Consumption plan:

```
azurepowershell-interactive  
  
# Function app and storage account names must be unique.  
  
# Variable block  
$randomIdentifier = Get-Random  
$location = "eastus"  
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"  
$tag = @{script = "create-function-app-consumption-python"}  
$storage = "msdocsaccount$randomIdentifier"  
$functionApp = "msdocs-serverless-python-function-$randomIdentifier"  
$skuStorage = "Standard_LRS"  
$functionsVersion = "4"  
$pythonVersion = "3.9" #Allowed values: 3.7, 3.8, and 3.9  
  
# Create a resource group  
Write-Host "Creating $resourceGroup in $location..."  
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag  
  
# Create an Azure storage account in the resource group.  
Write-Host "Creating $storage"  
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName  
$resourceGroup -SkuName $skuStorage  
  
# Create a serverless Python function app in the resource group.  
Write-Host "Creating $functionApp"  
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -Location  
$location -ResourceGroupName $resourceGroup -OSType Linux -Runtime Python -  
RuntimeVersion $pythonVersion -FunctionsVersion $functionsVersion
```

Create a scalable function app in a Premium plan

The following script creates a C# function app in an Elastic Premium plan that supports [dynamic scale](#):

```
azurepowershell-interactive
```

```

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
.setLocation = "eastus"
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "create-function-app-premium-plan"}
$storage = "msdocsaccount$randomIdentifier"
$premiumPlan = "msdocs-premium-plan-$randomIdentifier"
$functionApp = "msdocs-function-$randomIdentifier"
$skuStorage = "Standard_LRS" # Allowed values: Standard_LRS, Standard_GRS,
Standard_RAGRS, Standard_ZRS, Premium_LRS, Premium_ZRS, Standard_GZRS,
Standard_RAGZRS
$skuPlan = "EP1"
$functionsVersion = "4"

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage

# Create a Premium plan
Write-Host "Creating $premiumPlan"
New-AzFunctionAppPlan -Name $premiumPlan -ResourceGroupName $resourceGroup -
Location $location -Sku $skuPlan -WorkerType Windows

# Create a Function App
Write-Host "Creating $functionApp"
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -PlanName
$premiumPlan -ResourceGroupName $resourceGroup -Runtime DotNet -
FunctionsVersion $functionsVersion

```

Create a function app in a Dedicated plan

The following script creates a function app hosted in a Dedicated plan, which isn't scaled dynamically by Functions:

azurepowershell-interactive

```

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
.setLocation = "eastus"
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "create-function-app-app-service-plan"}
$storage = "msdocsaccount$randomIdentifier"

```

```

$appServicePlan = "msdocs-app-service-plan-$randomIdentifier"
$functionApp = "msdocs-serverless-function-$randomIdentifier"
$skuStorage = "Standard_LRS"
$skuPlan = "B1"
$functionsVersion = "4"

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage

# Create an App Service plan
Write-Host "Creating $appServicePlan"
New-AzFunctionAppPlan -Name $appServicePlan -ResourceGroupName
$resourceGroup -Location $location -Sku $skuPlan -WorkerType Windows

# Create a Function App
Write-Host "Creating $functionApp"
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -PlanName
$appServicePlan -ResourceGroupName $resourceGroup -Runtime DotNet -
FunctionsVersion $functionsVersion

```

Create a function app with a named Storage connection

The following script creates a function app with a named Storage connection in application settings:

azurepowershell-interactive

```

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
$location = "eastus"
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "create-function-app-connect-to-storage-account"}
$storage = "msdocsaccount$randomIdentifier"
$functionApp = "msdocs-serverless-function-$randomIdentifier"
$skuStorage = "Standard_LRS"
$functionsVersion = "4"

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

```

```

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage

# Create a serverless function app in the resource group.
Write-Host "Creating $functionApp"
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -Location
$location -ResourceGroupName $resourceGroup -Runtime DotNet -
FunctionsVersion $functionsVersion

# Get the storage account connection string.
$connstr = (Get-AzStorageAccount -StorageAccountName $storage -
ResourceGroupName $resourceGroup).Context.ConnectionString

# Update function app settings to connect to the storage account.
Update-AzFunctionAppSetting -Name $functionApp -ResourceGroupName
$resourceGroup -AppSetting @{StorageConStr = $connstr}

```

Create a function app with an Azure Cosmos DB connection

The following script creates a function app and a connected Azure Cosmos DB account:

azurepowershell-interactive

```

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
.setLocation = "eastus"
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "create-function-app-connect-to-cosmos-db"}
$storage = "msdocsaccount$randomIdentifier"
$functionApp = "msdocs-serverless-function-$randomIdentifier"
$skuStorage = "Standard_LRS"
$functionsVersion = "4"

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage

# Create a serverless function app in the resource group.
Write-Host "Creating $functionApp"
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -Location
$location -ResourceGroupName $resourceGroup -Runtime DotNet -

```

```

FunctionsVersion $functionsVersion

# Create an Azure Cosmos DB database account using the same function app
# name.
Write-Host "Creating $functionApp"
New-AzCosmosDBAccount -Name $functionApp -ResourceGroupName $resourceGroup -
Location $location

# Get the Azure Cosmos DB connection string.
$endpoint = (Get-AzCosmosDBAccount -Name $functionApp -ResourceGroupName
$resourceGroup).DocumentEndpoint
Write-Host $endpoint

$key = (Get-AzCosmosDBAccountKey -Name $functionApp -ResourceGroupName
$resourceGroup).PrimaryMasterKey
Write-Host $key

# Configure function app settings to use the Azure Cosmos DB connection
# string.
Update-AzFunctionAppSetting -Name $functionApp -ResourceGroupName
$resourceGroup -AppSetting @{CosmosDB_Endpoint = $endpoint; CosmosDB_Key =
$key}

```

Create a function app with continuous deployment

The following script creates a function app that has continuous deployment configured to publish from a public GitHub repository:

```

azurepowershell-interactive

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
$location = "eastus"
$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "deploy-function-app-with-function-github"}
$storage = "msdocsaccount$randomIdentifier"
$functionApp = "mygithubfunc$randomIdentifier"
$skuStorage = "Standard_LRS"
$functionsVersion = "4"
$runtime = "Node"
# Public GitHub repository containing an Azure Functions code project.
$gitrepo = "https://github.com/Azure-Samples/functions-quickstart-
javascript"
<# Set GitHub personal access token (PAT) to enable authenticated GitHub
deployment in your subscription when using a private repo.
$token = <Replace with a GitHub access token when using a private repo.>
$propertiesObject = @{

```

```

        token = $token
    }

Set-AzResource -PropertyObject $propertiesObject -ResourceId
/providers/Microsoft.Web/sourcecontrols/GitHub -ApiVersion 2018-02-01 -Force
#>

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage

# Create a function app in the resource group.
Write-Host "Creating $functionApp"
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -Location
$location -ResourceGroupName $resourceGroup -Runtime $runtime -
FunctionsVersion $functionsVersion

# Configure GitHub deployment from a public GitHub repo and deploy once.
$propertiesObject = @{
    repoUrl = $gitrepo
    branch = 'main'
    isManualIntegration = $True # $False when using a private repo
}

Set-AzResource -PropertyObject $propertiesObject -ResourceGroupName
$resourceGroup -ResourceType Microsoft.Web/sites/sourcecontrols -
ResourceName $functionApp/web -ApiVersion 2018-02-01 -Force

# Connect to function application
Invoke-RestMethod -Uri
"https://$functionApp.azurewebsites.net/api/httpexample?name=Azure"

```

Create a serverless Python function app and mount file share

The following script creates a Python function app on Linux and creates and mounts an external Azure Files share:

azurepowershell-interactive

```

# Function app and storage account names must be unique.

# Variable block
$randomIdentifier = Get-Random
$location = "eastus"

```

```

$resourceGroup = "msdocs-azure-functions-rg-$randomIdentifier"
$tag = @{script = "functions-cli-mount-files-storage-linux"}
$storage = "msdocsaccount$randomIdentifier"
$functionApp = "msdocs-serverless-function-$randomIdentifier"
$skuStorage = "Standard_LRS"
$functionsVersion = "4"
$pythonVersion = "3.9" #Allowed values: 3.7, 3.8, and 3.9
$share = "msdocs-fileshare-$randomIdentifier"
$directory = "msdocs-directory-$randomIdentifier"
$shareId = "msdocs-share-$randomIdentifier"
$mountPath = "/mounted-$randomIdentifier"

# Create a resource group
Write-Host "Creating $resourceGroup in $location..."
New-AzResourceGroup -Name $resourceGroup -Location $location -Tag $tag

# Create an Azure storage account in the resource group.
Write-Host "Creating $storage"
New-AzStorageAccount -Name $storage -Location $location -ResourceGroupName
$resourceGroup -SkuName $skuStorage

# Get the storage account key.
$keys = Get-AzStorageAccountKey -Name $storage -ResourceGroupName
$resourceGroup
$storageKey = $keys[0].Value

## Create a serverless Python function app in the resource group.
Write-Host "Creating $functionApp"
New-AzFunctionApp -Name $functionApp -StorageAccountName $storage -Location
$location -ResourceGroupName $resourceGroup -OSType Linux -Runtime Python -
RuntimeVersion $pythonVersion -FunctionsVersion $functionsVersion

# Create a share in Azure Files.
Write-Host "Creating $share"
$storageContext = New-AzStorageContext -StorageAccountName $storage -
StorageAccountKey $storageKey
New-AzStorageShare -Name $share -Context $storageContext

# Create a directory in the share.
Write-Host "Creating $directory in $share"
New-AzStorageDirectory -ShareName $share -Path $directory -Context
$storageContext

# Add a storage account configuration to the function app
Write-Host "Adding $storage configuration"
$storagePath = New-AzWebAppAzureStoragePath -Name $shareId -Type AzureFiles
-ShareName $share -AccountName $storage -MountPath $mountPath -AccessKey
$storageKey
Set-AzWebApp -Name $functionApp -ResourceGroupName $resourceGroup -
AzureStoragePath $storagePath

# Get a function app's storage account configurations.
(Get-AzWebApp -Name $functionApp -ResourceGroupName
$resourceGroup).AzureStoragePath

```

Mounted file shares are only supported on Linux. For more information, see [Mount file shares](#).

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, delete the resource group by running the following PowerShell command:

Azure CLI

```
Remove-AzResourceGroup -Name myResourceGroup
```

This command may take a minute to run.

Next steps

For more information on Azure PowerShell, see [Azure PowerShell documentation](#).

Best practices for reliable Azure Functions

Article • 07/12/2022

Azure Functions is an event-driven, compute-on-demand experience that extends the existing Azure App Service application platform with capabilities to implement code triggered by events occurring in Azure, in third-party service, and in on-premises systems. Functions lets you build solutions by connecting to data sources or messaging solutions, which makes it easier to process and react to events. Functions runs on Azure data centers, which are complex with many integrated components. In a hosted cloud environment, it's expected that VMs can occasionally restart or move, and systems upgrades will occur. Your functions apps also likely depend on external APIs, Azure Services, and other databases, which are also prone to periodic unreliability.

This article details some best practices for designing and deploying efficient function apps that remain healthy and perform well in a cloud-based environment.

Choose the correct hosting plan

When you create a function app in Azure, you must choose a hosting plan for your app. The plan you choose has an effect on performance, reliability, and cost. There are three basic hosting plans available for Functions:

- [Consumption plan](#)
- [Premium plan](#)
- [Dedicated \(App Service\) plan](#)

All hosting plans are generally available (GA) when running either Linux or Windows.

In the context of the App Service platform, the Premium plan used to dynamically host your functions is the Elastic Premium plan (EP). There are other Dedicated (App Service) plans called Premium. To learn more, see the [Premium plan](#) article.

The hosting plan you choose determines the following behaviors:

- How your function app is scaled based on demand and how instances allocation is managed.
- The resources available to each function app instance.
- Support for advanced functionality, such as Azure Virtual Network connectivity.

To learn more about choosing the correct hosting plan and for a detailed comparison between the plans, see [Azure Functions hosting options](#).

It's important that you choose the correct plan when you create your function app. Functions provides a limited ability to switch your hosting plan, primarily between Consumption and Elastic Premium plans. To learn more, see [Plan migration](#).

Configure storage correctly

Functions requires a storage account be associated with your function app. The storage account connection is used by the Functions host for operations such as managing triggers and logging function executions. It's also used when dynamically scaling function apps. To learn more, see [Storage considerations for Azure Functions](#).

A misconfigured file system or storage account in your function app can affect the performance and availability of your functions. For help with troubleshooting an incorrectly configured storage account, see the [storage troubleshooting](#) article.

Storage connection settings

Function apps that scale dynamically can run either from an Azure Files endpoint in your storage account or from the file servers associated with your scaled-out instances. This behavior is controlled by the following application settings:

- `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`
- `WEBSITE_CONTENTSHARE`

These settings are only supported when you are running in a Premium plan or in a Consumption plan on Windows.

When you create your function app either in the Azure portal or by using Azure CLI or Azure PowerShell, these settings are created for your function app when needed. When you create your resources from an Azure Resource Manager template (ARM template), you need to also include `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` in the template.

On your first deployment using an ARM template, don't include `WEBSITE_CONTENTSHARE`, which is generated for you.

You can use the following ARM template examples to help correctly configure these settings:

- [Consumption plan ↗](#)
- [Dedicated plan ↗](#)

- Premium plan with VNET integration ↗
- Consumption plan with a deployment slot ↗

Storage account configuration

When creating a function app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. Functions relies on Azure Storage for operations such as managing triggers and logging function executions. The storage account connection string for your function app is found in the `AzureWebJobsStorage` and `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` application settings.

Keep in mind the following considerations when creating this storage account:

- To reduce latency, create the storage account in the same region as the function app.
- To improve performance in production, use a separate storage account for each function app. This is especially true with Durable Functions and Event Hub triggered functions.
- For Event Hub triggered functions, don't use an account with [Data Lake Storage enabled](#) ↗.

Handling large data sets

When running on Linux, you can add extra storage by mounting a file share. Mounting a share is a convenient way for a function to process a large existing data set. To learn more, see [Mount file shares](#).

Organize your functions

As part of your solution, you likely develop and publish multiple functions. These functions are often combined into a single function app, but they can also run in separate function apps. In Premium and Dedicated (App Service) hosting plans, multiple function apps can also share the same resources by running in the same plan. How you group your functions and function apps can impact the performance, scaling, configuration, deployment, and security of your overall solution.

For Consumption and Premium plan, all functions in a function app are dynamically scaled together.

For more information on how to organize your functions, see [Function organization best practices](#).

Optimize deployments

When deploying a function app, it's important to keep in mind that the unit of deployment for functions in Azure is the function app. All functions in a function app are deployed at the same time, usually from the same deployment package.

Consider these options for a successful deployment:

- Have your functions run from the deployment package. This [run from package approach](#) provides the following benefits:
 - Reduces the risk of file copy locking issues.
 - Can be deployed directly to a production app, which does trigger a restart.
 - Know that all files in the package are available to your app.
 - Improves the performance of ARM template deployments.
 - May reduce cold-start times, particularly for JavaScript functions with large npm package trees.
- Consider using [continuous deployment](#) to connect deployments to your source control solution. Continuous deployments also let you run from the deployment package.
- For [Premium plan hosting](#), consider adding a warmup trigger to reduce latency when new instances are added. To learn more, see [Azure Functions warm-up trigger](#).
- To minimize deployment downtime and to be able to roll back deployments, consider using deployment slots. To learn more, see [Azure Functions deployment slots](#).

Write robust functions

There are several design principles you can follow when writing your function code that help with general performance and availability of your functions. These principles include:

- [Avoid long running functions](#).
- [Plan cross-function communication](#).
- [Write functions to be stateless](#).
- [Write defensive functions](#).

Because transient failures are common in cloud computing, you should use a [retry pattern](#) when accessing cloud-based resources. Many triggers and bindings already implement retry.

Design for security

Security is best considered during the planning phase and not after your functions are ready to go. To Learn how to securely develop and deploy functions, see [Securing Azure Functions](#).

Consider concurrency

As demand builds on your function app as a result of incoming events, function apps running in Consumption and Premium plans are scaled out. It's important to understand how your function app responds to load and how the triggers can be configured to handle incoming events. For a general overview, see [Event-driven scaling in Azure Functions](#).

Dedicated (App Service) plans require you to provide for scaling out your function apps.

Worker process count

In some cases, it's more efficient to handle the load by creating multiple processes, called language worker processes, in the instance before scale-out. The maximum number of language worker processes allowed is controlled by the `FUNCTIONS_WORKER_PROCESS_COUNT` setting. The default for this setting is `1`, which means that multiple processes aren't used. After the maximum number of processes are reached, the function app is scaled out to more instances to handle the load. This setting doesn't apply for [C# class library functions](#), which run in the host process.

When using `FUNCTIONS_WORKER_PROCESS_COUNT` on a Premium plan or Dedicated (App Service) plan, keep in mind the number of cores provided by your plan. For example, the Premium plan `EP2` provides two cores, so you should start with a value of `2` and increase by two as needed, up to the maximum.

Trigger configuration

When planning for throughput and scaling, it's important to understand how the different types of triggers process events. Some triggers allow you to control the batching behaviors and manage concurrency. Often adjusting the values in these

options can help each instance scale appropriately for the demands of the invoked functions. These configuration options are applied to all triggers in a function app, and are maintained in the host.json file for the app. See the Configuration section of the specific trigger reference for settings details.

To learn more about how Functions processes message streams, see [Azure Functions reliable event processing](#).

Plan for connections

Function apps running in [Consumption plan](#) are subject to connection limits. These limits are enforced on a per-instance basis. Because of these limits and as a general best practice, you should optimize your outbound connections from your function code. To learn more, see [Manage connections in Azure Functions](#).

Language-specific considerations

For your language of choice, keep in mind the following considerations:

C#

- [Use async code but avoid blocking calls](#).
- [Use cancellation tokens](#) (in-process only).

Maximize availability

Cold start is a key consideration for serverless architectures. To learn more, see [Cold starts](#). If cold start is a concern for your scenario, you can find a deeper dive in the post [Understanding serverless cold start](#).

Premium plan is the recommended plan for reducing cold starts while maintaining dynamic scale. You can use the following guidance to reduce cold starts and improve availability in all three hosting plans.

Plan	Guidance
------	----------

Plan	Guidance
Premium plan	<ul style="list-style-type: none"> • Implement a Warmup trigger in your function app • Set the values for Always-Ready instances and Max Burst limit • Use virtual network trigger support when using non-HTTP triggers on a virtual network
Dedicated plans	<ul style="list-style-type: none"> • Run on at least two instances with Azure App Service Health Check enabled • Implement autoscaling
Consumption plan	<ul style="list-style-type: none"> • Review your use of Singleton patterns and the concurrency settings for bindings and triggers to avoid artificially placing limits on how your function app scales. • Review the <code>functionAppScaleLimit</code> setting, which can limit scale-out • Check for a Daily Usage Quota (GB-Sec) limit set during development and testing. Consider removing this limit in production environments.

Monitor effectively

Azure Functions offers built-in integration with Azure Application Insights to monitor your function execution and traces written from your code. To learn more, see [Monitor Azure Functions](#). Azure Monitor also provides facilities for monitoring the health of the function app itself. To learn more, see [Monitoring with Azure Monitor](#).

You should be aware of the following considerations when using Application Insights integration to monitor your functions:

- Make sure that the [AzureWebJobsDashboard](#) application setting is removed. This setting was supported in older version of Functions. If it exists, removing `AzureWebJobsDashboard` improves performance of your functions.
- Review the [Application Insights logs](#). If data you expect to find is missing, consider adjusting the sampling settings to better capture your monitoring scenario. You can use the `excludedTypes` setting to exclude certain types from sampling, such as `Request` or `Exception`. To learn more, see [Configure sampling](#).

Azure Functions also allows you to [send system-generated and user-generated logs to Azure Monitor Logs](#). Integration with Azure Monitor Logs is currently in preview.

Build in redundancy

Your business needs might require that your functions always be available, even during a data center outage. To learn how to use a multi-regional approach to keep your critical functions always running, see [Azure Functions geo-disaster recovery and high-availability](#).

Next steps

[Manage your function app](#)

Improve the performance and reliability of Azure Functions

Article • 03/20/2023

This article provides guidance to improve the performance and reliability of your [serverless](#) function apps. For a more general set of Azure Functions best practices, see [Azure Functions best practices](#).

The following are best practices in how you build and architect your serverless solutions using Azure Functions.

Avoid long running functions

Large, long-running functions can cause unexpected timeout issues. To learn more about the timeouts for a given hosting plan, see [function app timeout duration](#).

A function can become large because of many Node.js dependencies. Importing dependencies can also cause increased load times that result in unexpected timeouts. Dependencies are loaded both explicitly and implicitly. A single module loaded by your code may load its own additional modules.

Whenever possible, refactor large functions into smaller function sets that work together and return responses fast. For example, a webhook or HTTP trigger function might require an acknowledgment response within a certain time limit; it's common for webhooks to require an immediate response. You can pass the HTTP trigger payload into a queue to be processed by a queue trigger function. This approach lets you defer the actual work and return an immediate response.

Make sure background tasks complete

When your function starts any tasks, callbacks, threads, processes, they must complete before your function code returns. Because Functions doesn't track these background threads, site shutdown can occur regardless of background thread status, which can cause unintended behavior in your functions.

For example, if a function starts a background task and returns a successful response before the task completes, the Functions runtime considers the execution as having completed successfully, regardless of the result of the background task. If this background task is performing essential work, it may be preempted by site shutdown, leaving that work in an unknown state.

Cross function communication

Durable Functions and Azure Logic Apps are built to manage state transitions and communication between multiple functions.

If not using Durable Functions or Logic Apps to integrate with multiple functions, it's best to use storage queues for cross-function communication. The main reason is that storage queues are cheaper and much easier to provision than other storage options.

Individual messages in a storage queue are limited in size to 64 KB. If you need to pass larger messages between functions, an Azure Service Bus queue could be used to support message sizes up to 256 KB in the Standard tier, and up to 100 MB in the Premium tier.

Service Bus topics are useful if you require message filtering before processing.

Event hubs are useful to support high volume communications.

Write functions to be stateless

Functions should be stateless and idempotent if possible. Associate any required state information with your data. For example, an order being processed would likely have an associated `state` member. A function could process an order based on that state while the function itself remains stateless.

Idempotent functions are especially recommended with timer triggers. For example, if you have something that absolutely must run once a day, write it so it can run anytime during the day with the same results. The function can exit when there's no work for a particular day. Also if a previous run failed to complete, the next run should pick up where it left off. This is particularly important for message-based bindings that retry on failure. For more information, see [Designing Azure Functions for identical input](#).

Write defensive functions

Assume your function could encounter an exception at any time. Design your functions with the ability to continue from a previous fail point during the next execution.

Consider a scenario that requires the following actions:

1. Query for 10,000 rows in a database.
2. Create a queue message for each of those rows to process further down the line.

Depending on how complex your system is, you may have: involved downstream services behaving badly, networking outages, or quota limits reached, etc. All of these can affect your function at any time. You need to design your functions to be prepared for it.

How does your code react if a failure occurs after inserting 5,000 of those items into a queue for processing? Track items in a set that you've completed. Otherwise, you might insert them again next time. This double-insertion can have a serious impact on your work flow, so [make your functions idempotent](#).

If a queue item was already processed, allow your function to be a no-op.

Take advantage of defensive measures already provided for components you use in the Azure Functions platform. For example, see [Handling poison queue messages](#) in the documentation for [Azure Storage Queue triggers and bindings](#).

For HTTP based functions consider [API versioning strategies](#) with Azure API Management. For example, if you have to update your HTTP based function app, deploy the new update to a separate function app and use API Management revisions or versions to direct clients to the new version or revision. Once all clients are using the version or revision and no more executions are left on the previous function app, you can deprovision the previous function app.

Function organization best practices

As part of your solution, you may develop and publish multiple functions. These functions are often combined into a single function app, but they can also run in separate function apps. In Premium and dedicated (App Service) hosting plans, multiple function apps can also share the same resources by running in the same plan. How you group your functions and function apps can impact the performance, scaling, configuration, deployment, and security of your overall solution. There aren't rules that apply to every scenario, so consider the information in this section when planning and developing your functions.

Organize functions for performance and scaling

Each function that you create has a memory footprint. While this footprint is usually small, having too many functions within a function app can lead to slower startup of your app on new instances. It also means that the overall memory usage of your function app might be higher. It's hard to say how many functions should be in a single app, which depends on your particular workload. However, if your function stores a lot of data in memory, consider having fewer functions in a single app.

If you run multiple function apps in a single Premium plan or dedicated (App Service) plan, these apps are all sharing the same resources allocated to the plan. If you have one function app that has a much higher memory requirement than the others, it uses a disproportionate amount of memory resources on each instance to which the app is deployed. Because this could leave less memory available for the other apps on each instance, you might want to run a high-memory-using function app like this in its own separate hosting plan.

ⓘ Note

When using the **Consumption plan**, we recommend you always put each app in its own plan, since apps are scaled independently anyway. For more information, see [Multiple apps in the same plan](#).

Consider whether you want to group functions with different load profiles. For example, if you have a function that processes many thousands of queue messages, and another that is only called occasionally but has high memory requirements, you might want to deploy them in separate function apps so they get their own sets of resources and they scale independently of each other.

Organize functions for configuration and deployment

Function apps have a `host.json` file, which is used to configure advanced behavior of function triggers and the Azure Functions runtime. Changes to the `host.json` file apply to all functions within the app. If you have some functions that need custom configurations, consider moving them into their own function app.

All functions in your local project are deployed together as a set of files to your function app in Azure. You might need to deploy individual functions separately or use features like [deployment slots](#) for some functions and not others. In such cases, you should deploy these functions (in separate code projects) to different function apps.

Organize functions by privilege

Connection strings and other credentials stored in application settings gives all of the functions in the function app the same set of permissions in the associated resource. Consider minimizing the number of functions with access to specific credentials by moving functions that don't use those credentials to a separate function app. You can always use techniques such as [function chaining](#) to pass data between functions in different function apps.

Scalability best practices

There are a number of factors that impact how instances of your function app scale. The details are provided in the documentation for [function scaling](#). The following are some best practices to ensure optimal scalability of a function app.

Share and manage connections

Reuse connections to external resources whenever possible. See [how to manage connections in Azure Functions](#).

Avoid sharing storage accounts

When you create a function app, you must associate it with a storage account. The storage account connection is maintained in the [AzureWebJobsStorage application setting](#).

To maximize performance, use a separate storage account for each function app. This is particularly important when you have Durable Functions or Event Hub triggered functions, which both generate a high volume of storage transactions. When your application logic interacts with Azure Storage, either directly (using the Storage SDK) or through one of the storage bindings, you should use a dedicated storage account. For example, if you have an Event Hub-triggered function writing some data to blob storage, use two storage accounts—one for the function app and another for the blobs being stored by the function.

Don't mix test and production code in the same function app

Functions within a function app share resources. For example, memory is shared. If you're using a function app in production, don't add test-related functions and resources to it. It can cause unexpected overhead during production code execution.

Be careful what you load in your production function apps. Memory is averaged across each function in the app.

If you have a shared assembly referenced in multiple .NET functions, put it in a common shared folder. Otherwise, you could accidentally deploy multiple versions of the same binary that behave differently between functions.

Don't use verbose logging in production code, which has a negative performance impact.

Use async code but avoid blocking calls

Asynchronous programming is a recommended best practice, especially when blocking I/O operations are involved.

In C#, always avoid referencing the `Result` property or calling `Wait` method on a `Task` instance. This approach can lead to thread exhaustion.

Tip

If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`. For more information, see [How to manage connections in Azure Functions](#).

Use multiple worker processes

By default, any host instance for Functions uses a single worker process. To improve performance, especially with single-threaded runtimes like Python, use the `FUNCTIONS_WORKER_PROCESS_COUNT` to increase the number of worker processes per host (up to 10). Azure Functions then tries to evenly distribute simultaneous function invocations across these workers.

The `FUNCTIONS_WORKER_PROCESS_COUNT` applies to each host that Functions creates when scaling out your application to meet demand.

Receive messages in batch whenever possible

Some triggers like Event Hub enable receiving a batch of messages on a single invocation. Batching messages has much better performance. You can configure the max batch size in the `host.json` file as detailed in the [host.json reference documentation](#)

For C# functions, you can change the type to a strongly-typed array. For example, instead of `EventData sensorEvent` the method signature could be `EventData[] sensorEvent`. For other languages, you'll need to explicitly set the cardinality property in your `function.json` to `many` in order to enable batching [as shown here ↗](#).

Configure host behaviors to better handle concurrency

The `host.json` file in the function app allows for configuration of host runtime and trigger behaviors. In addition to batching behaviors, you can manage concurrency for a number of triggers. Often adjusting the values in these options can help each instance scale appropriately for the demands of the invoked functions.

Settings in the `host.json` file apply across all functions within the app, within a *single instance* of the function. For example, if you had a function app with two HTTP functions and `maxConcurrentRequests` requests set to 25, a request to either HTTP trigger would count towards the shared 25 concurrent requests. When that function app is scaled to 10 instances, the ten functions effectively allow 250 concurrent requests (10 instances * 25 concurrent requests per instance).

Other host configuration options are found in the [host.json configuration article](#).

Next steps

For more information, see the following resources:

- [How to manage connections in Azure Functions](#)
- [Azure App Service best practices](#)

Manage connections in Azure Functions

Article • 11/18/2021

Functions in a function app share resources. Among those shared resources are connections: HTTP connections, database connections, and connections to services such as Azure Storage. When many functions are running concurrently in a Consumption plan, it's possible to run out of available connections. This article explains how to code your functions to avoid using more connections than they need.

ⓘ Note

Connection limits described in this article apply only when running in a **Consumption plan**. However, the techniques described here may be beneficial when running on any plan.

Connection limit

The number of available connections in a Consumption plan is limited partly because a function app in this plan runs in a [sandbox environment](#). One of the restrictions that the sandbox imposes on your code is a limit on the number of outbound connections, which is currently 600 active (1,200 total) connections per instance. When you reach this limit, the functions runtime writes the following message to the logs: `Host thresholds exceeded: Connections`. For more information, see the [Functions service limits](#).

This limit is per instance. When the [scale controller adds function app instances](#) to handle more requests, each instance has an independent connection limit. That means there's no global connection limit, and you can have much more than 600 active connections across all active instances.

When troubleshooting, make sure that you have enabled Application Insights for your function app. Application Insights lets you view metrics for your function apps like executions. For more information, see [View telemetry in Application Insights](#).

Static clients

To avoid holding more connections than necessary, reuse client instances rather than creating new ones with each function invocation. We recommend reusing client connections for any language that you might write your function in. For example, .NET

clients like the [HttpClient](#), [DocumentClient](#), and Azure Storage clients can manage connections if you use a single, static client.

Here are some guidelines to follow when you're using a service-specific client in an Azure Functions application:

- *Do not* create a new client with every function invocation.
- *Do* create a single, static client that every function invocation can use.
- *Consider* creating a single, static client in a shared helper class if different functions use the same service.

Client code examples

This section demonstrates best practices for creating and using clients from your function code.

HTTP requests

C#

Here's an example of C# function code that creates a static [HttpClient](#) instance:

```
C#  
  
// Create a single, static HttpClient  
private static HttpClient httpClient = new HttpClient();  
  
public static async Task Run(string input)  
{  
    var response = await httpClient.GetAsync("https://example.com");  
    // Rest of function  
}
```

A common question about [HttpClient](#) in .NET is "Should I dispose of my client?" In general, you dispose of objects that implement [IDisposable](#) when you're done using them. But you don't dispose of a static client because you aren't done using it when the function ends. You want the static client to live for the duration of your application.

Azure Cosmos DB clients

C#

[CosmosClient](#) connects to an Azure Cosmos DB instance. The Azure Cosmos DB documentation recommends that you [use a singleton Azure Cosmos DB client for the lifetime of your application](#). The following example shows one pattern for doing that in a function:

C#

```
#r "Microsoft.Azure.Cosmos"
using Microsoft.Azure.Cosmos;

private static Lazy<CosmosClient> lazyClient = new Lazy<CosmosClient>
(InitializeCosmosClient);
private static CosmosClient cosmosClient => lazyClient.Value;

private static CosmosClient InitializeCosmosClient()
{
    // Perform any initialization here
    var uri = "https://youraccount.documents.azure.com:443";
    var authKey = "authKey";

    return new CosmosClient(uri, authKey);
}

public static async Task Run(string input)
{
    Container container = cosmosClient.GetContainer("database",
"collection");
    MyItem item = new MyItem{ id = "myId", partitionKey =
"myPartitionKey", data = "example" };
    await container.UpsertItemAsync(item);

    // Rest of function
}
```

Also, create a file named "function.proj" for your trigger and add the below content :

C#

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>netcoreapp3.1</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="Microsoft.Azure.Cosmos"
Version="3.23.0" />
    </ItemGroup>
</Project>
```

SqlClient connections

Your function code can use the .NET Framework Data Provider for SQL Server ([SqlClient](#)) to make connections to a SQL relational database. This is also the underlying provider for data frameworks that rely on ADO.NET, such as [Entity Framework](#). Unlike [HttpClient](#) and [DocumentClient](#) connections, ADO.NET implements connection pooling by default. But because you can still run out of connections, you should optimize connections to the database. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

Tip

Some data frameworks, such as Entity Framework, typically get connection strings from the **ConnectionStrings** section of a configuration file. In this case, you must explicitly add SQL database connection strings to the **Connection strings** collection of your function app settings and in the **local.settings.json** file in your local project. If you're creating an instance of **SqlConnection** in your function code, you should store the connection string value in **Application settings** with your other connections.

Next steps

For more information about why we recommend static clients, see [Improper instantiation antipattern](#).

For more Azure Functions performance tips, see [Optimize the performance and reliability of Azure Functions](#).

Storage considerations for Azure Functions

Article • 07/30/2024

Azure Functions requires an Azure Storage account when you create a function app instance. The following storage services could be used by your function app:

[+] Expand table

Storage service	Functions usage
Azure Blob storage	Maintain bindings state and function keys ¹ . Deployment source for apps that run in a Flex Consumption plan . Used by default for task hubs in Durable Functions . Can be used to store function app code for Linux Consumption remote build or as part of external package URL deployments .
Azure Files ²	File share used to store and run your function app code in a Consumption Plan and Premium Plan .
Azure Queue storage	Used by default for task hubs in Durable Functions . Used for failure and retry handling in specific Azure Functions triggers . Used for object tracking by the Blob storage trigger .
Azure Table storage	Used by default for task hubs in Durable Functions .

1. Blob storage is the default store for function keys, but you can [configure an alternate store](#).
2. Azure Files is set up by default, but you can [create an app without Azure Files](#) under certain conditions.

Important considerations

You must strongly consider the following facts regarding the storage accounts used by your function apps:

- When your function app is hosted on the Consumption plan or Premium plan, your function code and configuration files are stored in Azure Files in the linked storage account. When you delete this storage account, the content is deleted and can't be recovered. For more information, see [Storage account was deleted](#)

- Important data, such as function code, [access keys](#), and other important service-related data, can be persisted in the storage account. You must carefully manage access to the storage accounts used by function apps in the following ways:
 - Audit and limit the access of apps and users to the storage account based on a least-privilege model. Permissions to the storage account can come from [data actions in the assigned role](#) or through permission to perform the [listKeys operation](#).
 - Monitor both control plane activity (such as retrieving keys) and data plane operations (such as writing to a blob) in your storage account. Consider maintaining storage logs in a location other than Azure Storage. For more information, see [Storage logs](#).

Storage account requirements

Storage accounts created as part of the function app create flow in the Azure portal are guaranteed to work with the new function app. When you choose to use an existing storage account, the list provided doesn't include certain unsupported storage accounts. The following restrictions apply to storage accounts used by your function app, so you must make sure an existing storage account meets these requirements:

- The account type must support Blob, Queue, and Table storage. Some storage accounts don't support queues and tables. These accounts include blob-only storage accounts and Azure Premium Storage. To learn more about storage account types, see [Storage account overview](#).
- You can't use a network-secured storage account when your function app is hosted in the [Consumption plan](#).
- When creating your function app in the portal, you're only allowed to choose an existing storage account in the same region as the function app you're creating. This is a performance optimization and not a strict limitation. To learn more, see [Storage account location](#).
- When creating your function app on a plan with [availability zone support](#) enabled, only [zone-redundant storage accounts](#) are supported.

When using deployment automation to create your function app with a network-secured storage account, you must include specific networking configurations in your ARM template or Bicep file. When you don't include these settings and resources, your automated deployment might fail in validation. For more specific ARM and Bicep

guidance, see [Secured deployments](#). For an overview on configuring storage accounts with networking, see [How to use a secured storage account with Azure Functions](#).

Storage account guidance

Every function app requires a storage account to operate. When that account is deleted, your function app won't run. To troubleshoot storage-related issues, see [How to troubleshoot storage-related issues](#). The following other considerations apply to the Storage account used by function apps.

Storage account location

For best performance, your function app should use a storage account in the same region, which reduces latency. The Azure portal enforces this best practice. If for some reason you need to use a storage account in a region different than your function app, you must create your function app outside of the portal.

The storage account must be accessible to the function app. If you need to use a secured storage account, consider [restricting your storage account to a virtual network](#).

Storage account connection setting

By default, function apps configure the `AzureWebJobsStorage` connection as a connection string stored in the [AzureWebJobsStorage application setting](#), but you can also [configure AzureWebJobsStorage to use an identity-based connection](#) without a secret.

Function apps running in a Consumption plan (Windows only) or an Elastic Premium plan (Windows or Linux) can use Azure Files to store the images required to enable dynamic scaling. For these plans, set the connection string for the storage account in the `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` setting and the name of the file share in the `WEBSITE_CONTENTSHARE` setting. This is usually the same account used for `AzureWebJobsStorage`. You can also [create a function app that doesn't use Azure Files](#), but scaling might be limited.

Note

A storage account connection string must be updated when you regenerate storage keys. [Read more about storage key management here](#).

Shared storage accounts

It's possible for multiple function apps to share the same storage account without any issues. For example, in Visual Studio you can develop multiple apps using the [Azurite storage emulator](#). In this case, the emulator acts like a single storage account. The same storage account used by your function app can also be used to store your application data. However, this approach isn't always a good idea in a production environment.

You might need to use separate storage accounts to [avoid host ID collisions](#).

Lifecycle management policy considerations

You shouldn't apply [lifecycle management policies](#) to your Blob Storage account used by your function app. Functions uses Blob storage to persist important information, such as [function access keys](#), and policies could remove blobs (such as keys) needed by the Functions host. If you must use policies, exclude containers used by Functions, which are prefixed with `azure-webjobs` or `scm`.

Storage logs

Because function code and keys might be persisted in the storage account, logging of activity against the storage account is a good way to monitor for unauthorized access. Azure Monitor resource logs can be used to track events against the storage data plane. See [Monitoring Azure Storage](#) for details on how to configure and examine these logs.

The [Azure Monitor activity log](#) shows control plane events, including the [listKeys operation](#). However, you should also configure resource logs for the storage account to track subsequent use of keys or other identity-based data plane operations. You should have at least the [StorageWrite log category](#) enabled to be able to identify modifications to the data outside of normal Functions operations.

To limit the potential impact of any broadly scoped storage permissions, consider using a nonstorage destination for these logs, such as Log Analytics. For more information, see [Monitoring Azure Blob Storage](#).

Optimize storage performance

To maximize performance, use a separate storage account for each function app. This is particularly important when you have Durable Functions or Event Hub triggered functions, which both generate a high volume of storage transactions. When your application logic interacts with Azure Storage, either directly (using the Storage SDK) or through one of the storage bindings, you should use a dedicated storage account. For example, if you have an Event Hub-triggered function writing some data to blob

storage, use two storage accounts—one for the function app and another for the blobs being stored by the function.

Consistent routing through virtual networks

Multiple function apps hosted in the same plan can also use the same storage account for the Azure Files content share (defined by

`WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`). When this storage account is also secured by a virtual network, all of these apps should also use the same value for `vnetContentShareEnabled` (formerly `WEBSITE_CONTENTOVERVNET`) to guarantee that traffic is routed consistently through the intended virtual network. A mismatch in this setting between apps using the same Azure Files storage account might result in traffic being routed through public networks, which causes access to be blocked by storage account network rules.

Working with blobs

A key scenario for Functions is file processing of files in a blob container, such as for image processing or sentiment analysis. To learn more, see [Process file uploads](#).

Trigger on a blob container

There are several ways to execute your function code based on changes to blobs in a storage container. Use the following table to determine which function trigger best fits your needs:

[+] [Expand table](#)

Strategy	Container (polling)	Container (events)	Queue trigger	Event Grid
Latency	High (up to 10 min)	Low	Medium	Low
Storage account limitations	Blob-only accounts not supported ¹	general purpose v1 not supported	none	general purpose v1 not supported
Trigger type	Blob storage	Blob storage	Queue storage	Event Grid
Extension version	Any	Storage v5.x+	Any	Any

Strategy	Container (polling)	Container (events)	Queue trigger	Event Grid
Processes existing blobs	Yes	No	No	No
Filters	Blob name pattern	Event filters	n/a	Event filters
Requires event subscription	No	Yes	No	Yes
Supports Flex Consumption plan	No	Yes	Yes	Yes
Supports high-scale ²	No	Yes	Yes	Yes
Description	Default trigger behavior, which relies on polling the container for updates. For more information, see the examples in the Blob storage trigger reference .	Consumes blob storage events from an event subscription. Requires a Source parameter value of EventGrid . For more information, see Tutorial: Trigger Azure Functions on blob containers using an event subscription .	Blob name string is manually added to a storage queue when a blob is added to the container. This value is passed directly by a Queue storage trigger to a Blob storage input binding on the same function.	Provides the flexibility of triggering on events besides those coming from a storage container. Use when need to also have nonstorage events trigger your function. For more information, see How to work with Event Grid triggers and bindings in Azure Functions .

1. Blob storage input and output bindings support blob-only accounts.
2. High scale can be loosely defined as containers that have more than 100,000 blobs in them or storage accounts that have more than 100 blob updates per second.

Storage data encryption

Azure Storage encrypts all data in a storage account at rest. For more information, see [Azure Storage encryption for data at rest](#).

By default, data is encrypted with Microsoft-managed keys. For additional control over encryption keys, you can supply customer-managed keys to use for encryption of blob

and file data. These keys must be present in Azure Key Vault for Functions to be able to access the storage account. To learn more, see [Encryption at rest using customer-managed keys](#).

In-region data residency

When all customer data must remain within a single region, the storage account associated with the function app must be one with [in-region redundancy](#). An in-region redundant storage account also must be used with [Azure Durable Functions](#).

Other platform-managed customer data is only stored within the region when hosting in an internally load-balanced App Service Environment (ASE). To learn more, see [ASE zone redundancy](#).

Host ID considerations

Functions uses a host ID value as a way to uniquely identify a particular function app in stored artifacts. By default, this ID is autogenerated from the name of the function app, truncated to the first 32 characters. This ID is then used when storing per-app correlation and tracking information in the linked storage account. When you have function apps with names longer than 32 characters and when the first 32 characters are identical, this truncation can result in duplicate host ID values. When two function apps with identical host IDs use the same storage account, you get a host ID collision because stored data can't be uniquely linked to the correct function app.

Note

This same kind of host ID collision can occur between a function app in a production slot and the same function app in a staging slot, when both slots use the same storage account.

Starting with version 3.x of the Functions runtime, host ID collision is detected and a warning is logged. In version 4.x, an error is logged and the host is stopped, resulting in a hard failure. More details about host ID collision can be found in [this issue](#).

Avoiding host ID collisions

You can use the following strategies to avoid host ID collisions:

- Use a separated storage account for each function app or slot involved in the collision.

- Rename one of your function apps to a value fewer than 32 characters in length, which changes the computed host ID for the app and removes the collision.
- Set an explicit host ID for one or more of the colliding apps. To learn more, see [Host ID override](#).

ⓘ Important

Changing the storage account associated with an existing function app or changing the app's host ID can impact the behavior of existing functions. For example, a Blob storage trigger tracks whether it's processed individual blobs by writing receipts under a specific host ID path in storage. When the host ID changes or you point to a new storage account, previously processed blobs could be reprocessed.

Override the host ID

You can explicitly set a specific host ID for your function app in the application settings by using the `AzureFunctionsWebHost__hostid` setting. For more information, see [AzureFunctionsWebHost__hostid](#).

When the collision occurs between slots, you must set a specific host ID for each slot, including the production slot. You must also mark these settings as [deployment settings](#) so they don't get swapped. To learn how to create app settings, see [Work with application settings](#).

Azure Arc-enabled clusters

When your function app is deployed to an Azure Arc-enabled Kubernetes cluster, a storage account might not be required by your function app. In this case, a storage account is only required by Functions when your function app uses a trigger that requires storage. The following table indicates which triggers might require a storage account and which don't.

[] [Expand table](#)

Not required	might require storage
<ul style="list-style-type: none"> • Azure Cosmos DB • HTTP • Kafka • RabbitMQ • Service Bus 	<ul style="list-style-type: none"> • Azure SQL • Blob storage • Event Grid • Event Hubs • IoT Hub • Queue storage

Not required	might require storage
	<ul style="list-style-type: none">• SendGrid• SignalR• Table storage• Timer• Twilio

To create a function app on an Azure Arc-enabled Kubernetes cluster without storage, you must use the Azure CLI command [az functionapp create](#). The version of the Azure CLI must include version 0.1.7 or a later version of the [appservice-kube extension](#). Use the `az --version` command to verify that the extension is installed and is the correct version.

Creating your function app resources using methods other than the Azure CLI requires an existing storage account. If you plan to use any triggers that require a storage account, you should create the account before you create the function app.

Create an app without Azure Files

The Azure Files service provides a shared file system that supports high-scale scenarios. When your function app runs on Windows in an Elastic Premium or Consumption plan, an Azure Files share is created by default in your storage account. That share is used by Functions to enable certain features, like log streaming. It is also used as a shared package deployment location, which guarantees the consistency of your deployed function code across all instances.

By default, function apps hosted in Premium and Consumption plans use [zip deployment](#), with deployment packages stored in this Azure file share. This section is only relevant to these hosting plans.

Using Azure Files requires the use of a connection string, which is stored in your app settings as [WEBSITE_CONTENTAZUREFILECONNECTIONSTRING](#). Azure Files doesn't currently support identity-based connections. If your scenario requires you to not store any secrets in app settings, you must remove your app's dependency on Azure Files. You can do this by creating your app without the default Azure Files dependency.

Note

You should also consider running in your function app in the Flex Consumption plan, which is currently in preview. The Flex Consumption plan provides greater control over the deployment package, including the ability use managed identity

connections. For more information, see [Configure deployment settings](#) in the Flex Consumption article.

To run your app without the Azure file share, you must meet the following requirements:

- You must [deploy your package to a remote Azure Blob storage container](#) and then set the URL that provides access to that package as the `WEBSITE_RUN_FROM_PACKAGE` app setting. This option lets you store your app content in Blob storage instead of Azure Files, which does support [managed identities](#).

You are responsible for manually updating the deployment package and maintaining the deployment package URL, which likely contains a shared access signature (SAS).

- Your app can't rely on a shared writeable file system.
- The app can't use version 1.x of the Functions runtime.
- Log streaming experiences in clients such as the Azure portal default to file system logs. You should instead rely on Application Insights logs.

If the above requirements suit your scenario, you can proceed to create a function app without Azure Files. You can do this by creating an app without the `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` and `WEBSITE_CONTENTSHARE` app settings. To get started, generate an ARM template for a standard deployment, remove the two settings, and then deploy the modified template.

Since Azure Files is used to enable dynamic scale-out for Functions, scaling could be limited when running your app without Azure Files in the Elastic Premium plan and Consumption plans running on Windows.

Mount file shares

This functionality is current only available when running on Linux.

You can mount existing Azure Files shares to your Linux function apps. By mounting a share to your Linux function app, you can use existing machine learning models or other data in your functions. You can use the following command to mount an existing share to your Linux function app.

Azure CLI

```
az webapp config storage-account add
```

In this command, `share-name` is the name of the existing Azure Files share, and `custom-id` can be any string that uniquely defines the share when mounted to the function app. Also, `mount-path` is the path from which the share is accessed in your function app. `mount-path` must be in the format `/dir-name`, and it can't start with `/home`.

For a complete example, see the scripts in [Create a Python function app and mount an Azure Files share](#).

Currently, only a `storage-type` of `AzureFiles` is supported. You can only mount five shares to a given function app. Mounting a file share can increase the cold start time by at least 200-300 ms, or even more when the storage account is in a different region.

The mounted share is available to your function code at the `mount-path` specified. For example, when `mount-path` is `/path/to/mount`, you can access the target directory by file system APIs, as in the following Python example:

Python

```
import os
...
files_in_share = os.listdir("/path/to/mount")
```

Next steps

Learn more about Azure Functions hosting options.

[Azure Functions scale and hosting](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | Get help at Microsoft Q&A

Azure Functions error handling and retries

Article • 04/26/2024

Handling errors in Azure Functions is important to help you avoid lost data, avoid missed events, and monitor the health of your application. It's also an important way to help you understand the retry behaviors of event-based triggers.

This article describes general strategies for error handling and the available retry strategies.

ⓘ Important

Preview retry policy support for certain triggers was removed in December 2022.

Retry policies for supported triggers are now generally available (GA). For a list of extensions that currently support retry policies, see the [Retries](#) section.

Handling errors

Errors that occur in an Azure function can come from:

- Use of built-in Functions [triggers and bindings](#).
- Calls to APIs of underlying Azure services.
- Calls to REST endpoints.
- Calls to client libraries, packages, or third-party APIs.

To avoid loss of data or missed messages, it's important to practice good error handling. This table describes some recommended error-handling practices and provides links to more information.

[] [Expand table](#)

Recommendation	Details
Enable Application Insights	Azure Functions integrates with Application Insights to collect error data, performance data, and runtime logs. You should use Application Insights to discover and better understand errors that occur in your function executions. To learn more, see Monitor Azure Functions .
Use structured error handling	Capturing and logging errors is critical to monitoring the health of your application. The top-most level of any function code should include a

Recommendation	Details
	try/catch block. In the catch block, you can capture and log errors. For information about what errors might be raised by bindings, see Binding error codes . Depending on your specific retry strategy, you might also raise a new exception to run the function again.
Plan your retry strategy	Several Functions bindings extensions provide built-in support for retries and others let you define retry policies, which are implemented by the Functions runtime. For triggers that don't provide retry behaviors, you should consider implementing your own retry scheme. For more information, see Retries .
Design for idempotency	The occurrence of errors when you're processing data can be a problem for your functions, especially when you're processing messages. It's important to consider what happens when the error occurs and how to avoid duplicate processing. To learn more, see Designing Azure Functions for identical input .

Retries

There are two kinds of retries available for your functions:

- Built-in retry behaviors of individual trigger extensions
- Retry policies provided by the Functions runtime

The following table indicates which triggers support retries and where the retry behavior is configured. It also links to more information about errors that come from the underlying services.

[] [Expand table](#)

Trigger/binding	Retry source	Configuration
Azure Cosmos DB	Retry policies	Function-level
Blob Storage	Binding extension	host.json
Event Grid	Binding extension	Event subscription
Event Hubs	Retry policies	Function-level
Kafka	Retry policies	Function-level
Queue Storage	Binding extension	host.json
RabbitMQ	Binding extension	Dead letter queue ↗

Trigger/binding	Retry source	Configuration
Service Bus	Binding extension	host.json*
Timer	Retry policies	Function-level

*Requires version 5.x of the Azure Service Bus extension. In older extension versions, retry behaviors are implemented by the [Service Bus dead letter queue](#).

Retry policies

Azure Functions lets you define retry policies for specific trigger types, which are enforced by the runtime. These trigger types currently support retry policies:

- [Azure Cosmos DB](#)
- [Event Hubs](#)
- [Kafka](#)
- [Timer](#)

Retry policies aren't supported in version 1.x of the Functions runtime.

The retry policy tells the runtime to rerun a failed execution until either successful completion occurs or the maximum number of retries is reached.

A retry policy is evaluated when a function executed by a supported trigger type raises an uncaught exception. As a best practice, you should catch all exceptions in your code and raise new exceptions for any errors that you want to result in a retry.

Important

Event Hubs checkpoints aren't written until after the retry policy for the execution has completed. Because of this behavior, progress on the specific partition is paused until the current batch is done processing.

The version 5.x of the Event Hubs extension supports additional retry capabilities for interactions between the Functions host and the event hub. For more information, see `clientRetryOptions` in the [Event Hubs host.json reference](#).

Retry strategies

You can configure two retry strategies that are supported by policy:

Fixed delay

A specified amount of time is allowed to elapse between each retry.

When running in a Consumption plan, you are only billed for time your function code is executing. You aren't billed for the wait time between executions in either of these retry strategies.

Max retry counts

You can configure the maximum number of times that a function execution is retried before eventual failure. The current retry count is stored in memory of the instance.

It's possible for an instance to have a failure between retry attempts. When an instance fails during a retry policy, the retry count is lost. When there are instance failures, the Event Hubs trigger is able to resume processing and retry the batch on a new instance, with the retry count reset to zero. The timer trigger doesn't resume on a new instance.

This behavior means that the maximum retry count is a best effort. In some rare cases, an execution could be retried more than the requested maximum number of times. For Timer triggers, the retries can be less than the maximum number requested.

Retry examples

Examples are provided for both fixed delay and exponential backoff strategies. To see examples for a specific strategy, you must first select that strategy in the previous tab.

Isolated worker model

Function-level retries are supported with the following NuGet packages:

- [Microsoft.Azure.Functions.Worker.Sdk](#) >= 1.9.0
- [Microsoft.Azure.Functions.Worker.Extensions.EventHubs](#) >= 5.2.0
- [Microsoft.Azure.Functions.Worker.Extensions.Kafka](#) >= 3.8.0
- [Microsoft.Azure.Functions.Worker.Extensions.Timer](#) >= 4.2.0

C#

```
[Function(nameof(TimerFunction))]
[FixedDelayRetry(5, "00:00:10")]
public static void Run([TimerTrigger("0 */5 * * *")] TimerInfo
timerInfo,
    FunctionContext context)
```

```

{
    var logger = context.GetLogger(nameof(TimerFunction));
    logger.LogInformation($"Function Ran. Next timer schedule =
{timerInfo.ScheduleStatus.Next}");
}

```

[] [Expand table](#)

Property	Description
MaxRetryCount	Required. The maximum number of retries allowed per function execution. -1 means to retry indefinitely.
DelayInterval	The delay used between retries. Specify it as a string with the format <code>HH:mm:ss</code> .

Binding error codes

When you're integrating with Azure services, errors might originate from the APIs of the underlying services. Information that relates to binding-specific errors is available in the "Exceptions and return codes" sections of the following articles:

- [Azure Cosmos DB](#)
- [Blob Storage](#)
- [Event Grid](#)
- [Event Hubs](#)
- [IoT Hub](#)
- [Notification Hubs](#)
- [Queue Storage](#)
- [Service Bus](#)
- [Table Storage](#)

Next steps

- [Azure Functions triggers and bindings concepts](#)
- [Best practices for reliable Azure functions](#)

Securing Azure Functions

Article • 07/18/2024

In many ways, planning for secure development, deployment, and operation of serverless functions is much the same as for any web-based or cloud-hosted application. [Azure App Service](#) provides the hosting infrastructure for your function apps. This article provides security strategies for running your function code, and how App Service can help you secure your functions.

The platform components of App Service, including Azure VMs, storage, network connections, web frameworks, management and integration features, are actively secured and hardened. App Service goes through vigorous compliance checks on a continuous basis to make sure that:

- Your app resources are [secured ↗](#) from the other customers' Azure resources.
- [VM instances and runtime software are regularly updated](#) to address newly discovered vulnerabilities.
- Communication of secrets (such as connection strings) between your app and other Azure resources (such as [SQL Database ↗](#)) stays within Azure and doesn't cross any network boundaries. Secrets are always encrypted when stored.
- All communication over the App Service connectivity features, such as [hybrid connection](#), is encrypted.
- Connections with remote management tools like Azure PowerShell, Azure CLI, Azure SDKs, REST APIs, are all encrypted.
- 24-hour threat management protects the infrastructure and platform against malware, distributed denial-of-service (DDoS), man-in-the-middle (MITM), and other threats.

For more information on infrastructure and platform security in Azure, see [Azure Trust Center ↗](#).

For a set of security recommendations that follow the [Microsoft cloud security benchmark](#), see [Azure Security Baseline for Azure Functions](#).

Secure operation

This section guides you on configuring and running your function app as securely as possible.

Defender for Cloud

Defender for Cloud integrates with your function app in the portal. It provides, for free, a quick assessment of potential configuration-related security vulnerabilities. Function apps running in a dedicated plan can also use Defender for Cloud's enhanced security features for an extra cost. To learn more, see [Protect your Azure App Service web apps and APIs](#).

Log and monitor

One way to detect attacks is through activity monitoring and logging analytics. Functions integrates with Application Insights to collect log, performance, and error data for your function app. Application Insights automatically detects performance anomalies and includes powerful analytics tools to help you diagnose issues and to understand how your functions are used. To learn more, see [Monitor Azure Functions](#).

Functions also integrates with Azure Monitor Logs to enable you to consolidate function app logs with system events for easier analysis. You can use diagnostic settings to configure streaming export of platform logs and metrics for your functions to the destination of your choice, such as a Logs Analytics workspace. To learn more, see [Monitoring Azure Functions with Azure Monitor Logs](#).

For enterprise-level threat detection and response automation, stream your logs and events to a Logs Analytics workspace. You can then connect Microsoft Sentinel to this workspace. To learn more, see [What is Microsoft Sentinel](#).

For more security recommendations for observability, see the [Azure security baseline for Azure Functions](#).

Secure HTTP endpoints

HTTP endpoints that are exposed publicly provide a vector of attack for malicious actors. When securing your HTTP endpoints, you should use a layered security approach. These techniques can be used to reduce the vulnerability of publicly exposed HTTP endpoints, ordered from most basic to most secure and restrictive:

- [Require HTTPS](#)
- [Require access keys](#)
- [Enable App Service Authentication/Authorization](#)
- [Use Azure API Management \(APIM\) to authenticate requests](#)
- [Deploy your function app to a virtual network](#)
- [Deploy your function app in isolation](#)

Require HTTPS

By default, clients can connect to function endpoints by using both HTTP or HTTPS. You should redirect HTTP to HTTPS because HTTPS uses the SSL/TLS protocol to provide a secure connection, which is both encrypted and authenticated. To learn how, see [Enforce HTTPS](#).

When you require HTTPS, you should also require the latest TLS version. To learn how, see [Enforce TLS versions](#).

For more information, see [Secure connections \(TLS\)](#).

Function access keys

Functions lets you use keys to make it harder to access your function endpoints. Unless the HTTP access level on an HTTP triggered function is set to `anonymous`, requests must include an access key in the request. For more information, see [Work with access keys in Azure Functions](#).

While access keys can provide some mitigation for unwanted access, the only way to truly secure your function endpoints is by implementing positive authentication of clients accessing your functions. You can then make authorization decisions based on identity.

For the highest level of security, you can also secure the entire application architecture inside a virtual network [using private endpoints](#) or by [running in isolation](#).

Enable App Service Authentication/Authorization

The App Service platform lets you use Microsoft Entra ID and several third-party identity providers to authenticate clients. You can use this strategy to implement custom authorization rules for your functions, and you can work with user information from your function code. To learn more, see [Authentication and authorization in Azure App Service](#) and [Working with client identities](#).

Use Azure API Management (APIM) to authenticate requests

APIM provides various API security options for incoming requests. To learn more, see [API Management authentication policies](#). With APIM in place, you can configure your function app to accept requests only from the IP address of your APIM instance. To learn more, see [IP address restrictions](#).

Permissions

As with any application or service, the goal is run your function app with the lowest possible permissions.

User management permissions

Functions supports built-in Azure role-based access control ([Azure RBAC](#)). Azure roles supported by Functions are [Contributor](#), [Owner](#), and [Reader](#).

Permissions are effective at the function app level. The Contributor role is required to perform most function app-level tasks. You also need the Contributor role along with the [Monitoring Reader permission](#) to be able to view log data in Application Insights. Only the Owner role can delete a function app.

Organize functions by privilege

Connection strings and other credentials stored in application settings gives all of the functions in the function app the same set of permissions in the associated resource. Consider minimizing the number of functions with access to specific credentials by moving functions that don't use those credentials to a separate function app. You can always use techniques such as [function chaining](#) to pass data between functions in different function apps.

Managed identities

A managed identity from Microsoft Entra ID allows your app to easily access other Microsoft Entra protected resources such as Azure Key Vault. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. For more about managed identities in Microsoft Entra ID, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to the app and is deleted if the app is deleted. An app can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your app. An app can have multiple user-assigned identities, and one user-assigned identity can be assigned to multiple Azure resources, such as two App Service apps.

Managed identities can be used in place of secrets for connections from some triggers and bindings. See [Identity-based connections](#).

For more information, see [How to use managed identities for App Service and Azure Functions](#).

Restrict CORS access

[Cross-origin resource sharing \(CORS\)](#) is a way to allow web apps running in another domain to make requests to your HTTP trigger endpoints. App Service provides built-in support for handing the required CORS headers in HTTP requests. CORS rules are defined on a function app level.

While it's tempting to use a wildcard that allows all sites to access your endpoint, this defeats the purpose of CORS, which is to help prevent cross-site scripting attacks. Instead, add a separate CORS entry for the domain of each web app that must access your endpoint.

Managing secrets

To be able to connect to the various services and resources need to run your code, function apps need to be able to access secrets, such as connection strings and service keys. This section describes how to store secrets required by your functions.

Never store secrets in your function code.

Application settings

By default, you store connection strings and secrets used by your function app and bindings as application settings. This makes these credentials available to both your function code and the various bindings used by the function. The application setting (key) name is used to retrieve the actual value, which is the secret.

For example, every function app requires an associated storage account, which is used by the runtime. By default, the connection to this storage account is stored in an application setting named `AzureWebJobsStorage`.

App settings and connection strings are stored encrypted in Azure. They're decrypted only before being injected into your app's process memory when the app starts. The encryption keys are rotated regularly. If you prefer to instead manage the secure storage of your secrets, the app setting should instead be references to Azure Key Vault.

You can also encrypt settings by default in the local.settings.json file when developing functions on your local computer. For more information, see [Encrypt the local settings file](#).

Key Vault references

While application settings are sufficient for most many functions, you may want to share the same secrets across multiple services. In this case, redundant storage of secrets results in more potential vulnerabilities. A more secure approach is to a central secret storage service and use references to this service instead of the secrets themselves.

[Azure Key Vault](#) is a service that provides centralized secrets management, with full control over access policies and audit history. You can use a Key Vault reference in the place of a connection string or key in your application settings. To learn more, see [Use Key Vault references for App Service and Azure Functions](#).

Identity-based connections

Identities may be used in place of secrets for connecting to some resources. This has the advantage of not requiring the management of a secret, and it provides more fine-grained access control and auditing.

When you're writing code that creates the connection to [Azure services that support Microsoft Entra authentication](#), you can choose to use an identity instead of a secret or connection string. Details for both connection methods are covered in the documentation for each service.

Some Azure Functions binding extensions can be configured to access services using identity-based connections. For more information, see [Configure an identity-based connection](#).

Set usage quotas

Consider setting a usage quota on functions running in a Consumption plan. When you set a daily GB-sec limit on the sum total execution of functions in your function app, execution is stopped when the limit is reached. This could potentially help mitigate against malicious code executing your functions. To learn how to estimate consumption for your functions, see [Estimating Consumption plan costs](#).

Data validation

The triggers and bindings used by your functions don't provide any additional data validation. Your code must validate any data received from a trigger or input binding. If an upstream service is compromised, you don't want unvalidated inputs flowing through your functions. For example, if your function stores data from an Azure Storage queue in a relational database, you must validate the data and parameterize your commands to avoid SQL injection attacks.

Don't assume that the data coming into your function has already been validated or sanitized. It's also a good idea to verify that the data being written to output bindings is valid.

Handle errors

While it seems basic, it's important to write good error handling in your functions. Unhandled errors bubble-up to the host and are handled by the runtime. Different bindings handle processing of errors differently. To learn more, see [Azure Functions error handling](#).

Disable remote debugging

Make sure that remote debugging is disabled, except when you are actively debugging your functions. You can disable remote debugging in the **General Settings** tab of your function app **Configuration** in the portal.

Restrict CORS access

Azure Functions supports cross-origin resource sharing (CORS). CORS is configured [in the portal](#) and through the [Azure CLI](#). The CORS allowed origins list applies at the function app level. With CORS enabled, responses include the `Access-Control-Allow-Origin` header. For more information, see [Cross-origin resource sharing](#).

Don't use wildcards in your allowed origins list. Instead, list the specific domains from which you expect to get requests.

Store data encrypted

Azure Storage encrypts all data in a storage account at rest. For more information, see [Azure Storage encryption for data at rest](#).

By default, data is encrypted with Microsoft-managed keys. For additional control over encryption keys, you can supply customer-managed keys to use for encryption of blob

and file data. These keys must be present in Azure Key Vault for Functions to be able to access the storage account. To learn more, see [Encryption at rest using customer-managed keys](#).

Secure related resources

A function app frequently depends on additional resources, so part of securing the app is securing these external resources. At a minimum, most function apps include a dependency on Application Insights and Azure Storage. Consult the [Azure security baseline for Azure Monitor](#) and the [Azure security baseline for Storage](#) for guidance on securing these resources.

Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

You should also consult the guidance for any resource types your application logic depends on, both as triggers and bindings and from your function code.

Secure deployment

Azure Functions tooling and integration make it easy to publish local function project code to Azure. It's important to understand how deployment works when considering security for an Azure Functions topology.

Deployment credentials

App Service deployments require a set of deployment credentials. These deployment credentials are used to secure your function app deployments. Deployment credentials are managed by the App Service platform and are encrypted at rest.

There are two kinds of deployment credentials:

- **User-level credentials:** one set of credentials for the entire Azure account. It can be used to deploy to App Service for any app, in any subscription, that the Azure account has permission to access. It's the default set that's surfaced in the portal GUI (such as the [Overview](#) and [Properties](#) of the app's [resource page](#)). When a user is granted app access via Role-Based Access Control (RBAC) or coadmin

permissions, that user can use their own user-level credentials until the access is revoked. Do not share these credentials with other Azure users.

- **App-level credentials:** one set of credentials for each app. It can be used to deploy to that app only. The credentials for each app are generated automatically at app creation. They can't be configured manually, but can be reset anytime. For a user to be granted access to app-level credentials via (RBAC), that user must be contributor or higher on the app (including Website Contributor built-in role). Readers are not allowed to publish, and can't access those credentials.

At this time, Key Vault isn't supported for deployment credentials. To learn more about managing deployment credentials, see [Configure deployment credentials for Azure App Service](#).

Disable FTP

By default, each function app has an FTP endpoint enabled. The FTP endpoint is accessed using deployment credentials.

FTP isn't recommended for deploying your function code. FTP deployments are manual, and they require you to synchronize triggers. To learn more, see [FTP deployment](#).

When you're not planning on using FTP, you should disable it in the portal. If you do choose to use FTP, you should [enforce FTPS](#).

Secure the scm endpoint

Every function app has a corresponding `scm` service endpoint that is used by the Advanced Tools (Kudu) service for deployments and other App Service [site extensions](#). The scm endpoint for a function app is always a URL in the form `https://<FUNCTION_APP_NAME>.scm.azurewebsites.net`. When you use network isolation to secure your functions, you must also account for this endpoint.

By having a separate scm endpoint, you can control deployments and other advanced tools functionalities for function app that are isolated or running in a virtual network. The scm endpoint supports both basic authentication (using deployment credentials) and single sign-on with your Azure portal credentials. To learn more, see [Accessing the Kudu service](#).

Continuous security validation

Since security needs to be considered at every step in the development process, it makes sense to also implement security validations in a continuous deployment environment. This is sometimes called DevSecOps. Using Azure DevOps for your deployment pipeline lets you integrate validation into the deployment process. For more information, see [Learn how to add continuous security validation to your CI/CD pipeline](#).

Network security

Restricting network access to your function app lets you control who can access your functions endpoints. Functions leverages App Service infrastructure to enable your functions to access resources without using internet-routable addresses or to restrict internet access to a function endpoint. To learn more about these networking options, see [Azure Functions networking options](#).

Set access restrictions

Access restrictions allow you to define lists of allow/deny rules to control traffic to your app. Rules are evaluated in priority order. If there are no rules defined, then your app will accept traffic from any address. To learn more, see [Azure App Service Access Restrictions](#).

Secure the storage account

When you create a function app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. You can replace this storage account with one that is secured by a virtual network with access enabled by service endpoints or private endpoints. For more information, see [Restrict your storage account to a virtual network](#).

Deploy your function app to a virtual network

[Azure Private Endpoint](#) is a network interface that connects you privately and securely to a service powered by Azure Private Link. Private Endpoint uses a private IP address from your virtual network, effectively bringing the service into your virtual network.

You can use Private Endpoint for your functions hosted in the [Premium](#) and [App Service](#) plans.

If you want to make calls to Private Endpoints, then you must make sure that your DNS lookups resolve to the private endpoint. You can enforce this behavior in one of the

following ways:

- Integrate with Azure DNS private zones. When your virtual network doesn't have a custom DNS server, this is done automatically.
- Manage the private endpoint in the DNS server used by your app. To do this you must know the private endpoint address and then point the endpoint you are trying to reach to that address using an A record.
- Configure your own DNS server to forward to [Azure DNS private zones](#).

To learn more, see [using Private Endpoints for Web Apps](#).

Deploy your function app in isolation

Azure App Service Environment provides a dedicated hosting environment in which to run your functions. These environments let you configure a single front-end gateway that you can use to authenticate all incoming requests. For more information, see [Configuring a Web Application Firewall \(WAF\) for App Service Environment](#).

Use a gateway service

Gateway services, such as [Azure Application Gateway](#) and [Azure Front Door](#) let you set up a Web Application Firewall (WAF). WAF rules are used to monitor or block detected attacks, which provide an extra layer of protection for your functions. To set up a WAF, your function app needs to be running in an ASE or using Private Endpoints (preview). To learn more, see [Using Private Endpoints](#).

Next steps

- [Azure Security Baseline for Azure Functions](#)
- [Azure Functions diagnostics](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Functions runtime versions overview

Article • 05/13/2024

Azure Functions currently supports two versions of the runtime host. The following table details the currently supported runtime versions, their support level, and when they should be used:

[+] Expand table

Version	Support level	Description
4.x	GA	<p>Recommended runtime version for functions in all languages.</p> <p>Check out Supported language versions.</p>
1.x	GA (support ends September 14, 2026 ↗)	Supported only for C# apps that must use .NET Framework. This version is in maintenance mode, with enhancements provided only in later versions. Support will end for version 1.x on September 14, 2026. We highly recommend you migrate your apps to version 4.x , which supports .NET Framework 4.8, .NET 6, .NET 7, and .NET 8.

ⓘ Important

As of December 13, 2022, function apps running on versions 2.x and 3.x of the Azure Functions runtime have reached the end of extended support. For more information, see [Retired versions](#).

This article details some of the differences between supported versions, how you can create each version, and how to change the version on which your functions run.

Levels of support

There are two levels of support:

- **Generally available (GA)** - Fully supported and approved for production use.
- **Preview** - Not yet supported, but expected to reach GA status in the future.

Languages

All functions in a function app must share the same language. You choose the language of functions in your function app when you create the app. The language of your function app is maintained in the [FUNCTIONS_WORKER_RUNTIME](#) setting, and can't be changed when there are existing functions.

The following table shows the .NET versions supported by Azure Functions. Select your preferred development language at the top of the article.

The supported version of .NET depends on both your Functions runtime version and your chosen execution model:

Isolated worker model		
Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .		
v4.x		
Expand table		
Supported version	Support level	Expected community EOL date
.NET 8	GA	November 10, 2026 ↗
.NET 6	GA	November 12, 2024 ↗
.NET Framework 4.8	GA	See policy ↗

.NET 7 was previously supported on the isolated worker model but reached the end of official support on [May 14, 2024 ↗](#).

For more information, see [Guide for running C# Azure Functions in an isolated worker process](#).

For information about planned changes to language support, see [Azure roadmap ↗](#).

For information about the language versions of previously supported versions of the Functions runtime, see [Retired runtime versions](#).

Run on a specific version

The version of the Functions runtime used by published apps in Azure is dictated by the [FUNCTIONS_EXTENSION_VERSION](#) application setting. In some cases and for certain languages, other settings can apply.

By default, function apps created in the Azure portal, by the Azure CLI, or from Visual Studio tools are set to version 4.x. You can modify this version if needed. You can only downgrade the runtime version to 1.x after you create your function app but before you add any functions. Updating to a later major version is allowed even with apps that have existing functions.

Migrating existing function apps

When your app has existing functions, you must take precautions before moving to a later major runtime version. The following articles detail breaking changes between major versions, including language-specific breaking changes. They also provide you with step-by-step instructions for a successful migration of your existing function app.

- [Migrate from runtime version 3.x to version 4.x](#)
- [Migrate from runtime version 1.x to version 4.x](#)

Changing version of apps in Azure

The following major runtime version values are used:

[\[+\] Expand table](#)

Value	Runtime target
~4	4.x
~1	1.x

Important

Don't arbitrarily change this app setting, because other app setting changes and changes to your function code might be required. For existing function apps, [follow the migration instructions](#).

Pinning to a specific minor version

To resolve issues your function app could have when running on the latest major version, you have to temporarily pin your app to a specific minor version. Pinning gives you time to get your app running correctly on the latest major version. The way that you pin to a minor version differs between Windows and Linux. To learn more, see [How to target Azure Functions runtime versions](#).

Older minor versions are periodically removed from Functions. For the latest news about Azure Functions releases, including the removal of specific older minor versions, monitor [Azure App Service announcements](#).

Minimum extension versions

There's technically not a correlation between binding extension versions and the Functions runtime version. However, starting with version 4.x the Functions runtime enforces a minimum version for all trigger and binding extensions.

If you receive a warning about a package not meeting a minimum required version, you should update that NuGet package to the minimum version as you normally would. The minimum version requirements for extensions used in Functions v4.x can be found in [the linked configuration file](#).

For C# script, update the extension bundle reference in the host.json as follows:

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[4.0.0, 5.0.0)"  
    }  
}
```

Retired versions

 **Important**

[Support will end for version 1.x of the Azure Functions runtime on September 14, 2026](#). We highly recommend that you [migrate your apps to version 4.x](#) for full support.

These versions of the Functions runtime reached the end of extended support on December 13, 2022.

[Expand table](#)

Version	Current support level	Previous support level
3.x	Out-of-support	GA
2.x	Out-of-support	GA

As soon as possible, you should migrate your apps to version 4.x to obtain full support. For a complete set of language-specific migration instructions, see [Migrate apps to Azure Functions version 4.x](#).

Apps using versions 2.x and 3.x can still be created and deployed from your CI/CD DevOps pipeline, and all existing apps continue to run without breaking changes. However, your apps aren't eligible for new features, security patches, and performance optimizations. You can only get related service support after you upgrade your apps to version 4.x.

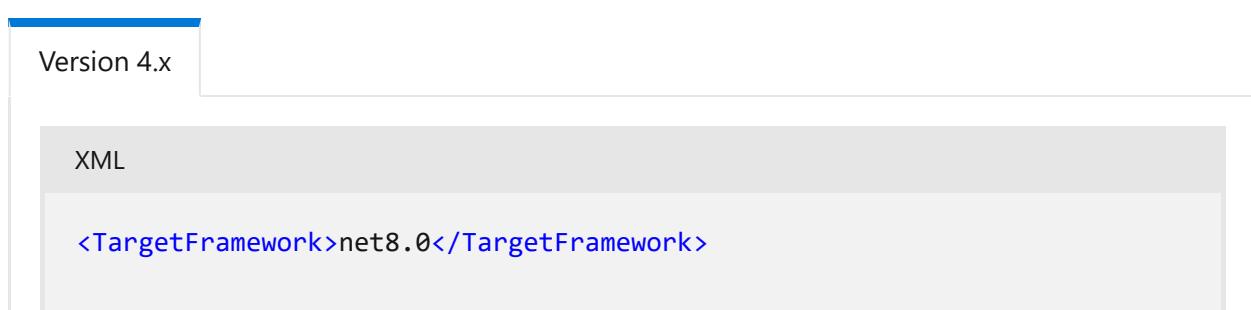
End of support for versions 2.x and 3.x is due to the end of support for .NET Core 3.1, which they had as a core dependency. This requirement affects all [languages supported by Azure Functions](#).

Locally developed application versions

You can make the following updates to function apps to locally change the targeted versions.

Visual Studio runtime versions

In Visual Studio, you select the runtime version when you create a project. Azure Functions tools for Visual Studio supports the two major runtime versions. The correct version is used when debugging and publishing based on project settings. The version settings are defined in the `.csproj` file in the following properties:



You can choose `net8.0`, `net6.0`, or `net48` as the target framework if you are using the [isolated worker model](#). If you are using the [in-process model](#), you can only choose `net6.0`, and you must include the `Microsoft.NET.Sdk.Functions` extension set to at least `4.0.0`.

.NET 7 was previously supported on the isolated worker model but reached the end of official support on [May 14, 2024](#).

Visual Studio Code and Azure Functions Core Tools

Azure Functions Core Tools is used for command-line development and also by the [Azure Functions extension](#) for Visual Studio Code. For more information, see [Install the Azure Functions Core Tools](#).

For Visual Studio Code development, you could also need to update the user setting for the `azureFunctions.projectRuntime` to match the version of the tools installed. This setting also updates the templates and languages used during function app creation.

Bindings

Starting with version 2.x, the runtime uses a new [binding extensibility model](#) that offers these advantages:

- Support for third-party binding extensions.
- Decoupling of runtime and bindings. This change allows binding extensions to be versioned and released independently. You can, for example, opt to upgrade to a version of an extension that relies on a newer version of an underlying SDK.
- A lighter execution environment, where only the bindings in use are known and loaded by the runtime.

Except for HTTP and timer triggers, all bindings must be explicitly added to the function app project, or registered in the portal. For more information, see [Register binding extensions](#).

The following table shows which bindings are supported in each runtime version.

This table shows the bindings that are supported in the major versions of the Azure Functions runtime:

[\[\]](#) Expand table

Type	1.x ¹	2.x and higher ²	Trigger	Input	Output
Blob storage	✓	✓	✓	✓	✓
Azure Cosmos DB	✓	✓	✓	✓	✓
Azure Data Explorer		✓		✓	✓
Azure SQL		✓	✓	✓	✓
Dapr ⁴		✓	✓	✓	✓
Event Grid	✓	✓	✓		✓
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
IoT Hub	✓	✓	✓		
Kafka ³		✓	✓		✓
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
Redis		✓	✓		
RabbitMQ ³		✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓	✓	✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ Support will end for version 1.x of the Azure Functions runtime on September 14, 2026 [↗](#). We highly recommend that you [migrate your apps to version 4.x](#) for full support.

² Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

³ Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

⁴ Supported only in Kubernetes, IoT Edge, and other self-hosted modes only.

Function app timeout duration

The timeout duration for functions in a function app is defined by the `functionTimeout` property in the [host.json](#) project file. This property applies specifically to function executions. After the trigger starts function execution, the function needs to return/respond within the timeout duration. For more information, see [Improve Azure Functions performance and reliability](#).

The following table shows the default and maximum values (in minutes) for specific plans:

[+] [Expand table](#)

Plan	Default	Maximum ¹
Consumption plan	5	10
Flex Consumption plan	30	Unlimited ³
Premium plan	30 ²	Unlimited ³
Dedicated plan	30 ²	Unlimited ³

¹ Regardless of the function app timeout setting, 230 seconds is the maximum amount of time that an HTTP triggered function can take to respond to a request. This is because of the [default idle timeout of Azure Load Balancer](#). For longer processing times, consider using the [Durable Functions](#) async pattern or [defer the actual work and return an immediate response](#).

² The default timeout for version 1.x of the Functions runtime is *unlimited*.

³ Guaranteed for up to 60 minutes. [OS and runtime patching](#), vulnerability patching, and [scale in behaviors](#) can still cancel function executions so [ensure to write robust functions](#). ⁴ In a Flex Consumption plan, the host doesn't enforce an execution time limit. However, there are currently no guarantees because the platform might need to terminate your instances during scale-in, deployments, or to apply updates.

Next steps

For more information, see the following resources:

- [Code and test Azure Functions locally](#)
- [How to target Azure Functions runtime versions](#)
- [Release notes ↗](#)

Azure Functions Consumption plan hosting

Article • 05/21/2024

When you're using the Consumption plan, instances of the Azure Functions host are dynamically added and removed based on the number of incoming events. The Consumption plan, along with the [Flex Consumption plan](#), is a fully *serverless* hosting option for Azure Functions.

Benefits

The Consumption plan scales automatically, even during periods of high load. When running functions in a Consumption plan, you're charged for compute resources only when your functions are running. On a Consumption plan, a function execution times out after a configurable period of time.

For a comparison of the Consumption plan against the other plan and hosting types, see [function scale and hosting options](#).

💡 Tip

If you want the benefits of dynamic scale and execution-only billing, but also need to integrate your app with virtual networks, you should instead consider hosting your app in the [Flex Consumption plan](#).

Billing

Billing is based on number of executions, execution time, and memory used. Usage is aggregated across all functions within a function app. For more information, see the [Azure Functions pricing page](#).

To learn more about how to estimate costs when running in a Consumption plan, see [Understanding Consumption plan costs](#).

Create a Consumption plan function app

When you create a function app in the Azure portal, the Consumption plan is the default. When using APIs to create your function app, you don't have to first create an

App Service plan as you do with Premium and Dedicated plans.

In Consumption plan hosting, each function app typically runs in its own plan. In the Azure portal or in code, you may also see the Consumption plan referred to as `Dynamic` or `Y1`.

Use the following links to learn how to create a serverless function app in a Consumption plan, either programmatically or in the Azure portal:

- [Azure CLI](#)
- [Azure portal](#)
- [Azure Resource Manager template](#)

You can also create function apps in a Consumption plan when you publish a Functions project from [Visual Studio Code](#) or [Visual Studio](#).

Multiple apps in the same plan

The general recommendation is for each function app to have its own Consumption plan. However, if needed, function apps in the same region can be assigned to the same Consumption plan. Keep in mind that there is a [limit to the number of function apps that can run in a Consumption plan](#). Function apps in the same plan still scale independently of each other.

Next steps

- [Azure Functions hosting options](#)
- [Event-driven scaling in Azure Functions](#)

Azure Functions Flex Consumption plan hosting

Article • 10/17/2024

Flex Consumption is a Linux-based Azure Functions hosting plan that builds on the Consumption *pay for what you use* serverless billing model. It gives you more flexibility and customizability by introducing private networking, instance memory size selection, and fast/large scale-out features still based on a *serverless* model.

Important

The Flex Consumption plan is currently in preview. For a list of current limitations when using this hosting plan, see [Considerations](#). For current information about billing during the preview, see [Billing](#).

You can review end-to-end samples that feature the Flex Consumption plan in the [Flex Consumption plan samples repository](#).

Benefits

The Flex Consumption plan builds on the strengths of the Consumption plan, which include dynamic scaling and execution-based billing. With Flex Consumption, you also get these extra features:

- [Always-ready instances](#)
- [Virtual network integration](#)
- Fast scaling based on concurrency for both HTTP and non-HTTP apps
- Multiple choices for instance memory sizes

This table helps you directly compare the features of Flex Consumption with the Consumption hosting plan:

 Expand table

Feature	Consumption	Flex Consumption
Scale to zero	 Yes	 Yes
Scale behavior	Event driven	Event driven (fast)
Virtual networks	 Not supported	 Supported

Feature	Consumption	Flex Consumption
Dedicated compute (mitigate cold starts)	✖ None	<input checked="" type="checkbox"/> Always ready instances (optional)
Billing	Execution-time only	Execution-time + always-ready instances
Scale-out instances (max)	200	1000

For a complete comparison of the Flex Consumption plan against the Consumption plan and all other plan and hosting types, see [function scale and hosting options](#).

Virtual network integration

Flex Consumption expands on the traditional benefits of Consumption plan by adding support for [virtual network integration](#). When your apps run in a Flex Consumption plan, they can connect to other Azure services secured inside a virtual network. All while still allowing you to take advantage of serverless billing and scale, together with the scale and throughput benefits of the Flex Consumption plan. For more information, see [Enable virtual network integration](#).

Instance memory

When you create your function app in a Flex Consumption plan, you can select the memory size of the instances on which your app runs. See [Billing](#) to learn how instance memory sizes affect the costs of your function app.

Currently, Flex Consumption offers instance memory size options of both 2,048 MB and 4,096 MB.

When deciding on which instance memory size to use with your apps, here are some things to consider:

- The 2,048-MB instance memory size is the default and should be used for most scenarios. Use the 4,096-MB instance memory size for scenarios where your app requires more concurrency or higher processing power. For more information, see [Configure instance memory](#).
- You can change the instance memory size at any time. For more information, see [Configure instance memory](#).
- Instance resources are shared between your function code and the Functions host.
- The larger the instance memory size, the more each instance can handle as far as concurrent executions or more intensive CPU or memory workloads. Specific scale

decisions are workload-specific.

- The default concurrency of HTTP triggers depends on the instance memory size. For more information, see [HTTP trigger concurrency](#).
- Available CPUs and network bandwidth are provided proportional to a specific instance size.

Per-function scaling

Concurrency is a key factor that determines how Flex Consumption function apps scale. To improve the scale performance of apps with various trigger types, the Flex Consumption plan provides a more deterministic way of scaling your app on a per-function basis.

This *per-function scaling* behavior is a part of the hosting platform, so you don't need to configure your app or change the code. For more information, see [Per-function scaling](#) in the Event-driven scaling article.

In per-function scaling, decisions are made for certain function triggers based on group aggregations. This table shows the defined set of function scale groups:

[+] [Expand table](#)

Scale groups	Triggers in group	Settings value
HTTP triggers	HTTP trigger SignalR trigger	<code>http</code>
Blob storage triggers (Event Grid-based)	Blob storage trigger	<code>blob</code>
Durable Functions	Orchestration trigger Activity trigger Entity trigger	<code>durable</code>

All other functions in the app are scaled individually in their own set of instances, which are referenced using the convention `function:<NAMED_FUNCTION>`.

Always ready instances

Flex Consumption includes an *always ready* feature that lets you choose instances that are always running and assigned to each of your per-function scale groups or functions. This is a great option for scenarios where you need to have a minimum number of

instances always ready to handle requests, for example, to reduce your application's cold start latency. The default is 0 (zero).

For example, if you set always ready to 2 for your HTTP group of functions, the platform keeps two instances always running and assigned to your app for your HTTP functions in the app. Those instances are processing your function executions, but depending on concurrency settings, the platform scales beyond those two instances with on-demand instances.

To learn how to configure always ready instances, see [Set always ready instance counts](#).

Concurrency

Concurrency refers to the number of parallel executions of a function on an instance of your app. You can set a maximum number of concurrent executions that each instance should handle at any given time. For more information, see [HTTP trigger concurrency](#).

Concurrency has a direct effect on how your app scales because at lower concurrency levels, you need more instances to handle the event-driven demand for a function. While you can control and fine tune the concurrency, we provide defaults that work for most cases. To learn how to set concurrency limits for HTTP trigger functions, see [Set HTTP concurrency limits](#).

Deployment

Deployments in the Flex Consumption plan follow a single path. After your project code is built and zipped into an application package, it is deployed to a blob storage container. On startup, your app gets the package and runs your function code from this package. By default, the same storage account used to store internal host metadata (AzureWebJobsStorage) is also used as the deployment container. However, you can use an alternative storage account or choose your preferred authentication method by [configuring your app's deployment settings](#). In streamlining the deployment path, there's no longer the need for app settings to influence deployment behavior.

Billing

There are two modes by which your costs are determined when running your apps in the Flex Consumption plan. Each mode is determined on a per-instance basis.

Billing mode	Description
On Demand	<p>When running in <i>on demand</i> mode, you are billed only for the amount of time your function code is executing on your available instances. In on demand mode, no minimum instance count is required. You're billed for:</p> <ul style="list-style-type: none"> The total amount of memory provisioned while each on demand instance is <i>actively</i> executing functions (in GB-seconds), minus a free grant of GB-s per month. The total number of executions, minus a free grant (number) of executions per month.
Always ready	<p>You can configure one or more instances, assigned to specific trigger types (HTTP/Durable/Blob) and individual functions, that are always available to be able handle requests. When you have any always ready instances enabled, you're billed for:</p> <ul style="list-style-type: none"> The total amount of memory provisioned across all of your always ready instances, known as the <i>baseline</i> (in GB-seconds). The total amount of memory provisioned during the time each always ready instance is <i>actively</i> executing functions (in GB-seconds). The total number of executions. <p>In always ready billing, there are no free grants.</p>

The minimum billable execution period for both execution modes is 1,000 ms. Past that, the billable activity period is rounded up to the nearest 100 ms. You can find details on the Flex Consumption plan billing meters in the [Monitoring reference](#).

For details about how costs are calculated when you run in a Flex Consumption plan, including examples, see [Consumption-based costs](#).

For the most up-to-date information on execution pricing, always ready baseline costs, and free grants for on demand executions, see the [Azure Functions pricing page](#).

Supported language stack versions

This table shows the language stack versions that are currently supported for Flex Consumption apps:

[Expand table](#)

Language stack	Required version
C# (isolated process mode) ¹	.NET 8 ²

Language stack	Required version
Java	Java 11, Java 17
Node.js	Node 20
PowerShell	PowerShell 7.4
Python	Python 3.10, Python 3.11

¹[C# in-process mode](#) isn't supported. You instead need to [migrate your .NET code project to run in the isolated worker model](#).

²Requires version [1.20.0](#) or later of [Microsoft.Azure.Functions.Worker](#) and version [1.16.2](#) or later of [Microsoft.Azure.Functions.Worker.Sdk](#).

Regional subscription memory quotas

Currently in preview each region in a given subscription has a memory limit of [512,000 MB](#) for all instances of apps running on Flex Consumption plans. This means that, in a given subscription and region, you could have any combination of instance memory sizes and counts, as long as they stay under the quota limit. For example, each the following examples would mean the quota has been reached and the apps would stop scaling:

- You have one 2,048 MB app scaled to 100 and a second 2,048 MB app scaled to 150 instances
- You have one 2,048 MB app that scaled out to 250 instances
- You have one 4,096 MB app that scaled out to 125 instances
- You have one 4,096 MB app scaled to 100 and one 2,048 MB app scaled to 50 instances

This quota can be increased to allow your Flex Consumption apps to scale further, depending on your requirements. If your apps require a larger quota please create a support ticket.

Deprecated properties and settings

In Flex Consumption, many of the standard application settings and site configuration properties used in Bicep, ARM templates, and overall control plane are deprecated or have moved and shouldn't be used when automating function app resource creation. For more information, see [Flex Consumption plan deprecations](#).

Considerations

Keep these other considerations in mind when using Flex Consumption plan during the current preview:

- **Host:** There is a 30 seconds timeout for the app initialization. If your function app takes longer than 30 seconds to start you will see gRPC related System.TimeoutException entries. This timeout will be configurable and a more clear exception will be implemented as part of [this host work item](#).
- **Durable Functions:** Due to the per function scaling nature of Flex Consumption, to ensure the best performance for Durable Functions we recommend setting the [Always Ready instance count](#) for the `durable` group to `1`. Also, with the Azure Storage provider, consider reducing the [queue polling interval](#) to 10 seconds or less. Only Azure Storage is supported as a backend storage providers for Flex Consumption hosted durable functions.
- **VNet Integration** Ensure that the `Microsoft.App` Azure resource provider is enabled for your subscription by [following these instructions](#). The subnet delegation required by Flex Consumption apps is `Microsoft.App/environments`.
- **Triggers:** All triggers are fully supported except for Kafka and Azure SQL triggers. The Blob storage trigger only supports the [Event Grid source](#). Non-C# function apps must use version `[4.0.0, 5.0.0)` of the [extension bundle](#), or a later version.
- **Regions:** Not all regions are currently supported. To learn more, see [View currently supported regions](#).
- **Deployments:** Deployment slots are not currently supported.
- **Scale:** The lowest maximum scale in preview is `40`. The highest currently supported value is `1000`.
- **Managed dependencies:** [Managed dependencies in PowerShell](#) aren't supported by Flex Consumption. You must instead [define your own custom modules](#).
- **Diagnostic settings:** Diagnostic settings are not currently supported.
- **Certificates:** Loading certificates with the WEBSITE_LOAD_CERTIFICATES app setting is currently not supported.
- **Key Vault References:** Key Vault references in app settings do not work when Key Vault is network access restricted, even if the function app has Virtual Network integration. The current workaround is to directly reference the Key Vault in code and read the required secrets.

Related articles

[Azure Functions hosting options Create and manage function apps in the Flex Consumption plan](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Functions Premium plan

Article • 04/07/2024

The Azure Functions Elastic Premium plan is a dynamic scale hosting option for function apps. For other hosting plan options, see the [hosting plan article](#).

Important

Azure Functions can run on the Azure App Service platform. In the App Service platform, plans that host Premium plan function apps are referred to as *Elastic* Premium plans, with SKU names like `EP1`. If you choose to run your function app on a Premium plan, make sure to create a plan with an SKU name that starts with "E", such as `EP1`. App Service plan SKU names that start with "P", such as `P1V2` (Premium V2 Small plan), are actually [Dedicated hosting plans](#). Because they are Dedicated and not Elastic Premium, plans with SKU names starting with "P" won't scale dynamically and may increase your costs.

Premium plan hosting provides the following benefits to your functions:

- Avoid cold starts with warm instances.
- Virtual network connectivity.
- Supports [longer runtime durations](#).
- [Choice of Premium instance sizes](#).
- More predictable pricing, compared with the Consumption plan.
- High-density app allocation for plans with multiple function apps.
- Supports [Linux container deployments](#).

When you're using the Premium plan, instances of the Azure Functions host are added and removed based on the number of incoming events, just like the [Consumption plan](#). Multiple function apps can be deployed to the same Premium plan, and the plan allows you to configure compute instance size, base plan size, and maximum plan size.

Billing

Billing for the Premium plan is based on the number of core seconds and memory allocated across instances. This billing differs from the Consumption plan, which is billed based on per-second resource consumption and executions. There's no execution charge with the Premium plan. This billing results in a minimum monthly cost per active plan, regardless if the function is active or idle. Keep in mind that all function apps in a

Premium plan share allocated instances. To learn more, see the [Azure Functions pricing page](#).

 **Note**

Every premium plan has at least one active (billed) instance at all times.

Create a Premium plan

When you create a function app in the Azure portal, the Consumption plan is the default. To create a function app that runs in a Premium plan, you must explicitly create or choose an Azure Functions Premium hosting plan using one of the *Elastic Premium* SKUs. The function app you create is then hosted in this plan. The Azure portal makes it easy to create both the Premium plan and the function app at the same time. You can run more than one function app in the same Premium plan, but they must both run on the same operating system (Windows or Linux).

The following articles show you how to programmatically create a function app with a Premium plan:

- [Azure CLI](#)
- [Azure Resource Manager template](#)

Eliminate cold starts

When events or executions don't occur in the Consumption plan, your app might scale to zero instances. When new events come in, a new instance with your app running on it must be specialized. Specializing new instances takes time, depending on the app. This extra latency on the first call is often called app *cold start*.

Premium plan provides two features that work together to effectively eliminate cold starts in your functions: *always ready instances* and *prewarmed instances*. Always ready instances are a category of preallocated instances unaffected by scaling, and the prewarmed ones are a buffer as you scale due to HTTP events.

When events begin to trigger the app, they're first routed to the always ready instances. As the function becomes active due to HTTP events, other instances are warmed as a buffer. These buffered instances are called prewarmed instances. This buffer reduces cold start for new instances required during scale.

Always ready instances

In the Premium plan, you can have your app always ready on a specified number of instances. Your app runs continuously on those instances, regardless of load. If load exceeds what your always ready instances can handle, more instances are added as necessary, up to your specified maximum.

This app-level setting also controls your plan's minimum instances. For example, consider having three function apps in the same Premium plan. When two of your apps have always ready instance count set to one and in a third instance it's set to five, the minimum number for your whole plan is five. This also reflects the minimum number of instances for which your plan is billed. The maximum number of always ready instances we support per app is 20.

Portal

You can configure the number of always ready instances in the Azure portal by selected your **Function App**, going to the **Platform Features** tab, and selecting the **Scale Out** options. In the function app edit window, always ready instances are specific to that app.

The screenshot shows the 'Elastic Scale out' section of the Azure portal. It includes a note about controlling scale bounds, a 'Plan Scale out' section with 'Maximum Burst' set to 20 and 'Minimum Instances' set to 1, and an 'App Scale out' section with 'Always Ready Instances' set to 2 and 'Enforce Scale Out Limit' set to 'No'. A tooltip for minimum instances explains that explicitly configuring instances is recommended for advanced scenarios.

Elastic Scale out

This allows you to control the bounds that your Premium plan can scale within. [Learn more](#)

Plan Scale out

Maximum Burst [?](#) 20

Minimum Instances [?](#) 1

Explicitly configuring the number of instances for your plan is only recommended for advanced scenarios, and the configured value will only be honored if it is greater than the always ready instance maximum for apps in your plan. [Learn more](#)

App Scale out

Always Ready Instances [?](#) 2

Enforce Scale Out Limit [?](#) No Yes

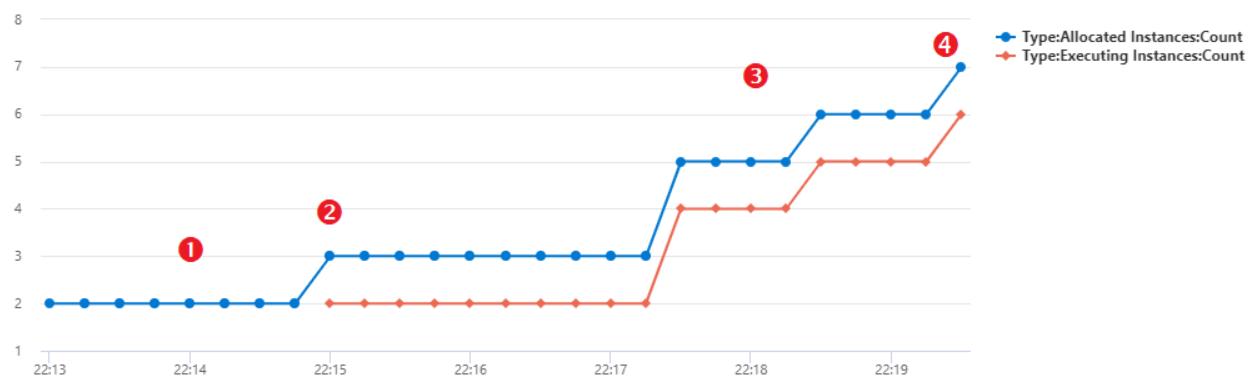
Prewarmed instances

The prewarmed instance count setting provides warmed instances as a buffer during HTTP scale and activation events. Prewarmed instances continue to buffer until the maximum scale-out limit is reached. The default prewarmed instance count is 1 and, for most scenarios, this value should remain as 1.

Consider a less-common scenario, such as an app running in a custom container. Because custom containers have a long warm-up, you could consider increasing this buffer of prewarmed instances. A prewarmed instance becomes active only after all active instances are in use.

You can also define a warmup trigger that is run during the prewarming process. You can use a warmup trigger to preload custom dependencies during the prewarming process so your functions are ready to start processing requests immediately. To learn more, see [Azure Functions warmup trigger](#).

Consider this example of how always-ready instances and prewarmed instances work together. A premium function app has two always ready instances configured, and the default of one prewarmed instance.



1. When the app is idle and no events are triggering, the app is provisioned and running with two instances. At this time, you're billed for the two always ready instances but aren't billed for a prewarmed instance as no prewarmed instance is allocated.
2. As your application starts receiving HTTP traffic, requests are load balanced across the two always-ready instances. As soon as those two instances start processing events, an instance gets added to fill the prewarmed buffer. The app is now running with three provisioned instances: the two always ready instances, and the third prewarmed and inactive buffer. You're billed for the three instances.
3. As load increases and your app needs more instances to handle HTTP traffic, that prewarmed instance is swapped to an active instance. HTTP load is now routed to all three instances, and a fourth instance is instantly provisioned to fill the prewarmed buffer.
4. This sequence of scaling and prewarming continues until the maximum instance count for the app is reached or load decreases causing the platform to scale back in after a period. No instances are prewarmed or activated beyond the maximum.

You can't change the prewarmed instance count setting in the portal, you must instead use the Azure CLI or Azure PowerShell.

Maximum function app instances

In addition to the [plan maximum burst count](#), you can configure a per-app maximum. The app maximum can be configured using the [app scale limit](#). The maximum app scale out limit cannot exceed the maximum burst instances of the plan.

Private network connectivity

Function apps deployed to a Premium plan can take advantage of [virtual network integration for web apps](#). When configured, your app can communicate with resources within your virtual network or secured via service endpoints. IP restrictions are also available on the app to restrict incoming traffic.

When assigning a subnet to your function app in a Premium plan, you need a subnet with enough IP addresses for each potential instance. We require an IP block with at least 100 available addresses.

For more information, see [integrate your function app with a virtual network](#).

Rapid elastic scale

More compute instances are automatically added for your app using the same rapid scaling logic as the Consumption plan. Apps in the same App Service Plan scale independently from one another based on the needs of an individual app. However, Functions apps in the same App Service Plan share VM resources to help reduce costs, when possible. The number of apps associated with a VM depends on the footprint of each app and the size of the VM.

To learn more about how scaling works, see [Event-driven scaling in Azure Functions](#).

Longer run duration

Functions in a Consumption plan are limited to 10 minutes for a single execution. In the Premium plan, the run duration defaults to 30 minutes to prevent runaway executions. However, you can [modify the host.json configuration](#) to make the duration unbounded for Premium plan apps, with the following limitations:

- Platform upgrades can trigger a managed shutdown and halt the function execution.
- Platform outages can cause an unhandled shutdown and halt the function execution.
- There's an idle timer that stops the worker after 60 minutes with no new executions.
- [Scale-in behavior](#) can cause worker shutdown after 60 minutes.
- [Slot swaps](#) can terminate executions on the source and target slots during the swap.

Migration

If you have an existing function app, you can use Azure CLI commands to migrate your app between a Consumption plan and a Premium plan on Windows. The specific commands depend on the direction of the migration. To learn more, see [Plan migration](#).

This migration isn't supported on Linux.

Premium plan settings

When you create the plan, there are two plan size settings: the minimum number of instances (or plan size) and the maximum burst limit.

If your app requires instances beyond the always-ready instances, it can continue to scale out until the number of instances hits the plan maximum burst limit, or the app maximum scale out limit if configured. You're billed for instances only while they're running and allocated to you, on a per-second basis. The platform makes its best effort at scaling your app out to the defined maximum limits.

Portal

You can configure the plan size and maximums in the Azure portal by selecting the **Scale Out** options under **Settings** of a function app deployed to that plan.



Elastic Scale out

This allows you to control the bounds that your Premium plan can scale within. [Learn more](#)

Plan Scale out

Maximum Burst [\(i\)](#)



Minimum Instances [\(i\)](#)



Explicitly configuring the number of instances for your plan is only recommended for advanced scenarios, and the configured value will only be honored if it is greater than the always ready instance maximum for apps in your plan. [Learn more](#)

App Scale out

Always Ready Instances [\(i\)](#)



Enforce Scale Out Limit [\(i\)](#)

No Yes

The minimum for every Premium plan is at least one instance. The actual minimum number of instances is determined for you based on the always ready instances requested by apps in the plan. For example, if app A requests five always ready instances, and app B requests two always ready instances in the same plan, the minimum plan size is determined as five. App A is running on all five, and app B is only running on 2.

Important

You are charged for each instance allocated in the minimum instance count regardless if functions are executing or not.

In most circumstances, this autocalculated minimum is sufficient. However, scaling beyond the minimum occurs at a best effort. It's possible, though unlikely, that at a specific time scale-out could be delayed if other instances are unavailable. By setting a minimum higher than the autocalculated minimum, you reserve instances in advance of scale-out.

Portal

You can configure the minimum instances in the Azure portal by selecting the **Scale Out** options under **Settings** of a function app deployed to that plan.

Elastic Scale out

This allows you to control the bounds that your Premium plan can scale within. [Learn more](#)

Plan Scale out

Maximum Burst 20

Minimum Instances 1

Explicitly configuring the number of instances for your plan is only recommended for advanced scenarios, and the configured value will only be honored if it is greater than the always ready instance maximum for apps in your plan. [Learn more](#)

App Scale out

Always Ready Instances 2

Enforce Scale Out Limit No Yes

Available instance SKUs

When creating or scaling your plan, you can choose between three instance sizes. You're billed for the total number of cores and memory provisioned, per second that each instance is allocated to you. Your app can automatically scale out to multiple instances as needed.

[Expand table](#)

SKU	Cores	Memory	Storage
EP1	1	3.5GB	250GB
EP2	2	7GB	250GB
EP3	4	14GB	250GB

Memory usage considerations

Running on a machine with more memory doesn't always mean that your function app uses all available memory.

For example, a JavaScript function app is constrained by the default memory limit in Node.js. To increase this fixed memory limit, add the app setting

`languageWorkers:node:arguments` with a value of `--max-old-space-size=<max memory in MB>`.

And for plans with more than 4 GB of memory, ensure the Bitness Platform Setting is set to `64 Bit` under [General Settings](#).

Region max scale-out

These are the currently supported maximum scale-out values for a single plan in each region and OS configuration:

[Expand table](#)

Region	Windows	Linux
Australia Central	100	20
Australia Central 2	100	Not Available
Australia East	100	40
Australia Southeast	100	20
Brazil South	100	20
Canada Central	100	100
Central India	100	20
Central US	100	100
China East 2	100	20
China North 2	100	20
East Asia	100	20
East US	100	100
East US 2	100	100
France Central	100	60
Germany West Central	100	20
Israel Central	100	20
Italy North	100	20
Japan East	100	20
Japan West	100	20
Jio India West	100	20
Korea Central	100	20
Korea South	40	20

Region	Windows	Linux
North Central US	100	20
North Europe	100	100
Norway East	100	20
South Africa North	100	20
South Africa West	20	20
South Central US	100	100
South India	100	Not Available
Southeast Asia	100	20
Switzerland North	100	20
Switzerland West	100	20
UAE North	100	20
UK South	100	100
UK West	100	20
USGov Arizona	100	20
USGov Texas	100	Not Available
USGov Virginia	100	20
West Central US	100	20
West Europe	100	100
West India	100	20
West US	100	100
West US 2	100	20
West US 3	100	20

For more information, see the [complete regional availability of Azure Functions](#).

Next steps

- [Understand Azure Functions hosting options](#)

- Event-driven scaling in Azure Functions

Dedicated hosting plans for Azure Functions

Article • 01/30/2023

This article is about hosting your function app with dedicated resources in an App Service plan, including in an App Service Environment (ASE). For other hosting options, see the [hosting plan article](#).

An App Service plan defines a set of dedicated compute resources for an app to run. These dedicated compute resources are analogous to the [server farm](#) in conventional hosting. One or more function apps can be configured to run on the same computing resources (App Service plan) as other App Service apps, such as web apps. The dedicated App Service plans supported for function app hosting include Basic, Standard, Premium, and Isolated SKUs. For details about how the App Service plan works, see the [Azure App Service plans in-depth overview](#).

ⓘ Important

Free and Shared tier App Service plans aren't supported by Azure Functions. For a lower-cost option hosting your function executions, you should instead consider the [Consumption plan](#), where you are billed based on function executions.

Consider a dedicated App Service plan in the following situations:

- You have existing, underutilized VMs that are already running other App Service instances.
- You want to provide a custom image on which to run your functions.

Billing

You pay for function apps in an App Service Plan as you would for other App Service resources. This differs from Azure Functions [Consumption plan](#) or [Premium plan](#) hosting, which have consumption-based cost components. You are billed only for the plan, regardless of how many function apps or web apps run in the plan. To learn more, see the [App Service pricing page](#).

Always On

If you run on an App Service plan, you should enable the **Always on** setting so that your function app runs correctly. On an App Service plan, the functions runtime goes idle after a few minutes of inactivity, so only HTTP triggers will "wake up" your functions. The **Always on** setting is available only on an App Service plan. On a Consumption plan, the platform activates function apps automatically.

Even with Always On enabled, the execution timeout for individual functions is controlled by the `functionTimeout` setting in the [host.json](#) project file.

Scaling

Using an App Service plan, you can manually scale out by adding more VM instances. You can also enable autoscale, though autoscale will be slower than the elastic scale of the Premium plan. For more information, see [Scale instance count manually or automatically](#). You can also scale up by choosing a different App Service plan. For more information, see [Scale up an app in Azure](#).

ⓘ Note

When running JavaScript (Node.js) functions on an App Service plan, you should choose a plan that has fewer vCPUs. For more information, see [Choose single-core App Service plans](#).

App Service Environments

Running in an App Service Environment (ASE) lets you fully isolate your functions and take advantage of higher numbers of instances than an App Service Plan. To get started, see [Introduction to the App Service Environments](#).

If you just want to run your function app in a virtual network, you can do this using the [Premium plan](#). To learn more, see [Establish Azure Functions private site access](#).

Next steps

- [Azure Functions hosting options](#)
- [Azure App Service plan overview](#)

Linux container support in Azure Functions

Article • 05/21/2024

When you plan and develop your individual functions to run in Azure Functions, you are typically focused on the code itself. Azure Functions makes it easy to deploy just your code project to a function app in Azure. When you deploy your code project to a function app that runs on Linux, the project runs in a container that is created for you automatically. This container is managed by Functions.

Functions also supports containerized function app deployments. In a containerized deployment, you create your own function app instance in a local Docker container from a supported based image. You can then deploy this *containerized* function app to a hosting environment in Azure. Creating your own function app container lets you customize or otherwise control the immediate runtime environment of your function code.

Container hosting options

There are several options for hosting your containerized function apps in Azure:

[+] Expand table

Hosting option	Benefits
Azure Container Apps	Azure Functions provides integrated support for developing, deploying, and managing containerized function apps on Azure Container Apps . Use Azure Container Apps to host your function app containers when you need to run your event-driven functions in Azure in the same environment as other microservices, APIs, websites, workflows, or any container hosted programs. Container Apps hosting lets you run your functions in a managed Kubernetes-based environment with built-in support for open-source monitoring, mTLS, Dapr, and KEDA. Container Apps uses the power of the underlying Azure Kubernetes Service (AKS) while removing the complexity of having to work with Kubernetes APIs.
Azure Arc-enabled Kubernetes clusters (preview)	You can host your function apps on Azure Arc-enabled Kubernetes clusters as either a code-only deployment or in a custom Linux container . Azure Arc lets you attach Kubernetes clusters so that you can manage and configure them in Azure. <i>Hosting Azure Functions containers on Azure Arc-enabled Kubernetes clusters is currently in preview.</i>

Hosting option	Benefits
Azure Functions	You can deploy your containerized function apps to run in either an Elastic Premium plan or a Dedicated plan . Premium plan hosting provides you with the benefits of dynamic scaling. You might want to use Dedicated plan hosting to take advantage of existing unused App Service plan resources.
Kubernetes	Because the Azure Functions runtime provides flexibility in hosting where and how you want, you can host and manage your function app containers directly in Kubernetes clusters. KEDA (Kubernetes-based Event Driven Autoscaling) pairs seamlessly with the Azure Functions runtime and tooling to provide event driven scale in Kubernetes. Just keep in mind that running your containerized function apps on Kubernetes, either by using KEDA or by direct deployment, is an open-source effort that you can use free of cost, with best-effort support provided by contributors and from the community.

Getting started

Use these links to get started working with Azure Functions in Linux containers:

[\[\] Expand table](#)

I want to...	See article:
Create my first containerized functions	Create a function app in a local Linux container
Create and deploy functions to Azure Container Apps	Create your first containerized functions on Azure Container Apps
Create and deploy containerized functions to Azure Functions	Create your first containerized Azure Functions
Create and deploy functions to Azure Arc-enabled Kubernetes	Create your first containerized Azure Functions on Azure Arc (preview)

Related articles

- [Working with containers and Azure Functions](#)

[Azure Arc-enabled Kubernetes clusters]

Azure Container Apps hosting of Azure Functions

Article • 07/04/2024

Azure Functions provides integrated support for developing, deploying, and managing containerized function apps on [Azure Container Apps](#). Use Azure Container Apps to host your function app containers when you need to run your event-driven functions in Azure in the same environment as other microservices, APIs, websites, workflows, or any container hosted programs. Container Apps hosting lets you run your functions in a fully managed, Kubernetes-based environment with built-in support for open-source monitoring, mTLS, Dapr, and Kubernetes Event-driven Autoscaling (KEDA).

You can write your function code in any [language stack supported by Functions](#). You can use the same Functions triggers and bindings with event-driven scaling. You can also use existing Functions client tools and the Azure portal to create containers, deploy function app containers to Container Apps, and configure continuous deployment.

Container Apps integration also means that network and observability configurations, which are defined at the Container App environment level, apply to your function app as they do to all microservices running in a Container Apps environment. You also get the other cloud-native capabilities of Container Apps, including KEDA, Dapr, Envoy. You can still use Application Insights to monitor your functions executions, and your function app can access the same virtual networking resources provided by the environment.

For a general overview of container hosting options for Azure Functions, see [Linux container support in Azure Functions](#).

Hosting and workload profiles

There are two primary hosting plans for Container Apps, a serverless [Consumption plan](#) and a [Dedicated plan](#), which uses workload profiles to better control your deployment resources. A workload profile determines the amount of compute and memory resources available to container apps deployed in an environment. These profiles are configured to fit the different needs of your applications.

The Consumption workload profile is the default profile added to every Workload profiles environment type. You can add Dedicated workload profiles to your environment as you create an environment or after it's created. To learn more about workload profiles, see [Workload profiles in Azure Container Apps](#).

Container Apps hosting of containerized function apps is supported in all [regions that support Container Apps](#).

If your app doesn't have specific hardware requirements, you can run your environment either in a Consumption plan or in a Dedicated plan using the default Consumption workload profile. When running functions on Container Apps, you're charged only for the Container Apps usage. For more information, see the [Azure Container Apps pricing page](#).

Azure Functions on Azure Container Apps supports GPU-enabled hosting in the Dedicated plan with workload profiles.

To learn how to create and deploy a function app container to Container Apps in the default Consumption plan, see [Create your first containerized functions on Azure Container Apps](#).

To learn how to create a Container Apps environment with workload profiles and deploy a function app container to a specific workload, see [Container Apps workload profiles](#).

Functions in containers

To use Container Apps hosting, your code must run on a function app in a Linux container that you create and maintain. Functions maintains a set of [language-specific base images](#) that you can use to generate your containerized function apps.

When you create a code project using [Azure Functions Core Tools](#) and include the `--docker option`, Core Tools generates the Dockerfile with the correct base image, which you can use as a starting point when creating your container.

Important

When creating your own containers, you are required to keep the base image of your container updated to the latest supported base image. Supported base images for Azure Functions are language-specific and are found in the [Azure Functions base image repos](#).

The Functions team is committed to publishing monthly updates for these base images. Regular updates include the latest minor version updates and security fixes for both the Functions runtime and languages. You should regularly update your container from the latest base image and redeploy the updated version of your container.

When you make changes to your functions code, you must rebuild and republish your container image. For more information, see [Update an image in the registry](#).

Deployment options

Azure Functions currently supports the following methods of deploying a containerized function app to Azure Container Apps:

- [Apache Maven ↗](#)
- [ARM templates](#)
- [Azure CLI](#)
- [Azure Developer CLI \(azd\) ↗](#)
- [Azure Functions Core Tools](#)
- [Azure Pipeline tasks ↗](#)
- [Azure portal ↗](#)
- [Bicep files ↗](#)
- [GitHub Actions ↗](#)
- [Visual Studio Code ↗](#)

Virtual network integration

When you host your function apps in a Container Apps environment, your functions are able to take advantage of both internally and externally accessible virtual networks. To learn more about environment networks, see [Networking in Azure Container Apps environment](#).

Configure scale rules

Azure Functions on Container Apps is designed to configure the scale parameters and rules as per the event target. You don't need to worry about configuring the KEDA scaled objects. You can still set minimum and maximum replica count when creating or modifying your function app. The following Azure CLI command sets the minimum and maximum replica count when creating a new function app in a Container Apps environment from an Azure Container Registry:

Azure CLI

```
az functionapp create --name <APP_NAME> --resource-group <MY_RESOURCE_GROUP>
--max-replicas 15 --min-replicas 1 --storage-account <STORAGE_NAME> --
environment MyContainerappEnvironment --image
```

```
<LOGIN_SERVER>/azurefunctionsimage:v1 --registry-username <USERNAME> --  
registry-password <SECURE_PASSWORD> --registry-server <LOGIN_SERVER>
```

The following command sets the same minimum and maximum replica count on an existing function app:

Azure CLI

```
az functionapp config container set --name <APP_NAME> --resource-group  
<MY_RESOURCE_GROUP> --max-replicas 15 --min-replicas 1
```

Managed resource groups

Azure Functions on Container Apps runs your containerized function app resources in specially managed resource groups. These managed resource groups help protect the consistency of your apps by preventing unintended or unauthorized modification or deletion of resources in the managed group, even by service principals.

A managed resource group is created for you the first time you create function app resources in a Container Apps environment. Container Apps resources required by your containerized function app run in this managed resource group. Any other function apps that you create in the same environment use this existing group.

A managed resource group gets removed automatically after all function app container resources are removed from the environment. While the managed resource group is visible, any attempts to modify or remove the managed resource group result in an error. To remove a managed resource group from an environment, remove all of the function app container resources and it gets removed for you.

If you run into any issues with these managed resource groups, you should contact support.

Considerations for Container Apps hosting

Keep in mind the following considerations when deploying your function app containers to Container Apps:

- While all triggers can be used, only the following triggers can dynamically scale (from zero instances) when running in a Container Apps environment:
 - HTTP
 - Azure Queue Storage
 - Azure Service Bus

- Azure Event Hubs
- Kafka
- Timer
- These limitations apply to Kafka triggers:
 - The protocol value of `ssl` isn't supported when hosted on Container Apps. Use a [different protocol value](#).
 - For a Kafka trigger to dynamically scale when connected to Event Hubs, the `username` property must resolve to an application setting that contains the actual username value. When the default `$ConnectionString` value is used, the Kafka trigger won't be able to cause the app to scale dynamically.
- For the built-in Container Apps [policy definitions](#), currently only environment-level policies apply to Azure Functions containers.
- You can use managed identities both for [trigger and binding connections](#) and for [deployments from an Azure Container Registry](#).
- When either your function app and Azure Container Registry-based deployment use managed identity-based connections, you can't modify the CPU and memory allocation settings in the portal. You must instead [use the Azure CLI](#).
- You currently can't move a Container Apps hosted function app deployment between resource groups or between subscriptions. Instead, you would have to recreate the existing containerized app deployment in a new resource group, subscription, or region.
- When using Container Apps, you don't have direct access to the lower-level Kubernetes APIs.
- The `containerapp` extension conflicts with the `appservice-kube` extension in Azure CLI. If you have previously published apps to Azure Arc, run `az extension list` and make sure that `appservice-kube` isn't installed. If it is, you can remove it by running `az extension remove -n appservice-kube`.

Next steps

- [Hosting and scale](#)
-

Feedback

Was this page helpful?

 Yes

 No

Deployment technologies in Azure Functions

Article • 04/02/2024

You can use a few different technologies to deploy your Azure Functions project code to Azure. This article provides an overview of the deployment methods available to you and recommendations for the best method to use in various scenarios. It also provides an exhaustive list of and key details about the underlying deployment technologies.

Deployment methods

The deployment technology you use to publish code to your function app in Azure depends on your specific needs and the point in the development cycle. For example, during development and testing you may deploy directly from your development tool, such as Visual Studio Code. When your app is in production, you're more likely to publish continuously from source control or by using an automated publishing pipeline, which can include validation and testing.

The following table describes the available deployment methods for your code project.

[+] Expand table

Deployment type	Methods	Best for...
Tools-based	<ul style="list-style-type: none">Visual Studio Code publishVisual Studio publishCore Tools publish	Deployments during development and other improvised deployments. Deploying your code on-demand using local development tools .
App Service-managed	<ul style="list-style-type: none">Deployment Center (CI/CD)Container deployments	Continuous deployment (CI/CD) from source control or from a container registry. Deployments are managed by the App Service platform (Kudu).
External pipelines	<ul style="list-style-type: none">Azure PipelinesGitHub Actions	Production pipelines that include validation, testing, and other actions that must be run as part of an automated deployment. Deployments are managed by the pipeline.

Specific deployments should use the best technology based on the specific scenario. Many of the deployment methods are based on [zip deployment](#), which is recommended for deployment.

Deployment technology availability

The deployment method also depends on the hosting plan and operating system on which you run your function app.

Currently, Functions offers three hosting plans:

- Consumption
- Premium
- Dedicated (App Service)

Each plan has different behaviors. Not all deployment technologies are available for each hosting plan and operating system. This chart provides information on the supported deployment technologies:

[\[+\] Expand table](#)

Deployment technology	Windows Consumption	Windows Premium	Windows Dedicated	Linux Consumption	Linux Premium	Linux Dedicated
External package URL ¹	✓	✓	✓	✓	✓	✓
Zip deploy	✓	✓	✓	✓	✓	✓
Docker container					✓	✓
Source control	✓	✓	✓		✓	✓
Local Git ¹	✓	✓	✓		✓	✓
FTPS ¹	✓	✓	✓		✓	✓
In-portal editing ²	✓	✓	✓	✓	✓	✓

¹ Deployment technologies that require you to [manually sync triggers](#) aren't recommended.

² In-portal editing is disabled when code is deployed to your function app from outside the portal. For more information, including language support details for in-portal editing, see [Language support details](#).

Key concepts

Some key concepts are critical to understanding how deployments work in Azure Functions.

Trigger syncing

When you change any of your triggers, the Functions infrastructure must be aware of the changes. Synchronization happens automatically for many deployment technologies. However, in some cases, you must manually sync your triggers.

You must manually sync triggers when using these deployment options:

- [External package URL](#)
- [Local Git](#)
- [FTPS](#)

You can sync triggers in one of three ways:

- Restart your function app in the Azure portal.
- Send an HTTP POST request to

```
https://<functionappname>.azurewebsites.net/admin/host/synctriggers?code=<API_KEY>
```

using the [master key](#).

- Send an HTTP POST request to

```
https://management.azure.com/subscriptions/<SUBSCRIPTION_ID>/resourceGroups/<RESOURCE_GROUP_NAME>/providers/Microsoft.Web/sites/<FUNCTION_APP_NAME>/syncfunctiontriggers?api-version=2016-08-01
```

. Replace the placeholders with your subscription ID, resource group name, and the name of your function app. This request requires an [access token](#) in the [Authorization request header](#).

When you deploy using an external package URL, you need to manually restart your function app to fully sync your deployment when the package changes without changing the URL, which includes initial deployment.

When your function app is secured by inbound network restrictions, the sync triggers endpoint can only be called from a client inside the virtual network.

Remote build

Azure Functions can automatically perform builds on the code it receives after zip deployments. These builds differ depending on whether your app is running on Windows or Linux.

Windows

All function apps running on Windows have a small management app, the `scm` site provided by [Kudu](#). This site handles much of the deployment and build logic for Azure Functions.

When an app is deployed to Windows, language-specific commands, like `dotnet restore` (C#) or `npm install` (JavaScript) are run.

The following considerations apply when using remote builds during deployment:

- Remote builds are supported for function apps running on Linux in the Consumption plan. However, deployment options are limited for these apps because they don't have an `scm` (Kudu) site.
- Function apps running on Linux a [Premium plan](#) or in a [Dedicated \(App Service\) plan](#) do have an `scm` (Kudu) site, but it's limited compared to Windows.
- Remote builds aren't performed when an app is using [run-from-package](#). To learn how to use remote build in these cases, see [Zip deploy](#).
- You may have issues with remote build when your app was created before the feature was made available (August 1, 2019). For older apps, either create a new function app or run `az functionapp update --resource-group <RESOURCE_GROUP_NAME> --name <APP_NAME>` to update your function app. This command might take two tries to succeed.

App content storage

Several deployment methods store the deployed or built application payload on the storage account associated with the function app. Functions tries to use the Azure Files content share when configured, but some methods instead store the payload in the blob storage instance associated with the `AzureWebJobsStorage` connection. See the details in the *Where app content is stored* paragraphs of each deployment technology covered in the next section.

Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

Deployment technology details

The following deployment methods are available in Azure Functions.

External package URL

You can use an external package URL to reference a remote package (.zip) file that contains your function app. The file is downloaded from the provided URL, and the app runs in [Run From Package](#) mode.

How to use it: Add [WEBSITE_RUN_FROM_PACKAGE](#) to your application settings. The value of this setting should be a URL (the location of the specific package file you want to run). You can add settings either [in the portal](#) or [by using the Azure CLI](#).

If you use Azure Blob storage, use a private container with a [shared access signature \(SAS\)](#) to give Functions access to the package. Any time the application restarts, it fetches a copy of the content. Your reference must be valid for the lifetime of the application.

When to use it: External package URL is the only supported deployment method for Azure Functions running on Linux in the Consumption plan, if the user doesn't want a [remote build](#) to occur. Whenever you deploy the package file that a function app references, you must [manually sync triggers](#), including the initial deployment. When you change the contents of the package file and not the URL itself, you must also restart your function app to sync triggers.

Where app content is stored: App content is stored at the URL specified. This could be on Azure Blobs, possibly in the storage account specified by the `AzureWebJobsStorage` connection. Some client tools may default to deploying to a blob in this account. For example, for Linux Consumption apps, the Azure CLI will attempt to deploy through a package stored in a blob on the account specified by `AzureWebJobsStorage`.

Zip deploy

Use zip deploy to push a .zip file that contains your function app to Azure. Optionally, you can set your app to start [running from package](#), or specify that a [remote build](#) occurs.

How to use it: Deploy by using your favorite client tool: [Visual Studio Code](#), [Visual Studio](#), or from the command line using the [Azure Functions Core Tools](#). By default, these tools use zip deployment and [run from package](#). Core Tools and the Visual

Studio Code extension both enable [remote build](#) when deploying to Linux. To manually deploy a .zip file to your function app, follow the instructions in [Deploy from a .zip file or URL](#).

When you deploy by using zip deploy, you can set your app to [run from package](#). To run from package, set the `WEBSITE_RUN_FROM_PACKAGE` application setting value to `1`. We recommend zip deployment. It yields faster loading times for your applications, and it's the default for VS Code, Visual Studio, and the Azure CLI.

When to use it: Zip deploy is the recommended deployment technology for Azure Functions.

Where app content is stored: App content from a zip deploy by default is stored on the file system, which may be backed by Azure Files from the storage account specified when the function app was created. In Linux Consumption, the app content instead is persisted on a blob in the storage account specified by the `AzureWebJobsStorage` connection.

Docker container

You can deploy a function app running in a Linux container.

How to use it: [Create your functions in a Linux container](#) then deploy the container to a Premium or Dedicated plan in Azure Functions or another container host. Use the [Azure Functions Core Tools](#) to create a customized Dockerfile for your project that you use to build a containerized function app. You can use the container in the following deployments:

- Deploy to Azure Functions resources you create in the Azure portal. For more information, see [Azure portal create using containers](#).
- Deploy to Azure Functions resources you create from the command line. Requires either a Premium or Dedicated (App Service) plan. To learn how, see [Create your first containerized Azure Functions](#).
- Deploy to Azure Container Apps (preview). To learn how, see [Create your first containerized Azure Functions on Azure Container Apps](#).
- Deploy to Azure Arc (preview). To learn how, see [Create your first containerized Azure Functions on Azure Arc \(preview\)](#).
- Deploy to a Kubernetes cluster. You can deploy to a cluster using [Azure Functions Core Tools](#). Use the `func kubernetes deploy` command.

When to use it: Use the Docker container option when you need more control over the Linux environment where your function app runs and where the container is hosted. This deployment mechanism is available only for functions running on Linux.

Where app content is stored: App content is stored in the specified container registry as a part of the image.

Source control

You can enable continuous integration between your function app and a source code repository. With source control enabled, an update to code in the connected source repository triggers deployment of the latest code from the repository. For more information, see the [Continuous deployment for Azure Functions](#).

How to use it: The easiest way to set up publishing from source control is from the Deployment Center in the Functions area of the portal. For more information, see [Continuous deployment for Azure Functions](#).

When to use it: Using source control is the best practice for teams that collaborate on their function apps. Source control is a good deployment option that enables more sophisticated deployment pipelines. Source control is usually enabled on a staging slot, which can be swapped into production after validation of updates from the repository. For more information, see [Azure Functions deployment slots](#).

Where app content is stored: The app content is in the source control system, but a locally cloned and built app content from is stored on the app file system, which may be backed by Azure Files from the storage account specified when the function app was created.

Local Git

You can use local Git to push code from your local machine to Azure Functions by using Git.

How to use it: Follow the instructions in [Local Git deployment to Azure App Service](#).

When to use it: To reduce the chance of errors, you should avoid using deployment methods that require the additional step of [manually syncing triggers](#). Use [zip deployment](#) when possible.

Where app content is stored: App content is stored on the file system, which may be backed by Azure Files from the storage account specified when the function app was created.

FTP/S

You can use FTP/S to directly transfer files to Azure Functions, although this deployment method isn't recommended. When you're not planning on using FTP, you should disable it. If you do choose to use FTP, you should enforce FTPS. To learn how in the Azure portal, see [Enforce FTPS](#).

How to use it: Follow the instructions in [FTPS deployment settings](#) to get the URL and credentials you can use to deploy to your function app using FTPS.

When to use it: To reduce the chance of errors, you should avoid using deployment methods that require the additional step of [manually syncing triggers](#). Use [zip deployment](#) when possible.

Where app content is stored: App content is stored on the file system, which may be backed by Azure Files from the storage account specified when the function app was created.

Portal editing

In the portal-based editor, you can directly edit the files that are in your function app (essentially deploying every time you save your changes).

How to use it: To be able to edit your functions in the [Azure portal](#), you must have [created your functions in the portal](#). To preserve a single source of truth, using any other deployment method makes your function read-only and prevents continued portal editing. To return to a state in which you can edit your files in the Azure portal, you can manually turn the edit mode back to `Read/Write` and remove any deployment-related application settings (like `WEBSITE_RUN_FROM_PACKAGE`).

When to use it: The portal is a good way to get started with Azure Functions. For more advanced development work, we recommend that you use one of the following client tools:

- [Visual Studio Code](#)
- [Azure Functions Core Tools \(command line\)](#)

- [Visual Studio](#)

Where app content is stored: App content is stored on the file system, which may be backed by Azure Files from the storage account specified when the function app was created.

The following table shows the operating systems and languages that support in-portal editing:

[\[+\] Expand table](#)

Language	Windows Consumption	Windows Premium	Windows Dedicated	Linux Consumption	Linux Premium	Linux Dedicated
C# ¹						
Java						
JavaScript (Node.js)	✓	✓	✓		✓	✓
Python ²				✓	✓	✓
PowerShell	✓	✓	✓			
TypeScript (Node.js)						

¹ In-portal editing is only supported for C# script files, which run in-process with the host. For more information, see the [Azure Functions C# script \(.csx\) developer reference](#).

² In-portal editing is only supported for the v1 Python programming model.

Deployment behaviors

When you deploy updates to your function app code, currently executing functions are terminated. After deployment completes, the new code is loaded to begin processing requests. Review [Improve the performance and reliability of Azure Functions](#) to learn how to write stateless and defensive functions.

If you need more control over this transition, you should use deployment slots.

Deployment slots

When you deploy your function app to Azure, you can deploy to a separate deployment slot instead of directly to production. Deploying to a deployment slot and then swapping into production after verification is the recommended way to configure [continuous deployment](#).

The way that you deploy to a slot depends on the specific deployment tool you use. For example, when using Azure Functions Core Tools, you include the `--slot` option to indicate the name of a specific slot for the `func azure functionapp publish` command.

For more information on deployment slots, see the [Azure Functions Deployment Slots](#) documentation for details.

Next steps

Read these articles to learn more about deploying your function apps:

- [Continuous deployment for Azure Functions](#)
- [Continuous delivery by using Azure Pipelines](#)
- [Zip deployments for Azure Functions](#)
- [Run your Azure Functions from a package file](#)
- [Automate resource deployment for your function app in Azure Functions](#)

Connect to eventing and messaging services from Azure Functions

Article • 11/11/2022

As a cloud computing service, Azure Functions is frequently used to move data between various Azure services. To make it easier for you to connect your code to other services, Functions implements a set of binding extensions to connect to these services. To learn more, see [Azure Functions triggers and bindings concepts](#).

By definition, Azure Functions executions are stateless. If you need to connect your code to services in a more stateful way, consider instead using [Durable Functions](#) or [Azure Logic Apps](#).

Triggers and bindings are provided to consuming and emitting data easier. There may be cases where you need more control over the service connection, or you just feel more comfortable using a client library provided by a service SDK. In those cases, you can use a client instance from the SDK in your function execution to access the service as you normally would. When using a client directly, you need to pay attention to the effect of scale and performance on client connections. To learn more, see the [guidance on using static clients](#).

You can't obtain the client instance used by a service binding from your function execution.

The rest of this article provides specific guidance for integrating your code with the specific Azure services supported by Functions.

Event Grid

Event Grid is an Azure service that sends HTTP requests to notify you about events that happen in publishers. A *publisher* is the service or resource that originates the event. For example, an Azure blob storage account is a publisher, and a [blob upload or deletion is an event](#). Some [Azure services have built-in support for publishing events to Event Grid](#).

Event *handlers* receive and process events. Azure Functions is one of several [Azure services that have built-in support for handling Event Grid events](#). Functions provides an Event Grid trigger, which invokes a function when an event is received from Event Grid. A similar output binding can be used to send events from your function to an [Event Grid custom topic](#).

You can also use an HTTP trigger to handle Event Grid Events. To learn more, see [Receive events to an HTTP endpoint](#). We recommend using the Event Grid trigger over HTTP trigger.

Azure Functions provides built-in integration with Azure Event Grid by using [triggers and bindings](#).

To learn how to configure and locally evaluate your Event Grid trigger and bindings, see [How to work with Event Grid triggers and bindings in Azure Functions](#)

For more information about Event Grid trigger and output binding definitions and examples, see one of the following reference articles:

- [Azure Event Grid bindings for Azure Functions](#)
- [Azure Event Grid trigger for Azure Functions](#)
- [Azure Event Grid output binding for Azure Functions](#)

Next steps

To learn more about Event Grid with Functions, see the following articles:

- [Azure Event Grid bindings for Azure Functions](#)
- [Tutorial: Automate resizing uploaded images using Event Grid](#)

Event-driven scaling in Azure Functions

Article • 08/01/2024

In the Consumption, Flex Consumption, and Premium plans, Azure Functions scales resources by adding more instances based on the number of events that trigger a function.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

The way in which your function app scales depends on the hosting plan:

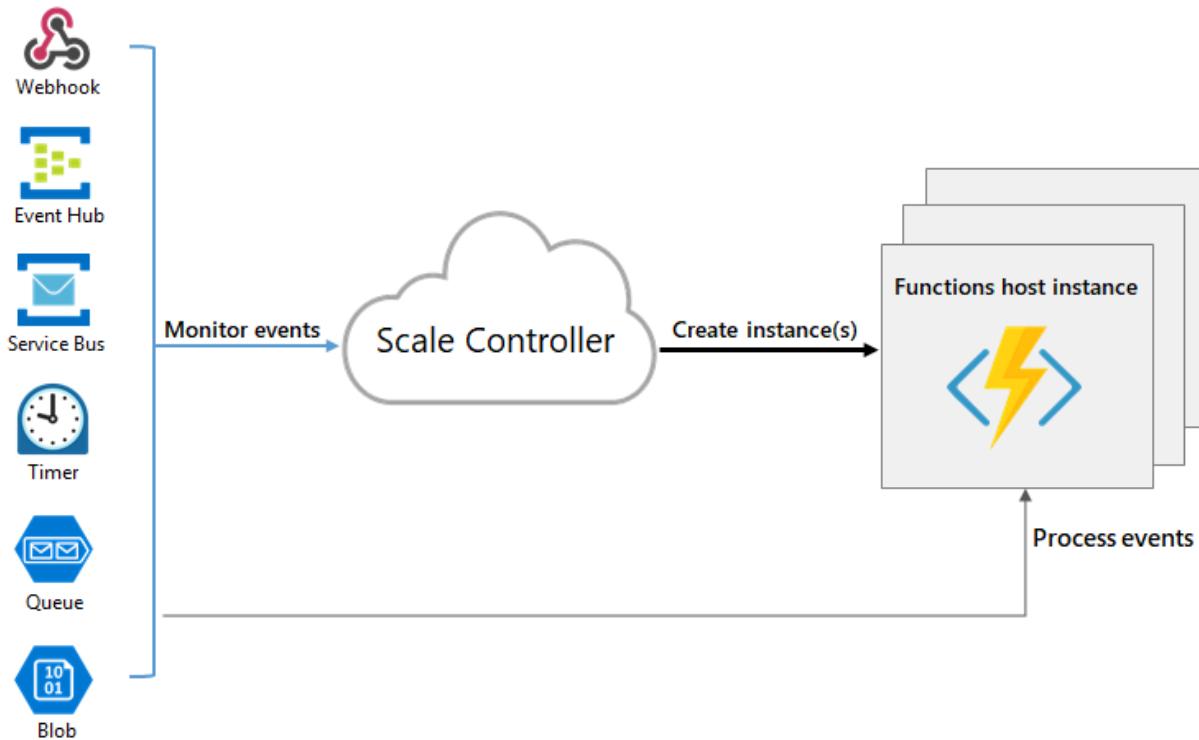
- **Consumption plan:** Each instance of the Functions host in the Consumption plan is limited, typically to 1.5 GB of memory and one CPU. An instance of the host supports the entire function app. As such, all functions within a function app share resource in an instance are scaled at the same time. When function apps share the same Consumption plan, they're still scaled independently.
- **Flex Consumption plan:** The plan uses a deterministic per-function scaling strategy, where each function is scaled independently, except for HTTP, Blob, and Durable Functions triggered functions which scale in their own groups. For more information, see [Per-function scaling](#). These instances are then scaled based on the concurrency of your requests.
- **Premium plan:** The specific size of the Premium plan determines the available memory and CPU for all apps in that plan on that instance. The plan scales out its instances based on the scaling needs of the apps in the plan, and the apps scale within the plan as needed.

Function code files are stored on Azure Files shares on the function's main storage account. When you delete the main storage account of the function app, the function code files are deleted and can't be recovered.

Runtime scaling

Azure Functions uses a component called the *scale controller* to monitor the rate of events and determine whether to scale out or scale in. The scale controller uses heuristics for each trigger type. For example, when you're using an Azure Queue storage trigger, it uses [target-based scaling](#).

The unit of scale for Azure Functions is the function app. When the function app is scaled out, more resources are allocated to run multiple instances of the Azure Functions host. Conversely, as compute demand is reduced, the scale controller removes function host instances. The number of instances is eventually "scaled in" when no functions are running within a function app.



Cold Start

Should your function app become idle for a few minutes, the platform might decide to scale the number of instances on which your app runs down to zero. The next request has the added latency of scaling from zero to one. This latency is referred to as a *cold start*. The number of dependencies required by your function app can affect the cold start time. Cold start is more of an issue for synchronous operations, such as HTTP triggers that must return a response. If cold starts are impacting your functions, consider using a plan other than the Consumption. The other plans offer these strategies to mitigate or eliminate cold starts:

- **Premium plan:** supports both prewarmed instances and always ready instances, with a minimum of one instance.
- **Flex Consumption plan:** supports an optional number of always ready instances, which can be defined on a per instance scaling basis.
- **Dedicated plan:** the plan itself doesn't scale dynamically, but you can run your app continuously with the **Always on** setting is enabled.

Understanding scaling behaviors

Scaling can vary based on several factors, and apps scale differently based on the triggers and language selected. There are a few intricacies of scaling behaviors to be aware of:

- **Maximum instances:** A single function app only scales out to a [maximum allowed by the plan](#). However, a single instance [can process more than one message or request at a time](#). You can [specify a lower maximum](#) to throttle scale as required.
- **New instance rate:** For HTTP triggers, new instances are allocated, at most, once per second. For non-HTTP triggers, new instances are allocated, at most, once every 30 seconds. Scaling is faster when running in a [Premium plan](#).
- **Target-based scaling:** Target-based scaling provides a fast and intuitive scaling model for customers and is currently supported for Service Bus queues and topics, Storage queues, Event Hubs, Apache Kafka, and Azure Cosmos DB extensions. Make sure to review [target-based scaling](#) to understand their scaling behavior.
- **Per-function scaling:** With some notable exceptions, functions running in the Flex Consumption plan scale on independent instances. The exceptions include HTTP triggers and Blob storage (Event Grid) triggers. Each of these trigger types scale together as a group on the same instances. Likewise, all Durable Functions triggers also share instances and scale together. For more information, see [per-function scaling](#).
- **Maximum monitored triggers:** Currently, the scale controller can only monitor up to 100 triggers to make scaling decisions. When your app has more than 100 event-based triggers, scale decisions are made based on only the first 100 triggers that execute. For more information, see [Best practices and patterns for scalable apps](#).

Limit scale-out

You might decide to restrict the maximum number of instances an app can use for scale-out. This is most common for cases where a downstream component like a database has limited throughput. For the maximum scale limits when running the various hosting plans, see [Scale limits](#).

Flex Consumption plan

By default, apps running in a Flex Consumption plan have a limit of `100` overall instances. Currently the lowest maximum instance count value is `40`, and the highest supported maximum instance count value is `1000`. When you use the `az functionapp create`

command to create a function app in the Flex Consumption plan, use the `--maximum-instance-count` parameter to set this maximum instance count for of your app.

Note that while you can change the maximum instance count of Flex Consumption apps up to 1000, your apps will reach a quota limit before reaching that number. Review [Regional subscription memory quotas](#) for more details.

This example creates an app with a maximum instance count of `200`:

Azure CLI

```
az functionapp create --resource-group <RESOURCE_GROUP> --name <APP_NAME> --storage <STORAGE_ACCOUNT_NAME> --runtime <LANGUAGE_RUNTIME> --runtime-version <RUNTIME_VERSION> --flexconsumption-location <REGION> --maximum-instance-count 200
```

This example uses the `az functionapp scale config set` command to change the maximum instance count for an existing app to `150`:

Azure CLI

```
az functionapp scale config set --resource-group <RESOURCE_GROUP> --name <APP_NAME> --maximum-instance-count 150
```

Consumption/Premium plans

In a Consumption or Elastic Premium plan, you can specify a lower maximum limit for your app by modifying the value of the `functionAppScaleLimit` site configuration setting. The `functionAppScaleLimit` can be set to `0` or `null` for unrestricted, or a valid value between `1` and the app maximum.

Azure CLI

Azure CLI

```
az resource update --resource-type Microsoft.Web/sites -g <RESOURCE_GROUP> -n <FUNCTION_APP-NAME>/config/web --set properties.functionAppScaleLimit=<SCALE_LIMIT>
```

Scale-in behaviors

Event-driven scaling automatically reduces capacity when demand for your functions is reduced. It does this by draining instances of their current function executions and then removes those instances. This behavior is logged as drain mode. The grace period for functions that are currently executing can extend up to 10 minutes for Consumption plan apps and up to 60 minutes for Premium plan apps. Event-driven scaling and this behavior don't apply to Dedicated plan apps.

The following considerations apply for scale-in behaviors:

- For app running on Windows in a Consumption plan, only apps created after May 2021 have drain mode behaviors enabled by default.
- To enable graceful shutdown for functions using the Service Bus trigger, use version 4.2.0 or a later version of the [Service Bus Extension](#).

Per-function scaling

Applies only to the Flex Consumption plan (preview).

The [Flex Consumption plan](#) is unique in that it implements a *per-function scaling* behavior. In per-function scaling, except for HTTP triggers, Blob (Event Grid) triggers, and Durable Functions, all other function trigger types in your app scale on independent instances. HTTP triggers in your app all scale together as a group on the same instances, as do all Blob (Event Grid), and all Durable Functions triggers, which have their own shared instances.

Consider a function app hosted a Flex Consumption plan that has these function:

[+] [Expand table](#)

function1	function2	function3	function4	function5	function6	function7
HTTP trigger	HTTP trigger	Orchestration trigger (Durable)	Activity trigger (Durable)	Service Bus trigger	Service Bus trigger	Event Hubs trigger

In this example:

- The two HTTP triggered functions (`function1` and `function2`) both run together on their own instances and scale together according to [HTTP concurrency settings](#).
- The two Durable functions (`function3` and `function4`) both run together on their own instances and scale together based on [configured concurrency throttles](#).
- The Service bus triggered function `function5` runs in its own and is scaled independently according to the [target-based scaling rules for Service Bus queues](#)

and topics.

- The Service bus triggered function `function6` runs in its own and is scaled independently according to the [target-based scaling rules for Service Bus queues and topics](#).
- The Event Hubs trigger (`function7`) runs in its own instances and is scaled independently according to the [target-based scaling rules for Event Hubs](#).

Best practices and patterns for scalable apps

There are many aspects of a function app that impacts how it scales, including host configuration, runtime footprint, and resource efficiency. For more information, see the [scalability section of the performance considerations article](#). You should also be aware of how connections behave as your function app scales. For more information, see [How to manage connections in Azure Functions](#).

If your app has more than 100 functions that use event-based triggers, consider breaking the app into one or more apps, where each app has less than 100 event-based functions.

For more information on scaling in Python and Node.js, see [Azure Functions Python developer guide - Scaling and concurrency](#) and [Azure Functions Node.js developer guide - Scaling and concurrency](#).

Next steps

To learn more, see the following articles:

- [Improve the performance and reliability of Azure Functions](#)
- [Azure Functions reliable event processing](#)
- [Azure Functions hosting options](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Target-based scaling

Article • 05/05/2024

Target-based scaling provides a fast and intuitive scaling model for customers and is currently supported for these binding extensions:

- [Apache Kafka](#)
- [Azure Cosmos DB](#)
- [Azure Event Hubs](#)
- [Azure Queue Storage](#)
- [Azure Service Bus \(queue and topics\)](#)

Target-based scaling replaces the previous Azure Functions incremental scaling model as the default for these extension types. Incremental scaling added or removed a maximum of one worker at [each new instance rate](#), with complex decisions for when to scale. In contrast, target-based scaling allows scale up of four instances at a time, and the scaling decision is based on a simple target-based equation:

$$\text{Desired instances} = \frac{\text{Event source length}}{\text{Target executions per instance}}$$

In this equation, *event source length* refers to the number of events that must be processed. The default *target executions per instance* values come from the SDKs used by the Azure Functions extensions. You don't need to make any changes for target-based scaling to work.

Considerations

The following considerations apply when using target-based scaling:

- Target-based scaling is enabled by default for function apps on the [Consumption plan](#), [Flex Consumption plan](#), and [Elastic Premium plans](#). Event-driven scaling isn't supported when running on [Dedicated \(App Service\) plans](#).
- Target-based scaling is enabled by default starting with version 4.19.0 of the Functions runtime.
- When you use target-based scaling, scale limits are still honored. For more information, see [Limit scale out](#).

- To achieve the most accurate scaling based on metrics, use only one target-based triggered function per function app. You should also consider running in a Flex Consumption plan, which offers [per-function scaling](#).
- When multiple functions in the same function app are all requesting to scale out at the same time, a sum across those functions is used to determine the change in desired instances. Functions requesting to scale out override functions requesting to scale in.
- When there are scale-in requests without any scale-out requests, the max scale in value is used.

Opting out

Target-based scaling is enabled by default for function apps hosted on a Consumption plan or on a Premium plans. To disable target-based scaling and fall back to incremental scaling, add the following app setting to your function app:

[\[+\] Expand table](#)

App Setting	Value
TARGET_BASED_SCALING_ENABLED	0

Customizing target-based scaling

You can make the scaling behavior more or less aggressive based on your app's workload by adjusting *target executions per instance*. Each extension has different settings that you can use to set *target executions per instance*.

This table summarizes the `host.json` values that are used for the *target executions per instance* values and the defaults:

[\[+\] Expand table](#)

Extension	host.json values	Default Value
Event Hubs (Extension v5.x+)	<code>extensions.eventHubs.maxEventBatchSize</code>	100*
Event Hubs (Extension v3.x+)	<code>extensions.eventHubs.eventProcessorOptions.maxBatchSize</code>	10

Extension	host.json values	Default Value
Event Hubs (if defined)	extensions.eventHubs.targetUnprocessedEventThreshold	n/a
Service Bus (Extension v5.x+, Single Dispatch)	extensions.serviceBus.maxConcurrentCalls	16
Service Bus (Extension v5.x+, Single Dispatch Sessions Based)	extensions.serviceBus.maxConcurrentSessions	8
Service Bus (Extension v5.x+, Batch Processing)	extensions.serviceBus.maxMessageBatchSize	1000
Service Bus (Functions v2.x+, Single Dispatch)	extensions.serviceBus.messageHandlerOptions.maxConcurrentCalls	16
Service Bus (Functions v2.x+, Single Dispatch Sessions Based)	extensions.serviceBus.sessionHandlerOptions.maxConcurrentSessions	2000
Service Bus (Functions v2.x+, Batch Processing)	extensions.serviceBus.batchOptions.maxMessageCount	1000
Storage Queue	extensions.queues.batchSize	16

* The default `maxEventBatchSize` changed in [v6.0.0](#) of the `Microsoft.Azure.WebJobs.Extensions.EventHubs` package. In earlier versions, this value was 10.

For some binding extensions, *target executions per instance* is set using a function attribute:

[\[+\] Expand table](#)

Extension	Function trigger setting	Default Value
Apache Kafka	<code>lagThreshold</code>	1000
Azure Cosmos DB	<code>maxItemsPerInvocation</code>	100

To learn more, see the [example configurations for the supported extensions](#).

Premium plan with runtime scale monitoring enabled

When [runtime scale monitoring](#) is enabled, the extensions themselves handle dynamic scaling. This is because the [scale controller](#) doesn't have access to services secured by a virtual network. After you enable runtime scale monitoring, you'll need to upgrade your extension packages to these minimum versions to unlock the extra target-based scaling functionality:

[\[+\] Expand table](#)

Extension Name	Minimum Version Needed
Apache Kafka	3.9.0
Azure Cosmos DB	4.1.0
Event Hubs	5.2.0
Service Bus	5.9.0
Storage Queue	5.1.0

Dynamic concurrency support

Target-based scaling introduces faster scaling, and uses defaults for *target executions per instance*. When using Service Bus, Storage queues, or Kafka, you can also enable [dynamic concurrency](#). In this configuration, the *target executions per instance* value is determined automatically by the dynamic concurrency feature. It starts with limited concurrency and identifies the best setting over time.

Supported extensions

The way in which you configure target-based scaling in your host.json file depends on the specific extension type. This section provides the configuration details for the extensions that currently support target-based scaling.

Service Bus queues and topics

The Service Bus extension support three execution models, determined by the `IsBatched` and `IsSessionsEnabled` attributes of your Service Bus trigger. The default value for `IsBatched` and `IsSessionsEnabled` is `false`.

[+] Expand table

Execution Model	<code>IsBatched</code>	<code>IsSessionsEnabled</code>	Setting Used for <i>target executions per instance</i>
Single dispatch processing	false	false	<code>maxConcurrentCalls</code>
Single dispatch processing (session-based)	false	true	<code>maxConcurrentSessions</code>
Batch processing	true	false	<code>maxMessageBatchSize</code> or <code>maxMessageCount</code>

ⓘ Note

Scale efficiency: For the Service Bus extension, use *Manage* rights on resources for the most efficient scaling. With *Listen* rights scaling reverts to incremental scale because the queue or topic length can't be used to inform scaling decisions. To learn more about setting rights in Service Bus access policies, see [Shared Access Authorization Policy](#).

Single dispatch processing

In this model, each invocation of your function processes a single message. The `maxConcurrentCalls` setting governs *target executions per instance*. The specific setting depends on the version of the Service Bus extension.

v5.x+

Modify the `host.json` setting `maxConcurrentCalls`, as in the following example:

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "serviceBus": {  
            "maxConcurrentCalls": 16  
        }  
    }  
}
```

Single dispatch processing (session-based)

In this model, each invocation of your function processes a single message. However, depending on the number of active sessions for your Service Bus topic or queue, each instance leases one or more sessions. The specific setting depends on the version of the Service Bus extension.

v5.x+

Modify the `host.json` setting `maxConcurrentSessions` to set *target executions per instance*, as in the following example:

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "serviceBus": {  
            "maxConcurrentSessions": 8  
        }  
    }  
}
```

Batch processing

In this model, each invocation of your function processes a batch of messages. The specific setting depends on the version of the Service Bus extension.

v5.x+

Modify the `host.json` setting `maxMessageBatchSize` to set *target executions per instance*, as in the following example:

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "serviceBus": {  
            "maxMessageBatchSize": 1000  
        }  
    }  
}
```

Event Hubs

For Azure Event Hubs, Azure Functions scales based on the number of unprocessed events distributed across all the partitions in the event hub. By default, the `host.json` attributes used for *target executions per instance* are `maxEventBatchSize` and `maxBatchSize`. However, if you choose to fine-tune target-based scaling, you can define a separate parameter `targetUnprocessedEventThreshold` that overrides to set *target executions per instance* without changing the batch settings. If `targetUnprocessedEventThreshold` is set, the total unprocessed event count is divided by this value to determine the number of instances, which is then be rounded up to a worker instance count that creates a balanced partition distribution.

ⓘ Note

Since Event Hubs is a partitioned workload, the target instance count for Event Hubs is capped by the number of partitions in your event hub.

The specific setting depends on the version of the Event Hubs extension.

v5.x+

Modify the `host.json` setting `maxEventBatchSize` to set *target executions per instance*, as in the following example:

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "eventHubs": {  
            "maxEventBatchSize" : 100  
        }  
    }  
}
```

```
    }  
}
```

When defined in `host.json`, `targetUnprocessedEventThreshold` is used as *target executions per instance* instead of `maxEventBatchSize`, as in the following example:

JSON

```
{  
  "version": "2.0",  
  "extensions": {  
    "eventHubs": {  
      "targetUnprocessedEventThreshold": 153  
    }  
  }  
}
```

Storage Queues

For v2.x+ of the Storage extension, modify the `host.json` setting `batchSize` to set *target executions per instance*:

JSON

```
{  
  "version": "2.0",  
  "extensions": {  
    "queues": {  
      "batchSize": 16  
    }  
  }  
}
```

ⓘ Note

Scale efficiency: For the storage queue extension, messages with `visibilityTimeout` are still counted in *event source length* by the Storage Queue APIs. This can cause overscaling of your function app. Consider using Service Bus queues que scheduled messages, `limiting scale out`, or not using `visibilityTimeout` for your solution.

Azure Cosmos DB

Azure Cosmos DB uses a function-level attribute, `MaxItemsPerInvocation`. The way you set this function-level attribute depends on your function language.

C#

For a compiled C# function, set `MaxItemsPerInvocation` in your trigger definition, as shown in the following examples for an in-process C# function:

C#

```
namespace CosmosDBSamplesV2
{
    public static class CosmosTrigger
    {
        [FunctionName("CosmosTrigger")]
        public static void Run([CosmosDBTrigger(
            databaseName: "ToDoItems",
            collectionName: "Items",
            MaxItemsPerInvocation: 100,
            ConnectionStringSetting = "CosmosDBConnection",
            LeaseCollectionName = "leases",
            CreateLeaseCollectionIfNotExists =
true)]IReadOnlyList<Document> documents,
            ILogger log)
        {
            if (documents != null && documents.Count > 0)
            {
                log.LogInformation($"Documents modified:
{documents.Count}");
                log.LogInformation($"First document Id:
{documents[0].Id}");
            }
        }
    }
}
```

ⓘ Note

Since Azure Cosmos DB is a partitioned workload, the target instance count for the database is capped by the number of physical partitions in your container. To learn more about Azure Cosmos DB scaling, see [physical partitions](#) and [lease ownership](#).

Apache Kafka

The Apache Kafka extension uses a function-level attribute, `LagThreshold`. For Kafka, the number of *desired instances* is calculated based on the total consumer lag divided by the `LagThreshold` setting. For a given lag, reducing the lag threshold increases the number of desired instances.

The way you set this function-level attribute depends on your function language. This example sets the threshold to `100`.

C#

For a compiled C# function, set `LagThreshold` in your trigger definition, as shown in the following examples for an in-process C# function for a Kafka Event Hubs trigger:

```
C#  
  
[FunctionName("KafkaTrigger")]
public static void Run(
    [KafkaTrigger("BrokerList",
        "topic",
        Username = "$ConnectionString",
        Password = "%EventHubConnectionString%",
        Protocol = BrokerProtocol.SaslSsl,
        AuthenticationMode = BrokerAuthenticationMode.Plain,
        ConsumerGroup = "$Default",
        LagThreshold = 100)] KafkaEventData<string> kevent,
    ILogger log)
{
    log.LogInformation($"C# Kafka trigger function processed a message:
{kevent.Value}");
}
```

Next steps

To learn more, see the following articles:

- [Improve the performance and reliability of Azure Functions](#)
- [Azure Functions reliable event processing](#)

Feedback

Was this page helpful?

 Yes

 No

Azure Functions reliable event processing

Article • 06/28/2022

Event processing is one of the most common scenarios associated with serverless architecture. This article describes how to create a reliable message processor with Azure Functions to avoid losing messages.

Challenges of event streams in distributed systems

Consider a system that sends events at a constant rate of 100 events per second. At this rate, within minutes multiple parallel Functions instances can consume the incoming 100 events every second.

However, any of the following less-optimal conditions are possible:

- What if the event publisher sends a corrupt event?
- What if your Functions instance encounters unhandled exceptions?
- What if a downstream system goes offline?

How do you handle these situations while preserving the throughput of your application?

With queues, reliable messaging comes naturally. When paired with a Functions trigger, the function creates a lock on the queue message. If processing fails, the lock is released to allow another instance to retry processing. Processing then continues until either the message is evaluated successfully, or it is added to a poison queue.

Even while a single queue message may remain in a retry cycle, other parallel executions continue to keep to dequeuing remaining messages. The result is that the overall throughput remains largely unaffected by one bad message. However, storage queues don't guarantee ordering and aren't optimized for the high throughput demands required by Event Hubs.

By contrast, Azure Event Hubs doesn't include a locking concept. To allow for features like high-throughput, multiple consumer groups, and replay-ability, Event Hubs events behave more like a video player. Events are read from a single point in the stream per partition. From the pointer you can read forwards or backwards from that location, but you have to choose to move the pointer for events to process.

When errors occur in a stream, if you decide to keep the pointer in the same spot, event processing is blocked until the pointer is advanced. In other words, if the pointer is stopped to deal with problems processing a single event, the unprocessed events begin piling up.

Azure Functions avoids deadlocks by advancing the stream's pointer regardless of success or failure. Since the pointer keeps advancing, your functions need to deal with failures appropriately.

How Azure Functions consumes Event Hubs events

Azure Functions consumes Event Hub events while cycling through the following steps:

1. A pointer is created and persisted in Azure Storage for each partition of the event hub.
2. When new messages are received (in a batch by default), the host attempts to trigger the function with the batch of messages.
3. If the function completes execution (with or without exception) the pointer advances and a checkpoint is saved to the storage account.
4. If conditions prevent the function execution from completing, the host fails to progress the pointer. If the pointer isn't advanced, then later checks end up processing the same messages.
5. Repeat steps 2–4

This behavior reveals a few important points:

- *Unhandled exceptions may cause you to lose messages.* Executions that result in an exception will continue to progress the pointer. Setting a [retry policy](#) will delay progressing the pointer until the entire retry policy has been evaluated.
- *Functions guarantees at-least-once delivery.* Your code and dependent systems may need to [account for the fact that the same message could be received twice](#).

Handling exceptions

As a general rule, every function should include a [try/catch block](#) at the highest level of code. Specifically, all functions that consume Event Hubs events should have a [catch](#) block. That way, when an exception is raised, the catch block handles the error before the pointer progresses.

Retry mechanisms and policies

Some exceptions are transient in nature and don't reappear when an operation is attempted again moments later. This is why the first step is always to retry the operation. You can leverage the function app [retry policies](#) or author retry logic within the function execution.

Introducing fault-handling behaviors to your functions allow you to define both basic and advanced retry policies. For instance, you could implement a policy that follows a workflow illustrated by the following rules:

- Try to insert a message three times (potentially with a delay between retries).
- If the eventual outcome of all retries is a failure, then add a message to a queue so processing can continue on the stream.
- Corrupt or unprocessed messages are then handled later.

ⓘ Note

[Polly](#) is an example of a resilience and transient-fault-handling library for C# applications.

Non-exception errors

Some issues arise even when an error is not present. For example, consider a failure that occurs in the middle of an execution. In this case, if a function doesn't complete execution, the offset pointer is never progressed. If the pointer doesn't advance, then any instance that runs after a failed execution continues to read the same messages. This situation provides an "at-least-once" guarantee.

The assurance that every message is processed at least one time implies that some messages may be processed more than once. Your function apps need to be aware of this possibility and must be built around the [principles of idempotency](#).

Stop and restart execution

While a few errors may be acceptable, what if your app experiences significant failures? You may want to stop triggering on events until the system reaches a healthy state. Having the opportunity to pause processing is often achieved with a circuit breaker pattern. The circuit breaker pattern allows your app to "break the circuit" of the event process and resume at a later time.

There are two pieces required to implement a circuit breaker in an event process:

- Shared state across all instances to track and monitor health of the circuit
- Master process that can manage the circuit state (open or closed)

Implementation details may vary, but to share state among instances you need a storage mechanism. You may choose to store state in Azure Storage, a Redis cache, or any other account that is accessible by a collection of functions.

[Azure Logic Apps](#) or [durable functions](#) are a natural fit to manage the workflow and circuit state. Other services may work just as well, but logic apps are used for this example. Using logic apps, you can pause and restart a function's execution giving you the control required to implement the circuit breaker pattern.

Define a failure threshold across instances

To account for multiple instances processing events simultaneously, persisting shared external state is needed to monitor the health of the circuit.

A rule you may choose to implement might enforce that:

- If there are more than 100 eventual failures within 30 seconds across all instances, then break the circuit and stop triggering on new messages.

The implementation details will vary given your needs, but in general you can create a system that:

1. Log failures to a storage account (Azure Storage, Redis, etc.)
2. When new failure is logged, inspect the rolling count to see if the threshold is met (for example, more than 100 in last 30 seconds).
3. If the threshold is met, emit an event to Azure Event Grid telling the system to break the circuit.

Managing circuit state with Azure Logic Apps

The following description highlights one way you could create an Azure Logic App to halt a Functions app from processing.

Azure Logic Apps comes with built-in connectors to different services, features stateful orchestrations, and is a natural choice to manage circuit state. After detecting the circuit needs to break, you can build a logic app to implement the following workflow:

1. Trigger an Event Grid workflow and stop the Azure Function (with the Azure Resource connector)

2. Send a notification email that includes an option to restart the workflow

The email recipient can investigate the health of the circuit and, when appropriate, restart the circuit via a link in the notification email. As the workflow restarts the function, messages are processed from the last Event Hub checkpoint.

Using this approach, no messages are lost, all messages are processed in order, and you can break the circuit as long as necessary.

Resources

- [Reliable event processing samples ↗](#)
- [Azure Durable Entity Circuit Breaker ↗](#)

Next steps

For more information, see the following resources:

- [Azure Functions error handling](#)
- [Automate resizing uploaded images using Event Grid](#)
- [Create a function that integrates with Azure Logic Apps](#)

Concurrency in Azure Functions

Article • 05/21/2024

This article describes the concurrency behaviors of event-driven triggers in Azure Functions. It also compares the static and dynamic concurrency models.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

In Functions, you can have multiple executing processes of a given function running concurrently on a single compute instance. For example, consider a case where you have three different functions in your function app that is scaled-out to multiple instances to handle an increased load. In this scenario, each function is executing in response to individual invocations across all three instances, and a given instance can handle multiple invocations of the same type. Keep in mind that the function executions on a single instance share the same memory, CPU, and connection resources. Because multiple function executions can run on each instance concurrently, each function needs to have a way to manage the number of concurrent executions.

When your app is hosted in a dynamic scale plan (Consumption, Flex Consumption, or Premium), the host scales the number of function app instances up or down based on the number of incoming events. To learn more, see [Event Driven Scaling](#). When you host your functions in a Dedicated (App Service) plan, you must manually configure your instances or [set up an autoscale scheme](#).

These scale decisions also are directly impacted by the concurrency of executions on a given instance. When an app in a dynamic scale plan hits a concurrency limit, it might need to scale to keep up with incoming demand.

Functions provides two main ways of managing concurrency:

- **Static concurrency:** You can configure host-level limits on concurrency, which are specific to individual triggers. This is the default concurrency behavior for Functions.
- **Dynamic concurrency:** For certain trigger types, the Functions host can automatically determine the best level of concurrency for that trigger in your app. You must [opt in to this concurrency model](#).

Static concurrency

By default, most triggers support a host-level static configuration model. In this model, each trigger type has a per-instance concurrency limit. However, for most triggers you can also request a specific per-instance concurrency for that trigger type. For example, the [Service Bus trigger](#) provides both a `MaxConcurrentCalls` and a `MaxConcurrentSessions` setting in the [host.json file](#). These settings together control the maximum number of messages each function processes concurrently on each instance. Other trigger types have built-in mechanisms for load-balancing invocations across instances. For example, Event Hubs and Azure Cosmos DB both use a partition-based scheme.

For trigger types that support concurrency configuration, the settings you choose are applied to all running instances. This allows you to control the maximum concurrency for your functions on each instance. For example, when your function is CPU or resource-intensive, you may choose to limit concurrency to keep instances healthy and rely on scaling to handle increased loads. Similarly, when your function is making requests to a downstream service that is being throttled, you should also consider limiting concurrency to avoid overloading the downstream service.

HTTP trigger concurrency

Applies only to the Flex Consumption plan (preview)

The Flex Consumption plan scales all HTTP trigger functions together as a group. For more information, see [Per-function scaling](#). The following table indicates the default concurrency setting for HTTP triggers on a given instance, based on the configured [instance memory size](#).

[] [Expand table](#)

Instance size (MB)	Default concurrency*
2048	16
4096	32

*For Python apps, the default HTTP trigger concurrency for all instances sizes is 1.

These defaults should work well for most cases, and you start with them. Consider that at a given number of HTTP requests, increasing the HTTP concurrency value reduces the number of instances required to handle HTTP requests. Likewise decreasing the HTTP concurrency value requires more instances to handle the same load.

If you need to fine tune the HTTP concurrency, you can do this by using the Azure CLI. For more information, see [Set HTTP concurrency limits](#).

The default concurrency values in the previous table only apply when you haven't set your own HTTP concurrency setting. When you haven't explicitly set an HTTP concurrency setting, the default concurrency increases as shown in the table when you change the instance size. After you specifically set an HTTP concurrency value, that value is maintained despite changes in the instance size.

Determine optimal static concurrency

While static concurrency configurations give you control of certain trigger behaviors, such as throttling your functions, it can be difficult to determine the optimal values for these settings. Generally, you have to arrive at acceptable values by an iterative process of load testing. Even after you determine a set of values that are working for a particular load profile, the number of events arriving from your connected services may change from day to day. This variability means your app often may run with suboptimal values. For example, your function app may process particularly demanding message payloads on the last day of the week, which requires you to throttle concurrency down. However, during the rest of the week the message payloads are simpler, which means you could use a higher concurrency level the rest of the week.

Ideally, we want the system to allow instances to process as much work as they can while keeping each instance healthy and latencies low, which is what dynamic concurrency is designed to do.

Dynamic concurrency

Functions now provides a dynamic concurrency model that simplifies configuring concurrency for all function apps running in the same plan.

Note

Dynamic concurrency is currently only supported for the Azure Blob, Azure Queue, and Service Bus triggers and requires you to use the versions listed in the [extension support section below](#).

Benefits

Using dynamic concurrency provides the following benefits:

- **Simplified configuration:** You no longer have to manually determine per-trigger concurrency settings. The system learns the optimal values for your workload over time.
- **Dynamic adjustments:** Concurrency is adjusted up or down dynamically in real time, which allows the system to adapt to changing load patterns over time.
- **Instance health protection:** The runtime limits concurrency to levels a function app instance can comfortably handle. This protects the app from overloading itself by taking on more work than it should.
- **Improved throughput:** Overall throughput is improved because individual instances aren't pulling more work than they can quickly process. This allows work to be load-balanced more effectively across instances. For functions that can handle higher loads, higher throughput can be obtained by increasing concurrency to values above the default configuration.

Dynamic concurrency configuration

Dynamic concurrency can be enabled at the host level in the host.json file. When enabled, the concurrency levels of any binding extensions that support this feature are adjusted automatically as needed. In these cases, dynamic concurrency settings override any manually configured concurrency settings.

By default, dynamic concurrency is disabled. With dynamic concurrency enabled, concurrency starts at 1 for each function, and is adjusted up to an optimal value, which is determined by the host.

You can enable dynamic concurrency in your function app by adding the following settings in your host.json file:

```
JSON

{
  "version": "2.0",
  "concurrency": {
    "dynamicConcurrencyEnabled": true,
    "snapshotPersistenceEnabled": true
  }
}
```

When `SnapshotPersistenceEnabled` is `true`, which is the default, the learned concurrency values are periodically persisted to storage so new instances start from those values instead of starting from 1 and having to redo the learning.

Concurrency manager

Behind the scenes, when dynamic concurrency is enabled there's a concurrency manager process running in the background. This manager constantly monitors instance health metrics, like CPU and thread utilization, and changes throttles as needed. When one or more throttles are enabled, function concurrency is adjusted down until the host is healthy again. When throttles are disabled, concurrency is allowed to increase. Various heuristics are used to intelligently adjust concurrency up or down as needed based on these throttles. Over time, concurrency for each function stabilizes to a particular level.

Concurrency levels are managed for each individual function. As such, the system balances between resource-intensive functions that require a low level of concurrency and more lightweight functions that can handle higher concurrency. The balance of concurrency for each function helps to maintain overall health of the function app instance.

When dynamic concurrency is enabled, you'll see dynamic concurrency decisions in your logs. For example, you'll see logs when various throttles are enabled, and whenever concurrency is adjusted up or down for each function. These logs are written under the **Host.Concurrency** log category in the traces table.

Extension support

Dynamic concurrency is enabled for a function app at the host level, and any extensions that support dynamic concurrency run in that mode. Dynamic concurrency requires collaboration between the host and individual trigger extensions. Only the listed versions of the following extensions support dynamic concurrency.

[] [Expand table](#)

Extension	Version	Description
Queue storage	version 5.x	The Azure Queue storage trigger has its own message polling loop. When using static config, concurrency is governed by the <code>BatchSize</code> / <code>NewBatchThreshold</code> config options. When using dynamic concurrency, those configuration values are ignored. Dynamic concurrency is integrated into the message loop, so the number of messages fetched per iteration are dynamically adjusted. When throttles are enabled (host is overloaded), message processing will be paused until throttles are disabled. When throttles are disabled, concurrency will increase.
Blob storage	version 5.x	Internally, the Azure Blob storage trigger uses the same infrastructure that the Azure Queue Trigger uses. When new/updated blobs need to be processed, messages are written to a platform managed control queue, and that queue is processed using the same logic used for

Extension	Version	Description
Service Bus	version 5.x	<p>QueueTrigger. When dynamic concurrency is enabled, concurrency for the processing of that control queue will be dynamically managed.</p> <p>The Service Bus trigger currently supports three execution models. Dynamic concurrency affects these execution models as follows:</p> <ul style="list-style-type: none"> • Single dispatch topic/queue processing: Each invocation of your function processes a single message. When using static config, concurrency is governed by the <code>MaxConcurrentCalls</code> config option. When using dynamic concurrency, that config value is ignored, and concurrency is adjusted dynamically. • Session based single dispatch topic/queue processing: Each invocation of your function processes a single message. Depending on the number of active sessions for your topic/queue, each instance leases one or more sessions. Messages in each session are processed serially, to guarantee ordering in a session. When dynamic concurrency isn't used, concurrency is governed by the <code>MaxConcurrentSessions</code> setting. With dynamic concurrency enabled, <code>MaxConcurrentSessions</code> is ignored and the number of sessions each instance is processing is dynamically adjusted. • Batch processing: Each invocation of your function processes a batch of messages, governed by the <code>MaxMessageCount</code> setting. Because batch invocations are serial, concurrency for your batch-triggered function is always one and dynamic concurrency doesn't apply.

Next steps

For more information, see the following resources:

- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)

Designing Azure Functions for identical input

Article • 06/28/2022

The reality of event-driven and message-based architecture dictates the need to accept identical requests while preserving data integrity and system stability.

To illustrate, consider an elevator call button. As you press the button, it lights up and an elevator is sent to your floor. A few moments later, someone else joins you in the lobby. This person smiles at you and presses the illuminated button a second time. You smile back and chuckle to yourself as you're reminded that the command to call an elevator is idempotent.

Pressing an elevator call button a second, third, or fourth time has no bearing on the final result. When you press the button, regardless of the number of times, the elevator is sent to your floor. Idempotent systems, like the elevator, result in the same outcome no matter how many times identical commands are issued.

When it comes to building applications, consider the following scenarios:

- What happens if your inventory control application tries to delete the same product more than once?
- How does your human resource application behave if there is more than one request to create an employee record for the same person?
- Where does the money go if your banking app gets 100 requests to make the same withdrawal?

There are many contexts where requests to a function may receive identical commands. Some situations include:

- Retry policies sending the same request many times.
- Cached commands replayed to the application.
- Application errors sending multiple identical requests.

To protect data integrity and system health, an idempotent application contains logic that may contain the following behaviors:

- Verifying of the existence of data before trying to execute a delete.
- Checking to see if data already exists before trying to execute a create action.
- Reconciling logic that creates eventual consistency in data.
- Concurrency controls.
- Duplication detection.

- Data freshness validation.
- Guard logic to verify input data.

Ultimately idempotency is achieved by ensuring a given action is possible and is only executed once.

Next steps

- [Azure Functions reliable event processing](#)
- [Concurrency in Azure Functions](#)
- [Azure Functions error handling and retries](#)

Azure Functions triggers and bindings concepts

Article • 09/10/2024

In this article, you learn the high-level concepts surrounding functions triggers and bindings.

Triggers cause a function to run. A trigger defines how a function is invoked and a function must have exactly one trigger. Triggers can also pass data into your function, as you would with method calls.

Binding to a function is a way of declaratively connecting your functions to other resources; bindings either pass data into your function (an *input binding*) or enable you to write data out from your function (an *output binding*) using *binding parameters*. Your function trigger is essentially a special type of input binding.

You can mix and match different bindings to suit your function's specific scenario. Bindings are optional and a function might have one or multiple input and/or output bindings.

Triggers and bindings let you avoid hardcoding access to other services. Your function receives data (for example, the content of a queue message) in function parameters. You send data (for example, to create a queue message) by using the return value of the function.

Consider the following examples of how you could implement different functions.

[] Expand table

Example scenario	Trigger	Input binding	Output binding
A new queue message arrives which runs a function to write to another queue.	Queue*	None	Queue*
A scheduled job reads Blob Storage contents and creates a new Azure Cosmos DB document.	Timer	Blob Storage	Azure Cosmos DB
The Event Grid is used to read an image from Blob Storage and a document from Azure Cosmos DB to send an email.	Event Grid	Blob Storage and Azure Cosmos DB	SendGrid

* Represents different queues

These examples aren't meant to be exhaustive, but are provided to illustrate how you can use triggers and bindings together. For a more comprehensive set of scenarios, see [Azure Functions scenarios](#).

Tip

Functions doesn't require you to use input and output bindings to connect to Azure services. You can always create an Azure SDK client in your code and use it instead for your data transfers. For more information, see [Connect to services](#).

Trigger and binding definitions

Triggers and bindings are defined differently depending on the development language. Make sure to select your language at the [top](#) of the article.

Bindings can be either input or output bindings. Not all services support both input and output bindings. See your specific binding extension for [specific bindings code examples](#).

This example shows an HTTP triggered function with an output binding that writes a message to an Azure Storage queue.

For C# class library functions, triggers and bindings are configured by decorating methods and parameters with C# attributes, where the specific attribute applied might depend on the C# runtime model:

Isolated worker model

The HTTP trigger (`HttpTrigger`) is defined on the `Run` method for a function named `HttpExample` that returns a `MultiResponse` object:

```
C#  
  
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
{
```

This example shows the `MultiResponse` object definition which both returns an `HttpResponse` to the HTTP request and also writes a message to a storage queue

using a `QueueOutput` binding:

C#

```
public class MultiResponse
{
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

For more information, see the [C# isolated worker model guide](#).

Legacy C# Script functions use a `function.json` definition file. For more information, see the [Azure Functions C# script \(.csx\) developer reference](#).

This example is an HTTP triggered function that creates a queue item for each HTTP request received.

Add bindings to a function

You can connect your function to other services by using input or output bindings. Add a binding by adding its specific definitions to your function. To learn how, see [Add bindings to an existing function in Azure Functions](#).

Supported bindings

This table shows the bindings that are supported in the major versions of the Azure Functions runtime:

[] [Expand table](#)

Type	1.x ¹	2.x and higher ²	Trigger	Input	Output
Blob storage	✓	✓	✓	✓	✓
Azure Cosmos DB	✓	✓	✓	✓	✓
Azure Data Explorer		✓		✓	✓
Azure SQL		✓	✓	✓	✓
Dapr ⁴		✓	✓	✓	✓
Event Grid	✓	✓	✓		✓

Type	1.x ¹	2.x and higher ²	Trigger	Input	Output
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
IoT Hub	✓	✓	✓		
Kafka ³		✓	✓		✓
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
Redis		✓	✓		
RabbitMQ ³		✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓	✓	✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

Notes:

- Support will end for version 1.x of the Azure Functions runtime on September 14, 2026 [↗](#). We highly recommend that you [migrate your apps to version 4.x](#) for full support.
- Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).
- Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).
- Supported in Kubernetes, IoT Edge, and other self-hosted modes only.

For information about which bindings are in preview or are approved for production use, see [Supported languages](#).

Specific binding extension versions are only supported while the underlying service SDK is supported. Changes to support in the underlying service SDK version affect the support for the consuming extension.

Bindings code examples

Use the following table to find more examples of specific binding types that show you how to work with bindings in your functions. First, choose the language tab that corresponds to your project.

Binding code for C# depends on the [specific process model](#).

Isolated process

[Expand table](#)

Service	Examples	Samples
Blob storage	Trigger Input Output	Link
Azure Cosmos DB	Trigger Input Output	Link
Azure Data Explorer	Input Output	Link
Azure SQL	Trigger Input Output	Link
Event Grid	Trigger Output	Link
Event Hubs	Trigger Output	
IoT Hub	Trigger Output	
HTTP	Trigger	Link
Queue storage	Trigger Output	Link
RabbitMQ	Trigger Output	
SendGrid	Output	

Service	Examples	Samples
Service Bus	Trigger Output	Link ↗
SignalR	Trigger Input Output	
Table storage	Input Output	
Timer	Trigger	Link ↗
Twilio	Output	Link ↗

Custom bindings

You can create custom input and output bindings. Bindings must be authored in .NET, but can be consumed from any supported language. For more information about creating custom bindings, see [Creating custom input and output bindings ↗](#).

Related content

- [Binding expressions and patterns](#)
- [How to register a binding expression](#)
- Testing:
 - [Strategies for testing your code in Azure Functions](#)
 - [Manually run a non HTTP-triggered function](#)
- [Handling binding errors](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

Register Azure Functions binding extensions

Article • 06/26/2024

Starting with Azure Functions version 2.x, the functions runtime only includes HTTP and timer triggers by default. Other [triggers and bindings](#) are available as separate packages.

.NET class library functions apps use bindings that are installed in the project as NuGet packages. Extension bundles allow non-.NET functions apps to use the same bindings without having to deal with the .NET infrastructure.

The following table indicates when and how you register bindings.

[+] Expand table

Development environment	Registration in Functions 1.x	Registration in Functions 2.x or later
Azure portal	Automatic	Automatic*
Non-.NET languages	Automatic	Use extension bundles (recommended) or explicitly install extensions
C# class library using Visual Studio	Use NuGet tools	Use NuGet tools
C# class library using Visual Studio Code	N/A	Use .NET Core CLI

* Portal uses extension bundles, including C# script.

Extension bundles

By default, extension bundles provide binding support for functions in these languages:

- Java
- JavaScript
- PowerShell
- Python
- C# script
- Other (custom handlers)

In rare cases where extension bundles can't be used, you can explicitly install binding extensions with your function app project. Extension bundles are supported for version 2.x and later version of the Functions runtime.

Extension bundles are a way to add a pre-defined set of compatible binding extensions to your function app. Extension bundles are versioned. Each version contains a specific set of binding extensions that are verified to work together. Select a bundle version based on the extensions that you need in your app.

When you create a non-.NET Functions project from tooling or in the portal, extension bundles are already enabled in the app's `host.json` file.

An extension bundle reference is defined by the `extensionBundle` section in a `host.json` as follows:

```
JSON
{
    "version": "2.0",
    "extensionBundle": {
        "id": "Microsoft.Azure.Functions.ExtensionBundle",
        "version": "[4.0.0, 5.0.0)"
    }
}
```

The following properties are available in `extensionBundle`:

[+] Expand table

Property	Description
<code>id</code>	The namespace for Microsoft Azure Functions extension bundles.
<code>version</code>	The version range of the bundle to install. The Functions runtime always picks the maximum permissible version defined by the version range or interval. For example, a <code>version</code> value range of <code>[4.0.0, 5.0.0)</code> allows all bundle versions from 4.0.0 up to but not including 5.0.0. For more information, see the interval notation for specifying version ranges .

The following table lists the currently available version ranges of the default `Microsoft.Azure.Functions.ExtensionBundle` bundles and links to the extensions they include.

[+] Expand table

Bundle version	Version in host.json	Included extensions
1.x	[1.* , 2.0.0)	See extensions.json ↗ used to generate the bundle.
2.x	[2.* , 3.0.0)	See extensions.json ↗ used to generate the bundle.
3.x	[3.3.0, 4.0.0)	See extensions.json ↗ used to generate the bundle.
4.x	[4.0.0, 5.0.0)	See extensions.json ↗ used to generate the bundle.

ⓘ Note

Even though host.json supports custom ranges for `version`, you should use a version range value from this table, such as [4.0.0, 5.0.0). For a complete list of extension bundle releases and extension versions in each release, see the [extension bundles release page](#) ↗.

Explicitly install extensions

For compiled C# class library projects ([in-process](#) and [isolated worker process](#)), you install the NuGet packages for the extensions that you need as you normally would. For examples see either the [Visual Studio Code developer guide](#) or the [Visual Studio developer guide](#). See the [extension bundles release page](#) ↗ to review combinations of extension versions that are verified compatible.

For non-.NET languages and C# script, when you can't use extension bundles you need to manually install required binding extensions in your local project. The easiest way is to use Azure Functions Core Tools. For more information, see [func extensions install](#).

For portal-only development, you need to manually create an extensions.csproj file in the root of your function app. To learn more, see [Manually install extensions](#).

Next steps

[Azure Function trigger and binding example](#)

Feedback

Was this page helpful?

👍 Yes

👎 No

Provide product feedback ↗

Azure Functions binding expression patterns

Article • 08/31/2023

One of the most powerful features of [triggers and bindings](#) is *binding expressions*. In the `function.json` file and in function parameters and code, you can use expressions that resolve to values from various sources.

Most expressions are identified by wrapping them in curly braces. For example, in a queue trigger function, `{queueTrigger}` resolves to the queue message text. If the `path` property for a blob output binding is `container/{queueTrigger}` and the function is triggered by a queue message `HelloWorld`, a blob named `HelloWorld` is created.

Types of binding expressions

- [App settings](#)
- [Trigger file name](#)
- [Trigger metadata](#)
- [JSON payloads](#)
- [New GUID](#)
- [Current date and time](#)

Binding expressions - app settings

As a best practice, secrets and connection strings should be managed using app settings, rather than configuration files. This limits access to these secrets and makes it safe to store files such as `function.json` in public source control repositories.

App settings are also useful whenever you want to change configuration based on the environment. For example, in a test environment, you may want to monitor a different queue or blob storage container.

App setting binding expressions are identified differently from other binding expressions: they are wrapped in percent signs rather than curly braces. For example if the blob output binding path is `%Environment%/newblob.txt` and the `Environment` app setting value is `Development`, a blob will be created in the `Development` container.

When a function is running locally, app setting values come from the `local.settings.json` file.

ⓘ Note

The `connection` property of triggers and bindings is a special case and automatically resolves values as app settings, without percent signs.

The following example is an Azure Queue Storage trigger that uses an app setting `%input_queue_name%` to define the queue to trigger on.

JSON

```
{  
  "bindings": [  
    {  
      "name": "order",  
      "type": "queueTrigger",  
      "direction": "in",  
      "queueName": "%input_queue_name%",  
      "connection": "MY_STORAGE_ACCT_APP_SETTING"  
    }  
  ]  
}
```

You can use the same approach in class libraries:

C#

```
[FunctionName("QueueTrigger")]
public static void Run(
    [QueueTrigger("%input_queue_name%")]string myQueueItem,
    ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");
}
```

Trigger file name

The `path` for a Blob trigger can be a pattern that lets you refer to the name of the triggering blob in other bindings and function code. The pattern can also include filtering criteria that specify which blobs can trigger a function invocation.

For example, in the following Blob trigger binding, the `path` pattern is `sample-images/{filename}`, which creates a binding expression named `filename`:

JSON

```
{  
  "bindings": [  
    {  
      "name": "image",  
      "type": "blobTrigger",  
      "path": "sample-images/{filename}",  
      "direction": "in",  
      "connection": "MyStorageConnection"  
    },  
    ...  
  ]
```

The expression `filename` can then be used in an output binding to specify the name of the blob being created:

JSON

```
...  
{  
  "name": "imageSmall",  
  "type": "blob",  
  "path": "sample-images-sm/{filename}",  
  "direction": "out",  
  "connection": "MyStorageConnection"  
}  
],  
}
```

Function code has access to this same value by using `filename` as a parameter name:

C#

```
// C# example of binding to {filename}  
public static void Run(Stream image, string filename, Stream imageSmall,  
ILogger log)  
{  
    log.LogInformation($"Blob trigger processing: {filename}");  
    // ...  
}
```

The same ability to use binding expressions and patterns applies to attributes in class libraries. In the following example, the attribute constructor parameters are the same `path` values as the preceding `function.json` examples:

C#

```
[FunctionName("ResizeImage")]  
public static void Run(  
    [BlobTrigger("sample-images/{filename}")] Stream image,
```

```
[Blob("sample-images-sm/{filename}", FileAccess.Write)] Stream
imageSmall,
    string filename,
    ILogger log)
{
    log.LogInformation($"Blob trigger processing: {filename}");
    // ...
}
```

You can also create expressions for parts of the file name. In the following example, function is triggered only on file names that match a pattern: `anyname-anyfile.csv`

JSON

```
{
    "name": "myBlob",
    "type": "blobTrigger",
    "direction": "in",
    "path": "testContainerName/{date}-{filetype}.csv",
    "connection": "OrderStorageConnection"
}
```

For more information on how to use expressions and patterns in the Blob path string, see the [Storage blob binding reference](#).

Trigger metadata

In addition to the data payload provided by a trigger (such as the content of the queue message that triggered a function), many triggers provide additional metadata values. These values can be used as input parameters in C# and F# or properties on the `context.bindings` object in JavaScript.

For example, an Azure Queue storage trigger supports the following properties:

- QueueTrigger - triggering message content if a valid string
- DequeueCount
- ExpirationTime
- Id
- InsertionTime
- NextVisibleTime
- PopReceipt

These metadata values are accessible in `function.json` file properties. For example, suppose you use a queue trigger and the queue message contains the name of a blob

you want to read. In the `function.json` file, you can use `queueTrigger` metadata property in the blob `path` property, as shown in the following example:

```
JSON

{
  "bindings": [
    {
      "name": "myQueueItem",
      "type": "queueTrigger",
      "queueName": "myqueue-items",
      "connection": "MyStorageConnection",
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "direction": "in",
      "connection": "MyStorageConnection"
    }
  ]
}
```

Details of metadata properties for each trigger are described in the corresponding reference article. For an example, see [queue trigger metadata](#). Documentation is also available in the **Integrate** tab of the portal, in the **Documentation** section below the binding configuration area.

JSON payloads

In some scenarios, you can refer to the trigger payload's properties in configuration for other bindings in the same function and in function code. This requires that the trigger payload is JSON and is smaller than a threshold specific to each trigger. Typically, the payload size needs to be less than 100MB, but you should check the reference content for each trigger. Using trigger payload properties may impact the performance of your application, and it forces the trigger parameter type to be simple types like strings or a custom object type representing JSON data. It cannot be used with streams, clients, or other SDK types.

The following example shows the `function.json` file for a webhook function that receives a blob name in JSON: `{"BlobName": "HelloWorld.txt"}`. A Blob input binding reads the blob, and the HTTP output binding returns the blob contents in the HTTP response. Notice that the Blob input binding gets the blob name by referring directly to the `BlobName` property (`"path": "strings/{BlobName}"`)

JSON

```
{  
  "bindings": [  
    {  
      "name": "info",  
      "type": "httpTrigger",  
      "direction": "in",  
      "webHookType": "genericJson"  
    },  
    {  
      "name": "blobContents",  
      "type": "blob",  
      "direction": "in",  
      "path": "strings/{BlobName}",  
      "connection": "AzureWebJobsStorage"  
    },  
    {  
      "name": "res",  
      "type": "http",  
      "direction": "out"  
    }  
  ]  
}
```

For this to work in C# and F#, you need a class that defines the fields to be deserialized, as in the following example:

C#

```
using System.Net;  
using Microsoft.Extensions.Logging;  
  
public class BlobInfo  
{  
    public string BlobName { get; set; }  
}  
  
public static HttpResponseMessage Run(HttpRequestMessage req, BlobInfo info,  
string blobContents, ILogger log)  
{  
    if (blobContents == null) {  
        return req.CreateResponse(HttpStatusCode.NotFound);  
    }  
  
    log.LogInformation($"Processing: {info.BlobName}");  
  
    return req.CreateResponse(HttpStatusCode.OK, new {  
        data = $"{blobContents}"  
    });  
}
```

In JavaScript, JSON deserialization is automatically performed.

JavaScript

```
module.exports = async function (context, info) {
    if ('BlobName' in info) {
        context.res = {
            body: { 'data': context.bindings.blobContents }
        }
    } else {
        context.res = {
            status: 404
        };
    }
}
```

Dot notation

If some of the properties in your JSON payload are objects with properties, you can refer to those directly by using dot (.) notation. This notation doesn't work for [Azure Cosmos DB](#) or [Table storage](#) bindings.

For example, suppose your JSON looks like this:

JSON

```
{
    "BlobName": {
        "FileName": "HelloWorld",
        "Extension": "txt"
    }
}
```

You can refer directly to `FileName` as `BlobName.FileName`. With this JSON format, here's what the `path` property in the preceding example would look like:

JSON

```
"path": "strings/{BlobName.FileName}.{BlobName.Extension}",
```

In C#, you would need two classes:

C#

```
public class BlobInfo
{
    public BlobName BlobName { get; set; }
}
public class BlobName
{
    public string FileName { get; set; }
    public string Extension { get; set; }
}
```

Create GUIDs

The `{rand-guid}` binding expression creates a GUID. The following blob path in a `function.json` file creates a blob with a name like `50710cb5-84b9-4d87-9d83-a03d6976a682.txt`.

JSON

```
{
    "type": "blob",
    "name": "blobOutput",
    "direction": "out",
    "path": "my-output-container/{rand-guid}.txt"
}
```

Current time

The binding expression `DateTime` resolves to `DateTime.UtcNow`. The following blob path in a `function.json` file creates a blob with a name like `2018-02-16T17-59-55Z.txt`.

JSON

```
{
    "type": "blob",
    "name": "blobOutput",
    "direction": "out",
    "path": "my-output-container/{DateTime}.txt"
}
```

Binding at runtime

In C# and other .NET languages, you can use an imperative binding pattern, as opposed to the declarative bindings in `function.json` and attributes. Imperative binding is useful

when binding parameters need to be computed at runtime rather than design time. To learn more, see the [C# developer reference](#) or the [C# script developer reference](#).

Next steps

[Using the Azure Function return value](#)

Azure Functions error handling and retries

Article • 04/26/2024

Handling errors in Azure Functions is important to help you avoid lost data, avoid missed events, and monitor the health of your application. It's also an important way to help you understand the retry behaviors of event-based triggers.

This article describes general strategies for error handling and the available retry strategies.

ⓘ Important

Preview retry policy support for certain triggers was removed in December 2022.

Retry policies for supported triggers are now generally available (GA). For a list of extensions that currently support retry policies, see the [Retries](#) section.

Handling errors

Errors that occur in an Azure function can come from:

- Use of built-in Functions [triggers and bindings](#).
- Calls to APIs of underlying Azure services.
- Calls to REST endpoints.
- Calls to client libraries, packages, or third-party APIs.

To avoid loss of data or missed messages, it's important to practice good error handling. This table describes some recommended error-handling practices and provides links to more information.

[] [Expand table](#)

Recommendation	Details
Enable Application Insights	Azure Functions integrates with Application Insights to collect error data, performance data, and runtime logs. You should use Application Insights to discover and better understand errors that occur in your function executions. To learn more, see Monitor Azure Functions .
Use structured error handling	Capturing and logging errors is critical to monitoring the health of your application. The top-most level of any function code should include a

Recommendation	Details
	try/catch block. In the catch block, you can capture and log errors. For information about what errors might be raised by bindings, see Binding error codes . Depending on your specific retry strategy, you might also raise a new exception to run the function again.
Plan your retry strategy	Several Functions bindings extensions provide built-in support for retries and others let you define retry policies, which are implemented by the Functions runtime. For triggers that don't provide retry behaviors, you should consider implementing your own retry scheme. For more information, see Retries .
Design for idempotency	The occurrence of errors when you're processing data can be a problem for your functions, especially when you're processing messages. It's important to consider what happens when the error occurs and how to avoid duplicate processing. To learn more, see Designing Azure Functions for identical input .

Retries

There are two kinds of retries available for your functions:

- Built-in retry behaviors of individual trigger extensions
- Retry policies provided by the Functions runtime

The following table indicates which triggers support retries and where the retry behavior is configured. It also links to more information about errors that come from the underlying services.

[] [Expand table](#)

Trigger/binding	Retry source	Configuration
Azure Cosmos DB	Retry policies	Function-level
Blob Storage	Binding extension	host.json
Event Grid	Binding extension	Event subscription
Event Hubs	Retry policies	Function-level
Kafka	Retry policies	Function-level
Queue Storage	Binding extension	host.json
RabbitMQ	Binding extension	Dead letter queue ↗

Trigger/binding	Retry source	Configuration
Service Bus	Binding extension	host.json*
Timer	Retry policies	Function-level

*Requires version 5.x of the Azure Service Bus extension. In older extension versions, retry behaviors are implemented by the [Service Bus dead letter queue](#).

Retry policies

Azure Functions lets you define retry policies for specific trigger types, which are enforced by the runtime. These trigger types currently support retry policies:

- [Azure Cosmos DB](#)
- [Event Hubs](#)
- [Kafka](#)
- [Timer](#)

Retry policies aren't supported in version 1.x of the Functions runtime.

The retry policy tells the runtime to rerun a failed execution until either successful completion occurs or the maximum number of retries is reached.

A retry policy is evaluated when a function executed by a supported trigger type raises an uncaught exception. As a best practice, you should catch all exceptions in your code and raise new exceptions for any errors that you want to result in a retry.

Important

Event Hubs checkpoints aren't written until after the retry policy for the execution has completed. Because of this behavior, progress on the specific partition is paused until the current batch is done processing.

The version 5.x of the Event Hubs extension supports additional retry capabilities for interactions between the Functions host and the event hub. For more information, see `clientRetryOptions` in the [Event Hubs host.json reference](#).

Retry strategies

You can configure two retry strategies that are supported by policy:

Fixed delay

A specified amount of time is allowed to elapse between each retry.

When running in a Consumption plan, you are only billed for time your function code is executing. You aren't billed for the wait time between executions in either of these retry strategies.

Max retry counts

You can configure the maximum number of times that a function execution is retried before eventual failure. The current retry count is stored in memory of the instance.

It's possible for an instance to have a failure between retry attempts. When an instance fails during a retry policy, the retry count is lost. When there are instance failures, the Event Hubs trigger is able to resume processing and retry the batch on a new instance, with the retry count reset to zero. The timer trigger doesn't resume on a new instance.

This behavior means that the maximum retry count is a best effort. In some rare cases, an execution could be retried more than the requested maximum number of times. For Timer triggers, the retries can be less than the maximum number requested.

Retry examples

Examples are provided for both fixed delay and exponential backoff strategies. To see examples for a specific strategy, you must first select that strategy in the previous tab.

Isolated worker model

Function-level retries are supported with the following NuGet packages:

- [Microsoft.Azure.Functions.Worker.Sdk](#) >= 1.9.0
- [Microsoft.Azure.Functions.Worker.Extensions.EventHubs](#) >= 5.2.0
- [Microsoft.Azure.Functions.Worker.Extensions.Kafka](#) >= 3.8.0
- [Microsoft.Azure.Functions.Worker.Extensions.Timer](#) >= 4.2.0

C#

```
[Function(nameof(TimerFunction))]
[FixedDelayRetry(5, "00:00:10")]
public static void Run([TimerTrigger("0 */5 * * *")] TimerInfo
timerInfo,
    FunctionContext context)
```

```
{  
    var logger = context.GetLogger(nameof(TimerFunction));  
    logger.LogInformation($"Function Ran. Next timer schedule =  
{timerInfo.ScheduleStatus.Next}");  
}
```

[+] [Expand table](#)

Property	Description
MaxRetryCount	Required. The maximum number of retries allowed per function execution. -1 means to retry indefinitely.
DelayInterval	The delay used between retries. Specify it as a string with the format HH:mm:ss.

Binding error codes

When you're integrating with Azure services, errors might originate from the APIs of the underlying services. Information that relates to binding-specific errors is available in the "Exceptions and return codes" sections of the following articles:

- [Azure Cosmos DB](#)
- [Blob Storage](#)
- [Event Grid](#)
- [Event Hubs](#)
- [IoT Hub](#)
- [Notification Hubs](#)
- [Queue Storage](#)
- [Service Bus](#)
- [Table Storage](#)

Next steps

- [Azure Functions triggers and bindings concepts](#)
- [Best practices for reliable Azure functions](#)

Shifting from Express.js to Azure Functions

Article • 09/20/2022

Express.js is one of the most popular Node.js frameworks for web developers and remains an excellent choice for building apps that serve API endpoints.

When migrating code to a serverless architecture, refactoring Express.js endpoints affects the following areas:

- **Middleware:** Express.js features a robust collection of middleware. Many middleware modules are no longer required in light of Azure Functions and [Azure API Management](#) capabilities. Ensure you can replicate or replace any logic handled by essential middleware before migrating endpoints.
- **Differing APIs:** The API used to process both requests and responses differs among Azure Functions and Express.js. The following example details the required changes.
- **Default route:** By default, Azure Functions endpoints are exposed under the `api` route. Routing rules are configurable via [routePrefix in the host.json file](#).
- **Configuration and conventions:** A Functions app uses the `function.json` file to define HTTP verbs, define security policies, and can configure the function's [input and output](#). By default, the folder name that which contains the function files defines the endpoint name, but you can change the name via the `route` property in the [function.json](#) file.

💡 Tip

Learn more through the interactive tutorial [Refactor Node.js and Express APIs to Serverless APIs with Azure Functions](#).

Example

Express.js

The following example shows a typical Express.js `GET` endpoint.

JavaScript

```
// server.js
app.get('/hello', (req, res) => {
  try {
    res.send("Success!");
  } catch(error) {
    const err = JSON.stringify(error);
    res.status(500).send(`Request error. ${err}`);
  }
});
```

When a `GET` request is sent to `/hello`, an `HTTP 200` response containing `Success` is returned. If the endpoint encounters an error, the response is an `HTTP 500` with the error details.

Azure Functions

Azure Functions organizes configuration and code files into a single folder for each function. By default, the name of the folder dictates the function name.

For instance, a function named `hello` has a folder with the following files.

files

```
| - hello
|   - function.json
|   - index.js
```

The following example implements the same result as the above Express.js endpoint, but with Azure Functions.

JavaScript

```
JavaScript

// hello/index.js
module.exports = async function (context, req) {
  try {
    context.res = { body: "Success!" };
  } catch(error) {
    const err = JSON.stringify(error);
    context.res = {
      status: 500,
      body: `Request error. ${err}`
    };
}
```

```
    }  
};
```

When moving to Functions, the following changes are made:

- **Module:** The function code is implemented as a JavaScript module.
- **Context and response object:** The [context](#) allows you to communicate with the Function's runtime. From the context, you can read request data and set the function's response. Synchronous code requires you to call 1.x `context.done()` to complete execution, while 2.x+ `async` functions resolve the request implicitly.
- **Naming convention:** The folder name used to contain the Azure Functions files is used as the endpoint name by default (this can be overridden in the [function.json](#)).
- **Configuration:** You define the HTTP verbs in the [function.json](#) file such as `POST` or `PUT`.

The following *function.json* file holds configuration information for the function.

JSON

```
{  
  "bindings": [  
    {  
      "authLevel": "function",  
      "type": "httpTrigger",  
      "direction": "in",  
      "name": "req",  
      "methods": ["get"]  
    },  
    {  
      "type": "http",  
      "direction": "out",  
      "name": "res"  
    }  
  ]  
}
```

By defining `get` in the `methods` array, the function is available to HTTP `GET` requests. If you want to your API to accept support `POST` requests, you can add `post` to the array as well.

Next steps

- Learn more with the interactive tutorial [Refactor Node.js and Express APIs to Serverless APIs with Azure Functions](#)

Securing Azure Functions

Article • 07/18/2024

In many ways, planning for secure development, deployment, and operation of serverless functions is much the same as for any web-based or cloud-hosted application. [Azure App Service](#) provides the hosting infrastructure for your function apps. This article provides security strategies for running your function code, and how App Service can help you secure your functions.

The platform components of App Service, including Azure VMs, storage, network connections, web frameworks, management and integration features, are actively secured and hardened. App Service goes through vigorous compliance checks on a continuous basis to make sure that:

- Your app resources are [secured ↗](#) from the other customers' Azure resources.
- [VM instances and runtime software are regularly updated](#) to address newly discovered vulnerabilities.
- Communication of secrets (such as connection strings) between your app and other Azure resources (such as [SQL Database ↗](#)) stays within Azure and doesn't cross any network boundaries. Secrets are always encrypted when stored.
- All communication over the App Service connectivity features, such as [hybrid connection](#), is encrypted.
- Connections with remote management tools like Azure PowerShell, Azure CLI, Azure SDKs, REST APIs, are all encrypted.
- 24-hour threat management protects the infrastructure and platform against malware, distributed denial-of-service (DDoS), man-in-the-middle (MITM), and other threats.

For more information on infrastructure and platform security in Azure, see [Azure Trust Center ↗](#).

For a set of security recommendations that follow the [Microsoft cloud security benchmark](#), see [Azure Security Baseline for Azure Functions](#).

Secure operation

This section guides you on configuring and running your function app as securely as possible.

Defender for Cloud

Defender for Cloud integrates with your function app in the portal. It provides, for free, a quick assessment of potential configuration-related security vulnerabilities. Function apps running in a dedicated plan can also use Defender for Cloud's enhanced security features for an extra cost. To learn more, see [Protect your Azure App Service web apps and APIs](#).

Log and monitor

One way to detect attacks is through activity monitoring and logging analytics. Functions integrates with Application Insights to collect log, performance, and error data for your function app. Application Insights automatically detects performance anomalies and includes powerful analytics tools to help you diagnose issues and to understand how your functions are used. To learn more, see [Monitor Azure Functions](#).

Functions also integrates with Azure Monitor Logs to enable you to consolidate function app logs with system events for easier analysis. You can use diagnostic settings to configure streaming export of platform logs and metrics for your functions to the destination of your choice, such as a Logs Analytics workspace. To learn more, see [Monitoring Azure Functions with Azure Monitor Logs](#).

For enterprise-level threat detection and response automation, stream your logs and events to a Logs Analytics workspace. You can then connect Microsoft Sentinel to this workspace. To learn more, see [What is Microsoft Sentinel](#).

For more security recommendations for observability, see the [Azure security baseline for Azure Functions](#).

Secure HTTP endpoints

HTTP endpoints that are exposed publicly provide a vector of attack for malicious actors. When securing your HTTP endpoints, you should use a layered security approach. These techniques can be used to reduce the vulnerability of publicly exposed HTTP endpoints, ordered from most basic to most secure and restrictive:

- [Require HTTPS](#)
- [Require access keys](#)
- [Enable App Service Authentication/Authorization](#)
- [Use Azure API Management \(APIM\) to authenticate requests](#)
- [Deploy your function app to a virtual network](#)
- [Deploy your function app in isolation](#)

Require HTTPS

By default, clients can connect to function endpoints by using both HTTP or HTTPS. You should redirect HTTP to HTTPS because HTTPS uses the SSL/TLS protocol to provide a secure connection, which is both encrypted and authenticated. To learn how, see [Enforce HTTPS](#).

When you require HTTPS, you should also require the latest TLS version. To learn how, see [Enforce TLS versions](#).

For more information, see [Secure connections \(TLS\)](#).

Function access keys

Functions lets you use keys to make it harder to access your function endpoints. Unless the HTTP access level on an HTTP triggered function is set to `anonymous`, requests must include an access key in the request. For more information, see [Work with access keys in Azure Functions](#).

While access keys can provide some mitigation for unwanted access, the only way to truly secure your function endpoints is by implementing positive authentication of clients accessing your functions. You can then make authorization decisions based on identity.

For the highest level of security, you can also secure the entire application architecture inside a virtual network [using private endpoints](#) or by [running in isolation](#).

Enable App Service Authentication/Authorization

The App Service platform lets you use Microsoft Entra ID and several third-party identity providers to authenticate clients. You can use this strategy to implement custom authorization rules for your functions, and you can work with user information from your function code. To learn more, see [Authentication and authorization in Azure App Service](#) and [Working with client identities](#).

Use Azure API Management (APIM) to authenticate requests

APIM provides various API security options for incoming requests. To learn more, see [API Management authentication policies](#). With APIM in place, you can configure your function app to accept requests only from the IP address of your APIM instance. To learn more, see [IP address restrictions](#).

Permissions

As with any application or service, the goal is run your function app with the lowest possible permissions.

User management permissions

Functions supports built-in Azure role-based access control ([Azure RBAC](#)). Azure roles supported by Functions are [Contributor](#), [Owner](#), and [Reader](#).

Permissions are effective at the function app level. The Contributor role is required to perform most function app-level tasks. You also need the Contributor role along with the [Monitoring Reader permission](#) to be able to view log data in Application Insights. Only the Owner role can delete a function app.

Organize functions by privilege

Connection strings and other credentials stored in application settings gives all of the functions in the function app the same set of permissions in the associated resource. Consider minimizing the number of functions with access to specific credentials by moving functions that don't use those credentials to a separate function app. You can always use techniques such as [function chaining](#) to pass data between functions in different function apps.

Managed identities

A managed identity from Microsoft Entra ID allows your app to easily access other Microsoft Entra protected resources such as Azure Key Vault. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. For more about managed identities in Microsoft Entra ID, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to the app and is deleted if the app is deleted. An app can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your app. An app can have multiple user-assigned identities, and one user-assigned identity can be assigned to multiple Azure resources, such as two App Service apps.

Managed identities can be used in place of secrets for connections from some triggers and bindings. See [Identity-based connections](#).

For more information, see [How to use managed identities for App Service and Azure Functions](#).

Restrict CORS access

[Cross-origin resource sharing \(CORS\)](#) is a way to allow web apps running in another domain to make requests to your HTTP trigger endpoints. App Service provides built-in support for handing the required CORS headers in HTTP requests. CORS rules are defined on a function app level.

While it's tempting to use a wildcard that allows all sites to access your endpoint, this defeats the purpose of CORS, which is to help prevent cross-site scripting attacks. Instead, add a separate CORS entry for the domain of each web app that must access your endpoint.

Managing secrets

To be able to connect to the various services and resources need to run your code, function apps need to be able to access secrets, such as connection strings and service keys. This section describes how to store secrets required by your functions.

Never store secrets in your function code.

Application settings

By default, you store connection strings and secrets used by your function app and bindings as application settings. This makes these credentials available to both your function code and the various bindings used by the function. The application setting (key) name is used to retrieve the actual value, which is the secret.

For example, every function app requires an associated storage account, which is used by the runtime. By default, the connection to this storage account is stored in an application setting named `AzureWebJobsStorage`.

App settings and connection strings are stored encrypted in Azure. They're decrypted only before being injected into your app's process memory when the app starts. The encryption keys are rotated regularly. If you prefer to instead manage the secure storage of your secrets, the app setting should instead be references to Azure Key Vault.

You can also encrypt settings by default in the local.settings.json file when developing functions on your local computer. For more information, see [Encrypt the local settings file](#).

Key Vault references

While application settings are sufficient for most many functions, you may want to share the same secrets across multiple services. In this case, redundant storage of secrets results in more potential vulnerabilities. A more secure approach is to a central secret storage service and use references to this service instead of the secrets themselves.

[Azure Key Vault](#) is a service that provides centralized secrets management, with full control over access policies and audit history. You can use a Key Vault reference in the place of a connection string or key in your application settings. To learn more, see [Use Key Vault references for App Service and Azure Functions](#).

Identity-based connections

Identities may be used in place of secrets for connecting to some resources. This has the advantage of not requiring the management of a secret, and it provides more fine-grained access control and auditing.

When you're writing code that creates the connection to [Azure services that support Microsoft Entra authentication](#), you can choose to use an identity instead of a secret or connection string. Details for both connection methods are covered in the documentation for each service.

Some Azure Functions binding extensions can be configured to access services using identity-based connections. For more information, see [Configure an identity-based connection](#).

Set usage quotas

Consider setting a usage quota on functions running in a Consumption plan. When you set a daily GB-sec limit on the sum total execution of functions in your function app, execution is stopped when the limit is reached. This could potentially help mitigate against malicious code executing your functions. To learn how to estimate consumption for your functions, see [Estimating Consumption plan costs](#).

Data validation

The triggers and bindings used by your functions don't provide any additional data validation. Your code must validate any data received from a trigger or input binding. If an upstream service is compromised, you don't want unvalidated inputs flowing through your functions. For example, if your function stores data from an Azure Storage queue in a relational database, you must validate the data and parameterize your commands to avoid SQL injection attacks.

Don't assume that the data coming into your function has already been validated or sanitized. It's also a good idea to verify that the data being written to output bindings is valid.

Handle errors

While it seems basic, it's important to write good error handling in your functions. Unhandled errors bubble-up to the host and are handled by the runtime. Different bindings handle processing of errors differently. To learn more, see [Azure Functions error handling](#).

Disable remote debugging

Make sure that remote debugging is disabled, except when you are actively debugging your functions. You can disable remote debugging in the **General Settings** tab of your function app **Configuration** in the portal.

Restrict CORS access

Azure Functions supports cross-origin resource sharing (CORS). CORS is configured [in the portal](#) and through the [Azure CLI](#). The CORS allowed origins list applies at the function app level. With CORS enabled, responses include the `Access-Control-Allow-Origin` header. For more information, see [Cross-origin resource sharing](#).

Don't use wildcards in your allowed origins list. Instead, list the specific domains from which you expect to get requests.

Store data encrypted

Azure Storage encrypts all data in a storage account at rest. For more information, see [Azure Storage encryption for data at rest](#).

By default, data is encrypted with Microsoft-managed keys. For additional control over encryption keys, you can supply customer-managed keys to use for encryption of blob

and file data. These keys must be present in Azure Key Vault for Functions to be able to access the storage account. To learn more, see [Encryption at rest using customer-managed keys](#).

Secure related resources

A function app frequently depends on additional resources, so part of securing the app is securing these external resources. At a minimum, most function apps include a dependency on Application Insights and Azure Storage. Consult the [Azure security baseline for Azure Monitor](#) and the [Azure security baseline for Storage](#) for guidance on securing these resources.

Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

You should also consult the guidance for any resource types your application logic depends on, both as triggers and bindings and from your function code.

Secure deployment

Azure Functions tooling and integration make it easy to publish local function project code to Azure. It's important to understand how deployment works when considering security for an Azure Functions topology.

Deployment credentials

App Service deployments require a set of deployment credentials. These deployment credentials are used to secure your function app deployments. Deployment credentials are managed by the App Service platform and are encrypted at rest.

There are two kinds of deployment credentials:

- **User-level credentials:** one set of credentials for the entire Azure account. It can be used to deploy to App Service for any app, in any subscription, that the Azure account has permission to access. It's the default set that's surfaced in the portal GUI (such as the [Overview](#) and [Properties](#) of the app's [resource page](#)). When a user is granted app access via Role-Based Access Control (RBAC) or coadmin

permissions, that user can use their own user-level credentials until the access is revoked. Do not share these credentials with other Azure users.

- **App-level credentials:** one set of credentials for each app. It can be used to deploy to that app only. The credentials for each app are generated automatically at app creation. They can't be configured manually, but can be reset anytime. For a user to be granted access to app-level credentials via (RBAC), that user must be contributor or higher on the app (including Website Contributor built-in role). Readers are not allowed to publish, and can't access those credentials.

At this time, Key Vault isn't supported for deployment credentials. To learn more about managing deployment credentials, see [Configure deployment credentials for Azure App Service](#).

Disable FTP

By default, each function app has an FTP endpoint enabled. The FTP endpoint is accessed using deployment credentials.

FTP isn't recommended for deploying your function code. FTP deployments are manual, and they require you to synchronize triggers. To learn more, see [FTP deployment](#).

When you're not planning on using FTP, you should disable it in the portal. If you do choose to use FTP, you should [enforce FTPS](#).

Secure the scm endpoint

Every function app has a corresponding `scm` service endpoint that is used by the Advanced Tools (Kudu) service for deployments and other App Service [site extensions](#). The scm endpoint for a function app is always a URL in the form `https://<FUNCTION_APP_NAME>.scm.azurewebsites.net`. When you use network isolation to secure your functions, you must also account for this endpoint.

By having a separate scm endpoint, you can control deployments and other advanced tools functionalities for function app that are isolated or running in a virtual network. The scm endpoint supports both basic authentication (using deployment credentials) and single sign-on with your Azure portal credentials. To learn more, see [Accessing the Kudu service](#).

Continuous security validation

Since security needs to be considered at every step in the development process, it makes sense to also implement security validations in a continuous deployment environment. This is sometimes called DevSecOps. Using Azure DevOps for your deployment pipeline lets you integrate validation into the deployment process. For more information, see [Learn how to add continuous security validation to your CI/CD pipeline](#).

Network security

Restricting network access to your function app lets you control who can access your functions endpoints. Functions leverages App Service infrastructure to enable your functions to access resources without using internet-routable addresses or to restrict internet access to a function endpoint. To learn more about these networking options, see [Azure Functions networking options](#).

Set access restrictions

Access restrictions allow you to define lists of allow/deny rules to control traffic to your app. Rules are evaluated in priority order. If there are no rules defined, then your app will accept traffic from any address. To learn more, see [Azure App Service Access Restrictions](#).

Secure the storage account

When you create a function app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. You can replace this storage account with one that is secured by a virtual network with access enabled by service endpoints or private endpoints. For more information, see [Restrict your storage account to a virtual network](#).

Deploy your function app to a virtual network

[Azure Private Endpoint](#) is a network interface that connects you privately and securely to a service powered by Azure Private Link. Private Endpoint uses a private IP address from your virtual network, effectively bringing the service into your virtual network.

You can use Private Endpoint for your functions hosted in the [Premium](#) and [App Service](#) plans.

If you want to make calls to Private Endpoints, then you must make sure that your DNS lookups resolve to the private endpoint. You can enforce this behavior in one of the

following ways:

- Integrate with Azure DNS private zones. When your virtual network doesn't have a custom DNS server, this is done automatically.
- Manage the private endpoint in the DNS server used by your app. To do this you must know the private endpoint address and then point the endpoint you are trying to reach to that address using an A record.
- Configure your own DNS server to forward to [Azure DNS private zones](#).

To learn more, see [using Private Endpoints for Web Apps](#).

Deploy your function app in isolation

Azure App Service Environment provides a dedicated hosting environment in which to run your functions. These environments let you configure a single front-end gateway that you can use to authenticate all incoming requests. For more information, see [Configuring a Web Application Firewall \(WAF\) for App Service Environment](#).

Use a gateway service

Gateway services, such as [Azure Application Gateway](#) and [Azure Front Door](#) let you set up a Web Application Firewall (WAF). WAF rules are used to monitor or block detected attacks, which provide an extra layer of protection for your functions. To set up a WAF, your function app needs to be running in an ASE or using Private Endpoints (preview). To learn more, see [Using Private Endpoints](#).

Next steps

- [Azure Security Baseline for Azure Functions](#)
- [Azure Functions diagnostics](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure security baseline for Functions

Article • 12/26/2023

This security baseline applies guidance from the [Microsoft cloud security benchmark version 1.0](#) to Functions. The Microsoft cloud security benchmark provides recommendations on how you can secure your cloud solutions on Azure. The content is grouped by the security controls defined by the Microsoft cloud security benchmark and the related guidance applicable to Functions.

You can monitor this security baseline and its recommendations using Microsoft Defender for Cloud. Azure Policy definitions will be listed in the Regulatory Compliance section of the Microsoft Defender for Cloud portal page.

When a feature has relevant Azure Policy Definitions, they are listed in this baseline to help you measure compliance with the Microsoft cloud security benchmark controls and recommendations. Some recommendations may require a paid Microsoft Defender plan to enable certain security scenarios.

ⓘ Note

Features not applicable to Functions have been excluded. To see how Functions completely maps to the Microsoft cloud security benchmark, see the [full Functions security baseline mapping file ↗](#).

Security profile

The security profile summarizes high-impact behaviors of Functions, which may result in increased security considerations.

ⓘ [Expand table](#)

Service Behavior Attribute	Value
Product Category	Compute, Web
Customer can access HOST / OS	No Access
Service can be deployed into customer's virtual network	True
Stores customer content at rest	True

Network security

For more information, see the [Microsoft cloud security benchmark: Network security](#).

NS-1: Establish network segmentation boundaries

Features

Virtual Network Integration

Description: Service supports deployment into customer's private Virtual Network (VNet). [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Deploy the service into a virtual network. Assign private IPs to the resource (where applicable) unless there is a strong reason to assign public IPs directly to the resource.

Note: Networking features are exposed by the service but need to be configured for the application. By default, public network access is allowed.

Reference: [Azure Functions networking options](#)

Network Security Group Support

Description: Service network traffic respects Network Security Groups rule assignment on its subnets. [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use network security groups (NSG) to restrict or monitor traffic by port, protocol, source IP address, or destination IP address. Create NSG rules to restrict your service's open ports (such as preventing management ports from being

accessed from untrusted networks). Be aware that by default, NSGs deny all inbound traffic but allow traffic from virtual network and Azure Load Balancers.

Reference: [Azure Functions networking options](#)

NS-2: Secure cloud services with network controls

Features

Azure Private Link

Description: Service native IP filtering capability for filtering network traffic (not to be confused with NSG or Azure Firewall). [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Deploy private endpoints for all Azure resources that support the Private Link feature, to establish a private access point for the resources.

Reference: [Azure Functions networking options](#)

Disable Public Network Access

Description: Service supports disabling public network access either through using service-level IP ACL filtering rule (not NSG or Azure Firewall) or using a 'Disable Public Network Access' toggle switch. [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Azure Functions can be configured with private endpoints, but there is not presently a single toggle for disabling public network access absent configuring private endpoints.

Configuration Guidance: Disable public network access either using the service-level IP ACL filtering rule or a toggling switch for public network access.

Identity management

For more information, see the [Microsoft cloud security benchmark: Identity management](#).

IM-1: Use centralized identity and authentication system

Features

Azure AD Authentication Required for Data Plane Access

Description: Service supports using Azure AD authentication for data plane access.

[Learn more.](#)

[] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Customer-owned endpoints may be configured to require Azure AD authentication requirements. System-provided endpoints for deployment operations and advanced developer tools support Azure AD but by default have the ability to alternatively use publishing credentials. These publishing credentials can be disabled. Some data plane endpoints on the app may be accessed by administrative keys configured in the Functions host, and these are not configurable with Azure AD requirements at this time.

Configuration Guidance: Use Azure Active Directory (Azure AD) as the default authentication method to control your data plane access.

Reference: [Configure deployment credentials - disable basic authentication](#)

Local Authentication Methods for Data Plane Access

Description: Local authentications methods supported for data plane access, such as a local username and password. [Learn more.](#)

[] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Feature notes: Deployment credentials are created by default, but they can be disabled. Some operations exposed by the application runtime may be performed using an administrative key, which cannot presently be disabled. This key can be stored in Azure Key Vault, and it can be regenerated at any time. Avoid the usage of local authentication methods or accounts, these should be disabled wherever possible. Instead use Azure AD to authenticate where possible.

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Reference: [Disable basic authentication](#)

IM-3: Manage application identities securely and automatically

Features

Managed Identities

Description: Data plane actions support authentication using managed identities. [Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure managed identities instead of service principals when possible, which can authenticate to Azure services and resources that support Azure Active Directory (Azure AD) authentication. Managed identity credentials are fully managed, rotated, and protected by the platform, avoiding hard-coded credentials in source code or configuration files.

Reference: [How to use managed identities for App Service and Azure Functions](#)

Service Principals

Description: Data plane supports authentication using service principals. [Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: There is no current Microsoft guidance for this feature configuration. Please review and determine if your organization wants to configure this security feature.

Microsoft Defender for Cloud monitoring

Azure Policy built-in definitions - Microsoft.Web:

[\[+\] Expand table](#)

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
App Service apps should use managed identity ↗	Use a managed identity for enhanced authentication security	AuditIfNotExists, Disabled	3.0.0 ↗

IM-7: Restrict resource access based on conditions

Features

Conditional Access for Data Plane

Description: Data plane access can be controlled using Azure AD Conditional Access Policies. [Learn more](#).

[\[+\] Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: For data plane endpoints which are not defined by the application, conditional access would need to be configured against Azure Service Management.

Configuration Guidance: Define the applicable conditions and criteria for Azure Active Directory (Azure AD) conditional access in the workload. Consider common use cases such as blocking or granting access from specific locations, blocking risky sign-in behavior, or requiring organization-managed devices for specific applications.

IM-8: Restrict the exposure of credential and secrets

Features

Service Credential and Secrets Support Integration and Storage in Azure Key Vault

Description: Data plane supports native use of Azure Key Vault for credential and secrets store. [Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Ensure that secrets and credentials are stored in secure locations such as Azure Key Vault, instead of embedding them into code or configuration files.

Reference: [Use Key Vault references for App Service and Azure Functions](#)

Privileged access

For more information, see the [Microsoft cloud security benchmark: Privileged access](#).

PA-1: Separate and limit highly privileged/administrative users

Features

Local Admin Accounts

Description: Service has the concept of a local administrative account. [Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

PA-7: Follow just enough administration (least privilege) principle

Features

Azure RBAC for Data Plane

Description: Azure Role-Based Access Control (Azure RBAC) can be used to manage access to service's data plane actions. [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: The only data-plane actions which can leverage Azure RBAC are the Kudu/SCM/deployment endpoints. These require permission over the `Microsoft.Web/sites/publish/Action` operation. Endpoints exposed by the customer application itself are not covered by Azure RBAC.

Configuration Guidance: Use Azure role-based access control (Azure RBAC) to manage Azure resource access through built-in role assignments. Azure RBAC roles can be assigned to users, groups, service principals, and managed identities.

Reference: [RBAC permissions required to access Kudu](#)

PA-8: Determine access process for cloud provider support

Features

Customer Lockbox

Description: Customer Lockbox can be used for Microsoft support access. [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: In support scenarios where Microsoft needs to access your data, use Customer Lockbox to review, then approve or reject each of Microsoft's data access requests.

Data protection

For more information, see the [Microsoft cloud security benchmark: Data protection](#).

DP-2: Monitor anomalies and threats targeting sensitive data

Features

Data Leakage/Loss Prevention

Description: Service supports DLP solution to monitor sensitive data movement (in customer's content). [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

DP-3: Encrypt sensitive data in transit

Features

Data in Transit Encryption

Description: Service supports data in-transit encryption for data plane. [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Function apps are created by default to support TLS 1.2 as a minimum version, but an app can be configured with a lower version through a configuration setting. HTTPS is not required of incoming requests by default, but this can also be set via a configuration setting, at which point any HTTP request will be automatically redirected to use HTTPS.

Configuration Guidance: Enable secure transfer in services where there is a native data in transit encryption feature built in. Enforce HTTPS on any web applications and services and ensure TLS v1.2 or later is used. Legacy versions such as SSL 3.0, TLS v1.0 should be disabled. For remote management of Virtual Machines, use SSH (for Linux) or RDP/TLS (for Windows) instead of an unencrypted protocol.

Reference: [Add and manage TLS/SSL certificates in Azure App Service](#)

Microsoft Defender for Cloud monitoring

Azure Policy built-in definitions - Microsoft.Web:

[+] Expand table

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
App Service apps should only be accessible over HTTPS ↗	Use of HTTPS ensures server/service authentication and protects data in transit from network layer eavesdropping attacks.	Audit, Disabled, Deny	4.0.0 ↗

DP-4: Enable data at rest encryption by default

Features

Data at Rest Encryption Using Platform Keys

Description: Data at-rest encryption using platform keys is supported, any customer content at rest is encrypted with these Microsoft managed keys. [Learn more](#).

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

DP-5: Use customer-managed key option in data at rest encryption when required

Features

Data at Rest Encryption Using CMK

Description: Data at-rest encryption using customer-managed keys is supported for customer content stored by the service. [Learn more](#).

[\[+\] Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Azure Functions does not directly support this feature, but an application can be configured to leverage services which do, in place of any possible data storage in Functions. Azure Files may be mounted as the file system, all App Settings, including secrets, may be stored in Azure Key Vault, and deployment options such as run-from-package may pull content from Azure Blob storage.

Configuration Guidance: If required for regulatory compliance, define the use case and service scope where encryption using customer-managed keys are needed. Enable and implement data at rest encryption using customer-managed key for those services.

Reference: [Encrypt your application data at rest using customer-managed keys](#)

DP-6: Use a secure key management process

Features

Key Management in Azure Key Vault

Description: The service supports Azure Key Vault integration for any customer keys, secrets, or certificates. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure Key Vault to create and control the life cycle of your encryption keys, including key generation, distribution, and storage. Rotate and revoke your keys in Azure Key Vault and your service based on a defined schedule or when there is a key retirement or compromise. When there is a need to use customer-managed key (CMK) in the workload, service, or application level, ensure you follow the best practices for key management: Use a key hierarchy to generate a separate data encryption key (DEK) with your key encryption key (KEK) in your key vault. Ensure keys are registered with Azure Key Vault and referenced via key IDs from the service or application. If you need to bring your own key (BYOK) to the service (such as importing HSM-protected keys from your on-premises HSMs into Azure Key Vault), follow recommended guidelines to perform initial key generation and key transfer.

Reference: [Use Key Vault references for App Service and Azure Functions](#)

DP-7: Use a secure certificate management process

Features

Certificate Management in Azure Key Vault

Description: The service supports Azure Key Vault integration for any customer certificates. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure Key Vault to create and control the certificate lifecycle, including creation, importing, rotation, revocation, storage, and purging of the certificate. Ensure the certificate generation follows defined standards without using any insecure properties, such as: insufficient key size, overly long validity period, insecure

cryptography. Setup automatic rotation of the certificate in Azure Key Vault and the Azure service (if supported) based on a defined schedule or when there is a certificate expiration. If automatic rotation is not supported in the application, ensure they are still rotated using manual methods in Azure Key Vault and the application.

Reference: [Add a TLS/SSL certificate in Azure App Service](#)

Asset management

For more information, see the [Microsoft cloud security benchmark: Asset management](#).

AM-2: Use only approved services

Features

Azure Policy Support

Description: Service configurations can be monitored and enforced via Azure Policy.
[Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Microsoft Defender for Cloud to configure Azure Policy to audit and enforce configurations of your Azure resources. Use Azure Monitor to create alerts when there is a configuration deviation detected on the resources. Use Azure Policy [deny] and [deploy if not exists] effects to enforce secure configuration across Azure resources.

Logging and threat detection

For more information, see the [Microsoft cloud security benchmark: Logging and threat detection](#).

LT-1: Enable threat detection capabilities

Features

Microsoft Defender for Service / Product Offering

Description: Service has an offering-specific Microsoft Defender solution to monitor and alert on security issues. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Defender for App Service includes Azure Functions. If this solution is enabled, function apps under the enablement scope will be included.

Configuration Guidance: Use Azure Active Directory (Azure AD) as the default authentication method to control your management plane access. When you get an alert from Microsoft Defender for Key Vault, investigate and respond to the alert.

Reference: [Defender for App Service](#)

LT-4: Enable logging for security investigation

Features

Azure Resource Logs

Description: Service produces resource logs that can provide enhanced service-specific metrics and logging. The customer can configure these resource logs and send them to their own data sink like a storage account or log analytics workspace. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Enable resource logs for the service. For example, Key Vault supports additional resource logs for actions that get a secret from a key vault or and Azure SQL has resource logs that track requests to a database. The content of resource logs varies by the Azure service and resource type.

Backup and recovery

For more information, see the [Microsoft cloud security benchmark: Backup and recovery](#).

BR-1: Ensure regular automated backups

Features

Azure Backup

Description: The service can be backed up by the Azure Backup service. [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Feature notes: A feature for backing up an application is available if hosted on a Standard, Premium, or Isolated App Service plan. This feature does not leverage Azure Backup and does not include event sources or externally linked storage. See [/azure/app-service/manage-backup](#) for more details.

Configuration Guidance: This feature is not supported to secure this service.

Service Native Backup Capability

Description: Service supports its own native backup capability (if not using Azure Backup). [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: A backup feature is available to apps running on Standard, Premium, and Isolated App Service plans. This does not include backing up event sources or externally provided storage.

Configuration Guidance: There is no current Microsoft guidance for this feature configuration. Please review and determine if your organization wants to configure this security feature.

Reference: [Back up and restore your app in Azure App Service](#)

Next steps

- See the [Microsoft cloud security benchmark overview](#)
- Learn more about [Azure security baselines](#)

Reliability in Azure Functions

Article • 09/11/2024

This article describes reliability support in [Azure Functions](#), and covers both intra-regional resiliency with [availability zones](#) and [cross-region recovery and business continuity](#). For a more detailed overview of reliability principles in Azure, see [Azure reliability](#).

Availability zone support for Azure Functions is available on both Premium (Elastic Premium) and Dedicated (App Service) plans. This article focuses on zone redundancy support for Premium plans. For zone redundancy on Dedicated plans, see [Migrate App Service to availability zone support](#).

Availability zone support

Azure availability zones are at least three physically separate groups of datacenters within each Azure region. Datacenters within each zone are equipped with independent power, cooling, and networking infrastructure. In the case of a local zone failure, availability zones are designed so that if the one zone is affected, regional services, capacity, and high availability are supported by the remaining two zones.

Failures can range from software and hardware failures to events such as earthquakes, floods, and fires. Tolerance to failures is achieved with redundancy and logical isolation of Azure services. For more detailed information on availability zones in Azure, see [Regions and availability zones](#).

Azure availability zones-enabled services are designed to provide the right level of reliability and flexibility. They can be configured in two ways. They can be either zone redundant, with automatic replication across zones, or zonal, with instances pinned to a specific zone. You can also combine these approaches. For more information on zonal vs. zone-redundant architecture, see [Recommendations for using availability zones and regions](#).

Azure Functions supports a [zone-redundant deployment](#).

When you configure Functions as zone redundant, the platform automatically spreads the function app instances across three zones in the selected region.

Instance spreading with a zone-redundant deployment is determined inside the following rules, even as the app scales in and out:

- The minimum function app instance count is three.

- When you specify a capacity larger than three, the instances are spread evenly only when the capacity is a multiple of 3.
- For a capacity value more than $3 \times N$, extra instances are spread across the remaining one or two zones.

Important

Azure Functions can run on the Azure App Service platform. In the App Service platform, plans that host Premium plan function apps are referred to as Elastic Premium plans, with SKU names like EP1. If you choose to run your function app on a Premium plan, make sure to create a plan with an SKU name that starts with "E", such as EP1. App Service plan SKU names that start with "P", such as P1V2 (Premium V2 Small plan), are actually [Dedicated hosting plans](#). Because they are Dedicated and not Elastic Premium, plans with SKU names starting with "P" won't scale dynamically and may increase your costs.

Regional availability

Zone-redundant Premium plans are available in the following regions:

[\[+\] Expand table](#)

Americas	Europe	Middle East	Africa	Asia Pacific
Brazil South	France Central	Israel Central	South Africa North	Australia East
Canada Central	Germany West Central	Qatar Central		Central India
Central US	Italy North	UAE North		China North 3
East US	North Europe			East Asia
East US 2	Norway East			Japan East
South Central US	Sweden Central			Southeast Asia
West US 2	Switzerland North			
West US 3	UK South			
	West Europe			

Prerequisites

Availability zone support is a property of the Premium plan. The following are the current requirements/limitations for enabling availability zones:

- You can only enable availability zones when creating a Premium plan for your function app. You can't convert an existing Premium plan to use availability zones.
- You must use a [zone redundant storage account \(ZRS\)](#) for your function app's [storage account](#). If you use a different type of storage account, Functions can show unexpected behavior during a zonal outage.
- Both Windows and Linux are supported.
- Must be hosted on an [Elastic Premium](#) or Dedicated hosting plan. To learn how to use zone redundancy with a Dedicated plan, see [Migrate App Service to availability zone support](#).
 - Availability zone support isn't currently available for function apps on [Consumption](#) plans.
- Function apps hosted on a Premium plan must have a minimum [always ready instances](#) count of three.
 - The platform enforces this minimum count behind the scenes if you specify an instance count fewer than three.
- If you aren't using Premium plan or a scale unit that supports availability zones, are in an unsupported region, or are unsure, see the [migration guidance](#).

Pricing

There's no extra cost associated with enabling availability zones. Pricing for a zone redundant Premium App Service plan is the same as a single zone Premium plan. For each App Service plan you use, you're charged based on the SKU you choose, the capacity you specify, and any instances you scale to based on your autoscale criteria. If you enable availability zones but specify a capacity less than three for an App Service plan, the platform enforces a minimum instance count of three for that App Service plan and charges you for those three instances.

Create a zone-redundant Premium plan and function app

There are currently two ways to deploy a zone-redundant Premium plan and function app. You can use either the [Azure portal](#) or an ARM template.

Azure portal

1. In the Azure portal, go to the [Create Function App](#) page. For more information about creating a function app in the portal, see [Create a function](#)

app.

2. Select **Functions Premium** and then select the **Select** button. This article describes how to create a zone redundant app in a Premium plan. Zone redundancy isn't currently available in Consumption plans. For information on zone redundancy on app service plans, see [Reliability in Azure App Service](#).
3. On the **Create Function App (Functions Premium)** page, on the **Basics** tab, enter the settings for your function app. Pay special attention to the settings in the following table (also highlighted in the following screenshot), which have specific requirements for zone redundancy.

 [Expand table](#)

Setting	Suggested value	Notes for zone redundancy
Region	Your preferred supported region	The region under which the new function app is created. You must pick a region that supports availability zones. See the region availability list .
Pricing plan	One of the Elastic Premium plans. For more information, see Available instance SKUs .	This article describes how to create a zone redundant app in a Premium plan. Zone redundancy isn't currently available in Consumption plans. For information on zone redundancy on App Service plans, see Reliability in Azure App Service .
Zone redundancy	Enabled	This setting specifies whether your app is zone redundant. You won't be able to select <code>Enabled</code> unless you have chosen a region that supports zone redundancy, as described previously.

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ <subscription name> ▾

Resource Group * ⓘ (New) myResourceGroup ▾
Create new

Instance Details

Function App name * Function App name .azurewebsites.net

Do you want to deploy code or container image? * Code Container Image

Runtime stack * .NET ▾

Version * 8 (LTS), isolated worker model ▾

Region * West US 3 ▾

i Not finding your App Service Plan? Try a different region or select your App Service Environment.

Operating System * Linux Windows

Environment details

Windows Plan (West US 3) * ⓘ (New) ASP-myResourceGroup-b0e0 ▾
Create new

Pricing plan Elastic Premium EP1 (210 total ACU, 3.5 GB memory, 1 vCPU) ▾

Zone redundancy

An App Service plan can be deployed as a zone redundant service in the regions that support it. This is a deployment time only decision. You can't make an App Service plan zone redundant after it has been deployed [Learn more](#)

Zone redundancy

- Enabled:** Your App Service plan and the apps in it will be zone redundant. The minimum App Service plan instance count will be three.
- Disabled:** Your App Service Plan and the apps in it will not be zone redundant. The minimum App Service plan instance count will be one.

4. On the **Storage** tab, enter the settings for your function app storage account. Pay special attention to the setting in the following table, which has specific requirements for zone redundancy.

i Expand table

Setting	Suggested value	Notes for zone redundancy
Storage account	A zone-redundant storage account	As described in the prerequisites section, we strongly recommend using a zone-redundant storage account for your zone-redundant function app.

5. For the rest of the function app creation process, create your function app as normal. There are no settings in the rest of the creation process that affect zone redundancy.

After the zone-redundant plan is created and deployed, any function app hosted on your new plan is considered zone-redundant.

Availability zone migration

Azure Function Apps currently doesn't support in-place migration of existing function apps instances. For information on how to migrate the public multitenant Premium plan from non-availability zone to availability zone support, see [Migrate App Service to availability zone support](#).

Zone down experience

All available function app instances of zone-redundant function apps are enabled and processing events. When a zone goes down, Functions detect lost instances and automatically attempts to find new replacement instances if needed. Elastic scale behavior still applies. However, in a zone-down scenario there's no guarantee that requests for additional instances can succeed, since back-filling lost instances occurs on a best-effort basis. Applications that are deployed in an availability zone enabled Premium plan continue to run even when other zones in the same region suffer an outage. However, it's possible that non-runtime behaviors could still be impacted from an outage in other availability zones. These impacted behaviors can include Premium plan scaling, application creation, application configuration, and application publishing. Zone redundancy for Premium plans only guarantees continued uptime for deployed applications.

When Functions allocates instances to a zone redundant Premium plan, it uses best effort zone balancing offered by the underlying Azure Virtual Machine Scale Sets. A Premium plan is considered balanced when each zone has either the same number of VMs (± 1 VM) in all of the other zones used by the Premium plan.

Cross-region disaster recovery and business continuity

Disaster recovery (DR) is about recovering from high-impact events, such as natural disasters or failed deployments that result in downtime and data loss. Regardless of the

cause, the best remedy for a disaster is a well-defined and tested DR plan and an application design that actively supports DR. Before you begin to think about creating your disaster recovery plan, see [Recommendations for designing a disaster recovery strategy](#).

When it comes to DR, Microsoft uses the [shared responsibility model](#). In a shared responsibility model, Microsoft ensures that the baseline infrastructure and platform services are available. At the same time, many Azure services don't automatically replicate data or fall back from a failed region to cross-replicate to another enabled region. For those services, you are responsible for setting up a disaster recovery plan that works for your workload. Most services that run on Azure platform as a service (PaaS) offerings provide features and guidance to support DR and you can use [service-specific features to support fast recovery](#) to help develop your DR plan.

This section explains some of the strategies that you can use to deploy Functions to allow for disaster recovery.

For disaster recovery for Durable Functions, see [Disaster recovery and geo-distribution in Azure Durable Functions](#).

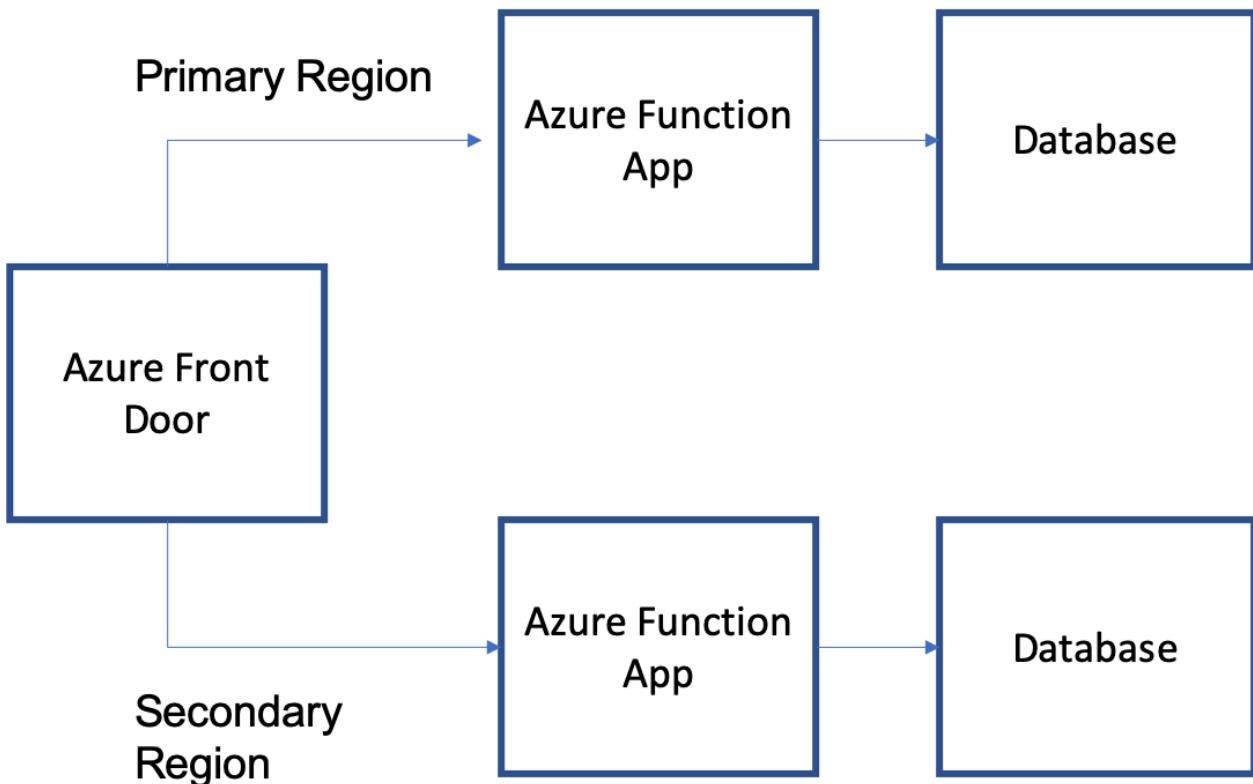
Multi-region disaster recovery

Because there is no built-in redundancy available, functions run in a function app in a specific Azure region. To avoid loss of execution during outages, you can redundantly deploy the same functions to function apps in multiple regions. To learn more about multi-region deployments, see the guidance in [Highly available multi-region web application](#).

When you run the same function code in multiple regions, there are two patterns to consider, [active-active](#) and [active-passive](#).

Active-active pattern for HTTP trigger functions

With an active-active pattern, functions in both regions are actively running and processing events, either in a duplicate manner or in rotation. It's recommended that you use an active-active pattern in combination with [Azure Front Door](#) for your critical HTTP triggered functions, which can route and round-robin HTTP requests between functions running in multiple regions. Front door can also periodically check the health of each endpoint. When a function in one region stops responding to health checks, Azure Front Door takes it out of rotation, and only forwards traffic to the remaining healthy functions.



ⓘ Important

Although, it's highly recommended that you use the [active-passive pattern](#) for non-HTTPS trigger functions. You can create active-active deployments for non-HTTP triggered functions. However, you need to consider how the two active regions interact or coordinate with one another. When you deploy the same function app to two regions with each triggering on the same Service Bus queue, they would act as competing consumers on de-queueing that queue. While this means each message is only being processed by either one of the instances, it also means there's still a single point of failure on the single Service Bus instance.

You could instead deploy two Service Bus queues, with one in a primary region, one in a secondary region. In this case, you could have two function apps, with each pointed to the Service Bus queue active in their region. The challenge with this topology is how the queue messages are distributed between the two regions. Often, this means that each publisher attempts to publish a message to *both* regions, and each message is processed by both active function apps. While this creates the desired active/active pattern, it also creates other challenges around duplication of compute and when or how data is consolidated.

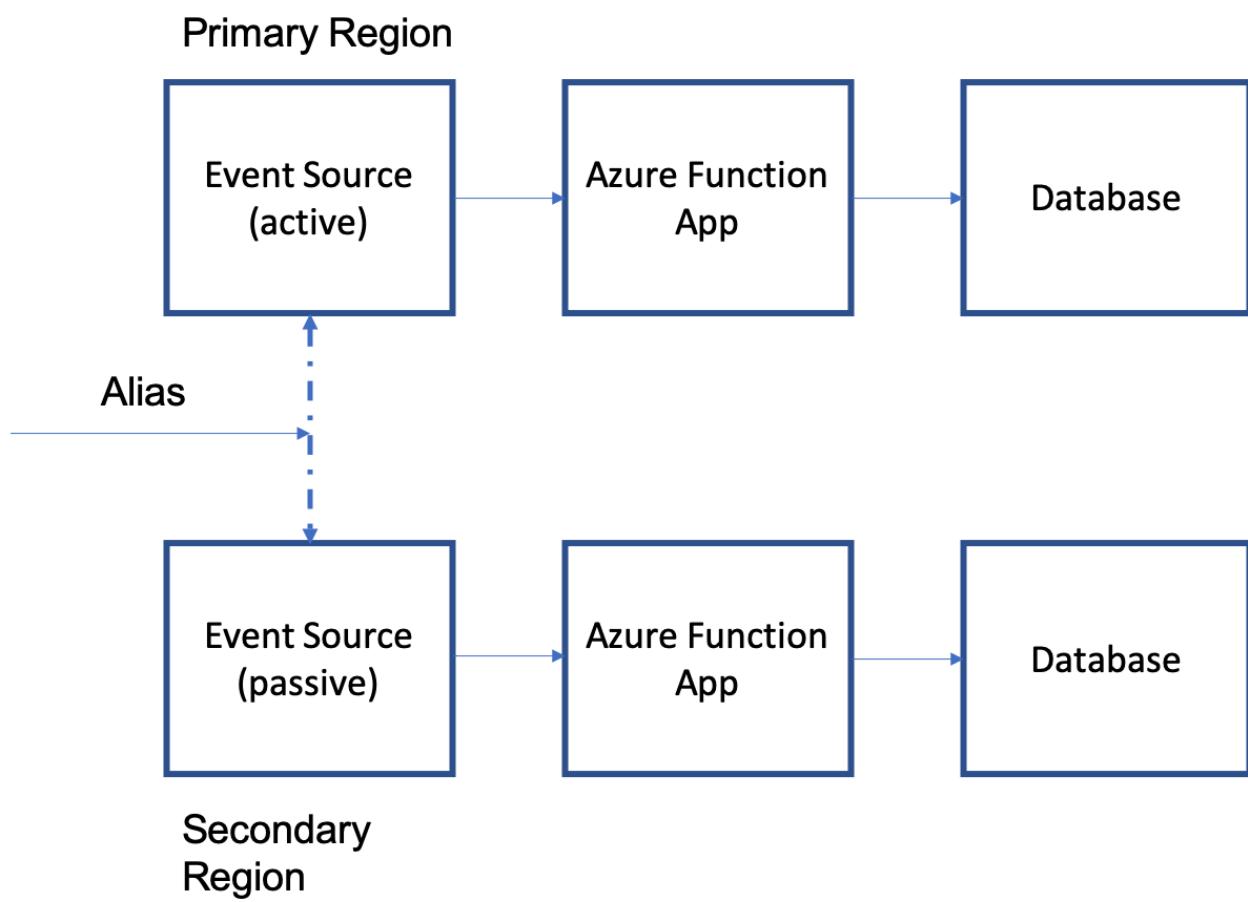
Active-passive pattern for non-HTTPS trigger functions

It's recommended that you use active-passive pattern for your event-driven, non-HTTP triggered functions, such as Service Bus and Event Hubs triggered functions.

To create redundancy for non-HTTP trigger functions, use an active-passive pattern. With an active-passive pattern, functions run actively in the region that's receiving events; while the same functions in a second region remain idle. The active-passive pattern provides a way for only a single function to process each message while providing a mechanism to fail over to the secondary region in a disaster. Function apps work with the failover behaviors of the partner services, such as [Azure Service Bus geo-recovery](#) and [Azure Event Hubs geo-recovery](#).

Consider an example topology using an Azure Event Hubs trigger. In this case, the active/passive pattern requires involve the following components:

- Azure Event Hubs deployed to both a primary and secondary region.
- [Geo-disaster enabled](#) to pair the primary and secondary event hubs. This also creates an *alias* you can use to connect to event hubs and switch from primary to secondary without changing the connection info.
- Function apps are deployed to both the primary and secondary (failover) region, with the app in the secondary region essentially being idle because messages aren't being sent there.
- Function app triggers on the *direct* (non-alias) connection string for its respective event hub.
- Publishers to the event hub should publish to the alias connection string.



Before failover, publishers sending to the shared alias route to the primary event hub. The primary function app is listening exclusively to the primary event hub. The secondary function app is passive and idle. As soon as failover is initiated, publishers sending to the shared alias are routed to the secondary event hub. The secondary function app now becomes active and starts triggering automatically. Effective failover to a secondary region can be driven entirely from the event hub, with the functions becoming active only when the respective event hub is active.

Read more on information and considerations for failover with [Service Bus](#) and [Event Hubs](#).

Next steps

- Disaster recovery and geo-distribution in Azure Durable Functions
- Create Azure Front Door
- Event Hubs failover considerations
- Azure Architecture Center's guide on availability zones
- Reliability in Azure

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Migrate your function app to a zone-redundant plan

Article • 01/18/2023

Availability zones support for Azure Functions is available on [Premium \(Elastic Premium\)](#) and [Dedicated \(App Service\)](#) plans. A zone-redundant function app plan automatically balances its instances between availability zones for higher availability. This article describes how to migrate to the public multi-tenant Premium plan with availability zone support. For migration to zone redundancy on Dedicated plans, refer [here](#).

Downtime requirements

Downtime will be dependent on how you decide to carry out the migration. Since you can't convert pre-existing Premium plans to use availability zones, migration will consist of a side-by-side deployment where you'll create new Premium plans. Downtime will depend on how you choose to redirect traffic from your old to your new availability zone enabled function app. For example, for HTTP based functions if you're using an [Application Gateway](#), a [custom domain](#), or [Azure Front Door](#), downtime will be dependent on the time it takes to update those respective services with your new app's information. Alternatively, you can route traffic to multiple apps at the same time using a service such as [Azure Traffic Manager](#) and only fully cutover to your new availability zone enabled apps when everything is deployed and fully tested. You can also [write defensive functions](#) to ensure messages are not lost during the migration for non-HTTP functions.

Migration guidance: Redeployment

If you want your function app to use availability zones, redeploy your app into a newly created availability zone enabled Premium function app plan.

How to redeploy

The following steps describe how to enable availability zones.

1. If you're already using the Premium SKU and are in one of the [supported regions](#), you can move on to the next step. Otherwise, you should create a new resource group in one of the supported regions.

2. Create a Premium plan in one of the supported regions and the resource group.
Ensure the [new Premium plan has zone redundancy enabled](#).
3. Create and deploy your function apps into the new Premium plan using your desired [deployment method](#).
4. After testing and enabling the new function apps, you can optionally disable or delete your previous non-availability zone apps.

Next steps

[Learn about the Azure Functions Premium plan](#)

[Learn about Azure Functions support for availability zone redundancy](#)

[ARM Quickstart Templates](#)

[Azure Functions geo-disaster recovery](#)

Azure Functions diagnostics overview

Article • 11/03/2021

When you're running a function app, you want to be prepared for any issues that may arise, from 4xx errors to trigger failures. Azure Functions diagnostics is an intelligent and interactive experience to help you troubleshoot your function app with no configuration or extra cost. When you do run into issues with your function app, Azure Functions diagnostics points out what's wrong. It guides you to the right information to more easily and quickly troubleshoot and resolve the issue. This article shows you the basics of how to use Azure Functions diagnostics to more quickly diagnose and solve common function app issues.

Start Azure Functions diagnostics

To start Azure Functions diagnostics:

1. Navigate to your function app in the [Azure portal](#).
2. Select **Diagnose and solve problems** to open Azure Functions diagnostics.
3. Choose a category that best describes the issue of your function app by using the keywords in the homepage tile. You can also type a keyword that best describes your issue in the search bar. For example, you could type `execution` to see a list of diagnostic reports related to your function app execution and open them directly from the homepage.

The screenshot shows the Azure Functions Diagnostics interface for a function app named 'myFunctionApp-dma'. The left sidebar has a red box around the 'Diagnose and solve problems' section. The main area has a search bar with 'execution' typed in, and a list of five results:

- Function App Health Check
- Function Compilation Error (.csx)
- Function Execution Performance
- Function Executions and Errors
- Timer Trigger Issue Analysis

Below this is a section titled 'Azure Functions Diagnostics' with a purple header 'Availability and Performance'. It contains a brief description and a list of keywords: Downtime, 5xx Errors, 4xx Errors, CPU.

Use the Interactive interface

Once you select a homepage category that best aligns with your function app's problem, Azure Functions diagnostics' interactive interface, named Genie, can guide you through diagnosing and solving problem of your app. You can use the tile shortcuts provided by Genie to view the full diagnostic report of the problem category that you're interested in. The tile shortcuts provide you a direct way of accessing your diagnostic metrics.

The screenshot shows the 'Availability and Performance' section of the Genie interface. It includes a welcome message: 'Hello! Welcome to Azure Functions Diagnostics! My name is Genie and I'm here to help you diagnose and solve problems.' Below this is a message: 'Here are some issues related to Availability and Performance that I can help with. Please select the tile that best describes your issue.' A grid of tiles provides links to various diagnostic tools:

AlwaysOn Check	Application Crashes	Application Insights Logging Sampling Check	Check RunFromPackage Logs	Function App Down or Reporting Errors	Function App Settings Check
Function Cold Start	Function Configuration Checks	Function Execution Performance	High CPU Analysis	HTTP 4xx Errors	Memory Analysis
Messaging Function Trigger Failure	RunOnStartup Check	TCP Connections	Timer Trigger Issue Analysis	Web App Restarted	

After selecting a tile, you can see a list of topics related to the issue described in the tile. These topics provide snippets of notable information from the full report. Select any of these topics to investigate the issues further. Also, you can select **View Full Report** to explore all the topics on a single page.

The screenshot shows the Azure Functions Diagnostics interface. At the top, there are two tabs: "Home" and "Availability and Performance" (which is currently selected). Below the tabs, a message from "Genie" says: "Hello! Welcome to Azure Functions Diagnostics! My name is Genie and I'm here to help you diagnose and solve problems." Another message below it says: "Here are some issues related to Availability and Performance that I can help with. Please select the tile that best describes your issue." In the bottom right corner of this area, there is a blue button labeled "I am interested in Function App Down or Reporting Errors". The main content area is titled "Function App Down or Reporting Errors" and contains five items, each with a status icon and a link to "View Full Report":

- Bad Async Function Pattern (green checkmark)
- Check RunFromPackage Logs (green checkmark)
- Function App General Information (blue info icon)
- Function App Offline History (green checkmark)
- Function App Settings Check (orange warning icon)

View a diagnostic report

After you choose a topic, you can view a diagnostic report specific to your function app. Diagnostic reports use status icons to indicate if there are any specific issues with your app. You see detailed description of the issue, recommended actions, related-metrics, and helpful docs. Customized diagnostic reports are generated from a series of checks run on your function app. Diagnostic reports can be a useful tool for pinpointing problems in your function app and guiding you towards resolving the issue.

Find the problem code

For script-based functions, you can use **Function Execution and Errors** under **Function App Down or Reporting Errors** to narrow down on the line of code causing exceptions or errors. You can use this tool for getting to the root cause and fixing issues from a specific line of code. This option isn't available for precompiled C# and Java functions.

Function Executions and Errors

Detects execution statistics and errors for every function inside the function app.

[Send Feedback](#)[Copy Report](#)

▼ ! Detected function(s) having execution failure rate more than 1%.



Description	Function (by failure rate)	Total Executions	Failure Rate(%)	Top Exception
	TimerTrigger1	16	68.75%	Type : System.ArgumentException Total Count : 7 Message : Parameter cannot be null Parameter name: number

Recommended Action Please review your functions code/config to see which part is causing the error and apply the fixes appropriately.

Monitor [Monitor Azure Functions Using Application Insights](#)

Exception Details

Timestamp : 10/30/2019 7:05:00 PM

Inner Exception Type: System.ArgumentException

Total Occurrences: 7

Latest Exception Message: Parameter cannot be null

Parameter name: number

Full Exception :

System.ArgumentException : Parameter cannot be null

Parameter name: number

at Submission#0.Run(TimerInfo myTimer,ILogger log) at D:\home\site\wwwroot\TimerTrigger1\run.csx : 17

Next steps

You can ask questions or provide feedback on Azure Functions diagnostics at

[UserVoice](#). Include **[Diag]** in the title of your feedback.

[Monitor your function apps](#)

Estimating consumption-based costs

Article • 05/21/2024

This article shows you how to estimate plan costs for the Consumption and Flex Consumption hosting plans.

Azure Functions currently offers four different hosting plans for your function apps, with each plan having its own pricing model:

[+] [Expand table](#)

Plan	Description
Consumption	You're only charged for the time that your function app runs. This plan includes a free grant on a per subscription basis.
Flex Consumption plan	You pay for execution time on the instances on which your functions are running, plus any <i>always ready</i> instances. Instances are dynamically added and removed based on the number of incoming events. Also supports virtual network integration.
Premium	Provides you with the same features and scaling mechanism as the Consumption plan, but with enhanced performance and virtual network integration. Cost is based on your chosen pricing tier. To learn more, see Azure Functions Premium plan .
Dedicated (App Service) (basic tier or higher)	When you need to run in dedicated VMs or in isolation, use custom images, or want to use your excess App Service plan capacity. Uses regular App Service plan billing . Cost is based on your chosen pricing tier.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

You should always choose the plan that best supports the feature, performance, and cost requirements for your function executions. To learn more, see [Azure Functions scale and hosting](#).

This article focuses on Consumption and Flex Consumption plans because in these plans billing depends on active periods of executions inside each instance.

Durable Functions can also run in both of these plans. To learn more about the cost considerations when using Durable Functions, see [Durable Functions billing](#).

Consumption-based costs

The way that consumption-based costs are calculated, including free grants, depends on the specific plan. For the most current cost and grant information, see the [Azure Functions pricing page](#).

Consumption plan

The execution *cost* of a single function execution is measured in *GB-seconds*. Execution cost is calculated by combining its memory usage with its execution time. A function that runs for longer costs more, as does a function that consumes more memory.

Consider a case where the amount of memory used by the function stays constant. In this case, calculating the cost is simple multiplication. For example, say that your function consumed 0.5 GB for 3 seconds. Then the execution cost is $0.5\text{GB} * 3\text{s} = 1.5 \text{ GB-seconds}$.

Since memory usage changes over time, the calculation is essentially the integral of memory usage over time. The system does this calculation by sampling the memory usage of the process (along with child processes) at regular intervals. As mentioned on the [pricing page](#), memory usage is rounded up to the nearest 128-MB bucket. When your process is using 160 MB, you're charged for 256 MB. The calculation takes into account concurrency, which is multiple concurrent function executions in the same process.

ⓘ Note

While CPU usage isn't directly considered in execution cost, it can have an impact on the cost when it affects the execution time of the function.

For an HTTP-triggered function, when an error occurs before your function code begins to execute you aren't charged for an execution. This means that 401 responses from the platform due to API key validation or the App Service Authentication / Authorization feature don't count against your execution cost. Similarly, 5xx status code responses aren't counted when they occur in the platform before your function processes the request. A 5xx response generated by the platform after your function code has started to execute is still counted as an execution, even when the error isn't raised from your function code.

Other related costs

When estimating the overall cost of running your functions in any plan, remember that the Functions runtime uses several other Azure services, which are each billed separately. When you estimate pricing for function apps, any triggers and bindings you have that integrate with other Azure services require you to create and pay for those other services.

For functions running in a Consumption plan, the total cost is the execution cost of your functions, plus the cost of bandwidth and other services.

When estimating the overall costs of your function app and related services, use the [Azure pricing calculator](#).

 Expand table

Related cost	Description
Storage account	Each function app requires that you have an associated General Purpose Azure Storage account , which is billed separately . This account is used internally by the Functions runtime, but you can also use it for Storage triggers and bindings. If you don't have a storage account, one is created for you when the function app is created. To learn more, see Storage account requirements .
Application Insights	Functions relies on Application Insights to provide a high-performance monitoring experience for your function apps. While not required, you should enable Application Insights integration . A free grant of telemetry data is included every month. To learn more, see the Azure Monitor pricing page .
Network bandwidth	You can incur costs for data transfer depending on the direction and scenario of the data movement. To learn more, see Bandwidth pricing details .

Behaviors affecting execution time

The following behaviors of your functions can affect the execution time:

- **Triggers and bindings:** The time taken to read input from and write output to your [function bindings](#) is counted as execution time. For example, when your function uses an output binding to write a message to an Azure storage queue, your execution time includes the time taken to write the message to the queue, which is included in the calculation of the function cost.
- **Asynchronous execution:** The time that your function waits for the results of an async request (`await` in C#) is counted as execution time. The GB-second

calculation is based on the start and end time of the function and the memory usage over that period. What is happening over that time in terms of CPU activity isn't factored into the calculation. You may be able to reduce costs during asynchronous operations by using [Durable Functions](#). You're not billed for time spent at awaits in orchestrator functions.

Viewing cost-related data

In [your invoice](#), you can view the cost-related data of **Total Executions - Functions** and **Execution Time - Functions**, along with the actual billed costs. However, this invoice data is a monthly aggregate for a past invoice period.

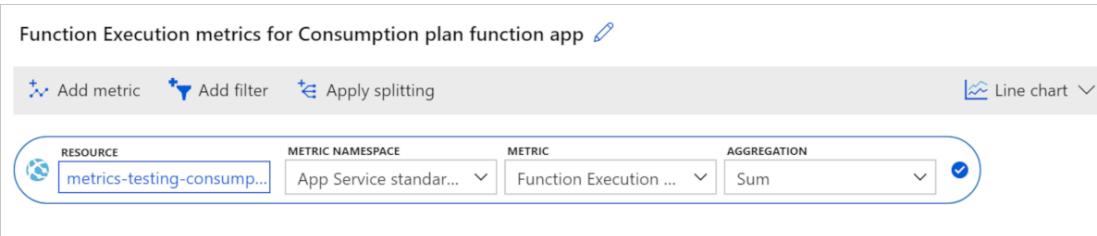
Function app-level metrics

To better understand the cost impact of your functions, you can use Azure Monitor to view cost-related metrics currently being generated by your function apps.

Portal

Use [Azure Monitor metrics explorer](#) to view cost-related data for your Consumption plan function apps in a graphical format.

1. In the [Azure portal](#)  , navigate to your function app.
2. In the left panel, scroll down to **Monitoring** and choose **Metrics**.
3. From **Metric**, choose **Function Execution Count** and **Sum for Aggregation**.
This adds the sum of the execution counts during chosen period to the chart.



4. Select **Add metric** and repeat steps 2-4 to add **Function Execution Units** to the chart.

The resulting chart contains the totals for both execution metrics in the chosen time range, which in this case is two hours.



As the number of execution units is so much greater than the execution count, the chart just shows execution units.

This chart shows a total of 1.11 billion [Function Execution Units](#) consumed in a two-hour period, measured in MB-milliseconds. To convert to GB-seconds, divide by 1024000. In this example, the function app consumed $11100000000 / 1024000 = 1083.98$ GB-seconds. You can take this value and multiply by the current price of execution time on the [Functions pricing page](#), which gives you the cost of these two hours, assuming you've already used any free grants of execution time.

Function-level metrics

Function execution units are a combination of execution time and your memory usage, which makes it a difficult metric for understanding memory usage. Memory data isn't a metric currently available through Azure Monitor. However, if you want to optimize the memory usage of your app, can use the performance counter data collected by Application Insights.

If you haven't already done so, [enable Application Insights in your function app](#). With this integration enabled, you can [query this telemetry data in the portal](#).

You can use either [Azure Monitor metrics explorer](#) in the [Azure portal](#) or REST APIs to get Monitor Metrics data.

Determine memory usage

Under **Monitoring**, select **Logs (Analytics)**, then copy the following telemetry query and paste it into the query window and select **Run**. This query returns the total memory usage at each sampled time.

```
performanceCounters  
| where name == "Private Bytes"  
| project timestamp, name, value
```

The results look like the following example:

[\[...\] Expand table](#)

timestamp [UTC]	name	value
9/12/2019, 1:05:14.947 AM	Private Bytes	209,932,288
9/12/2019, 1:06:14.994 AM	Private Bytes	212,189,184
9/12/2019, 1:06:30.010 AM	Private Bytes	231,714,816
9/12/2019, 1:07:15.040 AM	Private Bytes	210,591,744
9/12/2019, 1:12:16.285 AM	Private Bytes	216,285,184
9/12/2019, 1:12:31.376 AM	Private Bytes	235,806,720

Determine duration

Azure Monitor tracks metrics at the resource level, which for Functions is the function app. Application Insights integration emits metrics on a per-function basis. Here's an example analytics query to get the average duration of a function:

```
customMetrics  
| where name contains "Duration"  
| extend averageDuration = valueSum / valueCount  
| summarize averageDurationMilliseconds=avg(averageDuration) by name
```

[\[...\] Expand table](#)

name	averageDurationMilliseconds
QueueTrigger AvgDurationMs	16.087
QueueTrigger MaxDurationMs	90.249
QueueTrigger MinDurationMs	8.522

Next steps

[Learn more about Monitoring function apps](#)

Serverless REST APIs using Azure Functions

Article • 11/20/2022

Azure Functions is an essential compute service that you use to build serverless REST-based APIs. HTTP triggers expose REST endpoints that can be called by your clients, like browsers, mobile apps, and other backend services. With [native support for routes](#), a single HTTP triggered function can expose a highly functional REST API. Functions also provides its own basic key-based authorization scheme to help limit access only to specific clients. For more information, see [Azure Functions HTTP trigger](#)

In some scenarios, you may need your API to support a more complex set of REST behaviors. For example, you may need to combine multiple HTTP function endpoints into a single API. You might also want to pass requests through to one or more backend REST-based services. Finally, your APIs might require a higher-degree of security that lets you monetize its use.

Today, the recommended approach to build more complex and robust APIs based on your functions is to use the comprehensive API services provided by [Azure API Management](#). API Management uses a policy-based model to let you control routing, security, and OpenAPI integration. It also supports advanced policies like rate limiting monetization. Previous versions of the Functions runtime used the legacy Functions Proxies feature.

Important

Azure Functions proxies is a legacy feature for [versions 1.x through 3.x](#) of the Azure Functions runtime. Support for proxies can be re-enabled in version 4.x for you to successfully upgrade your function apps to the latest runtime version. As soon as possible, you should switch to integrating your function apps with Azure API Management. API Management lets you take advantage of a more complete set of features for defining, securing, managing, and monetizing your Functions-based APIs. For more information, see [API Management integration](#).

To learn how to re-enable proxies support in Functions version 4.x, see [Re-enable proxies in Functions v4.x](#).

Moving from Functions Proxies to API Management

When moving from Functions Proxies to using API Management, you must integrate your function app with an API Management instance, and then configure the API Management instance to behave like the previous proxy. The following section provides links to the relevant articles that help you succeed in using API Management with Azure Functions.

If you have challenges moving from proxies or if Azure API Management doesn't address your specific scenarios, post a request in the [API Management feedback forum](#).

API Management integration

API Management lets you import an existing function app. After import, each HTTP triggered function endpoint becomes an API that you can modify and manage. After import, you can also use API Management to generate an OpenAPI definition file for your APIs. During import, any endpoints with an `admin` authorization level are ignored. For more information about using API Management with Functions, see the following articles:

Article	Description
Expose serverless APIs from HTTP endpoints using Azure API Management	Shows how to create a new API Management instance from an existing function app in the Azure portal. Supports all languages.
Create serverless APIs in Visual Studio using Azure Functions and API Management integration	Shows how to use Visual Studio to create a C# project that uses the OpenAPI extension . The OpenAPI extension lets you define your .NET APIs by applying attributes directly to your C# code.
Quickstart: Create a new Azure API Management service instance by using the Azure portal	Create a new API Management instance in the portal. After you create an API Management instance, you can connect it to your function app. Other non-portal creation methods are supported.
Import an Azure function app as an API in Azure API Management	Shows how to import an existing function app to expose existing HTTP trigger endpoints as a managed API. This article supports both creating a new API and adding the endpoints to an existing managed API.

After you have your function app endpoints exposed by using API Management, the following articles provide general information about how to manage your Functions-based APIs in the API Management instance.

Article	Description
Edit an API	Shows you how to work with an existing API hosted in API Management.
Policies in Azure API Management	In API Management, publishers can change API behavior through configuration using policies. Policies are a collection of statements that are run sequentially on the request or response of an API.
API Management policy reference	Reference that details all supported API Management policies.
API Management policy samples	Helpful collection of samples using API Management policies in key scenarios.

Legacy Functions Proxies

The legacy [Functions Proxies feature](#) also provides a set of basic API functionality for version 3.x and older version of the Functions runtime.

Important

Azure Functions proxies is a legacy feature for [versions 1.x through 3.x](#) of the Azure Functions runtime. Support for proxies can be re-enabled in version 4.x for you to successfully upgrade your function apps to the latest runtime version. As soon as possible, you should switch to integrating your function apps with Azure API Management. API Management lets you take advantage of a more complete set of features for defining, securing, managing, and monetizing your Functions-based APIs. For more information, see [API Management integration](#).

To learn how to re-enable proxies support in Functions version 4.x, see [Re-enable proxies in Functions v4.x](#).

Some basic hints for how to perform equivalent tasks using API Management have been added to the [Functions Proxies article](#). We don't currently have documentation or tools to help you migrate an existing Functions Proxies implementation to API Management.

Next steps

Expose serverless APIs from HTTP endpoints using Azure API Management

Azure Functions networking options

Article • 06/21/2024

This article describes the networking features available across the hosting options for Azure Functions. All the following networking options give you some ability to access resources without using internet-routable addresses or to restrict internet access to a function app.

The [hosting models](#) have different levels of network isolation available. Choosing the correct one helps you meet your network isolation requirements.

[Expand table

Feature	Consumption plan	Flex Consumption plan	Premium plan	Dedicated plan/ASE	Container Apps*
Inbound IP restrictions	Yes	Yes	Yes	Yes	No
Inbound Private Endpoints	No	Yes	Yes	Yes	No
Virtual network integration	No	Yes (Regional)	Yes (Regional)	Yes (Regional and Gateway)	Yes
Virtual network triggers (non-HTTP)	No	Yes	Yes	Yes	Yes
Hybrid connections (Windows only)	No	No	Yes	Yes	No
Outbound IP restrictions	No	Yes	Yes	Yes	No

*For more information, see [Networking in Azure Container Apps environment](#).

Quickstart resources

Use the following resources to quickly get started with Azure Functions networking scenarios. These resources are referenced throughout the article.

- ARM templates, Bicep files, and Terraform templates:
 - [Private HTTP triggered function app ↗](#)
 - [Private Event Hubs triggered function app ↗](#)
- ARM templates only:
 - [Function app with Azure Storage private endpoints ↗](#).
 - [Azure function app with Virtual Network Integration ↗](#).
- Tutorials:
 - [Integrate Azure Functions with an Azure virtual network by using private endpoints](#)
 - [Restrict your storage account to a virtual network.](#)
 - [Control Azure Functions outbound IP with an Azure virtual network NAT gateway.](#)

Inbound networking features

The following features let you filter inbound requests to your function app.

Inbound access restrictions

You can use access restrictions to define a priority-ordered list of IP addresses that are allowed or denied access to your app. The list can include IPv4 and IPv6 addresses, or specific virtual network subnets using [service endpoints](#). When there are one or more entries, an implicit "deny all" exists at the end of the list. IP restrictions work with all function-hosting options.

Access restrictions are available in the [Flex Consumption plan](#), [Elastic Premium](#), [Consumption](#), and [App Service](#).

Note

With network restrictions in place, you can deploy only from within your virtual network, or when you've put the IP address of the machine you're using to access the Azure portal on the Safe Recipients list. However, you can still manage the function using the portal.

To learn more, see [Azure App Service static access restrictions](#).

Private endpoints

[Azure Private Endpoint](#) is a network interface that connects you privately and securely to a service powered by Azure Private Link. Private Endpoint uses a private IP address from your virtual network, effectively bringing the service into your virtual network.

You can use Private Endpoint for your functions hosted in the [Premium](#) and [App Service](#) plans.

If you want to make calls to Private Endpoints, then you must make sure that your DNS lookups resolve to the private endpoint. You can enforce this behavior in one of the following ways:

- Integrate with Azure DNS private zones. When your virtual network doesn't have a custom DNS server, this is done automatically.
- Manage the private endpoint in the DNS server used by your app. To do this you must know the private endpoint address and then point the endpoint you are trying to reach to that address using an A record.
- Configure your own DNS server to forward to [Azure DNS private zones](#).

To learn more, see [using Private Endpoints for Web Apps](#).

To call other services that have a private endpoint connection, such as storage or service bus, be sure to configure your app to make [outbound calls to private endpoints](#). For more details on using private endpoints with the storage account for your function app, visit [restrict your storage account to a virtual network](#).

Service endpoints

Using service endpoints, you can restrict many Azure services to selected virtual network subnets to provide a higher level of security. Regional virtual network integration enables your function app to reach Azure services that are secured with service endpoints. This configuration is supported on all [plans](#) that support virtual network integration. To access a service endpoint-secured service, you must do the following:

1. Configure regional virtual network integration with your function app to connect to a specific subnet.
2. Go to the destination service and configure service endpoints against the integration subnet.

To learn more, see [Virtual network service endpoints](#).

Use Service Endpoints

To restrict access to a specific subnet, create a restriction rule with a **Virtual Network** type. You can then select the subscription, virtual network, and subnet that you want to allow or deny access to.

If service endpoints aren't already enabled with `Microsoft.Web` for the subnet that you selected, they're automatically enabled unless you select the **Ignore missing Microsoft.Web service endpoints** check box. The scenario where you might want to enable service endpoints on the app but not the subnet depends mainly on whether you have the permissions to enable them on the subnet.

If you need someone else to enable service endpoints on the subnet, select the **Ignore missing Microsoft.Web service endpoints** check box. Your app is configured for service endpoints in anticipation of having them enabled later on the subnet.

Add Access Restriction

X

Name ⓘ



Action

Allow Deny

Priority *

Description



Type



Subscription *



Virtual Network *



Subnet *



Selected subnet 'networking-demos-vnet/nat-gw-subnet' does not have service endpoint enabled for Microsoft.Web. Enabling access may take up to 15 minutes to complete.



Ignore missing Microsoft.Web service endpoints

Add rule

You can't use service endpoints to restrict access to apps that run in an App Service Environment. When your app is in an App Service Environment, you can control access to it by applying IP access rules.

To learn how to set up service endpoints, see [Establish Azure Functions private site access](#).

Virtual network integration

Virtual network integration allows your function app to access resources inside a virtual network. Azure Functions supports two kinds of virtual network integration:

- The dedicated compute pricing tiers, which include the Basic, Standard, Premium, Premium v2, and Premium v3.
- The App Service Environment, which deploys directly into your virtual network with dedicated supporting infrastructure and is using the Isolated and Isolated v2 pricing tiers.

The virtual network integration feature is used in Azure App Service dedicated compute pricing tiers. If your app is in an [App Service Environment](#), it's already in a virtual network and doesn't require use of the VNet integration feature to reach resources in the same virtual network. For more information on all the networking features, see [App Service networking features](#).

Virtual network integration gives your app access to resources in your virtual network, but it doesn't grant inbound private access to your app from the virtual network. Private site access refers to making an app accessible only from a private network, such as from within an Azure virtual network. Virtual network integration is used only to make outbound calls from your app into your virtual network. The virtual network integration feature behaves differently when it's used with virtual networks in the same region and with virtual networks in other regions. The virtual network integration feature has two variations:

- **Regional virtual network integration:** When you connect to virtual networks in the same region, you must have a dedicated subnet in the virtual network you're integrating with.
- **Gateway-required virtual network integration:** When you connect directly to virtual networks in other regions or to a classic virtual network in the same region, you need an Azure Virtual Network gateway created in the target virtual network.

The virtual network integration feature:

- Requires a [supported Basic or Standard](#), Premium, Premium v2, Premium v3, or Elastic Premium App Service pricing tier.
- Supports TCP and UDP.
- Works with App Service apps and function apps.

There are some things that virtual network integration doesn't support, like:

- Mounting a drive.

- Windows Server Active Directory domain join.
- NetBIOS.

Gateway-required virtual network integration provides access to resources only in the target virtual network or in networks connected to the target virtual network with peering or VPNs. Gateway-required virtual network integration doesn't enable access to resources available across Azure ExpressRoute connections or work with service endpoints.

No matter which version is used, virtual network integration gives your app access to resources in your virtual network, but it doesn't grant inbound private access to your app from the virtual network. Private site access refers to making your app accessible only from a private network, such as from within an Azure virtual network. Virtual network integration is only for making outbound calls from your app into your virtual network.

Virtual network integration in Azure Functions uses shared infrastructure with App Service web apps. To learn more about the two types of virtual network integration, see:

- [Regional virtual network integration](#)
- [Gateway-required virtual network integration](#)

To learn how to set up virtual network integration, see [Enable virtual network integration](#).

Enable virtual network integration

1. In your function app in the [Azure portal](#), select **Networking**, then under **VNet Integration** select [Click here to configure](#).
2. Select **Add VNet**.

The screenshot shows the 'VNet Configuration' section of the 'VNet Integration' blade. It includes a note about securely accessing resources through Azure VNet, a 'Add VNet' button, and sections for 'VNet Details' and 'VNet Address Space'. The 'VNet Details' section shows 'VNet NAME' and 'LOCATION' both set to 'Not Configured'. The 'VNet Address Space' section shows 'Start Address' and 'End Address' both set to 'Not Configured'.

3. The drop-down list contains all of the Azure Resource Manager virtual networks in your subscription in the same region. Select the virtual network you want to integrate with.

The screenshot shows the 'Add VNet Integration' dialog box. It has fields for 'Subscription' (set to 'Private Test Sub CACHAI') and 'Virtual Network'. A dropdown menu under 'Virtual Network' shows 'Search virtual networks' and 'integration-vnet (East US)'. At the bottom is an 'OK' button.

- The Functions Flex Consumption and Elastic Premium plans only supports regional virtual network integration. If the virtual network is in the same region, either create a new subnet or select an empty, pre-existing subnet.
- To select a virtual network in another region, you must have a virtual network gateway provisioned with point to site enabled. Virtual network integration across regions is only supported for Dedicated plans, but global peerings work with regional virtual network integration.

During the integration, your app is restarted. When integration is finished, you see details on the virtual network you're integrated with. By default, Route All is enabled, and all traffic is routed into your virtual network.

If you wish for only your private traffic ([RFC1918](#) traffic) to be routed, please follow the steps in the [app service documentation](#).

Regional virtual network integration

Using regional virtual network integration enables your app to access:

- Resources in the same virtual network as your app.
- Resources in virtual networks peered to the virtual network your app is integrated with.
- Service endpoint secured services.
- Resources across Azure ExpressRoute connections.
- Resources across peered connections, which include Azure ExpressRoute connections.
- Private endpoints

When you use regional virtual network integration, you can use the following Azure networking features:

- **Network security groups (NSGs)**: You can block outbound traffic with an NSG that's placed on your integration subnet. The inbound rules don't apply because you can't use virtual network integration to provide inbound access to your app.
- **Route tables (UDRs)**: You can place a route table on the integration subnet to send outbound traffic where you want.

Note

When you route all of your outbound traffic into your virtual network, it's subject to the NSGs and UDRs that are applied to your integration subnet. When virtual network integrated, your function app's outbound traffic to public IP addresses is still sent from the addresses that are listed in your app properties, unless you provide routes that direct the traffic elsewhere.

Regional virtual network integration isn't able to use port 25.

For the Flex Consumption plan:

1. Ensure that the `Microsoft.App` Azure resource provider is enabled for your subscription by [following these instructions](#). The subnet delegation required by Flex Consumption apps is `Microsoft.App/environments`.
2. The subnet delegation required by Flex Consumption apps is `Microsoft.App/environments`. This is a change from Elastic Premium and App Service which have a different delegation requirement.
3. You can plan for 40 IP addresses to be used at the most for one function app, even if the app scales beyond 40. For example, if you have fifteen Flex Consumption function apps that will be VNet integrated into the same subnet, you can plan for $15 \times 40 = 600$ IP addresses used at the most. This limit is subject to change, and is not enforced.
4. The subnet can't already be in use for other purposes (like private or service endpoints, or [delegated](#) to any other hosting plan or service). While you can share the same subnet with multiple Flex Consumption apps, the networking resources will be shared across these function apps and this can lead to one function app impacting the performance of others on the same subnet.

There are some limitations with using virtual network:

- The feature is available from Flex Consumption, Elastic Premium, and App Service Premium V2 and Premium V3. It's also available in Standard but only from newer App Service deployments. If you are on an older deployment, you can only use the feature from a Premium V2 App Service plan. If you want to make sure you can use the feature in a Standard App Service plan, create your app in a Premium V3 App Service plan. Those plans are only supported on our newest deployments. You can scale down if you desire after that.
- The integration subnet can be used by only one App Service plan.
- The feature can't be used by Isolated plan apps that are in an App Service Environment.
- The feature requires an unused subnet that's a /28 or larger in an Azure Resource Manager virtual network.
- The app and the virtual network must be in the same region.
- You can't delete a virtual network with an integrated app. Remove the integration before you delete the virtual network.
- You can have up to two regional virtual network integrations per App Service plan. Multiple apps in the same App Service plan can use the same integration subnet.
- You can't change the subscription of an app or a plan while there's an app that's using regional virtual network integration.

Subnets

Virtual network integration depends on a dedicated subnet. When you provision a subnet, the Azure subnet loses five IPs from the start. For the Elastic Premium and App Service plans, one address is used from the integration subnet for each plan instance. When you scale your app to four instances, then four addresses are used. For Flex Consumption this does not apply and instances share IP addresses.

When you scale up or down in size, the required address space is doubled for a short period of time. This affects the real, available supported instances for a given subnet size. The following table shows both the maximum available addresses per CIDR block and the effect this has on horizontal scale:

[+] Expand table

CIDR block size	Max available addresses	Max horizontal scale (instances)*
/28	11	5
/27	27	13
/26	59	29

*Assumes that you need to scale up or down in either size or SKU at some point.

Since subnet size can't be changed after assignment, use a subnet that's large enough to accommodate whatever scale your app might reach. To avoid any issues with subnet capacity for Functions Elastic Premium plans, you should use a /24 with 256 addresses for Windows and a /26 with 64 addresses for Linux. When creating subnets in Azure portal as part of integrating with the virtual network, a minimum size of /24 and /26 is required for Windows and Linux respectively.

When you want your apps in another plan to reach a virtual network that's already connected to by apps in another plan, select a different subnet than the one being used by the pre-existing virtual network integration.

The feature is fully supported for both Windows and Linux apps, including [custom containers](#). All of the behaviors act the same between Windows apps and Linux apps.

Network security groups

You can use network security groups to block inbound and outbound traffic to resources in a virtual network. An app that uses regional virtual network integration can use a [network security group](#) to block outbound traffic to resources in your virtual network or the internet. To block traffic to public addresses, you must have virtual network integration with Route All enabled. The inbound rules in an NSG don't apply to

your app because virtual network integration affects only outbound traffic from your app.

To control inbound traffic to your app, use the Access Restrictions feature. An NSG that's applied to your integration subnet is in effect regardless of any routes applied to your integration subnet. If your function app is virtual network integrated with Route All enabled, and you don't have any routes that affect public address traffic on your integration subnet, all of your outbound traffic is still subject to NSGs assigned to your integration subnet. When Route All isn't enabled, NSGs are only applied to RFC1918 traffic.

Routes

You can use route tables to route outbound traffic from your app to wherever you want. By default, route tables only affect your RFC1918 destination traffic. When Route All is enabled, all of your outbound calls are affected. When [Route All](#) is disabled, only private traffic (RFC1918) is affected by your route tables. Routes that are set on your integration subnet won't affect replies to inbound app requests. Common destinations can include firewall devices or gateways.

If you want to route all outbound traffic on-premises, you can use a route table to send all outbound traffic to your ExpressRoute gateway. If you do route traffic to a gateway, be sure to set routes in the external network to send any replies back.

Border Gateway Protocol (BGP) routes also affect your app traffic. If you have BGP routes from something like an ExpressRoute gateway, your app outbound traffic is affected. By default, BGP routes affect only your RFC1918 destination traffic. When your function app is virtual network integrated with Route All enabled, all outbound traffic can be affected by your BGP routes.

Azure DNS private zones

After your app integrates with your virtual network, it uses the same DNS server that your virtual network is configured with and will work with the Azure DNS private zones linked to the virtual network.

Restrict your storage account to a virtual network

Note

To quickly deploy a function app with private endpoints enabled on the storage account, please refer to the following template: [Function app with Azure Storage private endpoints](#).

When you create a function app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. You can replace this storage account with one that is secured with service endpoints or private endpoints.

This feature is supported for all Windows and Linux virtual network-supported SKUs in the Dedicated (App Service) plan and for the Elastic Premium plans, as well as the Flex Consumption plan. The Consumption plan isn't supported. To learn how to set up a function with a storage account restricted to a private network, see [Restrict your storage account to a virtual network](#).

Use Key Vault references

You can use Azure Key Vault references to use secrets from Azure Key Vault in your Azure Functions application without requiring any code changes. Azure Key Vault is a service that provides centralized secrets management, with full control over access policies and audit history.

If virtual network integration is configured for the app, [Key Vault references](#) may be used to retrieve secrets from a network-restricted vault.

Virtual network triggers (non-HTTP)

Currently, you can use non-HTTP trigger functions from within a virtual network in one of two ways:

- Run your function app in an [Elastic Premium plan](#) and enable virtual network trigger support.
- Run your function app in a Flex Consumption, App Service plan or App Service Environment.

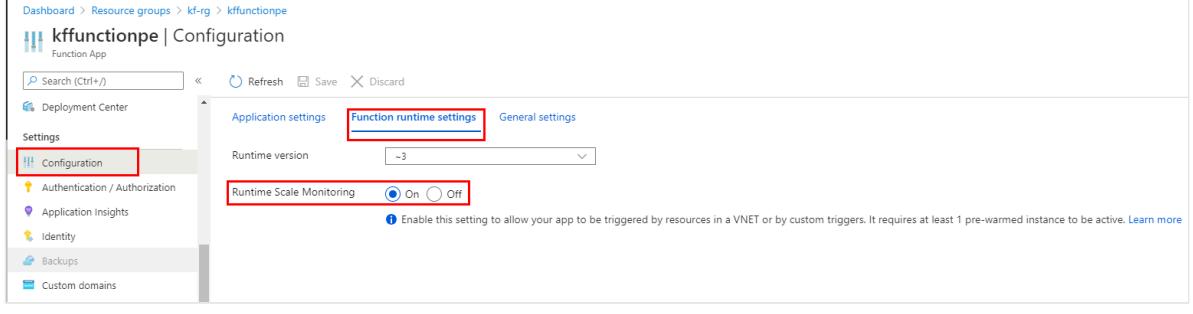
Elastic Premium plan with virtual network triggers

The [Elastic Premium plan](#) lets you create functions that are triggered by services inside a virtual network. These non-HTTP triggers are known as *virtual network triggers*.

By default, virtual network triggers don't cause your function app to scale beyond their pre-warmed instance count. However, certain extensions support virtual network

triggers that cause your function app to scale dynamically. You can enable this *dynamic scale monitoring* in your function app for supported extensions in one of these ways:

Azure portal



1. In the [Azure portal](#), navigate to your function app.
2. Under **Settings** select **Configuration**, then in the **Function runtime settings** tab set **Runtime Scale Monitoring** to **On**.
3. Select **Save** to update the function app configuration and restart the app.

Tip

Enabling the monitoring of virtual network triggers may have an impact on the performance of your application, though this impact is likely to be very small.

Support for dynamic scale monitoring of virtual network triggers isn't available in version 1.x of the Functions runtime.

The extensions in this table support dynamic scale monitoring of virtual network triggers. To get the best scaling performance, you should upgrade to versions that also support [target-based scaling](#).

[Expand table](#)

Extension (minimum version)	Runtime scale monitoring only	With target-based scaling
Microsoft.Azure.WebJobs.Extensions.CosmosDB	> 3.0.5	> 4.1.0
Microsoft.Azure.WebJobs.Extensions.DurableTask	> 2.0.0	n/a
Microsoft.Azure.WebJobs.Extensions.EventHubs	> 4.1.0	> 5.2.0
Microsoft.Azure.WebJobs.Extensions.ServiceBus	> 3.2.0	> 5.9.0

Extension (minimum version)	Runtime scale monitoring only	With target-based scaling
Microsoft.Azure.WebJobs.Extensions.Storage	> 3.0.10	> 5.1.0*

* Queue storage only.

ⓘ Important

When you enable virtual network trigger monitoring, only triggers for these extensions can cause your app to scale dynamically. You can still use triggers from extensions that aren't in this table, but they won't cause scaling beyond their pre-warmed instance count. For a complete list of all trigger and binding extensions, see [Triggers and bindings](#).

App Service plan and App Service Environment with virtual network triggers

When your function app runs in either an App Service plan or an App Service Environment, you can use non-HTTP trigger functions. For your functions to get triggered correctly, you must be connected to a virtual network with access to the resource defined in the trigger connection.

For example, assume you want to configure Azure Cosmos DB to accept traffic only from a virtual network. In this case, you must deploy your function app in an App Service plan that provides virtual network integration with that virtual network. Integration enables a function to be triggered by that Azure Cosmos DB resource.

Hybrid Connections

[Hybrid Connections](#) is a feature of Azure Relay that you can use to access application resources in other networks. It provides access from your app to an application endpoint. You can't use it to access your application. Hybrid Connections is available to functions that run on Windows in all but the Consumption plan.

As used in Azure Functions, each hybrid connection correlates to a single TCP host and port combination. This means that the hybrid connection's endpoint can be on any operating system and any application as long as you're accessing a TCP listening port. The Hybrid Connections feature doesn't know or care what the application protocol is or what you're accessing. It just provides network access.

To learn more, see the [App Service documentation for Hybrid Connections](#). These same configuration steps support Azure Functions.

ⓘ Important

Hybrid Connections is only supported on Windows plans. Linux isn't supported.

Outbound IP restrictions

Outbound IP restrictions are available in a Flex Consumption plan, Elastic Premium plan, App Service plan, or App Service Environment. You can configure outbound restrictions for the virtual network where your App Service Environment is deployed.

When you integrate a function app in an Elastic Premium plan or an App Service plan with a virtual network, the app can still make outbound calls to the internet by default. By integrating your function app with a virtual network with Route All enabled, you force all outbound traffic to be sent into your virtual network, where network security group rules can be used to restrict traffic. For Flex Consumption all traffic is already routed through the virtual network and Route All is not needed.

To learn how to control the outbound IP using a virtual network, see [Tutorial: Control Azure Functions outbound IP with an Azure virtual network NAT gateway](#).

Automation

The following APIs let you programmatically manage regional virtual network integrations:

- **Azure CLI:** Use the [az functionapp vnet-integration](#) commands to add, list, or remove a regional virtual network integration.
- **ARM templates:** Regional virtual network integration can be enabled by using an Azure Resource Manager template. For a full example, see [this Functions quickstart template ↗](#).

Testing considerations

When testing functions in a function app with private endpoints, you must do your testing from within the same virtual network, such as on a virtual machine (VM) in that network. To use the **Code + Test** option in the portal from that VM, you need to add following [CORS origins](#) to your function app:

- `https://functions-next.azure.com`
- `https://functions-staging.azure.com`
- `https://functions.azure.com`
- `https://portal.azure.com`

If you've restricted access to your function app with private endpoints or any other access restriction, you also must add the service tag `AzureCloud` to the allowed list. To update the allowed list:

1. Navigate to your function app and select **Settings > Networking** and then select **Inbound access configuration > Public network access**.
2. Make sure that **Public network access** is set to **Enabled** from **select virtual networks and IP addresses**.
3. **Add a rule** under Site access and rules:
 - a. Select `Service Tag` as the Source settings **Type** and `AzureCloud` as the **Service Tag**.
 - b. Make sure the action is **Allow**, and set your desired name and priority.

Troubleshooting

The feature is easy to set up, but that doesn't mean your experience will be problem free. If you encounter problems accessing your desired endpoint, there are some utilities you can use to test connectivity from the app console. There are two consoles that you can use. One is the Kudu console, and the other is the console in the Azure portal. To reach the Kudu console from your app, go to **Tools > Kudu**. You can also reach the Kudu console at [sitename].scm.azurewebsites.net. After the website loads, go to the **Debug console** tab. To get to the Azure portal-hosted console from your app, go to **Tools > Console**.

Tools

In native Windows apps, the tools `ping`, `nslookup`, and `tracert` won't work through the console because of security constraints (they work in [custom Windows containers](#)). To fill the void, two separate tools are added. To test DNS functionality, we added a tool named `nameresolver.exe`. The syntax is:

Console

```
nameresolver.exe hostname [optional: DNS Server]
```

You can use nameresolver to check the hostnames that your app depends on. This way you can test if you have anything misconfigured with your DNS or perhaps don't have access to your DNS server. You can see the DNS server that your app uses in the console by looking at the environmental variables WEBSITE_DNS_SERVER and WEBSITE_DNS_ALT_SERVER.

ⓘ Note

The nameresolver.exe tool currently doesn't work in custom Windows containers.

You can use the next tool to test for TCP connectivity to a host and port combination. This tool is called **tcpping** and the syntax is:

Console

```
tcpping.exe hostname [optional: port]
```

The **tcpping** utility tells you if you can reach a specific host and port. It can show success only if there's an application listening at the host and port combination, and there's network access from your app to the specified host and port.

Debug access to virtual network-hosted resources

A number of things can prevent your app from reaching a specific host and port. Most of the time it's one of these things:

- **A firewall is in the way.** If you have a firewall in the way, you hit the TCP timeout. The TCP timeout is 21 seconds in this case. Use the **tcpping** tool to test connectivity. TCP timeouts can be caused by many things beyond firewalls, but start there.
- **DNS isn't accessible.** The DNS timeout is 3 seconds per DNS server. If you have two DNS servers, the timeout is 6 seconds. Use nameresolver to see if DNS is working. You can't use nslookup, because that doesn't use the DNS your virtual network is configured with. If inaccessible, you could have a firewall or NSG blocking access to DNS or it could be down.

If those items don't answer your problems, look first for things like:

Regional virtual network integration

- Is your destination a non-RFC1918 address and you don't have **Route All** enabled?
- Is there an NSG blocking egress from your integration subnet?
- If you're going across Azure ExpressRoute or a VPN, is your on-premises gateway configured to route traffic back up to Azure? If you can reach endpoints in your virtual network but not on-premises, check your routes.
- Do you have enough permissions to set delegation on the integration subnet?
During regional virtual network integration configuration, your integration subnet is delegated to Microsoft.Web/serverFarms. The VNet integration UI delegates the subnet to Microsoft.Web/serverFarms automatically. If your account doesn't have sufficient networking permissions to set delegation, you'll need someone who can set attributes on your integration subnet to delegate the subnet. To manually delegate the integration subnet, go to the Azure Virtual Network subnet UI and set the delegation for Microsoft.Web/serverFarms.

Gateway-required virtual network integration

- Is the point-to-site address range in the RFC 1918 ranges (10.0.0.0-10.255.255.255 / 172.16.0.0-172.31.255.255 / 192.168.0.0-192.168.255.255)?
- Does the gateway show as being up in the portal? If your gateway is down, then bring it back up.
- Do certificates show as being in sync, or do you suspect that the network configuration was changed? If your certificates are out of sync or you suspect that a change was made to your virtual network configuration that wasn't synced with your ASPs, select **Sync Network**.
- If you're going across a VPN, is the on-premises gateway configured to route traffic back up to Azure? If you can reach endpoints in your virtual network but not on-premises, check your routes.
- Are you trying to use a coexistence gateway that supports both point to site and ExpressRoute? Coexistence gateways aren't supported with virtual network integration.

Debugging networking issues is a challenge because you can't see what's blocking access to a specific host:port combination. Some causes include:

- You have a firewall up on your host that prevents access to the application port from your point-to-site IP range. Crossing subnets often requires public access.
- Your target host is down.
- Your application is down.
- You had the wrong IP or hostname.
- Your application is listening on a different port than what you expected. You can match your process ID with the listening port by using "netstat -aon" on the endpoint host.

- Your network security groups are configured in such a manner that they prevent access to your application host and port from your point-to-site IP range.

You don't know what address your app actually uses. It could be any address in the integration subnet or point-to-site address range, so you need to allow access from the entire address range.

More debug steps include:

- Connect to a VM in your virtual network and attempt to reach your resource host:port from there. To test for TCP access, use the PowerShell command **Test-NetConnection**. The syntax is:

PowerShell

```
Test-NetConnection hostname [optional: -Port]
```

- Bring up an application on a VM and test access to that host and port from the console from your app by using **tcpping**.

On-premises resources

If your app can't reach a resource on-premises, check if you can reach the resource from your virtual network. Use the **Test-NetConnection** PowerShell command to check for TCP access. If your VM can't reach your on-premises resource, your VPN or ExpressRoute connection might not be configured properly.

If your virtual network-hosted VM can reach your on-premises system but your app can't, the cause is likely one of the following reasons:

- Your routes aren't configured with your subnet or point-to-site address ranges in your on-premises gateway.
- Your network security groups are blocking access for your point-to-site IP range.
- Your on-premises firewalls are blocking traffic from your point-to-site IP range.
- You're trying to reach a non-RFC 1918 address by using the regional virtual network integration feature.

Deleting the App Service plan or web app before disconnecting the VNet integration

If you deleted the web app or the App Service plan without disconnecting the VNet integration first, you will not be able to do any update/delete operations on the virtual network or subnet that was used for the integration with the deleted resource. A subnet

delegation 'Microsoft.Web/serverFarms' will remain assigned to your subnet and will prevent the update/delete operations.

In order to do update/delete the subnet or virtual network again you need to re-create the VNet integration and then disconnect it:

1. Re-create the App Service plan and web app (it is mandatory to use the exact same web app name as before).
2. Navigate to the 'Networking' blade on the web app and configure the VNet integration.
3. After the VNet integration is configured, select the 'Disconnect' button.
4. Delete the App Service plan or web app.
5. Update/Delete the subnet or virtual network.

If you still encounter issues with the VNet integration after following the steps above, please contact Microsoft Support.

Network troubleshooter

You can also use the Network troubleshooter to resolve connection issues. To open the network troubleshooter, go to the app in the Azure portal. Select **Diagnostic and solve problem**, and then search for **Network troubleshooter**.

Connection issues - It checks the status of the virtual network integration, including checking if the Private IP has been assigned to all instances of the plan and the DNS settings. If a custom DNS isn't configured, default Azure DNS is applied. The troubleshooter also checks for common Function app dependencies including connectivity for Azure Storage and other binding dependencies.

Overview

Search for common problems or tools

Ask Genie Refresh Feedback

Network/Connectivity Troubleshooter

Check your network connectivity and troubleshoot network issues

Tell us more about the problem you are experiencing:

Connectivity issues

- ✓ Connection between App Service worker(s) and VNet is healthy
- ✓ Dns setting is healthy ▾
- i App Service's VNet related behaviors will be changed by following App Settings ▾
- i Evaluated network connectivity for common Function App dependencies. ▾
 - ⚠ Network connectivity evaluation is not extensive, you may still experience problems with the App. See explanation at bottom of page.
 - ✓ Network connectivity test to Azure storage endpoint configured in app setting "AzureWebJobsStorage" was successful. ▾
 - ✓ Network connectivity test to the Azure storage endpoint configured in app setting "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING" was successful. ▾
- i Evaluated network connectivity for all Function input/output bindings. ⓘ
 - ⚠ Network connectivity evaluation is not extensive, you may still experience problems with the App. See explanation at bottom of page.
 - ✓ Function "ServiceBusTestTrigger" - all network connectivity tests were successful. ▾

Explanation of the results and recommended next steps

Configuration issues - This troubleshooter checks if your subnet is valid for virtual network Integration.

Network/Connectivity Troubleshooter

Check your network connectivity and troubleshoot network issues

Tell us more about the problem you are experiencing:

Configuration issues

Please select the subnet you want to integrate your app to

Subscription

Contoso

Virtual Network

FunctionNetwork

Subnet

OutboundSubnet

✓ Subnet 'OutboundSubnet' is valid for integration with App 'functionnetworkingdemo1'

🔗 Recommendations

- For setting up VNet integration, please see [Integrate your app with an Azure virtual network](#).
- App **functionnetworkingdemo1** is already integrated to subnet **OutboundSubnet**. If you are facing connectivity issues, please select I'm unable to connect to a resource, such as SQL or Redis or on-prem, in my Virtual Network option.

Subnet/VNet deletion issue - This troubleshooter checks if your subnet has any locks and if it has any unused Service Association Links that might be blocking the deletion of the VNet/subnet.

Next steps

To learn more about networking and Azure Functions:

- Follow the tutorial about getting started with virtual network integration
- Read the Functions networking FAQ
- Learn more about virtual network integration with App Service/Functions
- Learn more about virtual networks in Azure
- Enable more networking features and control with App Service Environments
- Connect to individual on-premises resources without firewall changes by using Hybrid Connections

Feedback

Was this page helpful?



[Provide product feedback ↗](#)

IP addresses in Azure Functions

Article • 06/29/2023

This article explains the following concepts related to IP addresses of function apps:

- Locating the IP addresses currently in use by a function app.
- Conditions that cause function app IP addresses to change.
- Restricting the IP addresses that can access a function app.
- Defining dedicated IP addresses for a function app.

IP addresses are associated with function apps, not with individual functions. Incoming HTTP requests can't use the inbound IP address to call individual functions; they must use the default domain name (`functionappname.azurewebsites.net`) or a custom domain name.

Function app inbound IP address

Each function app starts out by using a single inbound IP address. When running in a Consumption or Premium plan, additional inbound IP addresses may be added as event-driven scale-out occurs. To find the inbound IP address or addresses being used by your app, use the `nslookup` utility from your local computer, as in the following example:

```
command  
nslookup <APP_NAME>.azurewebsites.net
```

In this example, replace `<APP_NAME>` with your function app name. If your app uses a [custom domain name](#), use `nslookup` for that custom domain name instead.

Function app outbound IP addresses

Each function app has a set of available outbound IP addresses. Any outbound connection from a function, such as to a back-end database, uses one of the available outbound IP addresses as the origin IP address. You can't know beforehand which IP address a given connection will use. For this reason, your back-end service must open its firewall to all of the function app's outbound IP addresses.

 Tip

For some platform-level features such as [Key Vault references](#), the origin IP might not be one of the outbound IPs, and you should not configure the target resource to rely on these specific addresses. It is recommended that the app instead use a virtual network integration, as the platform will route traffic to the target resource through that network.

To find the outbound IP addresses available to a function app:

Azure portal

1. Sign in to the [Azure Resource Explorer](#).
2. Select **subscriptions** > {your subscription} > **providers** > **Microsoft.Web** > **sites**.
3. In the JSON panel, find the site with an `id` property that ends in the name of your function app.
4. See `outboundIpAddresses` and `possibleOutboundIpAddresses`.

The set of `outboundIpAddresses` is currently available to the function app. The set of `possibleOutboundIpAddresses` includes IP addresses that will be available only if the function app [scales to other pricing tiers](#).

Note

When a function app that runs on the [Consumption plan](#) or the [Premium plan](#) is scaled, a new range of outbound IP addresses may be assigned. When running on either of these plans, you can't rely on the reported outbound IP addresses to create a definitive allowlist. To be able to include all potential outbound addresses used during dynamic scaling, you'll need to add the entire data center to your allowlist.

Data center outbound IP addresses

If you need to add the outbound IP addresses used by your function apps to an allowlist, another option is to add the function apps' data center (Azure region) to an allowlist. You can [download a JSON file that lists IP addresses for all Azure data centers](#). Then find the JSON fragment that applies to the region that your function app runs in.

For example, the following JSON fragment is what the allowlist for Western Europe might look like:

```
JSON

{
  "name": "AzureCloud.westeurope",
  "id": "AzureCloud.westeurope",
  "properties": {
    "changeNumber": 9,
    "region": "westeurope",
    "platform": "Azure",
    "systemService": "",
    "addressPrefixes": [
      "13.69.0.0/17",
      "13.73.128.0/18",
      ... Some IP addresses not shown here
      "213.199.180.192/27",
      "213.199.183.0/24"
    ]
  }
}
```

For information about when this file is updated and when the IP addresses change, expand the **Details** section of the [Download Center page](#).

Inbound IP address changes

The inbound IP address **might** change when you:

- Delete a function app and recreate it in a different resource group.
- Delete the last function app in a resource group and region combination, and recreate it.
- Delete a TLS binding, such as during [certificate renewal](#).

When your function app runs in a [Consumption plan](#) or in a [Premium plan](#), the inbound IP address might also change even when you haven't taken any actions such as the ones [listed above](#).

Outbound IP address changes

The relative stability of the outbound IP address depends on the hosting plan.

Consumption and Premium plans

Because of autoscaling behaviors, the outbound IP can change at any time when running on a [Consumption plan](#) or in a [Premium plan](#).

If you need to control the outbound IP address of your function app, such as when you need to add it to an allow list, consider implementing a [virtual network NAT gateway](#) while running in a Premium hosting plan. You can also do this by running in a Dedicated (App Service) plan.

Dedicated plans

When running on Dedicated (App Service) plans, the set of available outbound IP addresses for a function app might change when you:

- Take any action that can change the inbound IP address.
- Change your Dedicated (App Service) plan pricing tier. The list of all possible outbound IP addresses your app can use, for all pricing tiers, is in the `possibleOutboundIPAddresses` property. See [Find outbound IPs](#).

Forcing an outbound IP address change

Use the following procedure to deliberately force an outbound IP address change in a Dedicated (App Service) plan:

1. Scale your App Service plan up or down between Standard and Premium v2 pricing tiers.
2. Wait 10 minutes.
3. Scale back to where you started.

IP address restrictions

You can configure a list of IP addresses that you want to allow or deny access to a function app. For more information, see [Azure App Service Static IP Restrictions](#).

Dedicated IP addresses

There are several strategies to explore when your function app requires static, dedicated IP addresses.

Virtual network NAT gateway for outbound static IP

You can control the IP address of outbound traffic from your functions by using a virtual network NAT gateway to direct traffic through a static public IP address. You can use this topology when running in a [Premium plan](#) or in a [Dedicated \(App Service\) plan](#). To learn more, see [Tutorial: Control Azure Functions outbound IP with an Azure virtual network NAT gateway](#).

App Service Environments

For full control over the IP addresses, both inbound and outbound, we recommend [App Service Environments](#) (the [Isolated tier](#) of App Service plans). For more information, see [App Service Environment IP addresses](#) and [How to control inbound traffic to an App Service Environment](#).

To find out if your function app runs in an App Service Environment:

Azure portal

1. Sign in to the [Azure portal](#).
2. Navigate to the function app.
3. Select the **Overview** tab.
4. The App Service plan tier appears under **App Service plan/pricing tier**. The App Service Environment pricing tier is **Isolated**.

The App Service Environment `sku` is `Isolated`.

Next steps

A common cause of IP changes is function app scale changes. [Learn more about function app scaling](#).

Azure Functions custom handlers

Article • 12/16/2021

Every Functions app is executed by a language-specific handler. While Azure Functions features many [language handlers](#) by default, there are cases where you may want to use other languages or runtimes.

Custom handlers are lightweight web servers that receive events from the Functions host. Any language that supports HTTP primitives can implement a custom handler.

Custom handlers are best suited for situations where you want to:

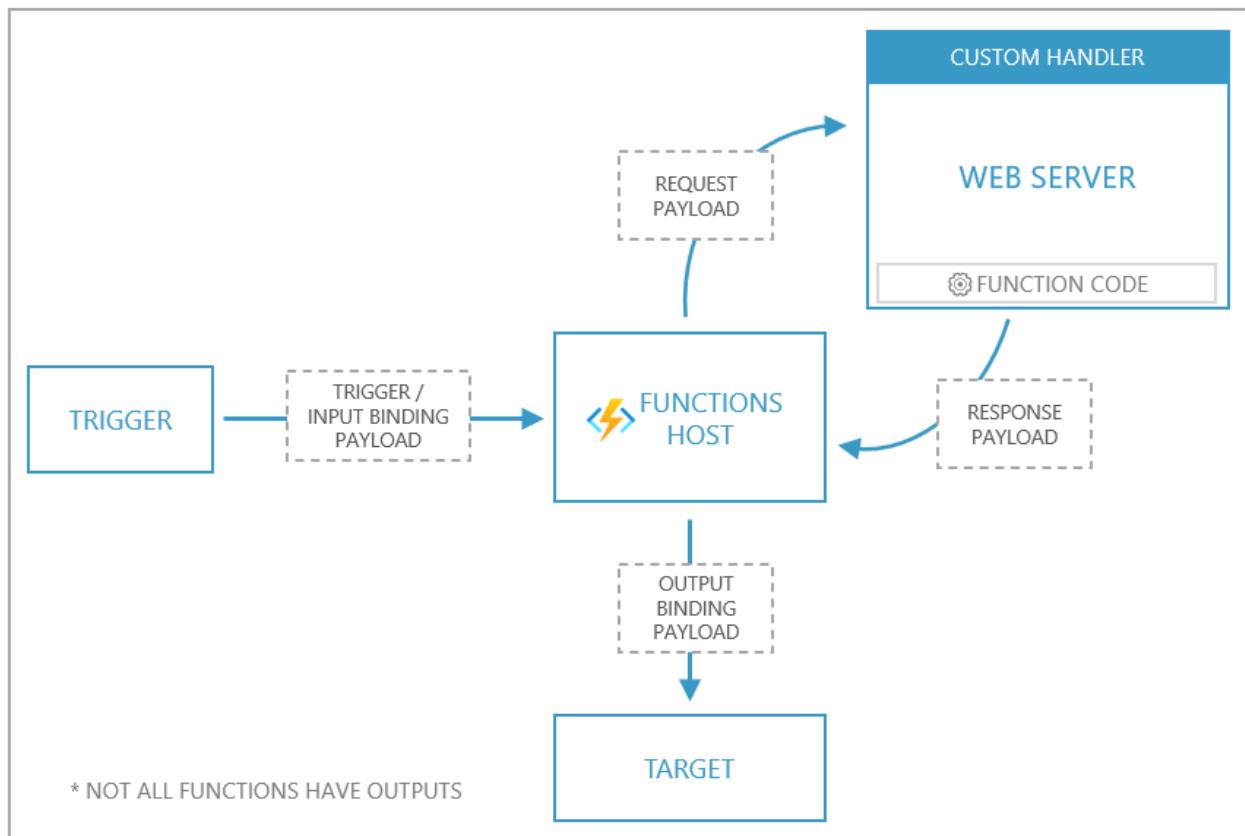
- Implement a function app in a language that's not currently offered out-of-the box, such as Go or Rust.
- Implement a function app in a runtime that's not currently featured by default, such as Deno.

With custom handlers, you can use [triggers and input and output bindings](#) via [extension bundles](#).

Get started with Azure Functions custom handlers with [quickstarts in Go and Rust](#).

Overview

The following diagram shows the relationship between the Functions host and a web server implemented as a custom handler.



1. Each event triggers a request sent to the Functions host. An event is any trigger that is supported by Azure Functions.
2. The Functions host then issues a **request payload** to the web server. The payload holds trigger and input binding data and other metadata for the function.
3. The web server executes the individual function, and returns a **response payload** to the Functions host.
4. The Functions host passes data from the response to the function's output bindings for processing.

An Azure Functions app implemented as a custom handler must configure the *host.json*, *local.settings.json*, and *function.json* files according to a few conventions.

Application structure

To implement a custom handler, you need the following aspects to your application:

- A *host.json* file at the root of your app
- A *local.settings.json* file at the root of your app
- A *function.json* file for each function (inside a folder that matches the function name)
- A command, script, or executable, which runs a web server

The following diagram shows how these files look on the file system for a function named "MyQueueFunction" and a custom handler executable named *handler.exe*.

Bash

```
| /MyQueueFunction  
|   function.json  
|  
| host.json  
| local.settings.json  
| handler.exe
```

Configuration

The application is configured via the *host.json* and *local.settings.json* files.

host.json

host.json tells the Functions host where to send requests by pointing to a web server capable of processing HTTP events.

A custom handler is defined by configuring the *host.json* file with details on how to run the web server via the `customHandler` section.

JSON

```
{  
  "version": "2.0",  
  "customHandler": {  
    "description": {  
      "defaultExecutablePath": "handler.exe"  
    }  
  }  
}
```

The `customHandler` section points to a target as defined by the `defaultExecutablePath`.

The execution target may either be a command, executable, or file where the web server is implemented.

Use the `arguments` array to pass any arguments to the executable. Arguments support expansion of environment variables (application settings) using `%%` notation.

You can also change the working directory used by the executable with `workingDirectory`.

JSON

```
{  
  "version": "2.0",  
  "customHandler": {  
    "description": {  
      "defaultExecutablePath": "app/handler.exe",  
      "arguments": [  
        "--database-connection-string",  
        "%DATABASE_CONNECTION_STRING%"  
      ],  
      "workingDirectory": "app"  
    }  
  }  
}
```

Bindings support

Standard triggers along with input and output bindings are available by referencing [extension bundles](#) in your `host.json` file.

local.settings.json

`local.settings.json` defines application settings used when running the function app locally. As it may contain secrets, `local.settings.json` should be excluded from source control. In Azure, use application settings instead.

For custom handlers, set `FUNCTIONS_WORKER_RUNTIME` to `Custom` in `local.settings.json`.

JSON

```
{  
  "IsEncrypted": false,  
  "Values": {  
    "FUNCTIONS_WORKER_RUNTIME": "Custom"  
  }  
}
```

Function metadata

When used with a custom handler, the `function.json` contents are no different from how you would define a function under any other context. The only requirement is that `function.json` files must be in a folder named to match the function name.

The following `function.json` configures a function that has a queue trigger and a queue output binding. Because it's in a folder named `MyQueueFunction`, it defines a function

named *MyQueueFunction*.

MyQueueFunction/function.json

```
JSON

{
  "bindings": [
    {
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "messages-incoming",
      "connection": "AzureWebJobsStorage"
    },
    {
      "name": "$return",
      "type": "queue",
      "direction": "out",
      "queueName": "messages-outgoing",
      "connection": "AzureWebJobsStorage"
    }
  ]
}
```

Request payload

When a queue message is received, the Functions host sends an HTTP post request to the custom handler with a payload in the body.

The following code represents a sample request payload. The payload includes a JSON structure with two members: `Data` and `Metadata`.

The `Data` member includes keys that match input and trigger names as defined in the bindings array in the *function.json* file.

The `Metadata` member includes [metadata generated from the event source](#).

```
JSON

{
  "Data": {
    "myQueueItem": "{ message: \"Message sent\" }"
  },
  "Metadata": {
    "DequeueCount": 1,
    "ExpirationTime": "2019-10-16T17:58:31+00:00",
    "Id": "800ae4b3-bdd2-4c08-badd-f08e5a34b865",
    "InsertionTime": "2019-10-09T17:58:31+00:00",
    "LastModifiedTime": "2019-10-09T17:58:31+00:00"
  }
}
```

```

    "NextVisibleTime": "2019-10-09T18:08:32+00:00",
    "PopReceipt": "AgAAAAAMAAAAAAAAGtnj8x+1QE=",
    "sys": {
        "MethodName": "QueueTrigger",
        "UtcNow": "2019-10-09T17:58:32.2205399Z",
        "RandGuid": "24ad4c06-24ad-4e5b-8294-3da9714877e9"
    }
}

```

Response payload

By convention, function responses are formatted as key/value pairs. Supported keys include:

Payload key	Data type	Remarks
Outputs	object	Holds response values as defined by the <code>bindings</code> array in <code>function.json</code> . For instance, if a function is configured with a queue output binding named "myQueueOutput", then <code>Outputs</code> contains a key named <code>myQueueOutput</code> , which is set by the custom handler to the messages that are sent to the queue.
Logs	array	Messages appear in the Functions invocation logs. When running in Azure, messages appear in Application Insights.
ReturnValue	string	Used to provide a response when an output is configured as <code>\$return</code> in the <code>function.json</code> file.

This is an example of a response payload.

JSON

```
{
  "Outputs": {
    "res": {
      "body": "Message enqueued"
    },
    "myQueueOutput": [
      "queue message 1",
      "queue message 2"
    ]
  },
  "Logs": [
    "Log message 1",
    "Log message 2"
  ]
}
```

```
],
  "ReturnValue": "{\"hello\":\"world\"}"
}
```

Examples

Custom handlers can be implemented in any language that supports receiving HTTP events. The following examples show how to implement a custom handler using the Go programming language.

Function with bindings

The scenario implemented in this example features a function named `order` that accepts a `POST` with a payload representing a product order. As an order is posted to the function, a Queue Storage message is created and an HTTP response is returned.

Implementation

In a folder named `order`, the `function.json` file configures the HTTP-triggered function.

order/function.json

JSON

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": ["post"]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    },
    {
      "type": "queue",
      "name": "message",
      "direction": "out",
      "queueName": "orders",
      "connection": "AzureWebJobsStorage"
    }
  ]
}
```

This function is defined as an [HTTP triggered function](#) that returns an [HTTP response](#) and outputs a [Queue storage](#) message.

At the root of the app, the `host.json` file is configured to run an executable file named `handler.exe` (`handler` in Linux or macOS).

JSON

```
{  
  "version": "2.0",  
  "customHandler": {  
    "description": {  
      "defaultExecutablePath": "handler.exe"  
    }  
  },  
  "extensionBundle": {  
    "id": "Microsoft.Azure.Functions.ExtensionBundle",  
    "version": "[1.*, 2.0.0)"  
  }  
}
```

This is the HTTP request sent to the Functions runtime.

HTTP

```
POST http://127.0.0.1:7071/api/order HTTP/1.1  
Content-Type: application/json  
  
{  
  "id": 1005,  
  "quantity": 2,  
  "color": "black"  
}
```

The Functions runtime will then send the following HTTP request to the custom handler:

HTTP

```
POST http://127.0.0.1:<FUNCTIONS_CUSTOMHANDLER_PORT>/order HTTP/1.1  
Content-Type: application/json  
  
{  
  "Data": {  
    "req": {  
      "Url": "http://localhost:7071/api/order",  
      "Method": "POST",  
      "Query": "{}",  
      "Headers": {  
        "Content-Type": [  
          "application/json"  
        ]  
      }  
    }  
  }  
}
```

```
        ]
    },
    "Params": {},
    "Body": "{\"id\":1005,\"quantity\":2,\"color\":\"black\"}"
}
},
"Metadata": {
}
}
```

ⓘ Note

Some portions of the payload were removed for brevity.

handler.exe is the compiled Go custom handler program that runs a web server and responds to function invocation requests from the Functions host.

Go

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
)

type InvokeRequest struct {
    Data      map[string]json.RawMessage
    Metadata map[string]interface{}
}

type InvokeResponse struct {
    Outputs    map[string]interface{}
    Logs       []string
    ReturnValue interface{}
}

func orderHandler(w http.ResponseWriter, r *http.Request) {
    var invokeRequest InvokeRequest

    d := json.NewDecoder(r.Body)
    d.Decode(&invokeRequest)

    var reqData map[string]interface{}
    json.Unmarshal(invokeRequest.Data["req"], &reqData)

    outputs := make(map[string]interface{})
    outputs["message"] = reqData["Body"]
```

```

resData := make(map[string]interface{})
resData["body"] = "Order enqueued"
outputs["res"] = resData
invokeResponse := InvokeResponse{outputs, nil, nil}

responseJson, _ := json.Marshal(invokeResponse)

w.Header().Set("Content-Type", "application/json")
w.Write(responseJson)
}

func main() {
    customHandlerPort, exists :=
os.LookupEnv("FUNCTIONS_CUSTOMHANDLER_PORT")
    if !exists {
        customHandlerPort = "8080"
    }
    mux := http.NewServeMux()
    mux.HandleFunc("/order", orderHandler)
    fmt.Println("Go server Listening on: ", customHandlerPort)
    log.Fatal(http.ListenAndServe(": "+customHandlerPort, mux))
}

```

In this example, the custom handler runs a web server to handle HTTP events and is set to listen for requests via the `FUNCTIONS_CUSTOMHANDLER_PORT`.

Even though the Functions host received original HTTP request at `/api/order`, it invokes the custom handler using the function name (its folder name). In this example, the function is defined at the path of `/order`. The host sends the custom handler an HTTP request at the path of `/order`.

As `POST` requests are sent to this function, the trigger data and function metadata are available via the HTTP request body. The original HTTP request body can be accessed in the payload's `Data.req.Body`.

The function's response is formatted into key/value pairs where the `Outputs` member holds a JSON value where the keys match the outputs as defined in the `function.json` file.

This is an example payload that this handler returns to the Functions host.

JSON

```
{
  "Outputs": {
    "message": "{\"id\":1005,\"quantity\":2,\"color\":\"black\"}",
    "res": {
      "body": "Order enqueued"
    }
}
```

```
  },
  "Logs": null,
  "ReturnValue": null
}
```

By setting the `message` output equal to the order data that came in from the request, the function outputs that order data to the configured queue. The Functions host also returns the HTTP response configured in `res` to the caller.

HTTP-only function

For HTTP-triggered functions with no additional bindings or outputs, you may want your handler to work directly with the HTTP request and response instead of the custom handler `request` and `response` payloads. This behavior can be configured in `host.json` using the `enableForwardingHttpRequest` setting.

ⓘ Important

The primary purpose of the custom handlers feature is to enable languages and runtimes that do not currently have first-class support on Azure Functions. While it may be possible to run web applications using custom handlers, Azure Functions is not a standard reverse proxy. Some features such as response streaming, HTTP/2, and WebSockets are not available. Some components of the HTTP request such as certain headers and routes may be restricted. Your application may also experience excessive **cold start**.

To address these circumstances, consider running your web apps on [Azure App Service](#).

The following example demonstrates how to configure an HTTP-triggered function with no additional bindings or outputs. The scenario implemented in this example features a function named `hello` that accepts a `GET` or `POST`.

Implementation

In a folder named `hello`, the `function.json` file configures the HTTP-triggered function.

hello/function.json

JSON

```
{  
  "bindings": [  
    {  
      "type": "httpTrigger",  
      "authLevel": "anonymous",  
      "direction": "in",  
      "name": "req",  
      "methods": ["get", "post"]  
    },  
    {  
      "type": "http",  
      "direction": "out",  
      "name": "res"  
    }  
  ]  
}
```

The function is configured to accept both `GET` and `POST` requests and the result value is provided via an argument named `res`.

At the root of the app, the `host.json` file is configured to run `handler.exe` and `enableForwardingHttpRequest` is set to `true`.

JSON

```
{  
  "version": "2.0",  
  "customHandler": {  
    "description": {  
      "defaultExecutablePath": "handler.exe"  
    },  
    "enableForwardingHttpRequest": true  
  }  
}
```

When `enableForwardingHttpRequest` is `true`, the behavior of HTTP-only functions differs from the default custom handlers behavior in these ways:

- The HTTP request does not contain the custom handlers `request` payload. Instead, the Functions host invokes the handler with a copy of the original HTTP request.
- The Functions host invokes the handler with the same path as the original request including any query string parameters.
- The Functions host returns a copy of the handler's HTTP response as the response to the original request.

The following is a POST request to the Functions host. The Functions host then sends a copy of the request to the custom handler at the same path.

HTTP

```
POST http://127.0.0.1:7071/api/hello HTTP/1.1
Content-Type: application/json

{
  "message": "Hello World!"
}
```

The file `handler.go` file implements a web server and HTTP function.

Go

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "os"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    if r.Method == "GET" {
        w.Write([]byte("hello world"))
    } else {
        body, _ := ioutil.ReadAll(r.Body)
        w.Write(body)
    }
}

func main() {
    customHandlerPort, exists :=
    os.LookupEnv("FUNCTIONS_CUSTOMHANDLER_PORT")
    if !exists {
        customHandlerPort = "8080"
    }
    mux := http.NewServeMux()
    mux.HandleFunc("/api/hello", helloHandler)
    fmt.Println("Go server Listening on: ", customHandlerPort)
    log.Fatal(http.ListenAndServe(": "+customHandlerPort, mux))
}
```

In this example, the custom handler creates a web server to handle HTTP events and is set to listen for requests via the `FUNCTIONS_CUSTOMHANDLER_PORT`.

`GET` requests are handled by returning a string, and `POST` requests have access to the request body.

The route for the order function here is `/api/hello`, same as the original request.

ⓘ Note

The `FUNCTIONS_CUSTOMHANDLER_PORT` is not the public facing port used to call the function. This port is used by the Functions host to call the custom handler.

Deploying

A custom handler can be deployed to every Azure Functions hosting option. If your handler requires operating system or platform dependencies (such as a language runtime), you may need to use a [custom container](#).

When creating a function app in Azure for custom handlers, we recommend you select .NET Core as the stack.

To deploy a custom handler app using Azure Functions Core Tools, run the following command.

```
Bash
```

```
func azure functionapp publish $functionAppName
```

ⓘ Note

Ensure all files required to run your custom handler are in the folder and included in the deployment. If your custom handler is a binary executable or has platform-specific dependencies, ensure these files match the target deployment platform.

Restrictions

- The custom handler web server needs to start within 60 seconds.

Samples

Refer to the [custom handler samples GitHub repo](#) ↗ for examples of how to implement functions in a variety of different languages.

Troubleshooting and support

Trace logging

If your custom handler process fails to start up or if it has problems communicating with the Functions host, you can increase the function app's log level to `Trace` to see more diagnostic messages from the host.

To change the function app's default log level, configure the `logLevel` setting in the `logging` section of `host.json`.

```
JSON

{
  "version": "2.0",
  "customHandler": {
    "description": {
      "defaultExecutablePath": "handler.exe"
    }
  },
  "logging": {
    "logLevel": {
      "default": "Trace"
    }
  }
}
```

The Functions host outputs extra log messages including information related to the custom handler process. Use the logs to investigate problems starting your custom handler process or invoking functions in your custom handler.

Locally, logs are printed to the console.

In Azure, [query Application Insights traces](#) to view the log messages. If your app produces a high volume of logs, only a subset of log messages are sent to Application Insights. [Disable sampling](#) to ensure all messages are logged.

Test custom handler in isolation

Custom handler apps are a web server process, so it may be helpful to start it on its own and test function invocations by sending mock [HTTP requests](#) using a tool like [cURL](#) ↗ or [Postman](#) ↗.

You can also use this strategy in your CI/CD pipelines to run automated tests on your custom handler.

Execution environment

Custom handlers run in the same environment as a typical Azure Functions app. Test your handler to ensure the environment contains all the dependencies it needs to run. For apps that require additional dependencies, you may need to run them using a [custom container image](#) hosted on Azure Functions [Premium plan](#).

Get support

If you need help on a function app with custom handlers, you can submit a request through regular support channels. However, due to the wide variety of possible languages used to build custom handlers apps, support is not unlimited.

Support is available if the Functions host has problems starting or communicating with the custom handler process. For problems specific to the inner workings of your custom handler process, such as issues with the chosen language or framework, our Support Team is unable to provide assistance in this context.

Next steps

Get started building an Azure Functions app in Go or Rust with the [custom handlers quickstart](#).

Supported languages in Azure Functions

Article • 02/25/2024

This article explains the levels of support offered for your preferred language when using Azure Functions. It also describes strategies for creating functions using languages not natively supported.

There are two levels of support:

- **Generally available (GA)** - Fully supported and approved for production use.
- Preview - Not yet supported, but expected to reach GA status in the future.

Languages by runtime version

The following table shows the .NET versions supported by Azure Functions. Select your preferred development language at the top of the article.

The supported version of .NET depends on both your Functions runtime version and your chosen execution model:

Isolated worker model		
Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .		
v4.x		
Expand table		
Supported version	Support level	Expected community EOL date
.NET 8	GA	November 10, 2026
.NET 6	GA	November 12, 2024
.NET Framework 4.8	GA	See policy

.NET 7 was previously supported on the isolated worker model but reached the end of official support on [May 14, 2024](#).

For more information, see [Guide for running C# Azure Functions in an isolated worker process](#).

For information about planned changes to language support, see [Azure roadmap](#).

Language support details

The following table shows which languages supported by Functions can run on Linux or Windows. It also indicates whether your language supports editing in the Azure portal. The language is based on the **Runtime stack** option you choose when [creating your function app in the Azure portal](#). This is the same as the `--worker-runtime` option when using the `func init` command in Azure Functions Core Tools.

[+] Expand table

Language	Runtime stack	Linux	Windows	In-portal editing
C# (isolated worker model)	.NET	✓	✓	
C# (in-process model)	.NET	✓	✓	
C# script	.NET	✓	✓	✓
JavaScript	Node.js	✓	✓	✓
Python	Python	✓	X	✓
Java	Java	✓	✓	
PowerShell	PowerShell Core	✓	✓	✓
TypeScript	Node.js	✓	✓	
Go/Rust/other	Custom Handlers	✓	✓	

For more information on operating system and language support, see [Operating system/runtime support](#).

When in-portal editing isn't available, you must instead [develop your functions locally](#).

Language major version support

Azure Functions provides a guarantee of support for the major versions of supported programming languages. For most languages, there are minor or patch versions released to update a supported major version. Examples of minor or patch versions include such as Python 3.9.1 and Node 14.17. After new minor versions of supported languages become available, the minor versions used by your functions apps are automatically upgraded to these newer minor or patch versions.

Note

Because Azure Functions can remove the support of older minor versions at any time after a new minor version is available, you shouldn't pin your function apps to a specific minor/patch version of a programming language.

Custom handlers

Custom handlers are lightweight web servers that receive events from the Azure Functions host. Any language that supports HTTP primitives can implement a custom handler. This means that custom handlers can be used to create functions in languages that aren't officially supported. To learn more, see [Azure Functions custom handlers](#).

Language extensibility

Starting with version 2.x, the runtime is designed to offer [language extensibility](#). The JavaScript and Java languages in the 2.x runtime are built with this extensibility.

Next steps

Isolated worker model

[.NET isolated worker process reference](#)

Guide for running C# Azure Functions in the isolated worker model

Article • 10/16/2024

This article is an introduction to working with Azure Functions in .NET, using the isolated worker model. This model allows your project to target versions of .NET independently of other runtime components. For information about specific .NET versions supported, see [supported version](#).

Use the following links to get started right away building .NET isolated worker model functions.

 Expand table

Getting started	Concepts	Samples
<ul style="list-style-type: none">• Using Visual Studio Code• Using command line tools• Using Visual Studio	<ul style="list-style-type: none">• Hosting options• Monitoring	<ul style="list-style-type: none">• Reference samples ↗

To learn just about deploying an isolated worker model project to Azure, see [Deploy to Azure Functions](#).

Benefits of the isolated worker model

There are two modes in which you can run your .NET class library functions: either [in the same process](#) as the Functions host runtime (*in-process*) or in an isolated worker process. When your .NET functions run in an isolated worker process, you can take advantage of the following benefits:

- **Fewer conflicts:** Because your functions run in a separate process, assemblies used in your app don't conflict with different versions of the same assemblies used by the host process.
- **Full control of the process:** You control the start-up of the app, which means that you can manage the configurations used and the middleware started.
- **Standard dependency injection:** Because you have full control of the process, you can use current .NET behaviors for dependency injection and incorporating middleware into your function app.
- **.NET version flexibility:** Running outside of the host process means that your functions can run on versions of .NET not natively supported by the Functions

runtime, including the .NET Framework.

If you have an existing C# function app that runs in-process, you need to migrate your app to take advantage of these benefits. For more information, see [Migrate .NET apps from the in-process model to the isolated worker model](#).

For a comprehensive comparison between the two modes, see [Differences between in-process and isolate worker process .NET Azure Functions](#).

Supported versions

Versions of the Functions runtime support specific versions of .NET. To learn more about Functions versions, see [Azure Functions runtime versions overview](#). Version support also depends on whether your functions run in-process or isolated worker process.

 **Note**

To learn how to change the Functions runtime version used by your function app, see [view and update the current runtime version](#).

The following table shows the highest level of .NET or .NET Framework that can be used with a specific version of Functions.

 Expand table

Functions runtime version	Isolated worker model	In-process model ⁵
Functions 4.x ¹	.NET 9.0 (preview) .NET 8.0 .NET 6.0 ² .NET Framework 4.8 ³	.NET 8.0 .NET 6.0 ²
Functions 1.x ⁴	n/a	.NET Framework 4.8

¹ .NET 7 was previously supported on the isolated worker model but reached the [end of official support](#) on May 14, 2024.

² .NET 6 reaches the [end of official support](#) on November 12, 2024.

³ The build process also requires the [.NET SDK](#).

⁴ Support ends for version 1.x of the Azure Functions runtime on September 14, 2026. For more information, see [this support announcement](#). For continued full support, you should [migrate your apps to version 4.x](#).

⁵ Support ends for the in-process model on November 10, 2026. For more information, see [this support announcement](#). For continued full support, you should [migrate your apps to the isolated worker model](#).

For the latest news about Azure Functions releases, including the removal of specific older minor versions, monitor [Azure App Service announcements](#).

Project structure

A .NET project for Azure Functions using the isolated worker model is basically a .NET console app project that targets a supported .NET runtime. The following are the basic files required in any .NET isolated project:

- C# project file (.csproj) that defines the project and dependencies.
- Program.cs file that's the entry point for the app.
- Any code files [defining your functions](#).
- `host.json` file that defines configuration shared by functions in your project.
- `local.settings.json` file that defines environment variables used by your project when run locally on your machine.

For complete examples, see the [.NET 8 sample project](#) and the [.NET Framework 4.8 sample project](#).

Package references

A .NET project for Azure Functions using the isolated worker model uses a unique set of packages, for both core functionality and binding extensions.

Core packages

The following packages are required to run your .NET functions in an isolated worker process:

- `Microsoft.Azure.Functions.Worker`
- `Microsoft.Azure.Functions.Worker.Sdk`

Version 2.x (Preview)

The 2.x versions of the core packages change the supported frameworks and bring in support for new .NET APIs from these later versions. When you target .NET 9 (Preview) or later, your app needs to reference version 2.0.0-preview1 or later of both packages.

The initial preview versions are compatible with code written against version 1.x. However, during the preview period, newer versions may introduce behavior changes that could influence the code you write.

When updating to the 2.x versions, note the following changes:

- Starting with version 2.0.0-preview2, [Microsoft.Azure.Functions.Worker.Sdk](#) adds default configurations for [SDK container builds](#).
- Starting with version 2.0.0-preview2 of [Microsoft.Azure.Functions.Worker](#):
 - This version adds support for `IHostApplicationBuilder`. Some examples in this guide include tabs to show alternatives using `IHostApplicationBuilder`. These examples require the 2.x versions.
 - Service provider scope validation is included by default if run in a development environment. This behavior matches ASP.NET Core.
 - The `EnableUserCodeException` option is enabled by default. The property is now marked as obsolete.
 - The `IncludeEmptyEntriesInMessagePayload` option is enabled by default. With this option enabled, trigger payloads that represent collections always include empty entries. For example, if a message is sent without a body, an empty entry would still be present in `string[]` for the trigger data. The inclusion of empty entries facilitates cross-referencing with metadata arrays which the function may also reference. You can disable this behavior by setting `IncludeEmptyEntriesInMessagePayload` to `false` in the `WorkerOptions` service configuration.
 - The `ILoggerExtensions` class is renamed to `FunctionsLoggerExtensions`. The rename prevents an ambiguous call error when using `LogMetric()` on an `ILogger` instance.

Extension packages

Because .NET isolated worker process functions use different binding types, they require a unique set of binding extension packages.

You find these extension packages under [Microsoft.Azure.Functions.Worker.Extensions](#).

Start-up and configuration

When you use the isolated worker model, you have access to the start-up of your function app, which is usually in `Program.cs`. You're responsible for creating and starting

your own host instance. As such, you also have direct access to the configuration pipeline for your app. With .NET Functions isolated worker process, you can much more easily add configurations, inject dependencies, and run your own middleware.

IHostBuilder

The following code shows an example of a [HostBuilder](#) pipeline:

```
C#  
  
var host = new HostBuilder()  
    .ConfigureFunctionsWorkerDefaults()  
    .ConfigureServices(s =>  
    {  
        s.AddApplicationInsightsTelemetryWorkerService();  
        s.ConfigureFunctionsApplicationInsights();  
        s.AddSingleton<IHttpResponderService,  
DefaultHttpResponderService>();  
        s.Configure<LoggerFilterOptions>(options =>  
        {  
            // The Application Insights SDK adds a default logging  
            filter that instructs ILogger to capture only Warning and more severe  
            logs. Application Insights requires an explicit override.  
            // Log levels can also be configured using appsettings.json.  
            // For more information, see https://learn.microsoft.com/en-us/azure/azure-  
            monitor/app/worker-service#ilogger-logs  
            LoggerFilterRule toRemove =  
            options.Rules.FirstOrDefault(rule => rule.ProviderName  
                ==  
                "Microsoft.Extensions.Logging.ApplicationInsights.ApplicationInsightsLog  
                gerProvider");  
  
            if (toRemove is not null)  
            {  
                options.Rules.Remove(toRemove);  
            }  
        });  
    })  
    .Build();
```

This code requires `using Microsoft.Extensions.DependencyInjection;`.

Before calling `Build()` on the `IHostBuilder`, you should:

- Call either `ConfigureFunctionsWebApplication()` if using [ASP.NET Core integration](#) or `ConfigureFunctionsWorkerDefaults()` otherwise. See [HTTP trigger](#) for details on these options.
If you're writing your application using F#, some trigger and binding extensions require extra configuration. See the setup documentation for the

[Blobs extension](#), the [Tables extension](#), and the [Cosmos DB extension](#) when you plan to use these extensions in an F# app.

- Configure any services or app configuration your project requires. See [Configuration](#) for details.

If you're planning to use Application Insights, you need to call

```
AddApplicationInsightsTelemetryWorkerService()
```

```
ConfigureFunctionsApplicationInsights()
```

in the `ConfigureServices()` delegate. See [Application Insights](#) for details.

If your project targets .NET Framework 4.8, you also need to add `FunctionsDebugger.Enable();` before creating the HostBuilder. It should be the first line of your `Main()` method. For more information, see [Debugging when targeting .NET Framework](#).

The [HostBuilder](#) is used to build and return a fully initialized [IHost](#) instance, which you run asynchronously to start your function app.

```
C#
```

```
await host.RunAsync();
```

Configuration

The type of builder you use determines how you can configure the application.

```
IHostBuilder
```

The [ConfigureFunctionsWorkerDefaults](#) method is used to add the settings required for the function app to run. The method includes the following functionality:

- Default set of converters.
- Set the default [JsonSerializerOptions](#) to ignore casing on property names.
- Integrate with Azure Functions logging.
- Output binding middleware and features.
- Function execution middleware.
- Default gRPC support.

```
C#
```

```
.ConfigureFunctionsWorkerDefaults()
```

Having access to the host builder pipeline means that you can also set any app-specific configurations during initialization. You can call the [ConfigureAppConfiguration](#) method on [HostBuilder](#) one or more times to add any configuration sources required by your code. To learn more about app configuration, see [Configuration in ASP.NET Core](#).

These configurations only apply to the worker code you author, and they don't directly influence the configuration of the Functions host or triggers and bindings. To make changes to the functions host or trigger and binding configuration, you still need to use the [host.json file](#).

ⓘ Note

Custom configuration sources cannot be used for configuration of triggers and bindings. Trigger and binding configuration must be available to the Functions platform, and not just your application code. You can provide this configuration through the [application settings](#), [Key Vault references](#), or [App Configuration references](#) features.

Dependency injection

The isolated worker model uses standard .NET mechanisms for injecting services.

IHostBuilder

When you use a `HostBuilder`, call [ConfigureServices](#) on the host builder and use the extension methods on [IServiceCollection](#) to inject specific services. The following example injects a singleton service dependency:

C#

```
.ConfigureServices(services =>
{
    services.AddSingleton<IHttpResponderService,
DefaultHttpResponderService>();
})
```

This code requires `using Microsoft.Extensions.DependencyInjection;`. To learn more, see [Dependency injection in ASP.NET Core](#).

Register Azure clients

Dependency injection can be used to interact with other Azure services. You can inject clients from the [Azure SDK for .NET](#) using the [Microsoft.Extensions.Azure](#) package. After installing the package, [register the clients](#) by calling `AddAzureClients()` on the service collection in `Program.cs`. The following example configures a [named client](#) for Azure Blobs:

```
IHostBuilder

C#

using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Azure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices((hostContext, services) =>
{
    services.AddAzureClients(clientBuilder =>
    {

        clientBuilder.AddBlobServiceClient(hostContext.Configuration.GetSection(
            "MyStorageConnection"))
            .WithName("copierOutputBlob");
    });
})
.Build();

host.Run();
```

The following example shows how we can use this registration and [SDK types](#) to copy blob contents as a stream from one container to another using an injected client:

```
C#


using Microsoft.Extensions.Azure;
using Microsoft.Extensions.Logging;

namespace MyFunctionApp
{
    public class BlobCopier
    {
        private readonly ILogger<BlobCopier> _logger;
        private readonly BlobContainerClient _copyContainerClient;
```

```

        public BlobCopier	ILogger<BlobCopier> logger,
IAzureClientFactory<BlobServiceClient> blobClientFactory)
{
    _logger = logger;
    _copyContainerClient =
blobClientFactory.CreateClient("copierOutputBlob").GetBlobContainerClient("s
amples-workitems-copy");
    _copyContainerClient.CreateIfNotExists();
}

[Function("BlobCopier")]
public async Task Run([BlobTrigger("samples-workitems/{name}", Connection = "MyStorageConnection")] Stream myBlob, string name)
{
    await _copyContainerClient.UploadBlobAsync(name, myBlob);
    _logger.LogInformation($"Blob {name} copied!");
}
}

```

The `ILogger<T>` in this example was also obtained through dependency injection, so it's registered automatically. To learn more about configuration options for logging, see [Logging](#).

💡 Tip

The example used a literal string for the name of the client in both `Program.cs` and the function. Consider instead using a shared constant string defined on the function class. For example, you could add `public const string CopyStorageClientName = nameof(_copyContainerClient);` and then reference `BlobCopier.CopyStorageClientName` in both locations. You could similarly define the configuration section name with the function rather than in `Program.cs`.

Middleware

The isolated worker model also supports middleware registration, again by using a model similar to what exists in ASP.NET. This model gives you the ability to inject logic into the invocation pipeline, and before and after functions execute.

The `ConfigureFunctionsWorkerDefaults` extension method has an overload that lets you register your own middleware, as you can see in the following example.

IHostBuilder

C#

```
var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults(workerApplication =>
{
    // Register our custom middlewares with the worker

    workerApplication.UseMiddleware<ExceptionHandlingMiddleware>();

    workerApplication.UseMiddleware<MyCustomMiddleware>();

    workerApplication.UseWhen<StampHttpHeaderMiddleware>((context)
=>
{
    // We want to use this middleware only for http trigger
    invocations.
    return context.FunctionDefinition.InputBindings.Values
        .First(a => a.Type.EndsWith("Trigger")).Type
== "httpTrigger";
});
})
.Build();
```

The `UseWhen` extension method can be used to register a middleware that gets executed conditionally. You must pass to this method a predicate that returns a boolean value, and the middleware participates in the invocation processing pipeline when the return value of the predicate is `true`.

The following extension methods on `FunctionContext` make it easier to work with middleware in the isolated model.

[] [Expand table](#)

Method	Description
<code>GetHttpRequestDataAsync</code>	Gets the <code>HttpRequestData</code> instance when called by an HTTP trigger. This method returns an instance of <code>ValueTask<HttpRequestData?></code> , which is useful when you want to read message data, such as request headers and cookies.
<code>GetHttpResponseData</code>	Gets the <code>HttpResponseData</code> instance when called by an HTTP trigger.
<code>GetInvocationResult</code>	Gets an instance of <code>InvocationResult</code> , which represents the result of the current function execution. Use the <code>Value</code> property to get or set the value as needed.
<code>GetOutputBindings</code>	Gets the output binding entries for the current function execution. Each entry in the result of this method is of type <code>OutputBindingData</code> .

Method	Description
	You can use the <code>Value</code> property to get or set the value as needed.
<code>BindInputAsync</code>	Binds an input binding item for the requested <code>BindingMetadata</code> instance. For example, you can use this method when you have a function with a <code>BlobInput</code> input binding that needs to be used by your middleware.

This is an example of a middleware implementation that reads the `HttpRequestData` instance and updates the `HttpResponseData` instance during function execution:

C#

```
internal sealed class StampHttpHeaderMiddleware : IFunctionsWorkerMiddleware
{
    public async Task Invoke(FunctionContext context,
    FunctionExecutionDelegate next)
    {
        var requestData = await context.GetHttpRequestDataAsync();

        string correlationId;
        if (requestData!.Headers.TryGetValues("x-correlationId", out var
values))
        {
            correlationId = values.First();
        }
        else
        {
            correlationId = Guid.NewGuid().ToString();
        }

        await next(context);

        context.GetHttpResponseData()!.Headers.Add("x-correlationId",
correlationId);
    }
}
```

This middleware checks for the presence of a specific request header(x-correlationId), and when present uses the header value to stamp a response header. Otherwise, it generates a new GUID value and uses that for stamping the response header. For a more complete example of using custom middleware in your function app, see the [custom middleware reference sample ↗](#).

Customizing JSON serialization

The isolated worker model uses `System.Text.Json` by default. You can customize the behavior of the serializer by configuring services as part of your `Program.cs` file. This section covers general-purpose serialization and won't influence [HTTP trigger JSON serialization with ASP.NET Core integration](#), which must be configured separately.

IHostBuilder

The following example shows this using `ConfigureFunctionsWebApplication`, but it will also work for `ConfigureFunctionsWorkerDefaults`:

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()

.ConfigureFunctionsWebApplication((IFunctionsWorkerApplicationBuilder
builder) =>
{
    builder.Services.Configure<JsonSerializerOptions>
(jsonSerializerOptions =>
{
    jsonSerializerOptions.PropertyNamingPolicy =
JsonNamingPolicy.CamelCase;
    jsonSerializerOptions.DefaultIgnoreCondition =
JsonIgnoreCondition.WhenWritingNull;
    jsonSerializerOptions.ReferenceHandler =
ReferenceHandler.Preserve;

    // override the default value
    jsonSerializerOptions.PropertyNameCaseInsensitive = false;
});
})
.Build();

host.Run();
```

You might want to instead use JSON.NET (`Newtonsoft.Json`) for serialization. To do this, you would install the [Microsoft.Azure.Core.NewtonsoftJson](#) package. Then, in your service registration, you would reassign the `Serializer` property on the `WorkerOptions` configuration. The following example shows this using `ConfigureFunctionsWebApplication`, but it will also work for `ConfigureFunctionsWorkerDefaults`:

IHostBuilder

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()

.ConfigureFunctionsWebApplication((IFunctionsWorkerApplicationBuilder
builder) =>
{
    builder.Services.Configure<WorkerOptions>(workerOptions =>
    {
        var settings =
NewtonsoftJsonObjectSerializer.CreateJsonSerializerSettings();
        settings.ContractResolver = new
CamelCasePropertyNamesContractResolver();
        settings.NullValueHandling = NullValueHandling.Ignore;

        workerOptions.Serializer = new
NewtonsoftJsonObjectSerializer(settings);
    });
})
.Build();

host.Run();
```

Methods recognized as functions

A function method is a public method of a public class with a `Function` attribute applied to the method and a trigger attribute applied to an input parameter, as shown in the following example:

C#

```
[Function(nameof(QueueFunction))]
[QueueOutput("output-queue")]
public string[] Run([QueueTrigger("input-queue")] Album myQueueItem,
FunctionContext context)
```

The trigger attribute specifies the trigger type and binds input data to a method parameter. The previous example function is triggered by a queue message, and the queue message is passed to the method in the `myQueueItem` parameter.

The `Function` attribute marks the method as a function entry point. The name must be unique within a project, start with a letter and only contain letters, numbers, `_`, and `-`, up to 127 characters in length. Project templates often create a method named `Run`, but the method name can be any valid C# method name. The method must be a public member of a public class. It should generally be an instance method so that services can be passed in via [dependency injection](#).

Function parameters

Here are some of the parameters that you can include as part of a function method signature:

- [Bindings](#), which are marked as such by decorating the parameters as attributes. The function must contain exactly one trigger parameter.
- An [execution context object](#), which provides information about the current invocation.
- A [cancellation token](#), used for graceful shutdown.

Execution context

.NET isolated passes a `FunctionContext` object to your function methods. This object lets you get an `ILogger` instance to write to the logs by calling the `GetLogger` method and supplying a `categoryName` string. You can use this context to obtain an `ILogger` without having to use dependency injection. To learn more, see [Logging](#).

Cancellation tokens

A function can accept a `CancellationToken` parameter, which enables the operating system to notify your code when the function is about to be terminated. You can use this notification to make sure the function doesn't terminate unexpectedly in a way that leaves data in an inconsistent state.

Cancellation tokens are supported in .NET functions when running in an isolated worker process. The following example raises an exception when a cancellation request is received:

C#

```
[Function(nameof(ThrowOnCancellation))]
public async Task ThrowOnCancellation(
    [EventHubTrigger("sample-workitem-1", Connection =
    "EventHubConnection")] string[] messages,
```

```

        FunctionContext context,
        CancellationToken cancellationToken)
{
    _logger.LogInformation("C# EventHub {functionName} trigger function
processing a request.", nameof(ThrowOnCancellation));

    foreach (var message in messages)
    {
        cancellationToken.ThrowIfCancellationRequested();
        await Task.Delay(6000); // task delay to simulate message processing
        _logger.LogInformation("Message '{msg}' was processed.", message);
    }
}

```

The following example performs clean-up actions when a cancellation request is received:

C#

```

[Function(nameof(HandleCancellationCleanup))]
public async Task HandleCancellationCleanup(
    [EventHubTrigger("sample-workitem-2", Connection =
"EventHubConnection")] string[] messages,
    FunctionContext context,
    CancellationToken cancellationToken)
{
    _logger.LogInformation("C# EventHub {functionName} trigger function
processing a request.", nameof(HandleCancellationCleanup));

    foreach (var message in messages)
    {
        if (cancellationToken.IsCancellationRequested)
        {
            _logger.LogInformation("A cancellation token was received,
taking precautionary actions.");
            // Take precautions like noting how far along you are with
processing the batch
            _logger.LogInformation("Precautionary activities complete.");
            break;
        }

        await Task.Delay(6000); // task delay to simulate message processing
        _logger.LogInformation("Message '{msg}' was processed.", message);
    }
}

```

Bindings

Bindings are defined by using attributes on methods, parameters, and return types. Bindings can provide data as strings, arrays, and serializable types, such as plain old

class objects (POCOs). For some binding extensions, you can also [bind to service-specific types](#) defined in service SDKs.

For HTTP triggers, see the [HTTP trigger](#) section.

For a complete set of reference samples using triggers and bindings with isolated worker process functions, see the [binding extensions reference sample ↗](#).

Input bindings

A function can have zero or more input bindings that can pass data to a function. Like triggers, input bindings are defined by applying a binding attribute to an input parameter. When the function executes, the runtime tries to get data specified in the binding. The data being requested is often dependent on information provided by the trigger using binding parameters.

Output bindings

To write to an output binding, you must apply an output binding attribute to the function method, which defines how to write to the bound service. The value returned by the method is written to the output binding. For example, the following example writes a string value to a message queue named `output-queue` by using an output binding:

C#

```
[Function(nameof(QueueFunction))]
[QueueOutput("output-queue")]
public string[] Run([QueueTrigger("input-queue")] Album myQueueItem,
FunctionContext context)
{
    // Use a string array to return more than one message.
    string[] messages = {
        $"Album name = {myQueueItem.Name}",
        $"Album songs = {myQueueItem.Songs.ToString()}};

    _logger.LogInformation("{msg1},{msg2}", messages[0], messages[1]);

    // Queue Output messages
    return messages;
}
```

Multiple output bindings

The data written to an output binding is always the return value of the function. If you need to write to more than one output binding, you must create a custom return type. This return type must have the output binding attribute applied to one or more properties of the class. The following example is an HTTP-triggered function using [ASP.NET Core integration](#) which writes to both the HTTP response and a queue output binding:

C#

```
public class MultipleOutputBindings
{
    private readonly ILogger<MultipleOutputBindings> _logger;

    public MultipleOutputBindings(ILogger<MultipleOutputBindings> logger)
    {
        _logger = logger;
    }

    [Function("MultipleOutputBindings")]
    public MyOutputType Run([HttpTrigger(AuthorizationLevel.Function,
    "post")] HttpRequest req)
    {
        _logger.LogInformation("C# HTTP trigger function processed a
request.");
        var myObject = new MyOutputType
        {
            Result = new OkObjectResult("C# HTTP trigger function processed
a request."),
            MessageText = "some output"
        };
        return myObject;
    }

    public class MyOutputType
    {
        [HttpPost]
        public IActionResult Result { get; set; }

        [QueueOutput("myQueue")]
        public string MessageText { get; set; }
    }
}
```

When using custom return types for multiple output bindings with ASP.NET Core integration, you must add the `[HttpPost]` attribute to the property that provides the result. The `HttpPost` attribute is available when using [SDK 1.17.3-preview2 or later](#) along with [version 3.2.0 or later of the HTTP extension](#) and [version 1.3.0 or later of the ASP.NET Core extension](#).

SDK types

For some service-specific binding types, binding data can be provided using types from service SDKs and frameworks. These provide more capability beyond what a serialized string or plain-old CLR object (POCO) can offer. To use the newer types, your project needs to be updated to use newer versions of core dependencies.

[+] Expand table

Dependency	Version requirement
Microsoft.Azure.Functions.Worker	1.18.0 or later
Microsoft.Azure.Functions.Worker.Sdk	1.13.0 or later

When testing SDK types locally on your machine, you also need to use [Azure Functions Core Tools](#), version 4.0.5000 or later. You can check your current version using the `func version` command.

Each trigger and binding extension also has its own minimum version requirement, which is described in the extension reference articles. The following service-specific bindings provide SDK types:

[+] Expand table

Service	Trigger	Input binding	Output binding
Azure Blobs	Generally Available	Generally Available	<i>SDK types not recommended.¹</i>
Azure Queues	Generally Available	<i>Input binding doesn't exist</i>	<i>SDK types not recommended.¹</i>
Azure Service Bus	Generally Available	<i>Input binding doesn't exist</i>	<i>SDK types not recommended.¹</i>
Azure Event Hubs	Generally Available	<i>Input binding doesn't exist</i>	<i>SDK types not recommended.¹</i>
Azure Cosmos DB	<i>SDK types not used²</i>	Generally Available	<i>SDK types not recommended.¹</i>
Azure Tables	<i>Trigger doesn't exist</i>	Generally Available	<i>SDK types not recommended.¹</i>
Azure Event Grid	Generally Available	<i>Input binding doesn't exist</i>	<i>SDK types not recommended.¹</i>

¹ For output scenarios in which you would use an SDK type, you should create and work with SDK clients directly instead of using an output binding. See [Register Azure clients](#) for a dependency injection example.

² The Cosmos DB trigger uses the [Azure Cosmos DB change feed](#) and exposes change feed items as JSON-serializable types. The absence of SDK types is by-design for this scenario.

ⓘ Note

When using [binding expressions](#) that rely on trigger data, SDK types for the trigger itself cannot be used.

HTTP trigger

[HTTP triggers](#) allow a function to be invoked by an HTTP request. There are two different approaches that can be used:

- An [ASP.NET Core integration model](#) that uses concepts familiar to ASP.NET Core developers
- A [built-in model](#), which doesn't require extra dependencies and uses custom types for HTTP requests and responses. This approach is maintained for backward compatibility with previous .NET isolated worker apps.

ASP.NET Core integration

This section shows how to work with the underlying HTTP request and response objects using types from ASP.NET Core including [HttpRequest](#), [HttpResponse](#), and [IActionResult](#). This model isn't available to [apps targeting .NET Framework](#), which should instead use the [built-in model](#).

ⓘ Note

Not all features of ASP.NET Core are exposed by this model. Specifically, the ASP.NET Core middleware pipeline and routing capabilities are not available. ASP.NET Core integration requires you to use updated packages.

To enable ASP.NET Core integration for HTTP:

1. Add a reference in your project to the [Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore](#) package, version 1.0.0 or later.

2. Update your project to use these specific package versions:

- [Microsoft.Azure.Functions.Worker.Sdk](#), version 1.11.0 or later
- [Microsoft.Azure.Functions.Worker](#), version 1.16.0 or later.

3. In your `Program.cs` file, update the host builder configuration to call

`ConfigureFunctionsWebApplication()`. This replaces

`ConfigureFunctionsWorkerDefaults()` if you would use that method otherwise. The following example shows a minimal setup without other customizations:

IHostBuilder

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWebApplication()
    .Build();

host.Run();
```

4. Update any existing HTTP-triggered functions to use the ASP.NET Core types. This example shows the standard `HttpRequest` and an `IActionResult` used for a simple "hello, world" function:

C#

```
[Function("HttpFunction")]
public IActionResult Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get")] HttpRequest req)
{
    return new OkObjectResult($"Welcome to Azure Functions,
{req.Query["name"]}");
}
```

JSON serialization with ASP.NET Core integration

ASP.NET Core has its own serialization layer, and it is not affected by [customizing general serialization configuration](#). To customize the serialization behavior used for your HTTP triggers, you need to include an `.AddMvc()` call as part of service registration. The returned `IMvcBuilder` can be used to modify ASP.NET Core's JSON serialization settings. The following example shows how to configure JSON.NET (`Newtonsoft.Json`) for serialization using this approach:

```
IHostBuilder

C#

using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWebApplication()
    .ConfigureServices(services =>
{
    services.AddApplicationInsightsTelemetryWorkerService();
    services.ConfigureFunctionsApplicationInsights();
    services.AddMvc().AddNewtonsoftJson();
})
.Build();
host.Run();
```

Built-in HTTP model

In the built-in model, the system translates the incoming HTTP request message into an `HttpRequestData` object that is passed to the function. This object provides data from the request, including `Headers`, `Cookies`, `Identities`, `URL`, and optionally a message `Body`. This object is a representation of the HTTP request but isn't directly connected to the underlying HTTP listener or the received message.

Likewise, the function returns an `HttpResponseData` object, which provides data used to create the HTTP response, including message `StatusCode`, `Headers`, and optionally a message `Body`.

The following example demonstrates the use of `HttpRequestData` and `HttpResponseData`:

```
C#
```

```
[Function(nameof(HttpFunction))]
public static HttpResponseData
```

```

Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)])
HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger(nameof(HttpFunction));
    logger.LogInformation("message logged");

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-8");
    response.WriteString("Welcome to .NET isolated worker !!");

    return response;
}

```

Logging

You can write to logs by using an [ILogger<T>](#) or [ILogger](#) instance. The logger can be obtained through [dependency injection](#) of an [ILogger<T>](#) or of an [ILoggerFactory](#):

C#

```

public class MyFunction {

    private readonly ILogger<MyFunction> _logger;

    public MyFunction(ILogger<MyFunction> logger) {
        _logger = logger;
    }

    [Function(nameof(MyFunction))]
    public void Run([BlobTrigger("samples-workitems/{name}", Connection =
    "")] string myBlob, string name)
    {
        _logger.LogInformation($"C# Blob trigger function Processed blob\n
Name: {name} \n Data: {myBlob}");
    }
}

```

The logger can also be obtained from a [FunctionContext](#) object passed to your function. Call the [GetLogger<T>](#) or [GetLogger](#) method, passing a string value that is the name for the category in which the logs are written. The category is usually the name of the specific function from which the logs are written. To learn more about categories, see the [monitoring article](#).

Use the methods of [ILogger<T>](#) and [ILogger](#) to write various log levels, such as [LogWarning](#) or [.LogError](#). To learn more about log levels, see the [monitoring article](#). You can customize the log levels for components added to your code by registering filters:

IHostBuilder

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices(services =>
{
    // Registers IHttpClientFactory.
    // By default this sends a lot of Information-level logs.
    services.AddHttpClient();
})
.ConfigureLogging(logging =>
{
    // Disable IHttpClientFactory Informational logs.
    // Note -- you can also remove the handler that does the
    logging:
https://github.com/aspnet/HttpClientFactory/issues/196#issuecomment-432755765
    logging.AddFilter("System.Net.Http.HttpClient",
LogLevel.Warning);
})
.Build();
```

As part of configuring your app in `Program.cs`, you can also define the behavior for how errors are surfaced to your logs. The default behavior depends on the type of builder you're using.

IHostBuilder

When you use a `HostBuilder`, by default, exceptions thrown by your code can end up wrapped in an `RpcException`. To remove this extra layer, set the `EnableUserCodeException` property to "true" as part of configuring the builder:

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults(builder => {}, options =>
{
    options.EnableUserCodeException = true;
```

```
    })
    .Build();

host.Run();
```

Application Insights

You can configure your isolated process application to emit logs directly to [Application Insights](#). This behavior replaces the default behavior of [relaying logs through the host](#), and is recommended because it gives you control over how those logs are emitted.

Install packages

To write logs directly to Application Insights from your code, add references to these packages in your project:

- [Microsoft.Azure.Functions.Worker.ApplicationInsights](#), version 1.0.0 or later.
- [Microsoft.ApplicationInsights.WorkerService](#).

You can run the following commands to add these references to your project:

.NET CLI

```
dotnet add package Microsoft.ApplicationInsights.WorkerService
dotnet add package Microsoft.Azure.Functions.Worker.ApplicationInsights
```

Configure startup

With the packages installed, you must call

`AddApplicationInsightsTelemetryWorkerService()` and

`ConfigureFunctionsApplicationInsights()` during service configuration in your `Program.cs` file, as in this example:

IHostBuilder

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
```

```
.ConfigureServices(services => {
    services.AddApplicationInsightsTelemetryWorkerService();
    services.ConfigureFunctionsApplicationInsights();
})
.Build();

host.Run();
```

The call to `ConfigureFunctionsApplicationInsights()` adds an `ITelemetryModule`, which listens to a Functions-defined `ActivitySource`. This creates the dependency telemetry required to support distributed tracing. To learn more about `AddApplicationInsightsTelemetryWorkerService()` and how to use it, see [Application Insights for Worker Service applications](#).

Managing log levels

Important

The Functions host and the isolated process worker have separate configuration for log levels, etc. Any [Application Insights configuration in host.json](#) will not affect the logging from the worker, and similarly, configuration made in your worker code will not impact logging from the host. You need to apply changes in both places if your scenario requires customization at both layers.

The rest of your application continues to work with `ILogger` and `ILogger<T>`. However, by default, the Application Insights SDK adds a logging filter that instructs the logger to capture only warnings and more severe logs. If you want to disable this behavior, remove the filter rule as part of service configuration:

IHostBuilder

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices(services => {
        services.AddApplicationInsightsTelemetryWorkerService();
        services.ConfigureFunctionsApplicationInsights();
```

```
        })
    .ConfigureLogging(logging =>
{
    logging.Services.Configure<LoggerFilterOptions>(options =>
    {
        LoggerFilterRule defaultRule =
options.Rules.FirstOrDefault(rule => rule.ProviderName
            ==
"Microsoft.Extensions.Logging.ApplicationInsights.ApplicationInsightsLog
gerProvider");
        if (defaultRule is not null)
        {
            options.Rules.Remove(defaultRule);
        }
    });
})
.Build();

host.Run();
```

Performance optimizations

This section outlines options you can enable that improve performance around [cold start](#).

In general, your app should use the latest versions of its core dependencies. At a minimum, you should update your project as follows:

1. Upgrade [Microsoft.Azure.Functions.Worker](#) to version 1.19.0 or later.
2. Upgrade [Microsoft.Azure.Functions.Worker.Sdk](#) to version 1.16.4 or later.
3. Add a framework reference to `Microsoft.AspNetCore.App`, unless your app targets .NET Framework.

The following snippet shows this configuration in the context of a project file:

XML

```
<ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker"
Version="1.21.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
Version="1.16.4" />
</ItemGroup>
```

Placeholders

Placeholders are a platform capability that improves cold start for apps targeting .NET 6 or later. To use this optimization, you must explicitly enable placeholders using these steps:

1. Update your project configuration to use the latest dependency versions, as detailed in the previous section.
2. Set the [WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED](#) application setting to 1, which you can do by using this [az functionapp config appsettings set](#) command:

Azure CLI

```
az functionapp config appsettings set -g <groupName> -n <appName> --  
settings 'WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED=1'
```

In this example, replace `<groupName>` with the name of the resource group, and replace `<appName>` with the name of your function app.

3. Make sure that the [netFrameworkVersion](#) property of the function app matches your project's target framework, which must be .NET 6 or later. You can do this by using this [az functionapp config set](#) command:

Azure CLI

```
az functionapp config set -g <groupName> -n <appName> --net-framework-  
version <framework>
```

In this example, also replace `<framework>` with the appropriate version string, such as `v8.0`, according to your target .NET version.

4. Make sure that your function app is configured to use a 64-bit process, which you can do by using this [az functionapp config set](#) command:

Azure CLI

```
az functionapp config set -g <groupName> -n <appName> --use-32bit-  
worker-process false
```

 **Important**

When setting the [WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED](#) to 1, all other function app configurations must be set correctly. Otherwise, your function app might fail to start.

Optimized executor

The function executor is a component of the platform that causes invocations to run. An optimized version of this component is enabled by default starting with version 1.16.2 of the SDK. No other configuration is required.

ReadyToRun

You can compile your function app as [ReadyToRun binaries](#). ReadyToRun is a form of ahead-of-time compilation that can improve startup performance to help reduce the effect of cold starts when running in a [Consumption plan](#). ReadyToRun is available in .NET 6 and later versions and requires [version 4.0 or later](#) of the Azure Functions runtime.

ReadyToRun requires you to build the project against the runtime architecture of the hosting app. **If these are not aligned, your app will encounter an error at startup.** Select your runtime identifier from this table:

[+] [Expand table](#)

Operating System	App is 32-bit ¹	Runtime identifier
Windows	True	win-x86
Windows	False	win-x64
Linux	True	N/A (not supported)
Linux	False	linux-x64

¹ Only 64-bit apps are eligible for some other performance optimizations.

To check if your Windows app is 32-bit or 64-bit, you can run the following CLI command, substituting `<group_name>` with the name of your resource group and `<app_name>` with the name of your application. An output of "true" indicates that the app is 32-bit, and "false" indicates 64-bit.

Azure CLI

```
az functionapp config show -g <group_name> -n <app_name> --query  
"use32BitWorkerProcess"
```

You can change your application to 64-bit with the following command, using the same substitutions:

Azure CLI

```
az functionapp config set -g <group_name> -n <app_name> --use-32bit-worker-process false`
```

To compile your project as ReadyToRun, update your project file by adding the `<PublishReadyToRun>` and `<RuntimeIdentifier>` elements. The following example shows a configuration for publishing to a Windows 64-bit function app.

XML

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <AzureFunctionsVersion>v4</AzureFunctionsVersion>
  <RuntimeIdentifier>win-x64</RuntimeIdentifier>
  <PublishReadyToRun>true</PublishReadyToRun>
</PropertyGroup>
```

If you don't want to set the `<RuntimeIdentifier>` as part of the project file, you can also configure this as part of the publishing gesture itself. For example, with a Windows 64-bit function app, the .NET CLI command would be:

.NET CLI

```
dotnet publish --runtime win-x64
```

In Visual Studio, the **Target Runtime** option in the publish profile should be set to the correct runtime identifier. When set to the default value of **Portable**, ReadyToRun isn't used.

Deploy to Azure Functions

When you deploy your function code project to Azure, it must run in either a function app or in a Linux container. The function app and other required Azure resources must exist before you deploy your code.

You can also deploy your function app in a Linux container. For more information, see [Working with containers and Azure Functions](#).

Create Azure resources

You can create your function app and other required resources in Azure using one of these methods:

- [Visual Studio](#): Visual Studio can create resources for you during the code publishing process.
- [Visual Studio Code](#): Visual Studio Code can connect to your subscription, create the resources needed by your app, and then publish your code.
- [Azure CLI](#): You can use the Azure CLI to create the required resources in Azure.
- [Azure PowerShell](#): You can use Azure PowerShell to create the required resources in Azure.
- [Deployment templates](#): You can use ARM templates and Bicep files to automate the deployment of the required resources to Azure. Make sure your template includes any [required settings](#).
- [Azure portal](#): You can create the required resources in the [Azure portal](#) ↗.

Publish your application

After creating your function app and other required resources in Azure, you can deploy the code project to Azure using one of these methods:

- [Visual Studio](#): Simple manual deployment during development.
- [Visual Studio Code](#): Simple manual deployment during development.
- [Azure Functions Core Tools](#): Deploy project file from the command line.
- [Continuous deployment](#): Useful for ongoing maintenance, frequently to a [staging slot](#).
- [Deployment templates](#): You can use ARM templates or Bicep files to automate package deployments.

For more information, see [Deployment technologies in Azure Functions](#).

Deployment payload

Many of the deployment methods make use of a zip archive. If you're creating the zip archive yourself, it must follow the structure outlined in this section. If it doesn't, your app may experience errors at startup.

The deployment payload should match the output of a `dotnet publish` command, though without the enclosing parent folder. The zip archive should be made from the following files:

- `.azurefunctions/`
- `extensions.json`
- `functions.metadata`
- `host.json`

- `worker.config.json`
- Your project executable (a console app)
- Other supporting files and directories peer to that executable

These files are generated by the build process, and they aren't meant to be edited directly.

When preparing a zip archive for deployment, you should only compress the contents of the output directory, not the enclosing directory itself. When the archive is extracted into the current working directory, the files listed above need to be immediately visible.

Deployment requirements

There are a few requirements for running .NET functions in the isolated worker model in Azure, depending on the operating system:

Windows

- `FUNCTIONS_WORKER_RUNTIME` must be set to a value of `dotnet-isolated`.
- `netFrameworkVersion` must be set to the desired version.

When you create your function app in Azure using the methods in the previous section, these required settings are added for you. When you create these resources [by using ARM templates or Bicep files for automation](#), you must make sure to set them in the template.

Debugging

When running locally using Visual Studio or Visual Studio Code, you're able to debug your .NET isolated worker project as normal. However, there are two debugging scenarios that don't work as expected.

Remote Debugging using Visual Studio

Because your isolated worker process app runs outside the Functions runtime, you need to attach the remote debugger to a separate process. To learn more about debugging using Visual Studio, see [Remote Debugging](#).

Debugging when targeting .NET Framework

If your isolated project targets .NET Framework 4.8, the current preview scope requires manual steps to enable debugging. These steps aren't required if using another target framework.

Your app should start with a call to `FunctionsDebugger.Enable();` as its first operation. This occurs in the `Main()` method before initializing a HostBuilder. Your `Program.cs` file should look similar to this:

```
IHostBuilder  
C#  
  
using System;  
using System.Diagnostics;  
using Microsoft.Extensions.Hosting;  
using Microsoft.Azure.Functions.Worker;  
using NetFxWorker;  
  
namespace MyDotnetFrameworkProject  
{  
    internal class Program  
    {  
        static void Main(string[] args)  
        {  
            FunctionsDebugger.Enable();  
  
            var host = new HostBuilder()  
                .ConfigureFunctionsWorkerDefaults()  
                .Build();  
  
            host.Run();  
        }  
    }  
}
```

Next, you need to manually attach to the process using a .NET Framework debugger. Visual Studio doesn't do this automatically for isolated worker process .NET Framework apps yet, and the "Start Debugging" operation should be avoided.

In your project directory (or its build output directory), run:

```
Azure CLI  
  
func host start --dotnet-isolated-debug
```

This starts your worker, and the process stops with the following message:

Azure CLI

```
Azure Functions .NET Worker (PID: <process id>) initialized in debug mode.  
Waiting for debugger to attach...
```

Where `<process id>` is the ID for your worker process. You can now use Visual Studio to manually attach to the process. For instructions on this operation, see [How to attach to a running process](#).

After the debugger is attached, the process execution resumes, and you'll be able to debug.

Preview .NET versions

Before a generally available release, a .NET version might be released in a *Preview* or *Go-live* state. See the [.NET Official Support Policy](#) for details on these states.

While it might be possible to target a given release from a local Functions project, function apps hosted in Azure might not have that release available. Azure Functions can only be used with Preview or Go-live releases noted in this section.

Azure Functions currently can be used with the following "Preview" or "Go-live" .NET releases:

[] [Expand table](#)

Operating system	.NET preview version
Windows	.NET 9 Preview 6 ^{1, 2}
Linux	.NET 9 RC2 ^{1, 3}

¹ To successfully target .NET 9, your project needs to reference the [2.x versions of the core packages](#). If using Visual Studio, .NET 9 requires version 17.12 or later.

² Support for Windows might not appear in some clients during the preview period.

³ .NET 9 is not yet supported on the Flex Consumption SKU.

See [Supported versions](#) for a list of generally available releases that you can use.

Using a preview .NET SDK

To use Azure Functions with a preview version of .NET, you need to update your project by:

1. Installing the relevant .NET SDK version in your development
2. Changing the `TargetFramework` setting in your `.csproj` file

When you deploy to your function app in Azure, you also need to ensure that the framework is made available to the app. During the preview period, some tools and experiences may not surface the new preview version as an option. If you don't see the preview version included in the Azure portal, for example, you can use the REST API, Bicep templates, or the Azure CLI to configure the version manually.

Windows

For apps hosted on Windows, use the following Azure CLI command. Replace `<groupName>` with the name of the resource group, and replace `<appName>` with the name of your function app. Replace `<framework>` with the appropriate version string, such as `v8.0`.

Azure CLI

```
az functionapp config set -g <groupName> -n <appName> --net-framework-version <framework>
```

Considerations for using .NET preview versions

Keep these considerations in mind when using Functions with preview versions of .NET:

- When you author your functions in Visual Studio, you must use [Visual Studio Preview](#), which supports building Azure Functions projects with .NET preview SDKs.
- Make sure you have the latest Functions tools and templates. To update your tools:
 1. Navigate to **Tools > Options**, choose **Azure Functions** under **Projects and Solutions**.
 2. Select **Check for updates** and install updates as prompted.
- During a preview period, your development environment might have a more recent version of the .NET preview than the hosted service. This can cause your function app to fail when deployed. To address this, you can specify the version of the SDK to use in [global.json](#).

1. Run the `dotnet --list-sdks` command and note the preview version you're currently using during local development.
2. Run the `dotnet new globaljson --sdk-version <SDK_VERSION> --force` command, where `<SDK_VERSION>` is the version you're using locally. For example, `dotnet new globaljson --sdk-version dotnet-sdk-8.0.100-preview.7.23376.3 --force` causes the system to use the .NET 8 Preview 7 SDK when building your project.

 Note

Because of the just-in-time loading of preview frameworks, function apps running on Windows can experience increased cold start times when compared against earlier GA versions.

Next steps

[Learn more about best practices for Azure Functions](#)

[Migrate .NET apps to the isolated worker model](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Develop C# class library functions using Azure Functions

Article • 07/07/2024

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

This article is an introduction to developing Azure Functions by using C# in .NET class libraries. These class libraries are used to run *in-process with the Functions runtime*. Your .NET functions can alternatively run *_isolated* from the Functions *runtime*, which offers several advantages. To learn more, see [the isolated worker model](#). For a comprehensive comparison between these two models, see [Differences between the in-process model and the isolated worker model](#).

ⓘ Important

This article supports .NET class library functions that run in-process with the runtime. Your C# functions can also run out-of-process and isolated from the Functions runtime. The isolated worker process model is the only way to run non-LTS versions of .NET and .NET Framework apps in current versions of the Functions runtime. To learn more, see [.NET isolated worker process functions](#). For a comprehensive comparison between isolated worker process and in-process .NET Functions, see [Differences between in-process and isolate worker process .NET Azure Functions](#).

As a C# developer, you may also be interested in one of the following articles:

[+] Expand table

Getting started	Concepts	Guided learning/samples
<ul style="list-style-type: none">Using Visual StudioUsing Visual Studio CodeUsing command line tools	<ul style="list-style-type: none">Hosting optionsPerformance considerationsVisual Studio developmentDependency injection	<ul style="list-style-type: none">Create serverless applicationsC# samples

Azure Functions supports C# and C# script programming languages. If you're looking for guidance on [using C# in the Azure portal](#), see [C# script \(.csx\) developer reference](#).

Supported versions

Versions of the Functions runtime support specific versions of .NET. To learn more about Functions versions, see [Azure Functions runtime versions overview](#). Version support also depends on whether your functions run in-process or isolated worker process.

ⓘ Note

To learn how to change the Functions runtime version used by your function app, see [view and update the current runtime version](#).

The following table shows the highest level of .NET or .NET Framework that can be used with a specific version of Functions.

[] Expand table

Functions runtime version	Isolated worker model	In-process model ⁵
Functions 4.x ¹	.NET 8.0 .NET 6.0 ² .NET Framework 4.8 ³	.NET 8.0 .NET 6.0 ²
Functions 1.x ⁴	n/a	.NET Framework 4.8

¹ .NET 7 was previously supported on the isolated worker model but reached the [end of official support](#) on May 14, 2024.

² .NET 6 reaches the [end of official support](#) on November 12, 2024.

³ The build process also requires the [.NET SDK](#).

⁴ Support ends for version 1.x of the Azure Functions runtime on September 14, 2026. For more information, see [this support announcement](#). For continued full support, you should [migrate your apps to version 4.x](#).

⁵ Support ends for the in-process model on November 10, 2026. For more information, see [this support announcement](#). For continued full support, you should [migrate your apps to the isolated worker model](#).

For the latest news about Azure Functions releases, including the removal of specific older minor versions, monitor [Azure App Service announcements](#).

Updating to target .NET 8

ⓘ Note

Targeting .NET 8 with the in-process model is not yet enabled for Linux, for apps hosted in App Service Environments, or for apps in sovereign clouds. Updates will be communicated on [this tracking thread on GitHub](#).

Apps using the in-process model can target .NET 8 by following the steps outlined in this section. However, if you choose to exercise this option, you should still begin planning your [migration to the isolated worker model](#) in advance of [support ending for the in-process model on November 10, 2026](#).

Many apps can change the configuration of the function app in Azure without updates to code or redeployment. To run .NET 8 with the in-process model, three configurations are required:

- The application setting `FUNCTIONS_WORKER_RUNTIME` must be set with the value "dotnet".
- The application setting `FUNCTIONS_EXTENSION_VERSION` must be set with the value "~4".
- The application setting `FUNCTIONS_INPROC_NET8_ENABLED` must be set with the value "1".
- You must [update the stack configuration](#) to reference .NET 8.

Support for .NET 8 still uses version 4.x of the Functions runtime, and no change to the configured runtime version is required.

To update your local project, first make sure you are using the latest versions of local tools. Then ensure that the project references [version 4.4.0 or later of Microsoft.NET.Sdk.Functions](#). You can then change your `TargetFramework` to "net8.0". You must also update `local.settings.json` to include both `FUNCTIONS_WORKER_RUNTIME` set to "dotnet" and `FUNCTIONS_INPROC_NET8_ENABLED` set to "1".

The following is an example of a minimal `local.settings.json` file with these changes:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <AzureFunctionsVersion>V4</AzureFunctionsVersion>
  </PropertyGroup>
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Sdk.Functions" Version="4.4.0"
/>
</ItemGroup>
<ItemGroup>
  <None Update="host.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
  <None Update="local.settings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <CopyToPublishDirectory>Never</CopyToPublishDirectory>
  </None>
</ItemGroup>
</Project>
```

The following is an example of a minimal `local.settings.json` file with these changes:

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_INPROC_NET8_ENABLED": "1",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet"
  }
}
```

If your app uses [Microsoft.Azure.DurableTask.Netherite.AzureFunctions](#), ensure it targets version 1.5.3 or later. Due to a behavior change in .NET 8, apps with older versions of the package will throw an ambiguous constructor exception.

You might need to make other changes to your app based on the version support of its other dependencies.

Functions class library project

In Visual Studio, the **Azure Functions** project template creates a C# class library project that contains the following files:

- [host.json](#) - stores configuration settings that affect all functions in the project when running locally or in Azure.
- [local.settings.json](#) - stores app settings and connection strings that are used when running locally. This file contains secrets and isn't published to your function app in Azure. Instead, [add app settings to your function app](#).

When you build the project, a folder structure that looks like the following example is generated in the build output directory:

```
<framework.version>
| - bin
| - MyFirstFunction
| | - function.json
| - MySecondFunction
| | - function.json
| - host.json
```

This directory is what gets deployed to your function app in Azure. The binding extensions required in [version 2.x](#) of the Functions runtime are [added to the project as NuGet packages](#).

ⓘ Important

The build process creates a *function.json* file for each function. This *function.json* file is not meant to be edited directly. You can't change binding configuration or disable the function by editing this file. To learn how to disable a function, see [How to disable functions](#).

Methods recognized as functions

In a class library, a function is a method with a `FunctionName` and a trigger attribute, as shown in the following example:

C#

```
public static class SimpleExample
{
    [FunctionName("QueueTrigger")]
    public static void Run(
        [QueueTrigger("myqueue-items")] string myQueueItem,
        ILogger log)
    {
        log.LogInformation($"C# function processed: {myQueueItem}");
    }
}
```

The `FunctionName` attribute marks the method as a function entry point. The name must be unique within a project, start with a letter and only contain letters, numbers, `_`, and

- , up to 127 characters in length. Project templates often create a method named `Run`, but the method name can be any valid C# method name. The above example shows a static method being used, but functions aren't required to be static.

The trigger attribute specifies the trigger type and binds input data to a method parameter. The example function is triggered by a queue message, and the queue message is passed to the method in the `myQueueItem` parameter.

Method signature parameters

The method signature may contain parameters other than the one used with the trigger attribute. Here are some of the other parameters that you can include:

- [Input and output bindings](#) marked as such by decorating them with attributes.
- An `ILogger` or `TraceWriter` (version 1.x-only) parameter for [logging](#).
- A `CancellationToken` parameter for [graceful shutdown](#).
- [Binding expressions](#) parameters to get trigger metadata.

The order of parameters in the function signature doesn't matter. For example, you can put trigger parameters before or after other bindings, and you can put the logger parameter before or after trigger or binding parameters.

Output bindings

A function can have zero or multiple output bindings defined by using output parameters.

The following example modifies the preceding one by adding an output queue binding named `myQueueItemCopy`. The function writes the contents of the message that triggers the function to a new message in a different queue.

```
C#  
  
public static class SimpleExampleWithOutput  
{  
    [FunctionName("CopyQueueMessage")]  
    public static void Run(  
        [QueueTrigger("myqueue-items-source")] string myQueueItem,  
        [Queue("myqueue-items-destination")] out string myQueueItemCopy,  
        ILogger log)  
    {  
        log.LogInformation($"CopyQueueMessage function processed:  
{myQueueItem}");  
        myQueueItemCopy = myQueueItem;  
    }  
}
```

```
    }  
}
```

Values assigned to output bindings are written when the function exits. You can use more than one output binding in a function by simply assigning values to multiple output parameters.

The binding reference articles ([Storage queues](#), for example) explain which parameter types you can use with trigger, input, or output binding attributes.

Binding expressions example

The following code gets the name of the queue to monitor from an app setting, and it gets the queue message creation time in the `insertionTime` parameter.

C#

```
public static class BindingExpressionsExample  
{  
    [FunctionName("LogQueueMessage")]  
    public static void Run(  
        [QueueTrigger("%queueappsetting%")] string myQueueItem,  
        DateTimeOffset insertionTime,  
        ILogger log)  
    {  
        log.LogInformation($"Message content: {myQueueItem}");  
        log.LogInformation($"Created at: {insertionTime}");  
    }  
}
```

Autogenerated function.json

The build process creates a `function.json` file in a function folder in the build folder. As noted earlier, this file isn't meant to be edited directly. You can't change binding configuration or disable the function by editing this file.

The purpose of this file is to provide information to the scale controller to use for [scaling decisions on the Consumption plan](#). For this reason, the file only has trigger info, not input/output bindings.

The generated `function.json` file includes a `configurationSource` property that tells the runtime to use .NET attributes for bindings, rather than `function.json` configuration. Here's an example:

JSON

```
{  
  "generatedBy": "Microsoft.NET.Sdk.Functions-1.0.0.0",  
  "configurationSource": "attributes",  
  "bindings": [  
    {  
      "type": "queueTrigger",  
      "queueName": "%input-queue-name%",  
      "name": "myQueueItem"  
    }  
  ],  
  "disabled": false,  
  "scriptFile": "..\bin\FunctionApp1.dll",  
  "entryPoint": "FunctionApp1.QueueTrigger.Run"  
}
```

Microsoft.NET.Sdk.Functions

The `function.json` file generation is performed by the NuGet package [Microsoft.NET.Sdk.Functions](#).

The following example shows the relevant parts of the `.csproj` files that have different target frameworks of the same `Sdk` package:

v4.x

XML

```
<PropertyGroup>  
  <TargetFramework>net8.0</TargetFramework>  
  <AzureFunctionsVersion>v4</AzureFunctionsVersion>  
</PropertyGroup>  
<ItemGroup>  
  <PackageReference Include="Microsoft.NET.Sdk.Functions"  
    Version="4.4.0" />  
</ItemGroup>
```

Among the `Sdk` package dependencies are triggers and bindings. A 1.x project refers to 1.x triggers and bindings because those triggers and bindings target the .NET Framework, while 4.x triggers and bindings target .NET Core.

The `Sdk` package also depends on [Newtonsoft.Json](#), and indirectly on [WindowsAzure.Storage](#). These dependencies make sure that your project uses the versions of those packages that work with the Functions runtime version that the project

targets. For example, `Newtonsoft.Json` has version 11 for .NET Framework 4.6.1, but the Functions runtime that targets .NET Framework 4.6.1 is only compatible with `Newtonsoft.Json` 9.0.1. So your function code in that project also has to use `Newtonsoft.Json` 9.0.1.

The source code for `Microsoft.NET.Sdk.Functions` is available in the GitHub repo [azure-functions-vs-build-sdk](#).

Local runtime version

Visual Studio uses the [Azure Functions Core Tools](#) to run Functions projects on your local computer. The Core Tools is a command-line interface for the Functions runtime.

If you install the Core Tools using the Windows installer (MSI) package or by using npm, it doesn't affect the Core Tools version used by Visual Studio. For the Functions runtime version 1.x, Visual Studio stores Core Tools versions in `%USERPROFILE%\AppData\Local\Azure.Functions.Cli` and uses the latest version stored there. For Functions 4.x, the Core Tools are included in the [Azure Functions and Web Jobs Tools](#) extension. For Functions 1.x, you can see what version is being used in the console output when you run a Functions project:

terminal

```
[3/1/2018 9:59:53 AM] Starting Host (HostId=contoso2-1518597420,  
Version=2.0.11353.0, ProcessId=22020, Debug=False, Attempt=0,  
FunctionsExtensionVersion=)
```

ReadyToRun

You can compile your function app as [ReadyToRun binaries](#). ReadyToRun is a form of ahead-of-time compilation that can improve startup performance to help reduce the impact of [cold-start](#) when running in a [Consumption plan](#).

ReadyToRun is available in .NET 6 and later versions and requires [version 4.0 of the Azure Functions runtime](#).

To compile your project as ReadyToRun, update your project file by adding the `<PublishReadyToRun>` and `<RuntimeIdentifier>` elements. The following is the configuration for publishing to a Windows 32-bit function app.

XML

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <AzureFunctionsVersion>v4</AzureFunctionsVersion>
  <PublishReadyToRun>true</PublishReadyToRun>
  <RuntimeIdentifier>win-x86</RuntimeIdentifier>
</PropertyGroup>
```

ⓘ Important

Starting in .NET 6, support for Composite ReadyToRun compilation has been added. Check out [ReadyToRun Cross platform and architecture restrictions](#).

You can also build your app with ReadyToRun from the command line. For more information, see the `-p:PublishReadyToRun=true` option in [dotnet publish](#).

Supported types for bindings

Each binding has its own supported types; for instance, a blob trigger attribute can be applied to a string parameter, a POCO parameter, a `CloudBlockBlob` parameter, or any of several other supported types. The [binding reference article for blob bindings](#) lists all supported parameter types. For more information, see [Triggers and bindings](#) and the [binding reference docs for each binding type](#).

ⓘ Tip

If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`. For more information, see [How to manage connections in Azure Functions](#).

Binding to method return value

You can use a method return value for an output binding, by applying the attribute to the method return value. For examples, see [Triggers and bindings](#).

Use the return value only if a successful function execution always results in a return value to pass to the output binding. Otherwise, use `ICollector` or `IAsyncCollector`, as shown in the following section.

Writing multiple output values

To write multiple values to an output binding, or if a successful function invocation might not result in anything to pass to the output binding, use the [ICollector](#) or [IAsyncCollector](#) types. These types are write-only collections that are written to the output binding when the method completes.

This example writes multiple queue messages into the same queue using [ICollector](#):

```
C#  
  
public static class ICollectorExample  
{  
    [FunctionName("CopyQueueMessageICollector")]  
    public static void Run(  
        [QueueTrigger("myqueue-items-source-3")] string myQueueItem,  
        [Queue("myqueue-items-destination")] ICollector<string>  
myDestinationQueue,  
        ILogger log)  
    {  
        log.LogInformation($"C# function processed: {myQueueItem}");  
        myDestinationQueue.Add($"Copy 1: {myQueueItem}");  
        myDestinationQueue.Add($"Copy 2: {myQueueItem}");  
    }  
}
```

Async

To make a function [asynchronous](#), use the `async` keyword and return a `Task` object.

```
C#  
  
public static class AsyncExample  
{  
    [FunctionName("BlobCopy")]  
    public static async Task RunAsync(  
        [BlobTrigger("sample-images/{blobName}")] Stream blobInput,  
        [Blob("sample-images-copies/{blobName}", FileAccess.Write)] Stream  
blobOutput,  
        CancellationToken token,  
        ILogger log)  
    {  
        log.LogInformation($"BlobCopy function processed.");  
        await blobInput.CopyToAsync(blobOutput, 4096, token);  
    }  
}
```

You can't use `out` parameters in `async` functions. For output bindings, use the [function return value](#) or a [collector object](#) instead.

Cancellation tokens

A function can accept a [CancellationToken](#) parameter, which enables the operating system to notify your code when the function is about to be terminated. You can use this notification to make sure the function doesn't terminate unexpectedly in a way that leaves data in an inconsistent state.

Consider the case when you have a function that processes messages in batches. The following Azure Service Bus-triggered function processes an array of [ServiceBusReceivedMessage](#) objects, which represents a batch of incoming messages to be processed by a specific function invocation:

C#

```
using Azure.Messaging.ServiceBus;
using System.Threading;

namespace ServiceBusCancellationToken
{
    public static class servicebus
    {
        [FunctionName("servicebus")]
        public static void Run([ServiceBusTrigger("csharpguitar", Connection
= "SB_CONN")]
                               ServiceBusReceivedMessage[] messages, CancellationToken
cancellationToken, ILogger log)
        {
            try
            {
                foreach (var message in messages)
                {
                    if (cancellationToken.IsCancellationRequested)
                    {
                        log.LogInformation("A cancellation token was
received. Taking precautionary actions.");
                        //Take precautions like noting how far along you are
with processing the batch
                        log.LogInformation("Precautionary activities --
complete--.");
                        break;
                    }
                    else
                    {
                        //business logic as usual
                        log.LogInformation($"Message: {message} was
processed.");
                    }
                }
            }
            catch (Exception ex)
            {
```

```
        log.LogInformation($"Something unexpected happened:  
{ex.Message}");  
    }  
}  
}
```

Logging

In your function code, you can write output to logs that appear as traces in Application Insights. The recommended way to write to the logs is to include a parameter of type [ILogger](#), which is typically named `log`. Version 1.x of the Functions runtime used `TraceWriter`, which also writes to Application Insights, but doesn't support structured logging. Don't use `Console.WriteLine` to write your logs, since this data isn't captured by Application Insights.

ILogger

In your function definition, include an [ILogger](#) parameter, which supports [structured logging](#).

With an `ILogger` object, you call `Log<level>` extension methods on `ILogger` to create logs. The following code writes `Information` logs with category `Function`.

```
<YOUR_FUNCTION_NAME>.User.:
```

```
cs
```

```
public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,  
ILogger logger)  
{  
    logger.LogInformation("Request for item with key={itemKey}.", id);
```

To learn more about how Functions implements `ILogger`, see [Collecting telemetry data](#). Categories prefixed with `Function` assume you're using an `ILogger` instance. If you choose to instead use an `ILogger<T>`, the category name may instead be based on `T`.

Structured logging

The order of placeholders, not their names, determines which parameters are used in the log message. Suppose you have the following code:

```
C#
```

```
string partitionKey = "partitionKey";
string rowKey = "rowKey";
logger.LogInformation("partitionKey={partitionKey}, rowKey={rowKey}",
partitionKey, rowKey);
```

If you keep the same message string and reverse the order of the parameters, the resulting message text would have the values in the wrong places.

Placeholders are handled this way so that you can do structured logging. Application Insights stores the parameter name-value pairs and the message string. The result is that the message arguments become fields that you can query on.

If your logger method call looks like the previous example, you can query the field `customDimensions.prop__rowKey`. The `prop__` prefix is added to ensure there are no collisions between fields the runtime adds and fields your function code adds.

You can also query on the original message string by referencing the field `customDimensions.prop__{OriginalFormat}`.

Here's a sample JSON representation of `customDimensions` data:

JSON

```
{
  "customDimensions": {
    "prop__{OriginalFormat}":"C# Queue trigger function processed:
{message}",
    "Category":"Function",
    "LogLevel":"Information",
    "prop__message":"c9519cbf-b1e6-4b9b-bf24-cb7d10b1bb89"
  }
}
```

Log custom telemetry

There's a Functions-specific version of the Application Insights SDK that you can use to send custom telemetry data from your functions to Application Insights:

[Microsoft.Azure.WebJobs.Logging.ApplicationInsights](#). Use the following command from the command prompt to install this package:

Command

Windows Command Prompt

```
dotnet add package Microsoft.Azure.WebJobs.Logging.ApplicationInsights -  
-version <VERSION>
```

In this command, replace `<VERSION>` with a version of this package that supports your installed version of [Microsoft.Azure.WebJobs](#).

The following C# examples uses the [custom telemetry API](#). The example is for a .NET class library, but the Application Insights code is the same for C# script.

v4.x

Version 2.x and later versions of the runtime use newer features in Application Insights to automatically correlate telemetry with the current operation. There's no need to manually set the operation `Id`, `ParentId`, or `Name` fields.

cs

```
using System;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.AspNetCore.Http;  
using Microsoft.Extensions.Logging;  
  
using Microsoft.ApplicationInsights;  
using Microsoft.ApplicationInsights.DataContracts;  
using Microsoft.ApplicationInsights.Extensibility;  
using System.Linq;  
  
namespace functionapp0915  
{  
    public class HttpTrigger2  
    {  
        private readonly TelemetryClient telemetryClient;  
  
        /// Using dependency injection will guarantee that you use the  
        /// same configuration for telemetry collected automatically and manually.  
        public HttpTrigger2(TelemetryConfiguration  
telemetryConfiguration)  
        {  
            this.telemetryClient = new  
TelemetryClient(telemetryConfiguration);  
        }  
  
        [FunctionName("HttpTrigger2")]  
        public Task<IActionResult> Run(  
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
```

```

    null)];
        HttpRequest req, ExecutionContext context, ILogger log)
    {
        log.LogInformation("C# HTTP trigger function processed a
request.");
        DateTime start = DateTime.UtcNow;

        // Parse query parameter
        string name = req.Query
            .FirstOrDefault(q => string.Compare(q.Key, "name", true)
== 0)
            .Value;

        // Write an event to the customEvents table.
        var evt = new EventTelemetry("Function called");
        evt.Context.User.Id = name;
        this.telemetryClient.TrackEvent(evt);

        // Generate a custom metric, in this case let's use
ContentLength.

        this.telemetryClient.GetMetric("contentLength").TrackValue(req.ContentLe
ngth);

        // Log a custom dependency in the dependencies table.
        var dependency = new DependencyTelemetry
        {
            Name = "GET api/planets/1/",
            Target = "swapi.co",
            Data = "https://swapi.co/api/planets/1/",
            Timestamp = start,
            Duration = DateTime.UtcNow - start,
            Success = true
        };
        dependency.Context.User.Id = name;
        this.telemetryClient.TrackDependency(dependency);

        return Task.FromResult<IActionResult>(new OkResult());
    }
}
}

```

In this example, the custom metric data gets aggregated by the host before being sent to the customMetrics table. To learn more, see the [GetMetric](#) documentation in Application Insights.

When running locally, you must add the `APPINSIGHTS_INSTRUMENTATIONKEY` setting, with the Application Insights key, to the [local.settings.json](#) file.

Don't call `TrackRequest` or `StartOperation<RequestTelemetry>` because you'll see duplicate requests for a function invocation. The Functions runtime automatically tracks

requests.

Don't set `telemetryClient.Context.Operation.Id`. This global setting causes incorrect correlation when many functions are running simultaneously. Instead, create a new telemetry instance (`DependencyTelemetry`, `EventTelemetry`) and modify its `Context` property. Then pass in the telemetry instance to the corresponding `Track` method on `TelemetryClient` (`TrackDependency()`, `TrackEvent()`, `TrackMetric()`). This method ensures that the telemetry has the correct correlation details for the current function invocation.

Testing functions

The following articles show how to run an in-process C# class library function locally for testing purposes:

- [Visual Studio](#)
- [Visual Studio Code](#)
- [Command line](#)

Environment variables

To get an environment variable or an app setting value, use

`System.Environment.GetEnvironmentVariable`, as shown in the following code example:

C#

```
public static class EnvironmentVariablesExample
{
    [FunctionName("GetEnvironmentVariables")]
    public static void Run([TimerTrigger("0 */5 * * *")]TimerInfo myTimer,
    ILogger log)
    {
        log.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");
        log.LogInformation(GetEnvironmentVariable("AzureWebJobsStorage"));
        log.LogInformation(GetEnvironmentVariable("WEBSITE_SITE_NAME"));
    }

    private static string GetEnvironmentVariable(string name)
    {
        return name + ": " +
            System.Environment.GetEnvironmentVariable(name,
            EnvironmentVariableTarget.Process);
    }
}
```

App settings can be read from environment variables both when developing locally and when running in Azure. When developing locally, app settings come from the `Values` collection in the `local.settings.json` file. In both environments, local and Azure, `GetEnvironmentVariable("<app setting name>")` retrieves the value of the named app setting. For instance, when you're running locally, "My Site Name" would be returned if your `local.settings.json` file contains `{ "Values": { "WEBSITE_SITE_NAME": "My Site Name" } }`.

The `System.Configuration.ConfigurationManager.AppSettings` property is an alternative API for getting app setting values, but we recommend that you use `GetEnvironmentVariable` as shown here.

Binding at runtime

In C# and other .NET languages, you can use an [imperative ↗](#) binding pattern, as opposed to the [declarative ↗](#) bindings in attributes. Imperative binding is useful when binding parameters need to be computed at runtime rather than design time. With this pattern, you can bind to supported input and output bindings on-the-fly in your function code.

Define an imperative binding as follows:

- **Do not** include an attribute in the function signature for your desired imperative bindings.
- Pass in an input parameter `Binder binder` or `IBinder binder`.
- Use the following C# pattern to perform the data binding.

cs

```
using (var output = await binder.BindAsync<T>(new
    BindingTypeAttribute(...)))
{
    ...
}
```

`BindingTypeAttribute` is the .NET attribute that defines your binding, and `T` is an input or output type that's supported by that binding type. `T` can't be an `out` parameter type (such as `out JObject`). For example, the Mobile Apps table output binding supports [six output types ↗](#), but you can only use `ICollector<T>` or `IAsyncCollector<T>` with imperative binding.

Single attribute example

The following example code creates a [Storage blob output binding](#) with blob path that's defined at run time, then writes a string to the blob.

```
cs

public static class IBinderExample
{
    [FunctionName("CreateBlobUsingBinder")]
    public static void Run(
        [QueueTrigger("myqueue-items-source-4")] string myQueueItem,
        IBinder binder,
        ILogger log)
    {
        log.LogInformation($"CreateBlobUsingBinder function processed:
{myQueueItem}");
        using (var writer = binder.Bind<TextWriter>(new BlobAttribute(
            $"samples-output/{myQueueItem}", FileAccess.Write)))
        {
            writer.WriteLine("Hello World!");
        };
    }
}
```

[BlobAttribute](#) defines the [Storage blob](#) input or output binding, and [TextWriter](#) is a supported output binding type.

Multiple attributes example

The preceding example gets the app setting for the function app's main Storage account connection string (which is [AzureWebJobsStorage](#)). You can specify a custom app setting to use for the Storage account by adding the [StorageAccountAttribute](#) and passing the attribute array into [BindAsync<T>\(\)](#). Use a [Binder](#) parameter, not [IBinder](#). For example:

```
cs

public static class IBinderExampleMultipleAttributes
{
    [FunctionName("CreateBlobInDifferentStorageAccount")]
    public async static Task RunAsync(
        [QueueTrigger("myqueue-items-source-binder2")] string
myQueueItem,
        Binder binder,
        ILogger log)
    {
        log.LogInformation($"CreateBlobInDifferentStorageAccount function
```

```

    processed: {myQueueItem}");
    var attributes = new Attribute[]
    {
        new BlobAttribute($"samples-output/{myQueueItem}",
FileAccess.Write),
        new StorageAccountAttribute("MyStorageAccount")
    };
    using (var writer = await binder.BindAsync<TextWriter>(attributes))
    {
        await writer.WriteAsync("Hello World!!!");
    }
}
}

```

Triggers and bindings

This table shows the bindings that are supported in the major versions of the Azure Functions runtime:

[\[\] Expand table](#)

Type	1.x ¹	2.x and higher ²	Trigger	Input	Output
Blob storage	✓	✓	✓	✓	✓
Azure Cosmos DB	✓	✓	✓	✓	✓
Azure Data Explorer		✓		✓	✓
Azure SQL		✓	✓	✓	✓
Dapr ⁴		✓	✓	✓	✓
Event Grid	✓	✓	✓		✓
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
IoT Hub	✓	✓	✓		
Kafka ³		✓	✓		✓
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
Redis		✓	✓		

Type	1.x ¹	2.x and higher ²	Trigger	Input	Output
RabbitMQ ³		✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓	✓	✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ Support will end for version 1.x of the Azure Functions runtime on September 14, 2026 [↗](#). We highly recommend that you [migrate your apps to version 4.x](#) for full support.

² Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

³ Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

⁴ Supported only in Kubernetes, IoT Edge, and other self-hosted modes only.

Next steps

[Learn more about triggers and bindings](#)

[Learn more about best practices for Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Azure Functions C# script (.csx) developer reference

Article • 09/27/2023

This article is an introduction to developing Azure Functions by using C# script (.csx).

Azure Functions lets you develop functions using C# in one of the following ways:

Type	Execution process	Code extension	Development environment	Reference
C# script	in-process	.csx	Portal Core Tools	This article
C# class library	in-process	.cs	Visual Studio Visual Studio Code Core Tools	In-process C# class library functions
C# class library (isolated worker process)	in an isolated worker process	.cs	Visual Studio Visual Studio Code Core Tools	.NET isolated worker process functions

This article assumes that you've already read the [Azure Functions developers guide](#).

How .csx works

Data flows into your C# function via method arguments. Argument names are specified in a `function.json` file, and there are predefined names for accessing things like the function logger and cancellation tokens.

The .csx format allows you to write less "boilerplate" and focus on writing just a C# function. Instead of wrapping everything in a namespace and class, just define a `Run` method. Include any assembly references and namespaces at the beginning of the file as usual.

A function app's .csx files are compiled when an instance is initialized. This compilation step means things like cold start may take longer for C# script functions compared to C# class libraries. This compilation step is also why C# script functions are editable in the Azure portal, while C# class libraries aren't.

Folder structure

The folder structure for a C# script project looks like the following example:

```
FunctionsProject
| - MyFirstFunction
| | - run.csx
| | - function.json
| | - function.proj
| - MySecondFunction
| | - run.csx
| | - function.json
| | - function.proj
| - host.json
| - extensions.csproj
| - bin
```

There's a shared [host.json](#) file that can be used to configure the function app. Each function has its own code file (.csx) and binding configuration file (function.json).

The binding extensions required in [version 2.x and later versions](#) of the Functions runtime are defined in the `extensions.csproj` file, with the actual library files in the `bin` folder. When developing locally, you must [register binding extensions](#). When you develop functions in the Azure portal, this registration is done for you.

Binding to arguments

Input or output data is bound to a C# script function parameter via the `name` property in the `function.json` configuration file. The following example shows a `function.json` file and `run.csx` file for a queue-triggered function. The parameter that receives data from the queue message is named `myQueueItem` because that's the value of the `name` property.

JSON

```
{
  "disabled": false,
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "myQueueItem",
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting"
    }
  ]
}
```

C#

```
#r "Microsoft.WindowsAzure.Storage"

using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.Queue;
using System;

public static void Run(CloudQueueMessage myQueueItem, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem.AsString}");
}
```

The `#r` statement is explained [later in this article](#).

Supported types for bindings

Each binding has its own supported types; for instance, a blob trigger can be used with a string parameter, a POCO parameter, a `CloudBlockBlob` parameter, or any of several other supported types. The [binding reference article for blob bindings](#) lists all supported parameter types for blob triggers. For more information, see [Triggers and bindings](#) and the [binding reference docs](#) for each binding type.

Tip

If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`. For more information, see [How to manage connections in Azure Functions](#).

Referencing custom classes

If you need to use a custom Plain Old CLR Object (POCO) class, you can include the class definition inside the same file or put it in a separate file.

The following example shows a `run.csx` example that includes a POCO class definition.

C#

```
public static void Run(string myBlob, out MyClass myQueueItem)
{
    log.Verbose($"C# Blob trigger function processed: {myBlob}");
    myQueueItem = new MyClass() { Id = "myid" };
}
```

```
public class MyClass
{
    public string Id { get; set; }
}
```

A POCO class must have a getter and setter defined for each property.

Reusing .csx code

You can use classes and methods defined in other .csx files in your *run.csx* file. To do that, use `#load` directives in your *run.csx* file. In the following example, a logging routine named `MyLogger` is shared in *myLogger.csx* and loaded into *run.csx* using the `#load` directive:

Example *run.csx*:

```
C#  
  
#load "mylogger.csx"  
  
using Microsoft.Extensions.Logging;  
  
public static void Run(TimerInfo myTimer, ILogger log)  
{  
    log.LogInformation($"Log by run.csx: {DateTime.Now}");  
    MyLogger(log, $"Log by MyLogger: {DateTime.Now}");  
}
```

Example *mylogger.csx*:

```
C#  
  
public static void MyLogger(ILogger log, string logtext)  
{  
    log.LogInformation(logtext);  
}
```

Using a shared .csx file is a common pattern when you want to strongly type the data passed between functions by using a POCO object. In the following simplified example, an HTTP trigger and queue trigger share a POCO object named `Order` to strongly type the order data:

Example *run.csx* for HTTP trigger:

```
C#
```

```

#load "..\shared\order.csx"

using System.Net;
using Microsoft.Extensions.Logging;

public static async Task<HttpResponseMessage> Run(Order req,
IAsyncCollector<Order> outputQueueItem, ILogger log)
{
    log.LogInformation("C# HTTP trigger function received an order.");
    log.LogInformation(req.ToString());
    log.LogInformation("Submitting to processing queue.");

    if (req.orderId == null)
    {
        return new HttpResponseMessage(HttpStatusCode.BadRequest);
    }
    else
    {
        await outputQueueItem.AddAsync(req);
        return new HttpResponseMessage(HttpStatusCode.OK);
    }
}

```

Example *run.csx* for queue trigger:

```

C#

#load "..\shared\order.csx"

using System;
using Microsoft.Extensions.Logging;

public static void Run(Order myQueueItem, out Order outputQueueItem, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed order{myQueueItem}");
    log.LogInformation(myQueueItem.ToString());

    outputQueueItem = myQueueItem;
}

```

Example *order.csx*:

```

C#

public class Order
{
    public string orderId {get; set; }
    public string custName {get; set;}
    public string custAddress {get; set;}
    public string custEmail {get; set;}
}

```

```

    public string cartId {get; set; }

    public override String ToString()
    {
        return "\n{\n\torderId : " + orderId +
               "\n\tcustName : " + custName +
               "\n\tcustAddress : " + custAddress +
               "\n\tcustEmail : " + custEmail +
               "\n\tcartId : " + cartId + "\n}";
    }
}

```

You can use a relative path with the `#load` directive:

- `#load "mylogger.csx"` loads a file located in the function folder.
- `#load "loadedfiles\mylogger.csx"` loads a file located in a folder in the function folder.
- `#load "..\shared\mylogger.csx"` loads a file located in a folder at the same level as the function folder, that is, directly under `wwwroot`.

The `#load` directive works only with `.csx` files, not with `.cs` files.

Binding to method return value

You can use a method return value for an output binding, by using the name `$return` in `function.json`.

JSON

```
{
    "name": "$return",
    "type": "blob",
    "direction": "out",
    "path": "output-container/{id}"
}
```

Here's the C# script code using the return value, followed by an async example:

C#

```

public static string Run(WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\", \"name\": \"HelloWorld\", \"value\": \"Hello {1}!\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return json;
}

```

C#

```
public static Task<string> Run(WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return Task.FromResult(json);
}
```

Use the return value only if a successful function execution always results in a return value to pass to the output binding. Otherwise, use [ICollector](#) or [IAsyncCollector](#), as shown in the following section.

Writing multiple output values

To write multiple values to an output binding, or if a successful function invocation might not result in anything to pass to the output binding, use the [ICollector](#) or [IAsyncCollector](#) types. These types are write-only collections that are written to the output binding when the method completes.

This example writes multiple queue messages into the same queue using [ICollector](#):

C#

```
public static void Run(ICollector<string> myQueue, ILogger log)
{
    myQueue.Add("Hello");
    myQueue.Add("World!");
}
```

Logging

To log output to your streaming logs in C#, include an argument of type [ILogger](#). We recommend that you name it `log`. Avoid using `Console.WriteLine` in Azure Functions.

C#

```
public static void Run(string myBlob, ILogger log)
{
    log.LogInformation($"C# Blob trigger function processed: {myBlob}");
}
```

 Note

For information about a newer logging framework that you can use instead of `TraceWriter`, see the [ILogger](#) documentation in the .NET class library developer guide.

Custom metrics logging

You can use the `LogMetric` extension method on `ILogger` to create custom metrics in Application Insights. Here's a sample method call:

C#

```
logger.LogMetric("TestMetric", 1234);
```

This code is an alternative to calling `TrackMetric` by using the Application Insights API for .NET.

Async

To make a function [asynchronous](#), use the `async` keyword and return a `Task` object.

C#

```
public async static Task ProcessQueueMessageAsync(
    string blobName,
    Stream blobInput,
    Stream blobOutput)
{
    await blobInput.CopyToAsync(blobOutput, 4096);
}
```

You can't use `out` parameters in `async` functions. For output bindings, use the [function return value](#) or a [collector object](#) instead.

Cancellation tokens

A function can accept a `CancellationToken` parameter, which enables the operating system to notify your code when the function is about to be terminated. You can use this notification to make sure the function doesn't terminate unexpectedly in a way that leaves data in an inconsistent state.

The following example shows how to check for impending function termination.

C#

```
using System;
using System.IO;
using System.Threading;

public static void Run(
    string inputText,
    TextWriter logger,
    CancellationToken token)
{
    for (int i = 0; i < 100; i++)
    {
        if (token.IsCancellationRequested)
        {
            logger.WriteLine("Function was cancelled at iteration {0}", i);
            break;
        }
        Thread.Sleep(5000);
        logger.WriteLine("Normal processing for queue message={0}",
inputText);
    }
}
```

Importing namespaces

If you need to import namespaces, you can do so as usual, with the `using` clause.

C#

```
using System.Net;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

public static Task<HttpResponseMessage> Run(HttpRequestMessage req, ILogger
log)
```

The following namespaces are automatically imported and are therefore optional:

- `System`
- `System.Collections.Generic`
- `System.IO`
- `System.Linq`
- `System.Net.Http`
- `System.Threading.Tasks`
- `Microsoft.Azure.WebJobs`

- Microsoft.Azure.WebJobs.Host

Referencing external assemblies

For framework assemblies, add references by using the `#r "AssemblyName"` directive.

C#

```
#r "System.Web.Http"

using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

public static Task<HttpResponseMessage> Run(HttpRequestMessage req, ILogger
log)
```

The following assemblies are automatically added by the Azure Functions hosting environment:

- mscorelib
- System
- System.Core
- System.Xml
- System.Net.Http
- Microsoft.Azure.WebJobs
- Microsoft.Azure.WebJobs.Host
- Microsoft.Azure.WebJobs.Extensions
- System.Web.Http
- System.Net.Http.Formatting

The following assemblies may be referenced by simple-name, by runtime version:

v2.x+

- Newtonsoft.Json
- Microsoft.WindowsAzure.Storage^{*}

^{*}Removed in version 4.x of the runtime.

In code, assemblies are referenced like the following example:

C#

```
#r "AssemblyName"
```

Referencing custom assemblies

To reference a custom assembly, you can use either a *shared* assembly or a *private* assembly:

- Shared assemblies are shared across all functions within a function app. To reference a custom assembly, upload the assembly to a folder named `bin` in the root folder (`wwwroot`) of your function app.
- Private assemblies are part of a given function's context, and support side-loading of different versions. Private assemblies should be uploaded in a `bin` folder in the function directory. Reference the assemblies using the file name, such as `#r "MyAssembly.dll"`.

For information on how to upload files to your function folder, see the section on [package management](#).

Watched directories

The directory that contains the function script file is automatically watched for changes to assemblies. To watch for assembly changes in other directories, add them to the `watchDirectories` list in [host.json](#).

Using NuGet packages

The way that both binding extension packages and other NuGet packages are added to your function app depends on the [targeted version of the Functions runtime](#).

v2.x+

By default, the [supported set of Functions extension NuGet packages](#) are made available to your C# script function app by using extension bundles. To learn more, see [Extension bundles](#).

If for some reason you can't use extension bundles in your project, you can also use the Azure Functions Core Tools to install extensions based on bindings defined in

the function.json files in your app. When using Core Tools to register extensions, make sure to use the `--csx` option. To learn more, see [func extensions install](#).

By default, Core Tools reads the function.json files and adds the required packages to an `extensions.csproj` C# class library project file in the root of the function app's file system (`wwwroot`). Because Core Tools uses `dotnet.exe`, you can use it to add any NuGet package reference to this extensions file. During installation, Core Tools builds the `extensions.csproj` to install the required libraries. Here's an example `extensions.csproj` file that adds a reference to `Microsoft.ProjectOxford.Face` version `1.1.0`:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.ProjectOxford.Face"
Version="1.1.0" />
  </ItemGroup>
</Project>
```

① Note

For C# script (.csx), you must set `TargetFramework` to a value of `netstandard2.0`. Other target frameworks, such as `net6.0`, aren't supported.

To use a custom NuGet feed, specify the feed in a `Nuget.Config` file in the function app root folder. For more information, see [Configuring NuGet behavior](#).

If you're working on your project only in the portal, you'll need to manually create the `extensions.csproj` file or a `Nuget.Config` file directly in the site. To learn more, see [Manually install extensions](#).

Environment variables

To get an environment variable or an app setting value, use `System.Environment.GetEnvironmentVariable`, as shown in the following code example:

C#

```

public static void Run(TimerInfo myTimer, ILogger log)
{
    log.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");
    log.LogInformation(GetEnvironmentVariable("AzureWebJobsStorage"));
    log.LogInformation(GetEnvironmentVariable("WEBSITE_SITE_NAME"));
}

public static string GetEnvironmentVariable(string name)
{
    return name + ": " +
        System.Environment.GetEnvironmentVariable(name,
        EnvironmentVariableTarget.Process);
}

```

Retry policies

Functions supports two built-in retry policies. For more information, see [Retry policies](#).

Fixed delay

Here's the retry policy in the *function.json* file:

JSON

```
{
  "disabled": false,
  "bindings": [
    {
      ....
    }
  ],
  "retry": {
    "strategy": "fixedDelay",
    "maxRetryCount": 4,
    "delayInterval": "00:00:10"
  }
}
```

function.json property	Description
strategy	Use <code>fixedDelay</code> .
maxRetryCount	Required. The maximum number of retries allowed per function execution. <code>-1</code> means to retry indefinitely.
delayInterval	The delay that's used between retries. Specify it as a string with

function.json property	Description
	the format <code>HH:mm:ss</code> .

Binding at runtime

In C# and other .NET languages, you can use an [imperative](#) binding pattern, as opposed to the [declarative](#) bindings in *function.json*. Imperative binding is useful when binding parameters need to be computed at runtime rather than design time. With this pattern, you can bind to supported input and output bindings on-the-fly in your function code.

Define an imperative binding as follows:

- **Do not** include an entry in *function.json* for your desired imperative bindings.
- Pass in an input parameter [Binder binder](#) or [IBinder binder](#).
- Use the following C# pattern to perform the data binding.

C#

```
using (var output = await binder.BindAsync<T>(new
BindingTypeAttribute(...)))
{
    ...
}
```

`BindingTypeAttribute` is the .NET attribute that defines your binding and `T` is an input or output type that's supported by that binding type. `T` can't be an `out` parameter type (such as `out JObject`). For example, the Mobile Apps table output binding supports [six output types](#), but you can only use [ICollector<T>](#) or [IAsyncCollector<T>](#) for `T`.

Single attribute example

The following example code creates a [Storage blob output binding](#) with blob path that's defined at run time, then writes a string to the blob.

C#

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host.Bindings.Runtime;

public static async Task Run(string input, Binder binder)
{
    using (var writer = await binder.BindAsync<TextWriter>(new
```

```

        BlobAttribute("samples-output/path")))
    {
        writer.WriteLine("Hello World!!");
    }
}

```

`BlobAttribute` [defines the Storage blob input or output binding](#), and `TextWriter` is a supported output binding type.

Multiple attributes example

The preceding example gets the app setting for the function app's main Storage account connection string (which is `AzureWebJobsStorage`). You can specify a custom app setting to use for the Storage account by adding the `StorageAccountAttribute` [and passing the attribute array into `BindAsync<T>\(\)`](#). Use a `Binder` parameter, not `IBinder`. For example:

C#

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host.Bindings.Runtime;

public static async Task Run(string input, Binder binder)
{
    var attributes = new Attribute[]
    {
        new BlobAttribute("samples-output/path"),
        new StorageAccountAttribute("MyStorageAccount")
    };

    using (var writer = await binder.BindAsync<TextWriter>(attributes))
    {
        writer.WriteLine("Hello World!");
    }
}

```

The following table lists the .NET attributes for each binding type and the packages in which they're defined.

Binding	Attribute	Add reference
Azure Cosmos DB	<code>Microsoft.Azure.WebJobs.DocumentDBAttribute</code> ↳	#r "Microsoft.Azure.WebJobs.Extensions.CosmosDB"
Event Hubs	<code>Microsoft.Azure.WebJobs.ServiceBus.EventHubAttribute</code> ↳ , <code>Microsoft.Azure.WebJobs.ServiceBusAccountAttribute</code> ↳	#r "Microsoft.Azure.WebJobs.ServiceBus"

Binding	Attribute	Add reference
Mobile Apps	Microsoft.Azure.WebJobs.MobileTableAttribute	#r "Microsoft.Azure.WebJobs.Extensions.MobileApps"
Notification Hubs	Microsoft.Azure.WebJobs.NotificationHubAttribute	#r "Microsoft.Azure.WebJobs.Extensions.NotificationHubs"
Service Bus	Microsoft.Azure.WebJobs.ServiceBusAttribute , Microsoft.Azure.WebJobs.ServiceBusAccountAttribute	#r "Microsoft.Azure.WebJobs.ServiceBus"
Storage queue	Microsoft.Azure.WebJobs.QueueAttribute , Microsoft.Azure.WebJobs.StorageAccountAttribute	
Storage blob	Microsoft.Azure.WebJobs.BlobAttribute , Microsoft.Azure.WebJobs.StorageAccountAttribute	
Storage table	Microsoft.Azure.WebJobs.TableAttribute , Microsoft.Azure.WebJobs.StorageAccountAttribute	
Twilio	Microsoft.Azure.WebJobs.TwilioSmsAttribute	#r "Microsoft.Azure.WebJobs.Extensions.Twilio"

Convert a C# script app to a C# project

The easiest way to convert a C# script function app to a compiled C# class library project is to start with a new project. You can then, for each function, migrate the code and configuration from each run.csx file and function.json file in a function folder to a single new .cs class library code file. For example, when you have a C# script function named `HelloWorld` you'll have two files: `HelloWorld/run.csx` and `HelloWorld/function.json`. For this function, you create a code file named `HelloWorld.cs` in your new class library project.

If you are using C# scripting for portal editing, you can [download the app content to your local machine](#). Choose the **Site content** option instead of **Content and Visual Studio project**. You don't need to generate a project, and don't include application settings in the download. You're defining a new development environment, and this environment shouldn't have the same permissions as your hosted app environment.

These instructions show you how to convert C# script functions (which run in-process with the Functions host) to C# class library functions that run in an [isolated worker process](#).

1. Complete the [Create a functions app project](#) section from your preferred quickstart:

Azure CLI

[Create a C# function in Azure from the command line](#)

1. If your original C# script code includes an `extensions.csproj` file or any `function.proj` files, copy the package references from these file and add them to the new project's `.csproj` file in the same `ItemGroup` with the Functions core dependencies.

 Tip

Conversion provides a good opportunity to update to the latest versions of your dependencies. Doing so may require additional code changes in a later step.

2. Copy the contents of the original `host.json` file into the new project's `host.json` file, except for the `extensionBundles` section (compiled C# projects don't use [extension bundles](#) and you must explicitly add references to all extensions used by your functions). When merging `host.json` files, remember that the `host.json` schema is versioned, with most apps using version 2.0. The contents of the `extensions` section can differ based on specific versions of the binding extensions used by your functions. See individual extension reference articles to learn how to correctly configure the `host.json` for your specific versions.
3. For any [shared files referenced by a #load directive](#), create a new `.cs` file for each of these shared references. It's simplest to create a new `.cs` file for each shared class definition. If there are static methods without a class, you need to define new classes for these methods.
4. Perform the following tasks for each `<FUNCTION_NAME>` folder in your original project:
 - a. Create a new file named `<FUNCTION_NAME>.cs`, replacing `<FUNCTION_NAME>` with the name of the folder that defined your C# script function. You can create a new function code file from one of the trigger-specific templates in the following way:

Azure CLI

Using the `func new --name <FUNCTION_NAME>` command and choosing the correct trigger template at the prompt.

- b. Copy the `using` statements from your `run.csx` file and add them to the new file. You do not need any `#r` directives.
 - c. For any `#load` statement in your `run.csx` file, add a new `using` statement for the namespace you used for the shared code.
 - d. In the new file, define a class for your function under the namespace you are using for the project.
 - e. Create a new method named `RunHandler` or something similar. This new method serves as the new entry point for the function.
 - f. Copy the static method that represents your function, along with any functions it calls, from `run.csx` into your new class as a second method. From the new method you created in the previous step, call into this static method. This indirection step is helpful for navigating any differences as you continue the upgrade. You can keep the original method exactly the same and simply control its inputs from the new context. You may need to create parameters on the new method which you then pass into the static method call. After you have confirmed that the migration has worked as intended, you can remove this extra level of indirection.
 - g. For each binding in the `function.json` file, add the corresponding attribute to your new method. To quickly find binding examples, see [Manually add bindings based on examples](#).
 - h. Add any extension packages required by the bindings to your project, if you haven't already done so.
5. Recreate any application settings required by your app in the `Values` collection of the [local.settings.json file](#).
 6. Verify that your project runs locally:

Azure CLI

Use `func start` to run your app from the command line. For more information, see [Run functions locally](#).

7. Publish your project to a new function app in Azure:

Azure CLI

Create your Azure resources and deploy the code project to Azure by using the `func azure functionapp publish <APP_NAME>` command. For more information, see [Deploy project files](#).

Example function conversion

This section shows an example of the migration for a single function.

The original function in C# scripting has two files:

- `HelloWorld/function.json`
- `HelloWorld/run.csx`

The contents of `HelloWorld/function.json` are:

JSON

```
{  
  "bindings": [  
    {  
      "authLevel": "FUNCTION",  
      "name": "req",  
      "type": "httpTrigger",  
      "direction": "in",  
      "methods": [  
        "get",  
        "post"  
      ]  
    },  
    {  
      "name": "$return",  
      "type": "http",  
      "direction": "out"  
    }  
  ]  
}
```

The contents of `HelloWorld/run.csx` are:

C#

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    string responseMessage = string.IsNullOrEmpty(name)
        ? "This HTTP triggered function executed successfully. Pass a name
        in the query string or in the request body for a personalized response."
        : $"Hello, {name}. This HTTP triggered function executed
        successfully.";

    return new OkObjectResult(responseMessage);
}
```

After migrating to the isolated worker model with ASP.NET Core integration, these are replaced by a single `HelloWorld.cs`:

C#

```
using System.Net;
using Microsoft.Azure.Functions.Worker;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

namespace MyFunctionApp
{
    public class HelloWorld
    {
        private readonly ILogger _logger;

        public HelloWorld	ILoggerFactory loggerFactory
        {
            _logger = loggerFactory.CreateLogger<HelloWorld>();
        }
    }
}
```

```

[Function("HelloWorld")]
public async Task<IActionResult>
RunHandler([HttpTrigger(AuthorizationLevel.Function, "get")] HttpRequest
req)
{
    return await Run(req, _logger);
}

// From run.csx
public static async Task<IActionResult> Run(HttpContext req, ILogger
log)
{
    log.LogInformation("C# HTTP trigger function processed a
request.");

    string name = req.Query["name"];

    string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    string responseMessage = string.IsNullOrEmpty(name)
        ? "This HTTP triggered function executed successfully. Pass
a name in the query string or in the request body for a personalized
response."
        : $"Hello, {name}. This HTTP triggered function
executed successfully.";

    return new OkObjectResult(responseMessage);
}
}
}

```

Binding configuration and examples

This section contains references and examples for defining triggers and bindings in C# script.

Blob trigger

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json	Description
property	
type	Must be set to <code>blobTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the blob in function code.
path	The <code>container</code> to monitor. May be a blob name pattern .
connection	The name of an app setting or setting collection that specifies how to connect to Azure Blobs. See Connections .

The following example shows a blob trigger definition in a `function.json` file and code that uses the binding. The function writes a log when a blob is added or updated in the `samples-workitems` container.

Here's the binding data in the `function.json` file:

JSON

```
{
  "disabled": false,
  "bindings": [
    {
      "name": "myBlob",
      "type": "blobTrigger",
      "direction": "in",
      "path": "samples-workitems/{name}",
      "connection": "MyStorageAccountAppSetting"
    }
  ]
}
```

The string `{name}` in the blob trigger path `samples-workitems/{name}` creates a [binding expression](#) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](#).

Here's C# script code that binds to a `Stream`:

C#

```
public static void Run(Stream myBlob, string name, ILogger log)
{
  log.LogInformation($"C# Blob trigger function Processed blob\n Name: {name}");
}
```

```
{name} \n Size: {myBlob.Length} Bytes");  
}
```

Here's C# script code that binds to a `CloudBlockBlob`:

C#

```
#r "Microsoft.WindowsAzure.Storage"  
  
using Microsoft.WindowsAzure.Storage.Blob;  
  
public static void Run(CloudBlockBlob myBlob, string name, ILogger log)  
{  
    log.LogInformation($"C# Blob trigger function Processed blob\n Name:  
{name}\nURI:{myBlob.StorageUri}");  
}
```

Blob input

The following table explains the binding configuration properties for C# script that you set in the `function.json` file.

function.json property	Description
<code>type</code>	Must be set to <code>blob</code> .
<code>direction</code>	Must be set to <code>in</code> .
<code>name</code>	The name of the variable that represents the blob in function code.
<code>path</code>	The path to the blob.
<code>connection</code>	The name of an app setting or setting collection that specifies how to connect to Azure Blobs. See Connections .

The following example shows blob input and output bindings in a `function.json` file and C# script code that uses the bindings. The function makes a copy of a text blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named `{originalblobname}-Copy`.

In the `function.json` file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

JSON

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

C#

```
public static void Run(string myQueueItem, string myInputBlob, out string
myOutputBlob, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");
    myOutputBlob = myInputBlob;
}
```

Blob output

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json property	Description
type	Must be set to <code>blob</code> .

function.json	Description
property	
direction	Must be set to <code>out</code> .
name	The name of the variable that represents the blob in function code.
path	The path to the blob.
connection	The name of an app setting or setting collection that specifies how to connect to Azure Blobs. See Connections .

The following example shows blob input and output bindings in a `function.json` file and C# script code that uses the bindings. The function makes a copy of a text blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named `{originalblobname}-Copy`.

In the `function.json` file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

JSON

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

C#

```
public static void Run(string myQueueItem, string myInputBlob, out string myOutputBlob, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");
    myOutputBlob = myInputBlob;
}
```

RabbitMQ trigger

The following example shows a RabbitMQ trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function reads and logs the RabbitMQ message.

Here's the binding data in the *function.json* file:

JSON

```
{
  "bindings": [
    {
      "name": "myQueueItem",
      "type": "rabbitMQTrigger",
      "direction": "in",
      "queueName": "queue",
      "connectionStringSetting": "rabbitMQConnectionStringAppSetting"
    }
  ]
}
```

Here's the C# script code:

C#

```
using System;

public static void Run(string myQueueItem, ILogger log)
{
    log.LogInformation($"C# Script RabbitMQ trigger function processed: {
myQueueItem}");
}
```

Queue trigger

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json property	Description
type	Must be set to <code>queueTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	In the <i>function.json</i> file only. Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that contains the queue item payload in the function code.
queueName	The name of the queue to poll.
connection	The name of an app setting or setting collection that specifies how to connect to Azure Queues. See Connections .

The following example shows a queue trigger binding in a *function.json* file and C# script code that uses the binding. The function polls the `myqueue-items` queue and writes a log each time a queue item is processed.

Here's the *function.json* file:

```
JSON

{
  "disabled": false,
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "myQueueItem",
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting"
    }
  ]
}
```

Here's the C# script code:

```
C#

#r "Microsoft.WindowsAzure.Storage"

using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.Queue;
```

```

using System;

public static void Run(CloudQueueMessage myQueueItem,
    DateTimeOffset expirationTime,
    DateTimeOffset insertionTime,
    DateTimeOffset nextVisibleTime,
    string queueTrigger,
    string id,
    string popReceipt,
    int dequeueCount,
    ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItemAsString}\n" +
    $"queueTrigger={queueTrigger}\n" +
    $"expirationTime={expirationTime}\n" +
    $"insertionTime={insertionTime}\n" +
    $"nextVisibleTime={nextVisibleTime}\n" +
    $"id={id}\n" +
    $"popReceipt={popReceipt}\n" +
    $"dequeueCount={dequeueCount}");
}

```

Queue output

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json property	Description
type	Must be set to <code>queue</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>out</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the queue in function code. Set to <code>\$return</code> to reference the function return value.
queueName	The name of the queue.
connection	The name of an app setting or setting collection that specifies how to connect to Azure Queues. See Connections .

The following example shows an HTTP trigger binding in a *function.json* file and C# script code that uses the binding. The function creates a queue item with a `CustomQueueMessage` object payload for each HTTP request received.

Here's the `function.json` file:

JSON

```
{  
  "bindings": [  
    {  
      "type": "httpTrigger",  
      "direction": "in",  
      "authLevel": "function",  
      "name": "input"  
    },  
    {  
      "type": "http",  
      "direction": "out",  
      "name": "$return"  
    },  
    {  
      "type": "queue",  
      "direction": "out",  
      "name": "$return",  
      "queueName": "outqueue",  
      "connection": "MyStorageConnectionAppSetting"  
    }  
  ]  
}
```

Here's C# script code that creates a single queue message:

C#

```
public class CustomQueueMessage  
{  
    public string PersonName { get; set; }  
    public string Title { get; set; }  
}  
  
public static CustomQueueMessage Run(CustomQueueMessage input, ILogger log)  
{  
    return input;  
}
```

You can send multiple messages at once by using an `ICollector` or `IAsyncCollector` parameter. Here's C# script code that sends multiple messages, one with the HTTP request data and one with hard-coded values:

C#

```
public static void Run(  
    CustomQueueMessage input,
```

```

    ICollector<CustomQueueMessage> myQueueItems,
    ILogger log)
{
    myQueueItems.Add(input);
    myQueueItems.Add(new CustomQueueMessage { PersonName = "You", Title =
    "None" });
}

```

Table input

This section outlines support for the [Tables API version of the extension](#) only.

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json	Description
property	
type	Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal.
direction	Must be set to <code>in</code> . This property is set automatically when you create the binding in the Azure portal.
name	The name of the variable that represents the table or entity in function code.
tableName	The name of the table.
partitionKey	Optional. The partition key of the table entity to read.
rowKey	Optional. The row key of the table entity to read. Can't be used with <code>take</code> or <code>filter</code> .
take	Optional. The maximum number of entities to return. Can't be used with <code>rowKey</code> .
filter	Optional. An OData filter expression for the entities to return from the table. Can't be used with <code>rowKey</code> .
connection	The name of an app setting or setting collection that specifies how to connect to the table service. See Connections .

The following example shows a table input binding in a *function.json* file and C# script code that uses the binding. The function uses a queue trigger to read a single table row.

The *function.json* file specifies a `partitionKey` and a `rowKey`. The `rowKey` value `{queueTrigger}` indicates that the row key comes from the queue message string.

JSON

```
{  
  "bindings": [  
    {  
      "queueName": "myqueue-items",  
      "connection": "MyStorageConnectionAppSetting",  
      "name": "myQueueItem",  
      "type": "queueTrigger",  
      "direction": "in"  
    },  
    {  
      "name": "personEntity",  
      "type": "table",  
      "tableName": "Person",  
      "partitionKey": "Test",  
      "rowKey": "{queueTrigger}",  
      "connection": "MyStorageConnectionAppSetting",  
      "direction": "in"  
    }  
,  
    {"disabled": false  
  }  
}
```

Here's the C# script code:

C#

```
#r "Azure.Data.Tables"  
using Microsoft.Extensions.Logging;  
using Azure.Data.Tables;  
  
public static void Run(string myQueueItem, Person personEntity, ILogger log)  
{  
    log.LogInformation($"C# Queue trigger function processed:  
{myQueueItem}");  
    log.LogInformation($"Name in Person entity: {personEntity.Name}");  
}  
  
public class Person : ITableEntity  
{  
    public string Name { get; set; }  
  
    public string PartitionKey { get; set; }  
    public string RowKey { get; set; }  
    public DateTimeOffset? Timestamp { get; set; }  
    public ETag ETag { get; set; }  
}
```

Table output

This section outlines support for the [Tables API version of the extension](#) only.

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json property	Description
type	Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal.
direction	Must be set to <code>out</code> . This property is set automatically when you create the binding in the Azure portal.
name	The variable name used in function code that represents the table or entity. Set to <code>\$return</code> to reference the function return value.
tableName	The name of the table to which to write.
partitionKey	The partition key of the table entity to write.
rowKey	The row key of the table entity to write.
connection	The name of an app setting or setting collection that specifies how to connect to the table service. See Connections .

The following example shows a table output binding in a *function.json* file and C# script code that uses the binding. The function writes multiple table entities.

Here's the *function.json* file:

JSON

```
{
  "bindings": [
    {
      "name": "input",
      "type": "manualTrigger",
      "direction": "in"
    },
    {
      "tableName": "Person",
      "connection": "MyStorageConnectionAppSetting",
      "name": "tableBinding",
      "type": "table",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

C#

```
public static void Run(string input, ICollector<Person> tableBinding,
ILogger log)
{
    for (int i = 1; i < 10; i++)
    {
        log.LogInformation($"Adding Person entity {i}");
        tableBinding.Add(
            new Person() {
                PartitionKey = "Test",
                RowKey = i.ToString(),
                Name = "Name" + i.ToString() })
    }
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
}
```

Timer trigger

The following table explains the binding configuration properties for C# script that you set in the *function.json* file.

function.json	Description
property	
type	Must be set to <code>timerTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the timer object in function code.
schedule	A CRON expression or a TimeSpan value. A <code>TimeSpan</code> can be used only for a function app that runs on an App Service Plan. You can put the schedule expression in an app setting and set this property to the app setting name wrapped in % signs, as in this example: "%ScheduleAppSetting%".

function.json	Description
property	
runOnStartup	If <code>true</code> , the function is invoked when the runtime starts. For example, the runtime starts when the function app wakes up after going idle due to inactivity, when the function app restarts due to function changes, and when the function app scales out. <i>Use with caution.</i> <code>runOnStartup</code> should rarely if ever be set to <code>true</code> , especially in production.
useMonitor	Set to <code>true</code> or <code>false</code> to indicate whether the schedule should be monitored. Schedule monitoring persists schedule occurrences to aid in ensuring the schedule is maintained correctly even when function app instances restart. If not set explicitly, the default is <code>true</code> for schedules that have a recurrence interval greater than or equal to 1 minute. For schedules that trigger more than once per minute, the default is <code>false</code> .

The following example shows a timer trigger binding in a `function.json` file and a C# script function that uses the binding. The function writes a log indicating whether this function invocation is due to a missed schedule occurrence. The [TimerInfo](#) object is passed into the function.

Here's the binding data in the `function.json` file:

JSON

```
{
  "schedule": "0 */5 * * * *",
  "name": "myTimer",
  "type": "timerTrigger",
  "direction": "in"
}
```

Here's the C# script code:

C#

```
public static void Run(TimerInfo myTimer, ILogger log)
{
    if (myTimer.IsPastDue)
    {
        log.LogInformation("Timer is running late!");
    }
    log.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");
```

HTTP trigger

The following table explains the trigger configuration properties that you set in the `function.json` file:

function.json property	Description
<code>type</code>	Required - must be set to <code>httpTrigger</code> .
<code>direction</code>	Required - must be set to <code>in</code> .
<code>name</code>	Required - the variable name used in function code for the request or request body.
<code>authLevel</code>	Determines what keys, if any, need to be present on the request in order to invoke the function. For supported values, see Authorization level .
<code>methods</code>	An array of the HTTP methods to which the function responds. If not specified, the function responds to all HTTP methods. See customize the HTTP endpoint .
<code>route</code>	Defines the route template, controlling to which request URLs your function responds. The default value if none is provided is <code><functionname></code> . For more information, see customize the HTTP endpoint .
<code>webHookType</code>	<i>Supported only for the version 1.x runtime.</i> Configures the HTTP trigger to act as a webhook receiver for the specified provider. For supported values, see WebHook type .

The following example shows a trigger binding in a `function.json` file and a C# script function that uses the binding. The function looks for a `name` parameter either in the query string or the body of the HTTP request.

Here's the `function.json` file:

```
JSON

{
  "disabled": false,
  "bindings": [
    {
      "authLevel": "function",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
  ]}
```

```

        {
            "name": "$return",
            "type": "http",
            "direction": "out"
        }
    ]
}

```

Here's C# script code that binds to `HttpRequest`:

C#

```

#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

    string requestBody = String.Empty;
    using (StreamReader streamReader = new StreamReader(req.Body))
    {
        requestBody = await streamReader.ReadToEndAsync();
    }
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

You can bind to a custom object instead of `HttpRequest`. This object is created from the body of the request and parsed as JSON. Similarly, a type can be passed to the HTTP response output binding and returned as the response body, along with a `200` status code.

C#

```

using System.Net;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

```

```

public static string Run(Person person, ILogger log)
{
    return person.Name != null
        ? (ActionResult)new OkObjectResult($"Hello, {person.Name}")
        : new BadRequestObjectResult("Please pass an instance of Person.");
}

public class Person {
    public string Name {get; set;}
}

```

HTTP output

The following table explains the binding configuration properties that you set in the `function.json` file.

Property	Description
<code>type</code>	Must be set to <code>http</code> .
<code>direction</code>	Must be set to <code>out</code> .
<code>name</code>	The variable name used in function code for the response, or <code>\$return</code> to use the return value.

Event Hubs trigger

The following table explains the trigger configuration properties that you set in the `function.json` file:

function.json	Description
property	
<code>type</code>	Must be set to <code>eventHubTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
<code>direction</code>	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
<code>name</code>	The name of the variable that represents the event item in function code.
<code>eventHubName</code>	Functions 2.x and higher. The name of the event hub. When the event hub name is also present in the connection string, that value overrides this property at runtime. Can be referenced via <code>app settings %eventHubName%</code> . In version 1.x, this property is named <code>path</code> .
<code>consumerGroup</code>	An optional property that sets the <code>consumer group</code> used to subscribe to

function.json	Description
property	events in the hub. If omitted, the <code>\$Default</code> consumer group is used.
connection	The name of an app setting or setting collection that specifies how to connect to Event Hubs. See Connections .

The following example shows an Event Hubs trigger binding in a `function.json` file and a C# script function that uses the binding. The function logs the message body of the Event Hubs trigger.

The following examples show Event Hubs binding data in the `function.json` file for Functions runtime version 2.x and later versions.

JSON

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "eventHubName": "MyEventHub",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the C# script code:

C#

```
using System;

public static void Run(string myEventHubMessage, TraceWriter log)
{
    log.Info($"C# function triggered to process a message:
{myEventHubMessage}");
}
```

To get access to event metadata in function code, bind to an `EventData` object. You can also access the same properties by using binding expressions in the method signature. The following example shows both ways to get the same data:

C#

```
#r "Microsoft.Azure.EventHubs"

using System.Text;
using System;
using Microsoft.ServiceBus.Messaging;
```

```

using Microsoft.Azure.EventHubs;

public void Run(EventData myEventHubMessage,
    DateTime enqueuedTimeUtc,
    Int64 sequenceNumber,
    string offset,
    TraceWriter log)
{
    log.Info($"Event: {Encoding.UTF8.GetString(myEventHubMessage.Body)}");
    log.Info($"EnqueuedTimeUtc={myEventHubMessage.SystemProperties.EnqueuedTimeUtc}");
    log.Info($"SequenceNumber={myEventHubMessage.SystemProperties.SequenceNumber}");
    log.Info($"Offset={myEventHubMessage.SystemProperties.Offset}");

    // Metadata accessed by using binding expressions
    log.Info($"EnqueuedTimeUtc={enqueuedTimeUtc}");
    log.Info($"SequenceNumber={sequenceNumber}");
    log.Info($"Offset={offset}");
}

```

To receive events in a batch, make `string` or `EventData` an array:

C#

```

public static void Run(string[] eventHubMessages, TraceWriter log)
{
    foreach (var message in eventHubMessages)
    {
        log.Info($"C# function triggered to process a message: {message}");
    }
}

```

Event Hubs output

The following table explains the binding configuration properties that you set in the `function.json` file.

function.json property	Description
<code>type</code>	Must be set to <code>eventHub</code> .
<code>direction</code>	Must be set to <code>out</code> . This parameter is set automatically when you create the binding in the Azure portal.
<code>name</code>	The variable name used in function code that represents the event.

function.json	Description
eventHubName	Functions 2.x and higher. The name of the event hub. When the event hub name is also present in the connection string, that value overrides this property at runtime. In Functions 1.x, this property is named <code>path</code> .
connection	The name of an app setting or setting collection that specifies how to connect to Event Hubs. To learn more, see Connections .

The following example shows an event hub trigger binding in a `function.json` file and a C# script function that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the `function.json` file for Functions runtime version 2.x and later versions.

JSON

```
{
  "type": "eventHub",
  "name": "outputEventHubMessage",
  "eventHubName": "myeventhub",
  "connection": "MyEventHubSendAppSetting",
  "direction": "out"
}
```

Here's C# script code that creates one message:

C#

```
using System;
using Microsoft.Extensions.Logging;

public static void Run(TimerInfo myTimer, out string outputEventHubMessage,
ILogger log)
{
    String msg = $"TimerTriggerCSharp1 executed at: {DateTime.Now}";
    log.LogInformation(msg);
    outputEventHubMessage = msg;
}
```

Here's C# script code that creates multiple messages:

C#

```
public static void Run(TimerInfo myTimer, ICollector<string>
outputEventHubMessage, ILogger log)
{
    string message = $"Message created at: {DateTime.Now}";
```

```

        log.LogInformation(message);
        outputEventHubMessage.Add("1 " + message);
        outputEventHubMessage.Add("2 " + message);
    }

```

Event Grid trigger

The following table explains the binding configuration properties for C# script that you set in the `function.json` file. There are no constructor parameters or properties to set in the `EventGridTrigger` attribute.

function.json	Description
property	
<code>type</code>	Required - must be set to <code>eventGridTrigger</code> .
<code>direction</code>	Required - must be set to <code>in</code> .
<code>name</code>	Required - the variable name used in function code for the parameter that receives the event data.

The following example shows an Event Grid trigger defined in the `function.json` file.

Here's the binding data in the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "type": "eventGridTrigger",
      "name": "eventGridEvent",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here's an example of a C# script function that uses an `EventGridEvent` binding parameter:

C#

```
#r "Azure.Messaging.EventGrid"
using Azure.Messaging.EventGrid;
using Microsoft.Extensions.Logging;
```

```
public static void Run(EventGridEvent eventGridEvent, ILogger log)
{
    log.LogInformation(eventGridEvent.Data.ToString());
}
```

Here's an example of a C# script function that uses a `JObject` binding parameter:

C#

```
#r "Newtonsoft.Json"

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

public static void Run(JObject eventGridEvent, TraceWriter log)
{
    log.Info(eventGridEvent.ToString(Formatting.Indented));
}
```

Event Grid output

The following table explains the binding configuration properties for C# script that you set in the `function.json` file.

function.json property	Description
<code>type</code>	Must be set to <code>eventGrid</code> .
<code>direction</code>	Must be set to <code>out</code> . This parameter is set automatically when you create the binding in the Azure portal.
<code>name</code>	The variable name used in function code that represents the event.
<code>topicEndpointUri</code>	The name of an app setting that contains the URI for the custom topic, such as <code>MyTopicEndpointUri</code> .
<code>topicKeySetting</code>	The name of an app setting that contains an access key for the custom topic.

The following example shows the Event Grid output binding data in the `function.json` file.

JSON

```
{
    "type": "eventGrid",
    "name": "outputEvent",
    "topicEndpointUri": "MyEventGridTopicUriSetting",
```

```
    "topicKeySetting": "MyEventGridTopicKeySetting",
    "direction": "out"
}
```

Here's C# script code that creates one event:

```
C#  
  
#r "Microsoft.Azure.EventGrid"  
using System;  
using Microsoft.Azure.EventGrid.Models;  
using Microsoft.Extensions.Logging;  
  
public static void Run(TimerInfo myTimer, out EventGridEvent outputEvent,  
ILogger log)  
{  
    outputEvent = new EventGridEvent("message-id", "subject-name", "event-  
data", "event-type", DateTime.UtcNow, "1.0");  
}
```

Here's C# script code that creates multiple events:

```
C#  
  
#r "Microsoft.Azure.EventGrid"  
using System;  
using Microsoft.Azure.EventGrid.Models;  
using Microsoft.Extensions.Logging;  
  
public static void Run(TimerInfo myTimer, ICollector<EventGridEvent>  
outputEvent, ILogger log)  
{  
    outputEvent.Add(new EventGridEvent("message-id-1", "subject-name",  
"event-data", "event-type", DateTime.UtcNow, "1.0"));  
    outputEvent.Add(new EventGridEvent("message-id-2", "subject-name",  
"event-data", "event-type", DateTime.UtcNow, "1.0"));  
}
```

Service Bus trigger

The following table explains the binding configuration properties that you set in the *function.json* file.

function.json property	Description
type	Must be set to <code>serviceBusTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.

function.json	Description
property	
direction	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the queue or topic message in function code.
queueName	Name of the queue to monitor. Set only if monitoring a queue, not for a topic.
topicName	Name of the topic to monitor. Set only if monitoring a topic, not for a queue.
subscriptionName	Name of the subscription to monitor. Set only if monitoring a topic, not for a queue.
connection	The name of an app setting or setting collection that specifies how to connect to Service Bus. See Connections .
accessRights	Access rights for the connection string. Available values are <code>manage</code> and <code>listen</code> . The default is <code>manage</code> , which indicates that the <code>connection</code> has the Manage permission. If you use a connection string that does not have the Manage permission, set <code>accessRights</code> to "listen". Otherwise, the Functions runtime might fail trying to do operations that require manage rights. In Azure Functions version 2.x and higher, this property is not available because the latest version of the Service Bus SDK doesn't support manage operations.
isSessionsEnabled	<code>true</code> if connecting to a session-aware queue or subscription. <code>false</code> otherwise, which is the default value.
autoComplete	<code>true</code> when the trigger should automatically call <code>complete</code> after processing, or if the function code will manually call <code>complete</code> .
	Setting to <code>false</code> is only supported in C#.
	If set to <code>true</code> , the trigger completes the message automatically if the function execution completes successfully, and abandons the message otherwise.
	When set to <code>false</code> , you are responsible for calling MessageReceiver methods to complete, abandon, or deadletter the message. If an exception is thrown (and none of the <code>MessageReceiver</code> methods are called), then the lock remains. Once the lock expires, the message is re-queued with the <code>DeliveryCount</code> incremented and the lock is automatically renewed.
	This property is available only in Azure Functions 2.x and higher.

The following example shows a Service Bus trigger binding in a *function.json* file and a C# script function that uses the binding. The function reads message metadata and logs a Service Bus queue message.

Here's the binding data in the *function.json* file:

JSON

```
{  
  "bindings": [  
    {  
      "queueName": "testqueue",  
      "connection": "MyServiceBusConnection",  
      "name": "myQueueItem",  
      "type": "serviceBusTrigger",  
      "direction": "in"  
    }  
,  
  {"disabled": false  
}
```

Here's the C# script code:

C#

```
using System;  
  
public static void Run(string myQueueItem,  
    Int32 deliveryCount,  
    DateTime enqueuedTimeUtc,  
    string messageId,  
    TraceWriter log)  
{  
    log.Info($"C# ServiceBus queue trigger function processed message:  
{myQueueItem}");  
  
    log.Info($"EnqueuedTimeUtc={enqueuedTimeUtc}");  
    log.Info($"DeliveryCount={deliveryCount}");  
    log.Info($"MessageId={messageId}");  
}
```

Service Bus output

The following table explains the binding configuration properties that you set in the *function.json* file.

function.json	Description
property	
type	Must be set to <code>serviceBus</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>out</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the queue or topic message in function code. Set to "\$return" to reference the function return value.
queueName	Name of the queue. Set only if sending queue messages, not for a topic.
topicName	Name of the topic. Set only if sending topic messages, not for a queue.
connection	The name of an app setting or setting collection that specifies how to connect to Service Bus. See Connections .
accessRights (v1 only)	Access rights for the connection string. Available values are <code>manage</code> and <code>listen</code> . The default is <code>manage</code> , which indicates that the <code>connection</code> has the Manage permission. If you use a connection string that does not have the Manage permission, set <code>accessRights</code> to "listen". Otherwise, the Functions runtime might fail trying to do operations that require manage rights. In Azure Functions version 2.x and higher, this property is not available because the latest version of the Service Bus SDK doesn't support manage operations.

The following example shows a Service Bus output binding in a `function.json` file and a C# script function that uses the binding. The function uses a timer trigger to send a queue message every 15 seconds.

Here's the binding data in the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "schedule": "0/15 * * * *",
      "name": "myTimer",
      "runsOnStartup": true,
      "type": "timerTrigger",
      "direction": "in"
    },
    {
      "name": "outputSbQueue",
      "type": "serviceBus",
      "queueName": "testqueue",
      "connection": "MyServiceBusConnection",
      "direction": "out"
    }
  ]
}
```

```
  ],
  "disabled": false
}
```

Here's C# script code that creates a single message:

```
C#

public static void Run(TimerInfo myTimer, ILogger log, out string
outputSbQueue)
{
    string message = $"Service Bus queue message created at:
{DateTime.Now}";
    log.LogInformation(message);
    outputSbQueue = message;
}
```

Here's C# script code that creates multiple messages:

```
C#

public static async Task Run(TimerInfo myTimer, ILogger log,
IAsyncCollector<string> outputSbQueue)
{
    string message = $"Service Bus queue messages created at:
{DateTime.Now}";
    log.LogInformation(message);
    await outputSbQueue.AddAsync("1 " + message);
    await outputSbQueue.AddAsync("2 " + message);
}
```

Azure Cosmos DB v2 trigger

This section outlines support for the [version 4.x+ of the extension](#) only.

The following table explains the binding configuration properties that you set in the `function.json` file.

function.json property	Description
<code>type</code>	Must be set to <code>cosmosDBTrigger</code> .
<code>direction</code>	Must be set to <code>in</code> . This parameter is set automatically when you create the trigger in the Azure portal.
<code>name</code>	The variable name used in function code that represents the list of documents with changes.

function.json property	Description
connection	The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account being monitored. For more information, see Connections .
databaseName	The name of the Azure Cosmos DB database with the container being monitored.
containerName	The name of the container being monitored.
leaseConnection	(Optional) The name of an app setting or setting container that specifies how to connect to the Azure Cosmos DB account that holds the lease container. When not set, the <code>connection</code> value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases container must have write permissions.
leaseDatabaseName	(Optional) The name of the database that holds the container used to store leases. When not set, the value of the <code>databaseName</code> setting is used.
leaseContainerName	(Optional) The name of the container used to store leases. When not set, the value <code>leases</code> is used.
createLeaseContainerIfNotExists	(Optional) When set to <code>true</code> , the leases container is automatically created when it doesn't already exist. The default value is <code>false</code> . When using Azure AD identities if you set the value to <code>true</code> , creating containers is not an allowed operation and your Function won't be able to start.
leasesContainerThroughput	(Optional) Defines the number of Request Units to assign when the leases container is created. This setting is only used when <code>createLeaseContainerIfNotExists</code> is set to <code>true</code> . This parameter is automatically set when the binding is created using the portal.
leaseContainerPrefix	(Optional) When set, the value is added as a prefix to the leases created in the Lease container for this function. Using a prefix allows two separate Azure Functions to share the same Lease container by using different prefixes.
feedPollDelay	(Optional) The time (in milliseconds) for the delay between polling a partition for new changes on the feed, after all current changes are drained. Default is 5,000 milliseconds, or 5 seconds.
leaseAcquireInterval	(Optional) When set, it defines, in milliseconds, the interval to kick off a task to compute if partitions are distributed evenly

function.json property	Description
	among known host instances. Default is 13000 (13 seconds).
leaseExpirationInterval	(Optional) When set, it defines, in milliseconds, the interval for which the lease is taken on a lease representing a partition. If the lease is not renewed within this interval, it will cause it to expire and ownership of the partition will move to another instance. Default is 60000 (60 seconds).
leaseRenewInterval	(Optional) When set, it defines, in milliseconds, the renew interval for all leases for partitions currently held by an instance. Default is 17000 (17 seconds).
maxItemsPerInvocation	(Optional) When set, this property sets the maximum number of items received per Function call. If operations in the monitored container are performed through stored procedures, transaction scope is preserved when reading items from the change feed. As a result, the number of items received could be higher than the specified value so that the items changed by the same transaction are returned as part of one atomic batch.
startFromBeginning	(Optional) This option tells the Trigger to read changes from the beginning of the container's change history instead of starting at the current time. Reading from the beginning only works the first time the trigger starts, as in subsequent runs, the checkpoints are already stored. Setting this option to <code>true</code> when there are leases already created has no effect.
startFromTime	(Optional) Gets or sets the date and time from which to initialize the change feed read operation. The recommended format is ISO 8601 with the UTC designator, such as <code>2021-02-16T14:19:29Z</code> . This is only used to set the initial trigger state. After the trigger has a lease state, changing this value has no effect.
preferredLocations	(Optional) Defines preferred locations (regions) for geo-replicated database accounts in the Azure Cosmos DB service. Values should be comma-separated. For example, "East US,South Central US,North Europe".

The following example shows an Azure Cosmos DB trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function writes log messages when Azure Cosmos DB records are added or modified.

Here's the binding data in the `function.json` file:

JSON

```
{
    "type": "cosmosDBTrigger",
    "name": "documents",
    "direction": "in",
    "leaseContainerName": "leases",
    "connection": "<connection-app-setting>",
    "databaseName": "Tasks",
    "containerName": "Items",
    "createLeaseContainerIfNotExists": true
}
```

Here's the C# script code:

C#

```
using System;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

// Customize the model with your own desired properties
public class ToDoItem
{
    public string id { get; set; }
    public string Description { get; set; }
}

public static void Run(IReadOnlyList<ToDoItem> documents, ILogger log)
{
    log.LogInformation("Documents modified " + documents.Count);
    log.LogInformation("First document Id " + documents[0].id);
}
```

Azure Cosmos DB v2 input

This section outlines support for the [version 4.x+ of the extension](#) only.

The following table explains the binding configuration properties that you set in the `function.json` file.

function.json property	Description
type	Must be set to <code>cosmosDB</code> .
direction	Must be set to <code>in</code> .
name	The variable name used in function code that represents the list of documents with changes.

function.json property	Description
connection	The name of an app setting or setting container that specifies how to connect to the Azure Cosmos DB account being monitored. For more information, see Connections .
databaseName	The name of the Azure Cosmos DB database with the container being monitored.
containerName	The name of the container being monitored.
partitionKey	Specifies the partition key value for the lookup. May include binding parameters. It is required for lookups in partitioned containers.
id	The ID of the document to retrieve. This property supports binding expressions . Don't set both the <code>id</code> and <code>sqlQuery</code> properties. If you don't set either one, the entire container is retrieved.
sqlQuery	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: <code>SELECT * FROM c WHERE c.departmentId = {departmentId}</code> . Don't set both the <code>id</code> and <code>sqlQuery</code> properties. If you don't set either one, the entire container is retrieved.
preferredLocations	(Optional) Defines preferred locations (regions) for geo-replicated database accounts in the Azure Cosmos DB service. Values should be comma-separated. For example, <code>East US,South Central US,North Europe</code> .

This section contains the following examples:

- [Queue trigger, look up ID from string](#)
- [Queue trigger, get multiple docs, using SqlQuery](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [HTTP trigger, get multiple docs, using SqlQuery](#)
- [HTTP trigger, get multiple docs, using DocumentClient](#)

The HTTP trigger examples refer to a simple `ToDoItem` type:

C#

```
namespace CosmosDBSamplesV2
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
```

```
    }  
}
```

Queue trigger, look up ID from string

The following example shows an Azure Cosmos DB input binding in a *function.json* file and a C# script function that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

JSON

```
{  
  "name": "inputDocument",  
  "type": "cosmosDB",  
  "databaseName": "MyDatabase",  
  "collectionName": "MyCollection",  
  "id" : "{queueTrigger}",  
  "partitionKey": "{partition key value}",  
  "connectionStringSetting": "MyAccount_COSMOSDB",  
  "direction": "in"  
}
```

Here's the C# script code:

C#

```
using System;  
  
// Change input document contents using Azure Cosmos DB input binding  
public static void Run(string myQueueItem, dynamic inputDocument)  
{  
    inputDocument.text = "This has changed.";  
}
```

Queue trigger, get multiple docs, using SqlQuery

The following example shows an Azure Cosmos DB input binding in a *function.json* file and a C# script function that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the *function.json* file:

JSON

```
{  
    "name": "documents",  
    "type": "cosmosDB",  
    "direction": "in",  
    "databaseName": "MyDb",  
    "collectionName": "MyCollection",  
    "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",  
    "connectionStringSetting": "CosmosDBConnection"  
}
```

Here's the C# script code:

C#

```
public static void Run(QueuePayload myQueueItem, IEnumerable<dynamic>  
documents)  
{  
    foreach (var doc in documents)  
    {  
        // operate on each document  
    }  
}  
  
public class QueuePayload  
{  
    public string departmentId { get; set; }  
}
```

HTTP trigger, look up ID from query string

The following example shows a C# script function that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID and partition key value to look up. That ID and partition key value are used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the *function.json* file:

JSON

```
{  
    "bindings": [  
        {  
            "authLevel": "anonymous",  
            "name": "req",  
            "type": "httpTrigger",  
            "direction": "in",  
            "method": "get",  
            "route": "GetItem/  
            "id": "  
            "partitionKey": "  
            "query": "  
            "host": "https://  
            "hostName": "myapp",  
            "port": 3001  
        }  
    ]  
}
```

```

    "type": "httpTrigger",
    "direction": "in",
    "methods": [
        "get",
        "post"
    ],
},
{
    "name": "$return",
    "type": "http",
    "direction": "out"
},
{
    "type": "cosmosDB",
    "name": "ToDoItem",
    "databaseName": "ToDoItems",
    "collectionName": "Items",
    "connectionStringSetting": "CosmosDBConnection",
    "direction": "in",
    "Id": "{Query.id}",
    "PartitionKey" : "{Query.partitionKeyValue}"
}
],
"disabled": false
}

```

Here's the C# script code:

```

C#

using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem
ToDoItem, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.LogInformation($"ToDo item not found");
    }
    else
    {
        log.LogInformation($"Found ToDo item, Description=
{ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}

```

HTTP trigger, look up ID from route data

The following example shows a C# script function that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID and partition key value to look up. That ID and partition key value are used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

```
JSON

{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ],
      "route": "todoitems/{partitionKeyValue}/{id}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "id": "{id}",
      "partitionKey": "{partitionKeyValue}"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
C#

using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem
  toItem, ILogger log)
{
```

```

    log.LogInformation("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.LogInformation($"ToDo item not found");
    }
    else
    {
        log.LogInformation($"Found ToDo item, Description=
{ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}

```

HTTP trigger, get multiple docs, using SqlQuery

The following example shows a C# script function that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

Here's the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItems",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "sqlQuery": "SELECT top 2 * FROM c order by c._ts desc"
    }
  ],
}
```

```
        "disabled": false
    }
```

Here's the C# script code:

C#

```
using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage Run(HttpContext req,
IEnumerable<ToDoItem> ToDoItems, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    foreach (ToDoItem ToDoItem in ToDoItems)
    {
        log.LogInformation(ToDoItem.Description);
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

HTTP trigger, get multiple docs, using DocumentClient

The following example shows a C# script function that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

Here's the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ]
}
```

```

},
{
  "type": "cosmosDB",
  "name": "client",
  "databaseName": "ToDoItems",
  "collectionName": "Items",
  "connectionStringSetting": "CosmosDBConnection",
  "direction": "inout"
}
],
"disabled": false
}

```

Here's the C# script code:

C#

```

#r "Microsoft.Azure.Documents.Client"

using System.Net;
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using Microsoft.Extensions.Logging;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,
DocumentClient client, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    Uri collectionUri = UriFactory.CreateDocumentCollectionUri("ToDoItems",
"Items");
    string searchterm = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "searchterm", true) == 0)
        .Value;

    if (searchterm == null)
    {
        return req.CreateResponse(HttpStatusCode.NotFound);
    }

    log.LogInformation($"Searching for word: {searchterm} using Uri:
{collectionUri.ToString()}");
    IDocumentQuery<ToDoItem> query = client.CreateDocumentQuery<ToDoItem>
(collectionUri)
    .Where(p => p.Description.Contains(searchterm))
    .AsDocumentQuery();

    while (query.HasMoreResults)
    {
        foreach (ToDoItem result in await query.ExecuteNextAsync())
        {
            log.LogInformation(result.Description);
        }
    }
}

```

```

    }
    return req.CreateResponse(HttpStatusCode.OK);
}

```

Azure Cosmos DB v2 output

This section outlines support for the [version 4.x+ of the extension](#) only.

The following table explains the binding configuration properties that you set in the `function.json` file.

function.json property	Description
<code>connection</code>	The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account being monitored. For more information, see Connections .
<code>databaseName</code>	The name of the Azure Cosmos DB database with the container being monitored.
<code>containerName</code>	The name of the container being monitored.
<code>createIfNotExists</code>	A boolean value to indicate whether the container is created when it doesn't exist. The default is <i>false</i> because new containers are created with reserved throughput, which has cost implications. For more information, see the pricing page .
<code>partitionKey</code>	When <code>createIfNotExists</code> is true, it defines the partition key path for the created container. May include binding parameters.
<code>containerThroughput</code>	When <code>createIfNotExists</code> is true, it defines the throughput of the created container.
<code>preferredLocations</code>	(Optional) Defines preferred locations (regions) for geo-replicated database accounts in the Azure Cosmos DB service. Values should be comma-separated. For example, <code>East US,South Central US,North Europe</code> .

This section contains the following examples:

- [Queue trigger, write one doc](#)
- [Queue trigger, write docs using IAsyncCollector](#)

Queue trigger, write one doc

The following example shows an Azure Cosmos DB output binding in a *function.json* file and a [C# script function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

JSON

```
{  
    "name": "John Henry",  
    "employeeId": "123456",  
    "address": "A town nearby"  
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

JSON

```
{  
    "id": "John Henry-123456",  
    "name": "John Henry",  
    "employeeId": "123456",  
    "address": "A town nearby"  
}
```

Here's the binding data in the *function.json* file:

JSON

```
{  
    "name": "employeeDocument",  
    "type": "cosmosDB",  
    "databaseName": "MyDatabase",  
    "collectionName": "MyCollection",  
    "createIfNotExists": true,  
    "connectionStringSetting": "MyAccount_COSMOSDB",  
    "direction": "out"  
}
```

Here's the C# script code:

C#

```
#r "Newtonsoft.Json"  
  
using Microsoft.Azure.WebJobs.Host;  
using Newtonsoft.Json.Linq;  
using Microsoft.Extensions.Logging;
```

```

    public static void Run(string myQueueItem, out object employeeDocument,
ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");

    dynamic employee = JObject.Parse(myQueueItem);

    employeeDocument = new {
        id = employee.name + "-" + employee.employeeId,
        name = employee.name,
        employeeId = employee.employeeId,
        address = employee.address
    };
}

```

Queue trigger, write docs using IAsyncCollector

To create multiple documents, you can bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

This example refers to a simple `ToDoItem` type:

C#

```

namespace CosmosDBSamplesV2
{
    public class ToDoItem
    {
        public string id { get; set; }
        public string Description { get; set; }
    }
}

```

Here's the function.json file:

JSON

```
{
  "bindings": [
    {
      "name": "ToDoItemsIn",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "todoqueueforwritemulti",
      "connectionStringSetting": "AzureWebJobsStorage"
    },
    {
      "type": "cosmosDB",

```

```

        "name": "ToDoItemsOut",
        "databaseName": "ToDoItems",
        "collectionName": "Items",
        "connectionStringSetting": "CosmosDBConnection",
        "direction": "out"
    }
],
"disabled": false
}

```

Here's the C# script code:

```
C#
using System;
using Microsoft.Extensions.Logging;

public static async Task Run(ToDoItem[] ToDoItemsIn,
IAsyncCollector<ToDoItem> ToDoItemsOut, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed
{ToDoItemsIn?.Length} items");

    foreach (ToDoItem ToDoItem in ToDoItemsIn)
    {
        log.LogInformation($"Description={ToDoItem.Description}");
        await ToDoItemsOut.AddAsync(ToDoItem);
    }
}
```

Azure Cosmos DB v1 trigger

The following example shows an Azure Cosmos DB trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function writes log messages when Azure Cosmos DB records are modified.

Here's the binding data in the *function.json* file:

JSON

```
{
    "type": "cosmosDBTrigger",
    "name": "documents",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "<connection-app-setting>",
    "databaseName": "Tasks",
    "collectionName": "Items",
```

```
        "createLeaseCollectionIfNotExists": true  
    }  
}
```

Here's the C# script code:

```
C#  
  
#r "Microsoft.Azure.Documents.Client"  
  
using System;  
using Microsoft.Azure.Documents;  
using System.Collections.Generic;  
  
public static void Run(IReadOnlyList<Document> documents, TraceWriter log)  
{  
    log.Info("Documents modified " + documents.Count);  
    log.Info("First document Id " + documents[0].Id);  
}
```

Azure Cosmos DB v1 input

This section contains the following examples:

- Queue trigger, look up ID from string
- Queue trigger, get multiple docs, using SqlQuery
- HTTP trigger, look up ID from query string
- HTTP trigger, look up ID from route data
- HTTP trigger, get multiple docs, using SqlQuery
- HTTP trigger, get multiple docs, using DocumentClient

The HTTP trigger examples refer to a simple `ToDoItem` type:

```
C#  
  
namespace CosmosDBSamplesV1  
{  
    public class ToDoItem  
    {  
        public string Id { get; set; }  
        public string Description { get; set; }  
    }  
}
```

Queue trigger, look up ID from string

The following example shows an Azure Cosmos DB input binding in a *function.json* file and a [C# script function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

```
JSON

{
    "name": "inputDocument",
    "type": "documentDB",
    "databaseName": "MyDatabase",
    "collectionName": "MyCollection",
    "id": "{queueTrigger}",
    "partitionKey": "{partition key value}",
    "connection": "MyAccount_COSMOSDB",
    "direction": "in"
}
```

Here's the C# script code:

```
C#

using System;

// Change input document contents using Azure Cosmos DB input binding
public static void Run(string myQueueItem, dynamic inputDocument)
{
    inputDocument.text = "This has changed.";
}
```

Queue trigger, get multiple docs, using SqlQuery

The following example shows an Azure Cosmos DB input binding in a *function.json* file and a [C# script function](#) that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the *function.json* file:

```
JSON

{
    "name": "documents",
```

```
"type": "documentdb",
"direction": "in",
"databaseName": "MyDb",
"collectionName": "MyCollection",
"sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",
"connection": "CosmosDBConnection"
}
```

Here's the C# script code:

C#

```
public static void Run(QueuePayload myQueueItem, IEnumerable<dynamic>
documents)
{
    foreach (var doc in documents)
    {
        // operate on each document
    }
}

public class QueuePayload
{
    public string departmentId { get; set; }
}
```

HTTP trigger, look up ID from query string

The following example shows a [C# script function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "myDatabase",
      "type": "cosmosDB",
      "direction": "out",
      "databaseName": "MyDb",
      "collectionName": "MyCollection"
    }
  ],
  "triggers": []
}
```

```
{
  "name": "$return",
  "type": "http",
  "direction": "out"
},
{
  "type": "documentDB",
  "name": "ToDoItem",
  "databaseName": "ToDoItems",
  "collectionName": "Items",
  "connection": "CosmosDBConnection",
  "direction": "in",
  "Id": "{Query.id}"
}
],
"disabled": true
}
```

Here's the C# script code:

```
C#
using System.Net;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem
ToDoItem, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.Info($"ToDo item not found");
    }
    else
    {
        log.Info($"Found ToDo item, Description={ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

HTTP trigger, look up ID from route data

The following example shows a [C# script function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

JSON

```
{  
  "bindings": [  
    {  
      "authLevel": "anonymous",  
      "name": "req",  
      "type": "httpTrigger",  
      "direction": "in",  
      "methods": [  
        "get",  
        "post"  
      ],  
      "route": "todoitems/{id}"  
    },  
    {  
      "name": "$return",  
      "type": "http",  
      "direction": "out"  
    },  
    {  
      "type": "documentDB",  
      "name": "ToDoItem",  
      "databaseName": "ToDoItems",  
      "collectionName": "Items",  
      "connection": "CosmosDBConnection",  
      "direction": "in",  
      "Id": "{id}"  
    }  
  ],  
  "disabled": false  
}
```

Here's the C# script code:

C#

```
using System.Net;  
  
public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem  
toDoItem, TraceWriter log)  
{  
    log.Info("C# HTTP trigger function processed a request.");  
  
    if (toDoItem == null)  
    {  
        log.Info($"ToDo item not found");  
    }  
    else  
    {  
        log.Info($"Found ToDo item, Description={toDoItem.Description}");  
    }  
}
```

```
        return req.CreateResponse(HttpStatusCode.OK);
    }
```

HTTP trigger, get multiple docs, using SqlQuery

The following example shows a [C# script function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

Here's the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "ToDoItems",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "sqlQuery": "SELECT top 2 * FROM c ORDER BY c._ts DESC"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

C#

```
using System.Net;
```

```

public static HttpResponseMessage Run(HttpRequestMessage req,
IEnumerable<ToDoItem> ToDoItems, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    foreach (ToDoItem ToDoItem in ToDoItems)
    {
        log.Info(ToDoItem.Description);
    }
    return req.CreateResponse(HttpStatusCode.OK);
}

```

HTTP trigger, get multiple docs, using DocumentClient

The following example shows a [C# script function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

Here's the `function.json` file:

JSON

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "client",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "inout"
    }
  ],
}
```

```
        "disabled": false
    }
```

Here's the C# script code:

```
C#  
  
#r "Microsoft.Azure.Documents.Client"  
  
using System.Net;  
using Microsoft.Azure.Documents.Client;  
using Microsoft.Azure.Documents.Linq;  
  
public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,  
DocumentClient client, TraceWriter log)  
{  
    log.Info("C# HTTP trigger function processed a request.");  
  
    Uri collectionUri = UriFactory.CreateDocumentCollectionUri("ToDoItems",  
"Items");  
    string searchterm = req.GetQueryNameValuePairs()  
        .FirstOrDefault(q => string.Compare(q.Key, "searchterm", true) == 0)  
        .Value;  
  
    if (searchterm == null)  
    {  
        return req.CreateResponse(HttpStatusCode.NotFound);  
    }  
  
    log.Info($"Searching for word: {searchterm} using Uri:  
{collectionUri.ToString()}");  
    IDocumentQuery<ToDoItem> query = client.CreateDocumentQuery<ToDoItem>(  
collectionUri)  
        .Where(p => p.Description.Contains(searchterm))  
        .AsDocumentQuery();  
  
    while (query.HasMoreResults)  
    {  
        foreach (ToDoItem result in await query.ExecuteNextAsync())  
        {  
            log.Info(result.Description);  
        }  
    }  
    return req.CreateResponse(HttpStatusCode.OK);  
}
```

Azure Cosmos DB v1 output

This section contains the following examples:

- Queue trigger, write one doc

- Queue trigger, write docs using `IAsyncCollector`

Queue trigger, write one doc

The following example shows an Azure Cosmos DB output binding in a `function.json` file and a [C# script function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

JSON

```
{
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

JSON

```
{
  "id": "John Henry-123456",
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

Here's the binding data in the `function.json` file:

JSON

```
{
  "name": "employeeDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "createIfNotExists": true,
  "connection": "MyAccount_COSMOSDB",
  "direction": "out"
}
```

Here's the C# script code:

C#

```

#r "Newtonsoft.Json"

using Microsoft.Azure.WebJobs.Host;
using Newtonsoft.Json.Linq;

public static void Run(string myQueueItem, out object employeeDocument,
TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    dynamic employee = JObject.Parse(myQueueItem);

    employeeDocument = new {
        id = employee.name + "-" + employee.employeeId,
        name = employee.name,
        employeeId = employee.employeeId,
        address = employee.address
    };
}

```

Queue trigger, write docs using IAsyncCollector

To create multiple documents, you can bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

This example refers to a simple `ToDoItem` type:

C#

```

namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}

```

Here's the function.json file:

JSON

```
{
  "bindings": [
    {
      "name": "ToDoItemsIn",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "todoqueueforwritemulti",
      "connection": "AzureWebJobsStorage"
    }
  ]
}
```

```

        "connection": "AzureWebJobsStorage"
    },
    {
        "type": "documentDB",
        "name": "ToDoItemsOut",
        "databaseName": "ToDoItems",
        "collectionName": "Items",
        "connection": "CosmosDBConnection",
        "direction": "out"
    }
],
"disabled": false
}

```

Here's the C# script code:

C#

```

using System;

public static async Task Run(ToDoItem[] ToDoItemsIn,
    IAsyncCollector<ToDoItem> ToDoItemsOut, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed {ToDoItemsIn?.Length} items");

    foreach (ToDoItem ToDoItem in ToDoItemsIn)
    {
        log.Info($"Description={ToDoItem.Description}");
        await ToDoItemsOut.AddAsync(ToDoItem);
    }
}

```

Azure SQL trigger

More samples for the Azure SQL trigger are available in the [GitHub repository](#).

The example refers to a `ToDoItem` class and a corresponding database table:

C#

```

namespace AzureSQL.ToDo
{
    public class ToDoItem
    {
        public Guid Id { get; set; }
        public int? order { get; set; }
        public string title { get; set; }
        public string url { get; set; }
        public bool? completed { get; set; }
    }
}

```

```
    }  
}
```

SQL

```
CREATE TABLE dbo.ToDo (  
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,  
    [order] INT NULL,  
    [title] NVARCHAR(200) NOT NULL,  
    [url] NVARCHAR(200) NOT NULL,  
    [completed] BIT NOT NULL  
);
```

Change tracking is enabled on the database and on the table:

SQL

```
ALTER DATABASE [SampleDatabase]  
SET CHANGE_TRACKING = ON  
(CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON);  
  
ALTER TABLE [dbo].[ToDo]  
ENABLE CHANGE_TRACKING;
```

The SQL trigger binds to a `IReadOnlyList<SqlChange<T>>`, a list of `SqlChange` objects each with two properties:

- **Item:** the item that was changed. The type of the item should follow the table schema as seen in the `ToDoItem` class.
- **Operation:** a value from `SqlChangeOperation` enum. The possible values are `Insert`, `Update`, and `Delete`.

The following example shows a SQL trigger in a function.json file and a [C# script function](#) that is invoked when there are changes to the `ToDo` table:

The following is binding data in the function.json file:

JSON

```
{  
    "name": "todoChanges",  
    "type": "sqlTrigger",  
    "direction": "in",  
    "tableName": "dbo.ToDo",  
    "connectionStringSetting": "SqlConnectionString"  
}
```

The following is the C# script function:

```
C#  
  
#r "Newtonsoft.Json"  
  
using System.Net;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Extensions.Primitives;  
using Newtonsoft.Json;  
  
public static void Run(IReadOnlyList<SqlChange<ToDoItem>> todoChanges,  
ILogger log)  
{  
    log.LogInformation($"C# SQL trigger function processed a request.");  
  
    foreach (SqlChange<ToDoItem> change in todoChanges)  
    {  
        ToDoItem ToDoItem = change.Item;  
        log.LogInformation($"Change operation: {change.Operation}");  
        log.LogInformation($"Id: {ToDoItem.Id}, Title: {ToDoItem.title},  
Url: {ToDoItem.url}, Completed: {ToDoItem.completed}");  
    }  
}
```

Azure SQL input

More samples for the Azure SQL input binding are available in the [GitHub repository](#).

This section contains the following examples:

- [HTTP trigger, get row by ID from query string](#)
- [HTTP trigger, delete rows](#)

The examples refer to a `ToDoItem` class and a corresponding database table:

```
C#  
  
namespace AzureSQL.ToDo  
{  
    public class ToDoItem  
    {  
        public Guid Id { get; set; }  
        public int? order { get; set; }  
        public string title { get; set; }  
        public string url { get; set; }  
        public bool? completed { get; set; }  
    }  
}
```

SQL

```
CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

HTTP trigger, get row by ID from query string

The following example shows an Azure SQL input binding in a *function.json* file and a [C# script function](#) that uses the binding. The function is triggered by an HTTP request that uses a query string to specify the ID. That ID is used to retrieve a `ToDoItem` record with the specified query.

ⓘ Note

The HTTP query string parameter is case-sensitive.

Here's the binding data in the *function.json* file:

JSON

```
{
  "authLevel": "anonymous",
  "type": "httpTrigger",
  "direction": "in",
  "name": "req",
  "methods": [
    "get"
  ],
  {
    "type": "http",
    "direction": "out",
    "name": "res"
  },
  {
    "name": "todoItem",
    "type": "sql",
    "direction": "in",
    "commandText": "select [Id], [order], [title], [url], [completed] from dbo.ToDo where Id = @Id",
    "commandType": "Text",
    "parameters": "@Id = {Query.id}",
  }
}
```

```
        "connectionStringSetting": "SqlConnectionString"
    }
```

Here's the C# script code:

C#

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using System.Collections.Generic;

public static IActionResult Run(HttpContext req, ILogger log,
IEnumerable<ToDoItem> todoItem)
{
    return new OkObjectResult(todoItem);
}
```

HTTP trigger, delete rows

The following example shows an Azure SQL input binding in a `function.json` file and a [C# script function](#) that uses the binding to execute a stored procedure with input from the HTTP request query parameter. In this example, the stored procedure deletes a single record or all records depending on the value of the parameter.

The stored procedure `dbo.DeleteToDo` must be created on the SQL database.

SQL

```
CREATE PROCEDURE [dbo].[DeleteToDo]
@Id NVARCHAR(100)
AS
DECLARE @UID UNIQUEIDENTIFIER = TRY_CAST(@ID AS UNIQUEIDENTIFIER)
IF @UID IS NOT NULL AND @Id != ''
BEGIN
    DELETE FROM dbo.ToDo WHERE Id = @UID
END
ELSE
BEGIN
    DELETE FROM dbo.ToDo WHERE @ID = ''
END

SELECT [Id], [order], [title], [url], [completed] FROM dbo.ToDo
GO
```

Here's the binding data in the *function.json* file:

JSON

```
{  
    "authLevel": "anonymous",  
    "type": "httpTrigger",  
    "direction": "in",  
    "name": "req",  
    "methods": [  
        "get"  
    ]  
},  
{  
    "type": "http",  
    "direction": "out",  
    "name": "res"  
},  
{  
    "name": "todoItems",  
    "type": "sql",  
    "direction": "in",  
    "commandText": "DeleteToDo",  
    "commandType": "StoredProcedure",  
    "parameters": "@Id = {Query.id}",  
    "connectionStringSetting": "SqlConnectionString"  
}
```

C#

```
using System;  
using System.Collections.Generic;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.AspNetCore.Http;  
using Microsoft.Extensions.Logging;  
  
namespace AzureSQL.ToDo  
{  
    public static class DeleteToDo  
    {  
        // delete all items or a specific item from querystring  
        // returns remaining items  
        // uses input binding with a stored procedure DeleteToDo to delete  
        items and return remaining items  
        [FunctionName("DeleteToDo")]  
        public static IActionResult Run(  
            [HttpTrigger(AuthorizationLevel.Anonymous, "delete", Route =  
"DeleteFunction")] HttpRequest req,  
            ILogger log,  
            [Sql(commandText: "DeleteToDo", commandType:  
System.Data.CommandType.StoredProcedure,
```

```

        parameters: "@Id={Query.id}", connectionStringSetting:
"SqlConnectionString")]
    IEnumerable<ToDoItem> ToDoItems)
{
    return new OkObjectResult(ToDoItems);
}
}
}

```

Here's the C# script code:

```

C#

#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using System.Collections.Generic;

public static IActionResult Run(HttpContext req, ILogger log,
IEnumerable<ToDoItem> todoItems)
{
    return new OkObjectResult(todoItems);
}

```

Azure SQL output

More samples for the Azure SQL output binding are available in the [GitHub repository](#).

This section contains the following examples:

- [HTTP trigger, write records to a table](#)
- [HTTP trigger, write to two tables](#)

The examples refer to a `ToDoItem` class and a corresponding database table:

```

C#

namespace AzureSQL.ToDo
{
    public class ToDoItem
    {
        public Guid Id { get; set; }
        public int? Order { get; set; }
        public string Title { get; set; }
        public string Url { get; set; }
    }
}

```

```
        public bool? completed { get; set; }  
    }  
}
```

SQL

```
CREATE TABLE dbo.ToDo (  
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,  
    [order] INT NULL,  
    [title] NVARCHAR(200) NOT NULL,  
    [url] NVARCHAR(200) NOT NULL,  
    [completed] BIT NOT NULL  
);
```

HTTP trigger, write records to a table

The following example shows a SQL output binding in a function.json file and a [C# script function](#) that adds records to a table, using data provided in an HTTP POST request as a JSON body.

The following is binding data in the function.json file:

JSON

```
{  
    "authLevel": "anonymous",  
    "type": "httpTrigger",  
    "direction": "in",  
    "name": "req",  
    "methods": [  
        "post"  
    ]  
},  
{  
    "type": "http",  
    "direction": "out",  
    "name": "res"  
},  
{  
    "name": "todoItem",  
    "type": "sql",  
    "direction": "out",  
    "commandText": "dbo.ToDo",  
    "connectionStringSetting": "SqlConnectionString"  
}
```

The following is sample C# script code:

C#

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static IActionResult Run(HttpContext req, ILogger log, out ToDoItem todoItem)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string requestBody = new StreamReader(req.Body).ReadToEnd();
    todoItem = JsonConvert.DeserializeObject<ToDoItem>(requestBody);

    return new OkObjectResult(todoItem);
}
```

HTTP trigger, write to two tables

The following example shows a SQL output binding in a function.json file and a [C# script function](#) that adds records to a database in two different tables (`dbo.ToDo` and `dbo.RequestLog`), using data provided in an HTTP POST request as a JSON body and multiple output bindings.

The second table, `dbo.RequestLog`, corresponds to the following definition:

SQL

```
CREATE TABLE dbo.RequestLog (
    Id int identity(1,1) primary key,
    RequestTimeStamp datetime2 not null,
    ItemCount int not null
)
```

The following is binding data in the function.json file:

JSON

```
{
    "authLevel": "anonymous",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
        "post"
    ]}
```

```

        ],
    },
    {
        "type": "http",
        "direction": "out",
        "name": "res"
    },
    {
        "name": "todoItem",
        "type": "sql",
        "direction": "out",
        "commandText": "dbo.ToDo",
        "connectionStringSetting": "SqlConnectionString"
    },
    {
        "name": "requestLog",
        "type": "sql",
        "direction": "out",
        "commandText": "dbo.RequestLog",
        "connectionStringSetting": "SqlConnectionString"
    }
}

```

The following is sample C# script code:

C#

```

#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static IActionResult Run(HttpContext req, ILogger log, out ToDoItem
todoItem, out RequestLog requestLog)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string requestBody = new StreamReader(req.Body).ReadToEnd();
    todoItem = JsonConvert.DeserializeObject<ToDoItem>(requestBody);

    requestLog = new RequestLog();
    requestLog.RequestTimeStamp = DateTime.Now;
    requestLog.ItemCount = 1;

    return new OkObjectResult(todoItem);
}

public class RequestLog {
    public DateTime RequestTimeStamp { get; set; }
    public int ItemCount { get; set; }
}

```

RabbitMQ output

The following example shows a RabbitMQ output binding in a *function.json* file and a [C# script function](#) that uses the binding. The function reads in the message from an HTTP trigger and outputs it to the RabbitMQ queue.

Here's the binding data in the *function.json* file:

```
JSON

{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "function",
      "name": "input",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "rabbitMQ",
      "name": "outputMessage",
      "queueName": "outputQueue",
      "connectionStringSetting": "rabbitMQConnectionStringAppSetting",
      "direction": "out"
    }
  ]
}
```

Here's the C# script code:

```
C#

using System;
using Microsoft.Extensions.Logging;

public static void Run(string input, out string outputMessage, ILogger log)
{
    log.LogInformation(input);
    outputMessage = input;
}
```

SendGrid output

The following example shows a SendGrid output binding in a *function.json* file and a [C# script function](#) that uses the binding.

Here's the binding data in the *function.json* file:

JSON

```
{  
  "bindings": [  
    {  
      "type": "queueTrigger",  
      "name": "mymsg",  
      "queueName": "myqueue",  
      "connection": "AzureWebJobsStorage",  
      "direction": "in"  
    },  
    {  
      "type": "sendGrid",  
      "name": "$return",  
      "direction": "out",  
      "apiKey": "SendGridAPIKeyAsAppSetting",  
      "from": "{FromEmail}",  
      "to": "{ToEmail}"  
    }  
  ]  
}
```

Here's the C# script code:

C#

```
#r "SendGrid"  
  
using System;  
using SendGrid.Helpers.Mail;  
using Microsoft.Azure.WebJobs.Host;  
  
public static SendGridMessage Run(Message mymsg, ILogger log)  
{  
    SendGridMessage message = new SendGridMessage()  
    {  
        Subject = $"{mymsg.Subject}"  
    };  
  
    message.AddContent("text/plain", $"{mymsg.Content}");  
  
    return message;  
}  
public class Message  
{  
    public string ToEmail { get; set; }  
    public string FromEmail { get; set; }  
}
```

```
    public string Subject { get; set; }
    public string Content { get; set; }
}
```

SignalR trigger

Here's example binding data in the *function.json* file:

JSON

```
{
  "type": "signalRTrigger",
  "name": "invocation",
  "hubName": "SignalRTTest",
  "category": "messages",
  "event": "SendMessage",
  "parameterNames": [
    "message"
  ],
  "direction": "in"
}
```

And, here's the code:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"
using System;
using Microsoft.Azure.WebJobs.Extensions.SignalRService;
using Microsoft.Extensions.Logging;

public static void Run(InvocationContext invocation, string message, ILogger logger)
{
    logger.LogInformation($"Receive {message} from
{invocationContext.ConnectionId}.");
}
```

SignalR input

The following example shows a SignalR connection info input binding in a *function.json* file and a [C# Script function](#) that uses the binding to return the connection information.

Here's binding data in the *function.json* file:

Example *function.json*:

JSON

```
{  
    "type": "signalRConnectionInfo",  
    "name": "connectionInfo",  
    "hubName": "chat",  
    "connectionStringSetting": "<name of setting containing SignalR Service  
connection string>",  
    "direction": "in"  
}
```

Here's the C# Script code:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"  
using Microsoft.Azure.WebJobs.Extensions.SignalRService;  
  
public static SignalRConnectionInfo Run(WebRequest req,  
SignalRConnectionInfo connectionInfo)  
{  
    return connectionInfo;  
}
```

You can set the `userId` property of the binding to the value from either header using a [binding expression](#): `{headers.x-ms-client-principal-id}` or `{headers.x-ms-client-principal-name}`.

Example function.json:

JSON

```
{  
    "type": "signalRConnectionInfo",  
    "name": "connectionInfo",  
    "hubName": "chat",  
    "userId": "{headers.x-ms-client-principal-id}",  
    "connectionStringSetting": "<name of setting containing SignalR Service  
connection string>",  
    "direction": "in"  
}
```

Here's the C# Script code:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"  
using Microsoft.Azure.WebJobs.Extensions.SignalRService;
```

```
public static SignalRConnectionInfo Run(HttpRequest req,
SignalRConnectionInfo connectionInfo)
{
    // connectionInfo contains an access key token with a name identifier
    // claim set to the authenticated user
    return connectionInfo;
}
```

SignalR output

Here's binding data in the *function.json* file:

Example *function.json*:

JSON

```
{
    "type": "signalR",
    "name": "signalRMessages",
    "hubName": "<hub_name>",
    "connectionStringSetting": "<name of setting containing SignalR Service
connection string>",
    "direction": "out"
}
```

Here's the C# Script code:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"
using Microsoft.Azure.WebJobs.Extensions.SignalRService;

public static Task Run(
    object message,
    IAsyncCollector<SignalRMessage> signalRMessages)
{
    return signalRMessages.AddAsync(
        new SignalRMessage
        {
            Target = "newMessage",
            Arguments = new [] { message }
        });
}
```

You can send a message only to connections that have been authenticated to a user by setting the *user ID* in the SignalR message.

Example function.json:

JSON

```
{  
  "type": "signalR",  
  "name": "signalRMessages",  
  "hubName": "<hub_name>",  
  "connectionStringSetting": "<name of setting containing SignalR Service  
connection string>",  
  "direction": "out"  
}
```

Here's the C# script code:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"  
using Microsoft.Azure.WebJobs.Extensions.SignalRService;  
  
public static Task Run(  
    object message,  
    IAsyncCollector<SignalRMessage> signalRMessages)  
{  
    return signalRMessages.AddAsync(  
        new SignalRMessage  
        {  
            // the message will only be sent to this user ID  
            UserId = "userId1",  
            Target = "newMessage",  
            Arguments = new [] { message }  
        });  
}
```

You can send a message only to connections that have been added to a group by setting the *group name* in the SignalR message.

Example function.json:

JSON

```
{  
  "type": "signalR",  
  "name": "signalRMessages",  
  "hubName": "<hub_name>",  
  "connectionStringSetting": "<name of setting containing SignalR Service  
connection string>",  
  "direction": "out"  
}
```

Here's the C# Script code:

```
C#  
  
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"  
using Microsoft.Azure.WebJobs.Extensions.SignalRService;  
  
public static Task Run(  
    object message,  
    IAsyncCollector<SignalRMessage> signalRMessages)  
{  
    return signalRMessages.AddAsync(  
        new SignalRMessage  
        {  
            // the message will be sent to the group with this name  
            GroupName = "myGroup",  
            Target = "newMessage",  
            Arguments = new [] { message }  
        });  
}
```

SignalR Service allows users or connections to be added to groups. Messages can then be sent to a group. You can use the `SignalR` output binding to manage groups.

The following example adds a user to a group.

Example `function.json`

```
JSON  
  
{  
    "type": "signalR",  
    "name": "signalRGroupActions",  
    "connectionStringSetting": "<name of setting containing SignalR Service connection string>",  
    "hubName": "chat",  
    "direction": "out"  
}
```

`Run.csx`

```
C#  
  
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"  
using Microsoft.Azure.WebJobs.Extensions.SignalRService;  
  
public static Task Run(  
    HttpRequest req,  
    ClaimsPrincipal claimsPrincipal,  
    IAsyncCollector<SignalRGroupAction> signalRGroupActions)
```

```
{
    var userIdClaim = claimsPrincipal.FindFirst(ClaimTypes.NameIdentifier);
    return signalRGroupActions.AddAsync(
        new SignalRGroupAction
    {
        UserId = userIdClaim.Value,
        GroupName = "myGroup",
        Action = GroupAction.Add
    });
}
```

The following example removes a user from a group.

Example *function.json*

JSON

```
{
    "type": "signalR",
    "name": "signalRGroupActions",
    "connectionStringSetting": "<name of setting containing SignalR Service connection string>",
    "hubName": "chat",
    "direction": "out"
}
```

Run.csx

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.SignalRService"
using Microsoft.Azure.WebJobs.Extensions.SignalRService;

public static Task Run(
    HttpRequest req,
    ClaimsPrincipal claimsPrincipal,
    IAsyncCollector<SignalRGroupAction> signalRGroupActions)
{
    var userIdClaim = claimsPrincipal.FindFirst(ClaimTypes.NameIdentifier);
    return signalRGroupActions.AddAsync(
        new SignalRGroupAction
    {
        UserId = userIdClaim.Value,
        GroupName = "myGroup",
        Action = GroupAction.Remove
    });
}
```

Twilio output

The following example shows a Twilio output binding in a `function.json` file and a [C# script function](#) that uses the binding. The function uses an `out` parameter to send a text message.

Here's binding data in the `function.json` file:

Example `function.json`:

```
JSON

{
  "type": "twilioSms",
  "name": "message",
  "accountSidSetting": "TwilioAccountSid",
  "authTokenSetting": "TwilioAuthToken",
  "from": "+1425XXXXXXX",
  "direction": "out",
  "body": "Azure Functions Testing"
}
```

Here's C# script code:

```
C#

#r "Newtonsoft.Json"
#r "Twilio"
#r "Microsoft.Azure.WebJobs.Extensions.Twilio"

using System;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Microsoft.Azure.WebJobs.Extensions.Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

public static void Run(string myQueueItem, out CreateMessageOptions message,
ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");

    // In this example the queue item is a JSON string representing an order
    // that contains the name of a
    // customer and a mobile number to send text updates to.
    dynamic order = JsonConvert.DeserializeObject(myQueueItem);
    string msg = "Hello " + order.name + ", thank you for your order.';

    // You must initialize the CreateMessageOptions variable with the "To"
    // phone number.
    message = new CreateMessageOptions(new PhoneNumber("+1704XXXXXXX"));
}
```

```
// A dynamic message can be set instead of the body in the output
binding. In this example, we use
// the order information to personalize a text message.
message.Body = msg;
}
```

You can't use out parameters in asynchronous code. Here's an asynchronous C# script code example:

C#

```
#r "Newtonsoft.Json"
#r "Twilio"
#r "Microsoft.Azure.WebJobs.Extensions.Twilio"

using System;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Microsoft.Azure.WebJobs.Extensions.Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

public static async Task Run(string myQueueItem,
IAsyncCollector<CreateMessageOptions> message, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");

    // In this example the queue item is a JSON string representing an order
    // that contains the name of a
    // customer and a mobile number to send text updates to.
    dynamic order = JsonConvert.DeserializeObject(myQueueItem);
    string msg = "Hello " + order.name + ", thank you for your order.";

    // You must initialize the CreateMessageOptions variable with the "To"
    // phone number.
    CreateMessageOptions smsText = new CreateMessageOptions(new
    PhoneNumber("+1704XXXXXXX"));

    // A dynamic message can be set instead of the body in the output
    // binding. In this example, we use
    // the order information to personalize a text message.
    smsText.Body = msg;

    await message.AddAsync(smsText);
}
```

Warmup trigger

The following example shows a warmup trigger in a `function.json` file and a [C# script function](#) that runs on each new instance when it's added to your app.

Not supported for version 1.x of the Functions runtime.

Here's the `function.json` file:

JSON

```
{  
  "bindings": [  
    {  
      "type": "warmupTrigger",  
      "direction": "in",  
      "name": "warmupContext"  
    }  
  ]  
}
```

C#

```
public static void Run(WarmupContext warmupContext, ILogger log)  
{  
    log.LogInformation("Function App instance is warm.");  
}
```

Next steps

[Learn more about triggers and bindings](#)

[Learn more about best practices for Azure Functions](#)

Differences between the isolated worker model and the in-process model for .NET on Azure Functions

Article • 06/17/2024

There are two execution models for .NET functions:

 Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

This article describes the current state of the functional and behavioral differences between the two models. To migrate from the in-process model to the isolated worker model, see [Migrate .NET apps from the in-process model to the isolated worker model](#).

Execution model comparison table

Use the following table to compare feature and functional differences between the two models:

 Expand table

Feature/behavior	Isolated worker model	In-process model ³
Supported .NET versions	Long Term Support (LTS) versions, Standard Term Support (STS) versions, .NET Framework	Long Term Support (LTS) versions, ending with .NET 8

Feature/behavior	Isolated worker model	In-process model ³
Core packages	Microsoft.Azure.Functions.Worker Microsoft.Azure.Functions.Worker.Sdk	Microsoft.NET.Sdk.Functions
Binding extension packages	Microsoft.Azure.Functions.Worker.Extensions.*	Microsoft.Azure.WebJobs.Extensions.*
Durable Functions	Supported	Supported
Model types exposed by bindings	Simple types JSON serializable types Arrays/enumerations Service SDK types⁴	Simple types JSON serializable types Arrays/enumerations Service SDK types ⁴
HTTP trigger model types	HttpRequestData / HttpResponseData HttpRequest / IActionResult (using ASP.NET Core integration) ⁵	HttpRequest / IActionResult HttpRequestMessage / HttpResponseMessage
Output binding interactions	Return values in an expanded model with: - single or multiple outputs - arrays of outputs	Return values (single output only), <code>out</code> parameters, IAsyncCollector
Imperative bindings ¹	Not supported - instead work with SDK types directly	Supported
Dependency injection	Supported (improved model consistent with .NET ecosystem)	Supported
Middleware	Supported	Not supported
Logging	ILogger<T>/ILogger obtained from FunctionContext or via dependency injection	ILogger passed to the function ILogger<T> via dependency injection
Application Insights dependencies	Supported	Supported
Cancellation tokens	Supported	Supported
Cold start times ²	Configurable optimizations	Optimized
ReadyToRun	Supported	Supported
[Flex Consumption]	Supported	Not supported

¹ When you need to interact with a service using parameters determined at runtime, using the corresponding service SDKs directly is recommended over using imperative bindings. The SDKs are less verbose, cover more scenarios, and have advantages for error handling and debugging purposes. This recommendation applies to both models.

² Cold start times could be additionally impacted on Windows when using some preview versions of .NET due to just-in-time loading of preview frameworks. This impact applies to both the in-process and out-of-process models but can be noticeable when comparing across different versions. This delay for preview versions isn't present on Linux plans.

³ C# Script functions also run in-process and use the same libraries as in-process class library functions. For more information, see the [Azure Functions C# script \(.csx\) developer reference](#).

⁴ Service SDK types include types from the [Azure SDK for .NET](#) such as `BlobClient`.

⁵ ASP.NET Core types are not supported for .NET Framework.

Supported versions

Versions of the Functions runtime support specific versions of .NET. To learn more about Functions versions, see [Azure Functions runtime versions overview](#). Version support also depends on whether your functions run in-process or isolated worker process.

Note

To learn how to change the Functions runtime version used by your function app, see [view and update the current runtime version](#).

The following table shows the highest level of .NET or .NET Framework that can be used with a specific version of Functions.

 [Expand table](#)

Functions runtime version	Isolated worker model	In-process model ⁵
Functions 4.x ¹	.NET 8.0 .NET 6.0 ² .NET Framework 4.8 ³	.NET 8.0 .NET 6.0 ²
Functions 1.x ⁴	n/a	.NET Framework 4.8

¹ .NET 7 was previously supported on the isolated worker model but reached the [end of official support](#) on May 14, 2024.

² .NET 6 reaches the [end of official support](#) on November 12, 2024.

³ The build process also requires the [.NET SDK](#).

⁴ Support ends for version 1.x of the Azure Functions runtime on September 14, 2026. For more information, see [this support announcement](#). For continued full support, you should [migrate your apps to version 4.x](#).

⁵ Support ends for the in-process model on November 10, 2026. For more information, see [this support announcement](#). For continued full support, you should [migrate your apps to the isolated worker model](#).

For the latest news about Azure Functions releases, including the removal of specific older minor versions, monitor [Azure App Service announcements](#).

Next steps

[Learn more about the isolated worker model](#)

[Migrate to the isolated worker model](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Azure Functions Node.js developer guide

Article • 02/28/2024

This guide is an introduction to developing Azure Functions using JavaScript or TypeScript. The article assumes that you have already read the [Azure Functions developer guide](#).

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of this page. The version you choose should match the version of the [@azure/functions](#) npm package you are using in your app. If you do not have that package listed in your `package.json`, the default is v3. Learn more about the differences between v3 and v4 in the [migration guide](#).

As a Node.js developer, you might also be interested in one of the following articles:

 Expand table

Getting started	Concepts	Guided learning
<ul style="list-style-type: none">Node.js function using Visual Studio CodeNode.js function with terminal/command promptNode.js function using the Azure portal	<ul style="list-style-type: none">Developer guideHosting optionsPerformance considerations	<ul style="list-style-type: none">Create serverless applicationsRefactor Node.js and Express APIs to Serverless APIs

Considerations

- The Node.js programming model shouldn't be confused with the Azure Functions runtime:
 - **Programming model:** Defines how you author your code and is specific to JavaScript and TypeScript.
 - **Runtime:** Defines underlying behavior of Azure Functions and is shared across all languages.
- The version of the programming model is strictly tied to the version of the [@azure/functions](#) npm package. It's versioned independently of the **runtime**.

Both the runtime and the programming model use the number 4 as their latest major version, but that's a coincidence.

- You can't mix the v3 and v4 programming models in the same function app. As soon as you register one v4 function in your app, any v3 functions registered in *function.json* files are ignored.

Supported versions

The following table shows each version of the Node.js programming model along with its supported versions of the Azure Functions runtime and Node.js.

[\[+\] Expand table](#)

Programming Model Version ↗	Support Level	Functions Runtime Version	Node.js Version ↗	Description
4.x	GA	4.25+	20.x, 18.x	Supports a flexible file structure and code-centric approach to triggers and bindings.
3.x	GA	4.x	20.x, 18.x, 16.x, 14.x	Requires a specific file structure with your triggers and bindings declared in a "function.json" file
2.x	n/a	3.x	14.x, 12.x, 10.x	Reached end of support on December 13, 2022. See Functions Versions for more info.
1.x	n/a	2.x	10.x, 8.x	Reached end of support on December 13, 2022. See Functions Versions for more info.

Folder structure

JavaScript

The recommended folder structure for a JavaScript project looks like the following example:

```
<project_root>
| - .vscode/
| - node_modules/
| - src/
| | - functions/
| | | - myFirstFunction.js
| | | - mySecondFunction.js
| - test/
| | - functions/
| | | - myFirstFunction.test.js
| | | - mySecondFunction.test.js
| - .funcignore
| - host.json
| - local.settings.json
| - package.json
```

The main project folder, `<project_root>`, can contain the following files:

- **.vscode/**: (Optional) Contains the stored Visual Studio Code configuration. To learn more, see [Visual Studio Code settings](#).
- **src/functions/**: The default location for all functions and their related triggers and bindings.
- **test/**: (Optional) Contains the test cases of your function app.
- **.funcignore**: (Optional) Declares files that shouldn't get published to Azure. Usually, this file contains `.vscode/` to ignore your editor setting, `test/` to ignore test cases, and `local.settings.json` to prevent local app settings being published.
- **host.json**: Contains configuration options that affect all functions in a function app instance. This file does get published to Azure. Not all options are supported when running locally. To learn more, see [host.json](#).
- **local.settings.json**: Used to store app settings and connection strings when it's running locally. This file doesn't get published to Azure. To learn more, see [local.settings.file](#).
- **package.json**: Contains configuration options like a list of package dependencies, the main entrypoint, and scripts.

Registering a function

The programming model loads your functions based on the `main` field in your `package.json`. You can set the `main` field to a single file or multiple files by using a [glob pattern](#). The following table shows example values for the `main` field:

[] Expand table

Example	Description
<code>src/index.js</code>	Register functions from a single root file.
<code>src/functions/*.js</code>	Register each function from its own file.
<code>src/{index.js,functions/*.js}</code>	A combination where you register each function from its own file, but you still have a root file for general app-level code.

In order to register a function, you must import the `app` object from the `@azure/functions` npm module and call the method specific to your trigger type. The first argument when registering a function is the function name. The second argument is an `options` object specifying configuration for your trigger, your handler, and any other inputs or outputs. In some cases where trigger configuration isn't necessary, you can pass the handler directly as the second argument instead of an `options` object.

Registering a function can be done from any file in your project, as long as that file is loaded (directly or indirectly) based on the `main` field in your `package.json` file. The function should be registered at a global scope because you can't register functions once executions have started.

The following example is a simple function that logs that it was triggered and responds with `Hello, world!`:

JavaScript

```
const { app } = require('@azure/functions');

app.http('helloWorld1', {
    methods: ['POST', 'GET'],
    handler: async (request, context) => {
        context.log('Http function was triggered.');
        return { body: 'Hello, world!' };
    }
});
```

Inputs and outputs

Your function is required to have exactly one primary input called the trigger. It may also have secondary inputs, a primary output called the return output, and/or secondary outputs. Inputs and outputs are also referred to as [bindings](#) outside the context of the Node.js programming model. Before v4 of the model, these bindings were configured in `function.json` files.

Trigger input

The trigger is the only required input or output. For most trigger types, you register a function by using a method on the `app` object named after the trigger type. You can specify configuration specific to the trigger directly on the `options` argument. For example, an HTTP trigger allows you to specify a route. During execution, the value corresponding to this trigger is passed in as the first argument to your handler.

JavaScript

```
const { app } = require('@azure/functions');

app.http('helloWorld1', {
    route: 'hello/world',
    handler: async (request, context) => {
        ...
    }
});
```

Return output

The return output is optional, and in some cases configured by default. For example, an HTTP trigger registered with `app.http` is configured to return an HTTP response output automatically. For most output types, you specify the return configuration on the `options` argument with the help of the `output` object exported from the `@azure/functions` module. During execution, you set this output by returning it from your handler.

The following example uses a [timer trigger](#) and a [storage queue output](#):

JavaScript

JavaScript

```
const { app, output } = require('@azure/functions');

app.timer('timerTrigger1', {
    schedule: '0 */5 * * * *',
    return: output.storageQueue({
        connection: 'storage_APPSETTING',
        ...
    }),
    handler: (myTimer, context) => {
        return { hello: 'world' }
    }
});
```

Extra inputs and outputs

In addition to the trigger and return, you may specify extra inputs or outputs on the `options` argument when registering a function. The `input` and `output` objects exported from the `@azure/functions` module provide type-specific methods to help construct the configuration. During execution, you get or set the values with `context.extraInputs.get` or `context.extraOutputs.set`, passing in the original configuration object as the first argument.

The following example is a function triggered by a `storage queue`, with an extra `storage blob input` that is copied to an extra `storage blob output`. The queue message should be the name of a file and replaces `{queueTrigger}` as the blob name to be copied, with the help of a [binding expression](#).

JavaScript

```
const { app, input, output } = require('@azure/functions');

const blobInput = input.storageBlob({
    connection: 'storage_APPSETTING',
    path: 'helloworld/{queueTrigger}',
});

const blobOutput = output.storageBlob({
    connection: 'storage_APPSETTING',
    path: 'helloworld/{queueTrigger}-copy',
});
```

```
app.storageQueue('copyBlob1', {
    queueName: 'copyblobqueue',
    connection: 'storage_APPSETTING',
    extraInputs: [blobInput],
    extraOutputs: [blobOutput],
    handler: (queueItem, context) => {
        const blobInputValue = context.extraInputs.get(blobInput);
        context.extraOutputs.set(blobOutput, blobInputValue);
    }
});
```

Generic inputs and outputs

The `app`, `trigger`, `input`, and `output` objects exported by the `@azure/functions` module provide type-specific methods for most types. For all the types that aren't supported, a `generic` method is provided to allow you to manually specify the configuration. The `generic` method can also be used if you want to change the default settings provided by a type-specific method.

The following example is a simple HTTP triggered function using generic methods instead of type-specific methods.

JavaScript

```
JavaScript

const { app, output, trigger } = require('@azure/functions');

app.generic('helloWorld1', {
    trigger: trigger.generic({
        type: 'httpTrigger',
        methods: ['GET', 'POST']
    }),
    return: output.generic({
        type: 'http'
    }),
    handler: async (request, context) => {
        context.log(`Http function processed request for url
        ${request.url}`);
        return { body: `Hello, world!` };
    }
});
```

Invocation context

Each invocation of your function is passed an invocation `context` object, with information about your invocation and methods used for logging. In the v4 model, the `context` object is typically the second argument passed to your handler.

The `InvocationContext` class has the following properties:

[+] Expand table

Property	Description
<code>invocationId</code>	The ID of the current function invocation.
<code>functionName</code>	The name of the function.
<code>extraInputs</code>	Used to get the values of extra inputs. For more information, see extra inputs and outputs .
<code>extraOutputs</code>	Used to set the values of extra outputs. For more information, see extra inputs and outputs .
<code>retryContext</code>	See retry context .
<code>traceContext</code>	The context for distributed tracing. For more information, see Trace Context ↗ .
<code>triggerMetadata</code>	Metadata about the trigger input for this invocation, not including the value itself. For example, an event hub trigger has an <code>enqueuedTimeUtc</code> property.
<code>options</code>	The options used when registering the function, after they've been validated and with defaults explicitly specified.

Retry context

The `retryContext` object has the following properties:

[+] Expand table

Property	Description
<code>retryCount</code>	A number representing the current retry attempt.
<code>maxRetryCount</code>	Maximum number of times an execution is retried. A value of <code>-1</code> means to retry indefinitely.
<code>exception</code>	Exception that caused the retry.

For more information, see [retry-policies](#).

Logging

In Azure Functions, it's recommended to use `context.log()` to write logs. Azure Functions integrates with Azure Application Insights to better capture your function app logs. Application Insights, part of Azure Monitor, provides facilities for collection, visual rendering, and analysis of both application logs and your trace outputs. To learn more, see [monitoring Azure Functions](#).

ⓘ Note

If you use the alternative Node.js `console.log` method, those logs are tracked at the app-level and will *not* be associated with any specific function. It is *highly recommended* to use `context` for logging instead of `console` so that all logs are associated with a specific function.

The following example writes a log at the default "information" level, including the invocation ID:

```
JavaScript
JavaScript
context.log(`Something has happened. Invocation ID:
"${context.invocationId}"`);
```

Log levels

In addition to the default `context.log` method, the following methods are available that let you write logs at specific levels:

[+] [Expand table](#)

Method	Description
<code>context.trace()</code>	Writes a trace-level event to the logs.
<code>context.debug()</code>	Writes a debug-level event to the logs.

Method	Description
<code>context.info()</code>	Writes an information-level event to the logs.
<code>context.warn()</code>	Writes a warning-level event to the logs.
<code>context.error()</code>	Writes an error-level event to the logs.

Configure log level

Azure Functions lets you define the threshold level to be used when tracking and viewing logs. To set the threshold, use the `logging.logLevel` property in the `host.json` file. This property lets you define a default level applied to all functions, or a threshold for each individual function. To learn more, see [How to configure monitoring for Azure Functions](#).

HTTP triggers

HTTP and webhook triggers use `HttpRequest` and `HttpResponse` objects to represent HTTP messages. The classes represent a subset of the [fetch standard](#), using Node.js's [undici](#) package.

HTTP Request

The request can be accessed as the first argument to your handler for an HTTP triggered function.

JavaScript

JavaScript

```
async (request, context) => {
    context.log(`Http function processed request for url
    ${request.url}`);
}
```

The `HttpRequest` object has the following properties:

[+] [Expand table](#)

Property	Type	Description
<code>method</code>	<code>string</code>	HTTP request method used to invoke this function.
<code>url</code>	<code>string</code>	Request URL.
<code>headers</code>	<code>Headers</code> ↗	HTTP request headers.
<code>query</code>	<code>URLSearchParams</code> ↗	Query string parameter keys and values from the URL.
<code>params</code>	<code>Record<string, string></code>	Route parameter keys and values.
<code>user</code>	<code>HttpRequestUser</code> <code>null</code>	Object representing logged-in user, either through Functions authentication, SWA Authentication, or null when no such user is logged in.
<code>body</code>	<code>ReadableStream</code> <code>null</code> ↗	Body as a readable stream.
<code>bodyUsed</code>	<code>boolean</code>	A boolean indicating if the body is already read.

In order to access a request or response's body, the following methods can be used:

[+] [Expand table](#)

Method	Return Type
<code>arrayBuffer()</code>	<code>Promise<ArrayBuffer></code> ↗
<code>blob()</code>	<code>Promise<Blob></code> ↗
<code>formData()</code>	<code>Promise<FormData></code> ↗
<code>json()</code>	<code>Promise<unknown></code>
<code>text()</code>	<code>Promise<string></code>

ⓘ Note

The body functions can be run only once; subsequent calls will resolve with empty strings/ArrayBuffers.

HTTP Response

The response can be set in several ways:

- As a simple interface with type `HttpServletResponseInit`: This option is the most concise way of returning responses.

JavaScript

```
return { body: `Hello, world!` };
```

The `HttpServletResponseInit` interface has the following properties:

[+] Expand table

Property	Type	Description
<code>body</code>	<code>BodyInit</code> (optional)	HTTP response body as one of <code>ArrayBuffer</code> , <code>AsyncIterable<Uint8Array></code> , <code>Blob</code> , <code>FormData</code> , <code>Iterable<Uint8Array></code> , <code>NodeJS.ArrayBufferView</code> , <code>URLSearchParams</code> , <code>null</code> , or <code>string</code> .
<code>jsonBody</code>	<code>any</code> (optional)	A JSON-serializable HTTP Response body. If set, the <code>HttpServletResponseInit.body</code> property is ignored in favor of this property.
<code>status</code>	<code>number</code> (optional)	HTTP response status code. If not set, defaults to <code>200</code> .
<code>headers</code>	<code>HeadersInit</code> (optional)	HTTP response headers.
<code>cookies</code>	<code>Cookie[]</code> (optional)	HTTP response cookies.

- As a class with type `HttpServletResponse`: This option provides helper methods for reading and modifying various parts of the response like the headers.

JavaScript

```
const response = new HttpServletResponse({ body: `Hello, world!` });
response.headers.set('content-type', 'application/json');
return response;
```

The `HttpResponse` class accepts an optional `HttpResponseInit` as an argument to its constructor and has the following properties:

[+] Expand table

Property	Type	Description
<code>status</code>	<code>number</code>	HTTP response status code.
<code>headers</code>	<code>Headers</code> ↗	HTTP response headers.
<code>cookies</code>	<code>Cookie[]</code>	HTTP response cookies.
<code>body</code>	<code>ReadableStream null</code> ↗	Body as a readable stream.
<code>bodyUsed</code>	<code>boolean</code>	A boolean indicating if the body has been read from already.

HTTP streams

HTTP streams is a feature that makes it easier to process large data, stream OpenAI responses, deliver dynamic content, and support other core HTTP scenarios. It lets you stream requests to and responses from HTTP endpoints in your Node.js function app. Use HTTP streams in scenarios where your app requires real-time exchange and interaction between client and server over HTTP. You can also use HTTP streams to get the best performance and reliability for your apps when using HTTP.

The existing `HttpRequest` and `HttpResponse` types in programming model v4 already support various ways of handling the message body, including as a stream.

Prerequisites

- The [@azure/functions npm package](#) ↗ version 4.3.0 or later.
- [Azure Functions runtime](#) version 4.28 or later.
- [Azure Functions Core Tools](#) version 4.0.5530 or a later version, which contains the correct runtime version.

Enable streams

Use these steps to enable HTTP streams in your function app in Azure and in your local projects:

1. If you plan to stream large amounts of data, modify the `FUNCTIONS_REQUEST_BODY_SIZE_LIMIT` setting in Azure. The default maximum body size allowed is `104857600`, which limits your requests to a size of ~100 MB.
2. For local development, also add `FUNCTIONS_REQUEST_BODY_SIZE_LIMIT` to the [local.settings.json file](#).
3. Add the following code to your app in any file included by your [main field](#).

```
JavaScript
```

```
JavaScript
```

```
const { app } = require('@azure/functions');

app.setup({ enableHttpStream: true });
```

Stream examples

This example shows an HTTP triggered function that receives data via an HTTP POST request, and the function streams this data to a specified output file:

```
JavaScript
```

```
JavaScript
```

```
const { app } = require('@azure/functions');
const { createWriteStream } = require('fs');
const { Writable } = require('stream');

app.http('httpTriggerStreamRequest', {
    methods: ['POST'],
    authLevel: 'anonymous',
    handler: async (request, context) => {
        const writeStream = createWriteStream('<output file path>');
        await request.body.pipeTo(Writable.toWeb(writeStream));

        return { body: 'Done!' };
    },
});
```

This example shows an HTTP triggered function that streams a file's content as the response to incoming HTTP GET requests:

JavaScript

JavaScript

```
const { app } = require('@azure/functions');
const { createReadStream } = require('fs');

app.http('httpTriggerStreamResponse', {
    methods: ['GET'],
    authLevel: 'anonymous',
    handler: async (request, context) => {
        const body = createReadStream('<input file path>');

        return { body };
    },
});
```

For a ready-to-run sample app using streams, check out this example on [GitHub](#).

Stream considerations

- Use `request.body` to obtain the maximum benefit from using streams. You can still continue to use methods like `request.text()`, which always return the body as a string.

Hooks

Use a hook to execute code at different points in the Azure Functions lifecycle. Hooks are executed in the order they're registered and can be registered from any file in your app. There are currently two scopes of hooks, "app" level and "invocation" level.

Invocation hooks

Invocation hooks are executed once per invocation of your function, either before in a `preInvocation` hook or after in a `postInvocation` hook. By default your hook executes for all trigger types, but you can also filter by type. The following example shows how to register an invocation hook and filter by trigger type:

JavaScript

JavaScript

```

const { app } = require('@azure/functions');

app.hook.preInvocation((context) => {
    if (context.invocationContext.options.trigger.type ===
'httpTrigger') {
        context.invocationContext.log(
            `preInvocation hook executed for http function
${context.invocationContext.functionName}`
        );
    }
});

app.hook.postInvocation((context) => {
    if (context.invocationContext.options.trigger.type ===
'httpTrigger') {
        context.invocationContext.log(
            `postInvocation hook executed for http function
${context.invocationContext.functionName}`
        );
    }
});

```

The first argument to the hook handler is a context object specific to that hook type.

The `PreInvocationContext` object has the following properties:

[+] Expand table

Property	Description
<code>inputs</code>	The arguments passed to the invocation.
<code>functionHandler</code>	The function handler for the invocation. Changes to this value affect the function itself.
<code>invocationContext</code>	The <code>invocation context</code> object passed to the function.
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

The `PostInvocationContext` object has the following properties:

[+] Expand table

Property	Description
<code>inputs</code>	The arguments passed to the invocation.

Property	Description
<code>result</code>	The result of the function. Changes to this value affect the overall result of the function.
<code>error</code>	The error thrown by the function, or null/undefined if there's no error. Changes to this value affect the overall result of the function.
<code>invocationContext</code>	The invocation context object passed to the function.
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

App hooks

App hooks are executed once per instance of your app, either during startup in an `appStart` hook or during termination in an `appTerminate` hook. App terminate hooks have a limited time to execute and don't execute in all scenarios.

The Azure Functions runtime currently [doesn't support](#) context logging outside of an invocation. Use the Application Insights [npm package](#) to log data during app level hooks.

The following example registers app hooks:

JavaScript

```
const { app } = require('@azure/functions');

app.hook.appStart((context) => {
    // add your logic here
});

app.hook.appTerminate((context) => {
    // add your logic here
});
```

The first argument to the hook handler is a context object specific to that hook type.

The `AppStartContext` object has the following properties:

[+] [Expand table](#)

Property	Description
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

The `AppTerminateContext` object has the following properties:

[+] Expand table

Property	Description
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

Scaling and concurrency

By default, Azure Functions automatically monitors the load on your application and creates more host instances for Node.js as needed. Azure Functions uses built-in (not user configurable) thresholds for different trigger types to decide when to add instances, such as the age of messages and queue size for QueueTrigger. For more information, see [How the Consumption and Premium plans work](#).

This scaling behavior is sufficient for many Node.js applications. For CPU-bound applications, you can improve performance further by using multiple language worker processes. You can increase the number of worker processes per host from the default of 1 up to a max of 10 by using the `FUNCTIONS_WORKER_PROCESS_COUNT` application setting. Azure Functions then tries to evenly distribute simultaneous function invocations across these workers. This behavior makes it less likely that a CPU-intensive function blocks other functions from running. The setting applies to each host that Azure Functions creates when scaling out your application to meet demand.

⚠ Warning

Use the `FUNCTIONS_WORKER_PROCESS_COUNT` setting with caution. Multiple processes running in the same instance can lead to unpredictable behavior and increase function load times. If you use this setting, it's *highly recommended* to offset these downsides by [running from a package file](#).

Node version

You can see the current version that the runtime is using by logging `process.version` from any function. See [supported versions](#) for a list of Node.js versions supported by each programming model.

Setting the Node version

The way that you upgrade your Node.js version depends on the OS on which your function app runs.

Windows

When running on Windows, the Node.js version is set by the [WEBSITE_NODE_DEFAULT_VERSION](#) application setting. This setting can be updated either by using the Azure CLI or in the Azure portal.

For more information about Node.js versions, see [Supported versions](#).

Before upgrading your Node.js version, make sure your function app is running on the latest version of the Azure Functions runtime. If you need to upgrade your runtime version, see [Migrate apps from Azure Functions version 3.x to version 4.x](#).

Azure CLI

Run the Azure CLI `az functionapp config appsettings set` command to update the Node.js version for your function app running on Windows:

```
Azure CLI  
az functionapp config appsettings set --settings  
WEBSITE_NODE_DEFAULT_VERSION=~20 \  
--name <FUNCTION_APP_NAME> --resource-group <RESOURCE_GROUP_NAME>
```

This sets the [WEBSITE_NODE_DEFAULT_VERSION](#) application setting the supported LTS version of `~20`.

After changes are made, your function app restarts. To learn more about Functions support for Node.js, see [Language runtime support policy](#).

Environment variables

Environment variables can be useful for operational secrets (connection strings, keys, endpoints, etc.) or environmental settings such as profiling variables. You can add environment variables in both your local and cloud environments and access them through `process.env` in your function code.

The following example logs the `WEBSITE_SITE_NAME` environment variable:

```
JavaScript
```

```
JavaScript
```

```
async function timerTrigger1(myTimer, context) {
    context.log(`WEBSITE_SITE_NAME:
${process.env["WEBSITE_SITE_NAME"]}`);
}
```

In local development environment

When you run locally, your functions project includes a [local.settings.json file](#), where you store your environment variables in the `Values` object.

```
JSON
```

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "CUSTOM_ENV_VAR_1": "hello",
    "CUSTOM_ENV_VAR_2": "world"
  }
}
```

In Azure cloud environment

When you run in Azure, the function app lets you set and use [Application settings](#), such as service connection strings, and exposes these settings as environment variables during execution.

There are several ways that you can add, update, and delete function app settings:

- In the Azure portal.
- By using the Azure CLI.

- By using Azure PowerShell.

Changes to function app settings require your function app to be restarted.

Worker environment variables

There are several Functions environment variables specific to Node.js:

languageWorkers_node_arguments

This setting allows you to specify custom arguments when starting your Node.js process. It's most often used locally to start the worker in debug mode, but can also be used in Azure if you need custom arguments.

⚠ Warning

If possible, avoid using `languageWorkers_node_arguments` in Azure because it can have a negative effect on cold start times. Rather than using pre-warmed workers, the runtime has to start a new worker from scratch with your custom arguments.

logging(LogLevel)_Worker

This setting adjusts the default log level for Node.js-specific worker logs. By default, only warning or error logs are shown, but you can set it to `information` or `debug` to help diagnose issues with the Node.js worker. For more information, see [configuring log levels](#).

ECMAScript modules (preview)

ⓘ Note

As ECMAScript modules are currently a preview feature in Node.js 14 or higher in Azure Functions.

[ECMAScript modules](#) (ES modules) are the new official standard module system for Node.js. So far, the code samples in this article use the CommonJS syntax. When running Azure Functions in Node.js 14 or higher, you can choose to write your functions using ES modules syntax.

To use ES modules in a function, change its filename to use a `.mjs` extension. The following `index.mjs` file example is an HTTP triggered function that uses ES modules syntax to import the `uuid` library and return a value.

```
JavaScript
```

```
JavaScript

import { v4 as uuidv4 } from 'uuid';

async function httpTrigger1(request, context) {
    return { body: uuidv4() };
}

app.http('httpTrigger1', {
    methods: ['GET', 'POST'],
    handler: httpTrigger1
});
```

Local debugging

It's recommended to use VS Code for local debugging, which starts your Node.js process in debug mode automatically and attaches to the process for you. For more information, see [run the function locally](#).

If you're using a different tool for debugging or want to start your Node.js process in debug mode manually, add `"languageWorkers_node_arguments": "--inspect"` under `Values` in your `local.settings.json`. The `--inspect` argument tells Node.js to listen for a debug client, on port 9229 by default. For more information, see the [Node.js debugging guide](#).

Recommendations

This section describes several impactful patterns for Node.js apps that we recommend you follow.

Choose single-vCPU App Service plans

When you create a function app that uses the App Service plan, we recommend that you select a single-vCPU plan rather than a plan with multiple vCPUs. Today, Functions runs Node.js functions more efficiently on single-vCPU VMs, and using larger VMs doesn't

produce the expected performance improvements. When necessary, you can manually scale out by adding more single-vCPU VM instances, or you can enable autoscale. For more information, see [Scale instance count manually or automatically](#).

Run from a package file

When you develop Azure Functions in the serverless hosting model, cold starts are a reality. *Cold start* refers to the first time your function app starts after a period of inactivity, taking longer to start up. For Node.js apps with large dependency trees in particular, cold start can be significant. To speed up the cold start process, [run your functions as a package file](#) when possible. Many deployment methods use this model by default, but if you're experiencing large cold starts you should check to make sure you're running this way.

Use a single static client

When you use a service-specific client in an Azure Functions application, don't create a new client with every function invocation because you can hit connection limits. Instead, create a single, static client in the global scope. For more information, see [managing connections in Azure Functions](#).

Use `async` and `await`

When writing Azure Functions in Node.js, you should write code using the `async` and `await` keywords. Writing code using `async` and `await` instead of callbacks or `.then` and `.catch` with Promises helps avoid two common problems:

- Throwing uncaught exceptions that [crash the Node.js process](#), potentially affecting the execution of other functions.
- Unexpected behavior, such as missing logs from `context.log`, caused by asynchronous calls that aren't properly awaited.

In the following example, the asynchronous method `fs.readFile` is invoked with an error-first callback function as its second parameter. This code causes both of the issues previously mentioned. An exception that isn't explicitly caught in the correct scope can crash the entire process (issue #1). Returning without ensuring the callback finishes means the http response will sometimes have an empty body (issue #2).

JavaScript

JavaScript

```
// DO NOT USE THIS CODE
const { app } = require('@azure/functions');
const fs = require('fs');

app.http('httpTriggerBadAsync', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    handler: async (request, context) => {
        let fileData;
        fs.readFile('./helloWorld.txt', (err, data) => {
            if (err) {
                context.error(err);
                // BUG #1: This will result in an uncaught exception
                that crashes the entire process
                throw err;
            }
            fileData = data;
        });
        // BUG #2: fileData is not guaranteed to be set before the
        invocation ends
        return { body: fileData };
    },
});
```

Use the `async` and `await` keywords to help avoid both of these issues. Most APIs in the Node.js ecosystem have been converted to support promises in some form. For example, starting in v14, Node.js provides an `fs/promises` API to replace the `fs` callback API.

In the following example, any unhandled exceptions thrown during the function execution only fail the individual invocation that raised the exception. The `await` keyword means that steps following `readFile` only execute after it's complete.

JavaScript

JavaScript

```
// Recommended pattern
const { app } = require('@azure/functions');
const fs = require('fs/promises');

app.http('httpTriggerGoodAsync', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    handler: async (request, context) => {
        try {
            const fileData = await fs.readFile('./helloWorld.txt');
            return { body: fileData };
        }
    }
});
```

```
        } catch (err) {
            context.error(err);
            // This rethrown exception will only fail the individual
            invocation, instead of crashing the whole process
            throw err;
        }
    },
});
```

Troubleshoot

See the [Node.js Troubleshoot guide](#).

Next steps

For more information, see the following resources:

- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Functions Node.js developer guide

Article • 02/28/2024

This guide is an introduction to developing Azure Functions using JavaScript or TypeScript. The article assumes that you have already read the [Azure Functions developer guide](#).

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of this page. The version you choose should match the version of the [@azure/functions](#) npm package you are using in your app. If you do not have that package listed in your `package.json`, the default is v3. Learn more about the differences between v3 and v4 in the [migration guide](#).

As a Node.js developer, you might also be interested in one of the following articles:

 Expand table

Getting started	Concepts	Guided learning
<ul style="list-style-type: none">Node.js function using Visual Studio CodeNode.js function with terminal/command promptNode.js function using the Azure portal	<ul style="list-style-type: none">Developer guideHosting optionsPerformance considerations	<ul style="list-style-type: none">Create serverless applicationsRefactor Node.js and Express APIs to Serverless APIs

Considerations

- The Node.js programming model shouldn't be confused with the Azure Functions runtime:
 - **Programming model:** Defines how you author your code and is specific to JavaScript and TypeScript.
 - **Runtime:** Defines underlying behavior of Azure Functions and is shared across all languages.
- The version of the programming model is strictly tied to the version of the [@azure/functions](#) npm package. It's versioned independently of the **runtime**.

Both the runtime and the programming model use the number 4 as their latest major version, but that's a coincidence.

- You can't mix the v3 and v4 programming models in the same function app. As soon as you register one v4 function in your app, any v3 functions registered in `function.json` files are ignored.

Supported versions

The following table shows each version of the Node.js programming model along with its supported versions of the Azure Functions runtime and Node.js.

[\[+\] Expand table](#)

Programming Model Version ↗	Support Level	Functions Runtime Version	Node.js Version ↗	Description
4.x	GA	4.25+	20.x, 18.x	Supports a flexible file structure and code-centric approach to triggers and bindings.
3.x	GA	4.x	20.x, 18.x, 16.x, 14.x	Requires a specific file structure with your triggers and bindings declared in a "function.json" file
2.x	n/a	3.x	14.x, 12.x, 10.x	Reached end of support on December 13, 2022. See Functions Versions for more info.
1.x	n/a	2.x	10.x, 8.x	Reached end of support on December 13, 2022. See Functions Versions for more info.

Folder structure

TypeScript

The recommended folder structure for a TypeScript project looks like the following example:

```
<project_root>
| - .vscode/
| - dist/
| - node_modules/
| - src/
| | - functions/
| | | - myFirstFunction.ts
| | | - mySecondFunction.ts
| - test/
| | - functions/
| | | - myFirstFunction.test.ts
| | | - mySecondFunction.test.ts
| - .funcignore
| - host.json
| - local.settings.json
| - package.json
| - tsconfig.json
```

The main project folder, `<project_root>`, can contain the following files:

- **.vscode/**: (Optional) Contains the stored Visual Studio Code configuration. To learn more, see [Visual Studio Code settings](#).
- **dist/**: Contains the compiled JavaScript code after you run a build. The name of this folder can be configured in your "tsconfig.json" file.
- **src/functions/**: The default location for all functions and their related triggers and bindings.
- **test/**: (Optional) Contains the test cases of your function app.
- **.funcignore**: (Optional) Declares files that shouldn't get published to Azure. Usually, this file contains `.vscode/` to ignore your editor setting, `test/` to ignore test cases, and `local.settings.json` to prevent local app settings being published.
- **host.json**: Contains configuration options that affect all functions in a function app instance. This file does get published to Azure. Not all options are supported when running locally. To learn more, see [host.json](#).
- **local.settings.json**: Used to store app settings and connection strings when it's running locally. This file doesn't get published to Azure. To learn more, see [local.settings.file](#).
- **package.json**: Contains configuration options like a list of package dependencies, the main entrypoint, and scripts.
- **tsconfig.json**: Contains TypeScript compiler options like the output directory.

Registering a function

The programming model loads your functions based on the `main` field in your `package.json`. You can set the `main` field to a single file or multiple files by using a [glob pattern](#). The following table shows example values for the `main` field:

TypeScript	 Expand table
Example	Description
<code>dist/src/index.js</code>	Register functions from a single root file.
<code>dist/src/functions/*.js</code>	Register each function from its own file.
<code>dist/src/{index.js,functions/*.js}</code>	A combination where you register each function from its own file, but you still have a root file for general app-level code.

In order to register a function, you must import the `app` object from the `@azure/functions` npm module and call the method specific to your trigger type. The first argument when registering a function is the function name. The second argument is an `options` object specifying configuration for your trigger, your handler, and any other inputs or outputs. In some cases where trigger configuration isn't necessary, you can pass the handler directly as the second argument instead of an `options` object.

Registering a function can be done from any file in your project, as long as that file is loaded (directly or indirectly) based on the `main` field in your `package.json` file. The function should be registered at a global scope because you can't register functions once executions have started.

The following example is a simple function that logs that it was triggered and responds with `Hello, world!`:

TypeScript
<pre>import { app, HttpRequest, HttpResponseInit, InvocationContext } from "@azure/functions"; async function helloWorld1(request: HttpRequest, context: InvocationContext): Promise<HttpResponseInit> { context.log('Http function was triggered.');</pre>

```
        return { body: 'Hello, world!' };

};

app.http('helloWorld1', {
    methods: ['GET', 'POST'],
    handler: helloWorld1
});
```

Inputs and outputs

Your function is required to have exactly one primary input called the trigger. It may also have secondary inputs, a primary output called the return output, and/or secondary outputs. Inputs and outputs are also referred to as [bindings](#) outside the context of the Node.js programming model. Before v4 of the model, these bindings were configured in `function.json` files.

Trigger input

The trigger is the only required input or output. For most trigger types, you register a function by using a method on the `app` object named after the trigger type. You can specify configuration specific to the trigger directly on the `options` argument. For example, an HTTP trigger allows you to specify a route. During execution, the value corresponding to this trigger is passed in as the first argument to your handler.

TypeScript

```
TypeScript

import { app, HttpRequest, HttpResponseInit, InvocationContext } from
"@azure/functions";

async function helloWorld1(request: HttpRequest, context:
InvocationContext): Promise<HttpResponseInit> {
    ...
};

app.http('helloWorld1', {
    route: 'hello/world',
    handler: helloWorld1
});
```

Return output

The return output is optional, and in some cases configured by default. For example, an HTTP trigger registered with `app.http` is configured to return an HTTP response output automatically. For most output types, you specify the return configuration on the `options` argument with the help of the `output` object exported from the `@azure/functions` module. During execution, you set this output by returning it from your handler.

The following example uses a [timer trigger](#) and a [storage queue output](#):

TypeScript

```
TypeScript

import { app, InvocationContext, Timer, output } from
"@azure/functions";

async function timerTrigger1(myTimer: Timer, context:
InvocationContext): Promise<any> {
    return { hello: 'world' }
}

app.timer('timerTrigger1', {
    schedule: '0 */5 * * *',
    return: output.storageQueue({
        connection: 'storage_APPSETTING',
        ...
    }),
    handler: timerTrigger1
});
```

Extra inputs and outputs

In addition to the trigger and return, you may specify extra inputs or outputs on the `options` argument when registering a function. The `input` and `output` objects exported from the `@azure/functions` module provide type-specific methods to help construct the configuration. During execution, you get or set the values with `context.extraInputs.get` or `context.extraOutputs.set`, passing in the original configuration object as the first argument.

The following example is a function triggered by a [storage queue](#), with an extra [storage blob input](#) that is copied to an extra [storage blob output](#). The queue message should be the name of a file and replaces `{queueTrigger}` as the blob name to be copied, with the help of a [binding expression](#).

TypeScript

TypeScript

```
import { app, InvocationContext, input, output } from
"@azure/functions";

const blobInput = input.storageBlob({
    connection: 'storage_APPSETTING',
    path: 'helloworld/{queueTrigger}',
});

const blobOutput = output.storageBlob({
    connection: 'storage_APPSETTING',
    path: 'helloworld/{queueTrigger}-copy',
});

async function copyBlob1(queueItem: unknown, context:
InvocationContext): Promise<void> {
    const blobInputValue = context.extraInputs.get(blobInput);
    context.extraOutputs.set(blobOutput, blobInputValue);
}

app.storageQueue('copyBlob1', {
    queueName: 'copyblobqueue',
    connection: 'storage_APPSETTING',
    extraInputs: [blobInput],
    extraOutputs: [blobOutput],
    handler: copyBlob1
});
```

Generic inputs and outputs

The `app`, `trigger`, `input`, and `output` objects exported by the `@azure/functions` module provide type-specific methods for most types. For all the types that aren't supported, a `generic` method is provided to allow you to manually specify the configuration. The `generic` method can also be used if you want to change the default settings provided by a type-specific method.

The following example is a simple HTTP triggered function using generic methods instead of type-specific methods.

TypeScript

TypeScript

```

import { app, InvocationContext, HttpRequest, HttpResponseInit, output,
trigger } from "@azure/functions";

async function helloWorld1(request: HttpRequest, context:
InvocationContext): Promise<HttpResponseInit> {
    context.log(`Http function processed request for url
"${request.url}"`);

    return { body: `Hello, world!` };
}

app.generic('helloWorld1', {
    trigger: trigger.generic({
        type: 'httpTrigger',
        methods: ['GET', 'POST']
    }),
    return: output.generic({
        type: 'http'
    }),
    handler: helloWorld1
});

```

Invocation context

Each invocation of your function is passed an invocation `context` object, with information about your invocation and methods used for logging. In the v4 model, the `context` object is typically the second argument passed to your handler.

The `InvocationContext` class has the following properties:

[] [Expand table](#)

Property	Description
<code>invocationId</code>	The ID of the current function invocation.
<code>functionName</code>	The name of the function.
<code>extraInputs</code>	Used to get the values of extra inputs. For more information, see extra inputs and outputs .
<code>extraOutputs</code>	Used to set the values of extra outputs. For more information, see extra inputs and outputs .
<code>retryContext</code>	See retry context .
<code>traceContext</code>	The context for distributed tracing. For more information, see Trace Context .

Property	Description
<code>triggerMetadata</code>	Metadata about the trigger input for this invocation, not including the value itself. For example, an event hub trigger has an <code>enqueuedTimeUtc</code> property.
<code>options</code>	The options used when registering the function, after they've been validated and with defaults explicitly specified.

Retry context

The `retryContext` object has the following properties:

[+] [Expand table](#)

Property	Description
<code>retryCount</code>	A number representing the current retry attempt.
<code>maxRetryCount</code>	Maximum number of times an execution is retried. A value of <code>-1</code> means to retry indefinitely.
<code>exception</code>	Exception that caused the retry.

For more information, see [retry-policies](#).

Logging

In Azure Functions, it's recommended to use `context.log()` to write logs. Azure Functions integrates with Azure Application Insights to better capture your function app logs. Application Insights, part of Azure Monitor, provides facilities for collection, visual rendering, and analysis of both application logs and your trace outputs. To learn more, see [monitoring Azure Functions](#).

ⓘ Note

If you use the alternative Node.js `console.log` method, those logs are tracked at the app-level and will *not* be associated with any specific function. It is *highly recommended* to use `context` for logging instead of `console` so that all logs are associated with a specific function.

The following example writes a log at the default "information" level, including the invocation ID:

TypeScript

TypeScript

```
context.log(`Something has happened. Invocation ID:  
"${context.invocationId}"`);
```

Log levels

In addition to the default `context.log` method, the following methods are available that let you write logs at specific levels:

[+] Expand table

Method	Description
<code>context.trace()</code>	Writes a trace-level event to the logs.
<code>context.debug()</code>	Writes a debug-level event to the logs.
<code>context.info()</code>	Writes an information-level event to the logs.
<code>context.warn()</code>	Writes a warning-level event to the logs.
<code>context.error()</code>	Writes an error-level event to the logs.

Configure log level

Azure Functions lets you define the threshold level to be used when tracking and viewing logs. To set the threshold, use the `logging.level` property in the `host.json` file. This property lets you define a default level applied to all functions, or a threshold for each individual function. To learn more, see [How to configure monitoring for Azure Functions](#).

HTTP triggers

HTTP and webhook triggers use `HttpRequest` and `HttpResponse` objects to represent HTTP messages. The classes represent a subset of the [fetch standard](#), using Node.js's [undici](#) package.

HTTP Request

The request can be accessed as the first argument to your handler for an HTTP triggered function.

TypeScript

```
async function helloWorld1(request: HttpRequest, context: InvocationContext): Promise<HttpResponseInit> {
    context.log(`Http function processed request for url
"${request.url}"`);
```

The `HttpRequest` object has the following properties:

[+] Expand table

Property	Type	Description
<code>method</code>	<code>string</code>	HTTP request method used to invoke this function.
<code>url</code>	<code>string</code>	Request URL.
<code>headers</code>	<code>Headers</code> ↗	HTTP request headers.
<code>query</code>	<code>URLSearchParams</code> ↗	Query string parameter keys and values from the URL.
<code>params</code>	<code>Record<string, string></code>	Route parameter keys and values.
<code>user</code>	<code>HttpRequestUser null</code>	Object representing logged-in user, either through Functions authentication, SWA Authentication, or null when no such user is logged in.
<code>body</code>	<code>ReadableStream null</code> ↗	Body as a readable stream.
<code>bodyUsed</code>	<code>boolean</code>	A boolean indicating if the body is already read.

In order to access a request or response's body, the following methods can be used:

[+] Expand table

Method	Return Type
<code>arrayBuffer()</code>	<code>Promise<ArrayBuffer></code> ↗
<code>blob()</code>	<code>Promise<Blob></code> ↗

Method	Return Type
<code>formData()</code>	<code>Promise<FormData></code>
<code>json()</code>	<code>Promise<unknown></code>
<code>text()</code>	<code>Promise<string></code>

ⓘ Note

The body functions can be run only once; subsequent calls will resolve with empty strings/ArrayBuffers.

HTTP Response

The response can be set in several ways:

- As a simple interface with type `HttpServletResponseInit`: This option is the most concise way of returning responses.

TypeScript

```
TypeScript

return { body: `Hello, world!` };
```

The `HttpServletResponseInit` interface has the following properties:

[] [Expand table](#)

Property	Type	Description
<code>body</code>	<code>BodyInit</code> (optional)	HTTP response body as one of <code>ArrayBuffer</code> , <code>AsyncIterable<Uint8Array></code> , <code>Blob</code> , <code>FormData</code> , <code>Iterable<Uint8Array></code> , <code>NodeJS.ArrayBufferView</code> , <code>URLSearchParams</code> , <code>null</code> , or <code>string</code> .
<code>jsonBody</code>	<code>any</code> (optional)	A JSON-serializable HTTP Response body. If set, the <code>HttpServletResponseInit.body</code> property is ignored in favor of this property.
<code>status</code>	<code>number</code> (optional)	HTTP response status code. If not set, defaults to <code>200</code> .

Property	Type	Description
<code>headers</code>	<code>HeadersInit</code> ↗	HTTP response headers. (optional)
<code>cookies</code>	<code>Cookie[]</code>	HTTP response cookies. (optional)

- As a class with type `HttpResponse`: This option provides helper methods for reading and modifying various parts of the response like the headers.

TypeScript

```
const response = new HttpResponse({ body: `Hello, world!` });
response.headers.set('content-type', 'application/json');
return response;
```

The `HttpResponse` class accepts an optional `HttpResponseInit` as an argument to its constructor and has the following properties:

[+] Expand table

Property	Type	Description
<code>status</code>	<code>number</code>	HTTP response status code.
<code>headers</code>	<code>Headers</code> ↗	HTTP response headers.
<code>cookies</code>	<code>Cookie[]</code>	HTTP response cookies.
<code>body</code>	<code>ReadableStream null</code> ↗	Body as a readable stream.
<code>bodyUsed</code>	<code>boolean</code>	A boolean indicating if the body has been read from already.

HTTP streams

HTTP streams is a feature that makes it easier to process large data, stream OpenAI responses, deliver dynamic content, and support other core HTTP scenarios. It lets you stream requests to and responses from HTTP endpoints in your Node.js function app. Use HTTP streams in scenarios where your app requires real-time exchange and

interaction between client and server over HTTP. You can also use HTTP streams to get the best performance and reliability for your apps when using HTTP.

The existing `HttpRequest` and `HttpResponse` types in programming model v4 already support various ways of handling the message body, including as a stream.

Prerequisites

- The [@azure/functions npm package](#) version 4.3.0 or later.
- [Azure Functions runtime](#) version 4.28 or later.
- [Azure Functions Core Tools](#) version 4.0.5530 or a later version, which contains the correct runtime version.

Enable streams

Use these steps to enable HTTP streams in your function app in Azure and in your local projects:

1. If you plan to stream large amounts of data, modify the `FUNCTIONS_REQUEST_BODY_SIZE_LIMIT` setting in Azure. The default maximum body size allowed is `104857600`, which limits your requests to a size of ~100 MB.
2. For local development, also add `FUNCTIONS_REQUEST_BODY_SIZE_LIMIT` to the [local.settings.json file](#).
3. Add the following code to your app in any file included by your [main field](#).

```
TypeScript

TypeScript
import { app } from '@azure/functions';
app.setup({ enableHttpStream: true });
```

Stream examples

This example shows an HTTP triggered function that receives data via an HTTP POST request, and the function streams this data to a specified output file:

```
TypeScript
```

TypeScript

```
import { app, HttpRequest, HttpResponseInit, InvocationContext } from
'@azure/functions';
import { createWriteStream } from 'fs';
import { Writable } from 'stream';

export async function httpTriggerStreamRequest(
    request: HttpRequest,
    context: InvocationContext
): Promise<HttpResponseInit> {
    const writeStream = createWriteStream('<output file path>');
    await request.body.pipeTo(Writable.toWeb(writeStream));

    return { body: 'Done!' };
}

app.http('httpTriggerStreamRequest', {
    methods: ['POST'],
    authLevel: 'anonymous',
    handler: httpTriggerStreamRequest,
});
```

This example shows an HTTP triggered function that streams a file's content as the response to incoming HTTP GET requests:

TypeScript

```
import { app, HttpRequest, HttpResponseInit, InvocationContext } from
'@azure/functions';
import { createReadStream } from 'fs';

export async function httpTriggerStreamResponse(
    request: HttpRequest,
    context: InvocationContext
): Promise<HttpResponseInit> {
    const body = createReadStream('<input file path>');

    return { body };
}

app.http('httpTriggerStreamResponse', {
    methods: ['GET'],
    authLevel: 'anonymous',
    handler: httpTriggerStreamResponse,
});
```

For a ready-to-run sample app using streams, check out this example on [GitHub](#).

Stream considerations

- Use `request.body` to obtain the maximum benefit from using streams. You can still continue to use methods like `request.text()`, which always return the body as a string.

Hooks

Use a hook to execute code at different points in the Azure Functions lifecycle. Hooks are executed in the order they're registered and can be registered from any file in your app. There are currently two scopes of hooks, "app" level and "invocation" level.

Invocation hooks

Invocation hooks are executed once per invocation of your function, either before in a `preInvocation` hook or after in a `postInvocation` hook. By default your hook executes for all trigger types, but you can also filter by type. The following example shows how to register an invocation hook and filter by trigger type:

TypeScript

```
TypeScript

import { app, PostInvocationContext, PreInvocationContext } from
'@azure/functions';

app.hook.preInvocation((context: PreInvocationContext) => {
    if (context.invocationContext.options.trigger.type ===
'httpTrigger') {
        context.invocationContext.log(
            `preInvocation hook executed for http function
${context.invocationContext.functionName}`
        );
    }
});

app.hook.postInvocation((context: PostInvocationContext) => {
    if (context.invocationContext.options.trigger.type ===
'httpTrigger') {
        context.invocationContext.log(
            `postInvocation hook executed for http function
${context.invocationContext.functionName}`
        );
}
```

```
    }  
});
```

The first argument to the hook handler is a context object specific to that hook type.

The `PreInvocationContext` object has the following properties:

[+] [Expand table](#)

Property	Description
<code>inputs</code>	The arguments passed to the invocation.
<code>functionHandler</code>	The function handler for the invocation. Changes to this value affect the function itself.
<code>invocationContext</code>	The invocation context object passed to the function.
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

The `PostInvocationContext` object has the following properties:

[+] [Expand table](#)

Property	Description
<code>inputs</code>	The arguments passed to the invocation.
<code>result</code>	The result of the function. Changes to this value affect the overall result of the function.
<code>error</code>	The error thrown by the function, or null/undefined if there's no error. Changes to this value affect the overall result of the function.
<code>invocationContext</code>	The invocation context object passed to the function.
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

App hooks

App hooks are executed once per instance of your app, either during startup in an `appStart` hook or during termination in an `appTerminate` hook. App terminate hooks

have a limited time to execute and don't execute in all scenarios.

The Azure Functions runtime currently [doesn't support](#) context logging outside of an invocation. Use the Application Insights [npm package](#) to log data during app level hooks.

The following example registers app hooks:

TypeScript

```
import { app, AppStartContext, AppTerminateContext } from '@azure/functions';

app.hook.appStart((context: AppStartContext) => {
    // add your logic here
});

app.hook.appTerminate((context: AppTerminateContext) => {
    // add your logic here
});
```

The first argument to the hook handler is a context object specific to that hook type.

The `AppStartContext` object has the following properties:

[+] [Expand table](#)

Property	Description
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

The `AppTerminateContext` object has the following properties:

[+] [Expand table](#)

Property	Description
<code>hookData</code>	The recommended place to store and share data between hooks in the same scope. You should use a unique property name so that it doesn't conflict with other hooks' data.

Scaling and concurrency

By default, Azure Functions automatically monitors the load on your application and creates more host instances for Node.js as needed. Azure Functions uses built-in (not user configurable) thresholds for different trigger types to decide when to add instances, such as the age of messages and queue size for QueueTrigger. For more information, see [How the Consumption and Premium plans work](#).

This scaling behavior is sufficient for many Node.js applications. For CPU-bound applications, you can improve performance further by using multiple language worker processes. You can increase the number of worker processes per host from the default of 1 up to a max of 10 by using the `FUNCTIONS_WORKER_PROCESS_COUNT` application setting. Azure Functions then tries to evenly distribute simultaneous function invocations across these workers. This behavior makes it less likely that a CPU-intensive function blocks other functions from running. The setting applies to each host that Azure Functions creates when scaling out your application to meet demand.

⚠ Warning

Use the `FUNCTIONS_WORKER_PROCESS_COUNT` setting with caution. Multiple processes running in the same instance can lead to unpredictable behavior and increase function load times. If you use this setting, it's *highly recommended* to offset these downsides by [running from a package file](#).

Node version

You can see the current version that the runtime is using by logging `process.version` from any function. See [supported versions](#) for a list of Node.js versions supported by each programming model.

Setting the Node version

The way that you upgrade your Node.js version depends on the OS on which your function app runs.

Windows

When running on Windows, the Node.js version is set by the `WEBSITE_NODE_DEFAULT_VERSION` application setting. This setting can be updated

either by using the Azure CLI or in the Azure portal.

For more information about Node.js versions, see [Supported versions](#).

Before upgrading your Node.js version, make sure your function app is running on the latest version of the Azure Functions runtime. If you need to upgrade your runtime version, see [Migrate apps from Azure Functions version 3.x to version 4.x](#).

Azure CLI

Run the Azure CLI `az functionapp config appsettings set` command to update the Node.js version for your function app running on Windows:

Azure CLI

```
az functionapp config appsettings set --settings  
WEBSITE_NODE_DEFAULT_VERSION=~20 \  
--name <FUNCTION_APP_NAME> --resource-group <RESOURCE_GROUP_NAME>
```

This sets the `WEBSITE_NODE_DEFAULT_VERSION` application setting the supported LTS version of `~20`.

After changes are made, your function app restarts. To learn more about Functions support for Node.js, see [Language runtime support policy](#).

Environment variables

Environment variables can be useful for operational secrets (connection strings, keys, endpoints, etc.) or environmental settings such as profiling variables. You can add environment variables in both your local and cloud environments and access them through `process.env` in your function code.

The following example logs the `WEBSITE_SITE_NAME` environment variable:

TypeScript

TypeScript

```
async function timerTrigger1(myTimer: Timer, context:  
InvocationContext): Promise<void> {  
    context.log(`WEBSITE_SITE_NAME:
```

```
    ${process.env[ "WEBSITE_SITE_NAME" ]} );
}
```

In local development environment

When you run locally, your functions project includes a [local.settings.json file](#), where you store your environment variables in the `Values` object.

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "CUSTOM_ENV_VAR_1": "hello",
    "CUSTOM_ENV_VAR_2": "world"
  }
}
```

In Azure cloud environment

When you run in Azure, the function app lets you set and use [Application settings](#), such as service connection strings, and exposes these settings as environment variables during execution.

There are several ways that you can add, update, and delete function app settings:

- [In the Azure portal](#).
- [By using the Azure CLI](#).
- [By using Azure PowerShell](#).

Changes to function app settings require your function app to be restarted.

Worker environment variables

There are several Functions environment variables specific to Node.js:

languageWorkers_node_arguments

This setting allows you to specify custom arguments when starting your Node.js process. It's most often used locally to start the worker in debug mode, but can also be used in Azure if you need custom arguments.

⚠️ Warning

If possible, avoid using `languageWorkers__node__arguments` in Azure because it can have a negative effect on cold start times. Rather than using pre-warmed workers, the runtime has to start a new worker from scratch with your custom arguments.

logging(LogLevel.Worker)

This setting adjusts the default log level for Node.js-specific worker logs. By default, only warning or error logs are shown, but you can set it to `information` or `debug` to help diagnose issues with the Node.js worker. For more information, see [configuring log levels](#).

ECMAScript modules (preview)

ⓘ Note

As ECMAScript modules are currently a preview feature in Node.js 14 or higher in Azure Functions.

[ECMAScript modules](#) (ES modules) are the new official standard module system for Node.js. So far, the code samples in this article use the CommonJS syntax. When running Azure Functions in Node.js 14 or higher, you can choose to write your functions using ES modules syntax.

To use ES modules in a function, change its filename to use a `.mjs` extension. The following `index.mjs` file example is an HTTP triggered function that uses ES modules syntax to import the `uuid` library and return a value.

TypeScript

TypeScript

```
import { app, HttpRequest, HttpResponseInit, InvocationContext } from
"@azure/functions";
import { v4 as uuidv4 } from 'uuid';

async function httpTrigger1(request: HttpRequest, context:
InvocationContext): Promise<HttpResponseInit> {
    return { body: uuidv4() };
};
```

```
app.http('httpTrigger1', {
    methods: ['GET', 'POST'],
    handler: httpTrigger1
});
```

Local debugging

It's recommended to use VS Code for local debugging, which starts your Node.js process in debug mode automatically and attaches to the process for you. For more information, see [run the function locally](#).

If you're using a different tool for debugging or want to start your Node.js process in debug mode manually, add `"languageWorkers__node__arguments": "--inspect"` under `Values` in your [local.settings.json](#). The `--inspect` argument tells Node.js to listen for a debug client, on port 9229 by default. For more information, see the [Node.js debugging guide](#).

Recommendations

This section describes several impactful patterns for Node.js apps that we recommend you follow.

Choose single-vCPU App Service plans

When you create a function app that uses the App Service plan, we recommend that you select a single-vCPU plan rather than a plan with multiple vCPUs. Today, Functions runs Node.js functions more efficiently on single-vCPU VMs, and using larger VMs doesn't produce the expected performance improvements. When necessary, you can manually scale out by adding more single-vCPU VM instances, or you can enable autoscale. For more information, see [Scale instance count manually or automatically](#).

Run from a package file

When you develop Azure Functions in the serverless hosting model, cold starts are a reality. *Cold start* refers to the first time your function app starts after a period of inactivity, taking longer to start up. For Node.js apps with large dependency trees in particular, cold start can be significant. To speed up the cold start process, [run your functions as a package file](#) when possible. Many deployment methods use this model by

default, but if you're experiencing large cold starts you should check to make sure you're running this way.

Use a single static client

When you use a service-specific client in an Azure Functions application, don't create a new client with every function invocation because you can hit connection limits. Instead, create a single, static client in the global scope. For more information, see [managing connections in Azure Functions](#).

Use `async` and `await`

When writing Azure Functions in Node.js, you should write code using the `async` and `await` keywords. Writing code using `async` and `await` instead of callbacks or `.then` and `.catch` with Promises helps avoid two common problems:

- Throwing uncaught exceptions that [crash the Node.js process ↗](#), potentially affecting the execution of other functions.
- Unexpected behavior, such as missing logs from `context.log`, caused by asynchronous calls that aren't properly awaited.

In the following example, the asynchronous method `fs.readFile` is invoked with an error-first callback function as its second parameter. This code causes both of the issues previously mentioned. An exception that isn't explicitly caught in the correct scope can crash the entire process (issue #1). Returning without ensuring the callback finishes means the http response will sometimes have an empty body (issue #2).

TypeScript

```
TypeScript

// DO NOT USE THIS CODE
import { app, HttpRequest, HttpResponseInit, InvocationContext } from
  '@azure/functions';
import * as fs from 'fs';

export async function httpTriggerBadAsync(request: HttpRequest, context: InvocationContext): Promise<HttpResponseInit> {
  let fileData: Buffer;
  fs.readFile('./helloWorld.txt', (err, data) => {
    if (err) {
      context.error(err);
      // BUG #1: This will result in an uncaught exception that
      crashes the entire process
      throw err;
    }
    fileData = data;
  })
  return {
    status: 200,
    body: fileData,
  };
}
```

```

        }
        fileData = data;
    });
    // BUG #2: fileData is not guaranteed to be set before the
    invocation ends
    return { body: fileData };
}

app.http('httpTriggerBadAsync', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    handler: httpTriggerBadAsync,
});

```

Use the `async` and `await` keywords to help avoid both of these issues. Most APIs in the Node.js ecosystem have been converted to support promises in some form. For example, starting in v14, Node.js provides an `fs/promises` API to replace the `fs` callback API.

In the following example, any unhandled exceptions thrown during the function execution only fail the individual invocation that raised the exception. The `await` keyword means that steps following `readFile` only execute after it's complete.

TypeScript

```

TypeScript

// Recommended pattern
import { app, HttpRequest, HttpResponseInit, InvocationContext } from
'@azure/functions';
import * as fs from 'fs/promises';

export async function httpTriggerGoodAsync(
    request: HttpRequest,
    context: InvocationContext
): Promise<HttpResponseInit> {
    try {
        const fileData = await fs.readFile('./helloWorld.txt');
        return { body: fileData };
    } catch (err) {
        context.error(err);
        // This rethrown exception will only fail the individual
        invocation, instead of crashing the whole process
        throw err;
    }
}

app.http('httpTriggerGoodAsync', {
    methods: ['GET', 'POST'],
}

```

```
    authLevel: 'anonymous',
    handler: httpTriggerGoodAsync,
});
```

Troubleshoot

See the [Node.js Troubleshoot guide](#).

Next steps

For more information, see the following resources:

- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Azure Functions Java developer guide

Article • 01/31/2024

This guide contains detailed information to help you succeed developing Azure Functions using Java.

As a Java developer, if you're new to Azure Functions, consider first reading one of the following articles:

[+] Expand table

Getting started	Concepts	Scenarios/samples
<ul style="list-style-type: none">Java function using Visual Studio CodeJava/Maven function with terminal/command promptJava function using GradleJava function using EclipseJava function using IntelliJ IDEA	<ul style="list-style-type: none">Developer guideHosting optionsPerformance considerations	<ul style="list-style-type: none">Java samples with different triggersEvent Hubs trigger and Azure Cosmos DB output bindingDependency injection samples ↗

Java function basics

A Java function is a `public` method, decorated with the annotation `@FunctionName`. This method defines the entry for a Java function, and must be unique in a particular package. The package can have multiple classes with multiple public methods annotated with `@FunctionName`. A single package is deployed to a function app in Azure. In Azure, the function app provides the deployment, execution, and management context for your individual Java functions.

Programming model

The concepts of [triggers and bindings](#) are fundamental to Azure Functions. Triggers start the execution of your code. Bindings give you a way to pass data to and return data from a function, without having to write custom data access code.

Create Java functions

To make it easier to create Java functions, there are Maven-based tooling and archetypes that use predefined Java templates to help you create projects with a specific function trigger.

Maven-based tooling

The following developer environments have Azure Functions tooling that lets you create Java function projects:

- [Visual Studio Code ↗](#)
- [Eclipse](#)
- [IntelliJ](#)

These articles show you how to create your first functions using your IDE of choice.

Project scaffolding

If you prefer command line development from the Terminal, the simplest way to scaffold Java-based function projects is to use [Apache Maven](#) archetypes. The Java Maven archetype for Azure Functions is published under the following *groupId:artifactId:com.microsoft.azure:azure-functions-archetype* ↗.

The following command generates a new Java function project using this archetype:

```
Bash
mvn archetype:generate \
-DarchetypeGroupId=com.microsoft.azure \
-DarchetypeArtifactId=azure-functions-archetype
```

To get started using this archetype, see the [Java quickstart](#).

Folder structure

Here's the folder structure of an Azure Functions Java project:

```
FunctionsProject
| - src
```

```
| | - main
| | | - java
| | | | - FunctionApp
| | | | | - MyFirstFunction.java
| | | | | - MySecondFunction.java
| - target
| | - azure-functions
| | | - FunctionApp
| | | | - FunctionApp.jar
| | | | - host.json
| | | | | - MyFirstFunction
| | | | | | - function.json
| | | | | - MySecondFunction
| | | | | | - function.json
| | | | - bin
| | | | - lib
| - pom.xml
```

You can use a shared [host.json](#) file to configure the function app. Each function has its own code file (.java) and binding configuration file (function.json).

You can put more than one function in a project. Avoid putting your functions into separate jars. The `FunctionApp` in the target directory is what gets deployed to your function app in Azure.

Triggers and annotations

Functions are invoked by a trigger, such as an HTTP request, a timer, or an update to data. Your function needs to process that trigger, and any other inputs, to produce one or more outputs.

Use the Java annotations included in the [com.microsoft.azure.functions.annotation.*](#) package to bind input and outputs to your methods. For more information, see the [Java reference docs](#).

Important

You must configure an Azure Storage account in your [local.settings.json](#) to run Azure Blob storage, Azure Queue storage, or Azure Table storage triggers locally.

Example:

Java

```
public class Function {
    public String echo(@HttpTrigger(name = "req",
```

```
        methods = {HttpMethod.POST}, authLevel =
AuthorizationLevel.ANONYMOUS)
    String req, ExecutionContext context) {
    return String.format(req);
}
}
```

Here's the generated corresponding `function.json` by the [azure-functions-maven-plugin](#):

JSON

```
{
  "scriptFile": "azure-functions-example.jar",
  "entryPoint": "com.example.Function.echo",
  "bindings": [
    {
      "type": "httpTrigger",
      "name": "req",
      "direction": "in",
      "authLevel": "anonymous",
      "methods": [ "GET", "POST" ]
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ]
}
```

Java versions

The version of Java on which your app runs in Azure is specified in the pom.xml file. The Maven archetype currently generates a pom.xml for Java 8, which you can change before publishing. The Java version in pom.xml should match the version on which you've locally developed and tested your app.

Supported versions

The following table shows current supported Java versions for each major version of the Functions runtime, by operating system:

[] [Expand table](#)

Functions version	Java versions (Windows)	Java versions (Linux)
4.x	17	21 (Preview)
	11	17
	8	11
		8
3.x	11	11
	8	8
2.x	8	n/a

Unless you specify a Java version for your deployment, the Maven archetype defaults to Java 8 during deployment to Azure.

Specify the deployment version

You can control the version of Java targeted by the Maven archetype by using the `-DjavaVersion` parameter. The value of this parameter can be either `8`, `11`, `17` or `21`.

The Maven archetype generates a pom.xml that targets the specified Java version. The following elements in pom.xml indicate the Java version to use:

[\[+\] Expand table](#)

Element	Java 8 value	Java 11 value	Java 17 value	Java 21 value (Preview, Linux)	Description
<code>Java.version</code>	1.8	11	17	21	Version of Java used by the maven-compiler-plugin.
<code>JavaVersion</code>	8	11	17	21	Java version hosted by the function app in Azure.

The following examples show the settings for Java 8 in the relevant sections of the pom.xml file:

`Java.version`

XML

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>

    <azure.functions.maven.plugin.version>1.6.0</azure.functions.maven.plugin.v
```

```
rsion>

<azure.functions.java.library.version>1.3.1</azure.functions.java.library.version>
    <functionAppName>fabrikam-functions-20200718015742191</functionAppName>
    <stagingDirectory>${project.build.directory}/azure-
functions/${functionAppName}</stagingDirectory>
</properties>
```

JavaVersion

XML

```
<runtime>
    <!-- runtime os, could be windows, linux or docker-->
    <os>windows</os>
    <javaVersion>8</javaVersion>
    <!-- for docker function, please set the following parameters -->
    <!-- <image>[hub-user/]repo-name[:tag]</image> -->
    <!-- <serverId></serverId> -->
    <!-- <registryUrl></registryUrl> -->
</runtime>
```

ⓘ Important

You must have the JAVA_HOME environment variable set correctly to the JDK directory that is used during code compiling using Maven. Make sure that the version of the JDK is at least as high as the `Java.version` setting.

Specify the deployment OS

Maven also lets you specify the operating system on which your function app runs in Azure. Use the `os` element to choose the operating system.

[+] Expand table

Element	Windows	Linux	Docker
<code>os</code>	windows	linux	docker

The following example shows the operating system setting in the `runtime` section of the `pom.xml` file:

XML

```
<runtime>
    <!-- runtime os, could be windows, linux or docker-->
    <os>windows</os>
    <javaVersion>8</javaVersion>
    <!-- for docker function, please set the following parameters -->
    <!-- <image>[hub-user/]repo-name[:tag]</image> -->
    <!-- <serverId></serverId> -->
    <!-- <registryUrl></registryUrl> -->
</runtime>
```

JDK runtime availability and support

Microsoft and [Adoptium](#) builds of OpenJDK are provided and supported on Functions for Java 8 (Adoptium), Java 11, 17 and 21 (MSFT). These binaries are provided as a no-cost, multi-platform, production-ready distribution of the OpenJDK for Azure. They contain all the components for building and running Java SE applications.

For local development or testing, you can download the [Microsoft build of OpenJDK](#) or [Adoptium Temurin](#) binaries for free. [Azure support](#) for issues with the JDKs and function apps is available with a [qualified support plan](#).

If you would like to continue using the Zulu for Azure binaries on your Function app, [configure your app accordingly](#). You can continue to use the Azul binaries for your site. However, any security patches or improvements are only available in new versions of the OpenJDK. Because of this, you should eventually remove this configuration so that your apps use the latest available version of Java.

Customize JVM

Functions lets you customize the Java virtual machine (JVM) used to run your Java functions. The [following JVM options](#) are used by default:

- `-XX:+TieredCompilation`
- `-XX:TieredStopAtLevel=1`
- `-noverify`
- `-Djava.net.preferIPv4Stack=true`
- `-jar`

You can provide other arguments to the JVM by using one of the following application settings, depending on the plan type:

Plan type	Setting name	Comment
Consumption plan	languageWorkers_java_arguments	This setting does increase the cold start times for Java functions running in a Consumption plan.
Premium plan Dedicated plan	JAVA_OPTS	

The following sections show you how to add these settings. To learn more about working with application settings, see the [Work with application settings](#) section.

Azure portal

In the [Azure portal](#), use the [Application Settings tab](#) to add either the `languageWorkers_java_arguments` or the `JAVA_OPTS` setting.

Azure CLI

You can use the `az functionapp config appsettings set` command to add these settings, as shown in the following example for the `-Djava.awt.headless=true` option:

The screenshot shows the Azure CLI interface. A dropdown menu is open, showing the option "Consumption plan". Below it, the command line interface shows the following command:

```
az functionapp config appsettings set \
    --settings "languageWorkers_java_arguments=-
Djava.awt.headless=true" \
    --name <APP_NAME> --resource-group <RESOURCE_GROUP>
```

This example enables headless mode. Replace `<APP_NAME>` with the name of your function app, and `<RESOURCE_GROUP>` with the resource group.

Third-party libraries

Azure Functions supports the use of third-party libraries. By default, all dependencies specified in your project `pom.xml` file are automatically bundled during the [mvn package](#) goal. For libraries not specified as dependencies in the `pom.xml` file, place them in a `lib`

directory in the function's root directory. Dependencies placed in the `lib` directory are added to the system class loader at runtime.

The `com.microsoft.azure.functions:azure-functions-java-library` dependency is provided on the classpath by default, and doesn't need to be included in the `lib` directory. Also, [azure-functions-java-worker](#) adds dependencies listed [here](#) to the classpath.

Data type support

You can use Plain old Java objects (POJOs), types defined in `azure-functions-java-library`, or primitive data types such as String and Integer to bind to input or output bindings.

POJOs

For converting input data to POJO, [azure-functions-java-worker](#) uses the `gson` library. POJO types used as inputs to functions should be `public`.

Binary data

Bind binary inputs or outputs to `byte[]`, by setting the `dataType` field in your `function.json` to `binary`:

```
Java

@FunctionName("BlobTrigger")
@StorageAccount("AzureWebJobsStorage")
public void blobTrigger(
    @BlobTrigger(name = "content", path = "myblob/{fileName}", dataType
= "binary") byte[] content,
    @BindingName("fileName") String fileName,
    final ExecutionContext context
) {
    context.getLogger().info("Java Blob trigger function processed a
blob.\n Name: " + fileName + "\n Size: " + content.length + " Bytes");
}
```

If you expect null values, use `Optional<T>`.

Bindings

Input and output bindings provide a declarative way to connect to data from within your code. A function can have multiple input and output bindings.

Input binding example

Java

```
package com.example;

import com.microsoft.azure.functions.annotation.*;

public class Function {
    @FunctionName("echo")
    public static String echo(
        @HttpTrigger(name = "req", methods = { HttpMethod.PUT }, authLevel =
AuthorizationLevel.ANONYMOUS, route = "items/{id}") String inputReq,
        @TableInput(name = "item", tableName = "items", partitionKey =
"Example", rowKey = "{id}", connection = "AzureWebJobsStorage")
TestInputData inputData,
        @TableOutput(name = "myOutputTable", tableName = "Person",
connection = "AzureWebJobsStorage") OutputBinding<Person> testOutputData
    ) {
        testOutputData.setValue(new Person(httpbody + "Partition", httpbody
+ "Row", httpbody + "Name"));
        return "Hello, " + inputReq + " and " + inputData.getKey() + ".";
    }

    public static class TestInputData {
        public String getKey() { return this.rowKey; }
        private String rowKey;
    }
    public static class Person {
        public String partitionKey;
        public String rowKey;
        public String name;

        public Person(String p, String r, String n) {
            this.partitionKey = p;
            this.rowKey = r;
            this.name = n;
        }
    }
}
```

You invoke this function with an HTTP request.

- HTTP request payload is passed as a `String` for the argument `inputReq`.
- One entry is retrieved from Table storage, and is passed as `TestInputData` to the argument `inputData`.

To receive a batch of inputs, you can bind to `String[]`, `POJO[]`, `List<String>`, or `List<POJO>`.

Java

```
@FunctionName("ProcessIotMessages")
public void processIotMessages(
    @EventHubTrigger(name = "message", eventHubName =
"%AzureWebJobsEventHubPath%", connection = "AzureWebJobsEventHubSender",
cardinality = Cardinality.MANY) List<TestEventData> messages,
    final ExecutionContext context)
{
    context.getLogger().info("Java Event Hub trigger received messages.
Batch size: " + messages.size());
}

public class TestEventData {
    public String id;
}
```

This function gets triggered whenever there's new data in the configured event hub. Because the `cardinality` is set to `MANY`, the function receives a batch of messages from the event hub. `EventData` from event hub gets converted to `TestEventData` for the function execution.

Output binding example

You can bind an output binding to the return value by using `$return`.

Java

```
package com.example;

import com.microsoft.azure.functions.annotation.*;

public class Function {
    @FunctionName("copy")
    @StorageAccount("AzureWebJobsStorage")
    @BlobOutput(name = "$return", path = "samples-output-java/{name}")
    public static String copy(@BlobTrigger(name = "blob", path = "samples-
input-java/{name}") String content) {
        return content;
    }
}
```

If there are multiple output bindings, use the return value for only one of them.

To send multiple output values, use `OutputBinding<T>` defined in the `azure-functions-java-library` package.

Java

```
@FunctionName("QueueOutputPOJOList")
    public HttpResponseMessage QueueOutputPOJOList(@HttpTrigger(name =
"req", methods = { HttpMethod.GET,
                    HttpMethod.POST }, authLevel = AuthorizationLevel.ANONYMOUS)
HttpRequestMessage<Optional<String>> request,
        @QueueOutput(name = "itemsOut", queueName = "test-output-java-
pojo", connection = "AzureWebJobsStorage") OutputBinding<List<TestData>>
itemsOut,
        final ExecutionContext context) {
    context.getLogger().info("Java HTTP trigger processed a request.");

    String query = request.getQueryParameters().get("queueMessageId");
    String queueMessageId = request.getBody().orElse(query);
    itemsOut.setValue(new ArrayList<TestData>());
    if (queueMessageId != null) {
        TestData testData1 = new TestData();
        testData1.id = "msg1"+queueMessageId;
        TestData testData2 = new TestData();
        testData2.id = "msg2"+queueMessageId;

        itemsOut.getValue().add(testData1);
        itemsOut.getValue().add(testData2);

        return request.createResponseBuilder(HttpStatus.OK).body("Hello,
" + queueMessageId).build();
    } else {
        return
request.createResponseBuilder(HttpStatus.INTERNAL_SERVER_ERROR)
            .body("Did not find expected items in CosmosDB input
list").build();
    }
}

public static class TestData {
    public String id;
}
```

You invoke this function on an `HttpRequest` object. It writes multiple values to Queue storage.

HttpRequestMessage and HttpResponseMessage

These are defined in `azure-functions-java-library`. They're helper types to work with `HttpTrigger` functions.

[+] Expand table

Specialized type	Target	Typical usage
<code>HttpRequestMessage<T></code>	HTTP Trigger	Gets method, headers, or queries
<code>HttpResponseMessage</code>	HTTP Output Binding	Returns status other than 200

Metadata

Few triggers send [trigger metadata](#) along with input data. You can use annotation `@BindingName` to bind to trigger metadata.

Java

```
package com.example;

import java.util.Optional;
import com.microsoft.azure.functions.annotation.*;

public class Function {
    @FunctionName("metadata")
    public static String metadata(
        @HttpTrigger(name = "req", methods = { HttpMethod.GET,
HttpMethod.POST }, authLevel = AuthorizationLevel.ANONYMOUS)
Optional<String> body,
        @BindingName("name") String queryValue
    ) {
        return body.orElse(queryValue);
    }
}
```

In the preceding example, the `queryValue` is bound to the query string parameter `name` in the HTTP request URL, `http://{example.host}/api/metadata?name=test`. Here's another example, showing how to bind to `Id` from queue trigger metadata.

Java

```
@FunctionName("QueueTriggerMetadata")
public void QueueTriggerMetadata(
    @QueueTrigger(name = "message", queueName = "test-input-javametadata",
connection = "AzureWebJobsStorage") String
message,@BindingName("Id") String metadataId,
```

```
    @QueueOutput(name = "output", queueName = "test-output-java-metadata", connection = "AzureWebJobsStorage") OutputBinding<TestData> output,
        final ExecutionContext context
    ) {
    context.getLogger().info("Java Queue trigger function processed a message: " + message + " with metadataId:" + metadataId );
    TestData testData = new TestData();
    testData.id = metadataId;
    output.setValue(testData);
}
```

ⓘ Note

The name provided in the annotation needs to match the metadata property.

Execution context

`ExecutionContext`, defined in the `azure-functions-java-library`, contains helper methods to communicate with the functions runtime. For more information, see the [ExecutionContext reference article](#).

Logger

Use `getLogger`, defined in `ExecutionContext`, to write logs from function code.

Example:

Java

```
import com.microsoft.azure.functions.*;
import com.microsoft.azure.functions.annotation.*;

public class Function {
    public String echo(@HttpTrigger(name = "req", methods =
{HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS) String req,
ExecutionContext context) {
        if (req.isEmpty()) {
            context.getLogger().warning("Empty request body received by
function " + context.getFunctionName() + " with invocation " +
context.getInvocationId());
        }
        return String.format(req);
    }
}
```

View logs and trace

You can use the Azure CLI to stream Java stdout and stderr logging, and other application logging.

Here's how to configure your function app to write application logging by using the Azure CLI:

Bash

Azure CLI

```
az webapp log config --name functionname --resource-group  
myResourceGroup --application-logging true
```

To stream logging output for your function app by using the Azure CLI, open a new command prompt, Bash, or Terminal session, and enter the following command:

Bash

Azure CLI

```
az webapp log tail --name webappname --resource-group myResourceGroup
```

The `az webapp log tail` command has options to filter output by using the `--provider` option.

To download the log files as a single ZIP file by using the Azure CLI, open a new command prompt, Bash, or Terminal session, and enter the following command:

Azure CLI

```
az webapp log download --resource-group resourcegroupname --name  
functionappname
```

You must have enabled file system logging in the Azure portal or the Azure CLI before running this command.

Environment variables

In Functions, [app settings](#), such as service connection strings, are exposed as environment variables during execution. You can access these settings by using, `System.getenv("AzureWebJobsStorage")`.

The following example gets the [application setting](#), with the key named `myAppSetting`:

Java

```
public class Function {
    public String echo(@HttpTrigger(name = "req", methods =
{HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS) String req,
ExecutionContext context) {
        context.getLogger().info("My app setting value: "+
System.getenv("myAppSetting"));
        return String.format(req);
    }
}
```

Use dependency injection in Java Functions

Azure Functions Java supports the dependency injection (DI) software design pattern, which is a technique to achieve [Inversion of Control \(IoC\)](#) between classes and their dependencies. Java Azure Functions provides a hook to integrate with popular Dependency Injection frameworks in your Functions Apps. [Azure Functions Java SPI](#) contains an interface [FunctionInstanceInjector](#). By implementing this interface, you can return an instance of your function class and your functions will be invoked on this instance. This gives frameworks like [Spring](#), [Quarkus](#), Google Guice, Dagger, etc. the ability to create the function instance and register it into their IOC container. This means you can use those Dependency Injection frameworks to manage your functions naturally.

Note

Microsoft Azure Functions Java SPI Types ([azure-function-java-spi](#)) is a package that contains all SPI interfaces for third parties to interact with Microsoft Azure functions runtime.

Function instance injector for dependency injection

[azure-function-java-spi](#) contains an interface `FunctionInstanceInjector`

Java

```
package com.microsoft.azure.functions.spi.inject;

/**
 * The instance factory used by DI framework to initialize function
instance.

 *
 * @since 1.0.0
 */

public interface FunctionInstanceInjector {

    /**
     * This method is used by DI framework to initialize the function
instance. This method takes in the customer class and returns

     * an instance create by the DI framework, later customer functions will
be invoked on this instance.

     * @param functionClass the class that contains customer functions

     * @param <T> customer functions class type

     * @return the instance that will be invoked on by azure functions java
worker

     * @throws Exception any exception that is thrown by the DI framework
during instance creation

    */

    <T> T getInstance(Class<T> functionClass) throws Exception;
}
```

For more examples that use `FunctionInstanceInjector` to integrate with Dependency injection frameworks refer to [this](#) repository.

Next steps

For more information about Azure Functions Java development, see the following resources:

- Best practices for Azure Functions
- Azure Functions developer reference
- Azure Functions triggers and bindings
- Local development and debug with [Visual Studio Code](#), [IntelliJ](#), and [Eclipse](#)
- [Remote Debug Java functions using Visual Studio Code](#)
- [Maven plugin for Azure Functions](#)
- Streamline function creation through the `azure-functions:add` goal, and prepare a staging directory for [ZIP file deployment](#).

Azure Functions PowerShell developer guide

Article • 03/09/2023

This article provides details about how you write Azure Functions using PowerShell.

A PowerShell Azure function (function) is represented as a PowerShell script that executes when triggered. Each function script has a related `function.json` file that defines how the function behaves, such as how it's triggered and its input and output parameters. To learn more, see the [Triggers and binding article](#).

Like other kinds of functions, PowerShell script functions take in parameters that match the names of all the input bindings defined in the `function.json` file. A `TriggerMetadata` parameter is also passed that contains additional information on the trigger that started the function.

This article assumes that you have already read the [Azure Functions developer reference](#). You should have also completed the [Functions quickstart for PowerShell](#) to create your first PowerShell function.

Folder structure

The required folder structure for a PowerShell project looks like the following. This default can be changed. For more information, see the `scriptFile` section below.

```
PSFunctionApp
| - MyFirstFunction
| | - run.ps1
| | - function.json
| - MySecondFunction
| | - run.ps1
| | - function.json
| - Modules
| | - myFirstHelperModule
| | | - myFirstHelperModule.psd1
| | | - myFirstHelperModule.psm1
| | - mySecondHelperModule
| | | - mySecondHelperModule.psd1
| | | - mySecondHelperModule.psm1
| - local.settings.json
| - host.json
| - requirements.psd1
| - profile.ps1
```

```
| - extensions.csproj  
| - bin
```

At the root of the project, there's a shared `host.json` file that can be used to configure the function app. Each function has a folder with its own code file (`.ps1`) and binding configuration file (`function.json`). The name of the `function.json` file's parent directory is always the name of your function.

Certain bindings require the presence of an `extensions.csproj` file. Binding extensions, required in [version 2.x and later versions](#) of the Functions runtime, are defined in the `extensions.csproj` file, with the actual library files in the `bin` folder. When developing locally, you must [register binding extensions](#). When developing functions in the Azure portal, this registration is done for you.

In PowerShell Function Apps, you may optionally have a `profile.ps1` which runs when a function app starts to run (otherwise known as a [cold start](#)). For more information, see [PowerShell profile](#).

Defining a PowerShell script as a function

By default, the Functions runtime looks for your function in `run.ps1`, where `run.ps1` shares the same parent directory as its corresponding `function.json`.

Your script is passed a number of arguments on execution. To handle these parameters, add a `param` block to the top of your script as in the following example:

PowerShell

```
# $TriggerMetadata is optional here. If you don't need it, you can safely  
remove it from the param block  
param($MyFirstInputBinding, $MySecondInputBinding, $TriggerMetadata)
```

TriggerMetadata parameter

The `TriggerMetadata` parameter is used to supply additional information about the trigger. The additional metadata varies from binding to binding but they all contain a `sys` property that contains the following data:

PowerShell

```
$TriggerMetadata.sys
```

Property	Description	Type
UtcNow	When, in UTC, the function was triggered	DateTime
MethodName	The name of the Function that was triggered	string
RandGuid	a unique guid to this execution of the function	string

Every trigger type has a different set of metadata. For example, the `$TriggerMetadata` for `QueueTrigger` contains the `InsertionTime`, `Id`, `DequeueCount`, among other things. For more information on the queue trigger's metadata, go to the [official documentation for queue triggers](#). Check the documentation on the [triggers](#) you're working with to see what comes inside the trigger metadata.

Bindings

In PowerShell, [bindings](#) are configured and defined in a function's `function.json`. Functions interact with bindings a number of ways.

Reading trigger and input data

Trigger and input bindings are read as parameters passed to your function. Input bindings have a `direction` set to `in` in `function.json`. The `name` property defined in `function.json` is the name of the parameter, in the `param` block. Since PowerShell uses named parameters for binding, the order of the parameters doesn't matter. However, it's a best practice to follow the order of the bindings defined in the `function.json`.

PowerShell

```
param($MyFirstInputBinding, $MySecondInputBinding)
```

Writing output data

In Functions, an output binding has a `direction` set to `out` in the `function.json`. You can write to an output binding by using the `Push-OutputBinding` cmdlet, which is available to the Functions runtime. In all cases, the `name` property of the binding as defined in `function.json` corresponds to the `Name` parameter of the `Push-OutputBinding` cmdlet.

The following shows how to call `Push-OutputBinding` in your function script:

PowerShell

```
param($MyFirstInputBinding, $MySecondInputBinding)

Push-OutputBinding -Name myQueue -Value $myValue
```

You can also pass in a value for a specific binding through the pipeline.

PowerShell

```
param($MyFirstInputBinding, $MySecondInputBinding)

Produce-MyOutputValue | Push-OutputBinding -Name myQueue
```

`Push-OutputBinding` behaves differently based on the value specified for `-Name`:

- When the specified name cannot be resolved to a valid output binding, then an error is thrown.
- When the output binding accepts a collection of values, you can call `Push-OutputBinding` repeatedly to push multiple values.
- When the output binding only accepts a singleton value, calling `Push-OutputBinding` a second time raises an error.

Push-OutputBinding syntax

The following are valid parameters for calling `Push-OutputBinding`:

Name	Type	Position	Description
<code>-Name</code>	String	1	The name of the output binding you want to set.
<code>-Value</code>	Object	2	The value of the output binding you want to set, which is accepted from the pipeline <code>ByValue</code> .
<code>-</code> <code>Clobber</code>	SwitchParameter	Named	(Optional) When specified, forces the value to be set for a specified output binding.

The following common parameters are also supported:

- `Verbose`
- `Debug`
- `ErrorAction`
- `ErrorVariable`
- `WarningAction`

- `WarningVariable`
- `OutBuffer`
- `PipelineVariable`
- `OutVariable`

For more information, see [About CommonParameters](#).

Push-OutputBinding example: HTTP responses

An HTTP trigger returns a response using an output binding named `response`. In the following example, the output binding of `response` has the value of "output #1":

PowerShell

```
PS >Push-OutputBinding -Name response -Value ([HttpResponseContext]@{  
    StatusCode = [System.Net.HttpStatusCode]::OK  
    Body = "output #1"  
})
```

Because the output is to HTTP, which accepts a singleton value only, an error is thrown when `Push-OutputBinding` is called a second time.

PowerShell

```
PS >Push-OutputBinding -Name response -Value ([HttpResponseContext]@{  
    StatusCode = [System.Net.HttpStatusCode]::OK  
    Body = "output #2"  
})
```

For outputs that only accept singleton values, you can use the `-Clobber` parameter to override the old value instead of trying to add to a collection. The following example assumes that you have already added a value. By using `-Clobber`, the response from the following example overrides the existing value to return a value of "output #3":

PowerShell

```
PS >Push-OutputBinding -Name response -Value ([HttpResponseContext]@{  
    StatusCode = [System.Net.HttpStatusCode]::OK  
    Body = "output #3"  
}) -Clobber
```

Push-OutputBinding example: Queue output binding

`Push-OutputBinding` is used to send data to output bindings, such as an [Azure Queue storage output binding](#). In the following example, the message written to the queue has a value of "output #1":

PowerShell

```
PS >Push-OutputBinding -Name outQueue -Value "output #1"
```

The output binding for a Storage queue accepts multiple output values. In this case, calling the following example after the first writes to the queue a list with two items: "output #1" and "output #2".

PowerShell

```
PS >Push-OutputBinding -Name outQueue -Value "output #2"
```

The following example, when called after the previous two, adds two more values to the output collection:

PowerShell

```
PS >Push-OutputBinding -Name outQueue -Value @("output #3", "output #4")
```

When written to the queue, the message contains these four values: "output #1", "output #2", "output #3", and "output #4".

Get-OutputBinding cmdlet

You can use the `Get-OutputBinding` cmdlet to retrieve the values currently set for your output bindings. This cmdlet retrieves a hashtable that contains the names of the output bindings with their respective values.

The following is an example of using `Get-OutputBinding` to return current binding values:

PowerShell

```
Get-OutputBinding
```

Output

Name	Value
-----	-----

```
MyQueue  
MyOtherQueue
```

```
myData  
myData
```

`Get-OutputBinding` also contains a parameter called `-Name`, which can be used to filter the returned binding, as in the following example:

```
PowerShell
```

```
Get-OutputBinding -Name MyQ*
```

```
Output
```

Name	Value
-----	-----
MyQueue	myData

Wildcards (*) are supported in `Get-OutputBinding`.

Logging

Logging in PowerShell functions works like regular PowerShell logging. You can use the logging cmdlets to write to each output stream. Each cmdlet maps to a log level used by Functions.

Functions logging level	Logging cmdlet
Error	<code>Write-Error</code>
Warning	<code>Write-Warning</code>
Information	<code>Write-Information</code> <code>Write-Host</code> <code>Write-Output</code> Writes to the <code>Information</code> log level.
Debug	<code>Write-Debug</code>
Trace	<code>Write-Progress</code> <code>Write-Verbose</code>

In addition to these cmdlets, anything written to the pipeline is redirected to the `Information` log level and displayed with the default PowerShell formatting.

 **Important**

Using the `Write-Verbose` or `Write-Debug` cmdlets is not enough to see verbose and debug level logging. You must also configure the log level threshold, which declares what level of logs you actually care about. To learn more, see [Configure the function app log level](#).

Configure the function app log level

Azure Functions lets you define the threshold level to make it easy to control the way Functions writes to the logs. To set the threshold for all traces written to the console, use the `logging.logLevel.default` property in the [host.json file](#). This setting applies to all functions in your function app.

The following example sets the threshold to enable verbose logging for all functions, but sets the threshold to enable debug logging for a function named `MyFunction`:

JSON

```
{  
  "logging": {  
    "logLevel": {  
      "Function.MyFunction": "Debug",  
      "default": "Trace"  
    }  
  }  
}
```

For more information, see [host.json reference](#).

Viewing the logs

If your Function App is running in Azure, you can use Application Insights to monitor it. Read [monitoring Azure Functions](#) to learn more about viewing and querying function logs.

If you're running your Function App locally for development, logs default to the file system. To see the logs in the console, set the `AZURE_FUNCTIONS_ENVIRONMENT` environment variable to `Development` before starting the Function App.

Triggers and bindings types

There are a number of triggers and bindings available to you to use with your function app. The full list of triggers and bindings [can be found here](#).

All triggers and bindings are represented in code as a few real data types:

- Hashtable
- string
- byte[]
- int
- double
- HttpRequestContext
- HttpResponseMessage

The first five types in this list are standard .NET types. The last two are used only by the [HttpTrigger trigger](#).

Each binding parameter in your functions must be one of these types.

HTTP triggers and bindings

HTTP and webhook triggers and HTTP output bindings use request and response objects to represent the HTTP messaging.

Request object

The request object that's passed into the script is of the type `HttpRequestContext`, which has the following properties:

Property	Description	Type
<code>Body</code>	An object that contains the body of the request. <code>Body</code> is serialized into the best type based on the data. For example, if the data is JSON, it's passed in as a hashtable. If the data is a string, it's passed in as a string.	object
<code>Headers</code>	A dictionary that contains the request headers.	<code>Dictionary<string,string></code> *
<code>Method</code>	The HTTP method of the request.	string
<code>Params</code>	An object that contains the routing parameters of the request.	<code>Dictionary<string,string></code> *
<code>Query</code>	An object that contains the query parameters.	<code>Dictionary<string,string></code> *
<code>Uri</code>	The URL of the request.	string

* All `Dictionary<string,string>` keys are case-insensitive.

Response object

The response object that you should send back is of the type `HttpContext`, which has the following properties:

Property	Description	Type
<code>Body</code>	An object that contains the body of the response.	object
<code>ContentType</code>	A short hand for setting the content type for the response.	string
<code>Headers</code>	An object that contains the response headers.	Dictionary or Hashtable
<code>StatusCode</code>	The HTTP status code of the response.	string or int

Accessing the request and response

When you work with HTTP triggers, you can access the HTTP request the same way you would with any other input binding. It's in the `param` block.

Use an `HttpContext` object to return a response, as shown in the following:

`function.json`

JSON

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "anonymous"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "Response"
    }
  ]
}
```

`run.ps1`

PowerShell

```
param($req, $TriggerMetadata)

$name = $req.Query.Name

Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = [System.Net.HttpStatusCode]::OK
    Body = "Hello $name!"
})
```

The result of invoking this function would be:

```
PS > irm http://localhost:5001?Name=Functions
Hello Functions!
```

Type-casting for triggers and bindings

For certain bindings like the blob binding, you're able to specify the type of the parameter.

For example, to have data from Blob storage supplied as a string, add the following type cast to my `param` block:

PowerShell

```
param([string] $myBlob)
```

PowerShell profile

In PowerShell, there's the concept of a PowerShell profile. If you're not familiar with PowerShell profiles, see [About profiles](#).

In PowerShell Functions, the profile script is executed once per PowerShell worker instance in the app when first deployed and after being idled ([cold start](#)). When concurrency is enabled by setting the `PSWorkerInProcConcurrencyUpperBound` value, the profile script is run for each runspace created.

When you create a function app using tools, such as Visual Studio Code and Azure Functions Core Tools, a default `profile.ps1` is created for you. The default profile is maintained [on the Core Tools GitHub repository](#) and contains:

- Automatic MSI authentication to Azure.

- The ability to turn on the Azure PowerShell `AzureRM` PowerShell aliases if you would like.

PowerShell versions

The following table shows the PowerShell versions available to each major version of the Functions runtime, and the .NET version required:

Functions version	PowerShell version	.NET version
4.x	PowerShell 7.2	.NET 6

You can see the current version by printing `$PSVersionTable` from any function.

To learn more about Azure Functions runtime support policy, please refer to this [article](#)

Running local on a specific version

Support for PowerShell 7.0 in Azure Functions has ended on 3 December 2022. To use PowerShell 7.2 when running locally, you need to add the setting

`"FUNCTIONS_WORKER_RUNTIME_VERSION" : "7.2"` to the `Values` array in the `local.setting.json` file in the project root. When running locally on PowerShell 7.2, your `local.settings.json` file looks like the following example:

```
JSON

{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "powershell",
    "FUNCTIONS_WORKER_RUNTIME_VERSION" : "7.2"
  }
}
```

ⓘ Note

In PowerShell Functions, the value "`~7`" for `FUNCTIONS_WORKER_RUNTIME_VERSION` refers to "`7.0.x`". We do not automatically upgrade PowerShell Function apps that have "`~7`" to "`7.2`". Going forward, for PowerShell Function Apps, we will require that apps specify both the major and minor version they want to target. Hence, it is necessary to mention "`7.2`" if you want to target "`7.2.x`"

Changing the PowerShell version

Support for PowerShell 7.0 in Azure Functions has ended on 3 December 2022. To upgrade your Function App to PowerShell 7.2, ensure the value of FUNCTIONS_EXTENSION_VERSION is set to ~4. To learn how to do this, see [View and update the current runtime version](#).

Use the following steps to change the PowerShell version used by your function app. You can do this either in the Azure portal or by using PowerShell.

Portal

1. In the [Azure portal](#), browse to your function app.
2. Under **Settings**, choose **Configuration**. In the **General settings** tab, locate the **PowerShell version**.

The screenshot shows the Azure Functions Configuration page for the 'AzureFunctions-PowerShell' function app. The 'General settings' tab is selected. In the 'Stack settings' section, the 'PowerShell Core Version' dropdown is open, showing three options: 'PowerShell 7.2' (selected), 'PowerShell 7.2', and 'PowerShell 7.0'. A red box highlights the 'Save' button at the top right of the configuration pane. Another red box highlights the 'General settings' tab. A third red box highlights the 'Configuration' link in the left sidebar under the 'Settings' category. A red arrow points from the 'PowerShell 7.2' option in the dropdown to the selected item.

3. Choose your desired **PowerShell Core version** and select **Save**. When warned about the pending restart choose **Continue**. The function app restarts on the chosen PowerShell version.

The function app restarts after the change is made to the configuration.

Dependency management

Functions lets you leverage [PowerShell gallery](#) for managing dependencies. With dependency management enabled, the requirements.psd1 file is used to automatically download required modules. You enable this behavior by setting the `managedDependency` property to `true` in the root of the [host.json file](#), as in the following example:

JSON

```
{  
    "managedDependency": {  
        "enabled": true  
    }  
}
```

When you create a new PowerShell functions project, dependency management is enabled by default, with the Azure [Az module](#) included. The maximum number of modules currently supported is 10. The supported syntax is `MajorNumber.*` or exact module version, as shown in the following requirements.psd1 example:

PowerShell

```
@{  
    Az = '1.*'  
   SqlServer = '21.1.18147'  
}
```

When you update the requirements.psd1 file, updated modules are installed after a restart.

Target specific versions

You may want to target a specific version of a module in your requirements.psd1 file. For example, if you wanted to use an older version of Az.Accounts than the one in the included Az module, you would need to target a specific version as shown in the following example:

PowerShell

```
@{  
    'Az.Accounts' = '1.9.5'  
}
```

In this case, you also need to add an import statement to the top of your profile.ps1 file, which looks like the following example:

PowerShell

```
Import-Module Az.Accounts -RequiredVersion '1.9.5'
```

In this way, the older version of the Az.Account module is loaded first when the function is started.

Dependency management considerations

The following considerations apply when using dependency management:

- Managed dependencies require access to <https://www.powershellgallery.com> to download modules. When running locally, make sure that the runtime can access this URL by adding any required firewall rules.
- Managed dependencies currently don't support modules that require the user to accept a license, either by accepting the license interactively, or by providing `-AcceptLicense` switch when invoking `Install-Module`.

Dependency management app settings

The following application settings can be used to change how the managed dependencies are downloaded and installed.

Function App setting	Default value	Description
MDMaxBackgroundUpgradePeriod	7.00:00:00 (seven days)	Controls the background update period for PowerShell function apps. To learn more, see MDMaxBackgroundUpgradePeriod .
MDSnapshotCheckPeriod	01:00:00 (one hour)	Specifies how often each PowerShell worker checks whether managed dependency upgrades have been installed. To learn more, see MDSnapshotCheckPeriod .
MDMinBackgroundUpgradePeriod	1.00:00:00 (one day)	The period of time after a previous upgrade check before another upgrade check is started. To learn more, see MDMinBackgroundUpgradePeriod .

Essentially, your app upgrade starts within `MDMaxBackgroundUpgradePeriod`, and the upgrade process completes within approximately the `MDSnapshotCheckPeriod`.

Custom modules

Leveraging your own custom modules in Azure Functions differs from how you would do it normally for PowerShell.

On your local computer, the module gets installed in one of the globally available folders in your `$env:PSModulePath`. When running in Azure, you don't have access to the modules installed on your machine. This means that the `$env:PSModulePath` for a PowerShell function app differs from `$env:PSModulePath` in a regular PowerShell script.

In Functions, `PSModulePath` contains two paths:

- A `Modules` folder that exists at the root of your function app.
- A path to a `Modules` folder that is controlled by the PowerShell language worker.

Function app-level modules folder

To use custom modules, you can place modules on which your functions depend in a `Modules` folder. From this folder, modules are automatically available to the functions runtime. Any function in the function app can use these modules.

ⓘ Note

Modules specified in the `requirements.psd1` file are automatically downloaded and included in the path so you don't need to include them in the modules folder. These are stored locally in the `$env:LOCALAPPDATA/AzureFunctions` folder and in the `/data/ManagedDependencies` folder when run in the cloud.

To take advantage of the custom module feature, create a `Modules` folder in the root of your function app. Copy the modules you want to use in your functions to this location.

PowerShell

```
mkdir ./Modules  
Copy-Item -Path /mymodules/mycustommodule -Destination ./Modules -Recurse
```

With a `Modules` folder, your function app should have the following folder structure:

```
PSFunctionApp  
| - MyFunction
```

```
| | - run.ps1
| | - function.json
| - Modules
| | - MyCustomModule
| | - MyOtherCustomModule
| | - MySpecialModule.psm1
| - local.settings.json
| - host.json
| - requirements.psd1
```

When you start your function app, the PowerShell language worker adds this `Modules` folder to the `$env:PSModulePath` so that you can rely on module autoloading just as you would in a regular PowerShell script.

Language worker level modules folder

Several modules are commonly used by the PowerShell language worker. These modules are defined in the last position of `PSModulePath`.

The current list of modules is as follows:

- [Microsoft.PowerShell.Archive](#) : module used for working with archives, like `.zip`, `.nupkg`, and others.
- `ThreadJob`: A thread-based implementation of the PowerShell job APIs.

By default, Functions uses the most recent version of these modules. To use a specific module version, put that specific version in the `Modules` folder of your function app.

Environment variables

In Functions, [app settings](#), such as service connection strings, are exposed as environment variables during execution. You can access these settings using `$env:NAME_OF_ENV_VAR`, as shown in the following example:

PowerShell

```
param($myTimer)

Write-Host "PowerShell timer trigger function ran! $(Get-Date)"
Write-Host $env:AzureWebJobsStorage
Write-Host $env:WEBSITE_SITE_NAME
```

There are several ways that you can add, update, and delete function app settings:

- In the Azure portal.
- By using the Azure CLI.
- By using Azure PowerShell.

Changes to function app settings require your function app to be restarted.

When running locally, app settings are read from the [local.settings.json](#) project file.

Concurrency

By default, the Functions PowerShell runtime can only process one invocation of a function at a time. However, this concurrency level might not be sufficient in the following situations:

- When you're trying to handle a large number of invocations at the same time.
- When you have functions that invoke other functions inside the same function app.

There are a few concurrency models that you could explore depending on the type of workload:

- Increase `FUNCTIONS_WORKER_PROCESS_COUNT`. This allows handling function invocations in multiple processes within the same instance, which introduces certain CPU and memory overhead. In general, I/O-bound functions will not suffer from this overhead. For CPU-bound functions, the impact may be significant.
- Increase the `PSWorkerInProcConcurrencyUpperBound` app setting value. This allows creating multiple runspaces within the same process, which significantly reduces CPU and memory overhead.

You set these environment variables in the [app settings](#) of your function app.

Depending on your use case, Durable Functions may significantly improve scalability. To learn more, see [Durable Functions application patterns](#).

ⓘ Note

You might get "requests are being queued due to no available runspaces" warnings, please note that this is not an error. The message is telling you that requests are being queued and they will be handled when the previous requests are completed.

Considerations for using concurrency

PowerShell is a *single-threaded* scripting language by default. However, concurrency can be added by using multiple PowerShell runspaces in the same process. The number of runspaces created, and therefore the number of concurrent threads per worker, is limited by the `PSWorkerInProcConcurrencyUpperBound` application setting. By default, the number of runspaces is set to 1,000 in version 4.x of the Functions runtime. In versions 3.x and below, the maximum number of runspaces is set to 1. The throughput will be impacted by the amount of CPU and memory available in the selected plan.

Azure PowerShell uses some *process-level* contexts and state to help save you from excess typing. However, if you turn on concurrency in your function app and invoke actions that change state, you could end up with race conditions. These race conditions are difficult to debug because one invocation relies on a certain state and the other invocation changed the state.

There's immense value in concurrency with Azure PowerShell, since some operations can take a considerable amount of time. However, you must proceed with caution. If you suspect that you're experiencing a race condition, set the `PSWorkerInProcConcurrencyUpperBound` app setting to `1` and instead use [language worker process level isolation](#) for concurrency.

Configure function `scriptFile`

By default, a PowerShell function is executed from `run.ps1`, a file that shares the same parent directory as its corresponding `function.json`.

The `scriptFile` property in the `function.json` can be used to get a folder structure that looks like the following example:

```
FunctionApp
| - host.json
| - myFunction
| | - function.json
| - lib
| | - PSFunction.ps1
```

In this case, the `function.json` for `myFunction` includes a `scriptFile` property referencing the file with the exported function to run.

JSON

```
{  
  "scriptFile": "../lib/PSFunction.ps1",  
  "bindings": [  
    // ...  
  ]  
}
```

Use PowerShell modules by configuring an entryPoint

This article has shown PowerShell functions in the default `run.ps1` script file generated by the templates. However, you can also include your functions in PowerShell modules. You can reference your specific function code in the module by using the `scriptFile` and `entryPoint` fields in the `function.json` configuration file.`

In this case, `entryPoint` is the name of a function or cmdlet in the PowerShell module referenced in `scriptFile`.

Consider the following folder structure:

```
FunctionApp  
| - host.json  
| - myFunction  
| | - function.json  
| - lib  
| | - PSFunction.psm1
```

Where `PSFunction.psm1` contains:

```
PowerShell  
  
function Invoke-PSTestFunc {  
    param($InputBinding, $TriggerMetadata)  
  
    Push-OutputBinding -Name OutputBinding -Value "output"  
}  
  
Export-ModuleMember -Function "Invoke-PSTestFunc"
```

In this example, the configuration for `myFunction` includes a `scriptFile` property that references `PSFunction.psm1`, which is a PowerShell module in another folder. The

`entryPoint` property references the `Invoke-PSTestFunc` function, which is the entry point in the module.

JSON

```
{  
  "scriptFile": "../lib/PSFunction.psm1",  
  "entryPoint": "Invoke-PSTestFunc",  
  "bindings": [  
    // ...  
  ]  
}
```

With this configuration, the `Invoke-PSTestFunc` gets executed exactly as a `run.ps1` would.

Considerations for PowerShell functions

When you work with PowerShell functions, be aware of the considerations in the following sections.

Cold Start

When developing Azure Functions in the [serverless hosting model](#), cold starts are a reality. *Cold start* refers to period of time it takes for your function app to start running to process a request. Cold start happens more frequently in the Consumption plan because your function app gets shut down during periods of inactivity.

Bundle modules instead of using Install-Module

Your script is run on every invocation. Avoid using `Install-Module` in your script. Instead use `Save-Module` before publishing so that your function doesn't have to waste time downloading the module. If cold starts are impacting your functions, consider deploying your function app to an [App Service plan](#) set to *always on* or to a [Premium plan](#).

Next steps

For more information, see the following resources:

- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)

Azure Functions Python developer guide

Article • 07/30/2024

This guide is an introduction to developing Azure Functions by using Python. The article assumes that you've already read the [Azure Functions developers guide](#).

Important

This article supports both the v1 and v2 programming model for Python in Azure Functions. The Python v1 model uses a `functions.json` file to define functions, and the new v2 model lets you instead use a decorator-based approach. This new approach results in a simpler file structure, and it's more code-centric. Choose the **v2** selector at the top of the article to learn about this new programming model.

As a Python developer, you might also be interested in these topics:

Get started

- [Visual Studio Code](#): Create your first Python app using Visual Studio Code.
- [Terminal or command prompt](#): Create your first Python app from the command prompt using Azure Functions Core Tools.
- [Samples](#): Review some existing Python apps in the Learn samples browser.

Development options

Both Python Functions programming models support local development in one of the following environments:

Python v2 programming model:

- [Visual Studio Code](#)
- [Terminal or command prompt](#)

Python v1 programming model:

- [Visual Studio Code](#)
- [Terminal or command prompt](#)

You can also create Python v1 functions in the Azure portal.

Tip

Although you can develop your Python-based Azure functions locally on Windows, Python is supported only on a Linux-based hosting plan when it's running in Azure. For more information, see the [list of supported operating system/runtime combinations](#).

Programming model

Azure Functions expects a function to be a stateless method in your Python script that processes input and produces output. By default, the runtime expects the method to be implemented as a global method in the *function_app.py* file.

Triggers and bindings can be declared and used in a function in a decorator based approach. They're defined in the same file, *function_app.py*, as the functions. As an example, the following *function_app.py* file represents a function trigger by an HTTP request.

Python

```
@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req):
    user = req.params.get("user")
    return f"Hello, {user}!"
```

You can also explicitly declare the attribute types and return type in the function by using Python type annotations. Doing so helps you use the IntelliSense and autocomplete features that are provided by many Python code editors.

Python

```
import azure.functions as func

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req: func.HttpRequest) -> str:
    user = req.params.get("user")
    return f"Hello, {user}!"
```

To learn about known limitations with the v2 model and their workarounds, see [Troubleshoot Python errors in Azure Functions](#).

Alternative entry point

The entry point is only in the *function_app.py* file. However, you can reference functions within the project in *function_app.py* by using [blueprints](#) or by importing.

Folder structure

The recommended folder structure for a Python functions project looks like the following example:

```
Windows Command Prompt

<project_root>/
| - .venv/
| - .vscode/
| - function_app.py
| - additional_functions.py
| - tests/
| | - test_my_function.py
| - .funcignore
| - host.json
| - local.settings.json
| - requirements.txt
| - Dockerfile
```

The main project folder, *<project_root>*, can contain the following files:

- *.venv/*: (Optional) Contains a Python virtual environment that's used by local development.
- *.vscode/*: (Optional) Contains the stored Visual Studio Code configuration. To learn more, see [Visual Studio Code settings](#).
- *function_app.py*: The default location for all functions and their related triggers and bindings.
- *additional_functions.py*: (Optional) Any other Python files that contain functions (usually for logical grouping) that are referenced in *function_app.py* through blueprints.
- *tests/*: (Optional) Contains the test cases of your function app.
- *.funcignore*: (Optional) Declares files that shouldn't get published to Azure. Usually, this file contains *.vscode/* to ignore your editor setting, *.venv/* to ignore local

Python virtual environment, `tests/` to ignore test cases, and `local.settings.json` to prevent local app settings being published.

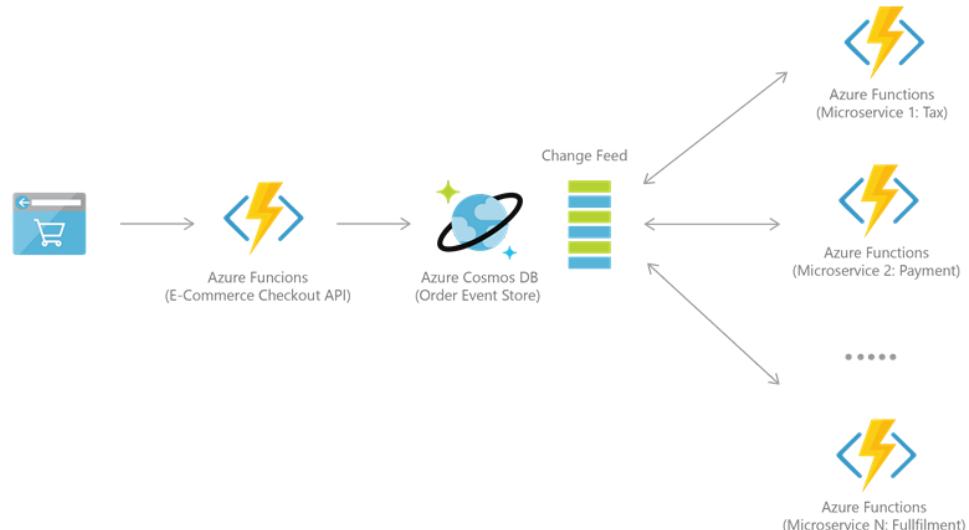
- `host.json`: Contains configuration options that affect all functions in a function app instance. This file does get published to Azure. Not all options are supported when running locally. To learn more, see [host.json](#).
- `local.settings.json`: Used to store app settings and connection strings when it's running locally. This file doesn't get published to Azure. To learn more, see [local.settings.json](#).
- `requirements.txt`: Contains the list of Python packages the system installs when it publishes to Azure.
- `Dockerfile`: (Optional) Used when publishing your project in a [custom container](#).

When you deploy your project to a function app in Azure, the entire contents of the main project folder, `<project_root>`, should be included in the package, but not the folder itself, which means that `host.json` should be in the package root. We recommend that you maintain your tests in a folder along with other functions (in this example, `tests/`). For more information, see [Unit testing](#).

Connect to a database

Azure Functions integrates well with [Azure Cosmos DB](#) for many [use cases](#), including IoT, ecommerce, gaming, etc.

For example, for [event sourcing](#), the two services are integrated to power event-driven architectures using Azure Cosmos DB's [change feed](#) functionality. The change feed provides downstream microservices the ability to reliably and incrementally read inserts and updates (for example, order events). This functionality can be used to provide a persistent event store as a message broker for state-changing events and drive order processing workflow between many microservices (which can be implemented as [serverless Azure Functions](#)).



To connect to Azure Cosmos DB, first [create an account, database, and container](#). Then you can connect your function code to Azure Cosmos DB using [trigger and bindings](#), like this [example](#).

To implement more complex app logic, you can also use the Python library for Cosmos DB. An asynchronous I/O implementation looks like this:

Python

```

pip install azure-cosmos
pip install aiohttp

from azure.cosmos.aio import CosmosClient
from azure.cosmos import exceptions
from azure.cosmos.partition_key import PartitionKey
import asyncio

# Replace these values with your Cosmos DB connection information
endpoint = "https://azure-cosmos-nosql.documents.azure.com:443/"
key = "master_key"
database_id = "cosmicwerx"
container_id = "cosmiccontainer"
partition_key = "/partition_key"

# Set the total throughput (RU/s) for the database and container
database_throughput = 1000

# Singleton CosmosClient instance
client = CosmosClient(endpoint, credential=key)

# Helper function to get or create database and container
async def get_or_create_container(client, database_id, container_id,
partition_key):

```

```

database = await client.create_database_if_not_exists(id=database_id)
print(f'Database "{database_id}" created or retrieved successfully.')

    container = await
database.create_container_if_not_exists(id=container_id,
partition_key=PartitionKey(path=partition_key))
    print(f'Container with id "{container_id}" created')

return container

async def create_products():
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    for i in range(10):
        await container.upsert_item({
            'id': f'item{i}',
            'productName': 'Widget',
            'productModel': f'Model {i}'
        })

async def get_products():
    items = []
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    async for item in container.read_all_items():
        items.append(item)
    return items

async def query_products(product_name):
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    query = f"SELECT * FROM c WHERE c.productName = '{product_name}'"
    items = []
    async for item in container.query_items(query=query,
enable_cross_partition_query=True):
        items.append(item)
    return items

async def main():
    await create_products()
    all_products = await get_products()
    print('All Products:', all_products)

    queried_products = await query_products('Widget')
    print('Queried Products:', queried_products)

if __name__ == "__main__":
    asyncio.run(main())

```

Blueprints

The Python v2 programming model introduces the concept of *blueprints*. A blueprint is a new class that's instantiated to register functions outside of the core function application. The functions registered in blueprint instances aren't indexed directly by the function runtime. To get these blueprint functions indexed, the function app needs to register the functions from blueprint instances.

Using blueprints provides the following benefits:

- Lets you break up the function app into modular components, which enables you to define functions in multiple Python files and divide them into different components per file.
- Provides extensible public function app interfaces to build and reuse your own APIs.

The following example shows how to use blueprints:

First, in an *http_blueprint.py* file, an HTTP-triggered function is first defined and added to a blueprint object.

Python

```
import logging

import azure.functions as func

bp = func.Blueprint()

@bp.route(route="default_template")
def default_template(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(
            f"Hello, {name}. This HTTP-triggered function "
            f"executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP-triggered function executed successfully. "
            "Pass a name in the query string or in the request body for a"
            " personalized response.",
```

```
    status_code=200
)
```

Next, in the `function_app.py` file, the `blueprint` object is imported and its functions are registered to the function app.

Python

```
import azure.functions as func
from http_blueprint import bp

app = func.FunctionApp()

app.register_functions(bp)
```

ⓘ Note

Durable Functions also supports blueprints. To create blueprints for Durable Functions apps, register your orchestration, activity, and entity triggers and client bindings using the [azure-functions-durable](#) Blueprint class, as shown [here](#). The resulting blueprint can then be registered as normal. See our [sample](#) for an example.

Triggers and inputs

Inputs are divided into two categories in Azure Functions: trigger input and other input. Although they're defined using different decorators, their usage is similar in Python code. Connection strings or secrets for trigger and input sources map to values in the `local.settings.json` file when they're running locally, and they map to the application settings when they're running in Azure.

As an example, the following code demonstrates how to define a Blob Storage input binding:

JSON

```
// local.settings.json
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "STORAGE_CONNECTION_STRING": "<AZURE_STORAGE_CONNECTION_STRING>",
    "AzureWebJobsStorage": "<azure-storage-connection-string>",
    "AzureWebJobsFeatureFlags": "EnableWorkerIndexing"
```

```
}
```

Python

```
# function_app.py
import azure.functions as func
import logging

app = func.FunctionApp()

@app.route(route="req")
@app.read_blob(arg_name="obj", path="samples/{id}",
               connection="STORAGE_CONNECTION_STRING")
def main(req: func.HttpRequest, obj: func.InputStream):
    logging.info(f'Python HTTP-triggered function processed: {obj.read()}')
```

When the function is invoked, the HTTP request is passed to the function as `req`. An entry is retrieved from the Azure Blob Storage account based on the *ID* in the route URL and made available as `obj` in the function body. Here, the specified storage account is the connection string that's found in the `STORAGE_CONNECTION_STRING` app setting.

For data intensive binding operations, you may want to use a separate storage account. For more information, see [Storage account guidance](#).

SDK type bindings (preview)

For select triggers and bindings, you can work with data types implemented by the underlying Azure SDKs and frameworks. These *SDK type bindings* let you interact with binding data as if you were using the underlying service SDK.

Functions supports Python SDK type bindings for Azure Blob storage, which lets you work with blob data using the underlying `BlobClient` type.

ⓘ Important

SDK type bindings support for Python is currently in preview:

- You must use the Python v2 programming model.
- Currently, only synchronous SDK types are supported.

Prerequisites

- Azure Functions runtime version version 4.34, or a later version.
- Python ↗ version 3.9, or a later supported version.

Enable SDK type bindings for the Blob storage extension

1. Add the `azurefunctions-extensions-bindings-blob` extension package to the `requirements.txt` file in the project, which should include at least these packages:

```
text  
  
azure-functions  
azurefunctions-extensions-bindings-blob
```

2. Add this code to the `function_app.py` file in the project, which imports the SDK type bindings:

```
Python  
  
import azurefunctions.extensions.bindings.blob as blob
```

SDK type bindings examples

This example shows how to get the `BlobClient` from both a Blob storage trigger (`blob_trigger`) and from the input binding on an HTTP trigger (`blob_input`):

```
Python  
  
import logging  
  
import azure.functions as func  
import azurefunctions.extensions.bindings.blob as blob  
  
app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)  
  
@app.blob_trigger(  
    arg_name="client", path="PATH/TO/BLOB", connection="AzureWebJobsStorage"  
)  
def blob_trigger(client: blob.BlobClient):  
    logging.info(  
        f"Python blob trigger function processed blob \n"  
        f"Properties: {client.get_blob_properties()}\n"  
        f"Blob content head: {client.download_blob().read(size=1)}"  
    )  
  
@app.route(route="file")
```

```
@app.blob_input(  
    arg_name="client", path="PATH/TO/BLOB", connection="AzureWebJobsStorage"  
)  
def blob_input(req: func.HttpRequest, client: blob.BlobClient):  
    logging.info(  
        f"Python blob input function processed blob \n"  
        f"Properties: {client.get_blob_properties()}\n"  
        f"Blob content head: {client.download_blob().read(size=1)}"  
    )  
    return "ok"
```

You can view other SDK type bindings samples for Blob storage in the Python extensions repository:

- [ContainerClient type ↗](#)
- [StorageStreamDownloader type ↗](#)

HTTP streams (preview)

HTTP streams lets you accept and return data from your HTTP endpoints using FastAPI request and response APIs enabled in your functions. These APIs lets the host process large data in HTTP messages as chunks instead of reading an entire message into memory.

This feature makes it possible to handle large data stream, OpenAI integrations, deliver dynamic content, and support other core HTTP scenarios requiring real-time interactions over HTTP. You can also use FastAPI response types with HTTP streams. Without HTTP streams, the size of your HTTP requests and responses are limited by memory restrictions that can be encountered when processing entire message payloads all in memory.

ⓘ Important

HTTP streams support for Python is currently in preview and requires you to use the Python v2 programming model.

Prerequisites

- [Azure Functions runtime](#) version 4.34.1, or a later version.
- [Python ↗](#) version 3.8, or a later [supported version](#).

Enable HTTP streams

HTTP streams are disabled by default. You need to enable this feature in your application settings and also update your code to use the FastAPI package. Note that when enabling HTTP streams, the function app will default to using HTTP streaming, and the original HTTP functionality will not work.

1. Add the `azurefunctions-extensions-http-fastapi` extension package to the `requirements.txt` file in the project, which should include at least these packages:

```
text  
  
azure-functions  
azurefunctions-extensions-http-fastapi
```

2. Add this code to the `function_app.py` file in the project, which imports the FastAPI extension:

```
Python  
  
from azurefunctions.extensions.http.fastapi import Request,  
StreamingResponse
```

3. When you deploy to Azure, add the following [application setting](#) in your function app:

```
"PYTHON_ENABLE_INIT_INDEXING": "1"
```

If you are deploying to Linux Consumption, also add

```
"PYTHON_ISOLATE_WORKER_DEPENDENCIES": "1"
```

When running locally, you also need to add these same settings to the `local.settings.json` project file.

HTTP streams examples

After you enable the HTTP streaming feature, you can create functions that stream data over HTTP.

This example is an HTTP triggered function that streams HTTP response data. You might use these capabilities to support scenarios like sending event data through a pipeline for real time visualization or detecting anomalies in large sets of data and providing instant notifications.

```
Python
```

```

import time

import azure.functions as func
from azurefunctions.extensions.http.fastapi import Request,
StreamingResponse

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

def generate_sensor_data():
    """Generate real-time sensor data."""
    for i in range(10):
        # Simulate temperature and humidity readings
        temperature = 20 + i
        humidity = 50 + i
        yield f"data: {{'temperature': {temperature}, 'humidity': {humidity}}}\n\n"
    time.sleep(1)

@app.route(route="stream", methods=[func.HttpMethod.GET])
async def stream_sensor_data(req: Request) -> StreamingResponse:
    """Endpoint to stream real-time sensor data."""
    return StreamingResponse(generate_sensor_data(), media_type="text/event-stream")

```

This example is an HTTP triggered function that receives and processes streaming data from a client in real time. It demonstrates streaming upload capabilities that can be helpful for scenarios like processing continuous data streams and handling event data from IoT devices.

Python

```

import azure.functions as func
from azurefunctions.extensions.http.fastapi import JSONResponse, Request

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="streaming_upload", methods=[func.HttpMethod.POST])
async def streaming_upload(req: Request) -> JSONResponse:
    """Handle streaming upload requests."""
    # Process each chunk of data as it arrives
    async for chunk in req.stream():
        process_data_chunk(chunk)

    # Once all data is received, return a JSON response indicating
    # successful processing
    return JSONResponse({"status": "Data uploaded and processed
successfully"})

```

```
def process_data_chunk(chunk: bytes):
    """Process each data chunk."""
    # Add custom processing logic here
    pass
```

Calling HTTP streams

You must use an HTTP client library to make streaming calls to a function's FastAPI endpoints. The client tool or browser you're using might not natively support streaming or could only return the first chunk of data.

You can use a client script like this to send streaming data to an HTTP endpoint:

Python

```
import httpx # Be sure to add 'httpx' to 'requirements.txt'
import asyncio

async def stream_generator(file_path):
    chunk_size = 2 * 1024 # Define your own chunk size
    with open(file_path, 'rb') as file:
        while chunk := file.read(chunk_size):
            yield chunk
            print(f"Sent chunk: {len(chunk)} bytes")

async def stream_to_server(url, file_path):
    timeout = httpx.Timeout(60.0, connect=60.0)
    async with httpx.AsyncClient(timeout=timeout) as client:
        response = await client.post(url,
content=stream_generator(file_path))
    return response

async def stream_response(response):
    if response.status_code == 200:
        async for chunk in response.aiter_raw():
            print(f"Received chunk: {len(chunk)} bytes")
    else:
        print(f"Error: {response}")

async def main():
    print('helloworld')
    # Customize your streaming endpoint served from core tool in variable
    'url' if different.
    url = 'http://localhost:7071/api/streaming_upload'
    file_path = r'<file path>'

    response = await stream_to_server(url, file_path)
    print(response)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Outputs

Output can be expressed both in return value and output parameters. If there's only one output, we recommend using the return value. For multiple outputs, you'll have to use output parameters.

To produce multiple outputs, use the `set()` method provided by the `azure.functions.Out` interface to assign a value to the binding. For example, the following function can push a message to a queue and also return an HTTP response.

Python

```
# function_app.py
import azure.functions as func

app = func.FunctionApp()

@app.write_blob(arg_name="msg", path="output-container/{name}",
                connection="CONNECTION_STRING")
def test_function(req: func.HttpRequest,
                  msg: func.Out[str]) -> str:

    message = req.params.get('body')
    msg.set(message)
    return message
```

Logging

Access to the Azure Functions runtime logger is available via a root [logging](#) handler in your function app. This logger is tied to Application Insights and allows you to flag warnings and errors that occur during the function execution.

The following example logs an info message when the function is invoked via an HTTP trigger.

Python

```
import logging

def main(req):
    logging.info('Python HTTP trigger function processed a request.')
```

More logging methods are available that let you write to the console at different trace levels:

[+] Expand table

Method	Description
<code>critical(_message_)</code>	Writes a message with level CRITICAL on the root logger.
<code>error(_message_)</code>	Writes a message with level ERROR on the root logger.
<code>warning(_message_)</code>	Writes a message with level WARNING on the root logger.
<code>info(_message_)</code>	Writes a message with level INFO on the root logger.
<code>debug(_message_)</code>	Writes a message with level DEBUG on the root logger.

To learn more about logging, see [Monitor Azure Functions](#).

Logging from created threads

To see logs coming from your created threads, include the `context` argument in the function's signature. This argument contains an attribute `thread_local_storage` that stores a local `invocation_id`. This can be set to the function's current `invocation_id` to ensure the context is changed.

Python

```
import azure.functions as func
import logging
import threading

def main(req, context):
    logging.info('Python HTTP trigger function processed a request.')
    t = threading.Thread(target=log_function, args=(context,))
    t.start()

def log_function(context):
    context.thread_local_storage.invocation_id = context.invocation_id
    logging.info('Logging from thread.')
```

Log custom telemetry

By default, the Functions runtime collects logs and other telemetry data that are generated by your functions. This telemetry ends up as traces in Application Insights. Request and dependency telemetry for certain Azure services are also collected by default by [triggers and bindings](#).

To collect custom request and custom dependency telemetry outside of bindings, you can use the [OpenCensus Python Extensions](#). This extension sends custom telemetry data to your Application Insights instance. You can find a list of supported extensions at the [OpenCensus repository](#).

ⓘ Note

To use the OpenCensus Python extensions, you need to enable [Python worker extensions](#) in your function app by setting `PYTHON_ENABLE_WORKER_EXTENSIONS` to `1`. You also need to switch to using the Application Insights connection string by adding the [APPLICATIONINSIGHTS CONNECTION STRING](#) setting to your [application settings](#), if it's not already there.

text

```
// requirements.txt
...
opencensus-extension-azure-functions
opencensus-ext-requests
```

Python

```
import json
import logging

import requests
from opencensus.extension.azure.functions import OpenCensusExtension
from opencensus.trace import config_integration

config_integration.trace_integrations(['requests'])

OpenCensusExtension.configure()

def main(req, context):
    logging.info('Executing HttpTrigger with OpenCensus extension')

    # You must use contexttracer to create spans
    with context.tracer.span("parent"):
        response = requests.get(url='http://example.com')

    return json.dumps({
        'method': req.method,
```

```
'response': response.status_code,
'ctx_func_name': context.function_name,
'ctx_func_dir': context.function_directory,
'ctx_invocation_id': context.invocation_id,
'ctx_trace_context_Traceparent': context.trace_context.Traceparent,
'ctx_trace_context_Tracestate': context.trace_context.Tracestate,
'ctx_retry_context_RetryCount': context.retry_context.retry_count,
'ctx_retry_context_MaxRetryCount':
context.retry_context.max_retry_count,
})
```

HTTP trigger

The HTTP trigger is defined as a method that takes a named binding parameter, which is an `HttpRequest` object, and returns an `HttpResponse` object. You apply the `function_name` decorator to the method to define the function name, while the HTTP endpoint is set by applying the `route` decorator.

This example is from the HTTP trigger template for the Python v2 programming model, where the binding parameter name is `req`. It's the sample code that's provided when you create a function by using Azure Functions Core Tools or Visual Studio Code.

Python

```
@app.function_name(name="HttpTrigger1")
@app.route(route="hello")
def test_function(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
    else:
        name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello, {name}. This HTTP-triggered
function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP-triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
            status_code=200
        )
```

From the `HttpRequest` object, you can get request headers, query parameters, route parameters, and the message body. In this function, you obtain the value of the `name` query parameter from the `params` parameter of the `HttpRequest` object. You read the JSON-encoded message body by using the `get_json` method.

Likewise, you can set the `status_code` and `headers` for the response message in the returned `HttpResponse` object.

To pass in a name in this example, paste the URL that's provided when you're running the function, and then append it with `?name={name}`.

Web frameworks

You can use Asynchronous Server Gateway Interface (ASGI)-compatible and Web Server Gateway Interface (WSGI)-compatible frameworks, such as Flask and FastAPI, with your HTTP-triggered Python functions. You must first update the `host.json` file to include an HTTP `routePrefix`, as shown in the following example:

```
JSON

{
  "version": "2.0",
  "logging": {
    {
      "applicationInsights": {
        {
          "samplingSettings": {
            {
              "isEnabled": true,
              "excludedTypes": "Request"
            }
          }
        },
        "extensionBundle": {
          {
            "id": "Microsoft.Azure.Functions.ExtensionBundle",
            "version": "[2.*, 3.0.0)"
          },
          "extensions": {
            {
              "http": {
                {
                  "routePrefix": ""
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The framework code looks like the following example:

ASGI

`AsgiFunctionApp` is the top-level function app class for constructing ASGI HTTP functions.

Python

```
# function_app.py

import azure.functions as func
from fastapi import FastAPI, Request, Response

fast_app = FastAPI()

@fast_app.get("/return_http_no_body")
async def return_http_no_body():
    return Response(content="", media_type="text/plain")

app = func.AsgiFunctionApp(app=fast_app,
                           http_auth_level=func.AuthLevel.ANONYMOUS)
```

Scaling and performance

For scaling and performance best practices for Python function apps, see the [Python scaling and performance](#) article.

Context

To get the invocation context of a function when it's running, include the `context` argument in its signature.

For example:

Python

```
import azure.functions

def main(req: azure.functions.HttpRequest,
        context: azure.functions.Context) -> str:
    return f'{context.invocation_id}'
```

The [Context](#) class has the following string attributes:

[+] [Expand table](#)

Attribute	Description
<code>function_directory</code>	The directory in which the function is running.
<code>function_name</code>	The name of the function.
<code>invocation_id</code>	The ID of the current function invocation.
<code>thread_local_storage</code>	The thread local storage of the function. Contains a local <code>invocation_id</code> for logging from created threads .
<code>trace_context</code>	The context for distributed tracing. For more information, see Trace Context .
<code>retry_context</code>	The context for retries to the function. For more information, see retry-policies .

Global variables

It isn't guaranteed that the state of your app will be preserved for future executions. However, the Azure Functions runtime often reuses the same process for multiple executions of the same app. To cache the results of an expensive computation, declare it as a global variable.

Python

```
CACHED_DATA = None

def main(req):
    global CACHED_DATA
    if CACHED_DATA is None:
        CACHED_DATA = load_json()

    # ... use CACHED_DATA in code
```

Environment variables

In Azure Functions, [application settings](#), such as service connection strings, are exposed as environment variables when they're running. There are two main ways to access these settings in your code.

Method	Description
<code>os.environ["myAppSetting"]</code>	Tries to get the application setting by key name, and raises an error when it's unsuccessful.
<code>os.getenv("myAppSetting")</code>	Tries to get the application setting by key name, and returns <code>null</code> when it's unsuccessful.

Both of these ways require you to declare `import os`.

The following example uses `os.environ["myAppSetting"]` to get the [application setting](#), with the key named `myAppSetting`:

Python

```
import logging
import os

import azure.functions as func

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req: func.HttpRequest) -> func.HttpResponse:
    # Get the setting named 'myAppSetting'
    my_app_setting_value = os.environ["myAppSetting"]
    logging.info(f'My app setting value:{my_app_setting_value}')
```

For local development, application settings are [maintained in the `local.settings.json` file](#).

When you're using the new programming model, enable the following app setting in the `local.settings.json` file, as shown here:

JSON

```
"AzureWebJobsFeatureFlags": "EnableWorkerIndexing"
```

When you're deploying the function, this setting isn't created automatically. You must explicitly create this setting in your function app in Azure for it to run by using the v2 model.

Python version

Azure Functions supports the following Python versions:

[+] Expand table

Functions version	Python* versions
4.x	3.11 3.10 3.9 3.8 3.7
3.x	3.9 3.8 3.7

* Official Python distributions

To request a specific Python version when you create your function app in Azure, use the `--runtime-version` option of the [az functionapp create](#) command. The Functions runtime version is set by the `--functions-version` option. The Python version is set when the function app is created, and it can't be changed for apps running in a Consumption plan.

The runtime uses the available Python version when you run it locally.

Changing Python version

To set a Python function app to a specific language version, you need to specify the language and the version of the language in the `LinuxFxVersion` field in the site configuration. For example, to change the Python app to use Python 3.8, set `linuxFxVersion` to `python|3.8`.

To learn how to view and change the `linuxFxVersion` site setting, see [How to target Azure Functions runtime versions](#).

For more general information, see the [Azure Functions runtime support policy](#) and [Supported languages in Azure Functions](#).

Package management

When you're developing locally by using Core Tools or Visual Studio Code, add the names and versions of the required packages to the `requirements.txt` file, and then install them by using `pip`.

For example, you can use the following `requirements.txt` file and `pip` command to install the `requests` package from PyPI.

```
txt
```

```
requests==2.19.1
```

```
Bash
```

```
pip install -r requirements.txt
```

When running your functions in an [App Service plan](#), dependencies that you define in `requirements.txt` are given precedence over built-in Python modules, such as `logging`. This precedence can cause conflicts when built-in modules have the same names as directories in your code. When running in a [Consumption plan](#) or an [Elastic Premium plan](#), conflicts are less likely because your dependencies aren't prioritized by default.

To prevent issues running in an App Service plan, don't name your directories the same as any Python native modules and don't include Python native libraries in your project's `requirements.txt` file.

Publishing to Azure

When you're ready to publish, make sure that all your publicly available dependencies are listed in the `requirements.txt` file. You can locate this file at the root of your project directory.

You can find the project files and folders that are excluded from publishing, including the virtual environment folder, in the root directory of your project.

There are three build actions supported for publishing your Python project to Azure: remote build, local build, and builds using custom dependencies.

You can also use Azure Pipelines to build your dependencies and publish by using continuous delivery (CD). To learn more, see [Continuous delivery with Azure Pipelines](#).

Remote build

When you use remote build, dependencies that are restored on the server and native dependencies match the production environment. This results in a smaller deployment package to upload. Use remote build when you're developing Python apps on Windows.

If your project has custom dependencies, you can [use remote build with extra index URL](#).

Dependencies are obtained remotely based on the contents of the *requirements.txt* file. [Remote build](#) is the recommended build method. By default, Core Tools requests a remote build when you use the following [func azure functionapp publish](#) command to publish your Python project to Azure.

```
Bash
```

```
func azure functionapp publish <APP_NAME>
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

The [Azure Functions Extension for Visual Studio Code](#) also requests a remote build by default.

Local build

Dependencies are obtained locally based on the contents of the *requirements.txt* file. You can prevent doing a remote build by using the following [func azure functionapp publish](#) command to publish with a local build:

```
command
```

```
func azure functionapp publish <APP_NAME> --build local
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

When you use the `--build local` option, project dependencies are read from the *requirements.txt* file, and those dependent packages are downloaded and installed locally. Project files and dependencies are deployed from your local computer to Azure. This results in a larger deployment package being uploaded to Azure. If for some reason you can't get the *requirements.txt* file by using Core Tools, you must use the custom dependencies option for publishing.

We don't recommend using local builds when you're developing locally on Windows.

Custom dependencies

When your project has dependencies that aren't found in the [Python Package Index](#), there are two ways to build the project. The first way, the *build* method, depends on

how you build the project.

Remote build with extra index URL

When your packages are available from an accessible custom package index, use a remote build. Before you publish, be sure to [create an app setting](#) named `PIP_EXTRA_INDEX_URL`. The value for this setting is the URL of your custom package index. Using this setting tells the remote build to run `pip install` by using the `--extra-index-url` option. To learn more, see the [Python pip install documentation](#).

You can also use basic authentication credentials with your extra package index URLs. To learn more, see [Basic authentication credentials](#) in the Python documentation.

Install local packages

If your project uses packages that aren't publicly available to our tools, you can make them available to your app by putting them in the `__app__/python_packages` directory. Before you publish, run the following command to install the dependencies locally:

command

```
pip install --target="<PROJECT_DIR>/python_packages/lib/site-packages" -r requirements.txt
```

When you're using custom dependencies, you should use the `--no-build` publishing option, because you've already installed the dependencies into the project folder.

command

```
func azure functionapp publish <APP_NAME> --no-build
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

Unit testing

Functions that are written in Python can be tested like other Python code by using standard testing frameworks. For most bindings, it's possible to create a mock input object by creating an instance of an appropriate class from the `azure.functions` package. Since the [azure.functions](#) package isn't immediately available, be sure to install it via your `requirements.txt` file as described in the [package management](#) section above.

With `my_second_function` as an example, the following is a mock test of an HTTP-triggered function:

First, create the `<project_root>/function_app.py` file and implement the `my_second_function` function as the HTTP trigger and `shared_code.my_second_helper_function`.

Python

```
# <project_root>/function_app.py
import azure.functions as func
import logging

# Use absolute import to resolve shared_code modules
from shared_code import my_second_helper_function

app = func.FunctionApp()

# Define the HTTP trigger that accepts the ?value=<int> query parameter
# Double the value and return the result in HttpResponseMessage
@app.function_name(name="my_second_function")
@app.route(route="hello")
def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Executing my_second_function.')

    initial_value: int = int(req.params.get('value'))
    doubled_value: int = my_second_helper_function.double(initial_value)

    return func.HttpResponse(
        body=f'{initial_value} * 2 = {doubled_value}',
        status_code=200
    )
```

Python

```
# <project_root>/shared_code/__init__.py
# Empty __init__.py file marks shared_code folder as a Python package
```

Python

```
# <project_root>/shared_code/my_second_helper_function.py

def double(value: int) -> int:
    return value * 2
```

You can start writing test cases for your HTTP trigger.

Python

```
# <project_root>/tests/test_my_second_function.py
import unittest
import azure.functions as func

from function_app import main

class TestFunction(unittest.TestCase):
    def test_my_second_function(self):
        # Construct a mock HTTP request.
        req = func.HttpRequest(method='GET',
                               body=None,
                               url='/api/my_second_function',
                               params={'value': '21'})

        # Call the function.
        func_call = main.build().get_user_function()
        resp = func_call(req)
        # Check the output.
        self.assertEqual(
            resp.get_body(),
            b'21 * 2 = 42',
        )
    
```

Inside your `.venv` Python virtual environment folder, install your favorite Python test framework, such as `pip install pytest`. Then run `pytest tests` to check the test result.

Temporary files

The `tempfile.gettempdir()` method returns a temporary folder, which on Linux is `/tmp`. Your application can use this directory to store temporary files that are generated and used by your functions when they're running.

ⓘ Important

Files written to the temporary directory aren't guaranteed to persist across invocations. During scale out, temporary files aren't shared between instances.

The following example creates a named temporary file in the temporary directory (`/tmp`):

Python

```
import logging
import azure.functions as func
import tempfile

from os import listdir
```

```
---  
tempFilePath = tempfile.gettempdir()  
fp = tempfile.NamedTemporaryFile()  
fp.write(b'Hello world!')  
filesDirListInTemp = listdir(tempFilePath)
```

We recommend that you maintain your tests in a folder that's separate from the project folder. This action keeps you from deploying test code with your app.

Preinstalled libraries

A few libraries come with the Python functions runtime.

The Python standard library

The Python standard library contains a list of built-in Python modules that are shipped with each Python distribution. Most of these libraries help you access system functionality, such as file input/output (I/O). On Windows systems, these libraries are installed with Python. On Unix-based systems, they're provided by package collections.

To view the library for your Python version, go to:

- [Python 3.8 standard library ↗](#)
- [Python 3.9 standard library ↗](#)
- [Python 3.10 standard library ↗](#)
- [Python 3.11 standard library ↗](#)

Azure Functions Python worker dependencies

The Azure Functions Python worker requires a specific set of libraries. You can also use these libraries in your functions, but they aren't a part of the Python standard. If your functions rely on any of these libraries, they might be unavailable to your code when it's running outside of Azure Functions.

Note

If your function app's *requirements.txt* file contains an `azure-functions-worker` entry, remove it. The functions worker is automatically managed by the Azure Functions platform, and we regularly update it with new features and bug fixes. Manually installing an old version of worker in the *requirements.txt* file might cause unexpected issues.

(!) Note

If your package contains certain libraries that might collide with worker's dependencies (for example, protobuf, tensorflow, or grpcio), configure **PYTHON_ISOLATE_WORKER_DEPENDENCIES** to 1 in app settings to prevent your application from referring to worker's dependencies.

The Azure Functions Python library

Every Python worker update includes a new version of the [Azure Functions Python library \(azure.functions\)](#). This approach makes it easier to continuously update your Python function apps, because each update is backwards-compatible. For a list of releases of this library, go to [azure-functions PyPi](#).

The runtime library version is fixed by Azure, and it can't be overridden by *requirements.txt*. The `azure-functions` entry in *requirements.txt* is only for linting and customer awareness.

Use the following code to track the actual version of the Python functions library in your runtime:

Python

```
getattr(azure.functions, '__version__', '< 1.2.1')
```

Runtime system libraries

For a list of preinstalled system libraries in Python worker Docker images, see the following:

[+] Expand table

Functions runtime	Debian version	Python versions
Version 3.x	Buster	Python 3.7 Python 3.8 Python 3.9

Python worker extensions

The Python worker process that runs in Azure Functions lets you integrate third-party libraries into your function app. These extension libraries act as middleware that can inject specific operations during the lifecycle of your function's execution.

Extensions are imported in your function code much like a standard Python library module. Extensions are run based on the following scopes:

[+] [Expand table](#)

Scope	Description
Application-level	When imported into any function trigger, the extension applies to every function execution in the app.
Function-level	Execution is limited to only the specific function trigger into which it's imported.

Review the information for each extension to learn more about the scope in which the extension runs.

Extensions implement a Python worker extension interface. This action lets the Python worker process call into the extension code during the function's execution lifecycle. To learn more, see [Create extensions](#).

Using extensions

You can use a Python worker extension library in your Python functions by doing the following:

1. Add the extension package in the *requirements.txt* file for your project.
2. Install the library into your app.
3. Add the following application settings:
 - Locally: Enter `"PYTHON_ENABLE_WORKER_EXTENSIONS": "1"` in the `Values` section of your [*local.settings.json* file](#).
 - Azure: Enter `PYTHON_ENABLE_WORKER_EXTENSIONS=1` in your [app settings](#).
4. Import the extension module into your function trigger.
5. Configure the extension instance, if needed. Configuration requirements should be called out in the extension's documentation.

Important

Third-party Python worker extension libraries aren't supported or warrantied by Microsoft. You must make sure that any extensions that you use in your function

app is trustworthy, and you bear the full risk of using a malicious or poorly written extension.

Third-parties should provide specific documentation on how to install and consume their extensions in your function app. For a basic example of how to consume an extension, see [Consuming your extension](#).

Here are examples of using extensions in a function app, by scope:

Application-level

Python

```
# <project_root>/requirements.txt
application-level-extension==1.0.0
```

Python

```
# <project_root>/Trigger/__init__.py

from application_level_extension import AppExtension
AppExtension.configure(key=value)

def main(req, context):
    # Use context.app_ext_attributes here
```

Creating extensions

Extensions are created by third-party library developers who have created functionality that can be integrated into Azure Functions. An extension developer designs, implements, and releases Python packages that contain custom logic designed specifically to be run in the context of function execution. These extensions can be published either to the PyPI registry or to GitHub repositories.

To learn how to create, package, publish, and consume a Python worker extension package, see [Develop Python worker extensions for Azure Functions](#).

Application-level extensions

An extension that's inherited from [AppExtensionBase](#) runs in an *application* scope.

`AppExtensionBase` exposes the following abstract class methods for you to implement:

[\[\] Expand table](#)

Method	Description
<code>init</code>	Called after the extension is imported.
<code>configure</code>	Called from function code when it's needed to configure the extension.
<code>post_function_load_app_level</code>	Called right after the function is loaded. The function name and function directory are passed to the extension. Keep in mind that the function directory is read-only, and any attempt to write to a local file in this directory fails.
<code>pre_invocation_app_level</code>	Called right before the function is triggered. The function context and function invocation arguments are passed to the extension. You can usually pass other attributes in the context object for the function code to consume.
<code>post_invocation_app_level</code>	Called right after the function execution finishes. The function context, function invocation arguments, and invocation return object are passed to the extension. This implementation is a good place to validate whether execution of the lifecycle hooks succeeded.

Function-level extensions

An extension that inherits from [FuncExtensionBase](#) runs in a specific function trigger.

`FuncExtensionBase` exposes the following abstract class methods for implementations:

[\[\] Expand table](#)

Method	Description
<code>__init__</code>	The constructor of the extension. It's called when an extension instance is initialized in a specific function. When you're implementing this abstract method, you might want to accept a <code>filename</code> parameter and pass it to the parent's method <code>super().__init__(filename)</code> for proper extension registration.
<code>post_function_load</code>	Called right after the function is loaded. The function name and function directory are passed to the extension. Keep in mind that the function directory is read-only, and any attempt to write to a local file in this directory fails.
<code>pre_invocation</code>	Called right before the function is triggered. The function context and function invocation arguments are passed to the extension. You can usually

Method	Description
	pass other attributes in the context object for the function code to consume.
<code>post_invocation</code>	Called right after the function execution finishes. The function context, function invocation arguments, and invocation return object are passed to the extension. This implementation is a good place to validate whether execution of the lifecycle hooks succeeded.

Cross-origin resource sharing

Azure Functions supports cross-origin resource sharing (CORS). CORS is configured [in the portal](#) and through the [Azure CLI](#). The CORS allowed origins list applies at the function app level. With CORS enabled, responses include the `Access-Control-Allow-Origin` header. For more information, see [Cross-origin resource sharing](#).

Cross-origin resource sharing (CORS) is fully supported for Python function apps.

Async

By default, a host instance for Python can process only one function invocation at a time. This is because Python is a single-threaded runtime. For a function app that processes a large number of I/O events or is being I/O bound, you can significantly improve performance by running functions asynchronously. For more information, see [Improve throughout performance of Python apps in Azure Functions](#).

Shared memory (preview)

To improve throughput, Azure Functions lets your out-of-process Python language worker share memory with the Functions host process. When your function app is hitting bottlenecks, you can enable shared memory by adding an application setting named `FUNCTIONS_WORKER_SHARED_MEMORY_DATA_TRANSFER_ENABLED` with a value of `1`. With shared memory enabled, you can then use the `DOCKER_SHM_SIZE` setting to set the shared memory to something like `268435456`, which is equivalent to 256 MB.

For example, you might enable shared memory to reduce bottlenecks when you're using Blob Storage bindings to transfer payloads larger than 1 MB.

This functionality is available only for function apps that are running in Premium and Dedicated (Azure App Service) plans. To learn more, see [Shared memory](#).

Known issues and FAQ

Here are two troubleshooting guides for common issues:

- [ModuleNotFoundError and ImportError](#)
- [Can't import 'cygrpc'](#)

Here are two troubleshooting guides for known issues with the v2 programming model:

- [Couldn't load file or assembly](#)
- [Unable to resolve the Azure Storage connection named Storage](#)

All known issues and feature requests are tracked in a [GitHub issues list ↗](#). If you run into a problem and can't find the issue in GitHub, open a new issue, and include a detailed description of the problem.

Next steps

For more information, see the following resources:

- [Azure Functions package API documentation](#)
- [Best practices for Azure Functions](#)
- [Azure Functions triggers and bindings](#)
- [Blob Storage bindings](#)
- [HTTP and webhook bindings](#)
- [Queue Storage bindings](#)
- [Timer triggers](#)

Having issues with using Python? Tell us what's going on. ↗

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Functions Python developer guide

Article • 07/30/2024

This guide is an introduction to developing Azure Functions by using Python. The article assumes that you've already read the [Azure Functions developers guide](#).

Important

This article supports both the v1 and v2 programming model for Python in Azure Functions. The Python v1 model uses a `functions.json` file to define functions, and the new v2 model lets you instead use a decorator-based approach. This new approach results in a simpler file structure, and it's more code-centric. Choose the **v2** selector at the top of the article to learn about this new programming model.

As a Python developer, you might also be interested in these topics:

Get started

- [Visual Studio Code](#): Create your first Python app using Visual Studio Code.
- [Terminal or command prompt](#): Create your first Python app from the command prompt using Azure Functions Core Tools.
- [Samples](#): Review some existing Python apps in the Learn samples browser.

Development options

Both Python Functions programming models support local development in one of the following environments:

Python v2 programming model:

- [Visual Studio Code](#)
- [Terminal or command prompt](#)

Python v1 programming model:

- [Visual Studio Code](#)
- [Terminal or command prompt](#)

You can also create Python v1 functions in the Azure portal.

Tip

Although you can develop your Python-based Azure functions locally on Windows, Python is supported only on a Linux-based hosting plan when it's running in Azure. For more information, see the [list of supported operating system/runtime combinations](#).

Programming model

Azure Functions expects a function to be a stateless method in your Python script that processes input and produces output. By default, the runtime expects the method to be implemented as a global method in the *function_app.py* file.

Triggers and bindings can be declared and used in a function in a decorator based approach. They're defined in the same file, *function_app.py*, as the functions. As an example, the following *function_app.py* file represents a function trigger by an HTTP request.

Python

```
@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req):
    user = req.params.get("user")
    return f"Hello, {user}!"
```

You can also explicitly declare the attribute types and return type in the function by using Python type annotations. Doing so helps you use the IntelliSense and autocomplete features that are provided by many Python code editors.

Python

```
import azure.functions as func

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req: func.HttpRequest) -> str:
    user = req.params.get("user")
    return f"Hello, {user}!"
```

To learn about known limitations with the v2 model and their workarounds, see [Troubleshoot Python errors in Azure Functions](#).

Alternative entry point

The entry point is only in the *function_app.py* file. However, you can reference functions within the project in *function_app.py* by using [blueprints](#) or by importing.

Folder structure

The recommended folder structure for a Python functions project looks like the following example:

```
Windows Command Prompt

<project_root>/
| - .venv/
| - .vscode/
| - function_app.py
| - additional_functions.py
| - tests/
| | - test_my_function.py
| - .funcignore
| - host.json
| - local.settings.json
| - requirements.txt
| - Dockerfile
```

The main project folder, *<project_root>*, can contain the following files:

- *.venv/*: (Optional) Contains a Python virtual environment that's used by local development.
- *.vscode/*: (Optional) Contains the stored Visual Studio Code configuration. To learn more, see [Visual Studio Code settings](#).
- *function_app.py*: The default location for all functions and their related triggers and bindings.
- *additional_functions.py*: (Optional) Any other Python files that contain functions (usually for logical grouping) that are referenced in *function_app.py* through blueprints.
- *tests/*: (Optional) Contains the test cases of your function app.
- *.funcignore*: (Optional) Declares files that shouldn't get published to Azure. Usually, this file contains *.vscode/* to ignore your editor setting, *.venv/* to ignore local

Python virtual environment, `tests/` to ignore test cases, and `local.settings.json` to prevent local app settings being published.

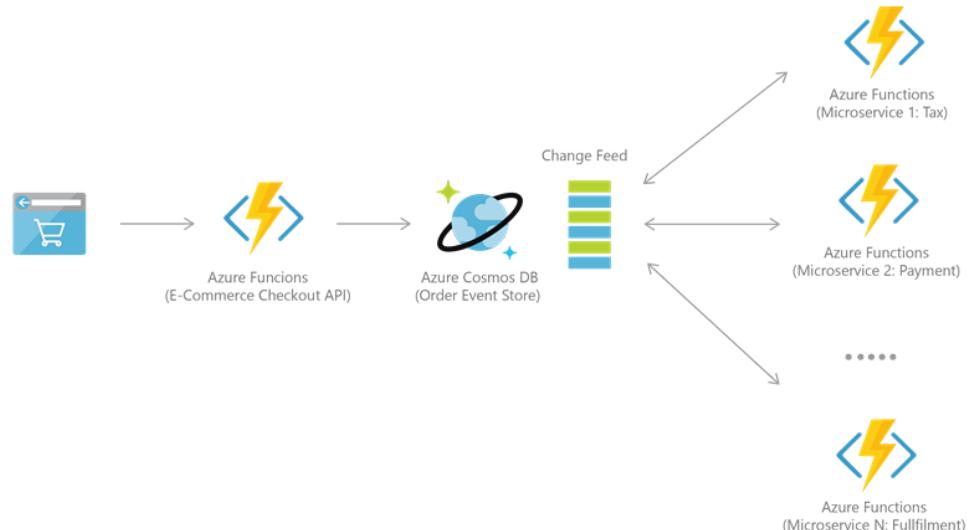
- `host.json`: Contains configuration options that affect all functions in a function app instance. This file does get published to Azure. Not all options are supported when running locally. To learn more, see [host.json](#).
- `local.settings.json`: Used to store app settings and connection strings when it's running locally. This file doesn't get published to Azure. To learn more, see [local.settings.json](#).
- `requirements.txt`: Contains the list of Python packages the system installs when it publishes to Azure.
- `Dockerfile`: (Optional) Used when publishing your project in a [custom container](#).

When you deploy your project to a function app in Azure, the entire contents of the main project folder, `<project_root>`, should be included in the package, but not the folder itself, which means that `host.json` should be in the package root. We recommend that you maintain your tests in a folder along with other functions (in this example, `tests/`). For more information, see [Unit testing](#).

Connect to a database

Azure Functions integrates well with [Azure Cosmos DB](#) for many [use cases](#), including IoT, ecommerce, gaming, etc.

For example, for [event sourcing](#), the two services are integrated to power event-driven architectures using Azure Cosmos DB's [change feed](#) functionality. The change feed provides downstream microservices the ability to reliably and incrementally read inserts and updates (for example, order events). This functionality can be used to provide a persistent event store as a message broker for state-changing events and drive order processing workflow between many microservices (which can be implemented as [serverless Azure Functions](#)).



To connect to Azure Cosmos DB, first [create an account, database, and container](#). Then you can connect your function code to Azure Cosmos DB using [trigger and bindings](#), like this [example](#).

To implement more complex app logic, you can also use the Python library for Cosmos DB. An asynchronous I/O implementation looks like this:

Python

```

pip install azure-cosmos
pip install aiohttp

from azure.cosmos.aio import CosmosClient
from azure.cosmos import exceptions
from azure.cosmos.partition_key import PartitionKey
import asyncio

# Replace these values with your Cosmos DB connection information
endpoint = "https://azure-cosmos-nosql.documents.azure.com:443/"
key = "master_key"
database_id = "cosmicwerx"
container_id = "cosmiccontainer"
partition_key = "/partition_key"

# Set the total throughput (RU/s) for the database and container
database_throughput = 1000

# Singleton CosmosClient instance
client = CosmosClient(endpoint, credential=key)

# Helper function to get or create database and container
async def get_or_create_container(client, database_id, container_id,
partition_key):

```

```

database = await client.create_database_if_not_exists(id=database_id)
print(f'Database "{database_id}" created or retrieved successfully.')

        container = await
database.create_container_if_not_exists(id=container_id,
partition_key=PartitionKey(path=partition_key))
        print(f'Container with id "{container_id}" created')

    return container

async def create_products():
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    for i in range(10):
        await container.upsert_item({
            'id': f'item{i}',
            'productName': 'Widget',
            'productModel': f'Model {i}'
        })

async def get_products():
    items = []
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    async for item in container.read_all_items():
        items.append(item)
    return items

async def query_products(product_name):
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    query = f"SELECT * FROM c WHERE c.productName = '{product_name}'"
    items = []
    async for item in container.query_items(query=query,
enable_cross_partition_query=True):
        items.append(item)
    return items

async def main():
    await create_products()
    all_products = await get_products()
    print('All Products:', all_products)

    queried_products = await query_products('Widget')
    print('Queried Products:', queried_products)

if __name__ == "__main__":
    asyncio.run(main())

```

Blueprints

The Python v2 programming model introduces the concept of *blueprints*. A blueprint is a new class that's instantiated to register functions outside of the core function application. The functions registered in blueprint instances aren't indexed directly by the function runtime. To get these blueprint functions indexed, the function app needs to register the functions from blueprint instances.

Using blueprints provides the following benefits:

- Lets you break up the function app into modular components, which enables you to define functions in multiple Python files and divide them into different components per file.
- Provides extensible public function app interfaces to build and reuse your own APIs.

The following example shows how to use blueprints:

First, in an *http_blueprint.py* file, an HTTP-triggered function is first defined and added to a blueprint object.

Python

```
import logging

import azure.functions as func

bp = func.Blueprint()

@bp.route(route="default_template")
def default_template(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(
            f"Hello, {name}. This HTTP-triggered function "
            f"executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP-triggered function executed successfully. "
            "Pass a name in the query string or in the request body for a"
            " personalized response.",
```

```
    status_code=200
)
```

Next, in the `function_app.py` file, the `blueprint` object is imported and its functions are registered to the function app.

Python

```
import azure.functions as func
from http_blueprint import bp

app = func.FunctionApp()

app.register_functions(bp)
```

ⓘ Note

Durable Functions also supports blueprints. To create blueprints for Durable Functions apps, register your orchestration, activity, and entity triggers and client bindings using the [azure-functions-durable](#) Blueprint class, as shown [here](#). The resulting blueprint can then be registered as normal. See our [sample](#) for an example.

Triggers and inputs

Inputs are divided into two categories in Azure Functions: trigger input and other input. Although they're defined using different decorators, their usage is similar in Python code. Connection strings or secrets for trigger and input sources map to values in the `local.settings.json` file when they're running locally, and they map to the application settings when they're running in Azure.

As an example, the following code demonstrates how to define a Blob Storage input binding:

JSON

```
// local.settings.json
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "STORAGE_CONNECTION_STRING": "<AZURE_STORAGE_CONNECTION_STRING>",
    "AzureWebJobsStorage": "<azure-storage-connection-string>",
    "AzureWebJobsFeatureFlags": "EnableWorkerIndexing"
```

```
}
```

Python

```
# function_app.py
import azure.functions as func
import logging

app = func.FunctionApp()

@app.route(route="req")
@app.read_blob(arg_name="obj", path="samples/{id}",
               connection="STORAGE_CONNECTION_STRING")
def main(req: func.HttpRequest, obj: func.InputStream):
    logging.info(f'Python HTTP-triggered function processed: {obj.read()}')
```

When the function is invoked, the HTTP request is passed to the function as `req`. An entry is retrieved from the Azure Blob Storage account based on the *ID* in the route URL and made available as `obj` in the function body. Here, the specified storage account is the connection string that's found in the `STORAGE_CONNECTION_STRING` app setting.

For data intensive binding operations, you may want to use a separate storage account. For more information, see [Storage account guidance](#).

SDK type bindings (preview)

For select triggers and bindings, you can work with data types implemented by the underlying Azure SDKs and frameworks. These *SDK type bindings* let you interact with binding data as if you were using the underlying service SDK.

Functions supports Python SDK type bindings for Azure Blob storage, which lets you work with blob data using the underlying `BlobClient` type.

ⓘ Important

SDK type bindings support for Python is currently in preview:

- You must use the Python v2 programming model.
- Currently, only synchronous SDK types are supported.

Prerequisites

- Azure Functions runtime version version 4.34, or a later version.
- Python ↗ version 3.9, or a later supported version.

Enable SDK type bindings for the Blob storage extension

1. Add the `azurefunctions-extensions-bindings-blob` extension package to the `requirements.txt` file in the project, which should include at least these packages:

```
text  
  
azure-functions  
azurefunctions-extensions-bindings-blob
```

2. Add this code to the `function_app.py` file in the project, which imports the SDK type bindings:

```
Python  
  
import azurefunctions.extensions.bindings.blob as blob
```

SDK type bindings examples

This example shows how to get the `BlobClient` from both a Blob storage trigger (`blob_trigger`) and from the input binding on an HTTP trigger (`blob_input`):

```
Python  
  
import logging  
  
import azure.functions as func  
import azurefunctions.extensions.bindings.blob as blob  
  
app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)  
  
@app.blob_trigger(  
    arg_name="client", path="PATH/TO/BLOB", connection="AzureWebJobsStorage"  
)  
def blob_trigger(client: blob.BlobClient):  
    logging.info(  
        f"Python blob trigger function processed blob \n"  
        f"Properties: {client.get_blob_properties()}\n"  
        f"Blob content head: {client.download_blob().read(size=1)}"  
    )  
  
@app.route(route="file")
```

```
@app.blob_input(  
    arg_name="client", path="PATH/TO/BLOB", connection="AzureWebJobsStorage"  
)  
def blob_input(req: func.HttpRequest, client: blob.BlobClient):  
    logging.info(  
        f"Python blob input function processed blob \n"  
        f"Properties: {client.get_blob_properties()}\n"  
        f"Blob content head: {client.download_blob().read(size=1)}"  
    )  
    return "ok"
```

You can view other SDK type bindings samples for Blob storage in the Python extensions repository:

- [ContainerClient type ↗](#)
- [StorageStreamDownloader type ↗](#)

HTTP streams (preview)

HTTP streams lets you accept and return data from your HTTP endpoints using FastAPI request and response APIs enabled in your functions. These APIs lets the host process large data in HTTP messages as chunks instead of reading an entire message into memory.

This feature makes it possible to handle large data stream, OpenAI integrations, deliver dynamic content, and support other core HTTP scenarios requiring real-time interactions over HTTP. You can also use FastAPI response types with HTTP streams. Without HTTP streams, the size of your HTTP requests and responses are limited by memory restrictions that can be encountered when processing entire message payloads all in memory.

ⓘ Important

HTTP streams support for Python is currently in preview and requires you to use the Python v2 programming model.

Prerequisites

- [Azure Functions runtime](#) version 4.34.1, or a later version.
- [Python ↗](#) version 3.8, or a later [supported version](#).

Enable HTTP streams

HTTP streams are disabled by default. You need to enable this feature in your application settings and also update your code to use the FastAPI package. Note that when enabling HTTP streams, the function app will default to using HTTP streaming, and the original HTTP functionality will not work.

1. Add the `azurefunctions-extensions-http-fastapi` extension package to the `requirements.txt` file in the project, which should include at least these packages:

```
text  
  
azure-functions  
azurefunctions-extensions-http-fastapi
```

2. Add this code to the `function_app.py` file in the project, which imports the FastAPI extension:

```
Python  
  
from azurefunctions.extensions.http.fastapi import Request,  
StreamingResponse
```

3. When you deploy to Azure, add the following [application setting](#) in your function app:

```
"PYTHON_ENABLE_INIT_INDEXING": "1"
```

If you are deploying to Linux Consumption, also add

```
"PYTHON_ISOLATE_WORKER_DEPENDENCIES": "1"
```

When running locally, you also need to add these same settings to the `local.settings.json` project file.

HTTP streams examples

After you enable the HTTP streaming feature, you can create functions that stream data over HTTP.

This example is an HTTP triggered function that streams HTTP response data. You might use these capabilities to support scenarios like sending event data through a pipeline for real time visualization or detecting anomalies in large sets of data and providing instant notifications.

```
Python
```

```

import time

import azure.functions as func
from azurefunctions.extensions.http.fastapi import Request,
StreamingResponse

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

def generate_sensor_data():
    """Generate real-time sensor data."""
    for i in range(10):
        # Simulate temperature and humidity readings
        temperature = 20 + i
        humidity = 50 + i
        yield f"data: {{'temperature': {temperature}, 'humidity': {humidity}}}\n\n"
    time.sleep(1)

@app.route(route="stream", methods=[func.HttpMethod.GET])
async def stream_sensor_data(req: Request) -> StreamingResponse:
    """Endpoint to stream real-time sensor data."""
    return StreamingResponse(generate_sensor_data(), media_type="text/event-stream")

```

This example is an HTTP triggered function that receives and processes streaming data from a client in real time. It demonstrates streaming upload capabilities that can be helpful for scenarios like processing continuous data streams and handling event data from IoT devices.

Python

```

import azure.functions as func
from azurefunctions.extensions.http.fastapi import JSONResponse, Request

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="streaming_upload", methods=[func.HttpMethod.POST])
async def streaming_upload(req: Request) -> JSONResponse:
    """Handle streaming upload requests."""
    # Process each chunk of data as it arrives
    async for chunk in req.stream():
        process_data_chunk(chunk)

    # Once all data is received, return a JSON response indicating
    # successful processing
    return JSONResponse({"status": "Data uploaded and processed
successfully"})

```

```
def process_data_chunk(chunk: bytes):
    """Process each data chunk."""
    # Add custom processing logic here
    pass
```

Calling HTTP streams

You must use an HTTP client library to make streaming calls to a function's FastAPI endpoints. The client tool or browser you're using might not natively support streaming or could only return the first chunk of data.

You can use a client script like this to send streaming data to an HTTP endpoint:

Python

```
import httpx # Be sure to add 'httpx' to 'requirements.txt'
import asyncio

async def stream_generator(file_path):
    chunk_size = 2 * 1024 # Define your own chunk size
    with open(file_path, 'rb') as file:
        while chunk := file.read(chunk_size):
            yield chunk
            print(f"Sent chunk: {len(chunk)} bytes")

async def stream_to_server(url, file_path):
    timeout = httpx.Timeout(60.0, connect=60.0)
    async with httpx.AsyncClient(timeout=timeout) as client:
        response = await client.post(url,
content=stream_generator(file_path))
    return response

async def stream_response(response):
    if response.status_code == 200:
        async for chunk in response.aiter_raw():
            print(f"Received chunk: {len(chunk)} bytes")
    else:
        print(f"Error: {response}")

async def main():
    print('helloworld')
    # Customize your streaming endpoint served from core tool in variable
    'url' if different.
    url = 'http://localhost:7071/api/streaming_upload'
    file_path = r'<file path>'

    response = await stream_to_server(url, file_path)
    print(response)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Outputs

Output can be expressed both in return value and output parameters. If there's only one output, we recommend using the return value. For multiple outputs, you'll have to use output parameters.

To produce multiple outputs, use the `set()` method provided by the `azure.functions.Out` interface to assign a value to the binding. For example, the following function can push a message to a queue and also return an HTTP response.

Python

```
# function_app.py
import azure.functions as func

app = func.FunctionApp()

@app.write_blob(arg_name="msg", path="output-container/{name}",
                connection="CONNECTION_STRING")
def test_function(req: func.HttpRequest,
                  msg: func.Out[str]) -> str:

    message = req.params.get('body')
    msg.set(message)
    return message
```

Logging

Access to the Azure Functions runtime logger is available via a root [logging](#) handler in your function app. This logger is tied to Application Insights and allows you to flag warnings and errors that occur during the function execution.

The following example logs an info message when the function is invoked via an HTTP trigger.

Python

```
import logging

def main(req):
    logging.info('Python HTTP trigger function processed a request.')
```

More logging methods are available that let you write to the console at different trace levels:

[+] Expand table

Method	Description
<code>critical(_message_)</code>	Writes a message with level CRITICAL on the root logger.
<code>error(_message_)</code>	Writes a message with level ERROR on the root logger.
<code>warning(_message_)</code>	Writes a message with level WARNING on the root logger.
<code>info(_message_)</code>	Writes a message with level INFO on the root logger.
<code>debug(_message_)</code>	Writes a message with level DEBUG on the root logger.

To learn more about logging, see [Monitor Azure Functions](#).

Logging from created threads

To see logs coming from your created threads, include the `context` argument in the function's signature. This argument contains an attribute `thread_local_storage` that stores a local `invocation_id`. This can be set to the function's current `invocation_id` to ensure the context is changed.

Python

```
import azure.functions as func
import logging
import threading

def main(req, context):
    logging.info('Python HTTP trigger function processed a request.')
    t = threading.Thread(target=log_function, args=(context,))
    t.start()

def log_function(context):
    context.thread_local_storage.invocation_id = context.invocation_id
    logging.info('Logging from thread.')
```

Log custom telemetry

By default, the Functions runtime collects logs and other telemetry data that are generated by your functions. This telemetry ends up as traces in Application Insights. Request and dependency telemetry for certain Azure services are also collected by default by [triggers and bindings](#).

To collect custom request and custom dependency telemetry outside of bindings, you can use the [OpenCensus Python Extensions](#). This extension sends custom telemetry data to your Application Insights instance. You can find a list of supported extensions at the [OpenCensus repository](#).

ⓘ Note

To use the OpenCensus Python extensions, you need to enable [Python worker extensions](#) in your function app by setting `PYTHON_ENABLE_WORKER_EXTENSIONS` to `1`. You also need to switch to using the Application Insights connection string by adding the [APPLICATIONINSIGHTS CONNECTION STRING](#) setting to your [application settings](#), if it's not already there.

text

```
// requirements.txt
...
opencensus-extension-azure-functions
opencensus-ext-requests
```

Python

```
import json
import logging

import requests
from opencensus.extension.azure.functions import OpenCensusExtension
from opencensus.trace import config_integration

config_integration.trace_integrations(['requests'])

OpenCensusExtension.configure()

def main(req, context):
    logging.info('Executing HttpTrigger with OpenCensus extension')

    # You must use contexttracer to create spans
    with context.tracer.span("parent"):
        response = requests.get(url='http://example.com')

    return json.dumps({
        'method': req.method,
```

```
'response': response.status_code,
'ctx_func_name': context.function_name,
'ctx_func_dir': context.function_directory,
'ctx_invocation_id': context.invocation_id,
'ctx_trace_context_Traceparent': context.trace_context.Traceparent,
'ctx_trace_context_Tracestate': context.trace_context.Tracestate,
'ctx_retry_context_RetryCount': context.retry_context.retry_count,
'ctx_retry_context_MaxRetryCount':
context.retry_context.max_retry_count,
})
```

HTTP trigger

The HTTP trigger is defined as a method that takes a named binding parameter, which is an `HttpRequest` object, and returns an `HttpResponse` object. You apply the `function_name` decorator to the method to define the function name, while the HTTP endpoint is set by applying the `route` decorator.

This example is from the HTTP trigger template for the Python v2 programming model, where the binding parameter name is `req`. It's the sample code that's provided when you create a function by using Azure Functions Core Tools or Visual Studio Code.

Python

```
@app.function_name(name="HttpTrigger1")
@app.route(route="hello")
def test_function(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
    else:
        name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello, {name}. This HTTP-triggered
function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP-triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
            status_code=200
        )
```

From the `HttpRequest` object, you can get request headers, query parameters, route parameters, and the message body. In this function, you obtain the value of the `name` query parameter from the `params` parameter of the `HttpRequest` object. You read the JSON-encoded message body by using the `get_json` method.

Likewise, you can set the `status_code` and `headers` for the response message in the returned `HttpResponse` object.

To pass in a name in this example, paste the URL that's provided when you're running the function, and then append it with `?name={name}`.

Web frameworks

You can use Asynchronous Server Gateway Interface (ASGI)-compatible and Web Server Gateway Interface (WSGI)-compatible frameworks, such as Flask and FastAPI, with your HTTP-triggered Python functions. You must first update the `host.json` file to include an HTTP `routePrefix`, as shown in the following example:

```
JSON

{
  "version": "2.0",
  "logging": {
    {
      "applicationInsights": {
        {
          "samplingSettings": {
            {
              "isEnabled": true,
              "excludedTypes": "Request"
            }
          }
        },
        "extensionBundle": {
          {
            "id": "Microsoft.Azure.Functions.ExtensionBundle",
            "version": "[2.*, 3.0.0)"
          },
          "extensions": {
            {
              "http": {
                {
                  "routePrefix": ""
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The framework code looks like the following example:

ASGI

`AsgiFunctionApp` is the top-level function app class for constructing ASGI HTTP functions.

Python

```
# function_app.py

import azure.functions as func
from fastapi import FastAPI, Request, Response

fast_app = FastAPI()

@fast_app.get("/return_http_no_body")
async def return_http_no_body():
    return Response(content="", media_type="text/plain")

app = func.AsgiFunctionApp(app=fast_app,
                           http_auth_level=func.AuthLevel.ANONYMOUS)
```

Scaling and performance

For scaling and performance best practices for Python function apps, see the [Python scaling and performance](#) article.

Context

To get the invocation context of a function when it's running, include the `context` argument in its signature.

For example:

Python

```
import azure.functions

def main(req: azure.functions.HttpRequest,
        context: azure.functions.Context) -> str:
    return f'{context.invocation_id}'
```

The [Context](#) class has the following string attributes:

[\[\] Expand table](#)

Attribute	Description
<code>function_directory</code>	The directory in which the function is running.
<code>function_name</code>	The name of the function.
<code>invocation_id</code>	The ID of the current function invocation.
<code>thread_local_storage</code>	The thread local storage of the function. Contains a local <code>invocation_id</code> for logging from created threads .
<code>trace_context</code>	The context for distributed tracing. For more information, see Trace Context .
<code>retry_context</code>	The context for retries to the function. For more information, see retry-policies .

Global variables

It isn't guaranteed that the state of your app will be preserved for future executions. However, the Azure Functions runtime often reuses the same process for multiple executions of the same app. To cache the results of an expensive computation, declare it as a global variable.

Python

```
CACHED_DATA = None

def main(req):
    global CACHED_DATA
    if CACHED_DATA is None:
        CACHED_DATA = load_json()

    # ... use CACHED_DATA in code
```

Environment variables

In Azure Functions, [application settings](#), such as service connection strings, are exposed as environment variables when they're running. There are two main ways to access these settings in your code.

Method	Description
<code>os.environ["myAppSetting"]</code>	Tries to get the application setting by key name, and raises an error when it's unsuccessful.
<code>os.getenv("myAppSetting")</code>	Tries to get the application setting by key name, and returns <code>null</code> when it's unsuccessful.

Both of these ways require you to declare `import os`.

The following example uses `os.environ["myAppSetting"]` to get the [application setting](#), with the key named `myAppSetting`:

Python

```
import logging
import os

import azure.functions as func

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req: func.HttpRequest) -> func.HttpResponse:
    # Get the setting named 'myAppSetting'
    my_app_setting_value = os.environ["myAppSetting"]
    logging.info(f'My app setting value:{my_app_setting_value}')
```

For local development, application settings are [maintained in the `local.settings.json` file](#).

When you're using the new programming model, enable the following app setting in the `local.settings.json` file, as shown here:

JSON

```
"AzureWebJobsFeatureFlags": "EnableWorkerIndexing"
```

When you're deploying the function, this setting isn't created automatically. You must explicitly create this setting in your function app in Azure for it to run by using the v2 model.

Python version

Azure Functions supports the following Python versions:

[+] Expand table

Functions version	Python* versions
4.x	3.11 3.10 3.9 3.8 3.7
3.x	3.9 3.8 3.7

* Official Python distributions

To request a specific Python version when you create your function app in Azure, use the `--runtime-version` option of the [az functionapp create](#) command. The Functions runtime version is set by the `--functions-version` option. The Python version is set when the function app is created, and it can't be changed for apps running in a Consumption plan.

The runtime uses the available Python version when you run it locally.

Changing Python version

To set a Python function app to a specific language version, you need to specify the language and the version of the language in the `LinuxFxVersion` field in the site configuration. For example, to change the Python app to use Python 3.8, set `linuxFxVersion` to `python|3.8`.

To learn how to view and change the `linuxFxVersion` site setting, see [How to target Azure Functions runtime versions](#).

For more general information, see the [Azure Functions runtime support policy](#) and [Supported languages in Azure Functions](#).

Package management

When you're developing locally by using Core Tools or Visual Studio Code, add the names and versions of the required packages to the `requirements.txt` file, and then install them by using `pip`.

For example, you can use the following `requirements.txt` file and `pip` command to install the `requests` package from PyPI.

```
txt
```

```
requests==2.19.1
```

```
Bash
```

```
pip install -r requirements.txt
```

When running your functions in an [App Service plan](#), dependencies that you define in `requirements.txt` are given precedence over built-in Python modules, such as `logging`. This precedence can cause conflicts when built-in modules have the same names as directories in your code. When running in a [Consumption plan](#) or an [Elastic Premium plan](#), conflicts are less likely because your dependencies aren't prioritized by default.

To prevent issues running in an App Service plan, don't name your directories the same as any Python native modules and don't include Python native libraries in your project's `requirements.txt` file.

Publishing to Azure

When you're ready to publish, make sure that all your publicly available dependencies are listed in the `requirements.txt` file. You can locate this file at the root of your project directory.

You can find the project files and folders that are excluded from publishing, including the virtual environment folder, in the root directory of your project.

There are three build actions supported for publishing your Python project to Azure: remote build, local build, and builds using custom dependencies.

You can also use Azure Pipelines to build your dependencies and publish by using continuous delivery (CD). To learn more, see [Continuous delivery with Azure Pipelines](#).

Remote build

When you use remote build, dependencies that are restored on the server and native dependencies match the production environment. This results in a smaller deployment package to upload. Use remote build when you're developing Python apps on Windows.

If your project has custom dependencies, you can [use remote build with extra index URL](#).

Dependencies are obtained remotely based on the contents of the *requirements.txt* file. [Remote build](#) is the recommended build method. By default, Core Tools requests a remote build when you use the following [func azure functionapp publish](#) command to publish your Python project to Azure.

```
Bash
```

```
func azure functionapp publish <APP_NAME>
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

The [Azure Functions Extension for Visual Studio Code](#) also requests a remote build by default.

Local build

Dependencies are obtained locally based on the contents of the *requirements.txt* file. You can prevent doing a remote build by using the following [func azure functionapp publish](#) command to publish with a local build:

```
command
```

```
func azure functionapp publish <APP_NAME> --build local
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

When you use the `--build local` option, project dependencies are read from the *requirements.txt* file, and those dependent packages are downloaded and installed locally. Project files and dependencies are deployed from your local computer to Azure. This results in a larger deployment package being uploaded to Azure. If for some reason you can't get the *requirements.txt* file by using Core Tools, you must use the custom dependencies option for publishing.

We don't recommend using local builds when you're developing locally on Windows.

Custom dependencies

When your project has dependencies that aren't found in the [Python Package Index](#), there are two ways to build the project. The first way, the *build* method, depends on

how you build the project.

Remote build with extra index URL

When your packages are available from an accessible custom package index, use a remote build. Before you publish, be sure to [create an app setting](#) named `PIP_EXTRA_INDEX_URL`. The value for this setting is the URL of your custom package index. Using this setting tells the remote build to run `pip install` by using the `--extra-index-url` option. To learn more, see the [Python pip install documentation](#).

You can also use basic authentication credentials with your extra package index URLs. To learn more, see [Basic authentication credentials](#) in the Python documentation.

Install local packages

If your project uses packages that aren't publicly available to our tools, you can make them available to your app by putting them in the `__app__/python_packages` directory. Before you publish, run the following command to install the dependencies locally:

command

```
pip install --target="<PROJECT_DIR>/python_packages/lib/site-packages" -r requirements.txt
```

When you're using custom dependencies, you should use the `--no-build` publishing option, because you've already installed the dependencies into the project folder.

command

```
func azure functionapp publish <APP_NAME> --no-build
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

Unit testing

Functions that are written in Python can be tested like other Python code by using standard testing frameworks. For most bindings, it's possible to create a mock input object by creating an instance of an appropriate class from the `azure.functions` package. Since the [azure.functions](#) package isn't immediately available, be sure to install it via your `requirements.txt` file as described in the [package management](#) section above.

With `my_second_function` as an example, the following is a mock test of an HTTP-triggered function:

First, create the `<project_root>/function_app.py` file and implement the `my_second_function` function as the HTTP trigger and `shared_code.my_second_helper_function`.

Python

```
# <project_root>/function_app.py
import azure.functions as func
import logging

# Use absolute import to resolve shared_code modules
from shared_code import my_second_helper_function

app = func.FunctionApp()

# Define the HTTP trigger that accepts the ?value=<int> query parameter
# Double the value and return the result in HttpResponseMessage
@app.function_name(name="my_second_function")
@app.route(route="hello")
def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Executing my_second_function.')

    initial_value: int = int(req.params.get('value'))
    doubled_value: int = my_second_helper_function.double(initial_value)

    return func.HttpResponse(
        body=f'{initial_value} * 2 = {doubled_value}',
        status_code=200
    )
```

Python

```
# <project_root>/shared_code/__init__.py
# Empty __init__.py file marks shared_code folder as a Python package
```

Python

```
# <project_root>/shared_code/my_second_helper_function.py

def double(value: int) -> int:
    return value * 2
```

You can start writing test cases for your HTTP trigger.

Python

```
# <project_root>/tests/test_my_second_function.py
import unittest
import azure.functions as func

from function_app import main

class TestFunction(unittest.TestCase):
    def test_my_second_function(self):
        # Construct a mock HTTP request.
        req = func.HttpRequest(method='GET',
                               body=None,
                               url='/api/my_second_function',
                               params={'value': '21'})

        # Call the function.
        func_call = main.build().get_user_function()
        resp = func_call(req)
        # Check the output.
        self.assertEqual(
            resp.get_body(),
            b'21 * 2 = 42',
        )
    
```

Inside your `.venv` Python virtual environment folder, install your favorite Python test framework, such as `pip install pytest`. Then run `pytest tests` to check the test result.

Temporary files

The `tempfile.gettempdir()` method returns a temporary folder, which on Linux is `/tmp`. Your application can use this directory to store temporary files that are generated and used by your functions when they're running.

ⓘ Important

Files written to the temporary directory aren't guaranteed to persist across invocations. During scale out, temporary files aren't shared between instances.

The following example creates a named temporary file in the temporary directory (`/tmp`):

Python

```
import logging
import azure.functions as func
import tempfile

from os import listdir
```

```
----  
tempFilePath = tempfile.gettempdir()  
fp = tempfile.NamedTemporaryFile()  
fp.write(b'Hello world!')  
filesDirListInTemp = listdir(tempFilePath)
```

We recommend that you maintain your tests in a folder that's separate from the project folder. This action keeps you from deploying test code with your app.

Preinstalled libraries

A few libraries come with the Python functions runtime.

The Python standard library

The Python standard library contains a list of built-in Python modules that are shipped with each Python distribution. Most of these libraries help you access system functionality, such as file input/output (I/O). On Windows systems, these libraries are installed with Python. On Unix-based systems, they're provided by package collections.

To view the library for your Python version, go to:

- [Python 3.8 standard library ↗](#)
- [Python 3.9 standard library ↗](#)
- [Python 3.10 standard library ↗](#)
- [Python 3.11 standard library ↗](#)

Azure Functions Python worker dependencies

The Azure Functions Python worker requires a specific set of libraries. You can also use these libraries in your functions, but they aren't a part of the Python standard. If your functions rely on any of these libraries, they might be unavailable to your code when it's running outside of Azure Functions.

Note

If your function app's *requirements.txt* file contains an `azure-functions-worker` entry, remove it. The functions worker is automatically managed by the Azure Functions platform, and we regularly update it with new features and bug fixes. Manually installing an old version of worker in the *requirements.txt* file might cause unexpected issues.

(!) Note

If your package contains certain libraries that might collide with worker's dependencies (for example, protobuf, tensorflow, or grpcio), configure **PYTHON_ISOLATE_WORKER_DEPENDENCIES** to 1 in app settings to prevent your application from referring to worker's dependencies.

The Azure Functions Python library

Every Python worker update includes a new version of the [Azure Functions Python library \(azure.functions\)](#). This approach makes it easier to continuously update your Python function apps, because each update is backwards-compatible. For a list of releases of this library, go to [azure-functions PyPi](#).

The runtime library version is fixed by Azure, and it can't be overridden by *requirements.txt*. The `azure-functions` entry in *requirements.txt* is only for linting and customer awareness.

Use the following code to track the actual version of the Python functions library in your runtime:

Python

```
getattr(azure.functions, '__version__', '< 1.2.1')
```

Runtime system libraries

For a list of preinstalled system libraries in Python worker Docker images, see the following:

[+] Expand table

Functions runtime	Debian version	Python versions
Version 3.x	Buster	Python 3.7 Python 3.8 Python 3.9

Python worker extensions

The Python worker process that runs in Azure Functions lets you integrate third-party libraries into your function app. These extension libraries act as middleware that can inject specific operations during the lifecycle of your function's execution.

Extensions are imported in your function code much like a standard Python library module. Extensions are run based on the following scopes:

[+] [Expand table](#)

Scope	Description
Application-level	When imported into any function trigger, the extension applies to every function execution in the app.
Function-level	Execution is limited to only the specific function trigger into which it's imported.

Review the information for each extension to learn more about the scope in which the extension runs.

Extensions implement a Python worker extension interface. This action lets the Python worker process call into the extension code during the function's execution lifecycle. To learn more, see [Create extensions](#).

Using extensions

You can use a Python worker extension library in your Python functions by doing the following:

1. Add the extension package in the *requirements.txt* file for your project.
2. Install the library into your app.
3. Add the following application settings:
 - Locally: Enter `"PYTHON_ENABLE_WORKER_EXTENSIONS": "1"` in the `Values` section of your [*local.settings.json* file](#).
 - Azure: Enter `PYTHON_ENABLE_WORKER_EXTENSIONS=1` in your [app settings](#).
4. Import the extension module into your function trigger.
5. Configure the extension instance, if needed. Configuration requirements should be called out in the extension's documentation.

Important

Third-party Python worker extension libraries aren't supported or warrantied by Microsoft. You must make sure that any extensions that you use in your function

app is trustworthy, and you bear the full risk of using a malicious or poorly written extension.

Third-parties should provide specific documentation on how to install and consume their extensions in your function app. For a basic example of how to consume an extension, see [Consuming your extension](#).

Here are examples of using extensions in a function app, by scope:

Application-level

Python

```
# <project_root>/requirements.txt
application-level-extension==1.0.0
```

Python

```
# <project_root>/Trigger/__init__.py

from application_level_extension import AppExtension
AppExtension.configure(key=value)

def main(req, context):
    # Use context.app_ext_attributes here
```

Creating extensions

Extensions are created by third-party library developers who have created functionality that can be integrated into Azure Functions. An extension developer designs, implements, and releases Python packages that contain custom logic designed specifically to be run in the context of function execution. These extensions can be published either to the PyPI registry or to GitHub repositories.

To learn how to create, package, publish, and consume a Python worker extension package, see [Develop Python worker extensions for Azure Functions](#).

Application-level extensions

An extension that's inherited from [AppExtensionBase](#) runs in an *application* scope.

`AppExtensionBase` exposes the following abstract class methods for you to implement:

[\[\] Expand table](#)

Method	Description
<code>init</code>	Called after the extension is imported.
<code>configure</code>	Called from function code when it's needed to configure the extension.
<code>post_function_load_app_level</code>	Called right after the function is loaded. The function name and function directory are passed to the extension. Keep in mind that the function directory is read-only, and any attempt to write to a local file in this directory fails.
<code>pre_invocation_app_level</code>	Called right before the function is triggered. The function context and function invocation arguments are passed to the extension. You can usually pass other attributes in the context object for the function code to consume.
<code>post_invocation_app_level</code>	Called right after the function execution finishes. The function context, function invocation arguments, and invocation return object are passed to the extension. This implementation is a good place to validate whether execution of the lifecycle hooks succeeded.

Function-level extensions

An extension that inherits from [FuncExtensionBase](#) runs in a specific function trigger.

`FuncExtensionBase` exposes the following abstract class methods for implementations:

[\[\] Expand table](#)

Method	Description
<code>__init__</code>	The constructor of the extension. It's called when an extension instance is initialized in a specific function. When you're implementing this abstract method, you might want to accept a <code>filename</code> parameter and pass it to the parent's method <code>super().__init__(filename)</code> for proper extension registration.
<code>post_function_load</code>	Called right after the function is loaded. The function name and function directory are passed to the extension. Keep in mind that the function directory is read-only, and any attempt to write to a local file in this directory fails.
<code>pre_invocation</code>	Called right before the function is triggered. The function context and function invocation arguments are passed to the extension. You can usually

Method	Description
	pass other attributes in the context object for the function code to consume.
<code>post_invocation</code>	Called right after the function execution finishes. The function context, function invocation arguments, and invocation return object are passed to the extension. This implementation is a good place to validate whether execution of the lifecycle hooks succeeded.

Cross-origin resource sharing

Azure Functions supports cross-origin resource sharing (CORS). CORS is configured [in the portal](#) and through the [Azure CLI](#). The CORS allowed origins list applies at the function app level. With CORS enabled, responses include the `Access-Control-Allow-Origin` header. For more information, see [Cross-origin resource sharing](#).

Cross-origin resource sharing (CORS) is fully supported for Python function apps.

Async

By default, a host instance for Python can process only one function invocation at a time. This is because Python is a single-threaded runtime. For a function app that processes a large number of I/O events or is being I/O bound, you can significantly improve performance by running functions asynchronously. For more information, see [Improve throughout performance of Python apps in Azure Functions](#).

Shared memory (preview)

To improve throughput, Azure Functions lets your out-of-process Python language worker share memory with the Functions host process. When your function app is hitting bottlenecks, you can enable shared memory by adding an application setting named `FUNCTIONS_WORKER_SHARED_MEMORY_DATA_TRANSFER_ENABLED` with a value of `1`. With shared memory enabled, you can then use the `DOCKER_SHM_SIZE` setting to set the shared memory to something like `268435456`, which is equivalent to 256 MB.

For example, you might enable shared memory to reduce bottlenecks when you're using Blob Storage bindings to transfer payloads larger than 1 MB.

This functionality is available only for function apps that are running in Premium and Dedicated (Azure App Service) plans. To learn more, see [Shared memory](#).

Known issues and FAQ

Here are two troubleshooting guides for common issues:

- [ModuleNotFoundError and ImportError](#)
- [Can't import 'cygrpc'](#)

Here are two troubleshooting guides for known issues with the v2 programming model:

- [Couldn't load file or assembly](#)
- [Unable to resolve the Azure Storage connection named Storage](#)

All known issues and feature requests are tracked in a [GitHub issues list ↗](#). If you run into a problem and can't find the issue in GitHub, open a new issue, and include a detailed description of the problem.

Next steps

For more information, see the following resources:

- [Azure Functions package API documentation](#)
- [Best practices for Azure Functions](#)
- [Azure Functions triggers and bindings](#)
- [Blob Storage bindings](#)
- [HTTP and webhook bindings](#)
- [Queue Storage bindings](#)
- [Timer triggers](#)

Having issues with using Python? Tell us what's going on. ↗

Feedback

Was this page helpful?

 Yes	 No
---	--

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Functions Python developer guide

Article • 07/30/2024

This guide is an introduction to developing Azure Functions by using Python. The article assumes that you've already read the [Azure Functions developers guide](#).

Important

This article supports both the v1 and v2 programming model for Python in Azure Functions. The Python v1 model uses a `functions.json` file to define functions, and the new v2 model lets you instead use a decorator-based approach. This new approach results in a simpler file structure, and it's more code-centric. Choose the **v2** selector at the top of the article to learn about this new programming model.

As a Python developer, you might also be interested in these topics:

Get started

- [Visual Studio Code](#): Create your first Python app using Visual Studio Code.
- [Terminal or command prompt](#): Create your first Python app from the command prompt using Azure Functions Core Tools.
- [Samples](#): Review some existing Python apps in the Learn samples browser.

Development options

Both Python Functions programming models support local development in one of the following environments:

Python v2 programming model:

- [Visual Studio Code](#)
- [Terminal or command prompt](#)

Python v1 programming model:

- [Visual Studio Code](#)
- [Terminal or command prompt](#)

You can also create Python v1 functions in the Azure portal.

Tip

Although you can develop your Python-based Azure functions locally on Windows, Python is supported only on a Linux-based hosting plan when it's running in Azure. For more information, see the [list of supported operating system/runtime combinations](#).

Programming model

Azure Functions expects a function to be a stateless method in your Python script that processes input and produces output. By default, the runtime expects the method to be implemented as a global method in the *function_app.py* file.

Triggers and bindings can be declared and used in a function in a decorator based approach. They're defined in the same file, *function_app.py*, as the functions. As an example, the following *function_app.py* file represents a function trigger by an HTTP request.

Python

```
@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req):
    user = req.params.get("user")
    return f"Hello, {user}!"
```

You can also explicitly declare the attribute types and return type in the function by using Python type annotations. Doing so helps you use the IntelliSense and autocomplete features that are provided by many Python code editors.

Python

```
import azure.functions as func

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req: func.HttpRequest) -> str:
    user = req.params.get("user")
    return f"Hello, {user}!"
```

To learn about known limitations with the v2 model and their workarounds, see [Troubleshoot Python errors in Azure Functions](#).

Alternative entry point

The entry point is only in the *function_app.py* file. However, you can reference functions within the project in *function_app.py* by using [blueprints](#) or by importing.

Folder structure

The recommended folder structure for a Python functions project looks like the following example:

```
Windows Command Prompt

<project_root>/
| - .venv/
| - .vscode/
| - function_app.py
| - additional_functions.py
| - tests/
| | - test_my_function.py
| - .funcignore
| - host.json
| - local.settings.json
| - requirements.txt
| - Dockerfile
```

The main project folder, *<project_root>*, can contain the following files:

- *.venv/*: (Optional) Contains a Python virtual environment that's used by local development.
- *.vscode/*: (Optional) Contains the stored Visual Studio Code configuration. To learn more, see [Visual Studio Code settings](#).
- *function_app.py*: The default location for all functions and their related triggers and bindings.
- *additional_functions.py*: (Optional) Any other Python files that contain functions (usually for logical grouping) that are referenced in *function_app.py* through blueprints.
- *tests/*: (Optional) Contains the test cases of your function app.
- *.funcignore*: (Optional) Declares files that shouldn't get published to Azure. Usually, this file contains *.vscode/* to ignore your editor setting, *.venv/* to ignore local

Python virtual environment, `tests/` to ignore test cases, and `local.settings.json` to prevent local app settings being published.

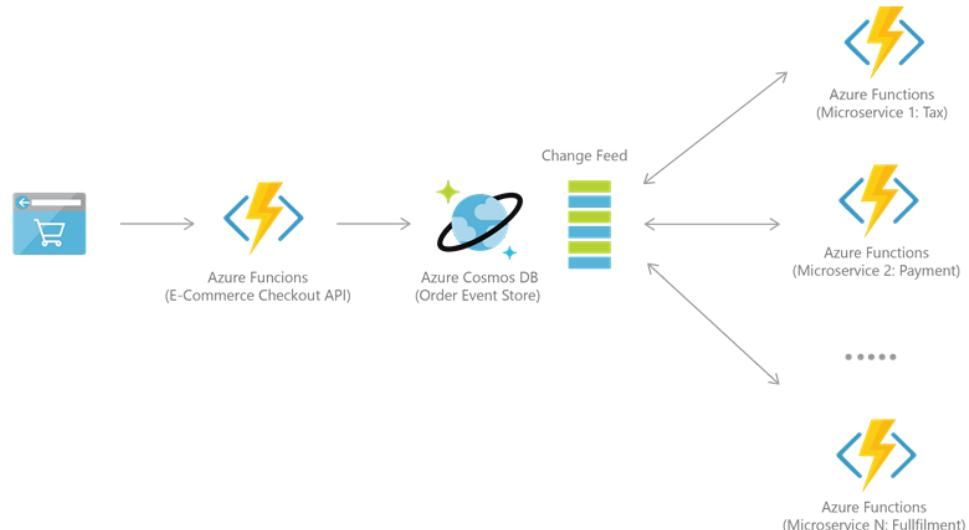
- `host.json`: Contains configuration options that affect all functions in a function app instance. This file does get published to Azure. Not all options are supported when running locally. To learn more, see [host.json](#).
- `local.settings.json`: Used to store app settings and connection strings when it's running locally. This file doesn't get published to Azure. To learn more, see [local.settings.json](#).
- `requirements.txt`: Contains the list of Python packages the system installs when it publishes to Azure.
- `Dockerfile`: (Optional) Used when publishing your project in a [custom container](#).

When you deploy your project to a function app in Azure, the entire contents of the main project folder, `<project_root>`, should be included in the package, but not the folder itself, which means that `host.json` should be in the package root. We recommend that you maintain your tests in a folder along with other functions (in this example, `tests/`). For more information, see [Unit testing](#).

Connect to a database

Azure Functions integrates well with [Azure Cosmos DB](#) for many [use cases](#), including IoT, ecommerce, gaming, etc.

For example, for [event sourcing](#), the two services are integrated to power event-driven architectures using Azure Cosmos DB's [change feed](#) functionality. The change feed provides downstream microservices the ability to reliably and incrementally read inserts and updates (for example, order events). This functionality can be used to provide a persistent event store as a message broker for state-changing events and drive order processing workflow between many microservices (which can be implemented as [serverless Azure Functions](#)).



To connect to Azure Cosmos DB, first [create an account, database, and container](#). Then you can connect your function code to Azure Cosmos DB using [trigger and bindings](#), like this [example](#).

To implement more complex app logic, you can also use the Python library for Cosmos DB. An asynchronous I/O implementation looks like this:

Python

```

pip install azure-cosmos
pip install aiohttp

from azure.cosmos.aio import CosmosClient
from azure.cosmos import exceptions
from azure.cosmos.partition_key import PartitionKey
import asyncio

# Replace these values with your Cosmos DB connection information
endpoint = "https://azure-cosmos-nosql.documents.azure.com:443/"
key = "master_key"
database_id = "cosmicwerx"
container_id = "cosmiccontainer"
partition_key = "/partition_key"

# Set the total throughput (RU/s) for the database and container
database_throughput = 1000

# Singleton CosmosClient instance
client = CosmosClient(endpoint, credential=key)

# Helper function to get or create database and container
async def get_or_create_container(client, database_id, container_id,
partition_key):

```

```

database = await client.create_database_if_not_exists(id=database_id)
print(f'Database "{database_id}" created or retrieved successfully.')

        container = await
database.create_container_if_not_exists(id=container_id,
partition_key=PartitionKey(path=partition_key))
        print(f'Container with id "{container_id}" created')

    return container

async def create_products():
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    for i in range(10):
        await container.upsert_item({
            'id': f'item{i}',
            'productName': 'Widget',
            'productModel': f'Model {i}'
        })

async def get_products():
    items = []
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    async for item in container.read_all_items():
        items.append(item)
    return items

async def query_products(product_name):
    container = await get_or_create_container(client, database_id,
container_id, partition_key)
    query = f"SELECT * FROM c WHERE c.productName = '{product_name}'"
    items = []
    async for item in container.query_items(query=query,
enable_cross_partition_query=True):
        items.append(item)
    return items

async def main():
    await create_products()
    all_products = await get_products()
    print('All Products:', all_products)

    queried_products = await query_products('Widget')
    print('Queried Products:', queried_products)

if __name__ == "__main__":
    asyncio.run(main())

```

Blueprints

The Python v2 programming model introduces the concept of *blueprints*. A blueprint is a new class that's instantiated to register functions outside of the core function application. The functions registered in blueprint instances aren't indexed directly by the function runtime. To get these blueprint functions indexed, the function app needs to register the functions from blueprint instances.

Using blueprints provides the following benefits:

- Lets you break up the function app into modular components, which enables you to define functions in multiple Python files and divide them into different components per file.
- Provides extensible public function app interfaces to build and reuse your own APIs.

The following example shows how to use blueprints:

First, in an *http_blueprint.py* file, an HTTP-triggered function is first defined and added to a blueprint object.

Python

```
import logging

import azure.functions as func

bp = func.Blueprint()

@bp.route(route="default_template")
def default_template(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(
            f"Hello, {name}. This HTTP-triggered function "
            f"executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP-triggered function executed successfully. "
            "Pass a name in the query string or in the request body for a"
            " personalized response.",
```

```
    status_code=200
)
```

Next, in the `function_app.py` file, the `blueprint` object is imported and its functions are registered to the function app.

Python

```
import azure.functions as func
from http_blueprint import bp

app = func.FunctionApp()

app.register_functions(bp)
```

ⓘ Note

Durable Functions also supports blueprints. To create blueprints for Durable Functions apps, register your orchestration, activity, and entity triggers and client bindings using the [azure-functions-durable](#) Blueprint class, as shown [here](#). The resulting blueprint can then be registered as normal. See our [sample](#) for an example.

Triggers and inputs

Inputs are divided into two categories in Azure Functions: trigger input and other input. Although they're defined using different decorators, their usage is similar in Python code. Connection strings or secrets for trigger and input sources map to values in the `local.settings.json` file when they're running locally, and they map to the application settings when they're running in Azure.

As an example, the following code demonstrates how to define a Blob Storage input binding:

JSON

```
// local.settings.json
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "STORAGE_CONNECTION_STRING": "<AZURE_STORAGE_CONNECTION_STRING>",
    "AzureWebJobsStorage": "<azure-storage-connection-string>"}
```

```
}
```

Python

```
# function_app.py
import azure.functions as func
import logging

app = func.FunctionApp()

@app.route(route="req")
@app.read_blob(arg_name="obj", path="samples/{id}",
               connection="STORAGE_CONNECTION_STRING")
def main(req: func.HttpRequest, obj: func.InputStream):
    logging.info(f'Python HTTP-triggered function processed: {obj.read()}')
```

When the function is invoked, the HTTP request is passed to the function as `req`. An entry is retrieved from the Azure Blob Storage account based on the *ID* in the route URL and made available as `obj` in the function body. Here, the specified storage account is the connection string that's found in the `STORAGE_CONNECTION_STRING` app setting.

For data intensive binding operations, you may want to use a separate storage account. For more information, see [Storage account guidance](#).

SDK type bindings (preview)

For select triggers and bindings, you can work with data types implemented by the underlying Azure SDKs and frameworks. These *SDK type bindings* let you interact with binding data as if you were using the underlying service SDK.

Functions supports Python SDK type bindings for Azure Blob storage, which lets you work with blob data using the underlying `BlobClient` type.

ⓘ Important

SDK type bindings support for Python is currently in preview:

- You must use the Python v2 programming model.
- Currently, only synchronous SDK types are supported.

Prerequisites

- Azure Functions runtime version version 4.34, or a later version.
- Python ↗ version 3.9, or a later supported version.

Enable SDK type bindings for the Blob storage extension

1. Add the `azurefunctions-extensions-bindings-blob` extension package to the `requirements.txt` file in the project, which should include at least these packages:

```
text  
  
azure-functions  
azurefunctions-extensions-bindings-blob
```

2. Add this code to the `function_app.py` file in the project, which imports the SDK type bindings:

```
Python  
  
import azurefunctions.extensions.bindings.blob as blob
```

SDK type bindings examples

This example shows how to get the `BlobClient` from both a Blob storage trigger (`blob_trigger`) and from the input binding on an HTTP trigger (`blob_input`):

```
Python  
  
import logging  
  
import azure.functions as func  
import azurefunctions.extensions.bindings.blob as blob  
  
app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)  
  
@app.blob_trigger(  
    arg_name="client", path="PATH/TO/BLOB", connection="AzureWebJobsStorage"  
)  
def blob_trigger(client: blob.BlobClient):  
    logging.info(  
        f"Python blob trigger function processed blob \n"  
        f"Properties: {client.get_blob_properties()}\n"  
        f"Blob content head: {client.download_blob().read(size=1)}"  
    )  
  
@app.route(route="file")
```

```
@app.blob_input(  
    arg_name="client", path="PATH/TO/BLOB", connection="AzureWebJobsStorage"  
)  
def blob_input(req: func.HttpRequest, client: blob.BlobClient):  
    logging.info(  
        f"Python blob input function processed blob \n"  
        f"Properties: {client.get_blob_properties()}\n"  
        f"Blob content head: {client.download_blob().read(size=1)}"  
    )  
    return "ok"
```

You can view other SDK type bindings samples for Blob storage in the Python extensions repository:

- [ContainerClient type ↗](#)
- [StorageStreamDownloader type ↗](#)

HTTP streams (preview)

HTTP streams lets you accept and return data from your HTTP endpoints using FastAPI request and response APIs enabled in your functions. These APIs lets the host process large data in HTTP messages as chunks instead of reading an entire message into memory.

This feature makes it possible to handle large data stream, OpenAI integrations, deliver dynamic content, and support other core HTTP scenarios requiring real-time interactions over HTTP. You can also use FastAPI response types with HTTP streams. Without HTTP streams, the size of your HTTP requests and responses are limited by memory restrictions that can be encountered when processing entire message payloads all in memory.

ⓘ Important

HTTP streams support for Python is currently in preview and requires you to use the Python v2 programming model.

Prerequisites

- [Azure Functions runtime](#) version 4.34.1, or a later version.
- [Python ↗](#) version 3.8, or a later [supported version](#).

Enable HTTP streams

HTTP streams are disabled by default. You need to enable this feature in your application settings and also update your code to use the FastAPI package. Note that when enabling HTTP streams, the function app will default to using HTTP streaming, and the original HTTP functionality will not work.

1. Add the `azurefunctions-extensions-http-fastapi` extension package to the `requirements.txt` file in the project, which should include at least these packages:

```
text  
  
azure-functions  
azurefunctions-extensions-http-fastapi
```

2. Add this code to the `function_app.py` file in the project, which imports the FastAPI extension:

```
Python  
  
from azurefunctions.extensions.http.fastapi import Request,  
StreamingResponse
```

3. When you deploy to Azure, add the following [application setting](#) in your function app:

```
"PYTHON_ENABLE_INIT_INDEXING": "1"
```

If you are deploying to Linux Consumption, also add

```
"PYTHON_ISOLATE_WORKER_DEPENDENCIES": "1"
```

When running locally, you also need to add these same settings to the `local.settings.json` project file.

HTTP streams examples

After you enable the HTTP streaming feature, you can create functions that stream data over HTTP.

This example is an HTTP triggered function that streams HTTP response data. You might use these capabilities to support scenarios like sending event data through a pipeline for real time visualization or detecting anomalies in large sets of data and providing instant notifications.

```
Python
```

```

import time

import azure.functions as func
from azurefunctions.extensions.http.fastapi import Request,
StreamingResponse

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

def generate_sensor_data():
    """Generate real-time sensor data."""
    for i in range(10):
        # Simulate temperature and humidity readings
        temperature = 20 + i
        humidity = 50 + i
        yield f"data: {{'temperature': {temperature}, 'humidity': {humidity}}}\n\n"
    time.sleep(1)

@app.route(route="stream", methods=[func.HttpMethod.GET])
async def stream_sensor_data(req: Request) -> StreamingResponse:
    """Endpoint to stream real-time sensor data."""
    return StreamingResponse(generate_sensor_data(), media_type="text/event-stream")

```

This example is an HTTP triggered function that receives and processes streaming data from a client in real time. It demonstrates streaming upload capabilities that can be helpful for scenarios like processing continuous data streams and handling event data from IoT devices.

Python

```

import azure.functions as func
from azurefunctions.extensions.http.fastapi import JSONResponse, Request

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="streaming_upload", methods=[func.HttpMethod.POST])
async def streaming_upload(req: Request) -> JSONResponse:
    """Handle streaming upload requests."""
    # Process each chunk of data as it arrives
    async for chunk in req.stream():
        process_data_chunk(chunk)

    # Once all data is received, return a JSON response indicating
    # successful processing
    return JSONResponse({"status": "Data uploaded and processed
successfully"})

```

```
def process_data_chunk(chunk: bytes):
    """Process each data chunk."""
    # Add custom processing logic here
    pass
```

Calling HTTP streams

You must use an HTTP client library to make streaming calls to a function's FastAPI endpoints. The client tool or browser you're using might not natively support streaming or could only return the first chunk of data.

You can use a client script like this to send streaming data to an HTTP endpoint:

Python

```
import httpx # Be sure to add 'httpx' to 'requirements.txt'
import asyncio

async def stream_generator(file_path):
    chunk_size = 2 * 1024 # Define your own chunk size
    with open(file_path, 'rb') as file:
        while chunk := file.read(chunk_size):
            yield chunk
            print(f"Sent chunk: {len(chunk)} bytes")

async def stream_to_server(url, file_path):
    timeout = httpx.Timeout(60.0, connect=60.0)
    async with httpx.AsyncClient(timeout=timeout) as client:
        response = await client.post(url,
content=stream_generator(file_path))
    return response

async def stream_response(response):
    if response.status_code == 200:
        async for chunk in response.aiter_raw():
            print(f"Received chunk: {len(chunk)} bytes")
    else:
        print(f"Error: {response}")

async def main():
    print('helloworld')
    # Customize your streaming endpoint served from core tool in variable
    'url' if different.
    url = 'http://localhost:7071/api/streaming_upload'
    file_path = r'<file path>'

    response = await stream_to_server(url, file_path)
    print(response)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Outputs

Output can be expressed both in return value and output parameters. If there's only one output, we recommend using the return value. For multiple outputs, you'll have to use output parameters.

To produce multiple outputs, use the `set()` method provided by the `azure.functions.Out` interface to assign a value to the binding. For example, the following function can push a message to a queue and also return an HTTP response.

Python

```
# function_app.py
import azure.functions as func

app = func.FunctionApp()

@app.write_blob(arg_name="msg", path="output-container/{name}",
                connection="CONNECTION_STRING")
def test_function(req: func.HttpRequest,
                  msg: func.Out[str]) -> str:

    message = req.params.get('body')
    msg.set(message)
    return message
```

Logging

Access to the Azure Functions runtime logger is available via a root [logging](#) handler in your function app. This logger is tied to Application Insights and allows you to flag warnings and errors that occur during the function execution.

The following example logs an info message when the function is invoked via an HTTP trigger.

Python

```
import logging

def main(req):
    logging.info('Python HTTP trigger function processed a request.')
```

More logging methods are available that let you write to the console at different trace levels:

[+] Expand table

Method	Description
<code>critical(_message_)</code>	Writes a message with level CRITICAL on the root logger.
<code>error(_message_)</code>	Writes a message with level ERROR on the root logger.
<code>warning(_message_)</code>	Writes a message with level WARNING on the root logger.
<code>info(_message_)</code>	Writes a message with level INFO on the root logger.
<code>debug(_message_)</code>	Writes a message with level DEBUG on the root logger.

To learn more about logging, see [Monitor Azure Functions](#).

Logging from created threads

To see logs coming from your created threads, include the `context` argument in the function's signature. This argument contains an attribute `thread_local_storage` that stores a local `invocation_id`. This can be set to the function's current `invocation_id` to ensure the context is changed.

Python

```
import azure.functions as func
import logging
import threading

def main(req, context):
    logging.info('Python HTTP trigger function processed a request.')
    t = threading.Thread(target=log_function, args=(context,))
    t.start()

def log_function(context):
    context.thread_local_storage.invocation_id = context.invocation_id
    logging.info('Logging from thread.')
```

Log custom telemetry

By default, the Functions runtime collects logs and other telemetry data that are generated by your functions. This telemetry ends up as traces in Application Insights. Request and dependency telemetry for certain Azure services are also collected by default by [triggers and bindings](#).

To collect custom request and custom dependency telemetry outside of bindings, you can use the [OpenCensus Python Extensions](#). This extension sends custom telemetry data to your Application Insights instance. You can find a list of supported extensions at the [OpenCensus repository](#).

ⓘ Note

To use the OpenCensus Python extensions, you need to enable [Python worker extensions](#) in your function app by setting `PYTHON_ENABLE_WORKER_EXTENSIONS` to `1`. You also need to switch to using the Application Insights connection string by adding the [APPLICATIONINSIGHTS CONNECTION STRING](#) setting to your [application settings](#), if it's not already there.

text

```
// requirements.txt
...
opencensus-extension-azure-functions
opencensus-ext-requests
```

Python

```
import json
import logging

import requests
from opencensus.extension.azure.functions import OpenCensusExtension
from opencensus.trace import config_integration

config_integration.trace_integrations(['requests'])

OpenCensusExtension.configure()

def main(req, context):
    logging.info('Executing HttpTrigger with OpenCensus extension')

    # You must use contexttracer to create spans
    with context.tracer.span("parent"):
        response = requests.get(url='http://example.com')

    return json.dumps({
        'method': req.method,
```

```
'response': response.status_code,
'ctx_func_name': context.function_name,
'ctx_func_dir': context.function_directory,
'ctx_invocation_id': context.invocation_id,
'ctx_trace_context_Traceparent': context.trace_context.Traceparent,
'ctx_trace_context_Tracestate': context.trace_context.Tracestate,
'ctx_retry_context_RetryCount': context.retry_context.retry_count,
'ctx_retry_context_MaxRetryCount':
context.retry_context.max_retry_count,
})
```

HTTP trigger

The HTTP trigger is defined as a method that takes a named binding parameter, which is an [HttpRequest](#) object, and returns an [HttpResponse](#) object. You apply the `function_name` decorator to the method to define the function name, while the HTTP endpoint is set by applying the `route` decorator.

This example is from the HTTP trigger template for the Python v2 programming model, where the binding parameter name is `req`. It's the sample code that's provided when you create a function by using Azure Functions Core Tools or Visual Studio Code.

Python

```
@app.function_name(name="HttpTrigger1")
@app.route(route="hello")
def test_function(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
    else:
        name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello, {name}. This HTTP-triggered
function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP-triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
            status_code=200
        )
```

From the `HttpRequest` object, you can get request headers, query parameters, route parameters, and the message body. In this function, you obtain the value of the `name` query parameter from the `params` parameter of the `HttpRequest` object. You read the JSON-encoded message body by using the `get_json` method.

Likewise, you can set the `status_code` and `headers` for the response message in the returned `HttpResponse` object.

To pass in a name in this example, paste the URL that's provided when you're running the function, and then append it with `?name={name}`.

Web frameworks

You can use Asynchronous Server Gateway Interface (ASGI)-compatible and Web Server Gateway Interface (WSGI)-compatible frameworks, such as Flask and FastAPI, with your HTTP-triggered Python functions. You must first update the `host.json` file to include an HTTP `routePrefix`, as shown in the following example:

```
JSON

{
  "version": "2.0",
  "logging": {
    {
      "applicationInsights": {
        {
          "samplingSettings": {
            {
              "isEnabled": true,
              "excludedTypes": "Request"
            }
          }
        },
        "extensionBundle": {
          {
            "id": "Microsoft.Azure.Functions.ExtensionBundle",
            "version": "[2.*, 3.0.0)"
          },
          "extensions": {
            {
              "http": {
                {
                  "routePrefix": ""
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The framework code looks like the following example:

ASGI

`AsgiFunctionApp` is the top-level function app class for constructing ASGI HTTP functions.

Python

```
# function_app.py

import azure.functions as func
from fastapi import FastAPI, Request, Response

fast_app = FastAPI()

@fast_app.get("/return_http_no_body")
async def return_http_no_body():
    return Response(content="", media_type="text/plain")

app = func.AsgiFunctionApp(app=fast_app,
                           http_auth_level=func.AuthLevel.ANONYMOUS)
```

Scaling and performance

For scaling and performance best practices for Python function apps, see the [Python scaling and performance](#) article.

Context

To get the invocation context of a function when it's running, include the `context` argument in its signature.

For example:

Python

```
import azure.functions

def main(req: azure.functions.HttpRequest,
         context: azure.functions.Context) -> str:
    return f'{context.invocation_id}'
```

The [Context](#) class has the following string attributes:

[Expand table](#)

Attribute	Description
<code>function_directory</code>	The directory in which the function is running.
<code>function_name</code>	The name of the function.
<code>invocation_id</code>	The ID of the current function invocation.
<code>thread_local_storage</code>	The thread local storage of the function. Contains a local <code>invocation_id</code> for logging from created threads .
<code>trace_context</code>	The context for distributed tracing. For more information, see Trace Context .
<code>retry_context</code>	The context for retries to the function. For more information, see retry-policies .

Global variables

It isn't guaranteed that the state of your app will be preserved for future executions. However, the Azure Functions runtime often reuses the same process for multiple executions of the same app. To cache the results of an expensive computation, declare it as a global variable.

Python

```
CACHED_DATA = None

def main(req):
    global CACHED_DATA
    if CACHED_DATA is None:
        CACHED_DATA = load_json()

    # ... use CACHED_DATA in code
```

Environment variables

In Azure Functions, [application settings](#), such as service connection strings, are exposed as environment variables when they're running. There are two main ways to access these settings in your code.

[\[\] Expand table](#)

Method	Description
<code>os.environ["myAppSetting"]</code>	Tries to get the application setting by key name, and raises an error when it's unsuccessful.
<code>os.getenv("myAppSetting")</code>	Tries to get the application setting by key name, and returns <code>null</code> when it's unsuccessful.

Both of these ways require you to declare `import os`.

The following example uses `os.environ["myAppSetting"]` to get the [application setting](#), with the key named `myAppSetting`:

Python

```
import logging
import os

import azure.functions as func

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="req")
def main(req: func.HttpRequest) -> func.HttpResponse:
    # Get the setting named 'myAppSetting'
    my_app_setting_value = os.environ["myAppSetting"]
    logging.info(f'My app setting value:{my_app_setting_value}')
```

For local development, application settings are [maintained in the `local.settings.json` file](#).

Python version

Azure Functions supports the following Python versions:

[\[\] Expand table](#)

Functions version	Python* versions
4.x	3.11
	3.10
	3.9
	3.8
	3.7

Functions version	Python* versions
3.x	3.9
	3.8
	3.7

* Official Python distributions

To request a specific Python version when you create your function app in Azure, use the `--runtime-version` option of the [az functionapp create](#) command. The Functions runtime version is set by the `--functions-version` option. The Python version is set when the function app is created, and it can't be changed for apps running in a Consumption plan.

The runtime uses the available Python version when you run it locally.

Changing Python version

To set a Python function app to a specific language version, you need to specify the language and the version of the language in the `LinuxFxVersion` field in the site configuration. For example, to change the Python app to use Python 3.8, set `linuxFxVersion` to `python|3.8`.

To learn how to view and change the `linuxFxVersion` site setting, see [How to target Azure Functions runtime versions](#).

For more general information, see the [Azure Functions runtime support policy](#) and [Supported languages in Azure Functions](#).

Package management

When you're developing locally by using Core Tools or Visual Studio Code, add the names and versions of the required packages to the `requirements.txt` file, and then install them by using `pip`.

For example, you can use the following `requirements.txt` file and `pip` command to install the `requests` package from PyPI.

```
txt
requests==2.19.1
```

Bash

```
pip install -r requirements.txt
```

When running your functions in an [App Service plan](#), dependencies that you define in `requirements.txt` are given precedence over built-in Python modules, such as `logging`. This precedence can cause conflicts when built-in modules have the same names as directories in your code. When running in a [Consumption plan](#) or an [Elastic Premium plan](#), conflicts are less likely because your dependencies aren't prioritized by default.

To prevent issues running in an App Service plan, don't name your directories the same as any Python native modules and don't include Python native libraries in your project's `requirements.txt` file.

Publishing to Azure

When you're ready to publish, make sure that all your publicly available dependencies are listed in the `requirements.txt` file. You can locate this file at the root of your project directory.

You can find the project files and folders that are excluded from publishing, including the virtual environment folder, in the root directory of your project.

There are three build actions supported for publishing your Python project to Azure: remote build, local build, and builds using custom dependencies.

You can also use Azure Pipelines to build your dependencies and publish by using continuous delivery (CD). To learn more, see [Continuous delivery with Azure Pipelines](#).

Remote build

When you use remote build, dependencies that are restored on the server and native dependencies match the production environment. This results in a smaller deployment package to upload. Use remote build when you're developing Python apps on Windows. If your project has custom dependencies, you can [use remote build with extra index URL](#).

Dependencies are obtained remotely based on the contents of the `requirements.txt` file. [Remote build](#) is the recommended build method. By default, Core Tools requests a remote build when you use the following `func azure functionapp publish` command to publish your Python project to Azure.

Bash

```
func azure functionapp publish <APP_NAME>
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

The [Azure Functions Extension for Visual Studio Code](#) also requests a remote build by default.

Local build

Dependencies are obtained locally based on the contents of the `requirements.txt` file. You can prevent doing a remote build by using the following [func azure functionapp publish](#) command to publish with a local build:

command

```
func azure functionapp publish <APP_NAME> --build local
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

When you use the `--build local` option, project dependencies are read from the `requirements.txt` file, and those dependent packages are downloaded and installed locally. Project files and dependencies are deployed from your local computer to Azure. This results in a larger deployment package being uploaded to Azure. If for some reason you can't get the `requirements.txt` file by using Core Tools, you must use the custom dependencies option for publishing.

We don't recommend using local builds when you're developing locally on Windows.

Custom dependencies

When your project has dependencies that aren't found in the [Python Package Index](#), there are two ways to build the project. The first way, the `build` method, depends on how you build the project.

Remote build with extra index URL

When your packages are available from an accessible custom package index, use a remote build. Before you publish, be sure to [create an app setting](#) named `PIP_EXTRA_INDEX_URL`. The value for this setting is the URL of your custom package

index. Using this setting tells the remote build to run `pip install` by using the `--extra-index-url` option. To learn more, see the [Python pip install documentation](#).

You can also use basic authentication credentials with your extra package index URLs. To learn more, see [Basic authentication credentials](#) in the Python documentation.

Install local packages

If your project uses packages that aren't publicly available to our tools, you can make them available to your app by putting them in the `_app_/python_packages` directory. Before you publish, run the following command to install the dependencies locally:

command

```
pip install --target="<PROJECT_DIR>/python_packages/lib/site-packages" -r requirements.txt
```

When you're using custom dependencies, you should use the `--no-build` publishing option, because you've already installed the dependencies into the project folder.

command

```
func azure functionapp publish <APP_NAME> --no-build
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

Unit testing

Functions that are written in Python can be tested like other Python code by using standard testing frameworks. For most bindings, it's possible to create a mock input object by creating an instance of an appropriate class from the `azure.functions` package. Since the `azure.functions` package isn't immediately available, be sure to install it via your `requirements.txt` file as described in the [package management](#) section above.

With `my_second_function` as an example, the following is a mock test of an HTTP-triggered function:

First, create the `<project_root>/function_app.py` file and implement the `my_second_function` function as the HTTP trigger and `shared_code.my_second_helper_function`.

Python

```
# <project_root>/function_app.py
import azure.functions as func
import logging

# Use absolute import to resolve shared_code modules
from shared_code import my_second_helper_function

app = func.FunctionApp()

# Define the HTTP trigger that accepts the ?value=<int> query parameter
# Double the value and return the result in HttpResponseMessage
@app.function_name(name="my_second_function")
@app.route(route="hello")
def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Executing my_second_function.')

    initial_value: int = int(req.params.get('value'))
    doubled_value: int = my_second_helper_function.double(initial_value)

    return func.HttpResponse(
        body=f'{initial_value} * 2 = {doubled_value}',
        status_code=200
    )
```

Python

```
# <project_root>/shared_code/__init__.py
# Empty __init__.py file marks shared_code folder as a Python package
```

Python

```
# <project_root>/shared_code/my_second_helper_function.py

def double(value: int) -> int:
    return value * 2
```

You can start writing test cases for your HTTP trigger.

Python

```
# <project_root>/tests/test_my_second_function.py
import unittest
import azure.functions as func

from function_app import main

class TestFunction(unittest.TestCase):
    def test_my_second_function(self):
```

```
# Construct a mock HTTP request.
req = func.HttpRequest(method='GET',
                       body=None,
                       url='/api/my_second_function',
                       params={'value': '21'})

# Call the function.
func_call = main.build().get_user_function()
resp = func_call(req)

# Check the output.
self.assertEqual(
    resp.get_body(),
    b'21 * 2 = 42',
)
```

Inside your `.venv` Python virtual environment folder, install your favorite Python test framework, such as `pip install pytest`. Then run `pytest tests` to check the test result.

Temporary files

The `tempfile.gettempdir()` method returns a temporary folder, which on Linux is `/tmp`. Your application can use this directory to store temporary files that are generated and used by your functions when they're running.

ⓘ Important

Files written to the temporary directory aren't guaranteed to persist across invocations. During scale out, temporary files aren't shared between instances.

The following example creates a named temporary file in the temporary directory (`/tmp`):

Python

```
import logging
import azure.functions as func
import tempfile

from os import listdir

#---
tempFilePath = tempfile.gettempdir()
fp = tempfile.NamedTemporaryFile()
fp.write(b'Hello world!')
filesDirListInTemp = listdir(tempFilePath)
```

We recommend that you maintain your tests in a folder that's separate from the project folder. This action keeps you from deploying test code with your app.

Preinstalled libraries

A few libraries come with the Python functions runtime.

The Python standard library

The Python standard library contains a list of built-in Python modules that are shipped with each Python distribution. Most of these libraries help you access system functionality, such as file input/output (I/O). On Windows systems, these libraries are installed with Python. On Unix-based systems, they're provided by package collections.

To view the library for your Python version, go to:

- [Python 3.8 standard library ↗](#)
- [Python 3.9 standard library ↗](#)
- [Python 3.10 standard library ↗](#)
- [Python 3.11 standard library ↗](#)

Azure Functions Python worker dependencies

The Azure Functions Python worker requires a specific set of libraries. You can also use these libraries in your functions, but they aren't a part of the Python standard. If your functions rely on any of these libraries, they might be unavailable to your code when it's running outside of Azure Functions.

Note

If your function app's *requirements.txt* file contains an `azure-functions-worker` entry, remove it. The functions worker is automatically managed by the Azure Functions platform, and we regularly update it with new features and bug fixes. Manually installing an old version of worker in the *requirements.txt* file might cause unexpected issues.

Note

If your package contains certain libraries that might collide with worker's dependencies (for example, protobuf, tensorflow, or grpcio), configure `PYTHON_ISOLATE_WORKER_DEPENDENCIES` to `1` in app settings to prevent your application from referring to worker's dependencies.

The Azure Functions Python library

Every Python worker update includes a new version of the [Azure Functions Python library \(azure.functions\)](#). This approach makes it easier to continuously update your Python function apps, because each update is backwards-compatible. For a list of releases of this library, go to [azure-functions PyPi](#).

The runtime library version is fixed by Azure, and it can't be overridden by *requirements.txt*. The `azure-functions` entry in *requirements.txt* is only for linting and customer awareness.

Use the following code to track the actual version of the Python functions library in your runtime:

Python

```
getattr(azure.functions, '__version__', '< 1.2.1')
```

Runtime system libraries

For a list of preinstalled system libraries in Python worker Docker images, see the following:

[+] Expand table

Functions runtime	Debian version	Python versions
Version 3.x	Buster	Python 3.7 Python 3.8 Python 3.9

Python worker extensions

The Python worker process that runs in Azure Functions lets you integrate third-party libraries into your function app. These extension libraries act as middleware that can inject specific operations during the lifecycle of your function's execution.

Extensions are imported in your function code much like a standard Python library module. Extensions are run based on the following scopes:

[+] Expand table

Scope	Description
Application-level	When imported into any function trigger, the extension applies to every function execution in the app.
Function-level	Execution is limited to only the specific function trigger into which it's imported.

Review the information for each extension to learn more about the scope in which the extension runs.

Extensions implement a Python worker extension interface. This action lets the Python worker process call into the extension code during the function's execution lifecycle. To learn more, see [Create extensions](#).

Using extensions

You can use a Python worker extension library in your Python functions by doing the following:

1. Add the extension package in the *requirements.txt* file for your project.
2. Install the library into your app.
3. Add the following application settings:
 - Locally: Enter `"PYTHON_ENABLE_WORKER_EXTENSIONS": "1"` in the `Values` section of your *local.settings.json* file.
 - Azure: Enter `PYTHON_ENABLE_WORKER_EXTENSIONS=1` in your [app settings](#).
4. Import the extension module into your function trigger.
5. Configure the extension instance, if needed. Configuration requirements should be called out in the extension's documentation.

Important

Third-party Python worker extension libraries aren't supported or warrantied by Microsoft. You must make sure that any extensions that you use in your function app is trustworthy, and you bear the full risk of using a malicious or poorly written extension.

Third-parties should provide specific documentation on how to install and consume their extensions in your function app. For a basic example of how to consume an extension, see [Consuming your extension](#).

Here are examples of using extensions in a function app, by scope:

Application-level

Python

```
# <project_root>/requirements.txt
application-level-extension==1.0.0
```

Python

```
# <project_root>/Trigger/__init__.py

from application_level_extension import AppExtension
AppExtension.configure(key=value)

def main(req, context):
    # Use context.app_ext_attributes here
```

Creating extensions

Extensions are created by third-party library developers who have created functionality that can be integrated into Azure Functions. An extension developer designs, implements, and releases Python packages that contain custom logic designed specifically to be run in the context of function execution. These extensions can be published either to the PyPI registry or to GitHub repositories.

To learn how to create, package, publish, and consume a Python worker extension package, see [Develop Python worker extensions for Azure Functions](#).

Application-level extensions

An extension that's inherited from [AppExtensionBase](#) runs in an *application* scope.

`AppExtensionBase` exposes the following abstract class methods for you to implement:

[] [Expand table](#)

Method	Description
<code>init</code>	Called after the extension is imported.
<code>configure</code>	Called from function code when it's needed to configure the extension.

Method	Description
<code>post_function_load_app_level</code>	Called right after the function is loaded. The function name and function directory are passed to the extension. Keep in mind that the function directory is read-only, and any attempt to write to a local file in this directory fails.
<code>pre_invocation_app_level</code>	Called right before the function is triggered. The function context and function invocation arguments are passed to the extension. You can usually pass other attributes in the context object for the function code to consume.
<code>post_invocation_app_level</code>	Called right after the function execution finishes. The function context, function invocation arguments, and invocation return object are passed to the extension. This implementation is a good place to validate whether execution of the lifecycle hooks succeeded.

Function-level extensions

An extension that inherits from [FuncExtensionBase](#) runs in a specific function trigger.

`FuncExtensionBase` exposes the following abstract class methods for implementations:

[+] [Expand table](#)

Method	Description
<code>__init__</code>	The constructor of the extension. It's called when an extension instance is initialized in a specific function. When you're implementing this abstract method, you might want to accept a <code>filename</code> parameter and pass it to the parent's method <code>super().__init__(filename)</code> for proper extension registration.
<code>post_function_load</code>	Called right after the function is loaded. The function name and function directory are passed to the extension. Keep in mind that the function directory is read-only, and any attempt to write to a local file in this directory fails.
<code>pre_invocation</code>	Called right before the function is triggered. The function context and function invocation arguments are passed to the extension. You can usually pass other attributes in the context object for the function code to consume.
<code>post_invocation</code>	Called right after the function execution finishes. The function context, function invocation arguments, and invocation return object are passed to the extension. This implementation is a good place to validate whether execution of the lifecycle hooks succeeded.

Cross-origin resource sharing

Azure Functions supports cross-origin resource sharing (CORS). CORS is configured in the portal and through the Azure CLI. The CORS allowed origins list applies at the function app level. With CORS enabled, responses include the `Access-Control-Allow-Origin` header. For more information, see [Cross-origin resource sharing](#).

Cross-origin resource sharing (CORS) is fully supported for Python function apps.

Async

By default, a host instance for Python can process only one function invocation at a time. This is because Python is a single-threaded runtime. For a function app that processes a large number of I/O events or is being I/O bound, you can significantly improve performance by running functions asynchronously. For more information, see [Improve throughout performance of Python apps in Azure Functions](#).

Shared memory (preview)

To improve throughput, Azure Functions lets your out-of-process Python language worker share memory with the Functions host process. When your function app is hitting bottlenecks, you can enable shared memory by adding an application setting named `FUNCTIONS_WORKER_SHARED_MEMORY_DATA_TRANSFER_ENABLED` with a value of `1`. With shared memory enabled, you can then use the `DOCKER_SHM_SIZE` setting to set the shared memory to something like `268435456`, which is equivalent to 256 MB.

For example, you might enable shared memory to reduce bottlenecks when you're using Blob Storage bindings to transfer payloads larger than 1 MB.

This functionality is available only for function apps that are running in Premium and Dedicated (Azure App Service) plans. To learn more, see [Shared memory](#).

Known issues and FAQ

Here are two troubleshooting guides for common issues:

- [ModuleNotFoundError and ImportError](#)
- [Can't import 'cygrpc'](#)

Here are two troubleshooting guides for known issues with the v2 programming model:

- Couldn't load file or assembly
- Unable to resolve the Azure Storage connection named Storage

All known issues and feature requests are tracked in a [GitHub issues list](#). If you run into a problem and can't find the issue in GitHub, open a new issue, and include a detailed description of the problem.

Next steps

For more information, see the following resources:

- [Azure Functions package API documentation](#)
- [Best practices for Azure Functions](#)
- [Azure Functions triggers and bindings](#)
- [Blob Storage bindings](#)
- [HTTP and webhook bindings](#)
- [Queue Storage bindings](#)
- [Timer triggers](#)

Having issues with using Python? Tell us what's going on. [↗](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Develop Python worker extensions for Azure Functions

Article • 04/25/2023

Azure Functions lets you integrate custom behaviors as part of Python function execution. This feature enables you to create business logic that customers can easily use in their own function apps. To learn more, see the [Python developer reference](#). Worker extensions are supported in both the v1 and v2 Python programming models.

In this tutorial, you'll learn how to:

- ✓ Create an application-level Python worker extension for Azure Functions.
- ✓ Consume your extension in an app the way your customers do.
- ✓ Package and publish an extension for consumption.

Prerequisites

Before you start, you must meet these requirements:

- [Python 3.7 or above](#). To check the full list of supported Python versions in Azure Functions, see the [Python developer guide](#).
- The [Azure Functions Core Tools](#), version 4.0.5095 or later, which supports using the extension with the [v2 Python programming model](#). Check your version with `func -version`.
- [Visual Studio Code](#) installed on one of the [supported platforms](#).

Create the Python Worker extension

The extension you create reports the elapsed time of an HTTP trigger invocation in the console logs and in the HTTP response body.

Folder structure

The folder for your extension project should be like the following structure:

```
<python_worker_extension_root>/  
| - .venv/
```

```
| - python_worker_extension_timer/
| | - __init__.py
| - setup.py
| - readme.md
```

Folder/file	Description
.venv/	(Optional) Contains a Python virtual environment used for local development.
python_worker_extension/	Contains the source code of the Python worker extension. This folder contains the main Python module to be published into PyPI.
setup.py	Contains the metadata of the Python worker extension package.
readme.md	Contains the instruction and usage of your extension. This content is displayed as the description in the home page in your PyPI project.

Configure project metadata

First you create `setup.py`, which provides essential information about your package. To make sure that your extension is distributed and integrated into your customer's function apps properly, confirm that `'azure-functions >= 1.7.0, < 2.0.0'` is in the `install_requires` section.

In the following template, you should change `author`, `author_email`, `install_requires`, `license`, `packages`, and `url` fields as needed.

Python

```
from setuptools import find_packages, setup
setup(
    name='python-worker-extension-timer',
    version='1.0.0',
    author='Your Name Here',
    author_email='your@email.here',
    classifiers=[
        'Intended Audience :: End Users/Desktop',
        'Development Status :: 5 - Production/Stable',
        'Intended Audience :: End Users/Desktop',
        'License :: OSI Approved :: Apache Software License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 3.7',
        'Programming Language :: Python :: 3.8',
        'Programming Language :: Python :: 3.9',
        'Programming Language :: Python :: 3.10',
    ],
    description='Python Worker Extension Demo',
    include_package_data=True,
```

```
long_description=open('readme.md').read(),
install_requires=[
    'azure-functions >= 1.7.0, < 2.0.0',
    # Any additional packages that will be used in your extension
],
extras_require={},
license='MIT',
packages=find_packages(where='.'),
url='https://your-github-or-pypi-link',
zip_safe=False,
)
```

Next, you'll implement your extension code in the application-level scope.

Implement the timer extension

Add the following code in `python_worker_extension_timer/__init__.py` to implement the application-level extension:

Python

```
import typing
from logging import Logger
from time import time
from azure.functions import AppExtensionBase, Context, HttpResponse
class TimerExtension(AppExtensionBase):
    """A Python worker extension to record elapsed time in a function
invocation
"""

@classmethod
def init(cls):
    # This records the starttime of each function
    cls.start_timestamps: typing.Dict[str, float] = {}

@classmethod
def configure(cls, *args, append_to_http_response:bool=False, **kwargs):
    # Customer can use
TimerExtension.configure(append_to_http_response=)
    # to decide whether the elapsed time should be shown in HTTP
response
    cls.append_to_http_response = append_to_http_response

@classmethod
def pre_invocation_app_level(
    cls, logger: Logger, context: Context,
    func_args: typing.Dict[str, object],
    *args, **kwargs
) -> None:
    logger.info(f'Recording start time of {context.function_name}')
    cls.start_timestamps[context.invocation_id] = time()
```

```

@classmethod
def post_invocation_app_level(
    cls, logger: Logger, context: Context,
    func_args: typing.Dict[str, object],
    func_ret: typing.Optional[object],
    *args, **kwargs
) -> None:
    if context.invocation_id in cls.start_timestamps:
        # Get the start_time of the invocation
        start_time: float =
    cls.start_timestamps.pop(context.invocation_id)
    end_time: float = time()
    # Calculate the elapsed time
    elapsed_time = end_time - start_time
    logger.info(f'Time taken to execute {context.function_name} is
{elapsed_time} sec')
    # Append the elapsed time to the end of HTTP response
    # if the append_to_http_response is set to True
    if cls.append_to_http_response and isinstance(func_ret,
    HttpResponse):
        func_ret._HttpResponse__body += f' (TimeElapsed:
{elapsed_time} sec)'.encode()

```

This code inherits from [AppExtensionBase](#) so that the extension applies to every function in the app. You could have also implemented the extension on a function-level scope by inheriting from [FuncExtensionBase](#).

The `init` method is a class method that's called by the worker when the extension class is imported. You can do initialization actions here for the extension. In this case, a hash map is initialized for recording the invocation start time for each function.

The `configure` method is customer-facing. In your readme file, you can tell your customers when they need to call `Extension.configure()`. The readme should also document the extension capabilities, possible configuration, and usage of your extension. In this example, customers can choose whether the elapsed time is reported in the `HttpResponse`.

The `pre_invocation_app_level` method is called by the Python worker before the function runs. It provides the information from the function, such as function context and arguments. In this example, the extension logs a message and records the start time of an invocation based on its `invocation_id`.

Similarly, the `post_invocation_app_level` is called after function execution. This example calculates the elapsed time based on the start time and current time. It also overwrites the return value of the HTTP response.

Create a readme.md

Create a `readme.md` file in the root of your extension project. This file contains the instructions and usage of your extension. The `readme.md` content is displayed as the description in the home page in your PyPI project.

markdown

```
# Python Worker Extension Timer
```

In this file, tell your customers when they need to call
``Extension.configure()``.

The `readme` should also document the extension capabilities, possible configuration, and usage of your extension.

Consume your extension locally

Now that you've created an extension, you can use it in an app project to verify it works as intended.

Create an HTTP trigger function

1. Create a new folder for your app project and navigate to it.
2. From the appropriate shell, such as Bash, run the following command to initialize the project:

Bash

```
func init --python
```

3. Use the following command to create a new HTTP trigger function that allows anonymous access:

Bash

```
func new -t HttpTrigger -n HttpTrigger -a anonymous
```

Activate a virtual environment

1. Create a Python virtual environment, based on OS as follows:

Windows

Console

```
py -m venv .venv
```

2. Activate the Python virtual environment, based on OS as follows:

Windows

Console

```
.venv\Scripts\Activate.ps1
```

Configure the extension

1. Install remote packages for your function app project using the following command:

Bash

```
pip install -r requirements.txt
```

2. Install the extension from your local file path, in editable mode as follows:

Bash

```
pip install -e <PYTHON_WORKER_EXTENSION_ROOT>
```

In this example, replace `<PYTHON_WORKER_EXTENSION_ROOT>` with the root file location of your extension project.

When a customer uses your extension, they'll instead add your extension package location to the requirements.txt file, as in the following examples:

PyPI

Python

```
# requirements.txt  
python_worker_extension_timer==1.0.0
```

3. Open the local.settings.json project file and add the following field to `Values`:

JSON

```
"PYTHON_ENABLE_WORKER_EXTENSIONS": "1"
```

When running in Azure, you instead add `PYTHON_ENABLE_WORKER_EXTENSIONS=1` to the [app settings in the function app](#).

4. Add following two lines before the `main` function in `_init.py_` file for the v1 programming model, or in the `function_app.py` file for the v2 programming model:

Python

```
from python_worker_extension_timer import TimerExtension  
TimerExtension.configure(append_to_http_response=True)
```

This code imports the `TimerExtension` module and sets the `append_to_http_response` configuration value.

Verify the extension

1. From your app project root folder, start the function host using `func host start -verbose`. You should see the local endpoint of your function in the output as `https://localhost:7071/api/HttpTrigger`.
2. In the browser, send a GET request to `https://localhost:7071/api/HttpTrigger`. You should see a response like the following, with the `TimeElapsed` data for the request appended.

```
This HTTP triggered function executed successfully. Pass a name in the query string or in the request body for a personalized response.  
(TimeElapsed: 0.0009996891021728516 sec)
```

Publish your extension

After you've created and verified your extension, you still need to complete these remaining publishing tasks:

- ✓ Choose a license.
- ✓ Create a `readme.md` and other documentation.
- ✓ Publish the extension library to a Python package registry or a version control system (VCS).

PyPI

To publish your extension to PyPI:

1. Run the following command to install `twine` and `wheel` in your default Python environment or a virtual environment:

Bash

```
pip install twine wheel
```

2. Remove the old `dist/` folder from your extension repository.

3. Run the following command to generate a new package inside `dist/`:

Bash

```
python setup.py sdist bdist_wheel
```

4. Run the following command to upload the package to PyPI:

Bash

```
twine upload dist/*
```

You may need to provide your PyPI account credentials during upload. You can also test your package upload with `twine upload -r testpypi dist/*`. For more information, see the [Twine documentation](#).

After these steps, customers can use your extension by including your package name in their `requirements.txt`.

For more information, see the [official Python packaging tutorial](#).

Examples

- You can view completed sample extension project from this article in the [python_worker_extension_timer](#) sample repository.
- OpenCensus integration is an open-source project that uses the extension interface to integrate telemetry tracing in Azure Functions Python apps. See the [opencensus-python-extensions-azure](#) repository to review the implementation of this Python worker extension.

Next steps

For more information about Azure Functions Python development, see the following resources:

- [Azure Functions Python developer guide](#)
- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)

Improve throughput performance of Python apps in Azure Functions

Article • 02/14/2023

When developing for Azure Functions using Python, you need to understand how your functions perform and how that performance affects the way your function app gets scaled. The need is more important when designing highly performant apps. The main factors to consider when designing, writing, and configuring your functions apps are horizontal scaling and throughput performance configurations.

Horizontal scaling

By default, Azure Functions automatically monitors the load on your application and creates more host instances for Python as needed. Azure Functions uses built-in thresholds for different trigger types to decide when to add instances, such as the age of messages and queue size for QueueTrigger. These thresholds aren't user configurable. For more information, see [Event-driven scaling in Azure Functions](#).

Improving throughput performance

The default configurations are suitable for most of Azure Functions applications. However, you can improve the performance of your applications' throughput by employing configurations based on your workload profile. The first step is to understand the type of workload that you're running.

Workload type	Function app characteristics	Examples
I/O-bound	<ul style="list-style-type: none">App needs to handle many concurrent invocations.App processes a large number of I/O events, such as network calls and disk read/writes.	<ul style="list-style-type: none">Web APIs
CPU-bound	<ul style="list-style-type: none">App does long-running computations, such as image resizing.App does data transformation.	<ul style="list-style-type: none">Data processingMachine learning inference

As real world function workloads are usually a mix of I/O and CPU bound, you should profile the app under realistic production loads.

Performance-specific configurations

After you understand the workload profile of your function app, the following are configurations that you can use to improve the throughput performance of your functions.

- [Async](#)
- [Multiple language worker](#)
- [Max workers within a language worker process](#)
- [Event loop](#)
- [Vertical Scaling](#)

Async

Because [Python is a single-threaded runtime](#), a host instance for Python can process only one function invocation at a time by default. For applications that process a large number of I/O events and/or is I/O bound, you can improve performance significantly by running functions asynchronously.

To run a function asynchronously, use the `async def` statement, which runs the function with [asyncio](#) directly:

Python

```
async def main():
    await some_nonblocking_socket_io_op()
```

Here's an example of a function with HTTP trigger that uses [aiohttp](#) http client:

Python

```
import aiohttp

import azure.functions as func

async def main(req: func.HttpRequest) -> func.HttpResponse:
    async with aiohttp.ClientSession() as client:
        async with client.get("PUT_YOUR_URL_HERE") as response:
            return func.HttpResponse(await response.text())

    return func.HttpResponse(body='NotFound', status_code=404)
```

A function without the `async` keyword is run automatically in a `ThreadPoolExecutor` thread pool:

Python

```
# Runs in a ThreadPoolExecutor threadpool. Number of threads is defined by
# PYTHON_THREADPOOL_THREAD_COUNT.
# The example is intended to show how default synchronous functions are
# handled.

def main():
    some_blocking_socket_io()
```

In order to achieve the full benefit of running functions asynchronously, the I/O operation/library that is used in your code needs to have `async` implemented as well. Using synchronous I/O operations in functions that are defined as asynchronous **may hurt** the overall performance. If the libraries you're using don't have `async` version implemented, you may still benefit from running your code asynchronously by [managing event loop](#) in your app.

Here are a few examples of client libraries that have implemented `async` patterns:

- [aiohttp](#) - Http client/server for `asyncio`
- [Streams API](#) - High-level `async/await`-ready primitives to work with network connection
- [Janus Queue](#) - Thread-safe `asyncio`-aware queue for Python
- [pyzmq](#) - Python bindings for ZeroMQ

Understanding `async` in Python worker

When you define `async` in front of a function signature, Python marks the function as a coroutine. When you call the coroutine, it can be scheduled as a task into an event loop. When you call `await` in an `async` function, it registers a continuation into the event loop, which allows the event loop to process the next task during the wait time.

In our Python Worker, the worker shares the event loop with the customer's `async` function and it's capable for handling multiple requests concurrently. We strongly encourage our customers to make use of `asyncio` compatible libraries, such as [aiohttp](#) and [pyzmq](#). Following these recommendations increases your function's throughput compared to those libraries when implemented synchronously.

ⓘ Note

If your function is declared as `async` without any `await` inside its implementation, the performance of your function will be severely impacted since the event loop will be blocked which prohibits the Python worker from handling concurrent requests.

Use multiple language worker processes

By default, every Functions host instance has a single language worker process. You can increase the number of worker processes per host (up to 10) by using the `FUNCTIONS_WORKER_PROCESS_COUNT` application setting. Azure Functions then tries to evenly distribute simultaneous function invocations across these workers.

For CPU bound apps, you should set the number of language workers to be the same as or higher than the number of cores that are available per function app. To learn more, see [Available instance SKUs](#).

I/O-bound apps may also benefit from increasing the number of worker processes beyond the number of cores available. Keep in mind that setting the number of workers too high can affect overall performance due to the increased number of required context switches.

The `FUNCTIONS_WORKER_PROCESS_COUNT` applies to each host that Azure Functions creates when scaling out your application to meet demand.

Note

Multiple Python workers are not supported by the Python v2 programming model at this time. This means that enabling intelligent concurrency and setting `FUNCTIONS_WORKER_PROCESS_COUNT` greater than 1 is not supported for functions developed using the v2 model.

Set up max workers within a language worker process

As mentioned in the async [section](#), the Python language worker treats functions and [coroutines](#) differently. A coroutine is run within the same event loop that the language worker runs on. On the other hand, a function invocation is run within a [ThreadPoolExecutor](#), which is maintained by the language worker as a thread.

You can set the value of maximum workers allowed for running sync functions using the `PYTHON_THREADPOOL_THREAD_COUNT` application setting. This value sets the `max_worker` argument of the `ThreadPoolExecutor` object, which lets Python use a pool of at most `max_worker` threads to execute calls asynchronously. The `PYTHON_THREADPOOL_THREAD_COUNT` applies to each worker that Functions host creates, and Python decides when to create a new thread or reuse the existing idle thread. For older Python versions(that is, 3.8, 3.7, and 3.6), `max_worker` value is set to 1. For Python version 3.9 , `max_worker` is set to `None`.

For CPU-bound apps, you should keep the setting to a low number, starting from 1 and increasing as you experiment with your workload. This suggestion is to reduce the time spent on context switches and allowing CPU-bound tasks to finish.

For I/O-bound apps, you should see substantial gains by increasing the number of threads working on each invocation. The recommendation is to start with the Python default (the number of cores) + 4 and then tweak based on the throughput values you're seeing.

For mixed workloads apps, you should balance both `FUNCTIONS_WORKER_PROCESS_COUNT` and `PYTHON_THREADPOOL_THREAD_COUNT` configurations to maximize the throughput. To understand what your function apps spend the most time on, we recommend profiling them and setting the values according to their behaviors. To learn about these application settings, see [Use multiple worker processes](#).

Note

Although these recommendations apply to both HTTP and non-HTTP triggered functions, you might need to adjust other trigger specific configurations for non-HTTP triggered functions to get the expected performance from your function apps. For more information about this, please refer to this [Best practices for reliable Azure Functions](#).

Managing event loop

You should use asyncio compatible third-party libraries. If none of the third-party libraries meet your needs, you can also manage the event loops in Azure Functions. Managing event loops give you more flexibility in compute resource management, and it also makes it possible to wrap synchronous I/O libraries into coroutines.

There are many useful Python official documents discussing the [Coroutines and Tasks ↗](#) and [Event Loop ↗](#) by using the built-in `asyncio` library.

Take the following `requests` library as an example, this code snippet uses the `asyncio` library to wrap the `requests.get()` method into a coroutine, running multiple web requests to SAMPLE_URL concurrently.

Python

```
import asyncio
import json
import logging
```

```

import azure.functions as func
from time import time
from requests import get, Response

async def invoke_get_request(eventloop: asyncio.AbstractEventLoop) ->
    Response:
    # Wrap requests.get function into a coroutine
    single_result = await eventloop.run_in_executor(
        None, # using the default executor
        get, # each task call invoke_get_request
        'SAMPLE_URL' # the url to be passed into the requests.get function
    )
    return single_result

async def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    eventloop = asyncio.get_event_loop()

    # Create 10 tasks for requests.get synchronous call
    tasks = [
        asyncio.create_task(
            invoke_get_request(eventloop)
        ) for _ in range(10)
    ]

    done_tasks, _ = await asyncio.wait(tasks)
    status_codes = [d.result().status_code for d in done_tasks]

    return func.HttpResponse(body=json.dumps(status_codes),
                            mimetype='application/json')

```

Vertical scaling

You might be able to get more processing units, especially in CPU-bound operation, by upgrading to premium plan with higher specifications. With higher processing units, you can adjust the number of worker processes count according to the number of cores available and achieve higher degree of parallelism.

Next steps

For more information about Azure Functions Python development, see the following resources:

- [Azure Functions Python developer guide](#)
- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)

Troubleshoot Python errors in Azure Functions

Article • 02/25/2024

This article provides information to help you troubleshoot errors with your Python functions in Azure Functions. This article supports both the v1 and v2 programming models. Choose the model you want to use from the selector at the top of the article.

ⓘ Note

The Python v2 programming model is only supported in the 4.x functions runtime. For more information, see [Azure Functions runtime versions overview](#).

Here are the troubleshooting sections for common issues in Python functions:

Specifically with the v2 model, here are some known issues and their workarounds:

- [Could not load file or assembly](#)
- [Unable to resolve the Azure Storage connection named Storage](#)

General troubleshooting guides for Python Functions include:

- [ModuleNotFoundError and ImportError](#)
- [Cannot import 'cygrpc'](#)
- [Python exited with code 137](#)
- [Python exited with code 139](#)
- [Sync triggers failed](#)
- [Development issues in the Azure portal](#)

Troubleshoot: ModuleNotFoundError

This section helps you troubleshoot module-related errors in your Python function app. These errors typically result in the following Azure Functions error message:

Exception: ModuleNotFoundError: No module named 'module_name'.

This error occurs when a Python function app fails to load a Python module. The root cause for this error is one of the following issues:

- [The package can't be found](#)

- The package isn't resolved with proper Linux wheel
- The package is incompatible with the Python interpreter version
- The package conflicts with other packages
- The package supports only Windows and macOS platforms

View project files

To identify the actual cause of your issue, you need to get the Python project files that run on your function app. If you don't have the project files on your local computer, you can get them in one of the following ways:

- If the function app has a `WEBSITE_RUN_FROM_PACKAGE` app setting and its value is a URL, download the file by copying and pasting the URL into your browser.
- If the function app has `WEBSITE_RUN_FROM_PACKAGE` set to `1`, go to `https://<app-name>.scm.azurewebsites.net/api/vfs/data/SitePackages` and download the file from the latest `href` URL.
- If the function app doesn't have either of the preceding app settings, go to `https://<app-name>.scm.azurewebsites.net/api/settings` and find the URL under `SCM_RUN_FROM_PACKAGE`. Download the file by copying and pasting the URL into your browser.
- If suggestions resolve the issue, go to `https://<app-name>.scm.azurewebsites.net/DebugConsole` and view the content under `/home/site/wwwroot`.

The rest of this article helps you troubleshoot potential causes of this error by inspecting your function app's content, identifying the root cause, and resolving the specific issue.

Diagnose ModuleNotFoundError

This section details potential root causes of module-related errors. After you figure out which is the likely root cause, you can go to the related mitigation.

The package can't be found

Go to `.python_packages/lib/python3.6/site-packages/<package-name>` or `.python_packages/lib/site-packages/<package-name>`. If the file path doesn't exist, this missing path is likely the root cause.

Using third-party or outdated tools during deployment might cause this issue.

To mitigate this issue, see [Enable remote build](#) or [Build native dependencies](#).

The package isn't resolved with the proper Linux wheel

Go to `.python_packages/lib/python3.6/site-packages/<package-name>-<version>-dist-info` or `.python_packages/lib/site-packages/<package-name>-<version>-dist-info`. Use your favorite text editor to open the *wheel* file and check the **Tag:** section. The issue might be that the tag value doesn't contain `linux`.

Python functions run only on Linux in Azure. The Functions runtime v2.x runs on Debian Stretch, and the v3.x runtime runs on Debian Buster. The artifact is expected to contain the correct Linux binaries. When you use the `--build local` flag in Core Tools, third-party, or outdated tools, it might cause older binaries to be used.

To mitigate the issue, see [Enable remote build](#) or [Build native dependencies](#).

The package is incompatible with the Python interpreter version

Go to `.python_packages/lib/python3.6/site-packages/<package-name>-<version>-dist-info` or `.python_packages/lib/site-packages/<package-name>-<version>-dist-info`. In your text editor, open the *METADATA* file and check the **Classifiers:** section. If the section doesn't contain `Python :: 3`, `Python :: 3.6`, `Python :: 3.7`, `Python :: 3.8`, or `Python :: 3.9`, the package version is either too old or, more likely, it's already out of maintenance.

You can check the Python version of your function app from the [Azure portal](#). Navigate to your function app's **Overview** resource page to find the runtime version. The runtime version supports Python versions as described in the [Azure Functions runtime versions overview](#).

To mitigate the issue, see [Update your package to the latest version](#) or [Replace the package with equivalents](#).

The package conflicts with other packages

If you've verified that the package is resolved correctly with the proper Linux wheels, there might be a conflict with other packages. In certain packages, the PyPi documentation might clarify the incompatible modules. For example, in [azure 4.0.0](#), you find the following statement:

This package isn't compatible with `azure-storage`. If you installed `azure-storage`, or if you installed `azure 1.x/2.x` and didn't uninstall `azure-storage`, you must uninstall `azure-storage` first.

You can find the documentation for your package version in <https://pypi.org/project/<package-name>/<package-version>>.

To mitigate the issue, see [Update your package to the latest version](#) or [Replace the package with equivalents](#).

The package supports only Windows and macOS platforms

Open the `requirements.txt` with a text editor and check the package in <https://pypi.org/project/<package-name>>. Some packages run only on Windows and macOS platforms. For example, `pywin32` runs on Windows only.

The `Module Not Found` error might not occur when you're using Windows or macOS for local development. However, the package fails to import on Azure Functions, which uses Linux at runtime. This issue is likely to be caused by using `pip freeze` to export the virtual environment into `requirements.txt` from your Windows or macOS machine during project initialization.

To mitigate the issue, see [Replace the package with equivalents](#) or [Handcraft requirements.txt](#).

Mitigate ModuleNotFoundError

The following are potential mitigations for module-related issues. Use the [previously mentioned diagnoses](#) to determine which of these mitigations to try.

Enable remote build

Make sure that remote build is enabled. The way that you make sure depends on your deployment method.

Visual Studio Code

Make sure that the latest version of the [Azure Functions extension for Visual Studio Code](#) is installed. Verify that the `.vscode/settings.json` file exists and it contains the setting `"azureFunctions.scmDoBuildDuringDeployment": true`. If it doesn't, create the file with the `azureFunctions.scmDoBuildDuringDeployment` setting enabled, and then redeploy the project.

Build native dependencies

Make sure that the latest versions of both Docker and [Azure Functions Core Tools](#) are installed. Go to your local function project folder, and use `func azure functionapp publish <app-name> --build-native-deps` for deployment.

Update your package to the latest version

In the latest package version of <https://pypi.org/project/<package-name>>, check the **Classifiers:** section. The package should be `OS Independent`, or compatible with `POSIX` or `POSIX :: Linux` in **Operating System**. Also, the programming language should contain: `Python :: 3`, `Python :: 3.6`, `Python :: 3.7`, `Python :: 3.8`, or `Python :: 3.9`.

If these package items are correct, you can update the package to the latest version by changing the line `<package-name>~=<latest-version>` in *requirements.txt*.

Handcraft requirements.txt

Some developers use `pip freeze > requirements.txt` to generate the list of Python packages for their developing environments. Although this convenience should work in most cases, there can be issues in cross-platform deployment scenarios, such as developing functions locally on Windows or macOS, but publishing to a function app, which runs on Linux. In this scenario, `pip freeze` can introduce unexpected operating system-specific dependencies or dependencies for your local development environment. These dependencies can break the Python function app when it's running on Linux.

The best practice is to check the import statement from each `.py` file in your project source code and then check in only the modules in the *requirements.txt* file. This practice guarantees that the resolution of packages can be handled properly on different operating systems.

Replace the package with equivalents

First, take a look into the latest version of the package in <https://pypi.org/project/<package-name>>. This package usually has its own GitHub page. Go to the **Issues** section on GitHub and search to see whether your issue has been fixed. If it has been fixed, update the package to the latest version.

Sometimes, the package might have been integrated into [Python Standard Library](#) (such as `pathlib`). If so, because we provide a certain Python distribution in Azure Functions (Python 3.6, Python 3.7, Python 3.8, and Python 3.9), the package in your *requirements.txt* file should be removed.

However, if you're finding that the issue hasn't been fixed, and you're on a deadline, we encourage you to do some research to find a similar package for your project. Usually, the Python community provides you with a wide variety of similar libraries that you can use.

Disable dependency isolation flag

Set the application setting `PYTHON_ISOLATE_WORKER_DEPENDENCIES` to a value of `0`.

Troubleshoot: cannot import 'cygrpc'

This section helps you troubleshoot 'cygrpc'-related errors in your Python function app. These errors typically result in the following Azure Functions error message:

Cannot import name 'cygrpc' from 'grpc._cython'

This error occurs when a Python function app fails to start with a proper Python interpreter. The root cause for this error is one of the following issues:

- [The Python interpreter mismatches OS architecture](#)
- [The Python interpreter isn't supported by Azure Functions Python Worker](#)

Diagnose the 'cygrpc' reference error

There are several possible causes for errors that reference `cygrpc`, which are detailed in this section.

The Python interpreter mismatches OS architecture

This mismatch is most likely caused by a 32-bit Python interpreter being installed on your 64-bit operating system.

If you're running on an x64 operating system, ensure that your Python version 3.6, 3.7, 3.8, or 3.9 interpreter is also on a 64-bit version.

You can check your Python interpreter bitness by running the following commands:

On Windows in PowerShell, run `py -c 'import platform; print(platform.architecture()[0])'`.

On a Unix-like shell, run `python3 -c 'import platform; print(platform.architecture()[0])'`.

If there's a mismatch between Python interpreter bitness and the operating system architecture, download a proper Python interpreter from [Python Software Foundation](#).

The Python interpreter isn't supported by Azure Functions Python Worker

The Azure Functions Python Worker supports only [specific Python versions](#).

Check to see whether your Python interpreter matches your expected version by `py --version` in Windows or `python3 --version` in Unix-like systems. Ensure that the return result is one of the [supported Python versions](#).

If your Python interpreter version doesn't meet the requirements for Azure Functions, instead download a Python interpreter version that is supported by Functions from the [Python Software Foundation](#).

Troubleshoot: python exited with code 137

Code 137 errors are typically caused by out-of-memory issues in your Python function app. As a result, you get the following Azure Functions error message:

```
Microsoft.Azure.WebJobs.Script.Workers.WorkerProcessExitException : python exited with code 137
```

This error occurs when a Python function app is forced to terminate by the operating system with a `SIGKILL` signal. This signal usually indicates an out-of-memory error in your Python process. The Azure Functions platform has a [service limitation](#) that terminates any function apps that exceed this limit.

To analyze the memory bottleneck in your function app, see [Profile Python function app in local development environment](#).

Troubleshoot: python exited with code 139

This section helps you troubleshoot segmentation fault errors in your Python function app. These errors typically result in the following Azure Functions error message:

Microsoft.Azure.WebJobs.Script.Workers.WorkerProcessExitException : python exited with code 139

This error occurs when a Python function app is forced to terminate by the operating system with a `SIGSEGV` signal. This signal indicates violation of the memory segmentation, which can result from an unexpected reading from or writing into a restricted memory region. In the following sections, we provide a list of common root causes.

A regression from third-party packages

In your function app's `requirements.txt` file, an unpinned package gets upgraded to the latest version during each deployment to Azure. Package updates can potentially introduce regressions that affect your app. To recover from such issues, comment out the import statements, disable the package references, or pin the package to a previous version in `requirements.txt`.

Unpickling from a malformed .pkl file

If your function app is using the Python pickle library to load a Python object from a `.pkl` file, it's possible that the file contains a malformed bytes string or an invalid address reference. To recover from this issue, try commenting out the `pickle.load()` function.

Pyodbc connection collision

If your function app is using the popular ODBC database driver [pyodbc](#), it's possible that multiple connections are open within a single function app. To avoid this issue, use the singleton pattern, and ensure that only one pyodbc connection is used across the function app.

Sync triggers failed

The error `Sync triggers failed` can be caused by several issues. One potential cause is a conflict between customer-defined dependencies and Python built-in modules when your functions run in an App Service plan. For more information, see [Package management](#).

Troubleshoot: could not load file or assembly

You can see this error when you're running locally using the v2 programming model. This error is caused by a known issue to be resolved in an upcoming release.

This is an example message for this error:

```
DurableTask.Netherite.AzureFunctions: Could not load file or assembly  
'Microsoft.Azure.WebJobs.Extensions.DurableTask, Version=2.0.0.0, Culture=neutral,  
PublicKeyToken=014045d636e89289'.  
The system cannot find the file specified.
```

The error occurs because of an issue with how the extension bundle was cached. To troubleshoot the issue, run this command with `--verbose` to see more details:

```
Console  
func host start --verbose
```

It's likely you're seeing this caching issue when you see an extension loading log like `Loading startup extension <>` that isn't followed by `Loaded extension <>`.

To resolve this issue:

1. Find the `.azure-functions-core-tools` path by running:

```
Console  
func GetExtensionBundlePath
```

2. Delete the `.azure-functions-core-tools` directory.

```
Bash  
Bash  
rm -r <insert path>/ .azure-functions-core-tools
```

The cache directory is recreated when you run Core Tools again.

Troubleshoot: unable to resolve the Azure Storage connection

You might see this error in your local output as the following message:

```
Microsoft.Azure.WebJobs.Extensions.DurableTask: Unable to resolve the Azure  
Storage connection named 'Storage'.  
Value cannot be null. (Parameter 'provider')
```

This error is a result of how extensions are loaded from the bundle locally. To resolve this error, take one of the following actions:

- Use a storage emulator such as [Azurite](#). This option is a good one when you aren't planning to use a storage account in your function application.
- Create a storage account and add a connection string to the `AzureWebJobsStorage` environment variable in the `localsettings.json` file. Use this option when you're using a storage account trigger or binding with your application, or if you have an existing storage account. To get started, see [Create a storage account](#).

Development issues in the Azure portal

When using the [Azure portal](#), take into account these known issues and their workarounds:

- There are general limitations for writing your function code in the portal. For more information, see [Development limitations in the Azure portal](#).
- To delete a function from a function app in the portal, remove the function code from the file itself. The **Delete** button doesn't work to remove the function when using the Python v2 programming model.
- When creating a function in the portal, you might be admonished to use a different tool for development. There are several scenarios where you can't edit your code in the portal, including when a syntax error has been detected. In these scenarios, use [Visual Studio Code](#) or [Azure Functions Core Tools](#) to develop and publish your function code.

Next steps

If you're unable to resolve your issue, contact the Azure Functions team:

[Report an unresolved issue](#)

Profile Python apps memory usage in Azure Functions

Article • 04/27/2023

During development or after deploying your local Python function app project to Azure, it's a good practice to analyze for potential memory bottlenecks in your functions. Such bottlenecks can decrease the performance of your functions and lead to errors. The following instructions show you how to use the [memory-profiler](#) Python package, which provides line-by-line memory consumption analysis of your functions as they execute.

ⓘ Note

Memory profiling is intended only for memory footprint analysis in development environments. Please do not apply the memory profiler on production function apps.

Prerequisites

Before you start developing a Python function app, you must meet these requirements:

- [Python 3.7 or above](#). To check the full list of supported Python versions in Azure Functions, see the [Python developer guide](#).
- The [Azure Functions Core Tools](#), version 4.x or greater. Check your version with `func --version`. To learn about updating, see [Azure Functions Core Tools on GitHub](#).
- [Visual Studio Code](#) installed on one of the [supported platforms](#).
- An active Azure subscription.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Memory profiling process

1. In your requirements.txt, add `memory-profiler` to ensure the package is bundled with your deployment. If you're developing on your local machine, you may want

to activate a Python virtual environment and do a package resolution by `pip`

```
install -r requirements.txt.
```

2. In your function script (for example, `__init__.py` for the Python v1 programming model and `function_app.py` for the v2 model), add the following lines above the `main()` function. These lines ensure the root logger reports the child logger names, so that the memory profiling logs are distinguishable by the prefix `memory_profiler_logs`.

Python

```
import logging
import memory_profiler
root_logger = logging.getLogger()
root_logger.handlers[0].setFormatter(logging.Formatter("%(name)s: %(message)s"))
profiler_logstream = memory_profiler.LogFile('memory_profiler_logs',
True)
```

3. Apply the following decorator above any functions that need memory profiling. The decorator doesn't work directly on the trigger entrypoint `main()` method. You need to create subfunctions and decorate them. Also, due to a memory-profiler known issue, when applying to an async coroutine, the coroutine return value is always `None`.

Python

```
@memory_profiler.profile(stream=profiler_logstream)
```

4. Test the memory profiler on your local machine by using Azure Functions Core Tools command `func host start`. When you invoke the functions, they should generate a memory usage report. The report contains file name, line of code, memory usage, memory increment, and the line content in it.
5. To check the memory profiling logs on an existing function app instance in Azure, you can query the memory profiling logs for recent invocations with [Kusto](#) queries in Application Insights, Logs.

The screenshot shows the Azure Application Insights Logs blade. On the left, there's a sidebar with various monitoring and diagnostic settings, including a red box around the 'Logs' option. The main area has a search bar and a 'New Query' button. A query editor window is open with the following Kusto query:

```

traces
| where timestamp > ago(1d)
| where message startswith_cs "memory_profiler_logs:"
| parse message with "memory_profiler_logs: LineNumber" " TotalMem_MiB" " IncreMem_MiB" " Occurrences" " Contents"
| union (
    traces
    | where timestamp > ago(1d)
    | where message startswith_cs "memory_profiler_logs: Filename:" 
    | parse message with "memory_profiler_logs: Filename:" "FileName"
    | project timestamp, FileName, itemId
)
| project timestamp, LineNumber=iff(FileName != "", FileName, LineNumber), TotalMem_MiB, IncreMem_MiB, Occurrences, Contents, RequestId=itemId
| order by timestamp asc

```

Below the query editor is a results table with columns: timestamp [UTC], LineNumber, TotalMem_MiB, IncreMem_MiB, Occurrences, and Contents. The table shows two rows of data, each with a timestamp of 3/26/2021, 12:24:22.23 AM, and two different LineNumbers (23 and 19) with their corresponding memory usage and occurrence counts.

Kusto

```

traces
| where timestamp > ago(1d)
| where message startswith_cs "memory_profiler_logs:"
| parse message with "memory_profiler_logs: LineNumber" " TotalMem_MiB" " IncreMem_MiB" " Occurrences" " Contents"
| union (
    traces
    | where timestamp > ago(1d)
    | where message startswith_cs "memory_profiler_logs: Filename:" 
    | parse message with "memory_profiler_logs: Filename:" "FileName"
    | project timestamp, FileName, itemId
)
| project timestamp, LineNumber=iff(FileName != "", FileName, LineNumber), TotalMem_MiB, IncreMem_MiB, Occurrences, Contents, RequestId=itemId
| order by timestamp asc

```

Example

Here's an example of performing memory profiling on an asynchronous and a synchronous HTTP trigger, named "HttpTriggerAsync" and "HttpTriggerSync" respectively. We'll build a Python function app that simply sends out GET requests to the Microsoft's home page.

Create a Python function app

A Python function app should follow Azure Functions specified [folder structure](#). To scaffold the project, we recommend using the Azure Functions Core Tools by running the following commands:



Bash

```
func init PythonMemoryProfilingDemo --python
cd PythonMemoryProfilingDemo
func new -l python -t HttpTrigger -n HttpTriggerAsync -a anonymous
func new -l python -t HttpTrigger -n HttpTriggerSync -a anonymous
```

Update file contents

The `requirements.txt` defines the packages that are used in our project. Besides the Azure Functions SDK and memory-profiler, we introduce `aiohttp` for asynchronous HTTP requests and `requests` for synchronous HTTP calls.

text

```
# requirements.txt

azure-functions
memory-profiler
aiohttp
requests
```

Create the asynchronous HTTP trigger.

v1

Replace the code in the asynchronous HTTP trigger `HttpTriggerAsync/_init_.py` with the following code, which configures the memory profiler, root logger format, and logger streaming binding.

Python

```
# HttpTriggerAsync/_init__.py

import azure.functions as func
import aiohttp
import logging
import memory_profiler

# Update root logger's format to include the logger name. Ensure logs
generated
# from memory profiler can be filtered by "memory_profiler_logs" prefix.
root_logger = logging.getLogger()
root_logger.handlers[0].setFormatter(logging.Formatter("%(name)s: %
(message)s"))
```

```
profiler_logstream = memory_profiler.LogFile('memory_profiler_logs',  
True)  
  
async def main(req: func.HttpRequest) -> func.HttpResponse:  
    await get_microsoft_page_async('https://microsoft.com')  
    return func.HttpResponse(  
        f"Microsoft page loaded.",  
        status_code=200  
    )  
  
@memory_profiler.profile(stream=profiler_logstream)  
async def get_microsoft_page_async(url: str):  
    async with aiohttp.ClientSession() as client:  
        async with client.get(url) as response:  
            await response.text()  
    # @memory_profiler.profile does not support return for coroutines.  
    # All returns become None in the parent functions.  
    # GitHub Issue:  
    # https://github.com/pythonprofilers/memory_profiler/issues/289
```

Create the synchronous HTTP trigger.

v1

Replace the code in the asynchronous HTTP trigger *HttpTriggerSync/_init__.py* with the following code.

Python

```
# HttpTriggerSync/_init__.py  
  
import azure.functions as func  
import requests  
import logging  
import memory_profiler  
  
# Update root logger's format to include the logger name. Ensure logs  
generated  
# from memory profiler can be filtered by "memory_profiler_logs" prefix.  
root_logger = logging.getLogger()  
root_logger.handlers[0].setFormatter(logging.Formatter("%(name)s: %  
(message)s"))  
profiler_logstream = memory_profiler.LogFile('memory_profiler_logs',  
True)  
  
def main(req: func.HttpRequest) -> func.HttpResponse:  
    content = profile_get_request('https://microsoft.com')  
    return func.HttpResponse(  
        f"Microsoft page response size: {len(content)}",  
        status_code=200  
    )
```

```
@memory_profiler.profile(stream=profiler_logstream)
def profile_get_request(url: str):
    response = requests.get(url)
    return response.content
```

Profile Python function app in local development environment

After you make the above changes, there are a few more steps to initialize a Python virtual environment for Azure Functions runtime.

1. Open a Windows PowerShell or any Linux shell as you prefer.
2. Create a Python virtual environment by `py -m venv .venv` in Windows, or `python3 -m venv .venv` in Linux.
3. Activate the Python virtual environment with `.venv\Scripts\Activate.ps1` in Windows PowerShell or `source .venv/bin/activate` in Linux shell.
4. Restore the Python dependencies with `pip install -r requirements.txt`
5. Start the Azure Functions runtime locally with Azure Functions Core Tools `func host start`
6. Send a GET request to `https://localhost:7071/api/HttpTriggerAsync` or `https://localhost:7071/api/HttpTriggerSync`.
7. It should show a memory profiling report similar to the following section in Azure Functions Core Tools.

text

```
Filename: <ProjectRoot>\HttpTriggerAsync\__init__.py
Line #      Mem usage     Increment  Occurrences   Line Contents
=====
19          45.1 MiB       45.1 MiB           1   @memory_profiler.profile
20                                async def
get_microsoft_page_async(url: str):
21          45.1 MiB       0.0 MiB           1       async with
aiohttp.ClientSession() as client:
22          46.6 MiB       1.5 MiB          10       async with
client.get(url) as response:
23          47.6 MiB       1.0 MiB           4       await
response.text()
```

Next steps

For more information about Azure Functions Python development, see the following resources:

- [Azure Functions Python developer guide](#)
- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)

functions Package

Reference

Modules

[\[\] Expand table](#)

blob
cosmosdb
durable_functions
eventgrid
eventhub
http
kafka
meta
queue
servicebus
timer

Classes

[\[\] Expand table](#)

Context	Function invocation context.
Document	An Azure Document. Document objects are <code>UserDict</code> subclasses and behave like dicts.
DocumentList	A <code>UserList</code> subclass containing a list of Document objects
EntityContext	A durable function entity context.
EventGridEvent	An EventGrid event message.
EventGridOutputEvent	An EventGrid event message.

EventHubEvent	A concrete implementation of Event Hub message type.
HttpRequest	An HTTP request object.
HttpResponse	An HTTP response object.
InputStream	File-like object representing an input blob.
KafkaConverter	
KafkaEvent	A concrete implementation of Kafka event message type.
KafkaTriggerConverter	
OrchestrationContext	A durable function orchestration context.
Out	An interface to set function output parameters.
QueueMessage	A Queue message object.
ServiceBusMessage	
TimerRequest	Timer request object.
WsgiMiddleware	

Functions

get_binding_registry

```
get_binding_registry()
```

Azure Functions developer guide

Article • 06/26/2024

In Azure Functions, all functions share some core technical concepts and components, regardless of your preferred language or development environment. This article is language-specific. Choose your preferred language at the top of the article.

This article assumes that you've already read the [Azure Functions overview](#).

If you prefer to jump right in, you can complete a quickstart tutorial using [Visual Studio](#), [Visual Studio Code](#), or from the [command prompt](#).

Code project

At the core of Azure Functions is a language-specific code project that implements one or more units of code execution called *functions*. Functions are simply methods that run in the Azure cloud based on events, in response to HTTP requests, or on a schedule. Think of your Azure Functions code project as a mechanism for organizing, deploying, and collectively managing your individual functions in the project when they're running in Azure. For more information, see [Organize your functions](#).

The way that you lay out your code project and how you indicate which methods in your project are functions depends on the development language of your project. For detailed language-specific guidance, see the [C# developers guide](#).

All functions must have a trigger, which defines how the function starts and can provide input to the function. Your functions can optionally define input and output bindings. These bindings simplify connections to other services without you having to work with client SDKs. For more information, see [Azure Functions triggers and bindings concepts](#).

Azure Functions provides a set of language-specific project and function templates that make it easy to create new code projects and add functions to your project. You can use any of the tools that support Azure Functions development to generate new apps and functions using these templates.

Development tools

The following tools provide an integrated development and publishing experience for Azure Functions in your preferred language:

- [Visual Studio](#)
- [Visual Studio Code](#)
- [Azure Functions Core Tools](#) (command prompt)

These tools integrate with [Azure Functions Core Tools](#) so that you can run and debug on your local computer using the Functions runtime. For more information, see [Code and test Azure Functions locally](#).

Deployment

When you publish your code project to Azure, you're essentially deploying your project to an existing function app resource. A function app provides an execution context in Azure in which your functions run. As such, it's the unit of deployment and management for your functions. From an Azure Resource perspective, a function app is equivalent to a site resource (`Microsoft.Web/sites`) in Azure App Service, which is equivalent to a web app.

A function app is composed of one or more individual functions that are managed, deployed, and scaled together. All of the functions in a function app share the same [pricing plan](#), [deployment method](#), and [runtime version](#). For more information, see [How to manage a function app](#).

When the function app and any other required resources don't already exist in Azure, you first need to create these resources before you can deploy your project files. You can create these resources in one of these ways:

- During [Visual Studio](#) publishing
- Using [Visual Studio Code](#)
- Programmatically using [Azure CLI](#), [Azure PowerShell](#), [ARM templates](#), or [Bicep templates](#)
- In the [Azure portal](#)

In addition to tool-based publishing, Functions supports other technologies for deploying source code to an existing function app. For more information, see [Deployment technologies in Azure Functions](#).

Connect to services

A major requirement of any cloud-based compute service is reading data from and writing data to other cloud services. Functions provides an extensive set of bindings that makes it easier for you to connect to services without having to work with client SDKs.

Whether you use the binding extensions provided by Functions or you work with client SDKs directly, you securely store connection data and do not include it in your code. For more information, see [Connections](#).

Bindings

Functions provides bindings for many Azure services and a few third-party services, which are implemented as extensions. For more information, see the [complete list of supported bindings](#).

Binding extensions can support both inputs and outputs, and many triggers also act as input bindings. Bindings let you configure the connection to services so that the Functions host can handle the data access for you. For more information, see [Azure Functions triggers and bindings concepts](#).

If you're having issues with errors coming from bindings, see the [Azure Functions Binding Error Codes](#) documentation.

Client SDKs

While Functions provides bindings to simplify data access in your function code, you're still able to use a client SDK in your project to directly access a given service, if you prefer. You might need to use client SDKs directly should your functions require a functionality of the underlying SDK that's not supported by the binding extension.

When using client SDKs, you should use the same process for [storing and accessing connection strings](#) used by binding extensions.

When you create a client SDK instance in your functions, you should get the connection info required by the client from [Environment variables](#).

Connections

As a security best practice, Azure Functions takes advantage of the application settings functionality of Azure App Service to help you more securely store strings, keys, and other tokens required to connect to other services. Application settings in Azure are stored encrypted and can be accessed at runtime by your app as environment variable `name value` pairs. For triggers and bindings that require a connection property, you set the application setting name instead of the actual connection string. You can't configure a binding directly with a connection string or key.

For example, consider a trigger definition that has a `connection` property. Instead of the connection string, you set `connection` to the name of an environment variable that contains the connection string. Using this secrets access strategy both makes your apps more secure and makes it easier for you to change connections across environments. For even more security, you can use identity-based connections.

The default configuration provider uses environment variables. These variables are defined in [application settings](#) when running in the Azure and in the [local settings file](#) when developing locally.

Connection values

When the connection name resolves to a single exact value, the runtime identifies the value as a *connection string*, which typically includes a secret. The details of a connection string depend on the service to which you connect.

However, a connection name can also refer to a collection of multiple configuration items, useful for configuring [identity-based connections](#). Environment variables can be treated as a collection by using a shared prefix that ends in double underscores `__`. The group can then be referenced by setting the connection name to this prefix.

For example, the `connection` property for an Azure Blob trigger definition might be `Storage1`. As long as there's no single string value configured by an environment variable named `Storage1`, an environment variable named `Storage1__blobServiceUri` could be used to inform the `blobServiceUri` property of the connection. The connection properties are different for each service. Refer to the documentation for the component that uses the connection.

 **Note**

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `__` to ensure names are resolved correctly.

For example, `Storage1:blobServiceUri`.

Configure an identity-based connection

Some connections in Azure Functions can be configured to use an identity instead of a secret. Support depends on the extension using the connection. In some cases, a connection string may still be required in Functions even though the service to which you're connecting supports identity-based connections. For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

 **Note**

When running in a Consumption or Elastic Premium plan, your app uses the [`WEBSITE AZUREFILESCONNECTIONSTRING`](#) and [`WEBSITE CONTENTSHARE`](#) settings when connecting to Azure Files on the storage account used by your function app. Azure Files doesn't support using managed identity when accessing the file share. For more information, see [Azure Files supported authentication scenarios](#)

The following components support identity-based connections:

 Expand table

Connection source	Plans supported	Learn more
Azure Blobs triggers and bindings	All	Azure Blobs extension version 5.0.0 or later, Extension bundle 3.3.0 or later
Azure Queues triggers and bindings	All	Azure Queues extension version 5.0.0 or later, Extension bundle 3.3.0 or later
Azure Tables (when using Azure Storage)	All	Azure Tables extension version 1.0.0 or later, Extension bundle 3.3.0 or later
Azure SQL Database	All	Connect a function app to Azure SQL with managed identity and SQL bindings
Azure Event Hubs triggers and bindings	All	Azure Event Hubs extension version 5.0.0 or later, Extension bundle 3.3.0 or later
Azure Service Bus triggers and bindings	All	Azure Service Bus extension version 5.0.0 or later, Extension bundle 3.3.0 or later
Azure Event Grid output binding	All	Azure Event Grid extension version 3.3.0 or later, Extension bundle 3.3.0 or later
Azure Cosmos DB triggers and bindings	All	Azure Cosmos DB extension version 4.0.0 or later, Extension bundle 4.0.2 or later
Azure SignalR triggers and bindings	All	Azure SignalR extension version 1.7.0 or later Extension bundle 3.6.1 or later
Durable Functions storage provider (Azure Storage)	All	Durable Functions extension version 2.7.0 or later, Extension bundle 3.3.0 or later
Host-required storage ("AzureWebJobsStorage")	All	Connecting to host storage with an identity

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or

custom roles which provide those permissions.

ⓘ Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

Choose one of these tabs to learn about permissions for each component:

Azure Blobs extension

You need to create a role assignment that provides access to your blob container at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Blob Storage extension in normal operation. Your application may require further permissions based on the code you write.

 Expand table

Binding type	Example built-in roles
Trigger	Storage Blob Data Owner and Storage Queue Data Contributor ¹ Extra permissions must also be granted to the AzureWebJobsStorage connection. ²
Input binding	Storage Blob Data Reader
Output binding	Storage Blob Data Owner

¹ The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue on the storage account specified by the connection.

² The AzureWebJobsStorage connection is used internally for blobs and queues that enable the trigger. If it's configured to use an identity-based connection, it needs extra permissions beyond the default requirement. The required permissions are covered by the [Storage Blob Data Owner](#), [Storage Queue Data Contributor](#), and [Storage Account Contributor](#) roles. To learn more, see [Connecting to host storage with an identity](#).

Common properties for identity-based connections

An identity-based connection for an Azure service accepts the following common properties, where <CONNECTION_NAME_PREFIX> is the value of your `connection` property in the trigger or binding definition:

[] [Expand table](#)

Property	Environment variable template	Description
Token Credential	<CONNECTION_NAME_PREFIX>__credential	Defines how a token should be obtained for the connection. This setting should be set to <code>managedidentity</code> if your deployed Azure Function intends to use managed identity authentication. This value is only valid when a managed identity is available in the hosting environment.
Client ID	<CONNECTION_NAME_PREFIX>__clientId	When <code>credential</code> is set to <code>managedidentity</code> , this property can be set to specify the user-assigned identity to be used when obtaining a token. The property accepts a client ID corresponding to a user-assigned identity assigned to the application. It's invalid to specify both a Resource ID and a client ID. If not specified, the system-assigned identity is used. This property is used differently in local development scenarios , when <code>credential</code> shouldn't be set.
Resource ID	<CONNECTION_NAME_PREFIX>__managedIdentityResourceId	When <code>credential</code> is set to <code>managedidentity</code> , this property can be set to specify the resource Identifier to be used when obtaining a token. The property accepts a resource identifier corresponding to the resource ID of the user-defined managed identity. It's invalid to specify both a resource ID and a client ID. If neither are specified, the system-assigned identity is used. This property is used differently in local development scenarios , when <code>credential</code> shouldn't be set.

Other options may be supported for a given connection type. Refer to the documentation for the component making the connection.

Local development with identity-based connections

ⓘ Note

Local development with identity-based connections requires version 4.0.3904 of [Azure Functions Core Tools](#), or a later version.

When you're running your function project locally, the above configuration tells the runtime to use your local developer identity. The connection attempts to get a token from the following locations, in order:

- A local cache shared between Microsoft applications
- The current user context in Visual Studio
- The current user context in Visual Studio Code
- The current user context in the Azure CLI

If none of these options are successful, an error occurs.

Your identity may already have some role assignments against Azure resources used for development, but those roles may not provide the necessary data access. Management roles like [Owner](#) aren't sufficient. Double-check what permissions are required for connections for each component, and make sure that you have them assigned to yourself.

In some cases, you may wish to specify use of a different identity. You can add configuration properties for the connection that point to the alternate identity based on a client ID and client Secret for a Microsoft Entra service principal. **This configuration option is not supported when hosted in the Azure Functions service.** To use an ID and secret on your local machine, define the connection with the following extra properties:

[] [Expand table](#)

Property	Environment variable template	Description
Tenant ID	<CONNECTION_NAME_PREFIX>__tenantId	The Microsoft Entra tenant (directory) ID.
Client ID	<CONNECTION_NAME_PREFIX>__clientId	The client (application) ID of an app registration in the tenant.
Client secret	<CONNECTION_NAME_PREFIX>__clientSecret	A client secret that was generated for the app registration.

Here's an example of `local.settings.json` properties required for identity-based connection to Azure Blobs:

JSON

```
{  
    "IsEncrypted": false,
```

```

    "Values": {
        "<CONNECTION_NAME_PREFIX>__blobServiceUri": "<blobServiceUri>",
        "<CONNECTION_NAME_PREFIX>__queueServiceUri": "<queueServiceUri>",
        "<CONNECTION_NAME_PREFIX>__tenantId": "<tenantId>",
        "<CONNECTION_NAME_PREFIX>__clientId": "<clientId>",
        "<CONNECTION_NAME_PREFIX>__clientSecret": "<clientSecret>"
    }
}

```

Connecting to host storage with an identity

The Azure Functions host uses the storage connection set in [AzureWebJobsStorage](#) to enable core behaviors such as coordinating singleton execution of timer triggers and default app key storage. This connection can also be configured to use an identity.

Caution

Other components in Functions rely on [AzureWebJobsStorage](#) for default behaviors. You should not move it to an identity-based connection if you are using older versions of extensions that do not support this type of connection, including triggers and bindings for Azure Blobs, Event Hubs, and Durable Functions. Similarly, [AzureWebJobsStorage](#) is used for deployment artifacts when using server-side build in Linux Consumption, and if you enable this, you will need to deploy via [an external deployment package](#).

In addition, your function app might be reusing [AzureWebJobsStorage](#) for other storage connections in their triggers, bindings, and/or function code. Make sure that all uses of [AzureWebJobsStorage](#) are able to use the identity-based connection format before changing this connection from a connection string.

To use an identity-based connection for [AzureWebJobsStorage](#), configure the following app settings:

 [Expand table](#)

Setting	Description	Example value
<code>AzureWebJobsStorage__blobServiceUri</code>	The data plane URI of the blob service of the storage account, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>
<code>AzureWebJobsStorage__queueServiceUri</code>	The data plane URI	<code>https://<storage_account_name>.queue.core.windows.net</code>

Setting	Description	Example value
	of the queue service of the storage account, using the HTTPS scheme.	
AzureWebJobsStorage__tableServiceUri	The data plane URI of a table service of the storage account, using the HTTPS scheme.	https://<storage_account_name>.table.core.windows.net

Common properties for identity-based connections may also be set as well.

If you're configuring `AzureWebJobsStorage` using a storage account that uses the default DNS suffix and service name for global Azure, following the `https://<accountName>.`

`[blob|queue|file|table].core.windows.net` format, you can instead set

`AzureWebJobsStorage__accountName` to the name of your storage account. The endpoints for each storage service are inferred for this account. This doesn't work when the storage account is in a sovereign cloud or has a custom DNS.

[] [Expand table](#)

Setting	Description	Example value
<code>AzureWebJobsStorage__accountName</code>	The account name of a storage account, valid only if the account isn't in a sovereign cloud and doesn't have a custom DNS. This syntax is unique to <code>AzureWebJobsStorage</code> and can't be used for other identity-based connections.	<storage_account_name>

You will need to create a role assignment that provides access to the storage account for "AzureWebJobsStorage" at runtime. Management roles like `Owner` are not sufficient. The `Storage Blob Data Owner` role covers the basic needs of Functions host storage - the runtime needs both read and write access to blobs and the ability to create containers. Several extensions use this connection as a default location for blobs, queues, and tables, and these uses may add requirements as noted in the table below. You may need additional permissions if you use "AzureWebJobsStorage" for any other purposes.

[\[\]](#) Expand table

Extension	Roles required	Explanation
No extension (host only)	Storage Blob Data Owner	Used for general coordination, default key store
Azure Blobs (trigger only)	All of: Storage Account Contributor, Storage Blob Data Owner, Storage Queue Data Contributor	The blob trigger internally uses Azure Queues and writes blob receipts . It uses AzureWebJobsStorage for these, regardless of the connection configured for the trigger.
Azure Event Hubs (trigger only)	(no change from default requirement) Storage Blob Data Owner	Checkpoints are persisted in blobs using the AzureWebJobsStorage connection.
Timer trigger	(no change from default requirement) Storage Blob Data Owner	To ensure one execution per event, locks are taken with blobs using the AzureWebJobsStorage connection.
Durable Functions	All of: Storage Blob Data Contributor, Storage Queue Data Contributor, Storage Table Data Contributor	Durable Functions uses blobs, queues, and tables to coordinate activity functions and maintain orchestration state. It uses the AzureWebJobsStorage connection for all of these by default, but you can specify a different connection in the Durable Functions extension configuration .

Reporting Issues

[\[\]](#) Expand table

Item	Description	Link
Runtime	Script Host, Triggers & Bindings, Language Support	File an Issue ↗
Templates	Code Issues with Creation Template	File an Issue ↗
Portal	User Interface or Experience Issue	File an Issue ↗

Open source repositories

The code for Azure Functions is open source, and you can find key components in these GitHub repositories:

- [Azure Functions ↗](#)
- [Azure Functions host ↗](#)
- [Azure Functions portal ↗](#)
- [Azure Functions templates ↗](#)
- [Azure WebJobs SDK ↗](#)
- [Azure WebJobs SDK Extensions ↗](#)
- [Azure Functions .NET worker \(isolated process\) ↗](#)

Next steps

For more information, see the following resources:

- [Azure Functions scenarios](#)
- [Code and test Azure Functions locally](#)
- [Best Practices for Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Code and test Azure Functions locally

Article • 08/21/2024

While you're able to develop and test Azure Functions in the [Azure portal](#), many developers prefer a local development experience. When you use Functions, using your favorite code editor and development tools to create and test functions on your local computer becomes easier. Your local functions can connect to live Azure services, and you can debug them on your local computer using the full Functions runtime.

This article provides links to specific development environments for your preferred language. It also provides some shared guidance for local development, such as working with the [local.settings.json file](#).

Local development environments

The way in which you develop functions on your local computer depends on your [language](#) and tooling preferences. The environments in the following table support local development:

[Expand table](#)

Environment	Languages	Description
Visual Studio Code	C# (in-process) C# (isolated worker process) JavaScript PowerShell Python	The Azure Functions extension for VS Code adds Functions support to VS Code. Requires the Core Tools. Supports development on Linux, macOS, and Windows, when using version 2.x of the Core Tools. To learn more, see Create your first function using Visual Studio Code .
Command prompt or terminal	C# (in-process) C# (isolated worker process) JavaScript PowerShell Python	Azure Functions Core Tools provides the core runtime and templates for creating functions, which enable local development. Version 2.x supports development on Linux, macOS, and Windows. All environments rely on Core Tools for the local Functions runtime.
Visual Studio	C# (in-process) C# (isolated	The Azure Functions tools are included in the Azure development workload of Visual Studio , starting with Visual Studio 2019. Lets you compile functions in a class library and publish the .dll to Azure. Includes the Core Tools for local

Environment	Languages	Description
	worker process	testing. To learn more, see Develop Azure Functions using Visual Studio .
Maven (various)	Java	Maven archetype supports Core Tools to enable development of Java functions. Version 2.x supports development on Linux, macOS, and Windows. To learn more, see Create your first function with Java and Maven . Also supports development using Eclipse and IntelliJ IDEA .

ⓘ Note

Because of limitations on editing function code in the [Azure portal](#), you should develop your functions locally and publish your code project to a function app in Azure. For more information, see [Development limitations in the Azure portal](#)

Each of these local development environments lets you create function app projects and use predefined function templates to create new functions. Each uses the Core Tools so that you can test and debug your functions against the real Functions runtime on your own machine just as you would any other app. You can also publish your function app project from any of these environments to Azure.

Local project files

A Functions project directory contains the following files in the project root folder, regardless of language:

[\[+\] Expand table](#)

File name	Description
host.json	To learn more, see the host.json reference .
local.settings.json	Settings used by Core Tools when running locally, including app settings. To learn more, see local settings file .
.gitignore	Prevents the local.settings.json file from being accidentally published to a Git repository. To learn more, see local settings file .
.vscode\extensions.json	Settings file used when opening the project folder in Visual Studio Code.

Other files in the project depend on your language and specific functions. For more information, see the developer guide for your language.

Local settings file

The local.settings.json file stores app settings and settings used by local development tools. Settings in the local.settings.json file are used only when you're running your project locally. When you publish your project to Azure, be sure to also add any required settings to the app settings for the function app.

ⓘ Important

Because the local.settings.json may contain secrets, such as connection strings, you should never store it in a remote repository. Tools that support Functions provide ways to synchronize settings in the local.settings.json file with the [app settings](#) in the function app to which your project is deployed.

The local settings file has this structure:

```
JSON

{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "<language worker>",
    "AzureWebJobsStorage": "<connection-string>",
    "MyBindingConnection": "<binding-connection-string>",
    "AzureWebJobs.HttpExample.Disabled": "true"
  },
  "Host": {
    "LocalHttpPort": 7071,
    "CORS": "*",
    "CORSCredentials": false
  },
  "ConnectionStrings": {
    "SQLConnectionString": "<sqlclient-connection-string>"
  }
}
```

These settings are supported when you run projects locally:

[+] Expand table

Setting	Description
IsEncrypted	When this setting is set to <code>true</code> , all values are encrypted with a local machine key. Used with <code>func settings</code> commands. Default value is <code>false</code> . You might want to encrypt the local.settings.json file on your local computer when it contains secrets, such as service connection strings. The host

Setting	Description
	automatically decrypts settings when it runs. Use the <code>func settings decrypt</code> command before trying to read locally encrypted settings.
Values	<p>Collection of application settings used when a project is running locally. These key-value (string-string) pairs correspond to application settings in your function app in Azure, like AzureWebJobsStorage. Many triggers and bindings have a property that refers to a connection string app setting, like <code>Connection</code> for the Blob storage trigger. For these properties, you need an application setting defined in the <code>values</code> array. See the subsequent table for a list of commonly used settings.</p> <p>Values must be strings and not JSON objects or arrays. Setting names can't include a double underline (<code>__</code>) and shouldn't include a colon (<code>:</code>). Double underline characters are reserved by the runtime, and the colon is reserved to support dependency injection.</p>
Host	Settings in this section customize the Functions host process when you run projects locally. These settings are separate from the <code>host.json</code> settings, which also apply when you run projects in Azure.
LocalHttpPort	Sets the default port used when running the local Functions host (<code>func host start</code> and <code>func run</code>). The <code>--port</code> command-line option takes precedence over this setting. For example, when running in Visual Studio IDE, you may change the port number by navigating to the "Project Properties -> Debug" window and explicitly specifying the port number in a <code>host start --port <your-port-number></code> command that can be supplied in the "Application Arguments" field.
CORS	Defines the origins allowed for cross-origin resource sharing (CORS) . Origins are supplied as a comma-separated list with no spaces. The wildcard value (*) is supported, which allows requests from any origin.
CORS Credentials	When set to <code>true</code> , allows <code>withCredentials</code> requests.
ConnectionStrings	A collection. Don't use this collection for the connection strings used by your function bindings. This collection is used only by frameworks that typically get connection strings from the <code>ConnectionStrings</code> section of a configuration file, like Entity Framework . Connection strings in this object are added to the environment with the provider type of <code>System.Data.SqlClient</code> . Items in this collection aren't published to Azure with other app settings. You must explicitly add these values to the <code>Connection strings</code> collection of your function app settings. If you're creating a <code>SqlConnection</code> in your function code, you should store the connection string value with your other connections in Application Settings in the portal.

The following application settings can be included in the `values` array when running locally:

Setting	Values	Description
<code>AzureWebJobsStorage</code>	Storage account connection string, or <code>UseDevelopmentStorage=true</code>	Contains the connection string for an Azure storage account. Required when using triggers other than HTTP. For more information, see the AzureWebJobsStorage reference. When you have the Azurite Emulator installed locally and you set <code>AzureWebJobsStorage</code> to <code>UseDevelopmentStorage=true</code> , Core Tools uses the emulator. For more information, see Local storage emulator .
<code>AzureWebJobs.<FUNCTION_NAME>.Disabled</code>	<code>true false</code>	To disable a function when running locally, add <code>"AzureWebJobs.<FUNCTION_NAME>.Disabled": "true"</code> to the collection, where <code><FUNCTION_NAME></code> is the name of the function. To learn more, see How to disable functions in Azure Functions .
<code>FUNCTIONS_WORKER_RUNTIME</code>	<code>dotnet</code> <code>dotnet-isolated</code> <code>node</code> <code>java</code> <code>powershell</code> <code>python</code>	Indicates the targeted language of the Functions runtime. Required for version 2.x and higher of the Functions runtime. This setting is generated for your project by Core Tools. To learn more, see the FUNCTIONS_WORKER_RUNTIME reference.
<code>FUNCTIONS_WORKER_RUNTIME_VERSION</code>	<code>~7</code>	Indicates to use PowerShell 7 when running locally. If not set, then PowerShell Core 6 is used. This setting is only used when running locally. The PowerShell runtime version is determined by the <code>powershellVersion</code> site configuration setting, when it runs in Azure, which can be set in the portal .

Synchronize settings

When you develop your functions locally, any local settings required by your app must also be present in app settings of the function app to which your code is deployed. You may also need to download current settings from the function app to your local project. While you can [manually configure app settings in the Azure portal](#), the following tools also let you synchronize app settings with local settings in your project:

- [Visual Studio Code](#)
- [Visual Studio](#)
- [Azure Functions Core Tools](#)

Triggers and bindings

When you develop your functions locally, you need to take trigger and binding behaviors into consideration. For HTTP triggers, you can simply call the HTTP endpoint on the local computer, using `http://localhost/`. For non-HTTP triggered functions, there are several options to run locally:

- The easiest way to test bindings during local development is to use connection strings that target live Azure services. You can target live services by adding the appropriate connection string settings in the `Values` array in the `local.settings.json` file. When you do this, local executions during testing impact live service data. Because of this, consider setting-up separate services to use during development and testing, and then switch to different services during production.
- For storage-based triggers, you can use a [local storage emulator](#).
- You can manually run non-HTTP trigger functions by using special administrator endpoints. For more information, see [Manually run a non HTTP-triggered function](#).

During local testing, you must be running the host provided by Core Tools (`func.exe`) locally. For more information, see [Azure Functions Core Tools](#).

HTTP test tools

During development, it's easy to call any of your function endpoints from a web browser when they support the HTTP GET method. However, for other HTTP methods that support payloads, such as POST or PUT, you need to use an HTTP test tool to create and send these HTTP requests to your function endpoints.

Caution

For scenarios where your requests must include sensitive data, make sure to use a tool that protects your data and reduces the risk of exposing any sensitive data to the public. Sensitive data you should protect might include: credentials, secrets, access tokens, API keys, geolocation data, even personally-identifiable information (PII).

You can keep your data secure by choosing an HTTP test tool that works either offline or locally, doesn't sync your data to the cloud, and doesn't require that you sign in to an online account. Some tools can also protect your data from accidental exposure by implementing specific security features.

Avoid using tools that centrally store your HTTP request history (including sensitive information), don't follow best security practices, or don't respect data privacy concerns.

Consider using one of these tools for securely sending HTTP requests to your function endpoints:

- Visual Studio Code [↗](#) with an extension from Visual Studio Marketplace [↗](#), such as REST Client [↗](#)
- PowerShell Invoke-RestMethod
- Microsoft Edge - Network Console tool
- Bruno [↗](#)
- curl [↗](#)

Local storage emulator

During local development, you can use the local [Azurite emulator](#) when testing functions with Azure Storage bindings (Queue Storage, Blob Storage, and Table Storage), without having to connect to remote storage services. Azurite integrates with Visual Studio Code and Visual Studio, and you can also run it from the command prompt using npm. For more information, see [Use the Azurite emulator for local Azure Storage development](#).

The following setting in the `Values` collection of the `local.settings.json` file tells the local Functions host to use Azurite for the default `AzureWebJobsStorage` connection:

JSON

```
"AzureWebJobsStorage": "UseDevelopmentStorage=true"
```

With this setting value, any Azure Storage trigger or binding that uses `AzureWebJobsStorage` as its connection connects to Azurite when running locally. Keep

these considerations in mind when using storage emulation during local execution:

- You must have Azurite installed and running.
- You should test with an actual storage connection to Azure services before publishing to Azure.
- When you publish your project, don't publish the `AzureWebJobsStorage` setting as `UseDevelopmentStorage=true`. In Azure, the `AzureWebJobsStorage` setting must always be the connection string of the storage account used by your function app. For more information, see [AzureWebJobsStorage](#).

Next steps

- To learn more about local development of compiled C# functions (both in-process and isolated worker process) using Visual Studio, see [Develop Azure Functions using Visual Studio](#).
- To learn more about local development of functions using VS Code on a Mac, Linux, or Windows computer, see the Visual Studio Code getting started article for your preferred language:
 - [C# \(in-process\)](#)
 - [C# \(isolated worker process\)](#)
 - [Java](#)
 - [JavaScript](#)
 - [PowerShell](#)
 - [Python](#)
 - [TypeScript](#)
- To learn more about developing functions from the command prompt or terminal, see [Work with Azure Functions Core Tools](#).

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Develop Azure Functions by using Visual Studio Code

Article • 07/17/2024

The [Azure Functions extension for Visual Studio Code](#) lets you locally develop functions and deploy them to Azure. If this experience is your first with Azure Functions, you can learn more at [An introduction to Azure Functions](#).

The Azure Functions extension provides these benefits:

- Edit, build, and run functions on your local development computer.
- Publish your Azure Functions project directly to Azure.
- Write your functions in various languages while taking advantage of the benefits of Visual Studio Code.

You're viewing the C# version of this article. Make sure to select your preferred Functions programming language at the start of the article.

If you're new to Functions, you might want to first complete the [Visual Studio Code quickstart article](#).

Important

Don't mix local development and portal development for a single function app. When you publish from a local project to a function app, the deployment process overwrites any functions that you developed in the portal.

Prerequisites

- [Visual Studio Code](#) installed on one of the [supported platforms](#).
- [Azure Functions extension](#). You can also install the [Azure Tools extension pack](#), which is recommended for working with Azure resources.
- An active [Azure subscription](#). If you don't yet have an account, you can create one from the extension in Visual Studio Code.

You also need these prerequisites to [run and debug your functions locally](#). They aren't required to just create or publish projects to Azure Functions.

- The [Azure Functions Core Tools](#), which enables an integrated local debugging experience. When you have the Azure Functions extension installed, the easiest way to install or update Core Tools is by running the `Azure Functions: Install or Update Azure Functions Core Tools` command from the command palette.
- The [C# extension](#) for Visual Studio Code.
- [.NET \(CLI\)](#), which is included in the .NET SDK.

Create an Azure Functions project

The Functions extension lets you create the required function app project at the same time you create your first function. Use these steps to create an HTTP-triggered function in a new project. An [HTTP trigger](#) is the simplest function trigger template to demonstrate.

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create New Project...`. Select the directory location for your project workspace, and then choose **Select**.

You can either create a new folder or choose an empty folder for the project workspace, but don't choose a project folder that's already part of a workspace.

You can instead run the command `Azure Functions: Create New Containerized Project...` to also get a Dockerfile generated for the project.

2. When prompted, **Select a language** for your project. If necessary, choose a specific language version.
3. Select the **HTTP trigger** function template, or you can select **Skip for now** to create a project without a function. You can always [add a function to your project](#) later.

💡 Tip

You can view additional templates by selecting the **Change template filter** option and setting the value to **Core** or **All**.

4. For the function name, enter **HttpExample**, select **Enter**, and then select **Function** authorization.

This authorization level requires that you provide a [function key](#) when you call the function endpoint.

- From the dropdown list, select **Add to workspace**.
- In the **Do you trust the authors of the files in this folder?** window, select **Yes**.

Visual Studio Code creates a function in your chosen language and in the template for an HTTP-triggered function.

Generated project files

The project template creates a project in your chosen language and installs the required dependencies. For any language, the new project has these files:

- **host.json**: Lets you configure the Functions host. These settings apply when you're running functions locally and when you're running them in Azure. For more information, see [host.json reference](#).
- **local.settings.json**: Maintains settings used when you're locally running functions. These settings are used only when you're running functions locally. For more information, see [Local settings file](#).

 **Important**

Because the **local.settings.json** file can contain secrets, make sure to exclude the file from your project source control.

- **Dockerfile** (optional): Lets you create a containerized function app from your project by using an approved base image for your project. You only get this file when you run the command `Azure Functions: Create New Containerized Project....`. You can add a Dockerfile to an existing project using the `func init --docker-only` command in [Core Tools](#).

Depending on your language, these other files are created:

An `HttpExample.cs` class library file, the contents of which vary depending on whether your project runs in an [isolated worker process](#) or [in-process](#) with the Functions host.

At this point, you're able to [run your HTTP trigger function locally](#).

Add a function to your project

You can add a new function to an existing project based on one of the predefined Functions trigger templates. To add a new function trigger, select F1 to open the

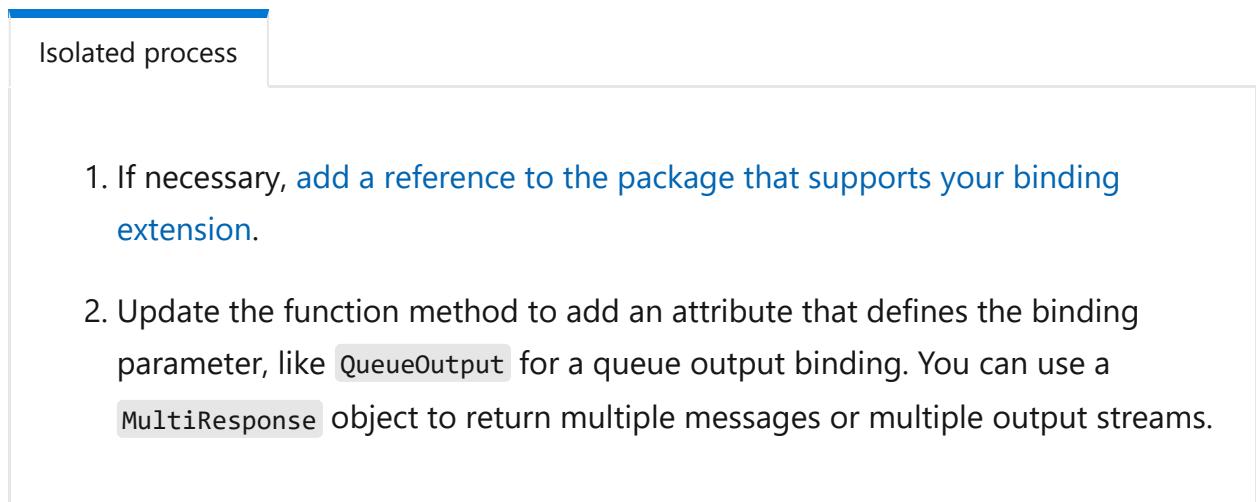
command palette, and then find and run the command **Azure Functions: Create Function**. Follow the prompts to choose your trigger type and define the required attributes of the trigger. If your trigger requires an access key or connection string to connect to a service, get that item ready before you create the function trigger.

This action adds a new C# class library (.cs) file to your project.

Connect to services

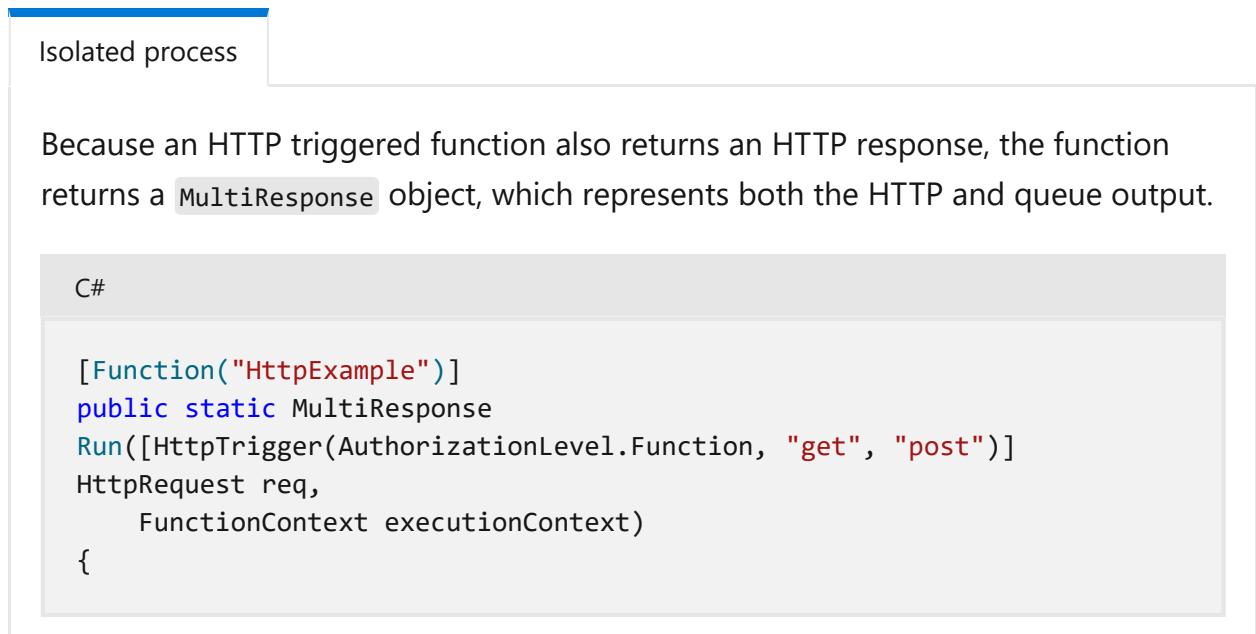
You can connect your function to other Azure services by adding input and output bindings. Bindings connect your function to other services without you having to write the connection code.

For example, the way that you define an output binding that writes data to a storage queue depends on your process model:



1. If necessary, add a reference to the package that supports your binding extension.
2. Update the function method to add an attribute that defines the binding parameter, like `QueueOutput` for a queue output binding. You can use a `MultiResponse` object to return multiple messages or multiple output streams.

The following example shows the function definition after adding a [Queue Storage output binding](#) to an [HTTP triggered function](#):



This example is the definition of the `MultiResponse` object that includes the output binding:

```
C#  
  
public class MultiResponse  
{  
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]  
    public string[] Messages { get; set; }  
    public IActionResult HttpResponse { get; set; }  
}
```

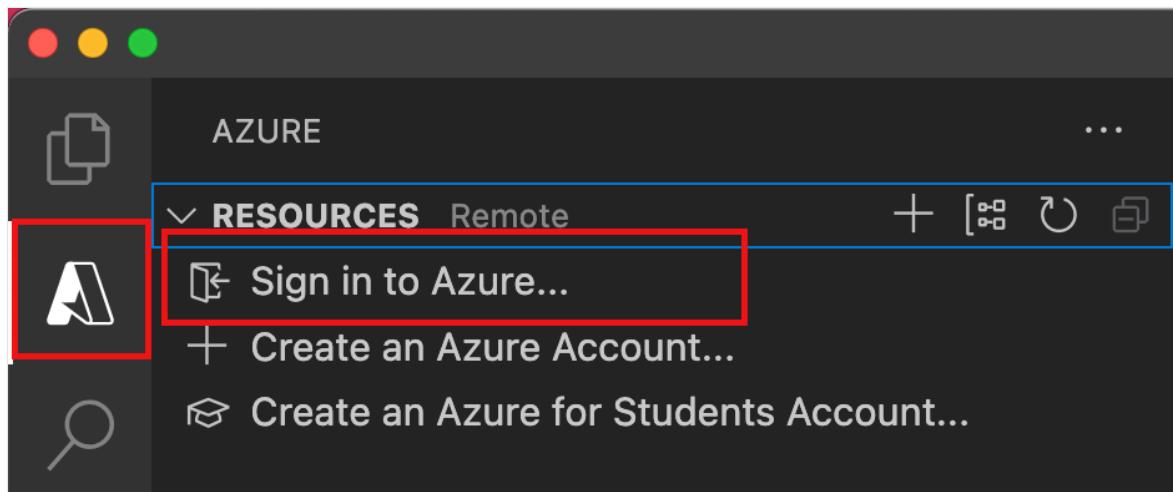
When applying that example to your own project, you might need to change `HttpRequest` to `HttpRequestData` and `IActionResult` to `HttpResponseData`, depending on if you are using [ASP.NET Core integration](#) or not.

Messages are sent to the queue when the function completes. The way you define the output binding depends on your process model. For more information, including links to example binding code that you can refer to, see [Add bindings to a function](#).

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

- When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
- After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create Azure resources

Before you can publish your Functions project to Azure, you must have a function app and related resources in your Azure subscription to run your code. The function app provides an execution context for your functions. When you publish from Visual Studio Code to a function app in Azure, the project is packaged and deployed to the selected function app in your Azure subscription.

When you create a function app in Azure, you can choose either a quick function app create path using defaults or a path that gives you advanced options, such as using existing Azure resources. This way, you have more control over creating the remote resources.

Quick create

In this section, you create a function app and related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors.

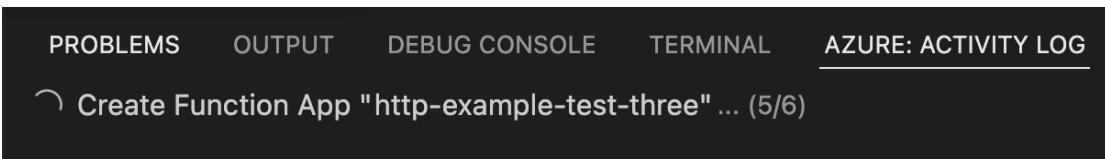
- In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
- At the prompts, provide the following information:

 Expand table

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Enter a name that is valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Select the language version you currently run locally.

Prompt	Action
Select a location for new resources	Select an Azure region. For better performance, select a region near you .

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

Tip

By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

Create an Azure Container Apps deployment

You use Visual Studio Code to create Azure resources for a containerized code project. When the extension detects the presence of a Dockerfile during resource creation, it asks you if you want to deploy the container image instead of just the code. Visual Studio Code creates an Azure Container Apps environment for your containerized code project that's integrated with Azure Functions. For more information, see [Azure Container Apps hosting of Azure Functions](#).

ⓘ Note

Container deployment requires the [Azure Container Apps extension for Visual Studio Code](#). This extension is currently in preview.

The create process depends on whether you choose a quick create or you need to use advanced options:



Quick create

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create Function App in Azure...`.
2. When prompted choose **Container image**.
3. Provide the following information at the prompts:

[] Expand table

Prompt	Selection
Select subscription	Choose the subscription to use. You won't see this prompt when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Select a location for new resources	For better performance, choose a region near you.

The extension shows the status of individual resources as they're being created in Azure in the **Azure: Activity Log** panel.

For more information about the resources required to run your containerized functions in Container Apps, see [Required resources](#).

Note

You can't currently use Visual Studio Code to deploy a containerized function app to an Azure Functions-integrated Container Apps environment. You must instead publish your container image to a container registry and then set that registry image as the deployment source for your Container Apps-hosted function app. For more information, see [Create your function app in a container](#) and [Update an image in the registry](#).

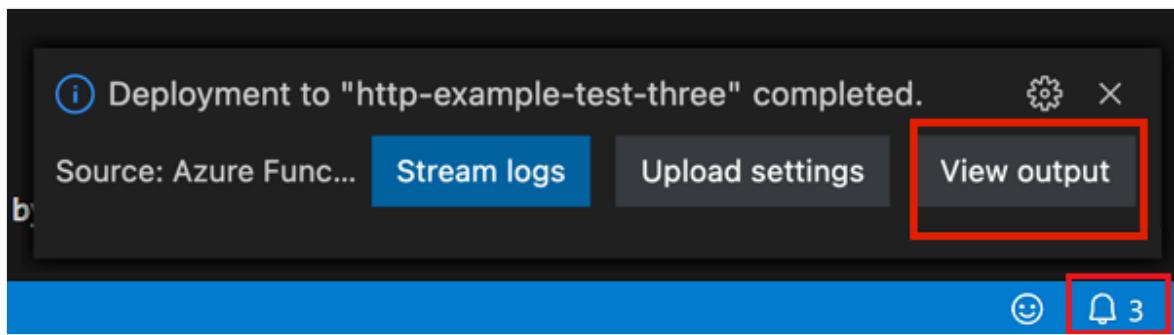
Deploy project files

We recommend setting up [continuous deployment](#) so that your function app in Azure is updated when you update source files in the connected source location. You can also deploy your project files from Visual Studio Code. When you publish from Visual Studio Code, you can take advantage of the [Zip deploy technology](#).

Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Get the URL of an HTTP triggered function in Azure

To call an HTTP-triggered function from a client, you need the function's URL, which is available after deployment to your function app. This URL includes any required function keys. You can use the extension to get these URLs for your deployed functions. If you just want to run the remote function in Azure, [use the Execute function now](#) functionality of the extension.

1. Select F1 to open the command palette, and then find and run the command **Azure Functions: Copy Function URL**.
2. Follow the prompts to select your function app in Azure and then the specific HTTP trigger that you want to invoke.

The function URL is copied to the clipboard, along with any required keys passed by the `code` query parameter. Use an HTTP tool to submit POST requests, or a browser to submit GET requests to the remote function.

When the extension gets the URL of a function in Azure, the extension uses your Azure account to automatically retrieve the keys needed to start the function. [Learn more about function access keys](#). Starting non-HTTP triggered functions requires using the admin key.

Run functions

The Azure Functions extension lets you run individual functions. You can run functions either in your project on your local development computer or in your Azure subscription.

For HTTP trigger functions, the extension calls the HTTP endpoint. For other kinds of triggers, the extension calls administrator APIs to start the function. The message body of the request sent to the function depends on the trigger type. When a trigger requires test data, you're prompted to enter data in a specific JSON format.

Run functions in Azure

To execute a function in Azure from Visual Studio Code, follow these steps:

1. In the command palette, enter **Azure Functions: Execute function now**, and select your Azure subscription.
2. From the list, choose your function app in Azure. If you don't see your function app, make sure you're signed in to the correct subscription.
3. From the list, choose the function that you want to run. In **Enter request body**, type the message body of the request, and press Enter to send this request message to your function.

The default text in **Enter request body** indicates the body's format. If your function app has no functions, a notification error is shown with this error.

When the function executes in Azure and returns a response, Visual Studio Code shows a notification.

You can also run your function from the **Azure: Functions** area by opening the shortcut menu for the function that you want to run from your function app in your Azure subscription, and then selecting **Execute Function Now....**

When you run your functions in Azure from Visual Studio Code, the extension uses your Azure account to automatically retrieve the keys needed to start the function. [Learn more about function access keys](#). Starting non-HTTP triggered functions requires using the admin key.

Run functions locally

The local runtime is the same runtime that hosts your function app in Azure. Local settings are read from the [local.settings.json file](#). To run your Functions project locally, you must meet [more requirements](#).

Configure the project to run locally

The Functions runtime uses an Azure Storage account internally for all trigger types other than HTTP and webhooks. So you need to set the **Values.AzureWebJobsStorage** key to a valid Azure Storage account connection string.

This section uses the [Azure Storage extension for Visual Studio Code](#) with [Azure Storage Explorer](#) to connect to and retrieve the storage connection string.

To set the storage account connection string:

1. In Visual Studio, open **Cloud Explorer**, expand **Storage Account > Your Storage Account**, and then select **Properties** and copy the **Primary Connection String** value.
2. In your project, open the local.settings.json file and set the value of the **AzureWebJobsStorage** key to the connection string you copied.
3. Repeat the previous step to add unique keys to the **Values** array for any other connections required by your functions.

For more information, see [Local settings file](#).

Debug functions locally

To debug your functions, select F5. If **Core Tools** isn't available, you're prompted to install it. When Core Tools is installed and running, output is shown in the Terminal. This step is the same as running the `func start` Core Tools command from the Terminal, but with extra build tasks and an attached debugger.

When the project is running, you can use the **Execute Function Now...** feature of the extension to trigger your functions as you would when the project is deployed to Azure. With the project running in debug mode, breakpoints are hit in Visual Studio Code as you would expect.

1. In the command palette, enter **Azure Functions: Execute function now** and choose **Local project**.
2. Choose the function you want to run in your project and type the message body of the request in **Enter request body**. Press Enter to send this request message to your function. The default text in **Enter request body** should indicate the format of the body. If your function app has no functions, a notification error is shown with this error.
3. When the function runs locally and after the response is received, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.

Keys aren't required when running locally, which applies to both function keys and admin-level keys.

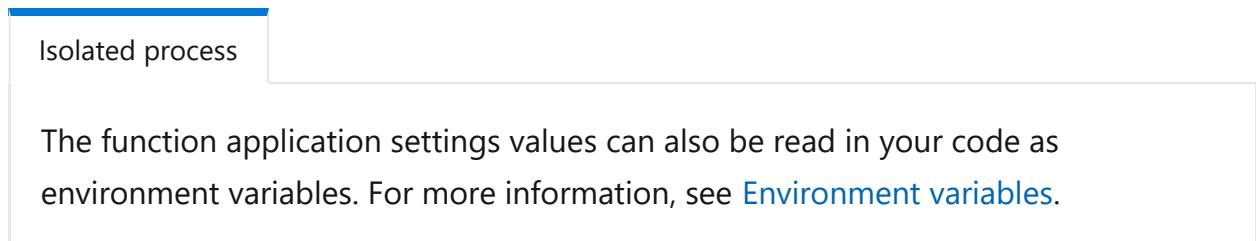
Work with app settings locally

When running in a function app in Azure, settings required by your functions are [stored securely in app settings](#). During local development, these settings are instead added to the `Values` collection in the `local.settings.json` file. The `local.settings.json` file also stores settings used by local development tools.

Items in the `Values` collection in your project's `local.settings.json` file are intended to mirror items in your function app's [application settings](#) in Azure.

By default, these settings aren't migrated automatically when the project is published to Azure. After publishing finishes, you're given the option of publishing settings from `local.settings.json` to your function app in Azure. To learn more, see [Publish application settings](#).

Values in `ConnectionStrings` are never published.

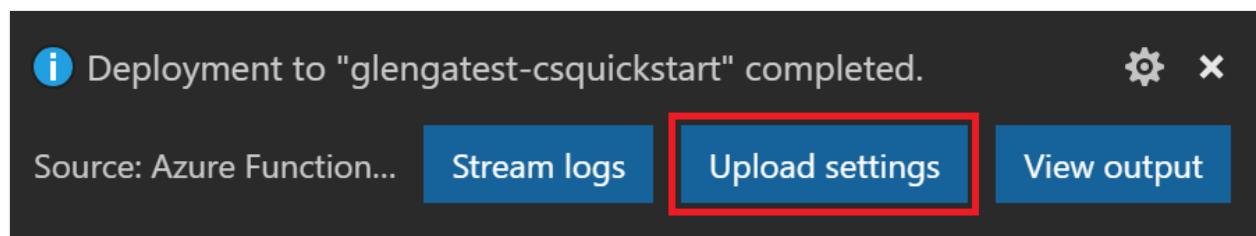


Application settings in Azure

The settings in the `local.settings.json` file in your project should be the same as the application settings in the function app in Azure. Any settings you add to `local.settings.json` you must also add to the function app in Azure. These settings aren't uploaded automatically when you publish the project. Likewise, any settings that you create in your function app [in the portal](#) must be downloaded to your local project.

Publish application settings

The easiest way to publish the required settings to your function app in Azure is to use the [Upload settings](#) link that appears after you publish your project:



You can also publish settings by using the [Azure Functions: Upload Local Setting](#) command in the command palette. You can add individual settings to application

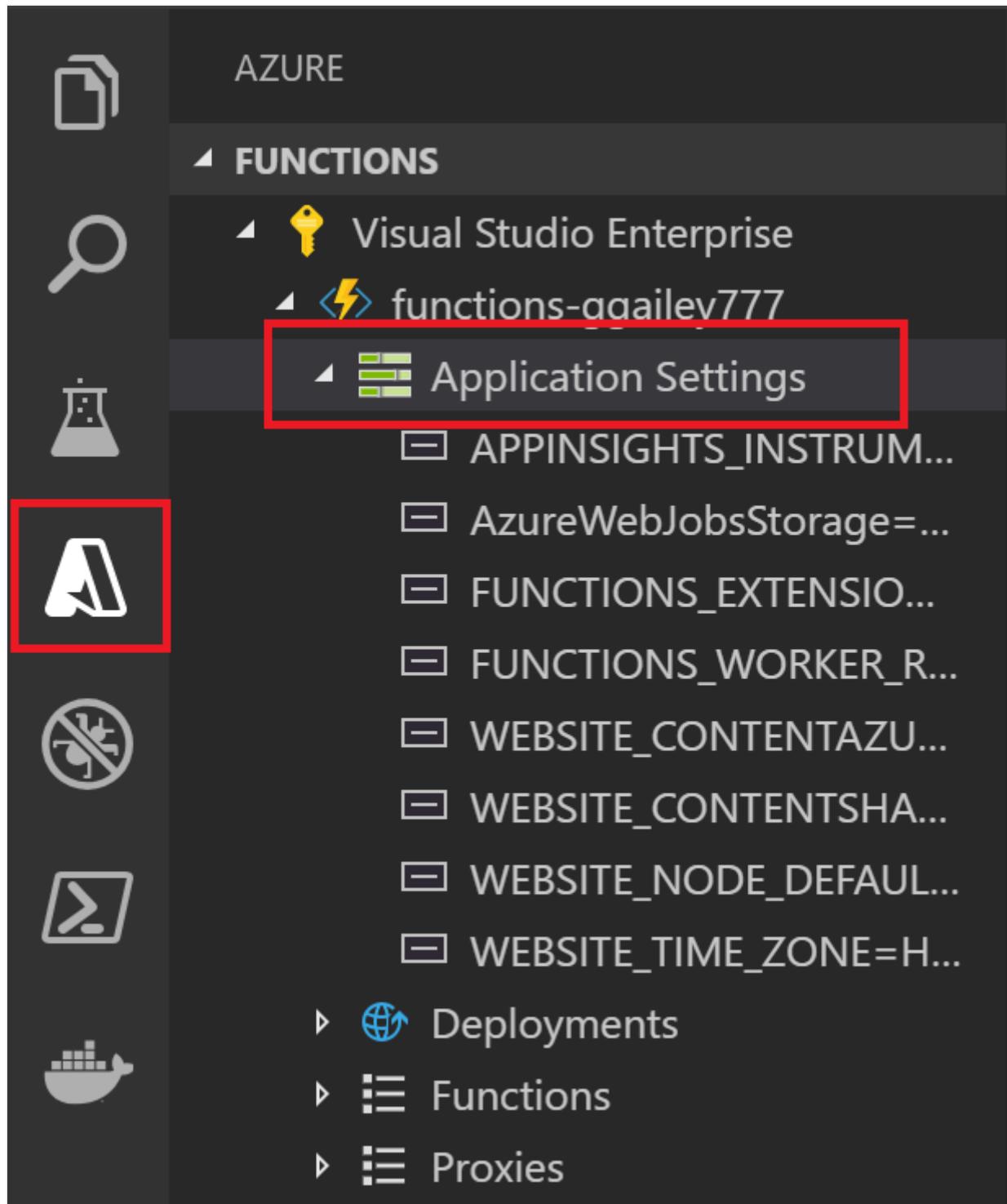
settings in Azure by using the **Azure Functions: Add New Setting** command.

 **Tip**

Be sure to save your local.settings.json file before you publish it.

If the local file is encrypted, it's decrypted, published, and encrypted again. If there are settings that have conflicting values in the two locations, you're prompted to choose how to proceed.

View existing app settings in the **Azure: Functions** area by expanding your subscription, your function app, and **Application Settings**.



Download settings from Azure

If you've created application settings in Azure, you can download them into your local.settings.json file by using the [Azure Functions: Download Remote Settings](#) command.

As with uploading, if the local file is encrypted, it's decrypted, updated, and encrypted again. If there are settings that have conflicting values in the two locations, you're prompted to choose how to proceed.

Install binding extensions

Except for HTTP and timer triggers, bindings are implemented in extension packages.

You must explicitly install the extension packages for the triggers and bindings that need them. The specific package you install depends on your project's process model.

Isolated process

Run the [dotnet add package](#) command in the Terminal window to install the extension packages that you need in your project. This template demonstrates how you add a binding for an [isolated-process class library](#):

terminal

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.  
<BINDING_TYPE_NAME> --version <TARGET_VERSION>
```

Replace `<BINDING_TYPE_NAME>` with the name of the package that contains the binding you need. You can find the desired binding reference article in the [list of supported bindings](#).

Replace `<TARGET_VERSION>` in the example with a specific version of the package, such as `3.0.0-beta5`. Valid versions are listed on the individual package pages at [NuGet.org](#). The major versions that correspond to the current Functions runtime are specified in the reference article for the binding.

Tip

You can also use the **NuGet** commands in [the C# Dev Kit](#) to install binding extension packages.

C# script uses [extension bundles](#).

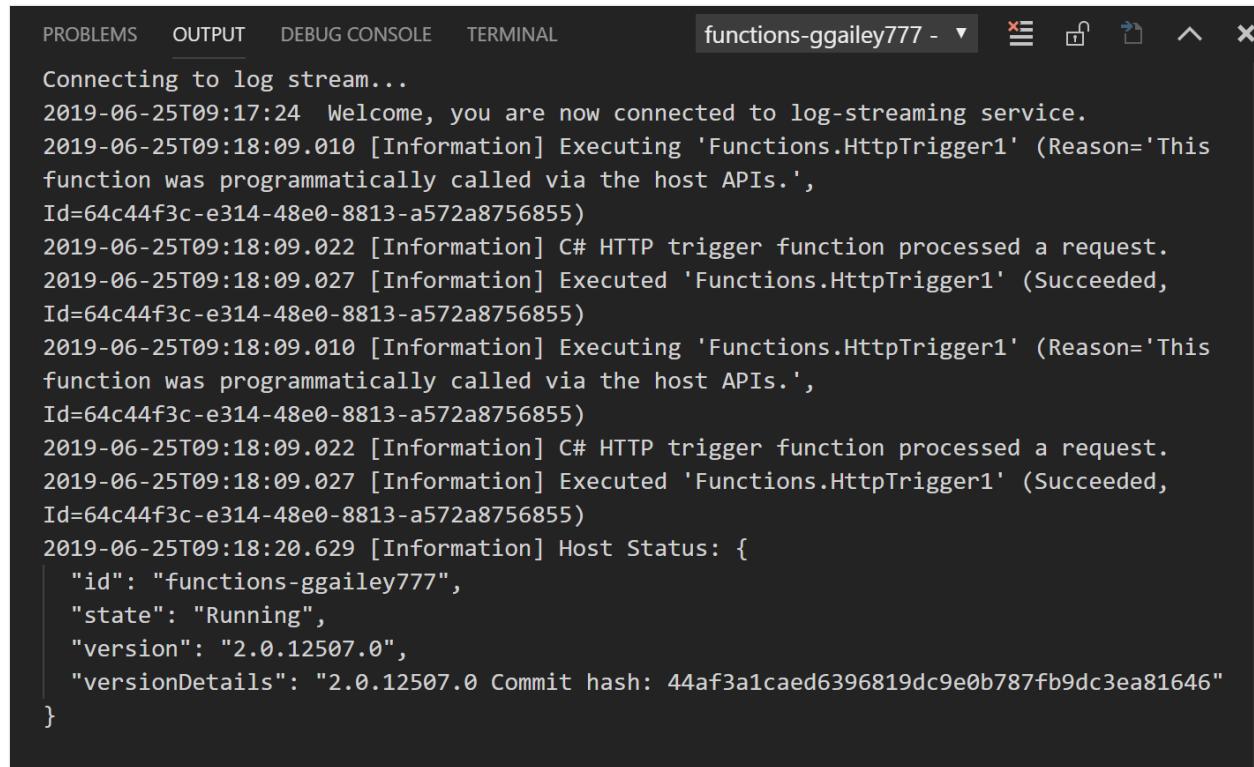
Monitoring functions

When you [run functions locally](#), log data is streamed to the Terminal console. You can also get log data when your Functions project is running in a function app in Azure. You can connect to streaming logs in Azure to see near-real-time log data. You should enable Application Insights for a more complete understanding of how your function app is behaving.

Streaming logs

When you're developing an application, it's often useful to see logging information in near-real time. You can view a stream of log files being generated by your functions.

Turn on logs from the command pallet with the `Azure Functions: Start streaming logs` command. This output is an example of streaming logs for a request to an HTTP-triggered function:



The screenshot shows the Visual Studio Code interface with the "OUTPUT" tab selected. The title bar says "functions-ggailey777 -". The log output is as follows:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
functions-ggailey777 - ▾ ✖ ⓘ ⌂ ⌄ ⌁ ×

Connecting to log stream...
2019-06-25T09:17:24 Welcome, you are now connected to log-streaming service.
2019-06-25T09:18:09.010 [Information] Executing 'Functions.HttpTrigger1' (Reason='This function was programmatically called via the host APIs.', Id=64c44f3c-e314-48e0-8813-a572a8756855)
2019-06-25T09:18:09.022 [Information] C# HTTP trigger function processed a request.
2019-06-25T09:18:09.027 [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=64c44f3c-e314-48e0-8813-a572a8756855)
2019-06-25T09:18:09.010 [Information] Executing 'Functions.HttpTrigger1' (Reason='This function was programmatically called via the host APIs.', Id=64c44f3c-e314-48e0-8813-a572a8756855)
2019-06-25T09:18:09.022 [Information] C# HTTP trigger function processed a request.
2019-06-25T09:18:09.027 [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=64c44f3c-e314-48e0-8813-a572a8756855)
2019-06-25T09:18:20.629 [Information] Host Status: {
    "id": "functions-ggailey777",
    "state": "Running",
    "version": "2.0.12507.0",
    "versionDetails": "2.0.12507.0 Commit hash: 44af3a1caed6396819dc9e0b787fb9dc3ea81646"
}
```

To learn more, see [Streaming logs](#).

Application Insights

You should monitor the execution of your functions by integrating your function app with Application Insights. When you create a function app in the Azure portal, this integration occurs by default. When you create your function app during Visual Studio publishing, you need to integrate Application Insights yourself. To learn how, see [Enable Application Insights integration](#).

To learn more about monitoring using Application Insights, see [Monitor Azure Functions](#).

C# script projects

By default, all C# projects are created as [C# compiled class library projects](#). If you prefer to work with C# script projects instead, you must select C# script as the default

language in the Azure Functions extension settings:

1. Select **File > Preferences > Settings**.
2. Go to **User Settings > Extensions > Azure Functions**.
3. Select **C#Script** from **Azure Function: Project Language**.

After you complete these steps, calls made to the underlying Core Tools include the `--csx` option, which generates and publishes C# script (.csx) project files. When you have this default language specified, all projects that you create default to C# script projects. You're not prompted to choose a project language when a default is set. To create projects in other languages, you must change this setting or remove it from the user `settings.json` file. After you remove this setting, you're again prompted to choose your language when you create a project.

Command palette reference

The Azure Functions extension provides a useful graphical interface in the area for interacting with your function apps in Azure. The same functionality is also available as commands in the command palette (F1). These Azure Functions commands are available:

[+] [Expand table](#)

Azure Functions command	Description
Add New Settings	Creates a new application setting in Azure. To learn more, see Publish application settings . You might also need to download this setting to your local settings .
Configure Deployment Source	Connects your function app in Azure to a local Git repository. To learn more, see Continuous deployment for Azure Functions .
Connect to GitHub Repository	Connects your function app to a GitHub repository.
Copy Function URL	Gets the remote URL of an HTTP-triggered function that's running in Azure. To learn more, see Get the URL of the deployed function .
Create function app in Azure	Creates a new function app in your subscription in Azure. To learn more, see the section on how to publish to a new function app in Azure .
Decrypt Settings	Decrypts local settings that have been encrypted by Azure Functions :

Azure Functions command	Description
	Encrypt Settings.
Delete Function App	Removes a function app from your subscription in Azure. When there are no other apps in the App Service plan, you're given the option to delete that too. Other resources, like storage accounts and resource groups, aren't deleted. To remove all resources, you should instead delete the resource group . Your local project isn't affected.
Delete Function	Removes an existing function from a function app in Azure. Because this deletion doesn't affect your local project, instead consider removing the function locally and then republishing your project .
Delete Proxy	Removes an Azure Functions proxy from your function app in Azure. To learn more about proxies, see Work with Azure Functions Proxies .
Delete Setting	Deletes a function app setting in Azure. This deletion doesn't affect settings in your local.settings.json file.
Disconnect from Repo	Removes the continuous deployment connection between a function app in Azure and a source control repository.
Download Remote Settings	Downloads settings from the chosen function app in Azure into your local.settings.json file. If the local file is encrypted, it's decrypted, updated, and encrypted again. If there are settings that have conflicting values in the two locations, you're prompted to choose how to proceed. Be sure to save changes to your local.settings.json file before you run this command.
Edit settings	Changes the value of an existing function app setting in Azure. This command doesn't affect settings in your local.settings.json file.
Encrypt settings	Encrypts individual items in the <code>Values</code> array in the local settings . In this file, <code>IsEncrypted</code> is also set to <code>true</code> , which specifies that the local runtime decrypt settings before using them. Encrypt local settings to reduce the risk of leaking valuable information. In Azure, application settings are always stored encrypted.
Execute Function Now	Manually starts a function using admin APIs. This command is used for testing, both locally during debugging and against functions running in Azure. When a function in Azure starts, the extension first automatically obtains an admin key, which it uses to call the remote admin APIs that start functions in Azure. The body of the message sent to the API depends on the type of trigger. Timer triggers don't require you to pass any data.
Initialize Project for Use with VS Code	Adds the required Visual Studio Code project files to an existing Functions project. Use this command to work with a project that you created by using Core Tools.

Azure Functions command	Description
Install or Update Azure Functions Core Tools	Installs or updates Azure Functions Core Tools , which is used to run functions locally.
Redeploy	Lets you redeploy project files from a connected Git repository to a specific deployment in Azure. To republish local updates from Visual Studio Code, republish your project .
Rename Settings	Changes the key name of an existing function app setting in Azure. This command doesn't affect settings in your local.settings.json file. After you rename settings in Azure, you should download those changes to the local project .
Restart	Restarts the function app in Azure. Deploying updates also restarts the function app.
Set AzureWebJobsStorage	Sets the value of the <code>AzureWebJobsStorage</code> application setting. This setting is required by Azure Functions. It's set when a function app is created in Azure.
Start	Starts a stopped function app in Azure.
Start Streaming Logs	Starts the streaming logs for the function app in Azure. Use streaming logs during remote troubleshooting in Azure if you need to see logging information in near-real time. To learn more, see Streaming logs .
Stop	Stops a function app that's running in Azure.
Stop Streaming Logs	Stops the streaming logs for the function app in Azure.
Toggle as Slot Setting	When enabled, ensures that an application setting persists for a given deployment slot.
Uninstall Azure Functions Core Tools	Removes Azure Functions Core Tools, which is required by the extension.
Upload Local Settings	Uploads settings from your local.settings.json file to the chosen function app in Azure. If the local file is encrypted, it's decrypted, uploaded, and encrypted again. If there are settings that have conflicting values in the two locations, you're prompted to choose how to proceed. Be sure to save changes to your local.settings.json file before you run this command.
View Commit in GitHub	Shows you the latest commit in a specific deployment when your function app is connected to a repository.
View Deployment Logs	Shows you the logs for a specific deployment to the function app in Azure.

Next steps

To learn more about Azure Functions Core Tools, see [Work with Azure Functions Core Tools](#).

To learn more about developing functions as .NET class libraries, see [Azure Functions C# developer reference](#). This article also provides links to examples of how to use attributes to declare the various types of bindings supported by Azure Functions.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Develop Azure Functions using Visual Studio

Article • 02/25/2024

Visual Studio lets you develop, test, and deploy C# class library functions to Azure. If this experience is your first with Azure Functions, see [An introduction to Azure Functions](#).

To get started right away, consider completing the [Functions quickstart for Visual Studio](#).

This article provides details about how to use Visual Studio to develop C# class library functions and publish them to Azure. There are two models for developing C# class library functions: the [Isolated worker model](#) and the [In-process model](#).

You're reading the isolated worker model version this article. You can choose your preferred model at the top of the article.

Unless otherwise noted, procedures and examples shown are for Visual Studio 2022. For more information about Visual Studio 2022 releases, see [the release notes](#) or the [preview release notes](#).

Prerequisites

- Visual Studio 2022, including the [Azure development](#) workload.
- Other resources that you need, such as an Azure Storage account, are created in your subscription during the publishing process.
- If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Create an Azure Functions project

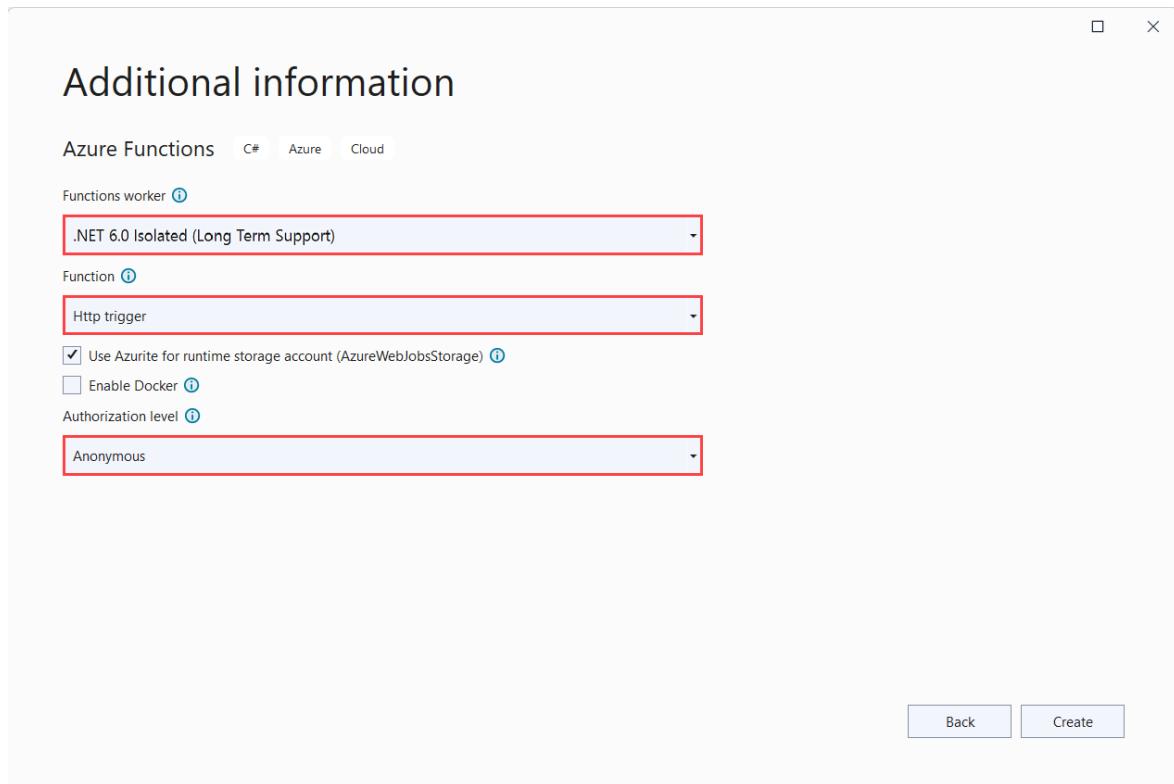
The Azure Functions project template in Visual Studio creates a C# class library project that you can publish to a function app in Azure. You can use a function app to group functions as a logical unit for easier management, deployment, scaling, and sharing of resources.

1. From the Visual Studio menu, select **File > New > Project**.
2. In **Create a new project**, enter *functions* in the search box, choose the **Azure Functions** template, and then select **Next**.

3. In **Configure your new project**, enter a **Project name** for your project, and then select **Create**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other nonalphanumeric characters.
4. For the **Create a new Azure Functions application** settings, use the values in the following table:

[] [Expand table](#)

Setting	Value	Description
.NET version	.NET 6 Isolated	This value creates a function project that runs in an isolated worker process . Isolated worker process supports other non-LTS version of .NET and also .NET Framework. For more information, see Azure Functions runtime versions overview .
Function template	HTTP trigger	This value creates a function triggered by an HTTP request.
Storage account (AzureWebJobsStorage)	Storage emulator	Because a function app in Azure requires a storage account, one is assigned or created when you publish your project to Azure. An HTTP trigger doesn't use an Azure Storage account connection string; all other trigger types require a valid Azure Storage account connection string.
Authorization level	Anonymous	The created function can be triggered by any client without providing a key. This authorization setting makes it easy to test your new function. For more information about keys and authorization, see Authorization keys and HTTP and webhook bindings .



Make sure you set the **Authorization level** to **Anonymous**. If you choose the default level of **Function**, you're required to present the [function key](#) in requests to access your function endpoint.

5. Select **Create** to create the function project and HTTP trigger function.

After you create an Azure Functions project, the project template creates a C# project, installs the `Microsoft.Azure.Functions.Worker` and `Microsoft.Azure.Functions.Worker.Sdk` NuGet packages, and sets the target framework.

The new project has the following files:

- **host.json**: Lets you configure the Functions host. These settings apply both when running locally and in Azure. For more information, see [host.json reference](#).
- **local.settings.json**: Maintains settings used when running functions locally. These settings aren't used when running in Azure. For more information, see [Local settings file](#).

Important

Because the `local.settings.json` file can contain secrets, you must exclude it from your project source control. Make sure the **Copy to Output Directory** setting for this file is set to **Copy if newer**.

For more information, see [Project structure](#) in the Isolated worker guide.

Work with app settings locally

When running in a function app in Azure, settings required by your functions are [stored securely in app settings](#). During local development, these settings are instead added to the `Values` collection in the `local.settings.json` file. The `local.settings.json` file also stores settings used by local development tools.

Items in the `Values` collection in your project's `local.settings.json` file are intended to mirror items in your function app's [application settings](#) in Azure.

Visual Studio doesn't automatically upload the settings in `local.settings.json` when you publish the project. To make sure that these settings also exist in your function app in Azure, upload them after you publish your project. For more information, see [Function app settings](#). The values in a `ConnectionStrings` collection are never published.

Your code can also read the function app settings values as environment variables. For more information, see [Environment variables](#).

Configure the project for local development

The Functions runtime uses an Azure Storage account internally. For all trigger types other than HTTP and webhooks, set the `Values.AzureWebJobsStorage` key to a valid Azure Storage account connection string. Your function app can also use the [Azurite emulator](#) for the `AzureWebJobsStorage` connection setting required by the project. To use the emulator, set the value of `AzureWebJobsStorage` to `UseDevelopmentStorage=true`. Change this setting to an actual storage account connection string before deployment. For more information, see [Local storage emulator](#).

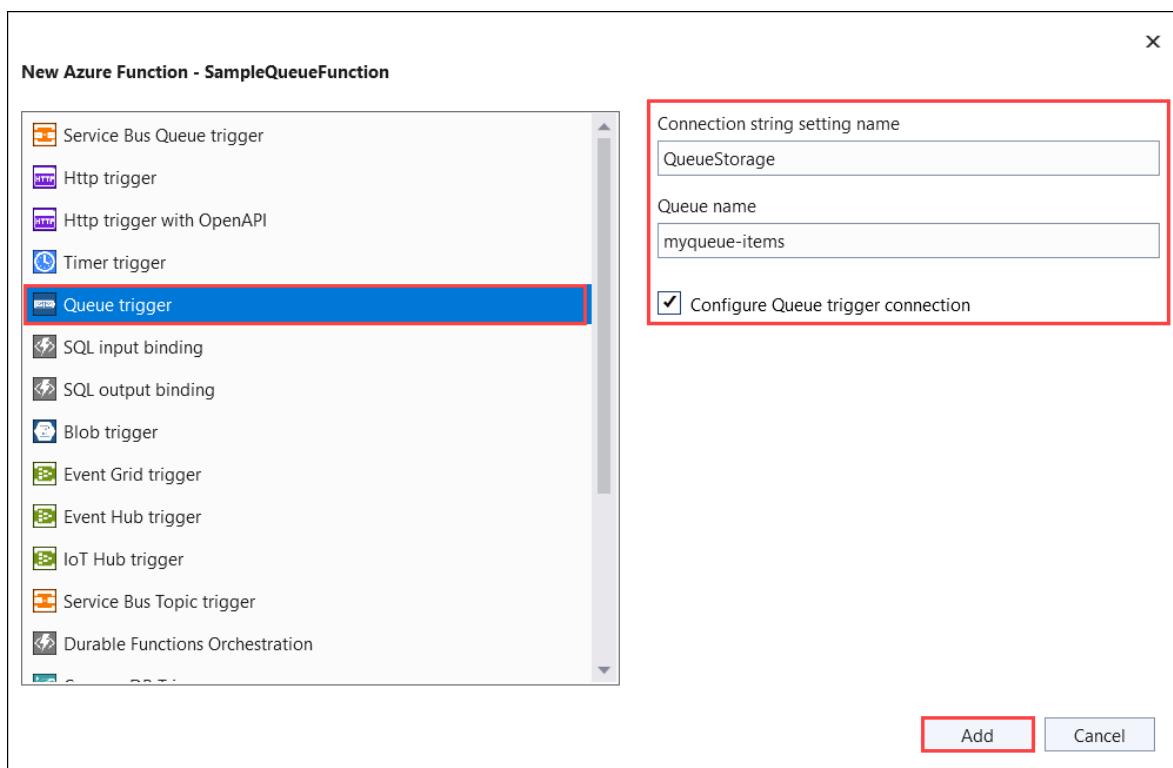
To set the storage account connection string:

1. In the Azure portal, navigate to your storage account.
2. In the **Access keys** tab, below **Security + networking**, copy the **Connection string of key1**.
3. In your project, open the `local.settings.json` file and set the value of the `AzureWebJobsStorage` key to the connection string you copied.
4. Repeat the previous step to add unique keys to the `values` array for any other connections required by your functions.

Add a function to your project

In C# class library functions, the bindings used by the function are defined by applying attributes in the code. When you create your function triggers from the provided templates, the trigger attributes are applied for you.

1. In **Solution Explorer**, right-click your project node and select **Add > New Azure Function**.
2. Enter a **Name** for the class, and then select **Add**.
3. Choose your trigger, set the required binding properties, and then select **Add**. The following example shows the settings for creating a Queue storage trigger function.



For an Azure Storage service trigger, check the **Configure connection** box and you're prompted to choose between using an Azurite storage emulator or referencing a provisioned Azure storage account. Select **Next** and if you choose a storage account, Visual Studio tries to connect to your Azure account and get the connection string. Choose **Save connection string value in Local user secrets file** and then **Finish** to create the trigger class.

This trigger example uses an application setting for the storage connection with a key named `QueueStorage`. This key, stored in the [local.settings.json file](#), either references the Azurite emulator or an Azure storage account.

4. Examine the newly added class. For example, the following C# class represents a basic Queue storage trigger function:

You see a static `Run()` method attributed with `Function`. This attribute indicates that the method is the entry point for the function.

C#

```
using System;
using Azure.Storage.Queues.Models;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace Company.Function
{
    public class QueueTriggerCSharp
    {
        private readonly ILogger<QueueTriggerCSharp> _logger;

        public QueueTriggerCSharp(ILogger<QueueTriggerCSharp> logger)
        {
            _logger = logger;
        }

        [Function(nameof(QueueTriggerCSharp))]
        public void Run([QueueTrigger("PathValue", Connection =
"ConnectionStringValue")] QueueMessage message)
        {
            _logger.LogInformation($"C# Queue trigger function
processed: {message.MessageText}");
        }
    }
}
```

A binding-specific attribute is applied to each binding parameter supplied to the entry point method. The attribute takes the binding information as parameters. In the previous example, the first parameter has a `QueueTrigger` attribute applied, indicating a Queue storage trigger function. The queue name and connection string setting name are passed as parameters to the `QueueTrigger` attribute. For more information, see [Azure Queue storage bindings for Azure Functions](#).

Use the above procedure to add more functions to your function app project. Each function in the project can have a different trigger, but a function must have exactly one trigger. For more information, see [Azure Functions triggers and bindings concepts](#).

Add bindings

As with triggers, input and output bindings are added to your function as binding attributes. Add bindings to a function as follows:

1. Make sure you [configure the project for local development](#).
2. Add the appropriate NuGet extension package for the specific binding by finding the binding-specific NuGet package requirements in the reference article for the binding. For example, find package requirements for the Event Hubs trigger in the [Event Hubs binding reference article](#).
3. Use the following command in the Package Manager Console to install a specific package:

```
PowerShell
```

```
Install-Package Microsoft.Azure.Functions.Worker.Extensions.  
<BINDING_TYPE> -Version <TARGET_VERSION>
```

In this example, replace `<BINDING_TYPE>` with the name specific to the binding extension and `<TARGET_VERSION>` with a specific version of the package, such as `4.0.0`. Valid versions are listed on the individual package pages at [NuGet.org](#).

4. If there are app settings that the binding needs, add them to the `values` collection in the [local setting file](#).

The function uses these values when it runs locally. When the function runs in the function app in Azure, it uses the [function app settings](#). Visual Studio makes it easy to [publish local settings to Azure](#).

5. Add the appropriate binding attribute to the method signature. In the following example, a queue message triggers the function, and the output binding creates a new queue message with the same text in a different queue.

```
C#
```

```
public class QueueTrigger  
{  
    private readonly ILogger _logger;  
  
    public QueueTrigger(ILocator loggerFactory)  
    {  
        _logger = loggerFactory.CreateLogger<QueueTrigger>();  
    }  
  
    [Function("CopyQueueMessage")]  
    [QueueOutput("myqueue-items-destination", Connection =  
    "QueueStorage")]  
    public string Run([QueueTrigger("myqueue-items-source", Connection  
    = "QueueStorage")] string myQueueItem)  
    {
```

```
        _logger.LogInformation($"C# Queue trigger function processed:  
        {myQueueItem}");  
        return myQueueItem;  
    }  
}
```

The `QueueOutput` attribute defines the binding on the method. For multiple output bindings, you would instead place this attribute on a string property of the returned object. For more information, see [Multiple output bindings](#).

The connection to Queue storage is obtained from the `QueueStorage` setting. For more information, see the reference article for the specific binding.

For a full list of the bindings supported by Functions, see [Supported bindings](#).

Run functions locally

Azure Functions Core Tools lets you run Azure Functions project on your local development computer. When you press F5 to debug a Functions project, the local Functions host (`func.exe`) starts to listen on a local port (usually 7071). Any callable function endpoints are written to the output, and you can use these endpoints for testing your functions. For more information, see [Work with Azure Functions Core Tools](#). You're prompted to install these tools the first time you start a function from Visual Studio.

To start your function in Visual Studio in debug mode:

1. Press F5. If prompted, accept the request from Visual Studio to download and install Azure Functions Core (CLI) tools. You might also need to enable a firewall exception so that the tools can handle HTTP requests.
2. With the project running, test your code as you would test a deployed function.

When you run Visual Studio in debug mode, breakpoints are hit as expected.

For a more detailed testing scenario using Visual Studio, see [Testing functions](#).

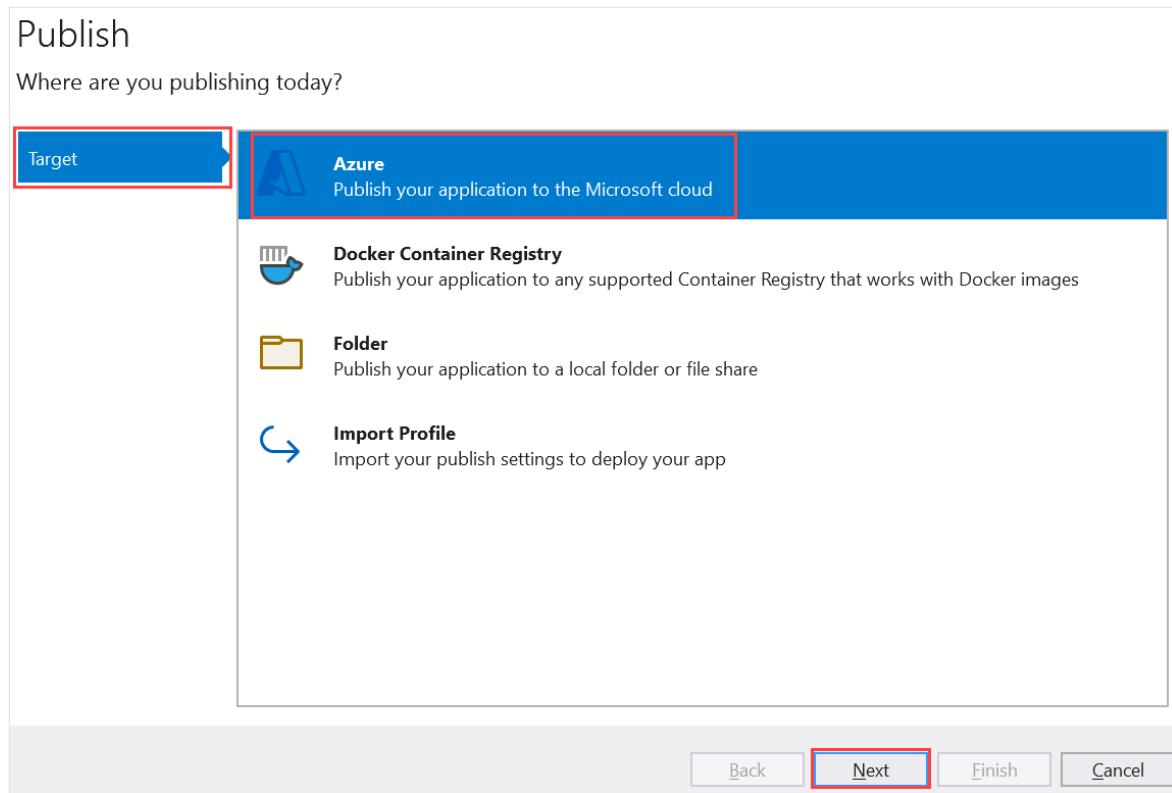
Publish to Azure

When you publish your functions project to Azure, Visual Studio uses [zip deployment](#) to deploy the project files. When possible, you should also select [Run-From-Package](#) so that the project runs in the deployment (.zip) package. For more information, see [Run your functions from a package file in Azure](#).

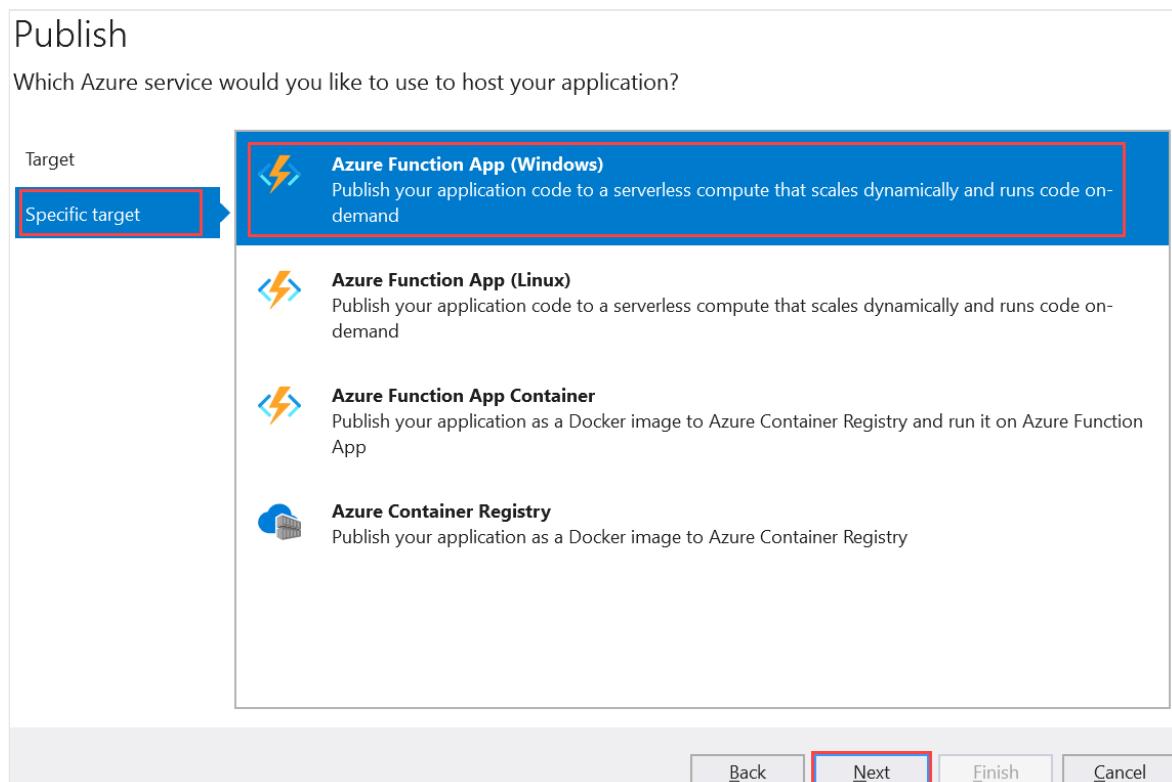
Don't deploy to Azure Functions using Web Deploy (`msdeploy`).

Use the following steps to publish your project to a function app in Azure.

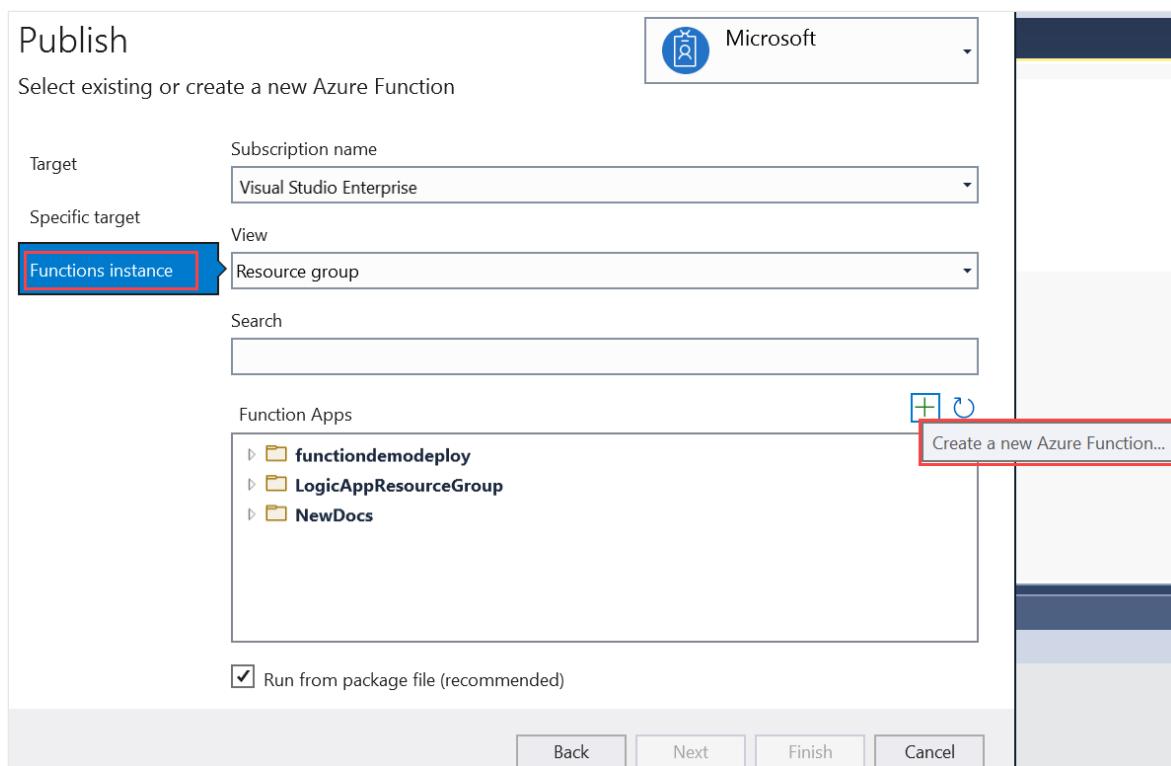
1. In **Solution Explorer**, right-click the project and select **Publish**. In **Target**, select **Azure** then **Next**.



2. Select **Azure Function App (Windows)** for the **Specific target**, which creates a function app that runs on Windows, and then select **Next**.



3. In the Function Instance, choose Create a new Azure Function...



4. Create a new instance using the values specified in the following table:

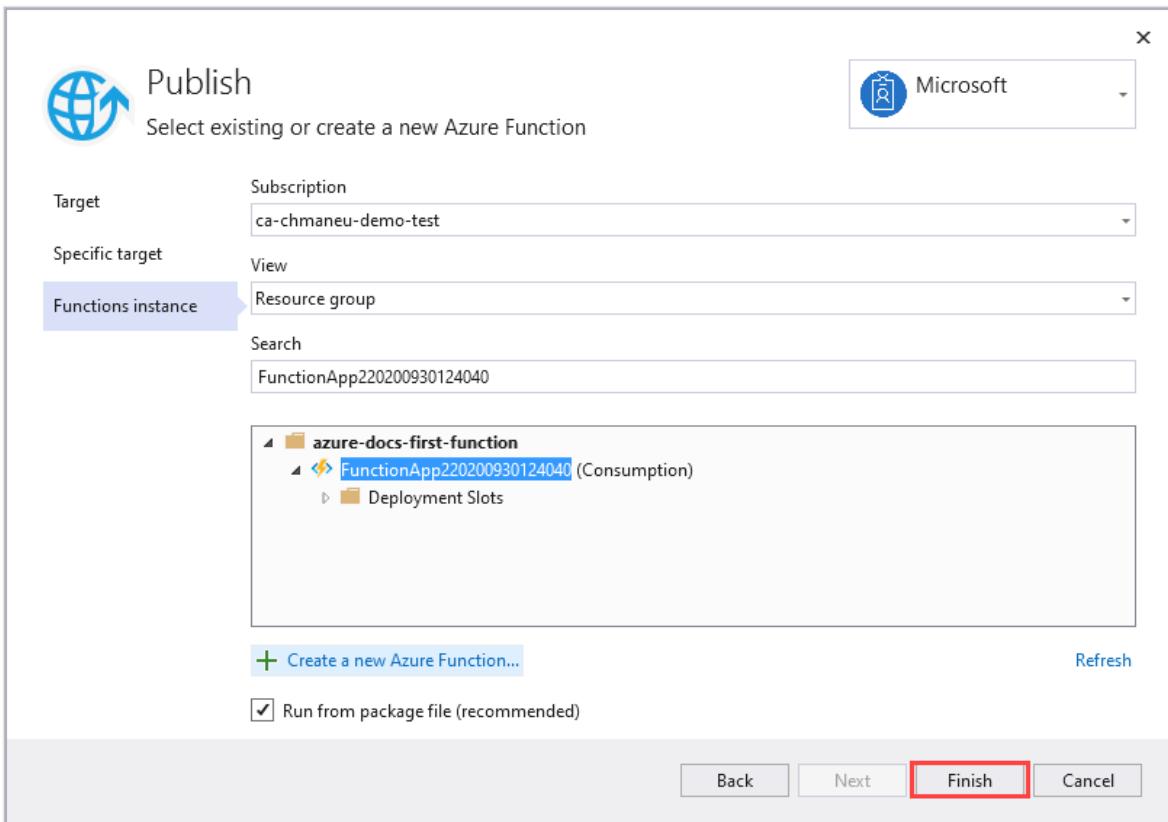
[\[+\] Expand table](#)

Setting	Value	Description
Name	Globally unique name	Name that uniquely identifies your new function app. Accept this name or enter a new name. Valid characters are: a-z, 0-9, and -.
Subscription	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the drop-down list.
Resource group	Name of your resource group	The resource group in which you want to create your function app. Select New to create a new resource group. You can also choose an existing resource group from the drop-down list.
Plan Type	Consumption	When you publish your project to a function app that runs in a Consumption plan , you pay only for executions of your functions app. Other hosting plans incur higher costs.
Location	Location of the app service	Choose a Location in a region near you or other services your functions access.
Azure Storage	General-purpose storage account	An Azure storage account is required by the Functions runtime. Select New to configure a general-purpose

Setting	Value	Description
		storage account. You can also choose an existing account that meets the storage account requirements .
Application Insights	Application Insights instance	You should enable Application Insights integration for your function app. Select New to create a new instance, either in a new or in an existing Log Analytics workspace. You can also choose an existing instance.

The screenshot shows the Azure portal's "Create a new Function App" dialog. The app is named "FunctionApp1320240130122517". It is set to use the "Visual Studio Enterprise" subscription and the "glengatest-new-app-vs*" resource group. The plan type is "Consumption", and the location is "Australia Central". Under "Azure Storage", the storage account "functionapp1320240130122* (Australia Central)" is selected. Under "Application Insights", the application insights instance "FunctionApp1320240130122517* (Australia Central)" is selected. At the bottom right, the "Create" button is highlighted with a red box.

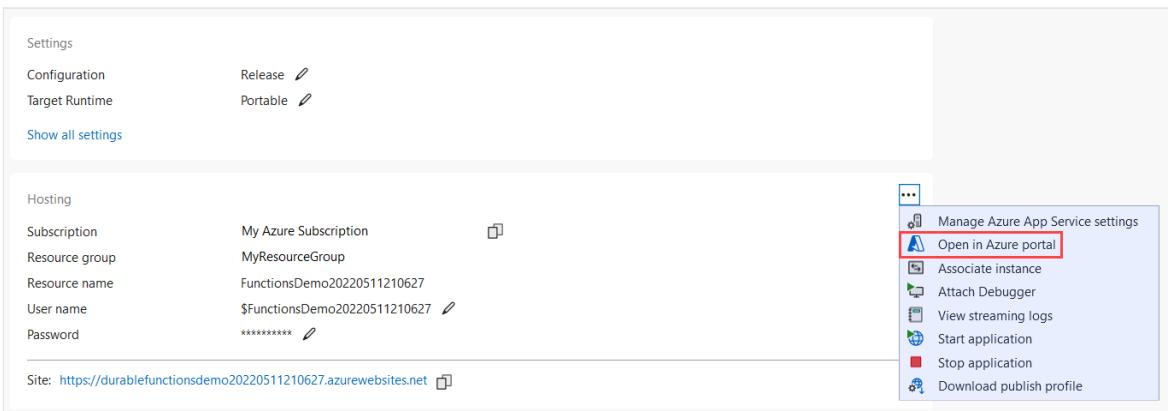
5. Select **Create** to create a function app and its related resources in Azure. The status of resource creation is shown in the lower-left of the window.
6. In the **Functions instance**, make sure that the **Run from package file** is checked. Your function app is deployed using [Zip Deploy](#) with [Run-From-Package](#) mode enabled. Zip Deploy is the recommended deployment method for your functions project resulting in better performance.



7. Select **Finish**, and on the Publish page, select **Publish** to deploy the package containing your project files to your new function app in Azure.

After the deployment completes, the root URL of the function app in Azure is shown in the **Publish** tab.

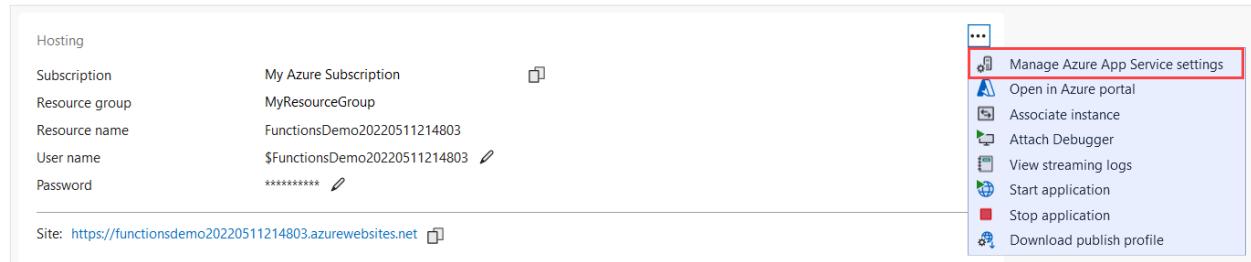
8. In the Publish tab, in the Hosting section, choose **Open in Azure portal**. This opens the new function app Azure resource in the Azure portal.



Function app settings

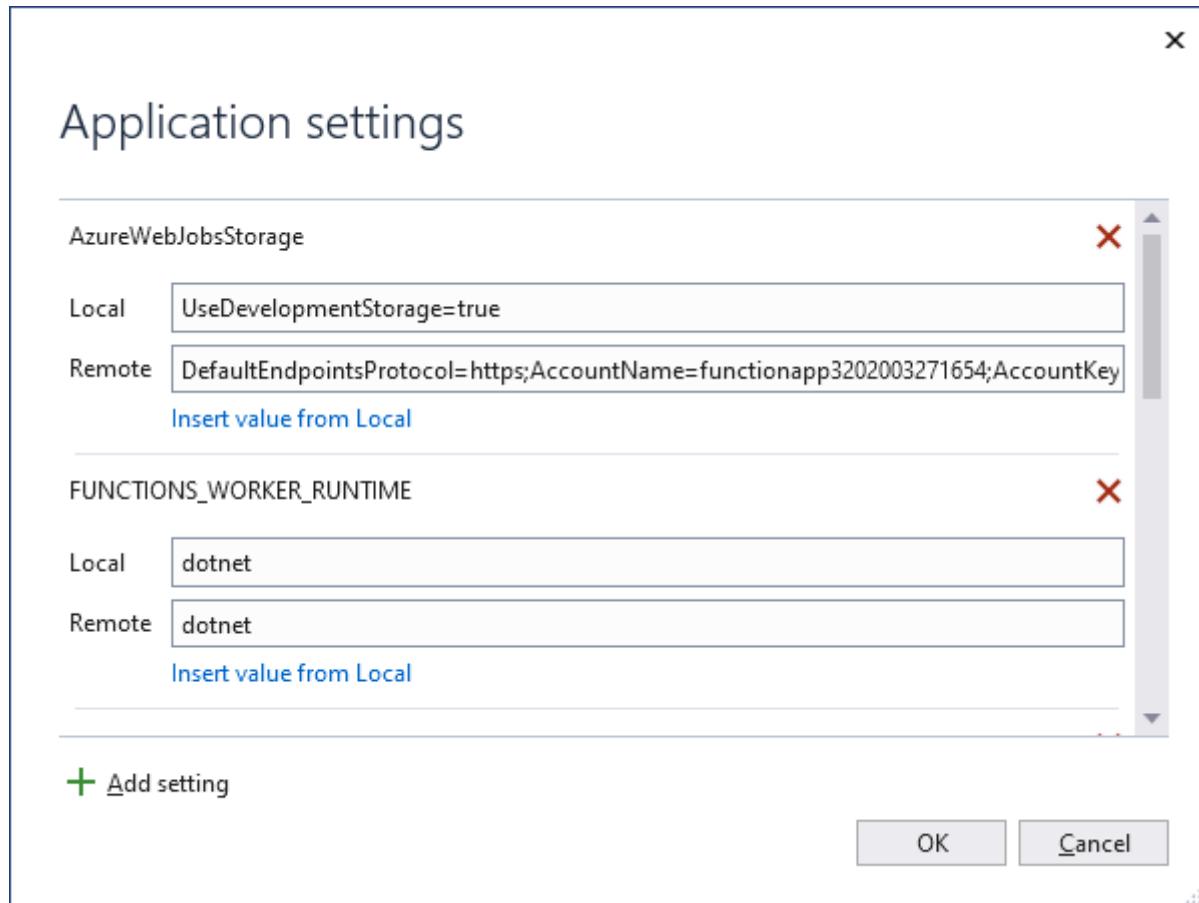
Visual Studio doesn't upload these settings automatically when you publish the project. Any settings you add in the local.settings.json you must also add to the function app in Azure.

The easiest way to upload the required settings to your function app in Azure is to expand the three dots next to the **Hosting** section and select the **Manage Azure App Service settings** link that appears after you successfully publish your project.



A screenshot of the Azure portal's 'Hosting' blade for a function app named 'FunctionsDemo20220511214803'. The 'Subscription' is set to 'My Azure Subscription' and 'Resource group' is 'MyResourceGroup'. The 'Resource name' is 'FunctionsDemo20220511214803', 'User name' is '\$FunctionsDemo20220511214803', and 'Password' is masked. Below the resource details is a 'Site' URL: <https://functionsdemo20220511214803.azurewebsites.net>. To the right of the resource details is a context menu with several options: 'Manage Azure App Service settings' (highlighted with a red box), 'Open in Azure portal', 'Associate instance', 'Attach Debugger', 'View streaming logs', 'Start application', 'Stop application', and 'Download publish profile'.

Selecting this link displays the **Application settings** dialog for the function app, where you can add new application settings or modify existing ones.



A screenshot of the 'Application settings' dialog. It contains two sections:

- AzureWebJobsStorage**:
 - Local**: Value: `UseDevelopmentStorage=true`
 - Remote**: Value: `DefaultEndpointsProtocol=https;AccountName=functionapp3202003271654;AccountKey=[REDACTED]`
Link: [Insert value from Local](#)
- FUNCTIONS_WORKER_RUNTIME**:
 - Local**: Value: `dotnet`
 - Remote**: Value: `dotnet`
Link: [Insert value from Local](#)

At the bottom left is a green plus icon with the text [Add setting](#). At the bottom right are 'OK' and 'Cancel' buttons.

Local displays a setting value in the local.settings.json file, and **Remote** displays a current setting value in the function app in Azure. Choose **Add setting** to create a new app setting. Use the **Insert value from Local** link to copy a setting value to the **Remote** field. Pending changes are written to the local settings file and the function app when you select **OK**.

ⓘ Note

By default, the local.settings.json file is not checked into source control. This means that if you clone a local Functions project from source control, the project doesn't

have a local.settings.json file. In this case, you need to manually create the local.settings.json file in the project root so that the **Application settings** dialog works as expected.

You can also manage application settings in one of these other ways:

- [Use the Azure portal](#).
- [Use the --publish-local-settings publish option in the Azure Functions Core Tools](#).
- [Use the Azure CLI](#).

Remote Debugging

To debug your function app remotely, you must publish a debug configuration of your project. You also need to enable remote debugging in your function app in Azure.

This section assumes you've already published to your function app using a release configuration.

Remote debugging considerations

- Remote debugging isn't recommended on a production service.
- If you have [Just My Code debugging](#) enabled, disable it.
- Avoid long stops at breakpoints when remote debugging. Azure treats a process that is stopped for longer than a few minutes as an unresponsive process, and shuts it down.
- While you're debugging, the server is sending data to Visual Studio, which could affect bandwidth charges. For information about bandwidth rates, see [Azure Pricing ↗](#).
- Remote debugging is automatically disabled in your function app after 48 hours. After 48 hours, you'll need to reenable remote debugging.

Attach the debugger

The way you attach the debugger depends on your execution mode. When debugging an isolated worker process app, you currently need to attach the remote debugger to a separate .NET process, and several other configuration steps are required.

When you're done, you should [disable remote debugging](#).

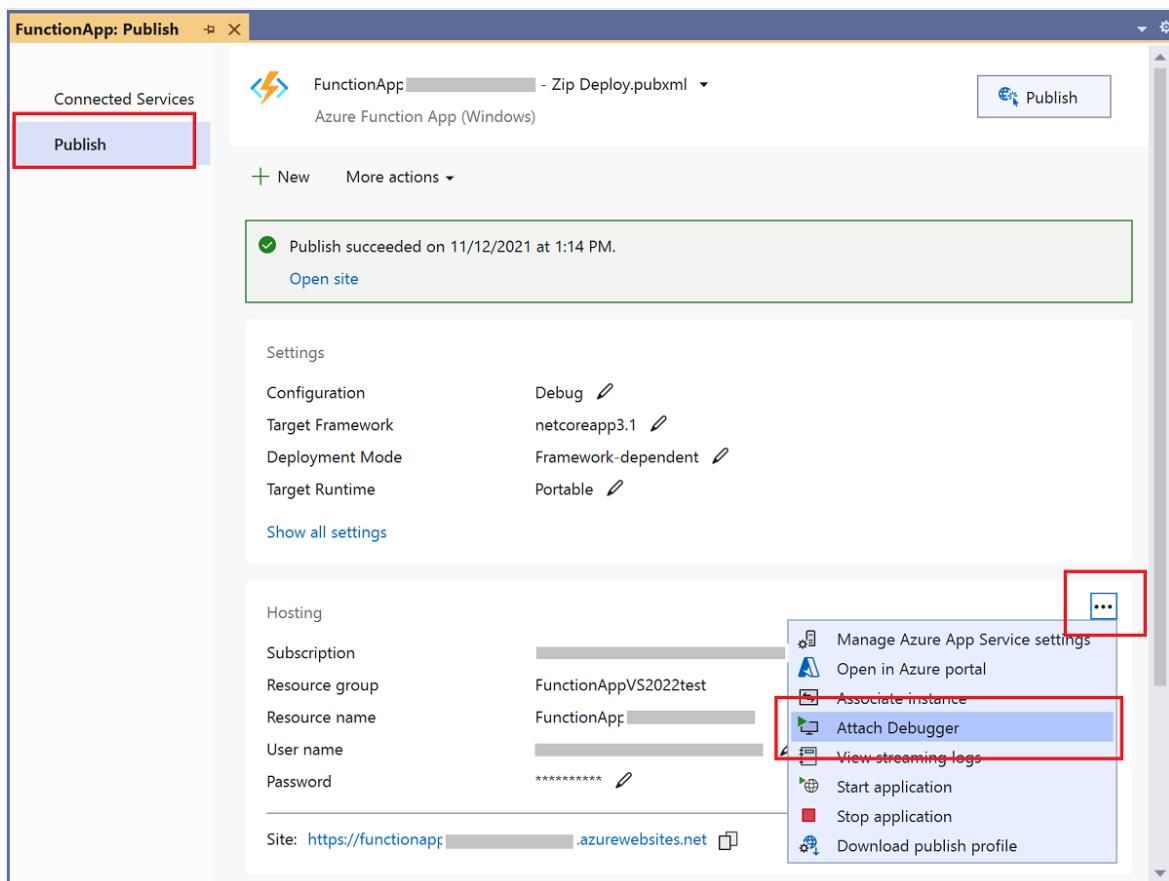
To attach a remote debugger to a function app running in a process separate from the Functions host:

1. From the **Publish** tab, select the ellipses (...) in the **Hosting** section, and then choose **Download publish profile**. This action downloads a copy of the publish profile and opens the download location. You need this file, which contains the credentials used to attach to your isolated worker process running in Azure.

✖ Caution

The .publishsettings file contains your credentials (unencoded) that are used to administer your function app. The security best practice for this file is to store it temporarily outside your source directories (for example in the Libraries\Documents folder), and then delete it after it's no longer needed. A malicious user who gains access to the .publishsettings file can edit, create, and delete your function app.

2. Again from the **Publish** tab, select the ellipses (...) in the **Hosting** section, and then choose **Attach debugger**.



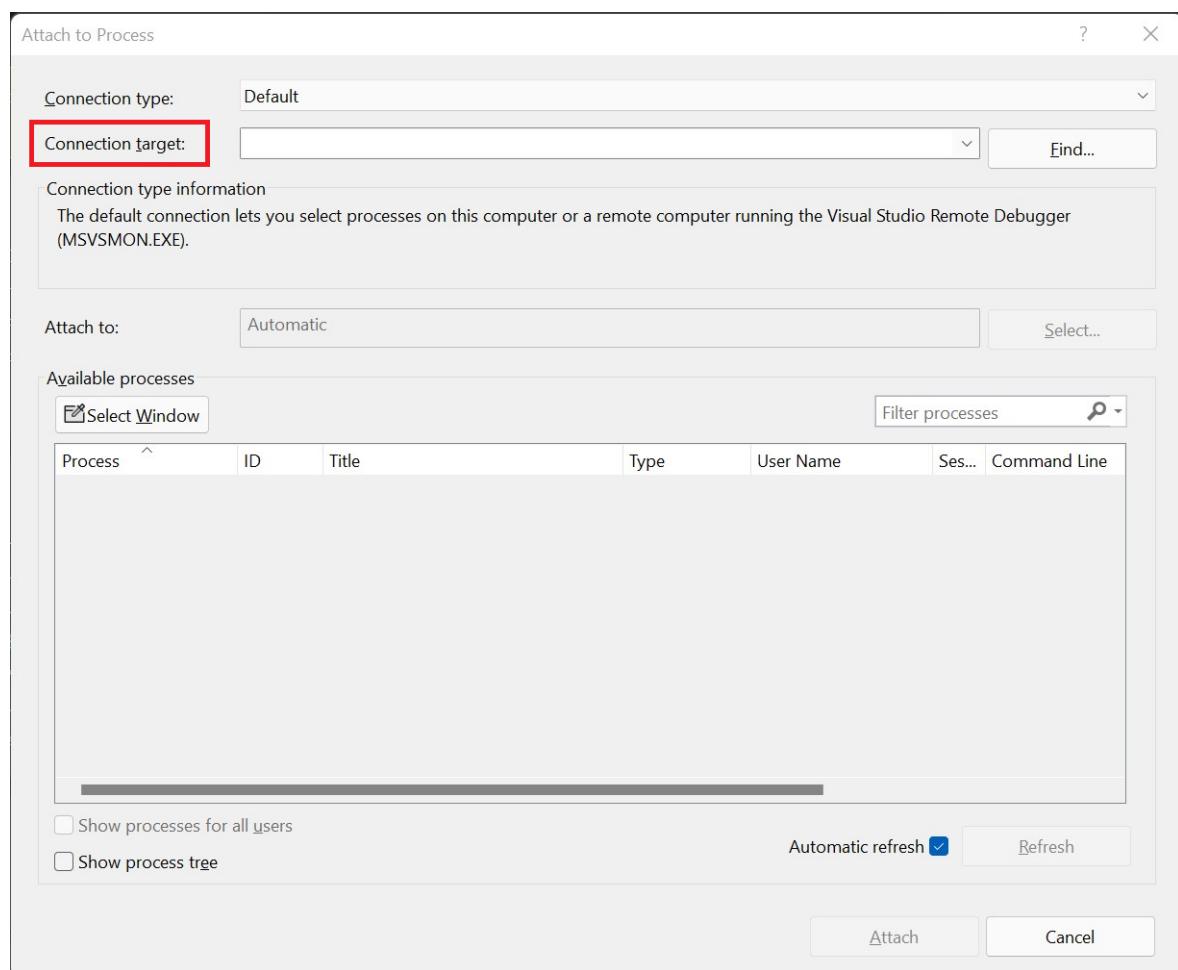
Visual Studio connects to your function app and enables remote debugging, if not already enabled.

ⓘ Note

Because the remote debugger isn't able to connect to the host process, you could see an error. In any case, the default debugging won't break into your code.

3. Back in Visual Studio, copy the URL for the **Site under Hosting** in the **Publish** page.
4. From the **Debug** menu, select **Attach to Process**, and in the **Attach to process** window, paste the URL in the **Connection Target**, remove `https://` and append the port `:4024`.

Verify that your target looks like `<FUNCTION_APP>.azurewebsites.net:4024` and press **Enter**.



5. If prompted, allow Visual Studio access through your local firewall.
6. When prompted for credentials, instead of local user credentials choose a different account (**More choices** on Windows). Provide the values of **userName** and **userPWD** from the published profile for **Email address** and **Password** in the authentication dialog on Windows. After a secure connection is established with the deployment server, the available processes are shown.



Windows Security

Enter your credentials

Visual Studio was unable to create a secure connection to net5-consumption-remote-debugging.scm.azurewebsites.net:4024. Authentication failed.

To retry, enter your credentials for net5-consumption-remote-debugging.scm.azurewebsites.net.

Email address **userName**

Password **userPWD**

Remember my credentials

7. Check **Show process from all users** and then choose **dotnet.exe** and select **Attach**. When the operation completes, you're attached to your C# class library code running in an isolated worker process. At this point, you can debug your function app as normal.

Disable remote debugging

After you're done remote debugging your code, you should disable remote debugging in the [Azure portal](#). Remote debugging is automatically disabled after 48 hours, in case you forget.

1. In the **Publish** tab in your project, select the ellipses (...) in the **Hosting** section, and choose **Open in Azure portal**. This action opens the function app in the Azure portal to which your project is deployed.
2. In the functions app, select **Configuration** under **settings**, choose **General Settings**, set **Remote Debugging** to **Off**, and select **Save** then **Continue**.

After the function app restarts, you can no longer remotely connect to your remote processes. You can use this same tab in the Azure portal to enable remote debugging outside of Visual Studio.

Monitoring functions

The recommended way to monitor the execution of your functions is by integrating your function app with Azure Application Insights. You should enable this integration when you create your function app during Visual Studio publishing.

If for some reason the integration wasn't done during publishing, you should still [enable Application Insights integration](#) for your function app in Azure.

To learn more about monitoring using Application Insights, see [Monitor Azure Functions](#).

Next steps

For more information about the Azure Functions Core Tools, see [Work with Azure Functions Core Tools](#).

[C# isolated worker model guide](#)

Develop Azure Functions locally using Core Tools

Article • 11/14/2023

Azure Functions Core Tools lets you develop and test your functions on your local computer. When you're ready, you can also use Core Tools to deploy your code project to Azure and work with application settings.

You're viewing the C# version of this article. Make sure to select your preferred Functions programming language at the top of the article.

If you want to get started right away, complete the [Core Tools quickstart article](#).

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. Visual Studio Code debugging requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

For help with version-related issues, see [Core Tools versions](#).

Create your local project

In the terminal window or from a command prompt, run the following command to create a project in the `MyProjFolder` folder:

Isolated process

Console

```
func init MyProjFolder --worker-runtime dotnet-isolated
```

By default this command creates a project that runs in-process with the Functions host on the current [Long-Term Support \(LTS\) version of .NET Core](#). You can use the `--target-framework` option to target a specific supported version of .NET, including .NET Framework. For more information, see the [func init](#) reference.

For a comparison between the two .NET process models, see the [process mode comparison article](#).

When you run `func init` without the `--worker-runtime` option, you're prompted to choose your project language. To learn more about the available options for the `func init` command, see the [func init](#) reference.

Create a function

To add a function to your project, run the `func new` command using the `--template` option to select your trigger template. The following example creates an HTTP trigger named `MyHttpTrigger`:

```
func new --template "Http Trigger" --name MyHttpTrigger
```

This example creates a Queue Storage trigger named `MyQueueTrigger`:

```
func new --template "Azure Queue Storage Trigger" --name MyQueueTrigger
```

The following considerations apply when adding functions:

- When you run `func new` without the `--template` option, you're prompted to choose a template.

- Use the [func templates list](#) command to see the complete list of available templates for your language.
- When you add a trigger that connects to a service, you'll also need to add an application setting that references a connection string or a managed identity to the local.settings.json file. Using app settings in this way prevents you from having to embed credentials in your code. For more information, see [Work with app settings locally](#).
- Core Tools also adds a reference to the specific binding extension to your C# project.

To learn more about the available options for the `func new` command, see the [func new](#) reference.

Add a binding to your function

Functions provides a set of service-specific input and output bindings, which make it easier for your function to connect to other Azure services without having to use the service-specific client SDKs. For more information, see [Azure Functions triggers and bindings concepts](#).

To add an input or output binding to an existing function, you must manually update the function definition.

The following example shows the function definition after adding a [Queue Storage output binding](#) to an [HTTP triggered function](#):

Isolated process

Because an HTTP triggered function also returns an HTTP response, the function returns a `MultiResponse` object, which represents both the HTTP and queue output.

```
C#
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
{
```

This example is the definition of the `MultiResponse` object that includes the output binding:

```
C#
```

```
public class MultiResponse
{
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

Messages are sent to the queue when the function completes. The way you define the output binding depends on your process model. For more information, including links to example binding code that you can refer to, see [Add bindings to a function](#).

The following considerations apply when adding bindings to a function:

- When you add bindings that connect to a service, you must also add an application setting that references a connection string or managed identity to the local.settings.json file. For more information, see [Work with app settings locally](#).
- When you add a binding that requires a new binding extension, you must also add a reference to that specific binding extension in your C# project.

For more information, including links to example binding code that you can refer to, see [Add bindings to a function](#).

Start the Functions runtime

Before you can run or debug the functions in your project, you need to start the Functions host from the root directory of your project. The host enables triggers for all functions in the project. Use this command to start the local runtime:

```
func start
```

When the Functions host starts, it outputs a list of functions in the project, including the URLs of any HTTP-triggered functions, like in this example:

```
Found the following functions:
Host.Functions.MyHttpTrigger

Job host started
Http Function MyHttpTrigger: http://localhost:7071/api/MyHttpTrigger
```

Keep in mind the following considerations when running your functions locally:

- By default, authorization isn't enforced locally for HTTP endpoints. This means that all local HTTP requests are handled as `authLevel = "anonymous"`. For more information, see the [HTTP binding article](#). You can use the `--enableAuth` option to require authorization when running locally. For more information, see [func start](#)
- While there's local storage emulation available, it's often best to validate your triggers and bindings against live services in Azure. You can maintain the connections to these services in the `local.settings.json` project file. For more information, see [Local settings file](#). Make sure to keep test and production data separate when testing against live Azure services.
- You can trigger non-HTTP functions locally without connecting to a live service. For more information, see [Run a local function](#).
- When you include your Application Insights connection information in the `local.settings.json` file, local log data is written to the specific Application Insights instance. To keep local telemetry data separate from production data, consider using a separate Application Insights instance for development and testing.
- When using version 1.x of the Core Tools, instead use the `func host start` command to start the local runtime.

Run a local function

With your local Functions host (`func.exe`) running, you can now trigger individual functions to run and debug your function code. The way in which you execute an individual function depends on its trigger type.

Note

Examples in this topic use the cURL tool to send HTTP requests from the terminal or a command prompt. You can use a tool of your choice to send HTTP requests to the local server. The cURL tool is available by default on Linux-based systems and Windows 10 build 17063 and later. On older Windows, you must first download and install the [cURL tool](#).

HTTP trigger

HTTP triggers are started by sending an HTTP request to the local endpoint and port as displayed in the func.exe output, which has this general format:

```
http://localhost:<PORT>/api/<FUNCTION_NAME>
```

In this URL template, <FUNCTION_NAME> is the name of the function or route and <PORT> is the local port on which func.exe is listening.

For example, this cURL command triggers the `MyHttpTrigger` quickstart function from a GET request with the *name* parameter passed in the query string:

```
curl --get http://localhost:7071/api/MyHttpTrigger?name=Azure%20Rocks
```

This example is the same function called from a POST request passing *name* in the request body, shown for both Bash shell and Windows command line:

Bash

```
curl --request POST http://localhost:7071/api/MyHttpTrigger --data  
'{"name": "Azure Rocks"}'
```

Windows Command Prompt

```
curl --request POST http://localhost:7071/api/MyHttpTrigger --data "  
'name': 'Azure Rocks'"
```

The following considerations apply when calling HTTP endpoints locally:

- You can make GET requests from a browser passing data in the query string. For all other HTTP methods, you must use cURL, Fiddler, Postman, or a similar HTTP testing tool that supports POST requests.
- Make sure to use the same server name and port that the Functions host is listening on. You see an endpoint like this in the output generated when starting the Function host. You can call this URL using any HTTP method supported by the trigger.

Publish to Azure

The Azure Functions Core Tools supports three types of deployment:

Deployment type	Command	Description
Project files	<code>func azure functionapp publish</code>	Deploys function project files directly to your function app using zip deployment .
Azure Container Apps	<code>func azurecontainerapps deploy</code>	Deploys a containerized function app to an existing Container Apps environment.
Kubernetes cluster	<code>func kubernetes deploy</code>	Deploys your Linux function app as a custom Docker container to a Kubernetes cluster.

You must have either the [Azure CLI](#) or [Azure PowerShell](#) installed locally to be able to publish to Azure from Core Tools. By default, Core Tools uses these tools to authenticate with your Azure account.

If you don't have these tools installed, you need to instead [get a valid access token](#) to use during deployment. You can present an access token using the `--access-token` option in the deployment commands.

Deploy project files

To publish your local code to a function app in Azure, use the [`func azure functionapp publish`](#) command, as in the following example:

```
func azure functionapp publish <FunctionAppName>
```

This command publishes project files from the current directory to the `<FunctionAppName>` as a .zip deployment package. If the project requires compilation, it's done remotely during deployment.

The following considerations apply to this kind of deployment:

- Publishing overwrites existing files in the remote function app deployment.
- You must have already [created a function app in your Azure subscription](#). Core Tools deploys your project code to this function app resource. To learn how to create a function app from the command prompt or terminal window using the

Azure CLI or Azure PowerShell, see [Create a Function App for serverless execution](#).

You can also [create these resources in the Azure portal](#). You get an error when you try to publish to a <FunctionAppName> that doesn't exist in your subscription.

- A project folder may contain language-specific files and directories that shouldn't be published. Excluded items are listed in a .funcignore file in the root project folder.
- By default, your project is deployed so that it [runs from the deployment package](#). To disable this recommended deployment mode, use the [--nozip option](#).
- A [remote build](#) is performed on compiled projects. This can be controlled by using the [--no-build option](#).
- Use the [--publish-local-settings](#) option to automatically create app settings in your function app based on values in the local.settings.json file.
- To publish to a specific named slot in your function app, use the [--slot option](#).

Deploy containers

Core Tools lets you deploy your [containerized function app](#) to both managed Azure Container Apps environments and Kubernetes clusters that you manage.

Container Apps

Use the following [func azurecontainerapps deploy](#) command to deploy an existing container image to a Container Apps environment:

command

```
func azurecontainerapps deploy --name <APP_NAME> --environment <ENVIRONMENT_NAME> --storage-account <STORAGE_CONNECTION> --resource-group <RESOURCE_GROUP> --image-name <IMAGE_NAME> [--registry-password] [--registry-server] [--registry-username]
```

When you deploy to an Azure Container Apps environment, the following considerations apply:

- The environment and storage account must already exist. The storage account connection string you provide is used by the deployed function app.

- You don't need to create a separate function app resource when deploying to Container Apps.
- Storage connection strings and other service credentials are important secrets. Make sure to securely store any script files using `func azurecontainerapps deploy` and don't store them in any publicly accessible source control systems. You can [encrypt the local.settings.json file](#) for added security.

For more information, see [Azure Container Apps hosting of Azure Functions](#).

Work with app settings locally

When running in a function app in Azure, settings required by your functions are [stored securely in app settings](#). During local development, these settings are instead added to the `Values` collection in the local.settings.json file. The local.settings.json file also stores settings used by local development tools.

Items in the `Values` collection in your project's local.settings.json file are intended to mirror items in your function app's [application settings](#) in Azure.

The following considerations apply when working with the local settings file:

- Because the local.settings.json may contain secrets, such as connection strings, you should never store it in a remote repository. Core Tools helps you encrypt this local settings file for improved security. For more information, see [Local settings file](#). You can also [encrypt the local.settings.json file](#) for added security.
- By default, local settings aren't migrated automatically when the project is published to Azure. Use the `--publish-local-settings` option when you publish your project files to make sure these settings are added to the function app in Azure. Values in the `ConnectionStrings` section are never published. You can also [upload settings from the local.settings.json file](#) at any time.
- You can download and overwrite settings in your local.settings.json file with settings from your function app in Azure. For more information, see [Download application settings](#).
- The function app settings values can also be read in your code as environment variables. For more information, see [Environment variables](#).
- When no valid storage connection string is set for `AzureWebJobsStorage` and a local storage emulator isn't being used, an error is shown. You can use Core Tools to [download a specific connection string](#) from any of your Azure Storage accounts.

Download application settings

From the project root, use the following command to download all application settings from the `myfunctionapp12345` app in Azure:

command

```
func azure functionapp fetch-app-settings myfunctionapp12345
```

This command overwrites any existing settings in the `local.settings.json` file with values from Azure. When not already present, new items are added to the collection. For more information, see the [func azure functionapp fetch-app-settings](#) command.

Download a storage connection string

Core Tools also make it easy to get the connection string of any storage account to which you have access. From the project root, use the following command to download the connection string from a storage account named `mystorage12345`.

command

```
func azure storage fetch-connection-string mystorage12345
```

This command adds a setting named `mystorage12345_STORAGE` to the `local.settings.json` file, which contains the connection string for the `mystorage12345` account. For more information, see the [func azure storage fetch-connection-string](#) command.

For improved security during development, consider [encrypting the local.settings.json file](#).

Upload local settings to Azure

When you publish your project files to Azure without using the `--publish-local-settings` option, settings in the `local.settings.json` file aren't set in your function app.

You can always rerun the `func azure functionapp publish` with the `--publish-settings-only` option to upload just the settings without republishing the project files.

The following example uploads just settings from the `Values` collection in the `local.settings.json` file to the function app in Azure named `myfunctionapp12345`:

command

```
func azure functionapp publish myfunctionapp12345 --publish-settings-only
```

Encrypt the local settings file

To improve security of connection strings and other valuable data in your local settings, Core Tools lets you encrypt the local.settings.json file. When this file is encrypted, the runtime automatically decrypts the settings when needed the same way it does with application setting in Azure. You can also decrypt a locally encrypted file to work with the settings.

Use the following command to encrypt the local settings file for the project:

command

```
func settings encrypt
```

Use the following command to decrypt an encrypted local setting, so that you can work with it:

command

```
func settings decrypt
```

When the settings file is encrypted and decrypted, the file's `IsEncrypted` setting also gets updated.

Configure binding extensions

Functions triggers and bindings are implemented as .NET extension (NuGet) packages. To be able to use a specific binding extension, that extension must be installed in the project.

This section doesn't apply to version 1.x of the Functions runtime. In version 1.x, supported binding were included in the core product extension.

For C# class library projects, add references to the specific NuGet packages for the binding extensions required by your functions. C# script (.csx) project must use [extension bundles](#).

Core Tools versions

Major versions of Azure Functions Core Tools are linked to specific major versions of the Azure Functions runtime. For example, version 4.x of Core Tools supports version 4.x of the Functions runtime. This version is the recommended major version of both the Functions runtime and Core Tools. You can determine the latest release version of Core Tools in the [Azure Functions Core Tools repository](#).

Run the following command to determine the version of your current Core Tools installation:

```
command
```

```
func --version
```

Unless otherwise noted, the examples in this article are for version 4.x.

The following considerations apply to Core Tools installations:

- You can only install one version of Core Tools on a given computer.
- When upgrading to the latest version of Core Tools, you should use the same method that you used for original installation to perform the upgrade. For example, if you used an MSI on Windows, uninstall the current MSI and install the latest one. Or if you used npm, rerun the `npm install` command.
- Version 2.x and 3.x of Core Tools were used with versions 2.x and 3.x of the Functions runtime, which have reached their end of life (EOL). For more information, see [Azure Functions runtime versions overview](#).
- Version 1.x of Core Tools is required when using version 1.x of the Functions Runtime, which is still supported. This version of Core Tools can only be run locally on Windows computers. If you're currently running on version 1.x, you should consider [migrating your app to version 4.x](#) today.

When using Visual Studio Code, you can integrate Rosetta with the built-in Terminal. For more information, see [Enable emulation in Visual Studio Code](#).

Next steps

Learn how to [develop, test, and publish Azure functions by using Azure Functions core tools](#). Azure Functions Core Tools is open source and hosted on [GitHub](#). To file a bug or feature request, [open a GitHub issue](#).

Create your first function in the Azure portal

Article • 09/11/2024

Azure Functions lets you run your code in a serverless environment without having to first create a virtual machine (VM) or publish a web application. In this article, you learn how to use Azure Functions to create a "hello world" HTTP trigger function in the Azure portal.

Choose your preferred programming language at the top of the article.

ⓘ Note

Editing your C# function code in the Azure portal is currently only supported for [C# script \(.csx\) functions](#). To learn more about the limitations on editing function code in the Azure portal, see [Development limitations in the Azure portal](#).

You should instead [develop your functions locally](#) and publish to a function app in Azure. Use one of the following links to get started with your chosen local development environment:

- [Visual Studio](#)
- [Visual Studio Code](#)
- [Terminal/command prompt](#)

Please review the [known issues](#) for development of Azure Functions using Python in the Azure portal.

Prerequisites

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Sign in to Azure

Sign in to the [Azure portal](#) with your Azure account.

Create a function app

You must have a function app to host the execution of your functions. A function app lets you group functions as a logical unit for easier management, deployment, scaling, and sharing of resources.

Use these steps to create your function app and related Azure resources, whether or not you're able to edit your code in the Azure portal.

To be able to create a C# script app that you can edit in the portal, choose **8 (LTS), in-process model** for **.NET Version**.

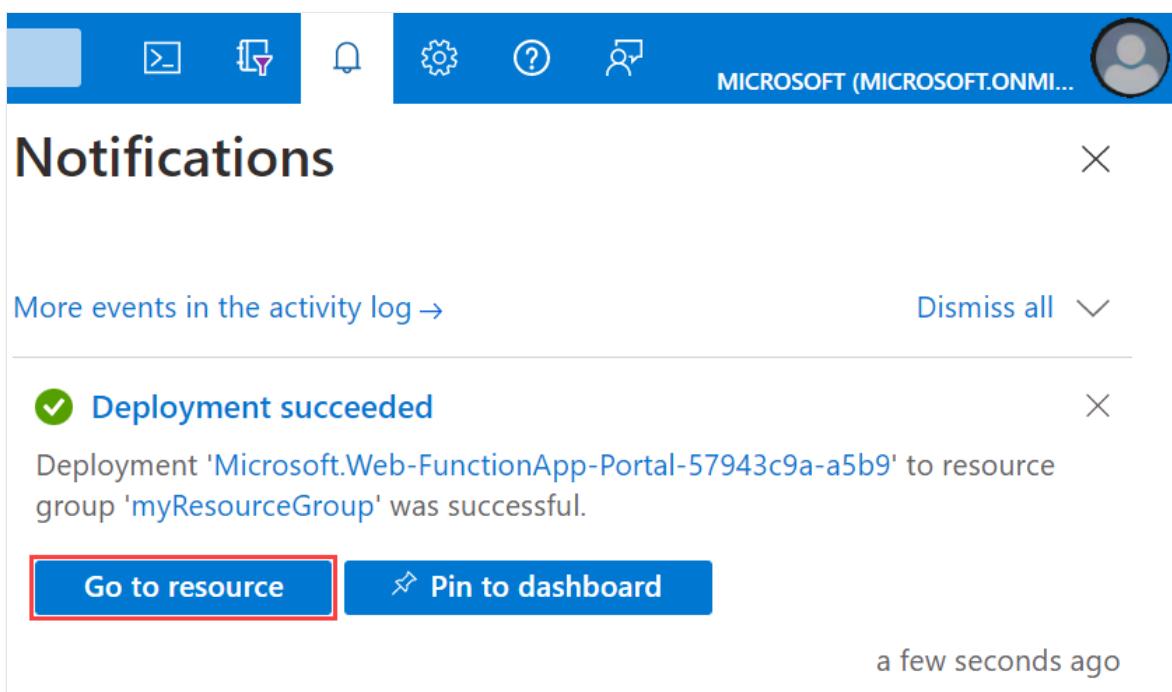
1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. Under **Select a hosting option**, select **Consumption > Select** to create your app in the default **Consumption** plan. In this [serverless](#) hosting option, you pay only for the time your functions run. [Premium plan](#) also offers dynamic scaling. When you run in an App Service plan, you must manage the [scaling of your function app](#).
4. On the **Basics** page, use the function app settings as specified in the following table:

[+] [Expand table](#)

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which you create your new function app.
Resource Group	<code>myResourceGroup</code>	Name for the new resource group in which you create your function app. You should create a new resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, Python, TypeScript, and C# script. To create a C# Script app that supports in-portal editing, you must choose a runtime Version that supports the in-process model . C# class library and Java functions must be developed locally .

Setting	Suggested value	Description
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Select a region that's near you or near other services that your functions can access.
Operating system	Windows	An operating system is preselected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows.

5. Accept the default options in the remaining tabs, including the default behavior of creating a new storage account on the **Storage** tab and a new Application Insight instance on the **Monitoring** tab. You can also choose to use an existing storage account or Application Insights instance.
6. Select **Review + create** to review the app configuration you chose, and then select **Create** to provision and deploy the function app.
7. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
8. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.



Next, create a function in the new function app.

Create an HTTP trigger function

1. In your function app, select **Overview**, and then select **+ Create** under **Functions**. If you don't see the **+ Create** button, you can instead [create your functions locally](#).
2. Under **Select a template**, scroll down and choose the **HTTP trigger** template.
3. In **Template details**, use `HttpExample` for **New Function**, select **Anonymous** from the **Authorization level** drop-down list, and then select **Create**.

Azure creates the HTTP trigger function. Now, you can run the new function by sending an HTTP request.

Create your functions locally

If you aren't able to create your function code in the portal, you can instead create a local project and publish the function code to your new function app.

1. In your function app, select **Overview**, and then in **Create functions in your preferred environment** under **Functions**.
2. Choose your preferred local development environment and follow the steps in the linked article to create and publish your first Azure Functions project.

💡 Tip

When publishing your new project, make sure to use the function app and related resources you just created.

Test the function

💡 Tip

The **Code + Test** functionality in the portal works even for functions that are read-only and can't be edited in the portal.

1. On the **Overview** page for your new function app, select your new HTTP triggered function in the **Functions** tab.
2. In the left menu, expand **Developer**, select **Code + Test**, and then select **Test/Run**.

3. In the Test/Run dialog, select Run.

An HTTP POST request is sent to your new function with a payload that contains the `name` value of `Azure`. You can also test the function by selecting **GET** for **HTTP method** and adding a `name` parameter with a value of `YOUR_NAME`.

💡 Tip

To test in an external browser, instead select **Get function URL**, copy the **default (Function key)** value, add the query string value `&name=<YOUR_NAME>` to the end of this URL, and then submit the URL in the address bar of your web browser.

4. When your function runs, trace information is written to the logs. To see the trace output, return to the **Code + Test** page in the portal and expand the **Logs** arrow at the bottom of the page. Call your function again to see the trace output written to the logs.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with options like 'Functions', 'App keys', 'App files', and 'Proxies'. The main content area is titled 'Overview' and shows details for a resource group named 'myResourceGroup'. The 'Resource group (change)' section is highlighted with a red box. It lists the following information:

- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags

On the right side, there are links for 'URL' (https://myfunctionapp.azurewebsites.net), 'Operating System' (Windows), 'App Service Plan' (ASP-myResourceGroup-a285 (Y1: 0)), 'Properties' (See More), and 'Runtime version' (3.0.13139.0). At the bottom, there are tabs for 'Metrics', 'Features (8)', 'Notifications (0)', and 'Quickstart'.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

Now that you've created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create your first Kotlin function in Azure using IntelliJ

Article • 01/07/2024

This article shows you how to create an HTTP-triggered Java function in an IntelliJ IDEA project, run and debug the project in the integrated development environment (IDE), and finally deploy the function project to a function app in Azure.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Set up your development environment

To create and publish Kotlin functions to Azure using IntelliJ, install the following software:

- [Java Developer Kit \(JDK\)](#), version 8
- [Apache Maven](#), version 3.0 or higher
- [IntelliJ IDEA](#), Community or Ultimate versions with Maven
- [Azure CLI](#)
- [Version 2.x](#) of the Azure Functions Core Tools. It provides a local development environment for writing, running, and debugging Azure Functions.

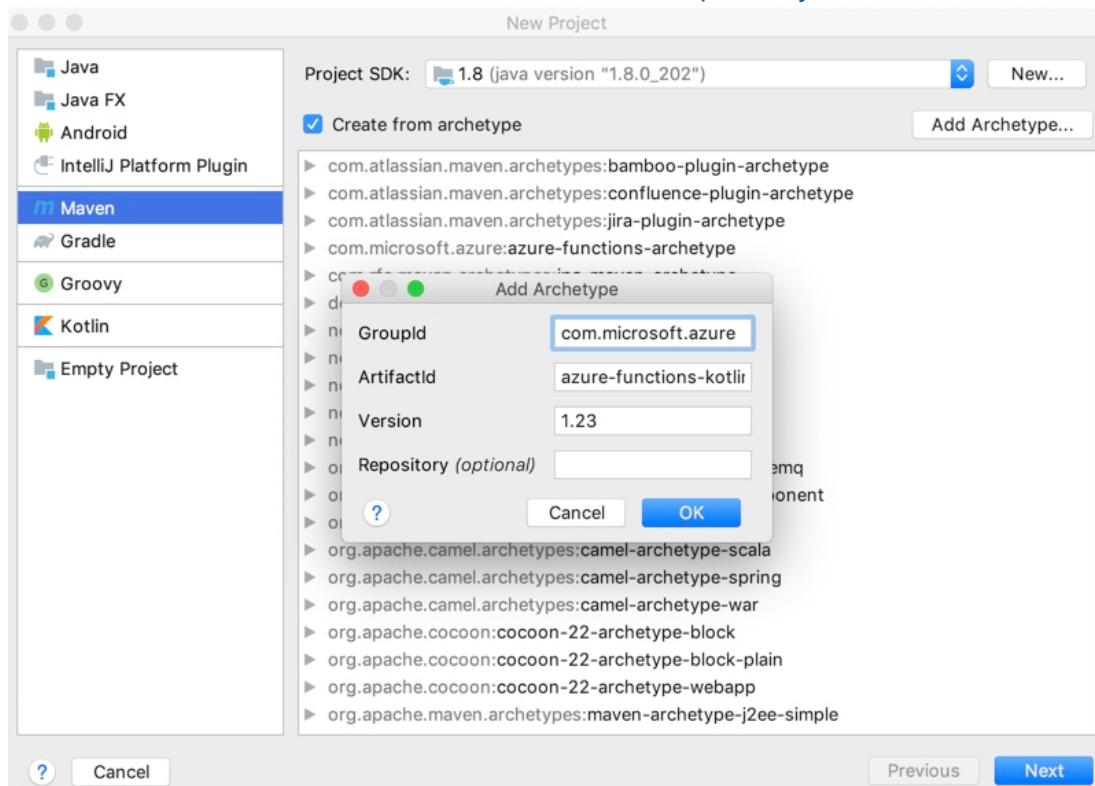
Important

The JAVA_HOME environment variable must be set to the install location of the JDK to complete the steps in this article.

Create a function project

1. In IntelliJ IDEA, select **Create New Project**.
2. In the **New Project** window, select **Maven** from the left pane.
3. Select the **Create from archetype** check box, and then select **Add Archetype** for the [azure-functions-kotlin-archetype](#).
4. In the **Add Archetype** window, complete the fields as follows:
 - *GroupId*: com.microsoft.azure
 - *ArtifactId*: azure-functions-kotlin-archetype

- **Version:** Use the latest version from [the central repository](#)



5. Select **OK**, and then select **Next**.
6. Enter your details for current project, and select **Finish**.

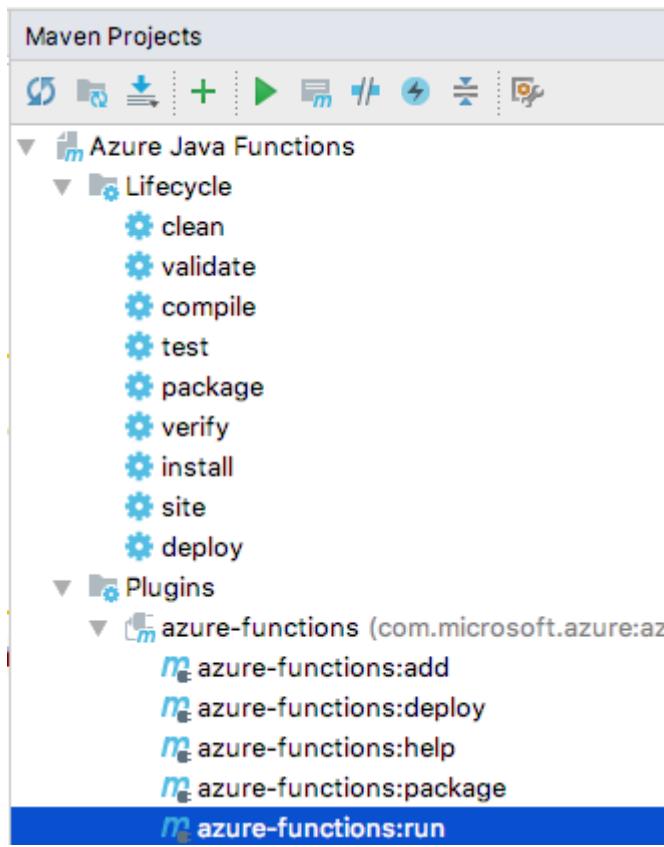
Maven creates the project files in a new folder with the same name as the *ArtifactId* value. The project's generated code is a simple [HTTP-triggered](#) function that echoes the body of the triggering HTTP request.

Run project locally in the IDE

i Note

To run and debug the project locally, make sure you've installed [Azure Functions Core Tools, version 2](#).

1. Import changes manually or enable [auto import](#).
2. Open the **Maven Projects** toolbar.
3. Expand **Lifecycle**, and then open **package**. The solution is built and packaged in a newly created target directory.
4. Expand **Plugins > azure-functions** and open **azure-functions:run** to start the Azure Functions local runtime.



5. Close the run dialog box when you're done testing your function. Only one function host can be active and running locally at a time.

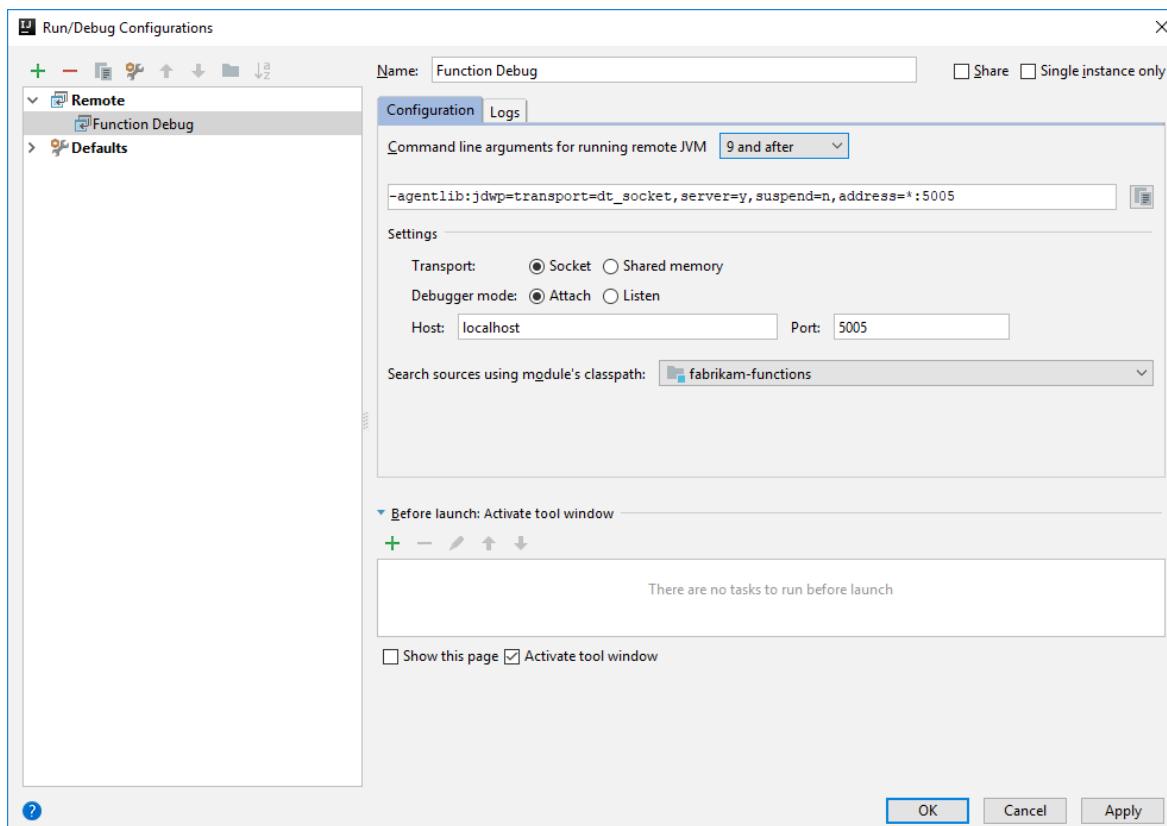
Debug the project in IntelliJ

1. To start the function host in debug mode, add `-DenableDebug` as the argument when you run your function. You can either change the configuration in [maven goals](#) or run the following command in a terminal window:

```
mvn azure-functions:run -DenableDebug
```

This command causes the function host to open a debug port at 5005.

2. On the Run menu, select **Edit Configurations**.
3. Select (+) to add a **Remote**.
4. Complete the *Name* and *Settings* fields, and then select **OK** to save the configuration.
5. After setup, select **Debug < Remote Configuration Name >** or press Shift+F9 on your keyboard to start debugging.



- When you're finished, stop the debugger and the running process. Only one function host can be active and running locally at a time.

Deploy the project to Azure

- Before you can deploy your project to a function app in Azure, you must [log in by using the Azure CLI](#).

```
Azure CLI
az login
```

- Deploy your code into a new function app by using the `azure-functions:deploy` Maven target. You can also select the `azure-functions:deploy` option in the Maven Projects window.

```
mvn azure-functions:deploy
```

- Find the URL for your HTTP trigger function in the Azure CLI output after the function app has been successfully deployed.

```
Output
```

```
[INFO] Successfully deployed Function App with package.  
[INFO] Deleting deployment package from Azure Storage...  
[INFO] Successfully deleted deployment package fabrikam-function-  
20170920120101928.20170920143621915.zip  
[INFO] Successfully deployed Function App at https://fabrikam-function-  
20170920120101928.azurewebsites.net  
[INFO] -----  
-----
```

Next steps

Now that you have deployed your first Kotlin function app to Azure, review the [Azure Functions Java developer guide](#) for more information on developing Java and Kotlin functions.

- Add additional function apps with different triggers to your project by using the `azure-functions:add` Maven target.

Create your first function in the Azure portal

Article • 09/11/2024

Azure Functions lets you run your code in a serverless environment without having to first create a virtual machine (VM) or publish a web application. In this article, you learn how to use Azure Functions to create a "hello world" HTTP trigger function in the Azure portal.

Choose your preferred programming language at the top of the article.

ⓘ Note

Editing your C# function code in the Azure portal is currently only supported for [C# script \(.csx\) functions](#). To learn more about the limitations on editing function code in the Azure portal, see [Development limitations in the Azure portal](#).

You should instead [develop your functions locally](#) and publish to a function app in Azure. Use one of the following links to get started with your chosen local development environment:

- [Visual Studio](#)
- [Visual Studio Code](#)
- [Terminal/command prompt](#)

Please review the [known issues](#) for development of Azure Functions using Python in the Azure portal.

Prerequisites

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Sign in to Azure

Sign in to the [Azure portal](#) with your Azure account.

Create a function app

You must have a function app to host the execution of your functions. A function app lets you group functions as a logical unit for easier management, deployment, scaling, and sharing of resources.

Use these steps to create your function app and related Azure resources, whether or not you're able to edit your code in the Azure portal.

To be able to create a C# script app that you can edit in the portal, choose **8 (LTS)**, **in-process model** for **.NET Version**.

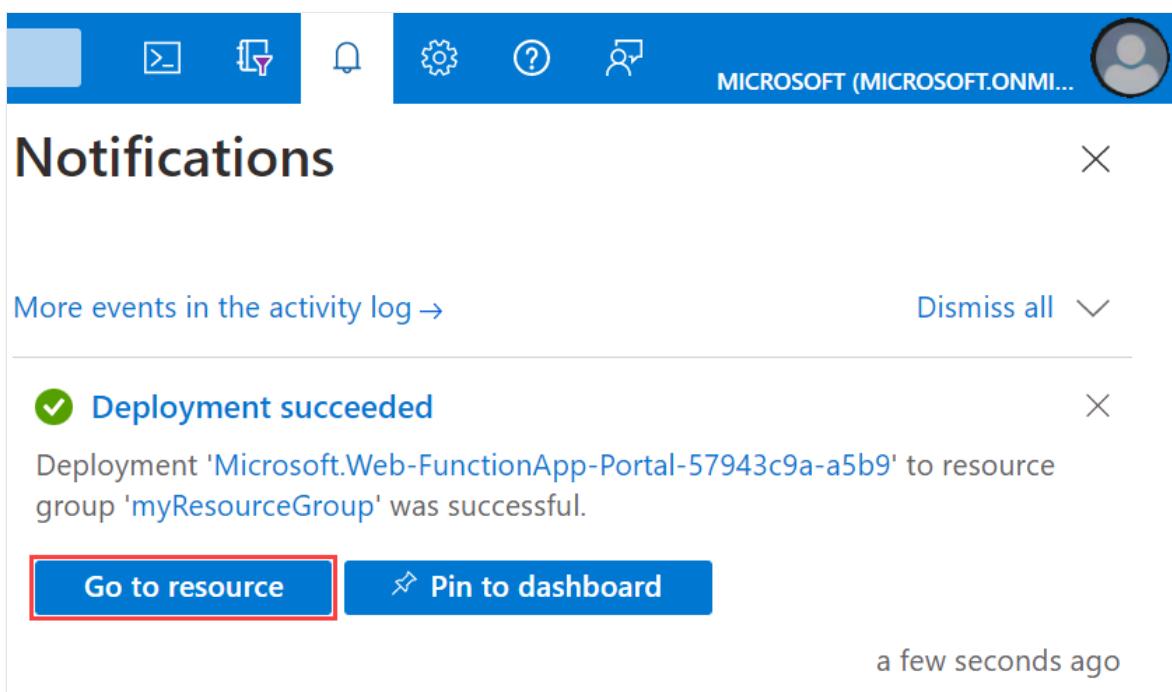
1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. Under **Select a hosting option**, select **Consumption > Select** to create your app in the default **Consumption** plan. In this [serverless](#) hosting option, you pay only for the time your functions run. [Premium plan](#) also offers dynamic scaling. When you run in an App Service plan, you must manage the [scaling of your function app](#).
4. On the **Basics** page, use the function app settings as specified in the following table:

[+] [Expand table](#)

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which you create your new function app.
Resource Group	<code>myResourceGroup</code>	Name for the new resource group in which you create your function app. You should create a new resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, Python, TypeScript, and C# script. To create a C# Script app that supports in-portal editing, you must choose a runtime Version that supports the in-process model . C# class library and Java functions must be developed locally .

Setting	Suggested value	Description
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Select a region that's near you or near other services that your functions can access.
Operating system	Windows	An operating system is preselected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows.

5. Accept the default options in the remaining tabs, including the default behavior of creating a new storage account on the **Storage** tab and a new Application Insight instance on the **Monitoring** tab. You can also choose to use an existing storage account or Application Insights instance.
6. Select **Review + create** to review the app configuration you chose, and then select **Create** to provision and deploy the function app.
7. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
8. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.



Next, create a function in the new function app.

Create an HTTP trigger function

1. In your function app, select **Overview**, and then select **+ Create** under **Functions**. If you don't see the **+ Create** button, you can instead [create your functions locally](#).
2. Under **Select a template**, scroll down and choose the **HTTP trigger** template.
3. In **Template details**, use `HttpExample` for **New Function**, select **Anonymous** from the **Authorization level** drop-down list, and then select **Create**.

Azure creates the HTTP trigger function. Now, you can run the new function by sending an HTTP request.

Create your functions locally

If you aren't able to create your function code in the portal, you can instead create a local project and publish the function code to your new function app.

1. In your function app, select **Overview**, and then in **Create functions in your preferred environment** under **Functions**.
2. Choose your preferred local development environment and follow the steps in the linked article to create and publish your first Azure Functions project.

💡 Tip

When publishing your new project, make sure to use the function app and related resources you just created.

Test the function

💡 Tip

The **Code + Test** functionality in the portal works even for functions that are read-only and can't be edited in the portal.

1. On the **Overview** page for your new function app, select your new HTTP triggered function in the **Functions** tab.
2. In the left menu, expand **Developer**, select **Code + Test**, and then select **Test/Run**.

3. In the Test/Run dialog, select Run.

An HTTP POST request is sent to your new function with a payload that contains the `name` value of `Azure`. You can also test the function by selecting **GET** for **HTTP method** and adding a `name` parameter with a value of `YOUR_NAME`.

💡 Tip

To test in an external browser, instead select **Get function URL**, copy the **default (Function key)** value, add the query string value `&name=<YOUR_NAME>` to the end of this URL, and then submit the URL in the address bar of your web browser.

4. When your function runs, trace information is written to the logs. To see the trace output, return to the **Code + Test** page in the portal and expand the **Logs** arrow at the bottom of the page. Call your function again to see the trace output written to the logs.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with options like 'Functions', 'App keys', 'App files', and 'Proxies'. The main content area is titled 'Overview' and shows details for a resource group named 'myResourceGroup'. The 'Resource group (change)' section is highlighted with a red box. It lists the following information:

- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags

On the right side, there are links for 'URL' (https://myfunctionapp.azurewebsites.net), 'Operating System' (Windows), 'App Service Plan' (ASP-myResourceGroup-a285 (Y1: 0)), 'Properties' (See More), and 'Runtime version' (3.0.13139.0). At the bottom, there are tabs for 'Metrics', 'Features (8)', 'Notifications (0)', and 'Quickstart'.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

Now that you've created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create a Function App in an App Service plan

Article • 01/13/2023

This Azure Functions sample script creates a function app, which is a container for your functions. The function app that is created uses a dedicated App Service plan, which means your server resources are always on.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square with a white right-pointing arrow.
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run [`az upgrade`](#).

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment, Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

```
Azure CLI

subscription=<subscriptionId> # add subscription here

az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

```
Azure CLI

# Function app and storage account names must be unique.

# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="eastus"
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"
tag="create-function-app-consumption"
storage="msdocsaccount$randomIdentifier"
appServicePlan="msdocs-app-service-plan-$randomIdentifier"
functionApp="msdocs-serverless-function-$randomIdentifier"
skuStorage="Standard_LRS"
skuPlan="B1"
functionsVersion="4"

# Create a resource group
echo "Creating $resourceGroup in \"$location\"..."
az group create --name $resourceGroup --location "$location" --tags $tag

# Create an Azure storage account in the resource group.
echo "Creating $storage"
```

```

az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage

# Create an App Service plan
echo "Creating $appServicePlan"
az functionapp plan create --name $appServicePlan --resource-group $resourceGroup --location "$location" --sku $skuPlan

# Create a Function App
echo "Creating $functionApp"
az functionapp create --name $functionApp --storage-account $storage --plan $appServicePlan --resource-group $resourceGroup --functions-version $functionsVersion

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the `az group delete` command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp plan create	Creates a Premium plan.
az functionapp create	Creates a function app in the App Service plan.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create a function app in a Premium plan - Azure CLI

Article • 01/13/2023

This Azure Functions sample script creates a function app, which is a container for your functions. The function app that is created uses a [scalable Premium plan](#).

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square icon with a white arrow pointing outwards.
[Launch Cloud Shell](#)
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Sample script

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, just select Try it from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to

<https://shell.azure.com>.

When Cloud Shell opens, verify that **Bash** is selected for your environment. Subsequent sessions will use Azure CLI in a Bash environment. Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press **Enter** to run it.

Sign in to Azure

Cloud Shell is automatically authenticated under the initial account signed-in with. Use the following script to sign in using a different subscription, replacing <Subscription ID> with your Azure Subscription ID. If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Azure CLI

```
subscription=<subscriptionId> # add subscription here  
az account set -s $subscription # ...or use 'az login'
```

For more information, see [set active subscription](#) or [log in interactively](#)

Run the script

Azure CLI

```
# Function app and storage account names must be unique.  
  
# Variable block  
let "randomIdentifier=$RANDOM*$RANDOM"  
location="eastus"  
resourceGroup="msdocs-azure-functions-rg-$randomIdentifier"  
tag="create-function-app-premium-plan"  
storage="msdocsaccount$randomIdentifier"  
premiumPlan="msdocs-premium-plan-$randomIdentifier"  
functionApp="msdocs-function-$randomIdentifier"  
skuStorage="Standard_LRS" # Allowed values: Standard_LRS, Standard_GRS,  
Standard_RAGRS, Standard_ZRS, Premium_LRS, Premium_ZRS, Standard_GZRS,  
Standard_RAGZRS  
skuPlan="EP1"  
functionsVersion="4"  
  
# Create a resource group  
echo "Creating $resourceGroup in \"$location\"..."  
az group create --name $resourceGroup --location "$location" --tags $tag  
  
# Create an Azure storage account in the resource group.  
echo "Creating $storage"
```

```

az storage account create --name $storage --location "$location" --resource-group $resourceGroup --sku $skuStorage

# Create a Premium plan
echo "Creating $premiumPlan"
az functionapp plan create --name $premiumPlan --resource-group $resourceGroup --location "$location" --sku $skuPlan

# Create a Function App
echo "Creating $functionApp"
az functionapp create --name $functionApp --storage-account $storage --plan $premiumPlan --resource-group $resourceGroup --functions-version $functionsVersion

```

Clean up resources

Use the following command to remove the resource group and all resources associated with it using the `az group delete` command - unless you have an ongoing need for these resources. Some of these resources may take a while to create, as well as to delete.

Azure CLI

```
az group delete --name $resourceGroup
```

Sample reference

Each command in the table links to command specific documentation. This script uses the following commands:

Command	Notes
az group create	Creates a resource group in which all resources are stored.
az storage account create	Creates an Azure Storage account.
az functionapp plan create	Creates a Premium plan in a specific SKU .
az functionapp create	Creates a function app in the App Service plan.

Next steps

For more information on the Azure CLI, see [Azure CLI documentation](#).

Additional Azure Functions CLI script samples can be found in the [Azure Functions documentation](#).

Create your first containerized Azure Functions

Article • 06/29/2023

In this article, you create a function app running in a Linux container and deploy it to Azure Functions.

Deploying your function code to Azure Functions in a container requires [Premium plan](#) or [Dedicated \(App Service\) plan](#) hosting. Completing this article incurs costs of a few US dollars in your Azure account, which you can minimize by [cleaning-up resources](#) when you're done.

Other options for deploying your function app container to Azure include:

- Azure Container Apps: to learn more, see [Deploy a container to Azure Container apps](#).
- Azure Arc (currently in preview): to learn more, see [Deploy a container to Azure Arc](#).

Choose your development language

First, you use Azure Functions tools to create your project code as a function app in a Docker container using a language-specific Linux base image. Make sure to select your language of choice at the top of the article.

Core Tools automatically generates a Dockerfile for your project that uses the most up-to-date version of the correct base image for your functions language. You should regularly update your container from the latest base image and redeploy from the updated version of your container. For more information, see [Creating containerized function apps](#).

Prerequisites

Before you begin, you must have the following requirements in place:

- Install the [.NET 8.0 SDK](#).
- Install [Azure Functions Core Tools](#) version 4.0.5198, or a later version.
- [Azure CLI](#) version 2.4 or a later version.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

To publish the containerized function app image you create to a container registry, you need a Docker ID and [Docker](#) running on your local computer. If you don't have a Docker ID, you can [create a Docker account](#).

Azure Container Registry

You also need to complete the [Create a container registry](#) section of the Container Registry quickstart to create a registry instance. Make a note of your fully qualified login server name.

Create and test the local functions project

In a terminal or command prompt, run the following command for your chosen language to create a function app project in the current folder:

Console

```
func init --worker-runtime dotnet-isolated --docker
```

The `--docker` option generates a *Dockerfile* for the project, which defines a suitable container for use with Azure Functions and the selected runtime.

Use the following command to add a function to your project, where the `--name` argument is the unique name of your function and the `--template` argument specifies the function's trigger. `func new` creates a C# code file in your project.

Console

```
func new --name HttpExample --template "HTTP trigger"
```

To test the function locally, start the local Azure Functions runtime host in the root of the project folder.

Console

```
func start
```

After you see the `HttpExample` endpoint written to the output, navigate to that endpoint. You should see a welcome message in the response output.

Press **Ctrl+C** (**Command+C** on macOS) to stop the host.

Build the container image and verify locally

(Optional) Examine the *Dockerfile* in the root of the project folder. The *Dockerfile* describes the required environment to run the function app on Linux. The complete list of supported base images for Azure Functions can be found in the [Azure Functions base image page](#).

In the root project folder, run the [docker build](#) command, provide a name as `azurefunctionsimage`, and tag as `v1.0.0`. Replace `<DOCKER_ID>` with your Docker Hub account ID. This command builds the Docker image for the container.

Console

```
docker build --tag <DOCKER_ID>/azurefunctionsimage:v1.0.0 .
```

When the command completes, you can run the new container locally.

To verify the build, run the image in a local container using the [docker run](#) command, replace `<DOCKER_ID>` again with your Docker Hub account ID, and add the ports argument as `-p 8080:80`:

Console

```
docker run -p 8080:80 -it <DOCKER_ID>/azurefunctionsimage:v1.0.0
```

After the image starts in the local container, browse to `http://localhost:8080/api/HttpExample`, which must display the same greeting message as before. Because the HTTP triggered function you created uses anonymous authorization, you can call the function running in the container without having to obtain an access key. For more information, see [authorization keys](#).

After verifying the function app in the container, press **Ctrl+C** (**Command+C** on macOS) to stop execution.

Publish the container image to a registry

To make your container image available for deployment to a hosting environment, you must push it to a container registry.

Azure Container Registry

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts. You should use a private registry service for publishing your containers to Azure services.

1. Use the following command to sign in to your registry instance:

Azure CLI

```
az acr login --name <REGISTRY_NAME>
```

In the previous command, replace `<REGISTRY_NAME>` with the name of your Container Registry instance.

2. Use the following command to tag your image with the fully qualified name of your registry login server:

docker

```
docker tag <DOCKER_ID>/azurefunctionsimage:v1.0.0  
<LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

Replace `<LOGIN_SERVER>` with the fully qualified name of your registry login server and `<DOCKER_ID>` with your Docker ID.

3. Use the following command to push the container to your registry instance:

docker

```
docker push <LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

4. Use the following command to enable the built-in admin account so that Functions can connect to the registry with a username and password:

Azure CLI

```
az acr update -n <REGISTRY_NAME> --admin-enabled true
```

1. Use the following command to retrieve the admin username and password, which Functions needs to connect to the registry:

```
Azure CLI
```

```
az acr credential show -n <REGISTRY_NAME> --query "[username, passwords[0].value]" -o tsv
```

 **Important**

The admin account username and password are important credentials. Make sure to store them securely and never in an accessible location like a public repository.

Create supporting Azure resources for your function

Before you can deploy your container to Azure, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the following commands to create these items. Both Azure CLI and PowerShell are supported. To create your Azure resources using Azure PowerShell, you also need the [Az PowerShell module](#), version 5.9.0 or later.

1. If you haven't done already, sign in to Azure.

```
Azure CLI
```

```
Azure CLI
```

```
az login
```

The [az login](#) command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsContainers-rg` in your chosen region.

Azure CLI

Azure CLI

```
az group create --name AzureFunctionsContainers-rg --location  
<REGION>
```

The `az group create` command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the [az account list-locations](#) command.

3. Create a general-purpose storage account in your resource group and region.

Azure CLI

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location <REGION>  
--resource-group AzureFunctionsContainers-rg --sku Standard_LRS
```

The `az storage account create` command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Storage names must contain 3 to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account [supported by Functions](#).

4. Use the command to create a Premium plan for Azure Functions named `myPremiumPlan` in the **Elastic Premium 1** pricing tier (`--sku EP1`), in your `<REGION>`, and in a Linux container (`--is-linux`).

Azure CLI

Azure CLI

```
az functionapp plan create --resource-group  
AzureFunctionsContainers-rg --name myPremiumPlan --location  
<REGION> --number-of-workers 1 --sku EP1 --is-linux
```

We use the Premium plan here, which can scale as needed. For more information about hosting, see [Azure Functions hosting plans comparison](#). For more information on how to calculate costs, see the [Functions pricing page](#).

The command also creates an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

Create and configure a function app on Azure with the image

A function app on Azure manages the execution of your functions in your Azure Functions hosting plan. In this section, you use the Azure resources from the previous section to create a function app from an image in a container registry and configure it with a connection string to Azure Storage.

1. Create a function app using the following command, depending on your container registry:

Azure Container Registry

Azure CLI

```
az functionapp create --name <APP_NAME> --storage-account  
<STORAGE_NAME> --resource-group AzureFunctionsContainers-rg --plan  
myPremiumPlan --image <LOGIN_SERVER>/azurefunctionsimage:v1.0.0 --  
registry-username <USERNAME> --registry-password <SECURE_PASSWORD>
```

In this example, replace `<STORAGE_NAME>` with the name you used in the previous section for the storage account. Also, replace `<APP_NAME>` with a globally unique name appropriate to you and `<DOCKER_ID>` or `<LOGIN_SERVER>` with your Docker Hub account ID or Container Registry server, respectively. When you're deploying from a custom container registry, the image name indicates the URL of the registry.

When you first create the function app, it pulls the initial image from your Docker Hub. You can also [Enable continuous deployment](#) to Azure from your container registry.

 **Tip**

You can use the [**DisableColor setting**](#) in the `host.json` file to prevent ANSI control characters from being written to the container logs.

2. Use the following command to get the connection string for the storage account you created:

Azure CLI

Azure CLI

```
az storage account show-connection-string --resource-group
AzureFunctionsContainers-rg --name <STORAGE_NAME> --query
connectionString --output tsv
```

The connection string for the storage account is returned by using the [az storage account show-connection-string](#) command.

Replace `<STORAGE_NAME>` with the name of the storage account you created earlier.

3. Use the following command to add the setting to the function app:

Azure CLI

Azure CLI

```
az functionapp config appsettings set --name <APP_NAME> --resource-
group AzureFunctionsContainers-rg --settings AzureWebJobsStorage=
<CONNECTION_STRING>
```

The [az functionapp config appsettings set](#) command creates the setting.

In this command, replace `<APP_NAME>` with the name of your function app and `<CONNECTION_STRING>` with the connection string from the previous step. The connection should be a long encoded string that begins with `DefaultEndpointProtocol=`.

4. The function can now use this connection string to access the storage account.

Verify your functions on Azure

With the image deployed to your function app in Azure, you can now invoke the function through HTTP requests.

1. Run the following `az functionapp function show` command to get the URL of your new function:

Azure CLI

```
az functionapp function show --resource-group AzureFunctionsContainers-rg --name <APP_NAME> --function-name HttpExample --query invokeUrlTemplate
```

Replace `<APP_NAME>` with the name of your function app.

2. Use the URL you just obtained to call the `HttpExample` function endpoint.

When you navigate to this URL, the browser must display similar output as when you ran the function locally.

Clean up resources

If you want to continue working with Azure Function using the resources you created in this article, you can leave all those resources in place. Because you created a Premium Plan for Azure Functions, you'll incur one or two USD per day in ongoing costs.

To avoid ongoing costs, delete the `AzureFunctionsContainers-rg` resource group to clean up all the resources in that group:

Azure CLI

```
az group delete --name AzureFunctionsContainers-rg
```

Next steps

[Working with custom containers and Azure Functions](#)

Create a function triggered by Azure Cosmos DB

Article • 12/31/2023

Learn how to create a function in the Azure portal that is triggered when data is added to or changed in Azure Cosmos DB. To learn more about Azure Cosmos DB, see [Azure Cosmos DB: Serverless database computing using Azure Functions](#).

ⓘ Note

In-portal editing is only supported for JavaScript, PowerShell, and C# Script functions. Python in-portal editing is supported only when running in the Consumption plan. When possible, you should [develop your functions locally](#).

To learn more about the limitations on editing function code in the Azure portal, see [Development limitations in the Azure portal](#).

Prerequisites

To complete this tutorial:

- If you don't have an Azure subscription, create a [free account](#) before you begin.

ⓘ Note

Azure Cosmos DB bindings are only supported for use with Azure Cosmos DB for NoSQL. Support for Azure Cosmos DB for Table is provided by using the [Table storage bindings](#), starting with extension 5.x. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API, including [Azure Cosmos DB for MongoDB](#), [Azure Cosmos DB for Cassandra](#), and [Azure Cosmos DB for Apache Gremlin](#).

Sign in to Azure

Sign in to the [Azure portal](#) with your Azure account.

Create an Azure Cosmos DB account

You must have an Azure Cosmos DB account that uses the SQL API before you create the trigger.

1. From the Azure portal menu or the [Home page](#), select **Create a resource**.
2. Search for **Azure Cosmos DB**. Select **Create > Azure Cosmos DB**.
3. On the **Create an Azure Cosmos DB account** page, select the **Create** option within the **Azure Cosmos DB for NoSQL** section.

Azure Cosmos DB provides several APIs:

- NoSQL, for document data
- PostgreSQL
- MongoDB, for document data
- Apache Cassandra
- Table
- Apache Gremlin, for graph data

To learn more about the API for NoSQL, see [Welcome to Azure Cosmos DB](#).

4. In the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos DB account.

[] [Expand table](#)

Setting	Value	Description
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Resource group name	Select a resource group, or select Create new , then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos DB account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name. The name can contain only lowercase letters, numbers, and the hyphen (-) character. It must be 3-44 characters.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select Provisioned throughput to create an account in provisioned throughput mode. Select Serverless to create an account in serverless mode.

Setting	Value	Description
Apply Azure Cosmos DB free tier discount	Apply or Do not apply	With Azure Cosmos DB free tier, you get the first 1000 RU/s and 25 GB of storage for free in an account. Learn more about free tier .
Limit total account throughput	Selected or not	Limit the total amount of throughput that can be provisioned on this account. This limit prevents unexpected charges related to provisioned throughput. You can update or remove this limit after your account is created.

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt in when creating the account. If you don't see the option to apply the free tier discount, another account in the subscription has already been enabled with free tier.

Home > Marketplace > Create an Azure Cosmos DB account >

Create Azure Cosmos DB Account - Azure Cosmos DB for NoSQL

Basics Global Distribution Networking Backup Policy Encryption Tags Review + create

Azure Cosmos DB is a fully managed NoSQL and relational database service for building scalable, high performance applications. [Try it for free](#), for 30 days with unlimited renewals. Go to production starting at \$24/month per database, multiple containers included. [Learn more](#)

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Contoso Subscription

Resource Group * myResourceGroup [Create new](#)

Instance Details

Account Name * mysqlaccount

Location * (US) East US 2

Capacity mode ⓘ Provisioned throughput Serverless [Learn more about capacity mode](#)

With Azure Cosmos DB free tier, you will get the first 1000 RU/s and 25 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$64/month discount per account.

Apply Free Tier Discount Apply Do Not Apply

Limit total account throughput Limit the total amount of throughput that can be provisioned on this account [Learn more](#)

Note

The following options are not available if you select **Serverless** as the **Capacity mode**:

! Note

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Limit total account throughput

5. In the **Global Distribution** tab, configure the following details. You can leave the default values for this quickstart:

[\[+\] Expand table](#)

Setting	Value	Description
Geo-Redundancy	Disable	Enable or disable global distribution on your account by pairing your region with a pair region. You can add more regions to your account later.
Multi-region Writes	Disable	Multi-region writes capability allows you to take advantage of the provisioned throughput for your databases and containers across the globe.
Availability Zones	Disable	Availability Zones help you further improve availability and resiliency of your application.

(!) Note

The following options are not available if you select **Serverless** as the **Capacity mode** in the previous **Basics** page:

- Geo-redundancy
- Multi-region Writes

6. Optionally, you can configure more details in the following tabs:

- **Networking.** Configure [access from a virtual network](#).
- **Backup Policy.** Configure either [periodic](#) or [continuous](#) backup policy.
- **Encryption.** Use either service-managed key or a [customer-managed key](#).
- **Tags.** Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups.

7. Select **Review + create**.

8. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

The screenshot shows the Azure portal's deployment overview page. At the top, it displays the deployment name: Microsoft.Azure.CosmosDB-20230302095414 | Overview. Below the title, there is a green checkmark icon followed by the message "Your deployment is complete". A detailed deployment summary follows, including the deployment name, subscription, resource group, start time, and correlation ID. On the left, a navigation menu lists "Overview", "Inputs", "Outputs", and "Template". On the right, there are buttons for "Delete", "Cancel", "Redeploy", "Download", and "Refresh". Below the summary, there are sections for "Deployment details" and "Next steps", with a prominent blue "Go to resource" button. At the bottom, there are links to "Give feedback" and "Tell us about your experience with deployment", along with a magnifying glass icon.

9. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the Azure portal's quick start page for a MySQL account named mysqlaccount. The title is "mysqlaccount | Quick start". The main content area says "Congratulations! Your Azure Cosmos DB account was created." It then asks to "Choose a platform" and provides options for ".NET", "Java", "Node.js", and "Python", with ".NET" selected. Two numbered steps are shown: Step 1: "Add a container" which explains that data is stored in containers and includes a "Create 'Items' container" button; Step 2: "Download and run your .NET app" which includes a "Download" button. On the left, a sidebar lists various management options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Cost Management, Quick start (selected), Notifications, Data Explorer, Settings, and Features.

Create a function app in Azure

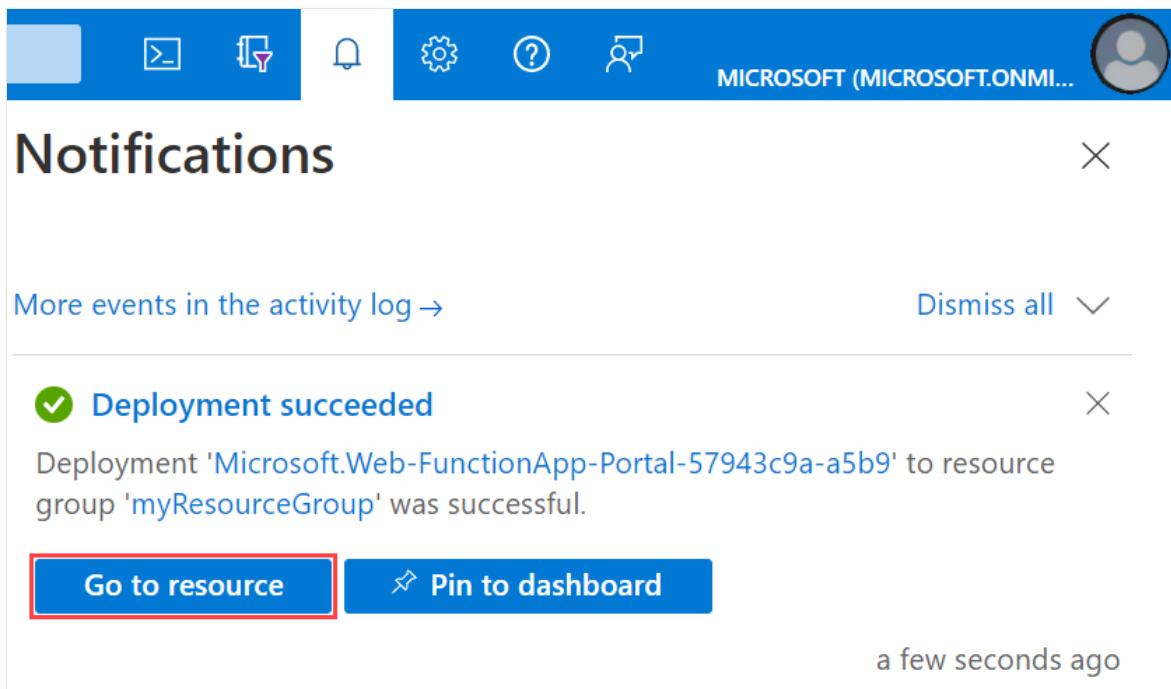
1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. On the **Basics** page, use the function app settings as specified in the following table:

Expand table

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which you create your new function app.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which you create your function app. You should create a new resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Do you want to deploy code or container image?	Code	Option to publish code files or a Docker container .
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, Python, TypeScript, and C# script. C# class library and Java functions must be developed locally .
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Select a region that's near you or near other services that your functions can access.
Operating system	Windows	An operating system is preselected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows. Container publishing is only supported on Linux.
Hosting options and plans	Consumption (Serverless)	Hosting plan that defines how resources are allocated to your function app. In the default Consumption plan, resources are added dynamically as required by your functions. In this serverless hosting, you pay only for the time your functions run. Premium plan also offers dynamic scaling. When you run in an App Service plan, you must manage the scaling of your function app .

- Accept the default options of creating a new storage account on the **Storage** tab and a new Application Insight instance on the **Monitoring** tab. You can also choose to use an existing storage account or Application Insights instance.

5. Select **Review + create** to review the app configuration you chose, and then select **Create** to provision and deploy the function app.
6. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
7. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.



Next, you create a function in the new function app.

Create Azure Cosmos DB trigger

1. In your function app, select **Overview**, and then select **+ Create** under **Functions**.
2. Under **Select a template**, scroll down and choose the **Azure Cosmos DB trigger** template.
3. In **Template details**, configure the new trigger with the settings as specified in this table, then select **Create**:

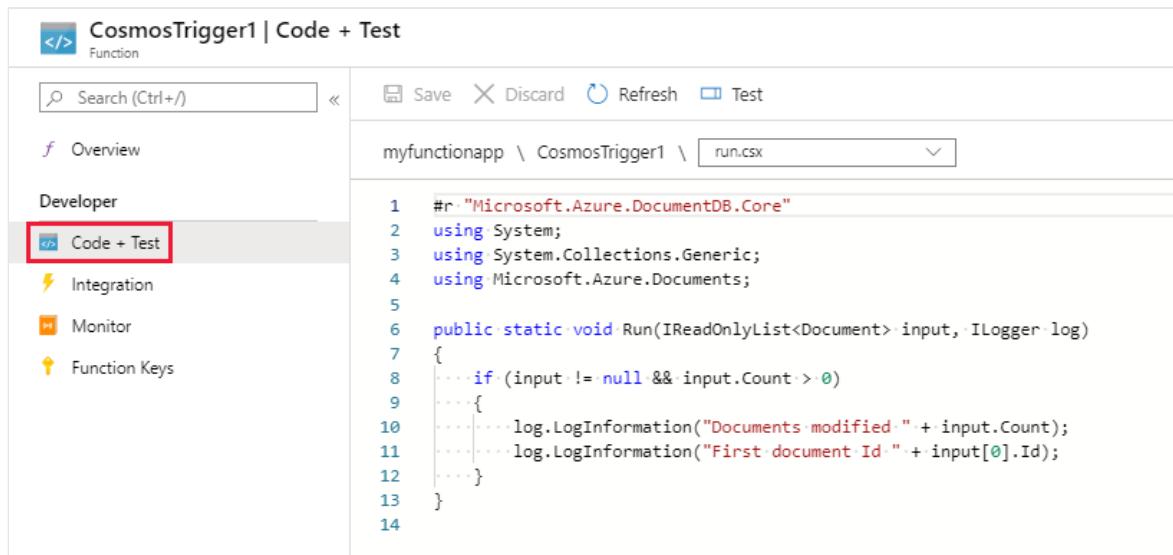
[] Expand table

Setting	Suggested value	Description
New function	Accept the default name	The name of the function.

Setting	Suggested value	Description
Azure Cosmos DB account connection	Accept the default new name	Select New , the Database Account you created earlier, and then OK . This action creates an application setting for your account connection. This setting is used by the binding to connection to the database.
Database name	Tasks	Name of the database that includes the collection to be monitored.
Collection name	Items	Name of the collection to be monitored.
Collection name for leases	leases	Name of the collection to store the leases.
Create lease collection if it does not exist	Yes	Checks for existence of the lease collection and automatically creates it.

Azure creates the Azure Cosmos DB triggered function based on the provided values.

4. To display the template-based function code, select **Code + Test**.



The screenshot shows the Azure Functions portal interface. The left sidebar has tabs for Overview, Developer (with sub-options like Code + Test, Integration, Monitor, and Function Keys), and a search bar. The main area shows the function details for 'CosmosTrigger1'. The 'Code + Test' tab is highlighted with a red box. The code editor displays the following C# code:

```

1  #r "Microsoft.Azure.DocumentDB.Core"
2  using System;
3  using System.Collections.Generic;
4  using Microsoft.Azure.Documents;
5
6  public static void Run(IReadOnlyList<Document> input, ILogger log)
7  {
8      if (input != null && input.Count > 0)
9      {
10          log.LogInformation("Documents modified: " + input.Count);
11          log.LogInformation("First document Id: " + input[0].Id);
12      }
13 }
14

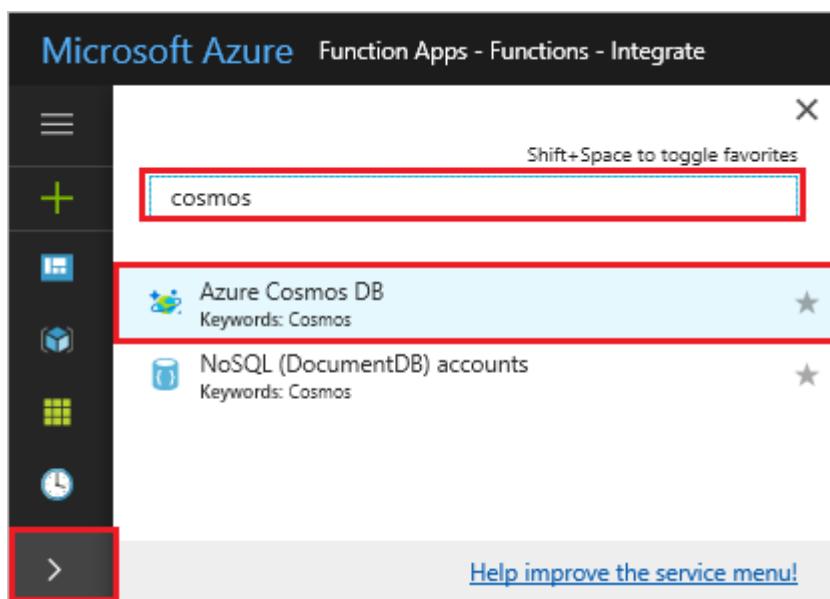
```

This function template writes the number of documents and the first document ID to the logs.

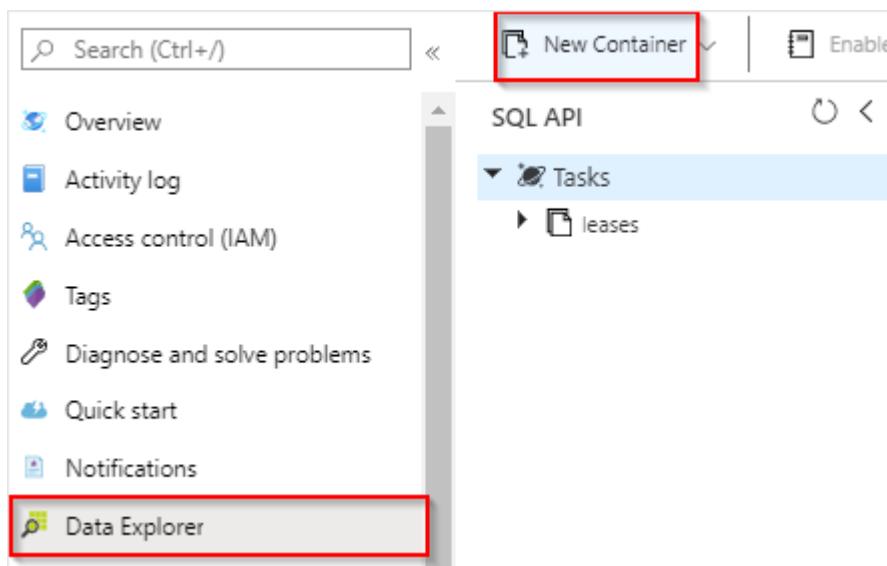
Next, you connect to your Azure Cosmos DB account and create the `Items` container in the `Tasks` database.

Create the Items container

1. Open a second instance of the [Azure portal](#) in a new tab in the browser.
2. On the left side of the portal, expand the icon bar, type `cosmos` in the search field, and select Azure Cosmos DB.



3. Choose your Azure Cosmos DB account, then select the **Data Explorer**.
4. Under **SQL API**, choose **Tasks** database and select **New Container**.



5. In **Add Container**, use the settings shown in the table below the image.

Add Container

* Database id ⓘ
 Create new Use existing
 Tasks

* Container id ⓘ
 Items

* Partition key ⓘ
 /category
 My partition key is larger than 100 bytes

* Throughput (400 - 100,000 RU/s) ⓘ
 Autopilot (preview) Manual
 400

Unique keys ⓘ
 + Add unique key

[+] Expand table

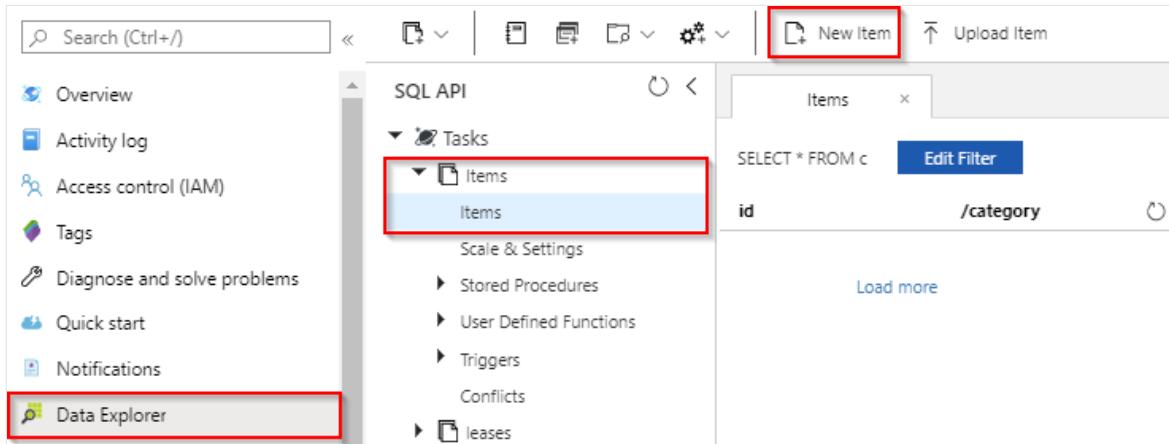
Setting	Suggested value	Description
Database ID	Tasks	The name for your new database. This must match the name defined in your function binding.
Container ID	Items	The name for the new container. This must match the name defined in your function binding.
Partition key	/category	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant container.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.

- Click OK to create the Items container. It may take a short time for the container to get created.

After the container specified in the function binding exists, you can test the function by adding items to this new container.

Test the function

1. Expand the new **Items** container in Data Explorer, choose **Items**, then select **New Item**.

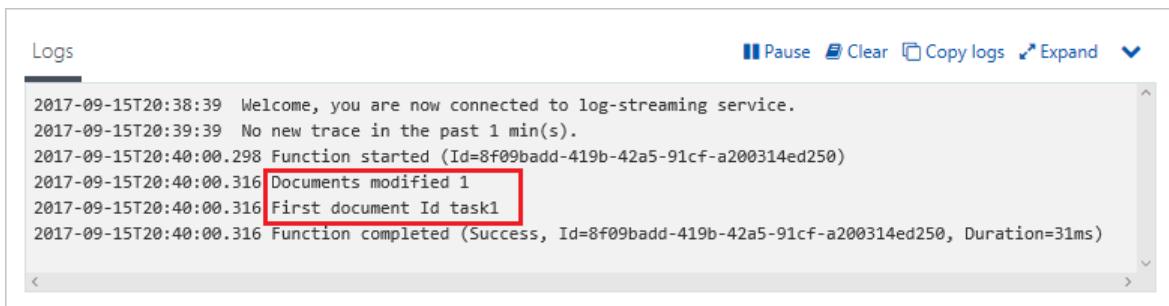


2. Replace the contents of the new item with the following content, then choose **Save**.

```
YAML

{
    "id": "task1",
    "category": "general",
    "description": "some task"
}
```

3. Switch to the first browser tab that contains your function in the portal. Expand the function logs and verify that the new document has triggered the function. See that the `task1` document ID value is written to the logs.



4. (Optional) Go back to your document, make a change, and click **Update**. Then, go back to the function logs and verify that the update has also triggered the function.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.

The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has a "Functions" section with "myfunctionapp" selected. The main content area is the "Overview" tab, which displays the following details:

Setting	Value
URL	https://myfunctionapp.azurewebsites.net
Operating System	Windows
App Service Plan	ASP-myResourceGroup-a285 (Y1: 0)
Properties	See More
Runtime version	3.0.13139.0

On the left, there is a sidebar with various links: Overview (highlighted), Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with Functions, App keys, App files, Proxies listed), Metrics, Features (8), Notifications (0), and Quickstart.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You have created a function that runs when a document is added or modified in your Azure Cosmos DB. For more information about Azure Cosmos DB triggers, see [Azure Cosmos DB bindings for Azure Functions](#).

Now that you've created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Create a function in Azure that's triggered by Blob storage

Article • 09/19/2024

Learn how to create a function triggered when files are uploaded to or updated in a Blob storage container.

ⓘ Note

In-portal editing is only supported for JavaScript, PowerShell, and C# Script functions. Python in-portal editing is supported only when running in the Consumption plan. To create a C# Script app that supports in-portal editing, you must choose a runtime **Version** that supports the **in-process model**.

When possible, you should [develop your functions locally](#).

To learn more about the limitations on editing function code in the Azure portal, see [Development limitations in the Azure portal](#).

Prerequisites

- An Azure subscription. If you don't have one, create a [free account](#) before you begin.

Create an Azure Function app

1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. Under **Select a hosting option**, select **Consumption > Select** to create your app in the default **Consumption** plan. In this [serverless](#) hosting option, you pay only for the time your functions run. [Premium plan](#) also offers dynamic scaling. When you run in an App Service plan, you must manage the [scaling of your function app](#).
4. On the **Basics** page, use the function app settings as specified in the following table:

 Expand table

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which you create your new function app.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which you create your function app. You should create a new resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z (case insensitive), 0-9, and -.
Runtime stack	Preferred language	<p>Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, Python, TypeScript, and C# script.</p> <p>To create a C# Script app that supports in-portal editing, you must choose a runtime Version that supports the in-process model.</p> <p>C# class library and Java functions must be developed locally.</p>
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Select a region that's near you or near other services that your functions can access.
Operating system	Windows	An operating system is preselected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows.

5. Accept the default options in the remaining tabs, including the default behavior of creating a new storage account on the **Storage** tab and a new Application Insight instance on the **Monitoring** tab. You can also choose to use an existing storage account or Application Insights instance.
6. Select **Review + create** to review the app configuration you chose, and then select **Create** to provision and deploy the function app.
7. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
8. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

The screenshot shows the Azure Notifications page. At the top, there are several icons: a blue square, a white square with a checkmark, a purple square with a gear, a bell, a gear, a question mark, and a user profile. To the right of these is the text "MICROSOFT (MICROSOFT.ONMI...)" and a user icon. Below the header, the word "Notifications" is displayed in large bold letters, with a close button "X" to its right. A message bar at the top says "More events in the activity log →" and "Dismiss all" with a dropdown arrow. The main content area contains a single notification card. The card has a green checkmark icon and the text "Deployment succeeded". Below this, it says "Deployment 'Microsoft.Web-FunctionApp-Portal-57943c9a-a5b9' to resource group 'myResourceGroup' was successful." At the bottom of the card are two buttons: "Go to resource" (which is highlighted with a red border) and "Pin to dashboard". The timestamp "a few seconds ago" is at the bottom right of the card.

You've successfully created your new function app. Next, you create a function in the new function app.

Create an Azure Blob storage triggered function

1. In your function app, select **Overview**, and then select **+ Create** under **Functions**.
2. Under **Select a template**, choose the **Blob trigger** template and select **Next**.
3. In **Template details**, configure the new trigger with the settings as specified in this table, then select **Create**:

[] Expand table

Setting	Suggested value	Description
Job type	Append to app	You only see this setting for a Python v2 app.
New Function	Unique in your function app	Name of this blob triggered function.
Path	samples-workitems/{name}	Location in Blob storage being monitored. The file name of the blob is passed in the binding as the <i>name</i> parameter.
Storage account connection	AzureWebJobsStorage	You can use the storage account connection already being used by your function app, or create a new one.

Azure creates the Blob Storage triggered function based on the provided values. Next, create the **samples-workitems** container.

Create the container

1. Return to the **Overview** page for your function app, select your **Resource group**, then find and select the storage account in your resource group.
2. In the storage account page, select **Data storage > Containers > + Container**.
3. In the **Name** field, type `samples-workitems`, and then select **Create** to create a container.
4. Select the new `samples-workitems` container, which you use to test the function by uploading a file to the container.

Test the function

1. In a new browser window, return to your function app page and select **Log stream**, which displays real-time logging for your app.
2. From the `samples-workitems` container page, select **Upload > Browse for files**, browse to a file on your local computer (such as an image file), and choose the file.
3. Select **Open** and then **Upload**.
4. Go back to your function app logs and verify that the blob has been read.

ⓘ Note

When your function app runs in the default Consumption plan, there may be a delay of up to several minutes between the blob being added or updated and the function being triggered. If you need low latency in your blob triggered functions, consider one of these [other blob trigger options](#).

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

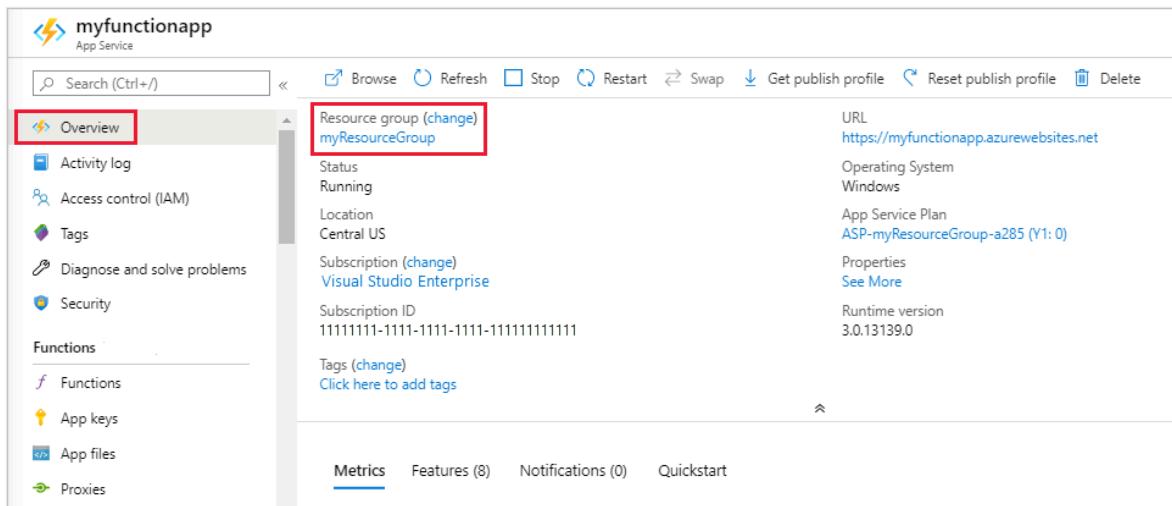
Resources in Azure refer to function apps, functions, storage accounts, and so forth.

They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has a "Functions" section with options like "Functions", "App keys", "App files", and "Proxies". The main content area is titled "Overview" and shows details for the resource group "myResourceGroup". The "Resource group" link is highlighted with a red box. The details shown include:

Setting	Value
Status	Running
Location	Central US
Subscription	Visual Studio Enterprise
Subscription ID	1111111-1111-1111-1111-111111111111
Tags	Click here to add tags
URL	https://myfunctionapp.azurewebsites.net
Operating System	Windows
App Service Plan	ASP-myResourceGroup-a285 (Y1: 0)
Properties	See More
Runtime version	3.0.13139.0

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You have created a function that runs when a blob is added to or updated in Blob storage. For more information about Blob storage triggers, see [Azure Functions Blob storage bindings](#).

Now that you've created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create a function triggered by Azure Queue storage

Article • 09/18/2024

Learn how to create a function that is triggered when messages are submitted to an Azure Storage queue.

ⓘ Note

In-portal editing is only supported for JavaScript, PowerShell, and C# Script functions. Python in-portal editing is supported only when running in the Consumption plan. To create a C# Script app that supports in-portal editing, you must choose a runtime **Version** that supports the **in-process model**.

When possible, you should [develop your functions locally](#).

To learn more about the limitations on editing function code in the Azure portal, see [Development limitations in the Azure portal](#).

Prerequisites

- An Azure subscription. If you don't have one, create a [free account](#) before you begin.

Create an Azure Function app

1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. Under **Select a hosting option**, select **Consumption > Select** to create your app in the default **Consumption** plan. In this [serverless](#) hosting option, you pay only for the time your functions run. [Premium plan](#) also offers dynamic scaling. When you run in an App Service plan, you must manage the [scaling of your function app](#).
4. On the **Basics** page, use the function app settings as specified in the following table:

[+] [Expand table](#)

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which you create your new function app.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which you create your function app. You should create a new resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z (case insensitive), 0-9, and -.
Runtime stack	Preferred language	<p>Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, Python, TypeScript, and C# script.</p> <p>To create a C# Script app that supports in-portal editing, you must choose a runtime Version that supports the in-process model.</p> <p>C# class library and Java functions must be developed locally.</p>
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Select a region that's near you or near other services that your functions can access.
Operating system	Windows	An operating system is preselected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows.

5. Accept the default options in the remaining tabs, including the default behavior of creating a new storage account on the **Storage** tab and a new Application Insight instance on the **Monitoring** tab. You can also choose to use an existing storage account or Application Insights instance.
6. Select **Review + create** to review the app configuration you chose, and then select **Create** to provision and deploy the function app.
7. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
8. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

The screenshot shows the Azure Notifications page. At the top, there are several icons: a blue square, a white square with a blue border, a white square with a blue 'X', a blue square with a white 'T', a blue bell, a blue gear, a blue question mark, and a blue user icon. To the right of these is the text "MICROSOFT (MICROSOFT.ONMI...)" and a user profile picture. Below the header, the title "Notifications" is displayed in large dark font, with a close button "X" to its right. A message "More events in the activity log →" is shown with a "Dismiss all" button to its right. The main content area contains a single notification card. The card has a green checkmark icon and the text "Deployment succeeded". Below this, it says "Deployment 'Microsoft.Web-FunctionApp-Portal-57943c9a-a5b9' to resource group 'myResourceGroup' was successful." At the bottom of the card are two buttons: "Go to resource" (highlighted with a red border) and "Pin to dashboard". The timestamp "a few seconds ago" is at the bottom right of the card.

Next, you create a function in the new function app.

Create a Queue triggered function

1. In your function app, select **Overview**, and then select **+ Create** under **Functions**.
2. Under **Select a template**, scroll down and choose the **Azure Queue Storage trigger** template.
3. In **Template details**, configure the new trigger with the settings as specified in this table, then select **Create**:

[+] Expand table

Setting	Suggested value	Description
Job type	Append to app	You only see this setting for a Python v2 app.
Name	Unique in your function app	Name of this queue triggered function.
Queue name	myqueue-items	Name of the queue to connect to in your Storage account.
Storage account connection	AzureWebJobsStorage	You can use the storage account connection already being used by your function app, or create a new one.

Azure creates the Queue Storage triggered function based on the provided values. Next, you connect to your Azure storage account and create the **myqueue-items**

storage queue.

Create the queue

1. Return to the **Overview** page for your function app, select your **Resource group**, then find and select the storage account in your resource group.
2. In the storage account page, select **Data storage > Queues > + Queue**.
3. In the **Name** field, type `myqueue-items`, and then select **Create**.
4. Select the new `myqueue-items` queue, which you use to test the function by adding a message to the queue.

Test the function

1. In a new browser window, return to your function app page and select **Log stream**, which displays real-time logging for your app.
2. In the `myqueue-items` queue, select **Add message**, type "Hello World!" in **Message text**, and select **OK**.
3. Go back to your function app logs and verify that the function ran to process the message from the queue.
4. Back in your storage queue, select **Refresh** and verify that the message has been processed and is no longer in the queue.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.

The screenshot shows the Azure Function App Overview page for 'myfunctionapp'. The left sidebar has a red box around the 'Overview' tab. The main content area shows the 'Resource group (change)' section with a red box around it, containing the value 'myResourceGroup'. Other details shown include Status: Running, Location: Central US, Subscription: Visual Studio Enterprise, Subscription ID: 11111111-1111-1111-1111-111111111111, and Tags: Click here to add tags. The URL listed is https://myfunctionapp.azurewebsites.net.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You have created a function that runs when a message is added to a storage queue. For more information about Queue storage triggers, see [Azure Functions Storage queue bindings](#).

Now that you have created your first function, let's add an output binding to the function that writes a message back to another queue.

[Add messages to an Azure Storage queue using Functions](#)

Feedback

Was this page helpful?

Yes

No

Create a function in the Azure portal that runs on a schedule

Article • 12/31/2023

Learn how to use the Azure portal to create a function that runs [serverless](#) on Azure based on a schedule that you define.

ⓘ Note

In-portal editing is only supported for JavaScript, PowerShell, and C# Script functions. Python in-portal editing is supported only when running in the Consumption plan. When possible, you should [develop your functions locally](#).

To learn more about the limitations on editing function code in the Azure portal, see [Development limitations in the Azure portal](#).

Prerequisites

To complete this tutorial:

Ensure that you have an Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.

Create a function app

1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. On the **Basics** page, use the function app settings as specified in the following table:

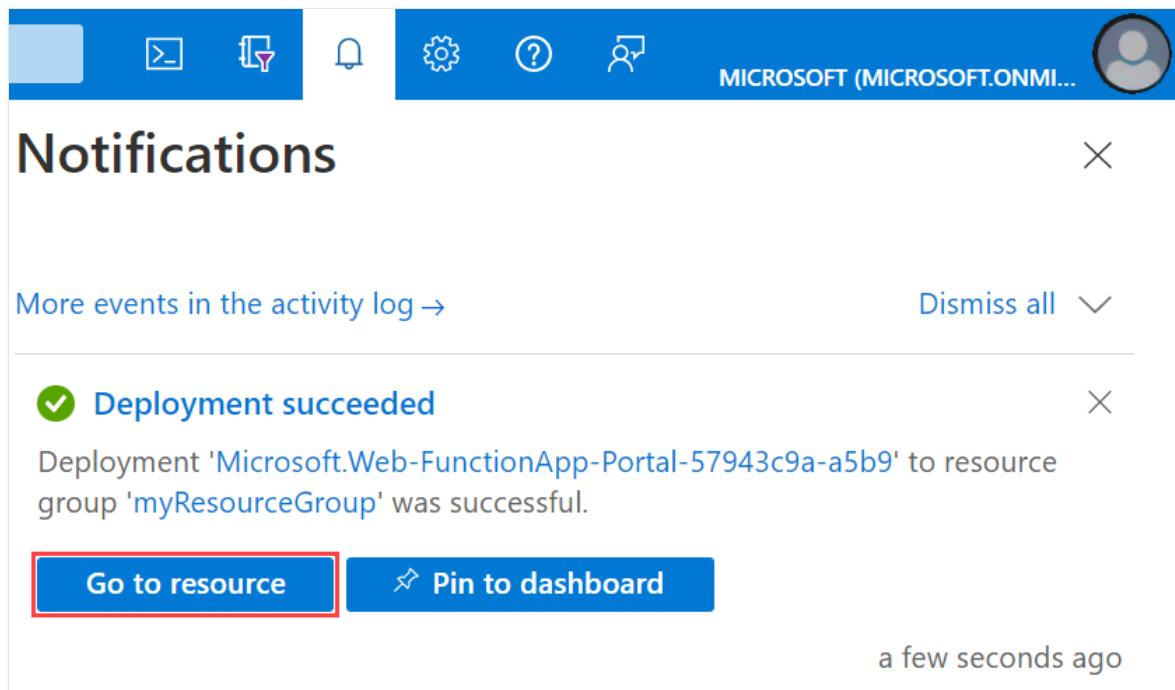
[\[+\] Expand table](#)

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which you create your new function app.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which you create your function app. You should create a new

Setting	Suggested value	Description
		resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Do you want to deploy code or container image?	Code	Option to publish code files or a Docker container .
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, Python, TypeScript, and C# script. C# class library and Java functions must be developed locally .
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Select a region that's near you or near other services that your functions can access.
Operating system	Windows	An operating system is preselected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows. Container publishing is only supported on Linux.
Hosting options and plans	Consumption (Serverless)	Hosting plan that defines how resources are allocated to your function app. In the default Consumption plan, resources are added dynamically as required by your functions. In this serverless hosting, you pay only for the time your functions run. Premium plan also offers dynamic scaling. When you run in an App Service plan, you must manage the scaling of your function app .

4. Accept the default options of creating a new storage account on the **Storage** tab and a new Application Insight instance on the **Monitoring** tab. You can also choose to use an existing storage account or Application Insights instance.
5. Select **Review + create** to review the app configuration you chose, and then select **Create** to provision and deploy the function app.
6. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.

7. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.



Your new function app is ready to use. Next, you'll create a function in the new function app.

A screenshot of the Azure Function App Overview page for "myFunctionApp". The left sidebar has sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Functions, App keys, App files, Proxies, Deployment, Deployment slots, Deployment Center, and Settings. The main area shows the "Essentials" section with details: Resource group (move) : myResourceGroup, Status : Running, Location (move) : East US, Subscription (move) : Azure, Subscription ID : , Tags (edit) : . It includes links for View Cost and JSON View. Below this is a "Functions" tab with sub-links for Metrics, Properties, and Notifications (0). A central panel says "Create functions in your preferred environment" and shows three options: "Create in Azure portal" (with a file icon), "VS Code Desktop" (with a code editor icon), and "Other editors or CLI" (with a terminal icon). Buttons for "Create function", "Create with VS Code Desktop", and "Set up your editor" are present.

Create a timer triggered function

1. In your function app, select **Overview**, and then select **+ Create** under **Functions**.

myFunctionApp

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Microsoft Defender for Cloud

Events (preview)

Functions

App keys

App files

Proxies

Deployment

Deployment slots

Deployment Center

Environment variables

Essentials

Resource group (move) : myResourceGroup

Status : Running

Location (move) : East US

Subscription (move) : Azure

Subscription ID :

Tags (edit) :

URL : <https://myfunctionapp.azurewebsites.net>

Operating System : Windows

App Service Plan : ASP-myResourceGroup-8441 (Y1-0)

Runtime version : 4.28.3.21820

View Cost | JSON View

Create functions in your preferred environment

Create in Azure portal

VS Code Desktop

Other editors or CLI

Create function

Create with VS Code Desktop

Set up your editor

2. Under **Select a template**, scroll down and choose the **Timer trigger** template.

Create function

Select development environment

Instructions will vary based on your development environment. [Learn more](#)

Development environ... [Develop in portal](#)

Select a template

Use a template to create a function. Triggers describe the type of events that invoke your functions. [Learn more](#)

Filter

Template	Description
HTTP trigger	A function that will be run whenever it receives an HTTP request, responding based on data in the body or query string
Timer trigger	A function that will be run on a specified schedule
Azure Queue Storage trigger	A function that will be run whenever a message is added to a specified Azure Storage queue

3. In **Template details**, configure the new trigger with the settings as specified in the table below the image, and then select **Create**.

Template details

We need more information to create the Timer trigger function. [Learn more](#)

New Function*

Schedule *①

Create Cancel

 Expand table

Setting	Suggested value	Description
Name	Default	Defines the name of your timer triggered function.
Schedule	0 */1 * * *	A six field CRON expression that schedules your function to run every minute.

Test the function

1. In your function, select **Code + Test** and expand the **Logs**.

TimerTrigger1 | Code + Test

Function

Search (Ctrl+ /) Save Discard Refresh Test/Run Upload

Overview myFunctionApp \ TimerTrigger1 \ run.csx

Developer

- Code + Test (selected)
- Integration
- Monitor
- Function Keys

```
1  using System;
2
3  public static void Run(TimerInfo myTimer, ILogger log)
4  {
5      log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
6  }
7
```

Logs

2. Verify execution by viewing the information written to the logs.

Logs

Filesystem Logs Log Level Stop Copy Clear Maximize Leave Feedback

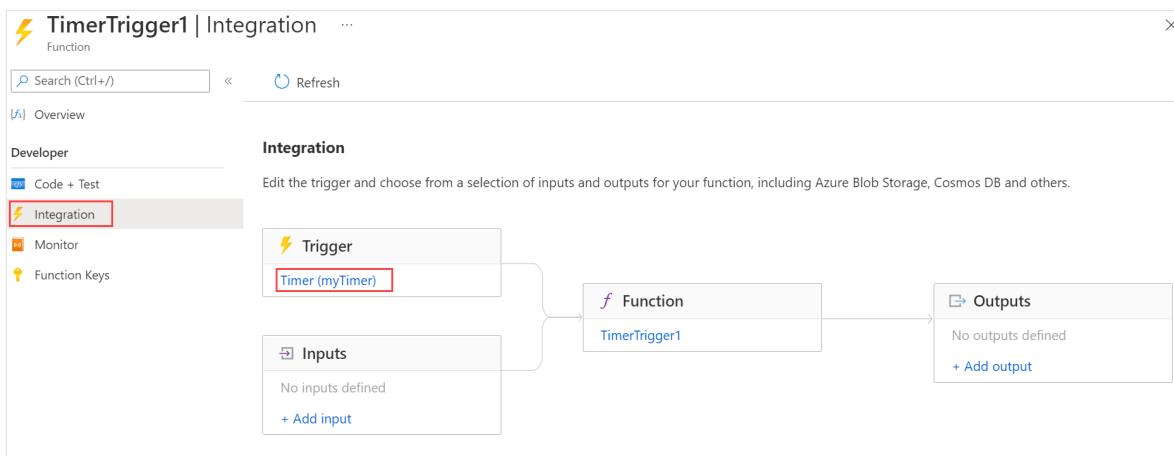
Connected!

2022-05-16T08:55:59.999 [Information] Executing 'Functions.TimerTrigger1' (Reason='Timer fired at 2022-05-16T08:55:59.9989726+00:00', Id=bb5e150c-87d1-4d18-85c3-0d8bee287063)
2022-05-16T08:55:59.999 [Information] C# Timer trigger function executed at: 5/16/2022 8:55:59 AM
2022-05-16T08:56:00.000 [Information] Executed 'Functions.TimerTrigger1' (Succeeded, Id=bb5e150c-87d1-4d18-85c3-0d8bee287063, Duration=0ms)

Now, you change the function's schedule so that it runs once every hour instead of every minute.

Update the timer schedule

1. In your function, select **Integration**. Here, you define the input and output bindings for your function and also set the schedule.
2. Select **Timer (myTimer)**.



3. Update the **Schedule** value to `0 0 */1 * * *`, and then select **Save**.

Edit Trigger

Save Discard Delete

Binding Type: Timer

Timestamp parameter name*: myTimer

Schedule*: 0 0 */1 * * *

You now have a function that runs once every hour, on the hour.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need

the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.

The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with options like 'Functions', 'App keys', 'App files', and 'Proxies'. The main content area is titled 'Overview' and shows details for the resource group 'myResourceGroup'. The 'Resource group' section is highlighted with a red box. It displays the following information:

- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags

At the bottom of the main content area, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart. The Metrics tab is currently selected.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've created a function that runs based on a schedule. For more information about timer triggers, see [Schedule code execution with Azure Functions](#).

Now that you've created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Connect functions to Azure services using bindings

Article • 08/24/2023

When you create a function, language-specific trigger code is added in your project from a set of trigger templates. If you want to connect your function to other services by using input or output bindings, you have to add specific binding definitions in your function. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Local development

When you develop functions locally, you need to update the function code to add bindings. For languages that use `function.json`, [Visual Studio Code](#) provides tooling to add bindings to a function.

Manually add bindings based on examples

When adding a binding to an existing function, you need to add binding-specific attributes to the function definition in code.

The following example shows the function definition after adding a [Queue Storage output binding](#) to an [HTTP triggered function](#):



```
In-process

C#
[FunctionName("HttpExample")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRequest req,
    [Queue("outqueue"), StorageAccount("AzureWebJobsStorage")]
    ICollector<string> msg,
    ILogger log)
```

The way you define the output binding depends on your process model. For more information, including links to example binding code that you can refer to, see [Add bindings to a function](#).

Use the following table to find examples of specific binding types that you can use to guide you in updating an existing function. First, choose the language tab that corresponds to your project.

Binding code for C# depends on the [specific process model](#).

In-process		
Service	Examples	Samples
Blob storage	Trigger Input Output	Link ↗
Azure Cosmos DB	Trigger Input Output	Link ↗
Azure Data Explorer	Input Output	Link ↗
Azure SQL	Trigger Input Output	Link
Event Grid	Trigger Output	Link ↗
Event Hubs	Trigger Output	
IoT Hub	Trigger Output	
HTTP	Trigger	Link ↗
Queue storage	Trigger Output	Link ↗
RabbitMQ	Trigger Output	
SendGrid	Output	
Service Bus	Trigger Output	Link ↗
SignalR	Trigger Input	

Service	Examples	Samples
	Output	
Table storage	Input Output	
Timer	Trigger	Link ↗
Twilio	Output	Link ↗

Visual Studio Code

When you use Visual Studio Code to develop your function and your function uses a `function.json` file, the Azure Functions extension can automatically add a binding to an existing `function.json` file. To learn more, see [Add input and output bindings](#).

Azure portal

When you develop your functions in the [Azure portal ↗](#), you add input and output bindings in the **Integrate** tab for a given function. The new bindings are added to either the `function.json` file or to the method attributes, depending on your language. The following articles show examples of how to add bindings to an existing function in the portal:

- [Queue storage output binding](#)
- [Azure Cosmos DB output binding](#)

Next steps

- [Azure Functions triggers and bindings concepts](#)

Store unstructured data using Azure Functions and Azure Cosmos DB

Article • 10/12/2022

Azure Cosmos DB [↗](#) is a great way to store unstructured and JSON data. Combined with Azure Functions, Azure Cosmos DB makes storing data quick and easy with much less code than required for storing data in a relational database.

ⓘ Note

At this time, the Azure Cosmos DB trigger, input bindings, and output bindings work with SQL API and Graph API accounts only.

In Azure Functions, input and output bindings provide a declarative way to connect to external service data from your function. In this article, learn how to update an existing function to add an output binding that stores unstructured data in an Azure Cosmos DB document.

Prerequisites

To complete this tutorial:

This topic uses as its starting point the resources created in [Create your first function from the Azure portal](#). If you haven't already done so, please complete these steps now to create your function app.

Create an Azure Cosmos DB account

You must have an Azure Cosmos DB account that uses the SQL API before you create the output binding.

1. From the Azure portal menu or the [Home page](#), select **Create a resource**.
2. Search for **Azure Cosmos DB**. Select **Create > Azure Cosmos DB**.
3. On the **Create an Azure Cosmos DB account** page, select the **Create** option within the **Azure Cosmos DB for NoSQL** section.

Azure Cosmos DB provides several APIs:

- NoSQL, for document data
- PostgreSQL
- MongoDB, for document data
- Apache Cassandra
- Table
- Apache Gremlin, for graph data

To learn more about the API for NoSQL, see [Welcome to Azure Cosmos DB](#).

4. In the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos DB account.

Setting	Value	Description
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Resource group name	Select a resource group, or select Create new , then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos DB account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URL, use a unique name. The name can contain only lowercase letters, numbers, and the hyphen (-) character. It must be 3-44 characters.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select Provisioned throughput to create an account in provisioned throughput mode. Select Serverless to create an account in serverless mode.
Apply Azure Cosmos DB free tier discount	Apply or Do not apply	With Azure Cosmos DB free tier, you get the first 1000 RU/s and 25 GB of storage for free in an account. Learn more about free tier .
Limit total account throughput	Selected or not	Limit the total amount of throughput that can be provisioned on this account. This limit prevents unexpected charges related to provisioned throughput. You can update or remove this limit after your account is created.

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt in when creating the account. If you don't see the option to apply

the free tier discount, another account in the subscription has already been enabled with free tier.

Home > Marketplace > Create an Azure Cosmos DB account >

Create Azure Cosmos DB Account - Azure Cosmos DB for NoSQL

Basics Global Distribution Networking Backup Policy Encryption Tags Review + create

Azure Cosmos DB is a fully managed NoSQL and relational database service for building scalable, high performance applications. Try it for free, for 30 days with unlimited renewals. Go to production starting at \$24/month per database, multiple containers included. [Learn more](#)

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Contoso Subscription

Resource Group * myResourceGroup [Create new](#)

Instance Details

Account Name * mysqlaccount

Location * (US) East US 2

Capacity mode ⓘ Provisioned throughput Serverless [Learn more about capacity mode](#)

With Azure Cosmos DB free tier, you will get the first 1000 RU/s and 25 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$64/month discount per account.

Apply Free Tier Discount Apply Do Not Apply

Limit total account throughput Limit the total amount of throughput that can be provisioned on this account
 ⓘ This limit will prevent unexpected charges related to provisioned throughput. You can update or remove this limit after your account is created.

[Review + create](#) [Previous](#) [Next: Global Distribution](#)

ⓘ Note

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Limit total account throughput

5. In the **Global Distribution** tab, configure the following details. You can leave the default values for this quickstart:

Setting	Value	Description
Geo-Redundancy	Disable	Enable or disable global distribution on your account by pairing your region with a pair region. You can add more regions to your account later.

Setting	Value	Description
Multi-region Writes	Disable	Multi-region writes capability allows you to take advantage of the provisioned throughput for your databases and containers across the globe.
Availability Zones	Disable	Availability Zones help you further improve availability and resiliency of your application.

① Note

The following options are not available if you select **Serverless** as the **Capacity mode** in the previous **Basics** page:

- Geo-redundancy
- Multi-region Writes

6. Optionally, you can configure more details in the following tabs:

- **Networking**. Configure [access from a virtual network](#).
- **Backup Policy**. Configure either [periodic](#) or [continuous](#) backup policy.
- **Encryption**. Use either service-managed key or a [customer-managed key](#).
- **Tags**. Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups.

7. Select **Review + create**.

8. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

The screenshot shows the Azure portal's deployment overview page. At the top, it displays the deployment name: Microsoft.Azure.CosmosDB-20230302095414 | Overview. Below this, a message says "Your deployment is complete". It provides deployment details: Deployment name: Microsoft.Azure.CosmosDB-20230302095414, Subscription: Contoso Subscription, Resource group: myResourceGroup, Start time: 3/2/2023, 9:54:17 AM, and Correlation ID: d7e2b2c3-580b-4e8b-ba8d-cc5d5dd8cd3b. There are buttons for Delete, Cancel, Redeploy, Download, and Refresh. On the left, there's a sidebar with options: Overview (selected), Inputs, Outputs, and Template. At the bottom, there are links for "Deployment details", "Next steps", and a "Go to resource" button.

9. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the Azure portal's quick start page for the mysqlaccount Azure Cosmos DB account. The title is "mysqlaccount | Quick start". It says "Congratulations! Your Azure Cosmos DB account was created." and "Now, let's connect to it using a sample app:". A "Choose a platform" section offers .NET, Java, Node.js, and Python. Step 1: Add a container is described with the note: "In Azure Cosmos DB, data is stored in containers." and a "Create 'Items' container" button. Step 2: Download and run your .NET app is described with the note: "Once container is created, download a sample .NET app connected to it, extract, build and run." and a "Download" button. On the left, a sidebar lists: Overview (selected), Activity log, Access control (IAM), Tags, Diagnose and solve problems, Cost Management, Quick start (selected), Notifications, Data Explorer, Settings, and Features.

Add an output binding

1. In the Azure portal, navigate to and select the function app you created previously.
2. Select **Functions**, and then select the HttpTrigger function.

Home > Function App > myFunctionApp-dma > myFunctionApp-dma | Functions

myFunctionApp-dma | Functions

App Service

+ Add Develop Locally Refresh Enable Disable Delete

Search (Ctrl+/
)

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Security

Functions (preview)

Functions (highlighted)

App keys App files

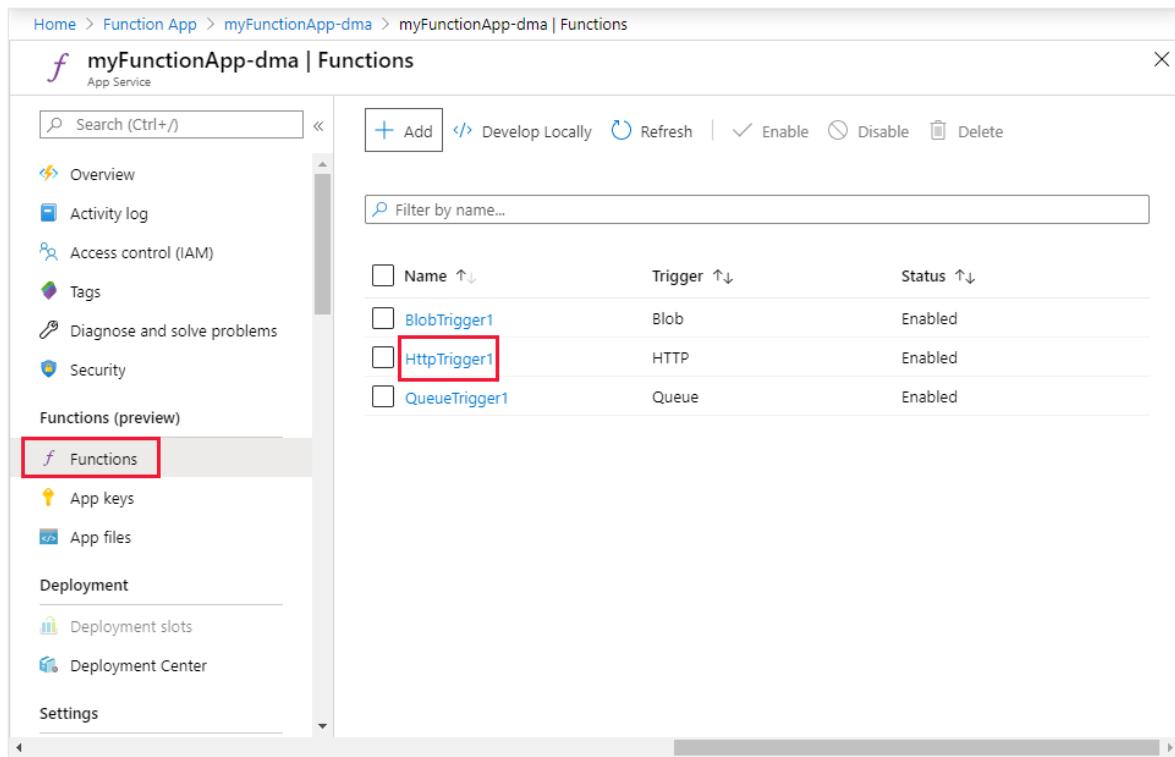
Deployment

Deployment slots Deployment Center

Settings

Name ↑↓	Trigger ↑↓	Status ↑↓
BlobTrigger1	Blob	Enabled
HttpTrigger1	HTTP	Enabled
QueueTrigger1	Queue	Enabled

Filter by name...



3. Select Integration and + Add output.

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

Trigger

HTTP (req)

Inputs

No inputs defined

+ Add input

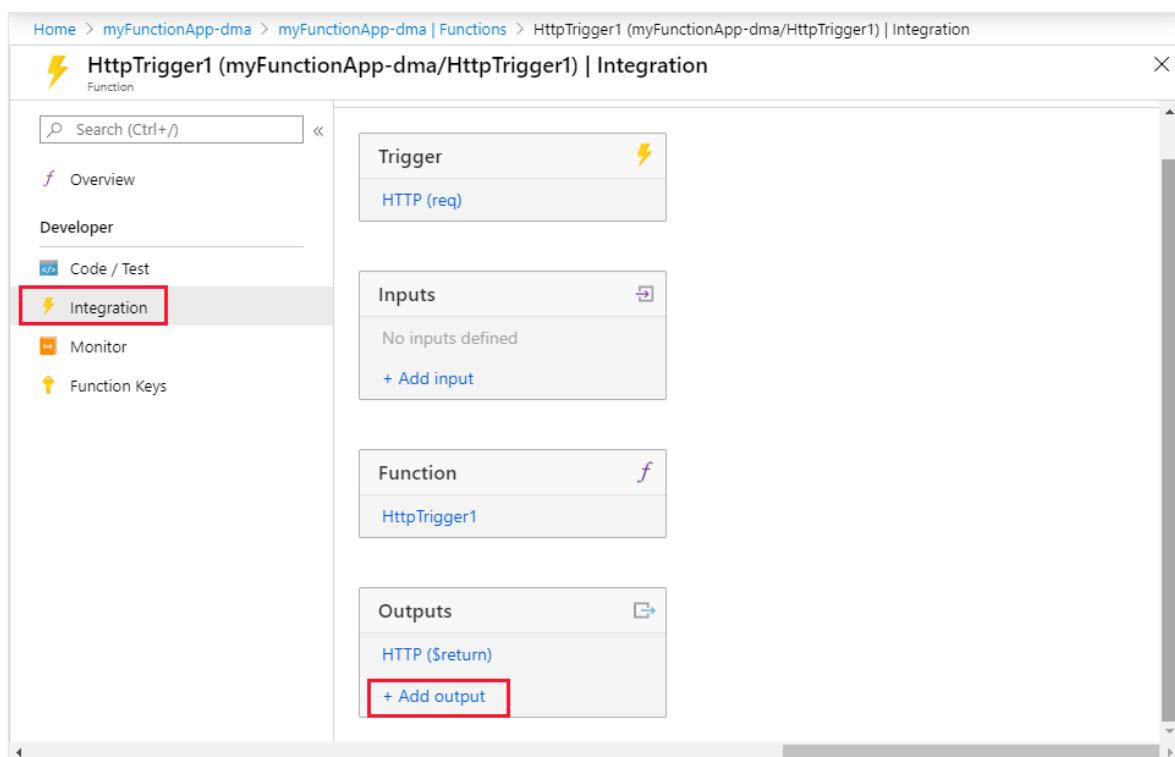
Function

HttpTrigger1

Outputs

HTTP (\$return)

+ Add output (highlighted)



4. Use the Create Output settings as specified in the table:

Create Output

X

Start by selecting the type of output binding you want to add.

Binding Type

Azure Cosmos DB

Azure Cosmos DB details

Document parameter name* ⓘ

taskDocument

Database name* ⓘ

taskDatabase

Collection Name* ⓘ

taskCollection

If true, creates the Cosmos DB databas...* ⓘ

Yes

Cosmos DB account connection* ⓘ

dma-cosmosdb_DOCUMENTDB

New

Partition key (optional) ⓘ

OK

Cancel

Setting	Suggested value	Description
Binding Type	Azure Cosmos DB	Name of the binding type to select to create the output binding to Azure Cosmos DB.
Document parameter name	taskDocument	Name that refers to the Azure Cosmos DB object in code.
Database name	taskDatabase	Name of database to save documents.

Setting	Suggested value	Description
Collection name	taskCollection	Name of the database collection.
If true, creates the Azure Cosmos DB database and collection	Yes	The collection doesn't already exist, so create it.
Azure Cosmos DB account connection	New setting	Select New , then choose Azure Cosmos DB Account and the Database account you created earlier, and then select OK . Creates an application setting for your account connection. This setting is used by the binding to connection to the database.

5. Select **OK** to create the binding.

Update the function code

Replace the existing function code with the following code, in your chosen language:

C#

Replace the existing C# function with the following code:

```
C#  
  

#r "Newtonsoft.Json"  
  

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;  
  

public static IActionResult Run(HttpContext req, out object taskDocument, ILogger log)
{
    string name = req.Query["name"];
    string task = req.Query["task"];
    string duedate = req.Query["duedate"];  
  

    // We need both name and task parameters.
    if (!string.IsNullOrEmpty(name) && !string.IsNullOrEmpty(task))
    {
        taskDocument = new
        {
            name,
```

```

        duedate,
        task
    };

    return (ActionResult)new OkResult();
}
else
{
    taskDocument = null;
    return (ActionResult)new BadRequestResult();
}
}

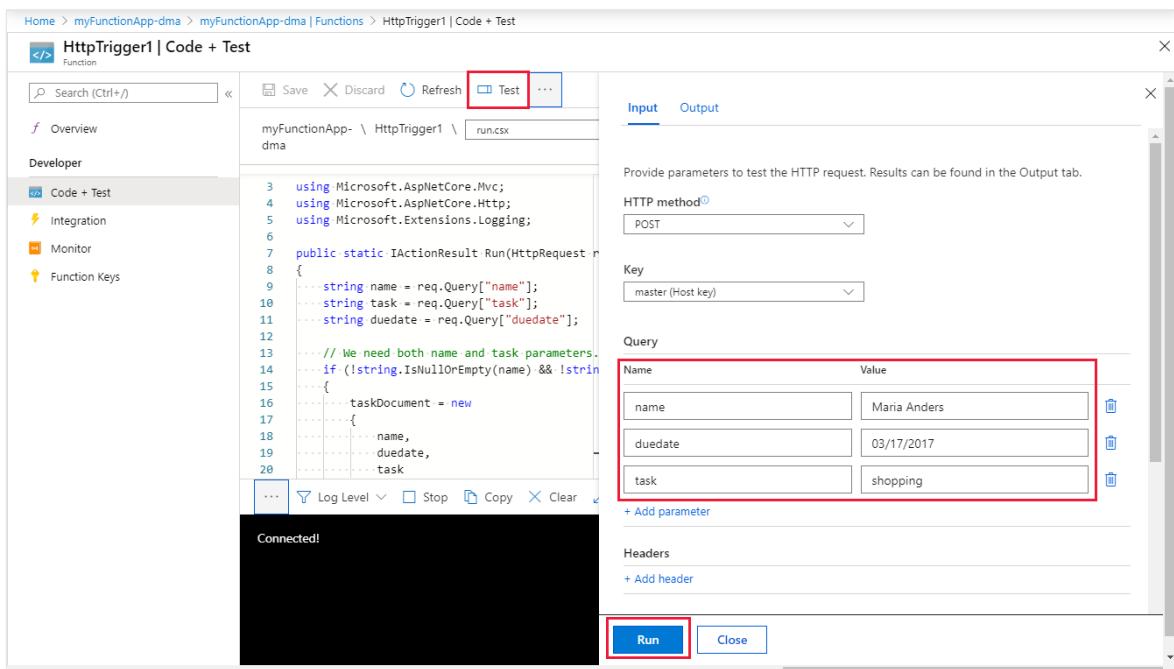
```

This code sample reads the HTTP Request query strings and assigns them to fields in the `taskDocument` object. The `taskDocument` binding sends the object data from this binding parameter to be stored in the bound document database. The database is created the first time the function runs.

Test the function and database

1. Select **Test/Run**. Under **Query**, select **+ Add parameter** and add the following parameters to the query string:

- `name`
- `task`
- `duedate`



2. Select **Run** and verify that a 200 status is returned.

```

9     string name = req.Query["name"];
10    string task = req.Query["task"];
11    string duedate = req.Query["duedate"];
12
13    // We need both name and task parameters.
14    if (!string.IsNullOrEmpty(name) && !string.IsNullOrEmpty(task))
15    {
16        taskDocument = new
17        {
18            name,
19            duedate,
20            task
21        };
22
23        return (ActionResult)new OkResult();
24    }
25    else

```

Connected!
2020-04-14T05:10:00Z [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=ce811ba9-dfba-42d1-a0cb-a7caed0fc5)
2020-04-14T05:09:59Z [Information] Executing 'Functions.HttpTrigger1' (Reason="This function was programmatically called via the host APIs.", Id=ce811ba9-dfba-42d1-a0cb-a7caed0fc5)

3. In the Azure portal, search for and select Azure Cosmos DB.

Azure Cosmos DB

Services See all

- Azure Cosmos DB**
- Azure Database for MySQL servers
- Azure Arc
- Azure Databricks
- Azure DevOps
- Azure Lighthouse
- Azure Migrate
- Azure Sentinel
- Azure SQL
- Azure Database for MariaDB servers

Resources

No results were found.

4. Choose your Azure Cosmos DB account, then select Data Explorer.

5. Expand the **TaskCollection** nodes, select the new document, and confirm that the document contains your query string values, along with some additional metadata.

	id	/p...
1	32be...	
2	ec24...	
3	9c03...	
4	7d43...	
5	566e...	
6	2c2f...	
7	0435...	
8	6fe4...	
9		
10		
11		

You've successfully added a binding to your HTTP trigger to store unstructured data in an Azure Cosmos DB instance.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

For more information about binding to an Azure Cosmos DB instance, see [Azure Functions Azure Cosmos DB bindings](#).

- [Azure Functions triggers and bindings concepts](#)
Learn how Functions integrates with other services.

- [Azure Functions developer reference](#)

Provides more technical information about the Functions runtime and a reference for coding functions and defining triggers and bindings.

- [Code and test Azure Functions locally](#)

Describes the options for developing your functions locally.

Add messages to an Azure Storage queue using Functions

Article • 07/02/2024

In Azure Functions, input and output bindings provide a declarative way to make data from external services available to your code. In this article, you use an output binding to create a message in a queue when an HTTP request triggers a function. You use Azure storage container to view the queue messages that your function creates.

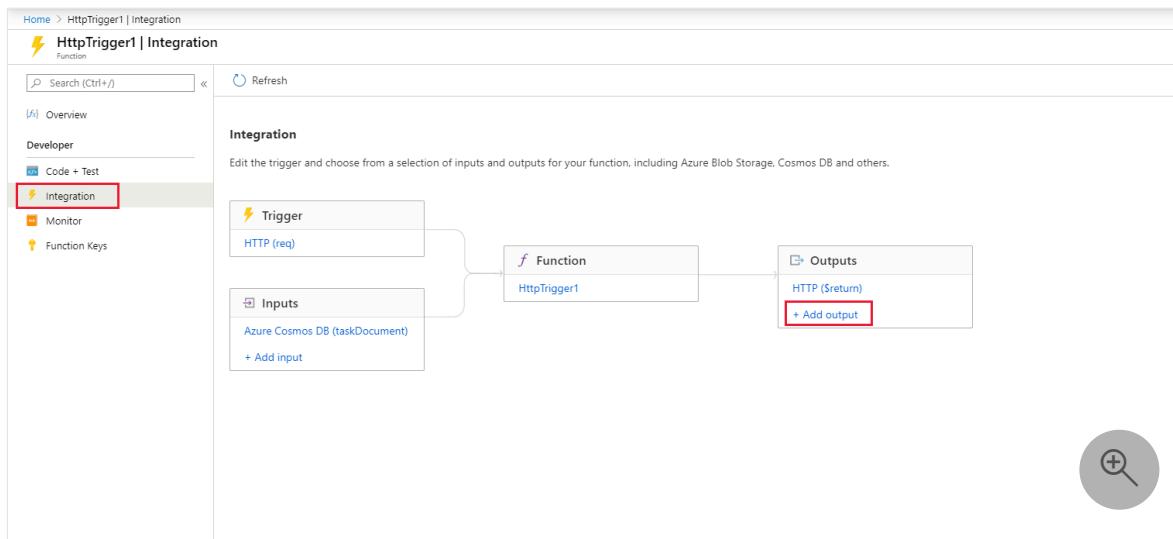
Prerequisites

- An Azure subscription. If you don't have one, create a [free account](#) before you begin.
- Follow the directions in [Create your first function in the Azure portal](#), omitting the **Clean up resources** step, to create the function app and function to use in this article.

Add an output binding

In this section, you use the portal UI to add an Azure Queue Storage output binding to the function you created in the prerequisites. This binding makes it possible to write minimal code to create a message in a queue. You don't need to write code for such tasks as opening a storage connection, creating a queue, or getting a reference to a queue. The Azure Functions runtime and queue output binding take care of those tasks for you.

1. In the Azure portal, search for and select the function app that you created in [Create your first function from the Azure portal](#).
2. In your function app, select the function that you created.
3. Select **Integration**, and then select **+ Add output**.



4. Select the **Azure Queue Storage** binding type and add the settings as specified in the table that follows this screenshot:

Create Output

Start by selecting the type of output binding you want to add.

Binding Type

Azure Queue Storage

Azure Queue Storage details

Message parameter name* (i)

outputQueueItem

Queue name* (i)

outqueue

Storage account connection* (i)

AzureWebJobsStorage

New

OK **Cancel**

Setting	Suggested value	description
Message parameter name	outputQueueItem	The name of the output binding parameter.
Queue name	outqueue	The name of the queue to connect to in your storage account.
Storage account connection	AzureWebJobsStorage	You can use the existing storage account connection used by your function app or create a new one.

5. Select **OK** to add the binding.

Now that you have an output binding defined, you need to update the code to use the binding to add messages to a queue.

Add code that uses the output binding

In this section, you add code that writes a message to the output queue. The message includes the value passed to the HTTP trigger in the query string. For example, if the query string includes `name=Azure`, the queue message is *Name passed to the function: Azure*.

1. In your function, select **Code + Test** to display the function code in the editor.
2. Update the function code, according to your function language:

C#

Add an `outputQueueItem` parameter to the method signature as shown in the following example:

cs

```
public static async Task<IActionResult> Run(HttpContext req,
      ICollector<string> outputQueueItem, ILogger log)
{
    ...
}
```

In the body of the function, just before the `return` statement, add code that uses the parameter to create a queue message:

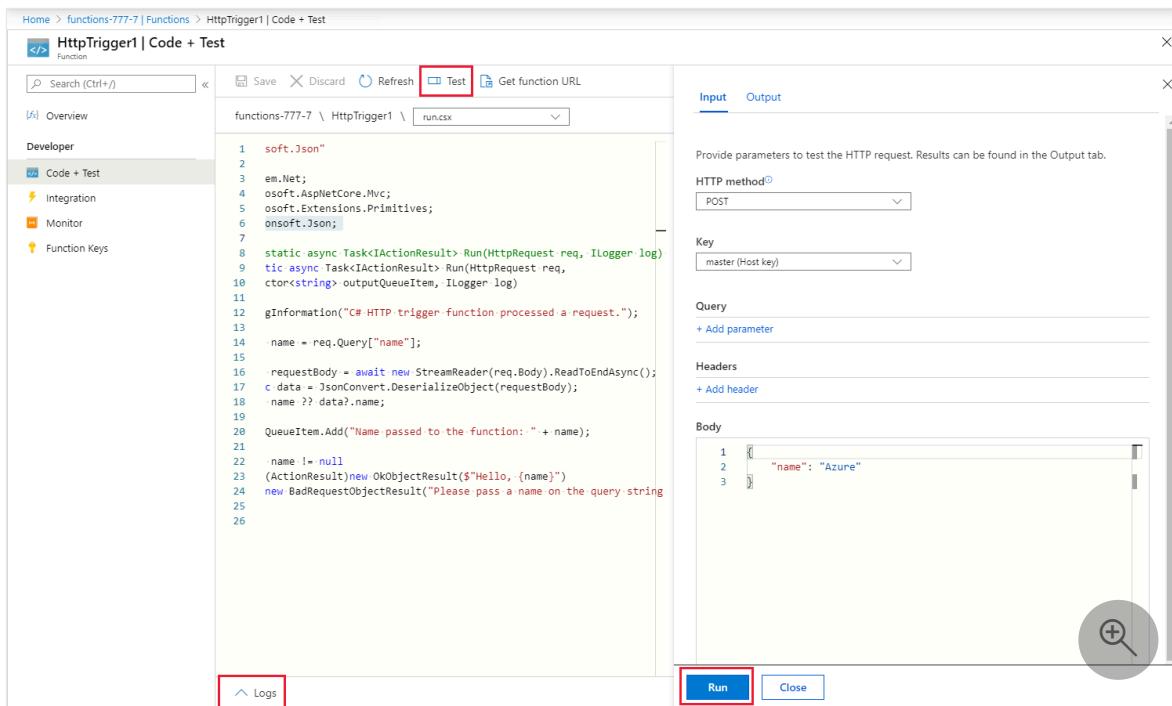
CS

```
outputQueueItem.Add("Name passed to the function: " + name);
```

3. Select **Save** to save your changes.

Test the function

1. After the code changes are saved, select **Test**.
2. Confirm that your test matches this screenshot, and then select **Run**.



Notice that the **Request body** contains the `name` value *Azure*. This value appears in the queue message created when the function is invoked.

As an alternative to selecting **Run**, you can call the function by entering a URL in a browser and specifying the `name` value in the query string. This browser method is shown in [Create your first function from the Azure portal](#).

3. Check the logs to make sure that the function succeeded.

A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You use storage account to verify that the queue and a message in it were created.

Find the storage account connected to AzureWebJobsStorage

1. In your function app, expand **Settings**, and then select **Environment variables**.
2. In the **App settings** tab, select **AzureWebJobsStorage**.

The screenshot shows the Azure portal interface for a function app named "function-app-py". On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, and Functions. The "Settings" section is expanded, and "Environment variables" is selected, highlighted with a red box. The main content area is titled "App settings" and shows a table of environment variables:

Name	Value	Deployment slot setting	Source	Delete
APPLICATIONINSIGHTS_CONN...	(Show value)		App Service	[Delete]
AzureWebJobsFeatureFlags	(Show value)		App Service	[Delete]
AzureWebJobsStorage	(Show value)		App Service	[Delete]
DEPLOYMENT_STORAGE_CON...	(Show value)		App Service	[Delete]
FUNCTIONS_EXTENSION_VERS...	(Show value)		App Service	[Delete]

At the bottom, there are "Apply" and "Discard" buttons, and a "Send us your feedback" link.

3. Locate and make note of the account name.

The screenshot shows the "Add/Edit application setting" dialog. The "Name" field contains "AzureWebJobsStorage" and the "Value" field contains "DefaultEndpointsProtocol=https;AccountName=storageaccountmyresbcb9;AccountKey=/fLt9Wtr+z/30JESevURLqh4aiO8...". A red box highlights the "AccountName" part of the value. There's a checkbox for "Deployment slot setting" which is unchecked. A search icon is visible at the bottom right.

Examine the output queue

1. In the resource group for your function app, select the storage account that you're using.
2. Under Queue service, select **Queues**, and select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name**

value of *Azure*, the queue message is *Name passed to the function: Azure*.

3. Run the function again.

A new message appears in the queue.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Related content

In this article, you added an output binding to an existing function. For more information about binding to Queue Storage, see [Queue Storage trigger and bindings](#).

- [Azure Functions triggers and bindings concepts](#)
Learn how Functions integrates with other services.
- [Azure Functions developer reference](#)
Provides more technical information about the Functions runtime and a reference for coding functions and defining triggers and bindings.
- [Code and test Azure Functions locally](#)
Describes the options for developing your functions locally.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Install [.NET Core CLI tools](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings...`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Except for HTTP and timer triggers, bindings are implemented as extension packages. Run the following [dotnet add package](#) command in the Terminal window to add the Storage extension package to your project.

```
Isolated process

Bash

dotnet add package
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues --prerelease
```

Now, you can add the storage output binding to your project.

Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

Isolated worker model

Open the `HttpExample.cs` project file and add the following `MultiResponse` class:

C#

```
public class MultiResponse
{
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Isolated worker model

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
```

```

    {
        var logger = ExecutionContext.GetLogger("HttpExample");
        logger.LogInformation("C# HTTP trigger function processed a
request.");
    }

        var message = "Welcome to Azure Functions!";

        var response = req.CreateResponse(HttpStatusCode.OK);
        response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
        response.WriteString(message);

        // Return a response to both HTTP trigger and storage output
binding.
        return new MultiResponse()
    {
        // Write a single message.
        Messages = new string[] { message },
        HttpResponseMessage = response
    };
}
}

```

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure. If you don't already have Core Tools installed locally, you are prompted to install it the first time you run your project.

1. To call your function, press **F5** to start the function app project. The **Terminal** panel displays the output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    AZURE: ACTIVITY LOG
✖ host start - Task ✓ + ▾ □ ▢ ▲ ×

Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

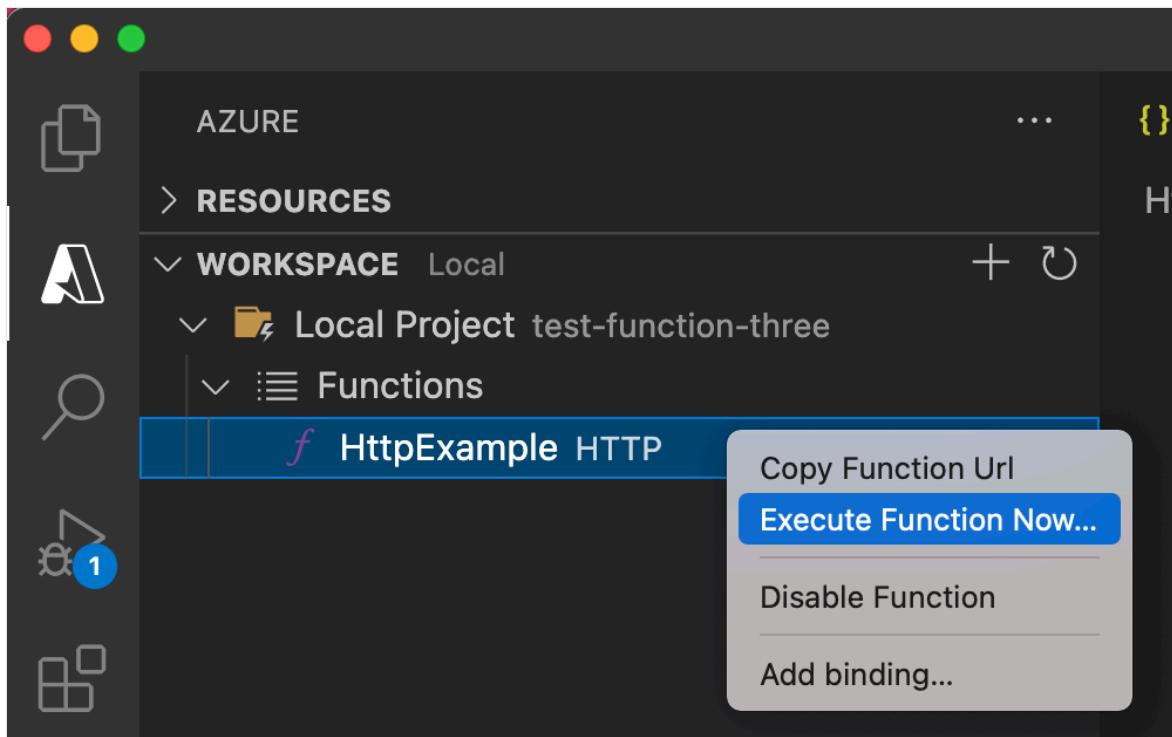
For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.

```

If you don't already have Core Tools installed, select **Install** to install Core Tools when prompted to do so.

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

- With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Windows) or `ctrl - click` (macOS) the `HttpExample` function and choose **Execute Function Now....**

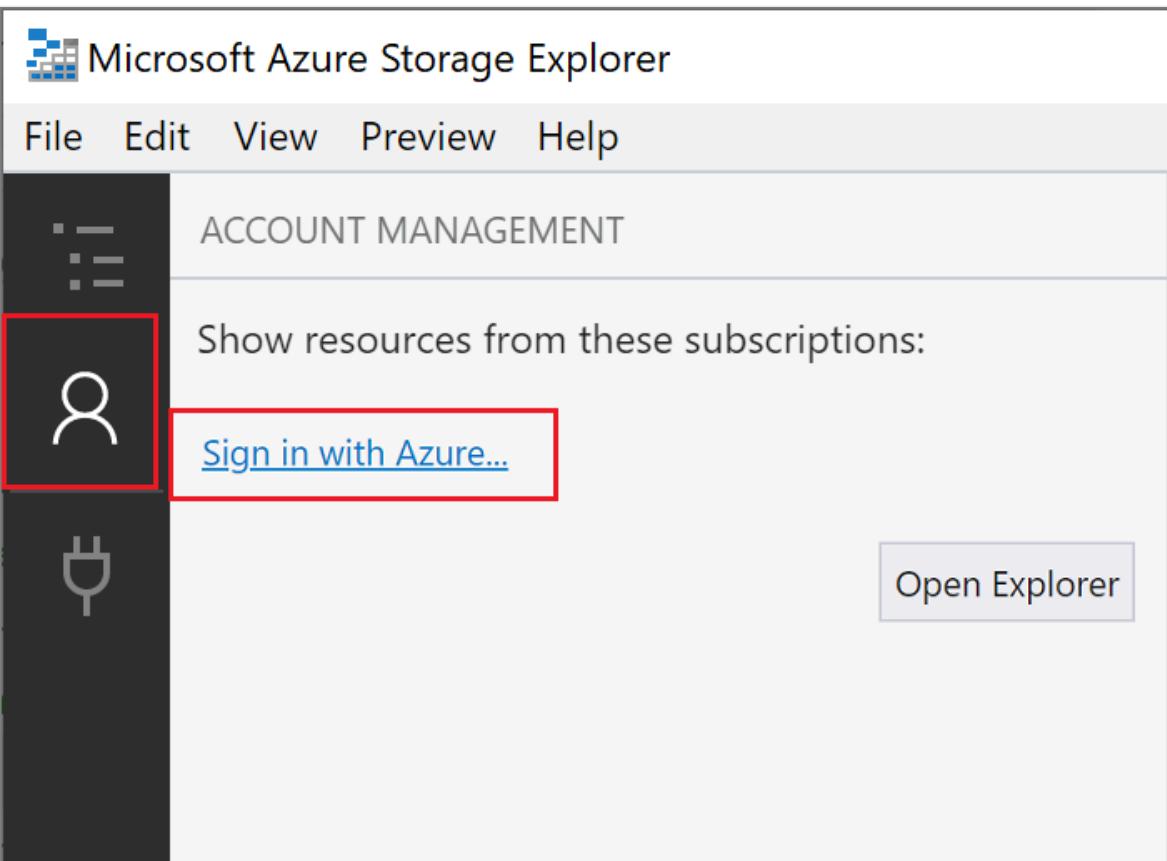


- In the **Enter request body**, press `Enter` to send a request message to your function.
- When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in the **Terminal** panel.
- Press `Ctrl + C` to stop Core Tools and disconnect the debugger.

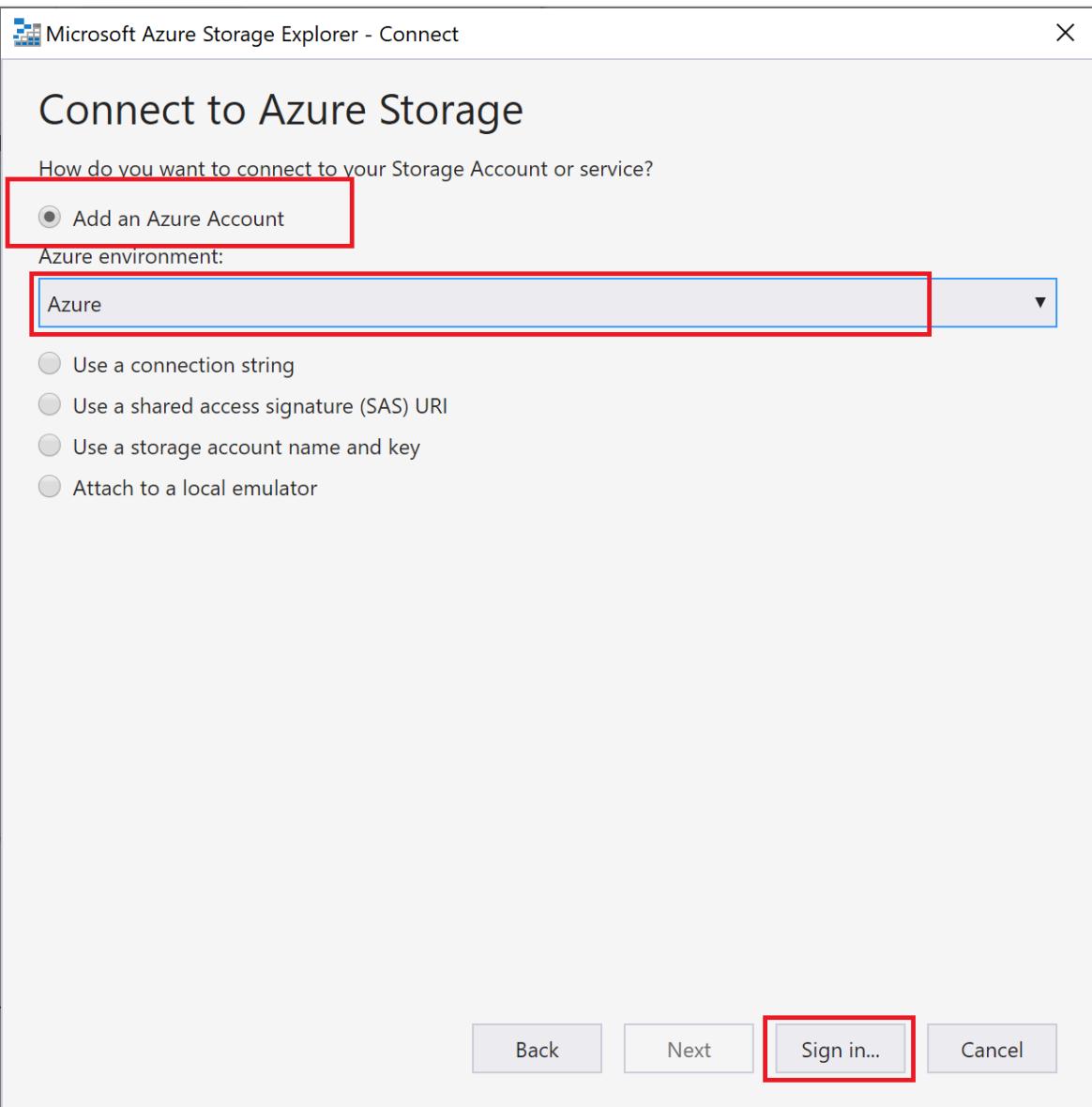
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

- Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

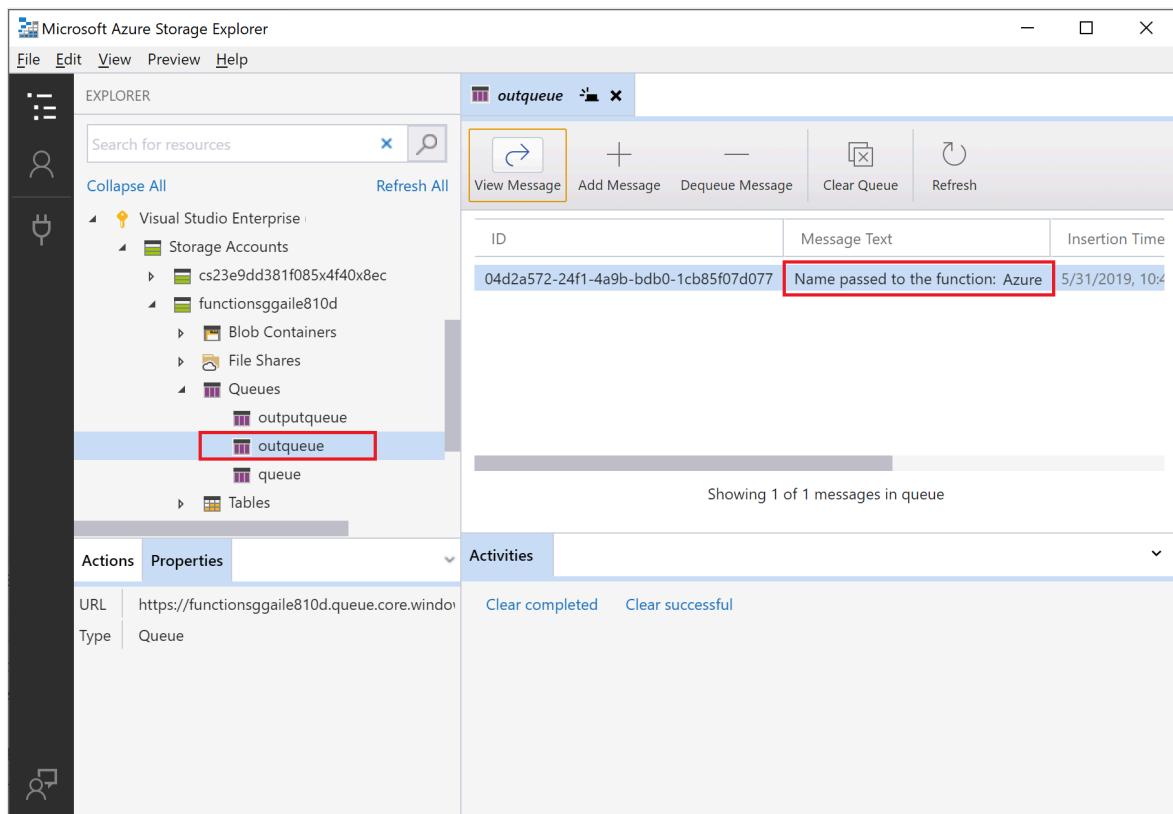


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

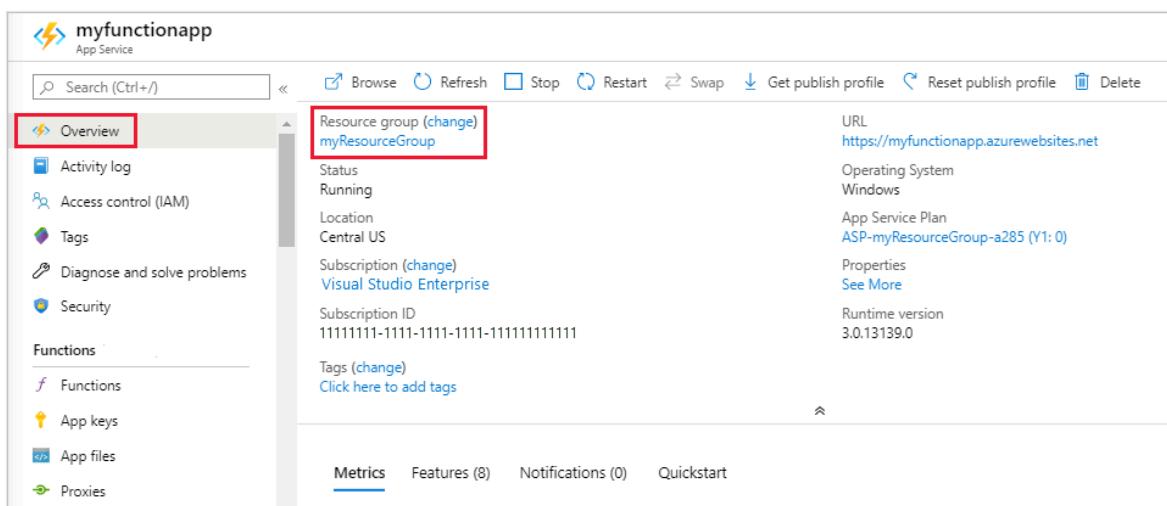
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the "Overview" tab, which displays details about the app: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (with a link to add tags). A red box highlights the "Resource group (change)" link, which points to "myResourceGroup". To the right of the resource group details, there are sections for URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties, and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in C#.](#)
- [Azure Functions C# developer reference](#)

Connect functions to Azure Storage using Visual Studio

Article • 03/31/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio to connect the function you created in the [previous quickstart article](#) to Azure Storage. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the Storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Prerequisites

Before you start this article, you must:

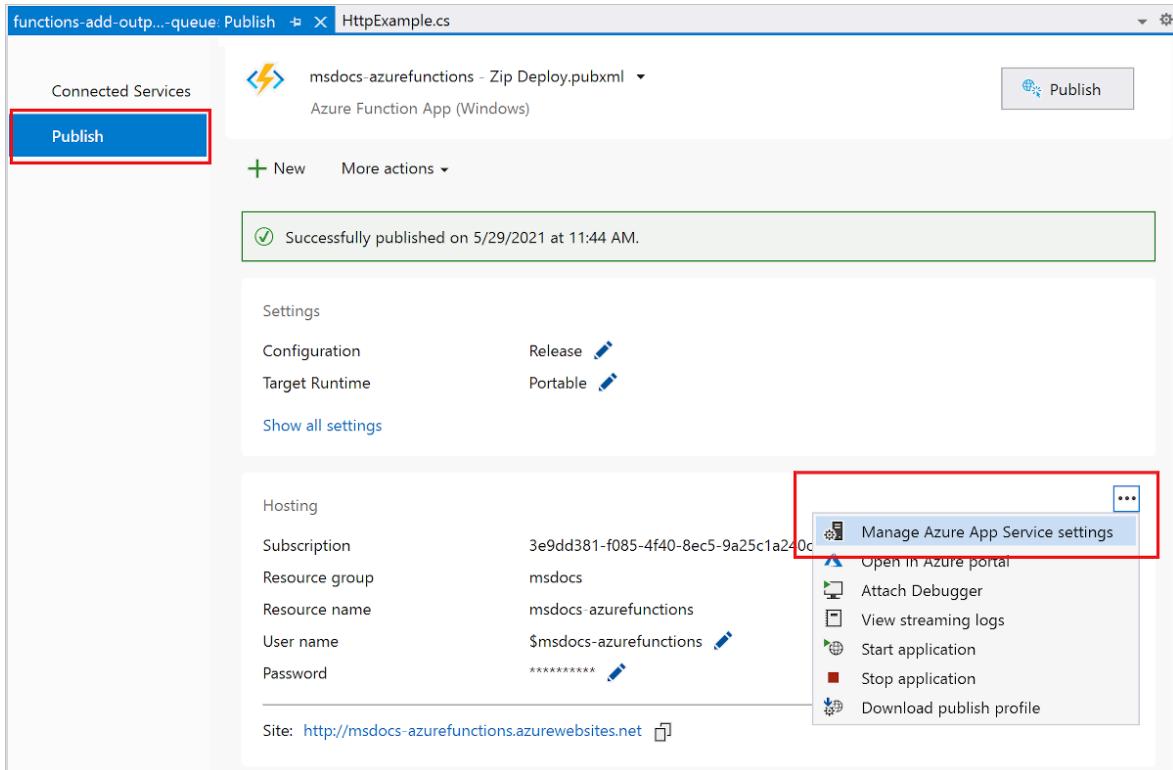
- Complete [part 1 of the Visual Studio quickstart](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Sign in to your Azure subscription from Visual Studio.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required Storage account. The connection string for this account is stored securely in app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your Storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. In **Solution Explorer**, right-click the project and select **Publish**.

2. In the Publish tab under **Hosting**, expand the three dots (...) and select **Manage Azure App Service settings**.



3. Under **AzureWebJobsStorage**, copy the **Remote** string value to **Local**, and then select **OK**.

The storage binding, which uses the `AzureWebJobsStorage` setting for the connection, can now connect to your Queue storage when running locally.

Register binding extensions

Because you're using a Queue storage output binding, you need the Storage bindings extension installed before you run the project. Except for HTTP and timer triggers, bindings are implemented as extension packages.

1. From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.
2. In the console, run the following `Install-Package` command to install the Storage extensions:

Isolated worker model

Bash

```
Install-Package  
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues
```

Now, you can add the storage output binding to your project.

Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

Isolated worker model

Open the *HttpExample.cs* project file and add the following `MultiResponse` class:

C#

```
public class MultiResponse  
{  
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]  
    public string[] Messages { get; set; }  
    public HttpResponseMessage HttpResponseMessage { get; set; }  
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpExample");
    logger.LogInformation("C# HTTP trigger function processed a
request.");

    var message = "Welcome to Azure Functions!";

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
    response.WriteString(message);

    // Return a response to both HTTP trigger and storage output
binding.
    return new MultiResponse()
    {
        // Write a single message.
        Messages = new string[] { message },
        HttpResponseMessage = response
    };
}
```

Run the function locally

1. To run your function, press `F5` in Visual Studio. You might need to enable a firewall exception so that the tools can handle HTTP requests. Authorization levels are never enforced when you run a function locally.
2. Copy the URL of your function from the Azure Functions runtime output.

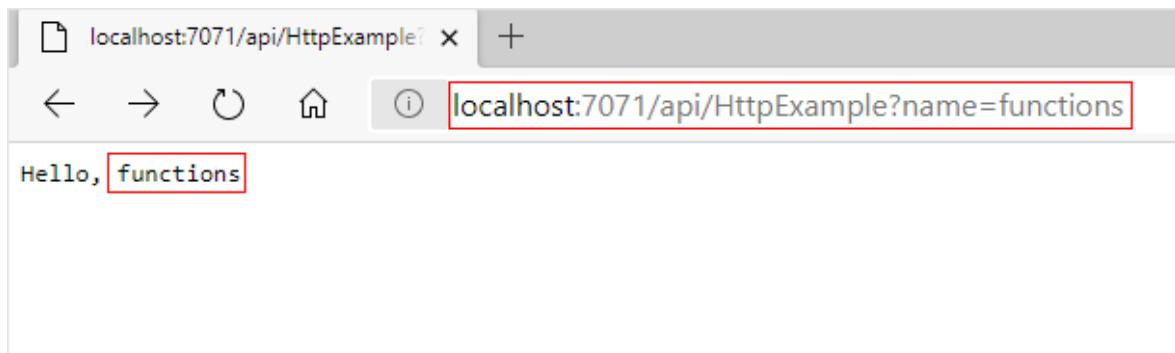
```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.49.0\cli_x64\func.exe
[5/27/2020 7:53:39 AM] Loading functions metadata
[5/27/2020 7:53:39 AM] 1 functions loaded
[5/27/2020 7:53:39 AM] Generating 1 job function(s)
[5/27/2020 7:53:40 AM] Found the following functions:
[5/27/2020 7:53:40 AM] FunctionApp2.HttpExample.Run
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Initializing function HTTP routes
[5/27/2020 7:53:40 AM] Mapped function route 'api/HttpExample' [get,post] to 'HttpExample'
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Host initialized (691ms)
[5/27/2020 7:53:40 AM] Host started (712ms)
[5/27/2020 7:53:40 AM] Job host started
Hosting environment: Development
Content root path: C:\source\repos\FunctionApp\FunctionApp\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[5/27/2020 7:53:47 AM] Host lock lease acquired by instance ID '00000000000000000000000000000000FB2CECE'.
```

- Paste the URL for the HTTP request into your browser's address bar and run the request. The following image shows the response in the browser to the local GET request returned by the function:



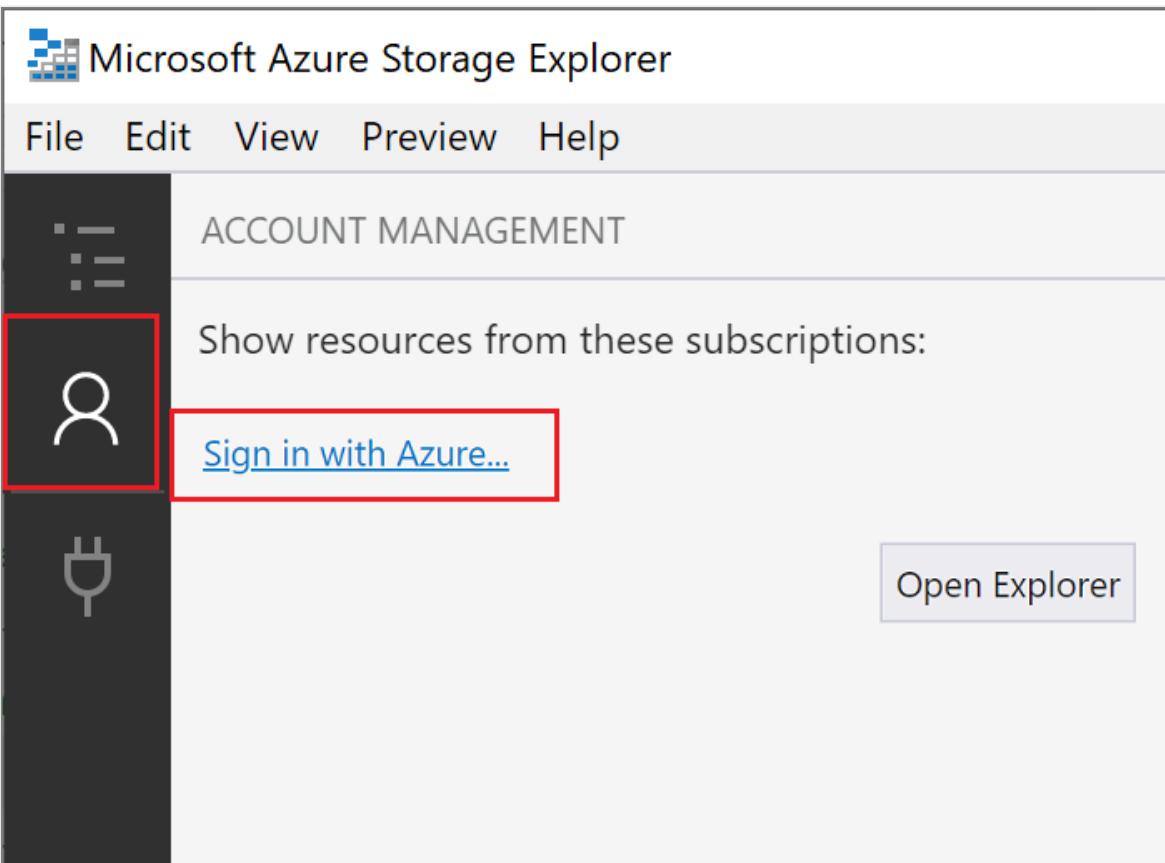
- To stop debugging, press **Shift + F5** in Visual Studio.

A new queue named `outqueue` is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

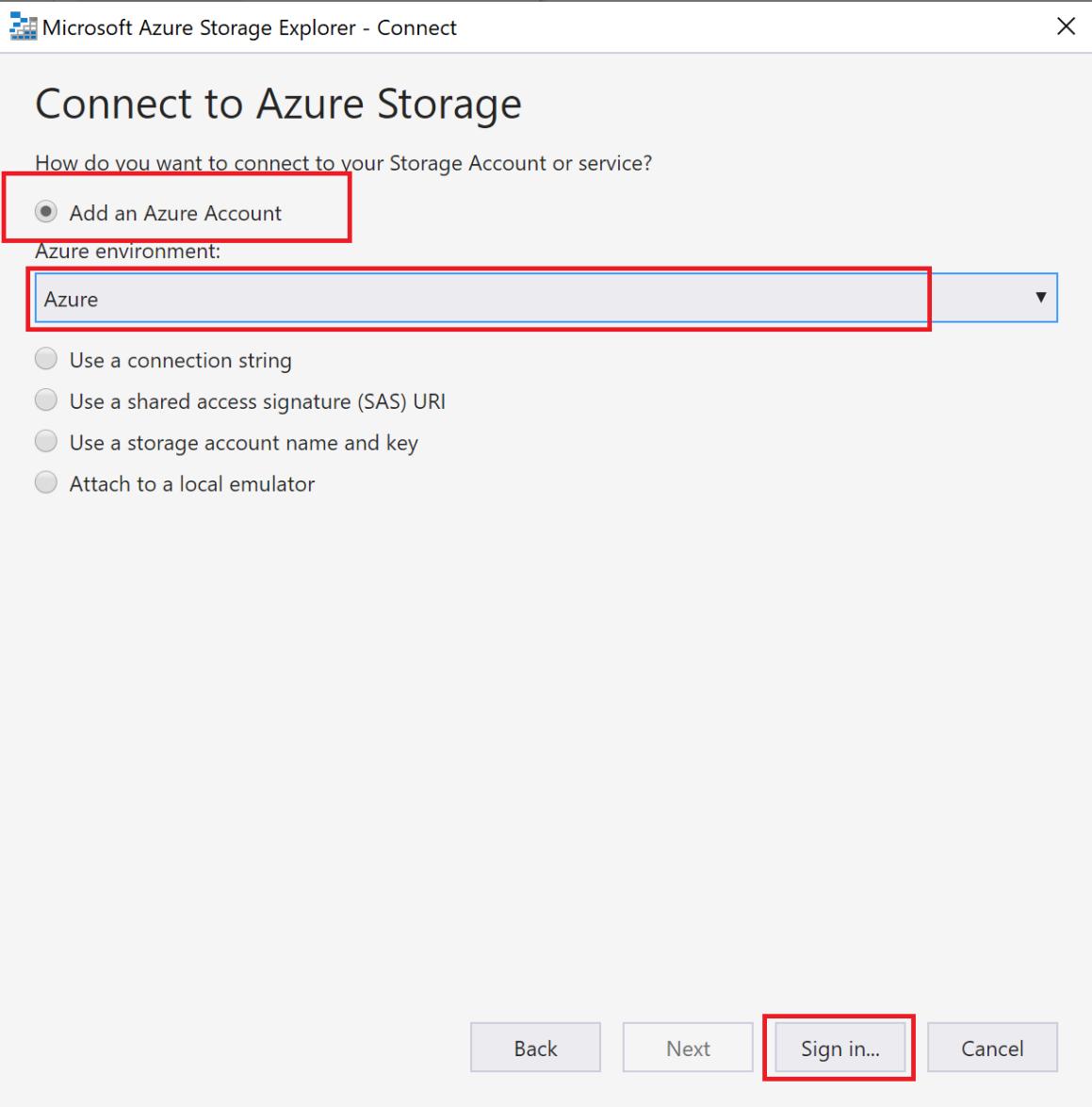
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

- Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

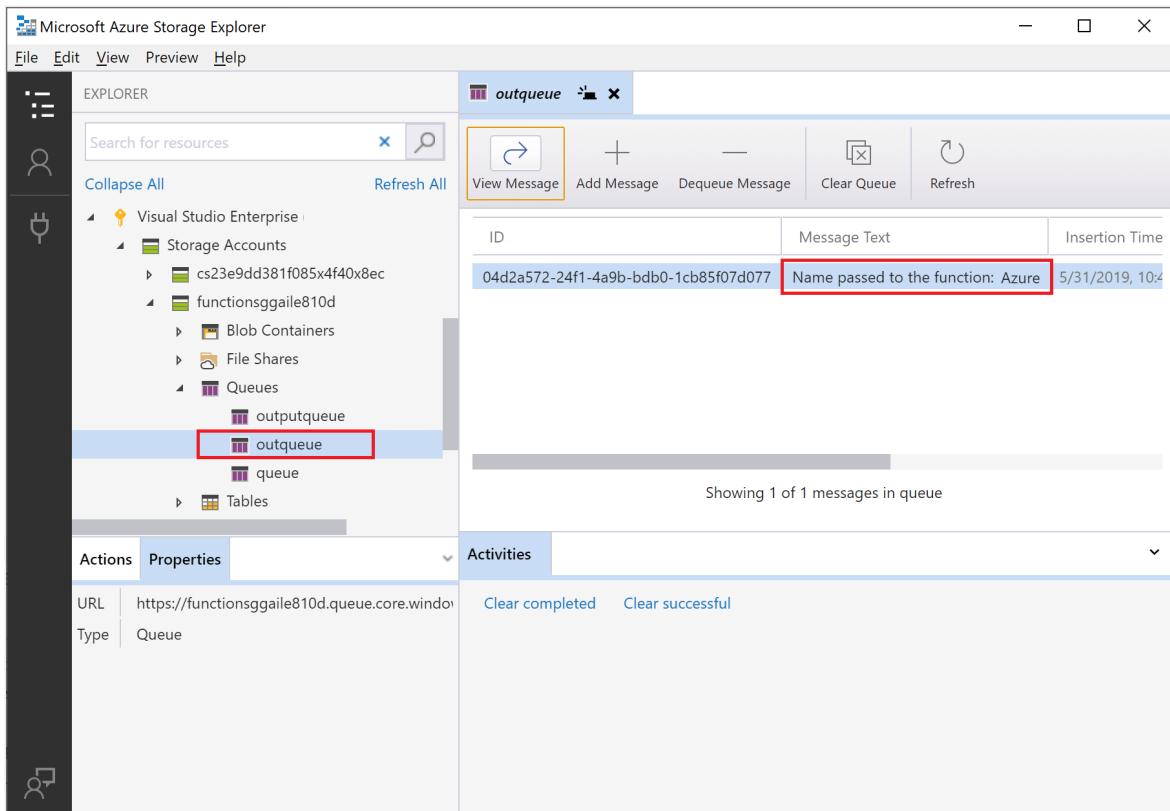


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Storage Explorer, expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default `name` value of *Azure*, the queue message is *Name passed to the function: Azure*.



- Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

- In **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
- After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL.
- Again [view the message in the storage queue](#) to verify that the output binding again generates a new message in the queue.

Clean up resources

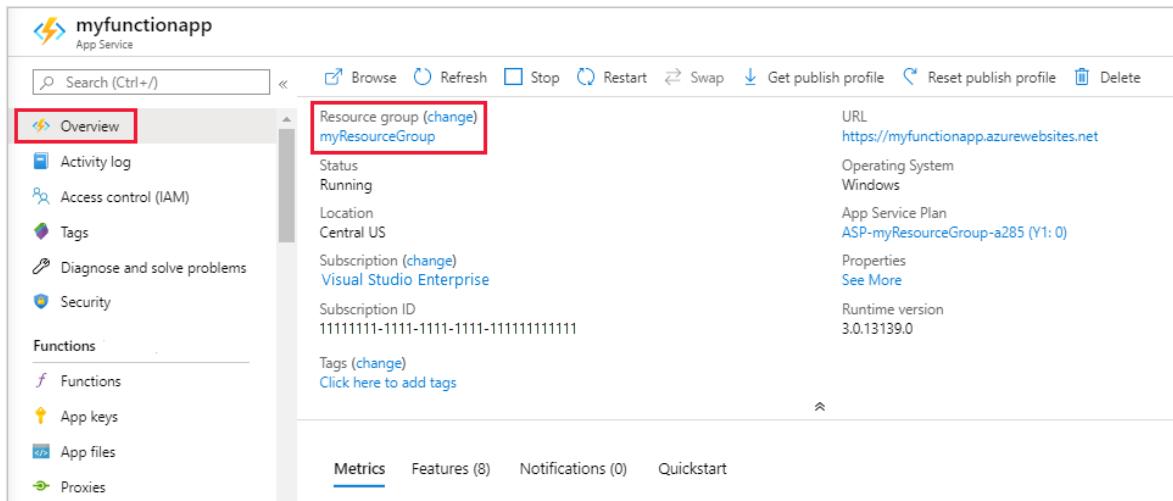
Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.



The screenshot shows the Azure portal interface for a function app named 'myfunctionapp'. The left sidebar has a 'Functions' section with options like Functions, App keys, App files, and Proxies. The main content area is titled 'Overview' and shows resource details. A red box highlights the 'Resource group (change)' section, which displays 'myResourceGroup'. To the right, detailed information is provided: URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Subscription (Visual Studio Enterprise), Subscription ID (1111111-1111-1111-1111-111111111111), and Runtime version (3.0.13139.0). Below this, there's a 'Tags (change)' section with a link to 'Click here to add tags'. At the bottom of the main content area, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. To learn more about developing Functions, see [Develop Azure Functions using Visual Studio](#).

Next, you should enable Application Insights monitoring for your function app:

[Enable Application Insights integration](#)

Connect your Java function to Azure Storage

Article • 03/08/2022

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to integrate the function you created in the [previous quickstart article](#) with an Azure Storage queue. The output binding that you add to this function writes data from an HTTP request to a message in the queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make this connection easier, you use the Storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Prerequisites

Before you start this article, complete the steps in [part 1 of the Java quickstart](#).

Download the function app settings

You've already created a function app in Azure, along with the required Storage account. The connection string for this account is stored securely in app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your Storage account when running the function locally, you must download app settings to the `local.settings.json` file.

From the root of the project, run the following Azure Functions Core Tools command to download settings to `local.settings.json`, replacing `<APP_NAME>` with the name of your function app from the previous article:

Bash

```
func azure functionapp fetch-app-settings <APP_NAME>
```

You might need to sign in to your Azure account.

ⓘ Important

This command overwrites any existing settings with values from your function app in Azure.

Because it contains secrets, the local.settings.json file never gets published, and it should be excluded from source control.

You need the value `AzureWebJobsStorage`, which is the Storage account connection string. You use this connection to verify that the output binding works as expected.

Enable extension bundles

The easiest way to install binding extensions is to enable [extension bundles](#). When you enable bundles, a predefined set of extension packages is automatically installed.

To enable extension bundles, open the host.json file and update its contents to match the following code:

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[3.*, 4.0.0)"  
    }  
}
```

You can now add the Storage output binding to your project.

Add an output binding

In a Java project, the bindings are defined as binding annotations on the function method. The `function.json` file is then autogenerated based on these annotations.

Browse to the location of your function code under `src/main/java`, open the `Function.java` project file, and add the following parameter to the `run` method definition:

Java

```
@QueueOutput(name = "msg", queueName = "outqueue", connection =  
"AzureWebJobsStorage") OutputBinding<String> msg
```

The `msg` parameter is an `OutputBinding<T>` type, which represents a collection of strings. These strings are written as messages to an output binding when the function completes. In this case, the output is a storage queue named `outqueue`. The connection string for the Storage account is set by the `connection` method. You pass the application setting that contains the Storage account connection string, rather than passing the connection string itself.

The `run` method definition must now look like the following example:

Java

```
@FunctionName("HttpTrigger-Java")  
public HttpResponseMessage run(  
    @HttpTrigger(name = "req", methods = {HttpMethod.GET,  
    HttpMethod.POST}, authLevel = AuthorizationLevel.FUNCTION)  
    HttpRequestMessage<Optional<String>> request,  
    @QueueOutput(name = "msg", queueName = "outqueue", connection =  
    "AzureWebJobsStorage")  
    OutputBinding<String> msg, final ExecutionContext context) {  
    ...  
}
```

Add code that uses the output binding

Now, you can use the new `msg` parameter to write to the output binding from your function code. Add the following line of code before the success response to add the value of `name` to the `msg` output binding.

Java

```
msg.setValue(name);
```

When you use an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Your `run` method must now look like the following example:

Java

```

public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET,
    HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @QueueOutput(name = "msg", queueName = "outqueue",
    connection = "AzureWebJobsStorage") OutputBinding<String> msg,
    final ExecutionContext context) {
    context.getLogger().info("Java HTTP trigger processed a request.");

    // Parse query parameter
    String query = request.getQueryParameters().get("name");
    String name = request.getBody().orElse(query);

    if (name == null) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Please pass a name on the query string or in the request
body").build();
    } else {
        // Write the name to the message queue.
        msg.setValue(name);

        return request.createResponseBuilder(HttpStatus.OK).body("Hello, " +
name).build();
    }
}

```

Update the tests

Because the archetype also creates a set of tests, you need to update these tests to handle the new `msg` parameter in the `run` method signature.

Browse to the location of your test code under `src/test/java`, open the `Function.java` project file, and replace the line of code under `//Invoke` with the following code:

Java

```

@SuppressWarnings("unchecked")
final OutputBinding<String> msg =
(OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);

```

You're now ready to try out the new output binding locally.

Run the function locally

As before, use the following command to build the project and start the Functions runtime locally:

Maven

Bash

```
mvn clean package  
mvn azure-functions:run
```

ⓘ Note

Because you enabled extension bundles in the host.json, the **Storage binding extension** was downloaded and installed for you during startup, along with the other Microsoft binding extensions.

As before, trigger the function from the command line using cURL in a new terminal window:

CMD

```
curl -w "\n" http://localhost:7071/api/HttpTrigger-Java --data  
AzureFunctions
```

This time, the output binding also creates a queue named `outqueue` in your Storage account and adds a message with this same string.

Next, you use the Azure CLI to view the new queue and verify that a message was added. You can also view your queue by using the [Microsoft Azure Storage Explorer](#) or in the [Azure portal](#).

Set the Storage account connection

Open the local.settings.json file and copy the value of `AzureWebJobsStorage`, which is the Storage account connection string. Set the `AZURE_STORAGE_CONNECTION_STRING` environment variable to the connection string by using this Bash command:

Bash

```
AZURE_STORAGE_CONNECTION_STRING=<STORAGE_CONNECTION_STRING>
```

When you set the connection string in the `AZURE_STORAGE_CONNECTION_STRING` environment variable, you can access your Storage account without having to provide authentication each time.

Query the Storage queue

You can use the [az storage queue list](#) command to view the Storage queues in your account, as in the following example:

Azure CLI

```
az storage queue list --output tsv
```

The output from this command includes a queue named `outqueue`, which is the queue that was created when the function ran.

Next, use the [az storage message peek](#) command to view the messages in this queue, as in this example:

Azure CLI

```
echo `echo $(az storage message peek --queue-name outqueue -o tsv --query
'[].{Message:content}')` | base64 --decode`
```

The string returned should be the same as the message you sent to test the function.

ⓘ Note

The previous example decodes the returned string from base64. This is because the Queue storage bindings write to and read from Azure Storage as **base64 strings**.

Redeploy the project

To update your published app, run the following command again:

Maven

Bash

```
mvn azure-functions:deploy
```

Again, you can use cURL to test the deployed function. As before, pass the value `AzureFunctions` in the body of the POST request to the URL, as in this example:

Bash

```
curl -w "\n" https://fabrikam-functions-  
20190929094703749.azurewebsites.net/api/HttpTrigger-Java?  
code=zYRohsTwB1Z68YF.... --data AzureFunctions
```

You can [examine the Storage queue message](#) again to verify that the output binding generates a new message in the queue, as expected.

Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to continue on with subsequent quickstarts or with the tutorials, don't clean up the resources created in this quickstart. If you don't plan to continue, use the following command to delete all resources created in this quickstart:

Azure CLI

```
az group delete --name myResourceGroup
```

Select `y` when prompted.

Next steps

You've updated your HTTP-triggered function to write data to a Storage queue. To learn more about developing Azure Functions with Java, see the [Azure Functions Java developer guide](#) and [Azure Functions triggers and bindings](#). For examples of complete Function projects in Java, see the [Java Functions samples](#).

Next, you should enable Application Insights monitoring for your function app:

[Enable Application Insights integration](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

When using the [Python v2 programming model](#), binding attributes are defined directly in the `function_app.py` file as decorators. From the previous quickstart, your `function_app.py` file already contains one decorator-based binding:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="hello", auth_level=func.AuthLevel.ANONYMOUS)
```

The `route` decorator adds `HttpTrigger` and `HttpOutput` binding to the function, which enables your function be triggered when http requests hit the specified route.

To write to an Azure Storage queue from this function, add the `queue_output` decorator to your function code:

Python

```
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
```

In the decorator, `arg_name` identifies the binding parameter referenced in your code, `queue_name` is name of the queue that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Storage account. In quickstarts you use the same storage account as the function app, which is in the `AzureWebJobsStorage` setting (from `local.settings.json` file). When the `queue_name` doesn't exist, the binding creates it on first use.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Update `HttpExample\function_app.py` to match the following code, add the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="HttpExample")
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
def HttpExample(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) ->
func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        msg.set(name)
        return func.HttpResponse(f"Hello, {name}. This HTTP triggered
function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
            status_code=200
        )
```

The `msg` parameter is an instance of the `azure.functions.Out class`. The `set` method writes a string message to the queue. In this case, it's the `name` passed to the function in the URL query string.

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

```
Console  
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools  
Core Tools Version: 4.0.5049 Commit hash: N/A (64-bit)  
Function Runtime Version: 4.15.2.20177  
  
[2023-03-17T03:27:12.372Z] Worker process started and initialized.  
  
Functions:  
  
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

⚠ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

💡 Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[1.*, 2.0.0)"  
    }  
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in *host.json*.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's *local.setting.json* file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

bash

Bash

```
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>"
```

2. (Optional) Use the [az storage queue list](#) command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

Azure CLI

```
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

bash

Azure CLI

```
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}')` | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the `LocalFunctionsProj` folder, use the `func azure functionapp publish` command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`.

The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)

Debug PowerShell Azure Functions locally

Article • 11/05/2020

Azure Functions lets you develop your functions as PowerShell scripts.

You can debug your PowerShell functions locally as you would any PowerShell scripts using the following standard development tools:

- [Visual Studio Code](#): Microsoft's free, lightweight, and open-source text editor with the PowerShell extension that offers a full PowerShell development experience.
- A PowerShell console: Debug using the same commands you would use to debug any other PowerShell process.

[Azure Functions Core Tools](#) supports local debugging of Azure Functions, including PowerShell functions.

Example function app

The function app used in this article has a single HTTP triggered function and has the following files:

```
PSFunctionApp
| - HttpTriggerFunction
| | - run.ps1
| | - function.json
| - local.settings.json
| - host.json
| - profile.ps1
```

This function app is similar to the one you get when you complete the [PowerShell quickstart](#).

The function code in `run.ps1` looks like the following script:

```
PowerShell

param($Request)

$name = $Request.Query.Name
```

```
if($name) {
    $status = 200
    $body = "Hello $name"
}
else {
    $status = 400
    $body = "Please pass a name on the query string or in the request body."
}

Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = $status
    Body = $body
})
```

Set the attach point

To debug any PowerShell function, the function needs to stop for the debugger to be attached. The `Wait-Debugger` cmdlet stops execution and waits for the debugger.

ⓘ Note

When using PowerShell 7, you don't need to add the `Wait-Debugger` call in your code.

All you need to do is add a call to the `Wait-Debugger` cmdlet just above the `if` statement, as follows:

PowerShell

```
param($Request)

$name = $Request.Query.Name

# This is where we will wait for the debugger to attach
Wait-Debugger

if($name) {
    $status = 200
    $body = "Hello $name"
}
# ...
```

Debugging starts at the `if` statement.

With `Wait-Debugger` in place, you can now debug the functions using either Visual Studio Code or a PowerShell console.

Debug in Visual Studio Code

To debug your PowerShell functions in Visual Studio Code, you must have the following installed:

- [PowerShell extension for Visual Studio Code](#)
- [Azure Functions extension for Visual Studio Code](#)
- [PowerShell Core 6.2 or higher](#)

After installing these dependencies, load an existing PowerShell Functions project, or [create your first PowerShell Functions project](#).

ⓘ Note

Should your project not have the needed configuration files, you are prompted to add them.

Set the PowerShell version

PowerShell Core installs side by side with Windows PowerShell. Set PowerShell Core as the PowerShell version to use with the PowerShell extension for Visual Studio Code.

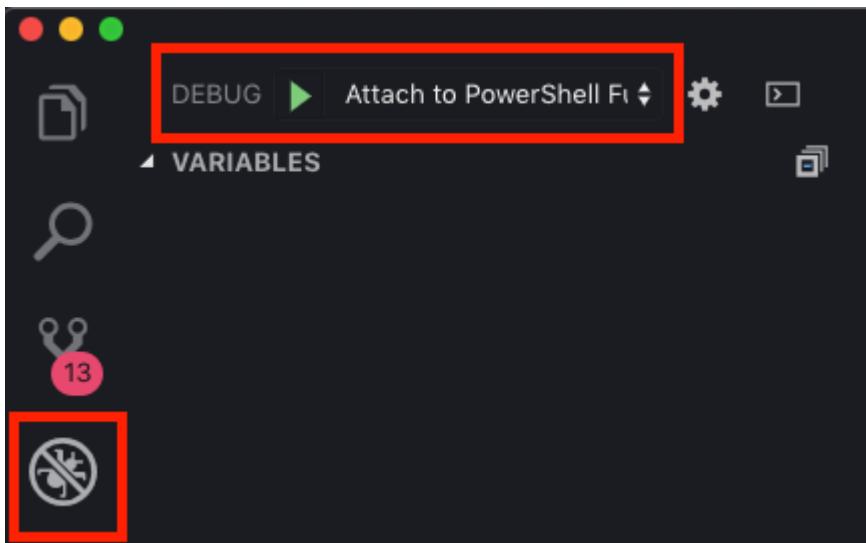
1. Press F1 to display the command pallet, then search for `Session`.
2. Choose **PowerShell: Show Session Menu**.
3. If your **Current session** isn't **PowerShell Core 6**, choose **Switch to: PowerShell Core 6**.

When you have a PowerShell file open, you see the version displayed in green at the bottom right of the window. Selecting this text also displays the session menu. To learn more, see the [Choosing a version of PowerShell to use with the extension](#).

Start the function app

Verify that `Wait-Debugger` is set in the function where you want to attach the debugger. With `Wait-Debugger` added, you can debug your function app using Visual Studio Code.

Choose the **Debug** pane and then **Attach to PowerShell function**.



You can also press the F5 key to start debugging.

The start debugging operation does the following tasks:

- Runs `func extensions install` in the terminal to install any Azure Functions extensions required by your function app.
- Runs `func host start` in the terminal to start the function app in the Functions host.
- Attach the PowerShell debugger to the PowerShell runspace within the Functions runtime.

Note

You need to ensure `PSWorkerInProcConcurrencyUpperBound` is set to 1 to ensure correct debugging experience in Visual Studio Code. This is the default.

With your function app running, you need a separate PowerShell console to call the HTTP triggered function.

In this case, the PowerShell console is the client. The `Invoke-RestMethod` is used to trigger the function.

In a PowerShell console, run the following command:

```
PowerShell
```

```
Invoke-RestMethod "http://localhost:7071/api/HttpTrigger?Name=Functions"
```

You'll notice that a response isn't immediately returned. That's because `Wait-Debugger` has attached the debugger and PowerShell execution went into break mode as soon as

it could. This is because of the [BreakAll concept](#), which is explained later. After you press the `continue` button, the debugger now breaks on the line right after `Wait-Debugger`.

At this point, the debugger is attached and you can do all the normal debugger operations. For more information on using the debugger in Visual Studio Code, see [the official documentation ↗](#).

After you continue and fully invoke your script, you'll notice that:

- The PowerShell console that did the `Invoke-RestMethod` has returned a result
- The PowerShell Integrated Console in Visual Studio Code is waiting for a script to be executed

Later when you invoke the same function, the debugger in PowerShell extension breaks right after the `Wait-Debugger`.

Debugging in a PowerShell Console

ⓘ Note

This section assumes you have read the [Azure Functions Core Tools docs](#) and know how to use the `func host start` command to start your function app.

Open up a console, `cd` into the directory of your function app, and run the following command:

```
sh  
func host start
```

With the function app running and the `Wait-Debugger` in place, you can attach to the process. You do need two more PowerShell consoles.

One of the consoles acts as the client. From this, you call `Invoke-RestMethod` to trigger the function. For example, you can run the following command:

```
PowerShell  
Invoke-RestMethod "http://localhost:7071/api/HttpTrigger?Name=Functions"
```

You'll notice that it doesn't return a response, which is a result of the `Wait-Debugger`. The PowerShell runspace is now waiting for a debugger to be attached. Let's get that

attached.

In the other PowerShell console, run the following command:

```
PowerShell  
Get-PSHostProcessInfo
```

This cmdlet returns a table that looks like the following output:

```
Output  
  
ProcessName ProcessId AppDomainName  
-----  
dotnet      49988  None  
pwsh        43796  None  
pwsh        49970  None  
pwsh        3533   None  
pwsh        79544  None  
pwsh        34881  None  
pwsh        32071  None  
pwsh        88785  None
```

Make note of the `ProcessId` for the item in the table with the `ProcessName` as `dotnet`. This process is your function app.

Next, run the following snippet:

```
PowerShell  
  
# This enters into the Azure Functions PowerShell process.  
# Put your value of `ProcessId` here.  
Enter-PSHostProcess -Id $ProcessId  
  
# This triggers the debugger.  
Debug-Runspace 1
```

Once started, the debugger breaks and shows something like the following output:

```
Debugging Runspace: Runspace1  
  
To end the debugging session type the 'Detach' command at the debugger  
prompt, or type 'Ctrl+C' otherwise.  
  
At /Path/To/PSFunctionApp/HttpTriggerFunction/run.ps1:13 char:1  
+ if($name) { ...
```

```
+ ~~~~~~  
[DBG]: [Process:49988]: [Runspace1]: PS /Path/To/PSFunctionApp>>
```

At this point, you're stopped at a breakpoint in the [PowerShell debugger](#). From here, you can do all of the usual debug operations, step over, step into, continue, quit, and others. To see the complete set of debug commands available in the console, run the `h` or `?` commands.

You can also set breakpoints at this level with the `Set-PSBreakpoint` cmdlet.

Once you continue and fully invoke your script, you'll notice that:

- The PowerShell console where you executed `Invoke-RestMethod` has now returned a result.
- The PowerShell console where you executed `Debug-Runspace` is waiting for a script to be executed.

You can invoke the same function again (using `Invoke-RestMethod` for example) and the debugger breaks in right after the `Wait-Debugger` command.

Considerations for debugging

Keep in mind the following issues when debugging your Functions code.

BreakAll might cause your debugger to break in an unexpected place

The PowerShell extension uses `Debug-Runspace`, which in turn relies on PowerShell's `BreakAll` feature. This feature tells PowerShell to stop at the first command that is executed. This behavior gives you the opportunity to set breakpoints within the debugged runspace.

The Azure Functions runtime runs a few commands before actually invoking your `run.ps1` script, so it's possible that the debugger ends up breaking within the `Microsoft.Azure.Functions.PowerShellWorker.psm1` or `Microsoft.Azure.Functions.PowerShellWorker.psd1`.

Should this break happen, run the `continue` or `c` command to skip over this breakpoint. You then stop at the expected breakpoint.

Troubleshooting

If you have difficulties during debugging, you should check for the following:

Check	Action
Run <code>func --version</code> from the terminal. If you get an error that <code>func</code> can't be found, Core Tools (<code>func.exe</code>) may be missing from the local <code>path</code> variable.	Reinstall Core Tools .
In Visual Studio Code, the default terminal needs to have access to <code>func.exe</code> . Make sure you aren't using a default terminal that doesn't have Core Tools installed, such as Windows Subsystem for Linux (WSL).	Set the default shell in Visual Studio Code to either PowerShell 7 (recommended) or Windows PowerShell 5.1.

Next steps

To learn more about developing Functions using PowerShell, see [Azure Functions PowerShell developer guide](#).

Tutorial: Trigger Azure Functions on blob containers using an event subscription

Article • 05/21/2024

Previous versions of the Azure Functions Blob Storage trigger poll your storage container for changes. More recent version of the Blob Storage extension (5.x+) instead use an Event Grid event subscription on the container. This event subscription reduces latency by triggering your function instantly as changes occur in the subscribed container.

This article shows how to create a function that runs based on events raised when a blob is added to a container. You use Visual Studio Code for local development and to validate your code before deploying your project to Azure.

- ✓ Create an event-based Blob Storage triggered function in a new project.
- ✓ Validate locally within Visual Studio Code using the Azurite emulator.
- ✓ Create a blob storage container in a new storage account in Azure.
- ✓ Create a function app in the Flex Consumption plan (preview).
- ✓ Create an event subscription to the new blob container.
- ✓ Deploy and validate your function code in Azure.

This article creates a C# app that runs in isolated worker mode, which supports .NET 8.0.

ⓘ Important

This tutorial has you use the [Flex Consumption plan](#), which is currently in preview. The Flex Consumption plan only supports the event-based version of the Blob Storage trigger. You can complete this tutorial using any other [hosting plan](#) for your function app.

Prerequisites

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [.NET 8.0 SDK ↗](#).
- [Visual Studio Code ↗](#) on one of the [supported platforms ↗](#).

- [C# extension](#) for Visual Studio Code.
- [Azure Functions extension](#) for Visual Studio Code.
- [Azure Storage extension](#) for Visual Studio Code.

ⓘ Note

The Azure Storage extension for Visual Studio Code is currently in preview.

Create a Blob triggered function

When you create a Blob Storage trigger function using Visual Studio Code, you also create a new project. You need to edit the function to consume an event subscription as the source, rather than use the regular polled container.

1. In Visual Studio Code, open your function app.
2. Press F1 to open the command palette, enter `Azure Functions: Create Function...`, and select `Create new project`.
3. For your project workspace, select the directory location. Make sure that you either create a new folder or choose an empty folder for the project workspace.
Don't choose a project folder that's already part of a workspace.
4. At the prompts, provide the following information:

 Expand table

Prompt	Action
Select a language	Select <code>C#</code> .
Select a .NET runtime	Select <code>.NET 8.0 Isolated LTS</code> .
Select a template for your project's first function	Select <code>Azure Blob Storage trigger (using Event Grid)</code> .
Provide a function name	Enter <code>BlobTriggerEventGrid</code> .
Provide a namespace	Enter <code>My.Functions</code> .
Select setting from "local.settings.json"	Select <code>create new local app setting</code> .
Select subscription	Select your subscription.

Prompt	Action
Select a storage account	Use Azurite emulator for local storage.
This is the path within your storage account that the trigger will monitor	Accept the default value <code>samples-workitems</code> .
Select how you would like to open your project	Select <code>Open in current window</code> .

Upgrade the Storage extension

To use the Event Grid-based Blob Storage trigger, you must have at least version 5.x of the Azure Functions Storage extension.

To upgrade your project with the required extension version, in the Terminal window, run this [dotnet add package](#) command:

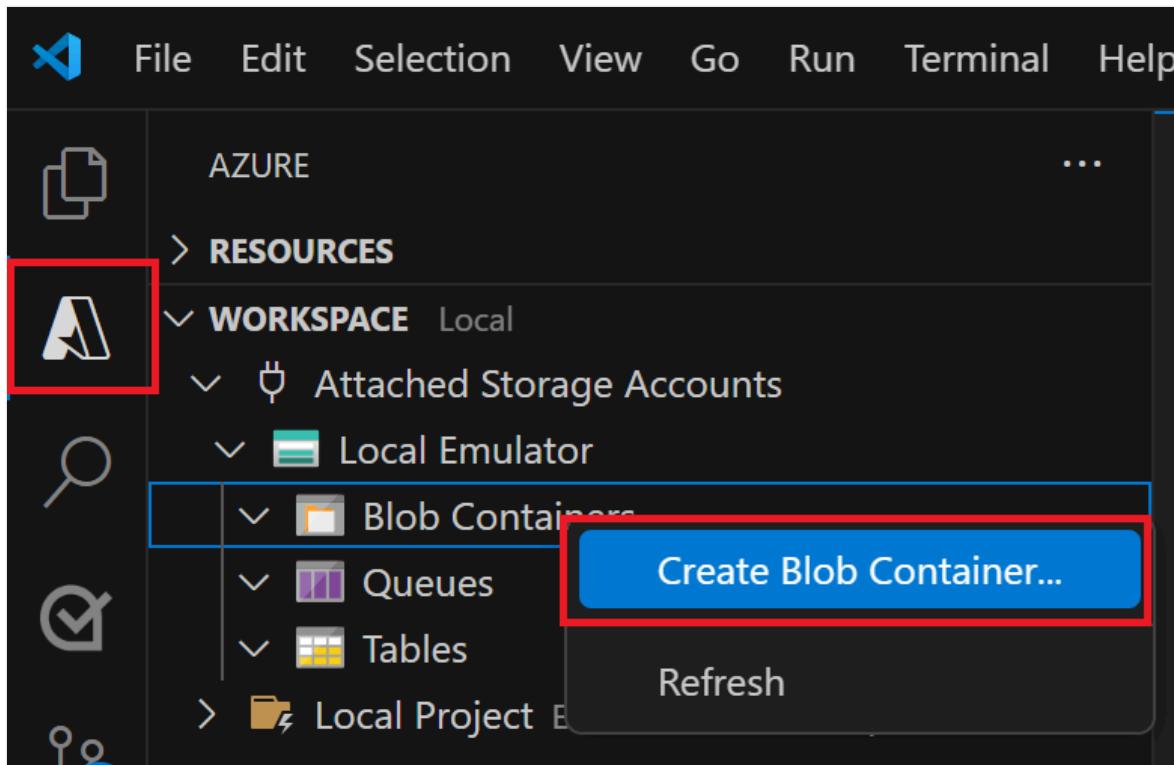
Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs
```

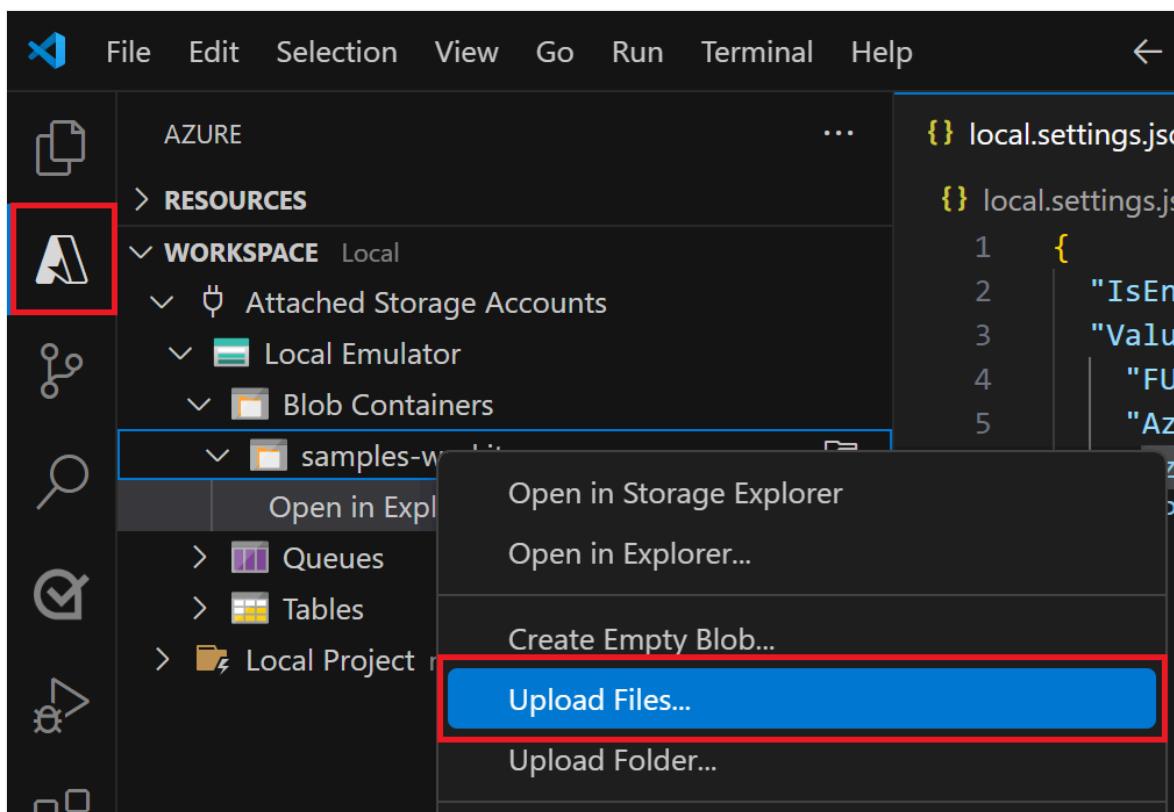
Prepare local storage emulation

Visual Studio Code uses Azurite to emulate Azure Storage services when running locally. You use Azurite to emulate the Azure Blob Storage service during local development and testing.

1. If haven't already done so, install the [Azurite v3 extension for Visual Studio Code](#).
2. Verify that the `local.settings.json` file has `"UseDevelopmentStorage=true"` set for `AzureWebJobsStorage`, which tells Core Tools to use Azurite instead of a real storage account connection when running locally.
3. Press F1 to open the command palette, type `Azurite: Start Blob Service`, and press enter, which starts the Azurite Blob Storage service emulator.
4. Select the Azure icon in the Activity bar, expand **Workspace > Attached Storage Accounts > Local Emulator**, right-click **Blob Containers**, select **Create Blob Container...**, enter the name `samples-workitems`, and press Enter.



5. Expand **Blob Containers** > **samples-workitems** and select **Upload files....**

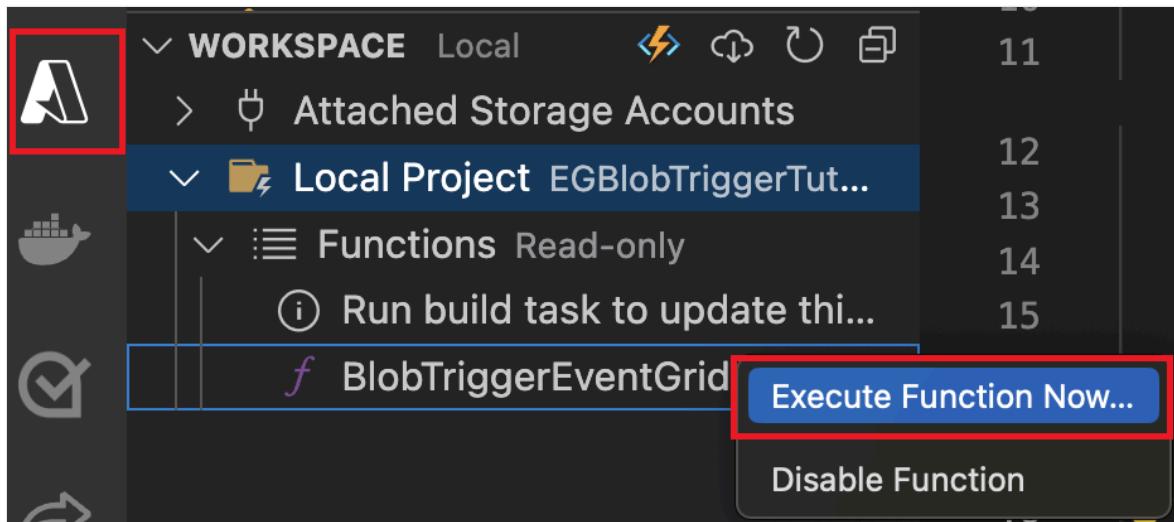


6. Choose a file to upload to the locally emulated container. This file gets processed later by your function to verify and debug your function code. A text file might work best with the Blob trigger template code.

Run the function locally

With a file in emulated storage, you can run your function to simulate an event raised by an Event Grid subscription. The event info passed to your trigger depends on the file you added to the local container.

1. Set any breakpoints and press F5 to start your project for local debugging. Azure Functions Core Tools should be running in your Terminal window.
2. Back in the Azure area, expand **Workspace > Local Project > Functions**, right-click the function, and select **Execute Function Now....**



3. In the request body dialog, type `samples-workitems/<TEST_FILE_NAME>`, replacing `<TEST_FILE_NAME>` with the name of the file you uploaded in the local storage emulator.
4. Press Enter to run the function. The value you provided is the path to your blob in the local emulator. This string gets passed to your trigger in the request payload, which simulates the payload when an event subscription calls your function to report a blob being added to the container.
5. Review the output of this function execution. You should see in the output the name of the file and its contents logged. If you set any breakpoints, you might need to continue the execution.

Now that you've successfully validated your function code locally, it's time to publish the project to a new function app in Azure.

Prepare the Azure Storage account

Event subscriptions to Azure Storage require a general-purpose v2 storage account. You can use the Azure Storage extension for Visual Studio Code to create this storage account.

1. In Visual Studio Code, press F1 again to open the command palette and enter `Azure Storage: Create Storage Account...`. Provide this information when prompted:

[+] Expand table

Prompt	Action
Enter the name of the new storage account	Provide a globally unique name. Storage account names must have 3 to 24 characters in length with only lowercase letters and numbers. For easier identification, we use the same name for the resource group and the function app name.
Select a location for new resources	For better performance, choose a region near you ↗ .

The extension creates a general-purpose v2 storage account with the name you provided. The same name is also used for the resource group that contains the storage account. The Event Grid-based Blob Storage trigger requires a general-purpose v2 storage account.

2. Press F1 again and in the command palette enter `Azure Storage: Create Blob Container...`. Provide this information when prompted:

[+] Expand table

Prompt	Action
Select a resource	Select the general-purpose v2 storage account that you created.
Enter a name for the new blob container	Enter <code>samples-workitems</code> , which is the container name referenced in your code project.

Your function app also needs a storage account to run. For simplicity, this tutorial uses the same storage account for your blob trigger and your function app. However, in production, you might want to use a separate storage account with your function app. For more information, see [Storage considerations for Azure Functions](#).

Create the function app

Use these steps to create a function app in the Flex Consumption plan. When your app is hosted in a Flex Consumption plan, Blob Storage triggers must use event subscriptions.

1. In the command pallet, enter **Azure Functions: Create function app in Azure... (Advanced)**.

2. Following the prompts, provide this information:

 Expand table

Prompt	Selection
Enter a globally unique name for the new function app.	Type a globally unique name that identifies your new function app and then select Enter. Valid characters for a function app name are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Select a hosting plan.	Choose Flex Consumption (Preview) .
Select a runtime stack.	Choose the language stack and version on which you've been running locally.
Select a resource group for new resources.	Choose the existing resource group in which you created the storage account.
Select a location for new resources.	Select a location in a supported region near you or near other services that your functions access. Unsupported regions aren't displayed. For more information, see View currently supported regions .
Select a storage account.	Choose the name of the storage account you created.
Select an Application Insights resource for your app.	Choose Create new Application Insights resource and at the prompt provide the name for the instance used to store runtime data from your functions.

A notification appears after your function app is created. Select **View Output** in this notification to view the creation results, including the Azure resources that you created.

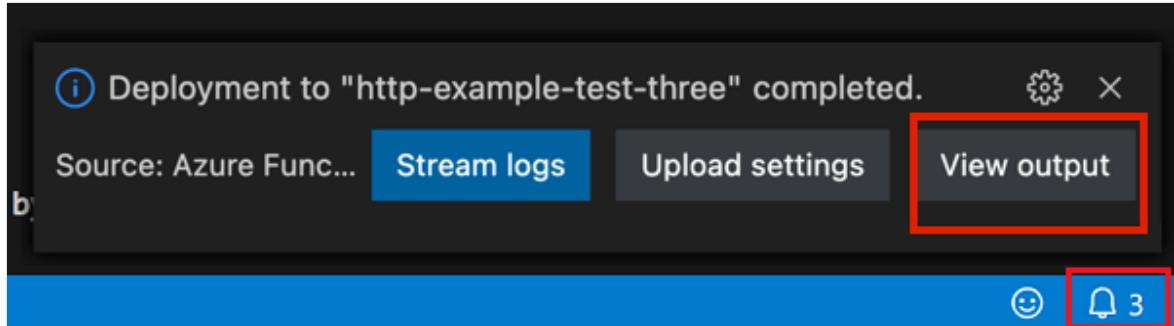
Deploy your function code

Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the **Resources** area of the Azure activity, locate the function app resource you just created, right-click the resource, and select **Deploy to function app....**

- When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
- After deployment completes, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower right corner to see it again.



Update application settings

Because required application settings from the `local.settings.json` file aren't automatically published, you must upload them to your function app so that your function runs correctly in Azure.

- In the command pallet, enter `Azure Functions: Download Remote Settings...`, and in the **Select a resource** prompt choose the name of your function app.
- When prompted that the `AzureWebJobsStorage` setting already exists, select **Yes** to overwrite the local emulator setting with the actual storage account connection string from Azure.
- In the `local.settings.json` file, replace the local emulator setting with same connection string used for `AzureWebJobsStorage`.
- Remove the `FUNCTIONS_WORKER_RUNTIME` entry, which isn't supported in a Flex Consumption plan.
- In the command pallet, enter `Azure Functions: Upload Local Settings...`, and in the **Select a resource** prompt choose the name of your function app.

Now both the Functions host and the trigger are sharing the same storage account.

Build the endpoint URL

To create an event subscription, you need to provide Event Grid with the URL of the specific endpoint to report Blob Storage events. This *blob extension* URL is composed of

these parts:

[+] Expand table

Part	Example
Base function app URL	<code>https://<FUNCTION_APP_NAME>.azurewebsites.net</code>
Blob-specific path	<code>/runtime/webhooks/blobs</code>
Function query string	<code>?functionName=Host.Functions.BlobTriggerEventGrid</code>
Blob extension access key	<code>&code=<BLOB_EXTENSION_KEY></code>

The blob extension access key is designed to make it more difficult for others to access your blob extension endpoint. To determine your blob extension access key:

1. In Visual Studio Code, choose the Azure icon in the Activity bar. In **Resources**, expand your subscription, expand **Function App**, right-click the function app you created, and select **Open in portal**.
2. Under **Functions** in the left menu, select **App keys**.
3. Under **System keys** select the key named **blobs_extension**, and copy the key **Value**. You include this value in the query string of new endpoint URL.
4. Create a new endpoint URL for the Blob Storage trigger based on the following example:

```
HTTP  
  
https://<FUNCTION_APP_NAME>.azurewebsites.net/runtime/webhooks/blobs?  
functionName=Host.Functions.BlobTriggerEventGrid&code=<BLOB_EXTENSION_KEY>
```

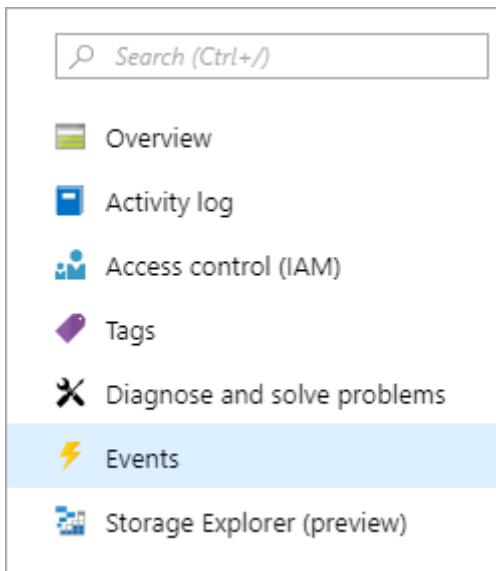
In this example, replace `<FUNCTION_APP_NAME>` with the name of your function app and replace `<BLOB_EXTENSION_KEY>` with the value you got from the portal. If you used a different name for your function, you'll also need to change the `functionName` query string value to your function name.

You can now use this endpoint URL to create an event subscription.

Create the event subscription

An event subscription, powered by Azure Event Grid, raises events based on changes in the subscribed blob container. This event is then sent to the blob extension endpoint for your function. After you create an event subscription, you can't update the endpoint URL.

1. In Visual Studio Code, choose the Azure icon in the Activity bar. In **Resources**, expand your subscription, expand **Storage accounts**, right-click the storage account you created earlier, and select **Open in portal**.
2. Sign in to the [Azure portal](#) and make a note of the **Resource group** for your storage account. You create your other resources in the same group to make it easier to clean up resources when you're done.
3. select the **Events** option from the left menu.



4. In the **Events** window, select the **+ Event Subscription** button, and provide values from the following table into the **Basic** tab:

[Expand table](#)

Setting	Suggested value	Description
Name	<code>myBlobEventSub</code>	Name that identifies the event subscription. You can use the name to quickly find the event subscription.
Event Schema	Event Grid Schema	Use the default schema for events.
System Topic Name	<code>samples-workitems-blobs</code>	Name for the topic, which represents the container. The topic is created with the first subscription, and you'll use it for future event subscriptions.

Setting	Suggested value	Description
Filter to Event Types	Blob Created	
Endpoint Type	Web Hook	The blob storage trigger uses a web hook endpoint.
Endpoint	Your Azure-based URL endpoint	Use the URL endpoint that you built, which includes the key value.

5. Select **Confirm selection** to validate the endpoint URL.

6. Select **Create** to create the event subscription.

Upload a file to the container

You can upload a file from your computer to your blob storage container using Visual Studio Code.

1. In Visual Studio Code, press F1 to open the command palette and type `Azure Storage: Upload Files....`.
2. In the **Open** dialog box, choose a file, preferably a text file, and select **Upload**.
3. Provide the following information at the prompts:

[\[+\] Expand table](#)

Setting	Suggested value	Description
Enter the destination directory of this upload	default	Just accept the default value of <code>/</code> , which is the container root.
Select a resource	Storage account name	Choose the name of the storage account you created in a previous step.
Select a resource type	Blob Containers	You're uploading to a blob container.
Select Blob Container	samples-workitems	This value is the name of the container you created in a previous step.

Browse your local file system to find a file to upload and then select the **Upload** button to upload the file.

Verify the function in Azure

Now that you uploaded a file to the `samples-workitems` container, the function should be triggered. You can verify by checking the following on the Azure portal:

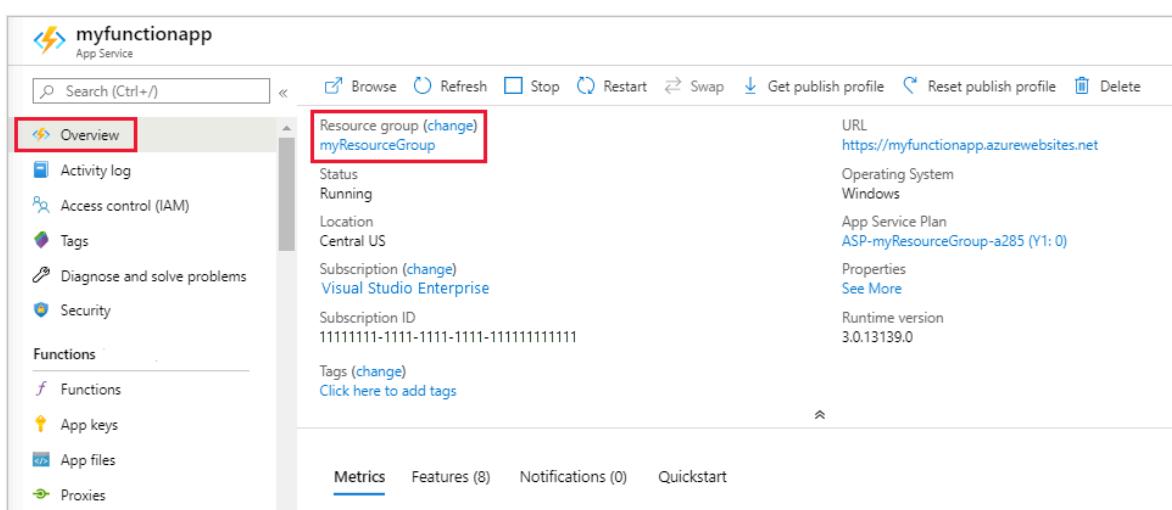
1. In your storage account, go to the **Events** page, select **Event Subscriptions**, and you should see that an event was delivered. There might be up a five-minute delay for the event to show up on the chart.
2. Back in your function app page in the portal, under **Functions** find your function and select **Invocations and more**. You should see traces written from your successful function execution.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.

5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

- [Working with blobs](#)
- [Automate resizing uploaded images using Event Grid](#)
- [Event Grid trigger for Azure Functions](#)

Create a load test for Azure Functions

Article • 05/13/2024

Learn how to create a load test for an app in Azure Functions with Azure Load Testing. In this article, you'll learn how to create a URL-based load test for your function app in the Azure portal, and then use the load testing dashboard to analyze performance issues and identify bottlenecks.

With the integrated load testing experience in Azure Functions, you can:

- Create a [URL-based load test](#) for functions with an HTTP trigger
- View the load test runs associated with a function app
- Create a load testing resource

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- A function app with at least one function with an HTTP trigger. If you need to create a function app, see [Getting started with Azure Functions](#).

Create a load test for a function app

You can create a URL-based load test directly from your Azure Function App in the Azure portal.

To create a load test for a function app:

1. In the [Azure portal](#), go to your function app.
2. On the left pane, select **Load Testing (Preview)** under the **Performance** section.

On this page, you can see the list of tests and the load test runs for this function app.

3. Optionally, select **Create load testing resource** if you don't have a load testing resource yet.
4. Select **Create test** to start creating a URL-based load test for the function app.
5. On the **Create test** page, first enter the test details:

[] [Expand table](#)

Field	Description
Load Testing Resource	Select your load testing resource.
Test name	Enter a unique test name.
Test description	(Optional) Enter a load test description.
Run test after creation	When selected, the load test starts automatically after creating the test.

6. Select **Add request** to add HTTP requests to the load test:

On the **Add request** page, enter the details for the request:

[] [Expand table](#)

Field	Description
Request name	Unique name within the load test to identify the request. You can use this request name when defining test criteria .
Function name	Select the function that you want to test

Field	Description
Key	Select the key required for accessing the function
HTTP method	Select an HTTP method from the list. Azure Load Testing supports GET, POST, PUT, DELETE, PATCH, HEAD, and OPTIONS.
Query parameters	(Optional) Enter query string parameters to append to the URL.
Headers	(Optional) Enter HTTP headers to include in the HTTP request.
Body	(Optional) Depending on the HTTP method, you can specify the HTTP body content. Azure Load Testing supports the following formats: raw data, JSON view, JavaScript, HTML, and XML.

Learn more about [adding HTTP requests to a load test](#).

7. Select the **Load configuration** tab to configure the load parameters for the load test.

[Expand table](#)

Field	Description
Engine instances	Enter the number of load test engine instances. The load test runs in parallel across all the engine instances.
Load pattern	Select the load pattern (linear, step, spike) for ramping up to the target number of virtual users.
Concurrent users per engine	Enter the number of <i>virtual users</i> to simulate on each of the test engines. The total number of virtual users for the load test is: #test engines * #users per engine.

Field	Description
Test duration (minutes)	Enter the duration of the load test in minutes.
Ramp-up time (minutes)	Enter the ramp-up time of the load test in minutes. The ramp-up time is the time it takes to reach the target number of virtual users.

8. Optionally, configure the network settings if the function app isn't publicly accessible.

Learn more about [load testing privately hosted endpoints](#).

Home > [octo-host | Load Testing \(Preview\)](#) >

Create test

Resource configuration [Load configuration](#) Review + create

Load configuration
Configure the number of test engines, load pattern and duration for your test. [Learn more](#)

Engine instances *	<input type="range" value="4"/> 4
Load pattern *	Linear
Concurrent users per engine *	250
Test duration (minutes) *	10
Ramp-up time (minutes) *	5

Total Requests

Previous Next [Review + create](#)

9. Select **Review + create** to review the test configuration, and then select **Create** to create the load test.

Azure Load Testing now creates the load test. If you selected **Run test after creation** previously, the load test starts automatically.

(!) Note

If the test was converted from a URL test to a JMX test directly from the Load Testing resource, the test cannot be modified from the function app.

View test runs

You can view the list of test runs and a summary overview of the test results directly from within the function app configuration in the Azure portal.

1. In the [Azure portal](#), go to your Azure Function App.
2. On the left pane, select **Load testing**.
3. In the **Test runs** tab, you can view the list of test runs for your function app.

For each test run, you can view the test details and a summary of the test outcome, such as average response time, throughput, and error state.

4. Select a test run to go to the Azure Load Testing dashboard and analyze the test run details.

Test run name	Test name	Resource name	Run on	Virtual users	Duration (in mi...)	Average respons...	Throughput	Error percentage	Status
TestRun_4/29/2024...	Test_query_func...	cool-new-resource	29/04/2024, 13:19:31	115	1.1	124.00	757.61	100.00	Failed

Next steps

- Learn more about [load testing Azure App Service applications](#).

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Use dependency injection in .NET Azure Functions

Article • 02/16/2024

Azure Functions supports the dependency injection (DI) software design pattern, which is a technique to achieve [Inversion of Control \(IoC\)](#) between classes and their dependencies.

- Dependency injection in Azure Functions is built on the .NET Core Dependency Injection features. Familiarity with [.NET Core dependency injection](#) is recommended. There are differences in how you override dependencies and how configuration values are read with Azure Functions on the Consumption plan.
- Support for dependency injection begins with Azure Functions 2.x.
- Dependency injection patterns differ depending on whether your C# functions run [in-process](#) or [out-of-process](#).

Important

The guidance in this article applies only to [C# class library functions](#), which run in-process with the runtime. This custom dependency injection model doesn't apply to [.NET isolated functions](#), which lets you run .NET functions out-of-process. The .NET isolated worker process model relies on regular ASP.NET Core dependency injection patterns. To learn more, see [Dependency injection](#) in the .NET isolated worker process guide.

Prerequisites

Before you can use dependency injection, you must install the following NuGet packages:

- [Microsoft.Azure.Functions.Extensions](#) ↗
- [Microsoft.NET.Sdk.Functions](#) ↗ package version 1.0.28 or later
- [Microsoft.Extensions.DependencyInjection](#) ↗ (currently, only version 2.x or later supported)

Register services

To register services, create a method to configure and add components to an `IFunctionsHostBuilder` instance. The Azure Functions host creates an instance of `IFunctionsHostBuilder` and passes it directly into your method.

⚠ Warning

For function apps running in the Consumption or Premium plans, modifications to configuration values used in triggers can cause scaling errors. Any changes to these properties by the `FunctionsStartup` class results in a function app startup error.

Injection of `IConfiguration` can lead to unexpected behavior. To learn more about adding configuration sources, see [Customizing configuration sources](#).

To register the method, add the `FunctionsStartup` assembly attribute that specifies the type name used during startup.

C#

```
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection;

[assembly: FunctionsStartup(typeof(MyNamespace.Startup))]

namespace MyNamespace;

public class Startup : FunctionsStartup
{
    public override void Configure(IFunctionsHostBuilder builder)
    {
        builder.Services.AddHttpClient();

        builder.Services.AddSingleton<IMyService>((s) => {
            return new MyService();
        });

        builder.Services.AddSingleton<ILoggerProvider, MyLoggerProvider>();
    }
}
```

This example uses the [Microsoft.Extensions.Http](#) package required to register an `HttpClient` at startup.

Caveats

A series of registration steps run before and after the runtime processes the startup class. Therefore, keep in mind the following items:

- *The startup class is meant for only setup and registration.* Avoid using services registered at startup during the startup process. For instance, don't try to log a message in a logger that is being registered during startup. This point of the registration process is too early for your services to be available for use. After the `Configure` method is run, the Functions runtime continues to register other dependencies, which can affect how your services operate.
- *The dependency injection container only holds explicitly registered types.* The only services available as injectable types are what are set up in the `Configure` method. As a result, Functions-specific types like `BindingContext` and `ExecutionContext` aren't available during setup or as injectable types.
- *Configuring ASP.NET authentication isn't supported.* The Functions host configures ASP.NET authentication services to properly expose APIs for core lifecycle operations. Other configurations in a custom `Startup` class can override this configuration, causing unintended consequences. For example, calling `builder.Services.AddAuthentication()` can break authentication between the portal and the host, leading to messages such as [Azure Functions runtime is unreachable](#).

Use injected dependencies

Constructor injection is used to make your dependencies available in a function. The use of constructor injection requires that you don't use static classes for injected services or for your function classes.

The following sample demonstrates how the `IMyService` and `HttpClient` dependencies are injected into an HTTP-triggered function.

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;
using System.Net.Http;
using System.Threading.Tasks;

namespace MyNamespace;
```

```

public class MyHttpTrigger
{
    private readonly HttpClient _client;
    private readonly IMyService _service;

    public MyHttpTrigger(IHttpClientFactory httpClientFactory, IMyService service)
    {
        this._client = httpClientFactory.CreateClient();
        this._service = service;
    }

    [FunctionName("MyHttpTrigger")]
    public async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route =
null)] HttpRequest req,
        ILogger log)
    {
        var response = await _client.GetAsync("https://microsoft.com");
        var message = _service.GetMessage();

        return new OkObjectResult("Response from function with injected
dependencies.");
    }
}

```

This example uses the [Microsoft.Extensions.Http](#) package required to register an `HttpClient` at startup.

Service lifetimes

Azure Functions apps provide the same service lifetimes as [ASP.NET Dependency Injection](#). For a Functions app, the different service lifetimes behave as follows:

- **Transient**: Transient services are created upon each resolution of the service.
- **Scoped**: The scoped service lifetime matches a function execution lifetime. Scoped services are created once per function execution. Later requests for that service during the execution reuse the existing service instance.
- **Singleton**: The singleton service lifetime matches the host lifetime and is reused across function executions on that instance. Singleton lifetime services are recommended for connections and clients, for example `DocumentClient` or `HttpClient` instances.

View or download a [sample of different service lifetimes](#) on GitHub.

Logging services

If you need your own logging provider, register a custom type as an instance of [ILoggerProvider](#), which is available through the [Microsoft.Extensions.Logging.Abstractions](#) NuGet package.

Application Insights is added by Azure Functions automatically.

⚠️ Warning

- Don't add `AddApplicationInsightsTelemetry()` to the services collection, which registers services that conflict with services provided by the environment.
- Don't register your own `TelemetryConfiguration` or `TelemetryClient` if you are using the built-in Application Insights functionality. If you need to configure your own `TelemetryClient` instance, create one via the injected `TelemetryConfiguration` as shown in [Log custom telemetry in C# functions](#).

ILogger<T> and ILoggerFactory

The host injects `ILogger<T>` and `ILoggerFactory` services into constructors. However, by default these new logging filters are filtered out of the function logs. You need to modify the `host.json` file to opt in to extra filters and categories.

The following example demonstrates how to add an `ILogger<HttpTrigger>` with logs that are exposed to the host.

C#

```
namespace MyNamespace;

public class HttpTrigger
{
    private readonly ILogger<HttpTrigger> _log;

    public HttpTrigger(ILogger<HttpTrigger> log)
    {
        _log = log;
    }

    [FunctionName("HttpTrigger")]
    public async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route =
null)] HttpRequest req)
    {
        _log.LogInformation("C# HTTP trigger function processed a
request.");
    }
}
```

```
// ...
}
```

The following example `host.json` file adds the log filter.

JSON

```
{
    "version": "2.0",
    "logging": {
        "applicationInsights": {
            "samplingSettings": {
                "isEnabled": true,
                "excludedTypes": "Request"
            }
        },
        "logLevel": {
            "MyNamespace.HttpTrigger": "Information"
        }
    }
}
```

For more information about log levels, see [Configure log levels](#).

Function app provided services

The function host registers many services. The following services are safe to take as a dependency in your application:

[+] [Expand table](#)

Service Type	Lifetime	Description
<code>Microsoft.Extensions.Configuration.IConfiguration</code>	Singleton	Runtime configuration
<code>Microsoft.Azure.WebJobs.Host.Executors.IHostIdProvider</code>	Singleton	Responsible for providing the ID of the host instance

If there are other services you want to take a dependency on, [create an issue and propose them on GitHub ↗](#).

Overriding host services

Overriding services provided by the host is currently not supported. If there are services you want to override, [create an issue and propose them on GitHub ↗](#).

Working with options and settings

Values defined in [app settings](#) are available in an `IConfiguration` instance, which allows you to read app settings values in the startup class.

You can extract values from the `IConfiguration` instance into a custom type. Copying the app settings values to a custom type makes it easy test your services by making these values injectable. Settings read into the configuration instance must be simple key/value pairs. For functions running in an Elastic Premium plan, application setting names can only contain letters, numbers (`0-9`), periods (`.`), colons (`:`) and underscores (`_`). For more information, see [App setting considerations](#).

Consider the following class that includes a property named consistent with an app setting:

```
C#  
  
public class MyOptions  
{  
    public string MyCustomSetting { get; set; }  
}
```

And a `local.settings.json` file that might structure the custom setting as follows:

```
JSON  
  
{  
    "IsEncrypted": false,  
    "Values": {  
        "MyOptions:MyCustomSetting": "Foobar"  
    }  
}
```

From inside the `Startup.Configure` method, you can extract values from the `IConfiguration` instance into your custom type using the following code:

```
C#  
  
builder.Services.AddOptions<MyOptions>()  
    .Configure<IConfiguration>((settings, configuration) =>  
    {
```

```
        configuration.GetSection("MyOptions").Bind(settings);
    });
}
```

Calling `Bind` copies values that have matching property names from the configuration into the custom instance. The options instance is now available in the IoC container to inject into a function.

The options object is injected into the function as an instance of the generic `IOptions` interface. Use the `Value` property to access the values found in your configuration.

C#

```
using System;
using Microsoft.Extensions.Options;

public class HttpTrigger
{
    private readonly MyOptions _settings;

    public HttpTrigger(IOptions<MyOptions> options)
    {
        _settings = options.Value;
    }
}
```

For more information, see [Options pattern in ASP.NET Core](#).

Using ASP.NET Core user secrets

When you develop your app locally, ASP.NET Core provides a [Secret Manager](#) tool that allows you to store secret information outside the project root. It makes it less likely that secrets are accidentally committed to source control. Azure Functions Core Tools (version 3.0.3233 or later) automatically reads secrets created by the ASP.NET Core Secret Manager.

To configure a .NET Azure Functions project to use user secrets, run the following command in the project root.

Bash

```
dotnet user-secrets init
```

Then use the `dotnet user-secrets set` command to create or update secrets.

Bash

```
dotnet user-secrets set MySecret "my secret value"
```

To access user secrets values in your function app code, use `IConfiguration` or `IOptions`.

Customizing configuration sources

To specify other configuration sources, override the `ConfigureAppConfiguration` method in your function app's `Startup` class.

The following sample adds configuration values from both base and optional environment-specific app settings files.

C#

```
using System.IO;
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

[assembly: FunctionsStartup(typeof(MyNamespace.Startup))]

namespace MyNamespace;

public class Startup : FunctionsStartup
{
    public override void
ConfigureAppConfiguration(IFunctionsConfigurationBuilder builder)
    {
        FunctionsHostBuilderContext context = builder.GetContext();

        builder.ConfigurationBuilder
            .AddJsonFile(Path.Combine(context.ApplicationRootPath,
"appsettings.json"), optional: true, reloadOnChange: false)
            .AddJsonFile(Path.Combine(context.ApplicationRootPath,
$"appsettings.{context.EnvironmentName}.json"), optional: true,
reloadOnChange: false)
            .AddEnvironmentVariables();
    }

    public override void Configure(IFunctionsHostBuilder builder)
    {
    }
}
```

Add configuration providers to the `ConfigurationBuilder` property of `IFunctionsConfigurationBuilder`. For more information on using configuration

providers, see [Configuration in ASP.NET Core](#).

A `FunctionsHostBuilderContext` is obtained from `IFunctionsConfigurationBuilder.GetContext()`. Use this context to retrieve the current environment name and resolve the location of configuration files in your function app folder.

By default, configuration files such as `appsettings.json` aren't automatically copied to the function app's output folder. Update your `.csproj` file to match the following sample to ensure the files are copied.

XML

```
<None Update="appsettings.json">
  <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</None>
<None Update="appsettings.Development.json">
  <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  <CopyToPublishDirectory>Never</CopyToPublishDirectory>
</None>
```

Next steps

For more information, see the following resources:

- [How to monitor your function app](#)
- [Best practices for functions](#)

Manage connections in Azure Functions

Article • 11/18/2021

Functions in a function app share resources. Among those shared resources are connections: HTTP connections, database connections, and connections to services such as Azure Storage. When many functions are running concurrently in a Consumption plan, it's possible to run out of available connections. This article explains how to code your functions to avoid using more connections than they need.

ⓘ Note

Connection limits described in this article apply only when running in a **Consumption plan**. However, the techniques described here may be beneficial when running on any plan.

Connection limit

The number of available connections in a Consumption plan is limited partly because a function app in this plan runs in a [sandbox environment](#). One of the restrictions that the sandbox imposes on your code is a limit on the number of outbound connections, which is currently 600 active (1,200 total) connections per instance. When you reach this limit, the functions runtime writes the following message to the logs: `Host thresholds exceeded: Connections`. For more information, see the [Functions service limits](#).

This limit is per instance. When the [scale controller adds function app instances](#) to handle more requests, each instance has an independent connection limit. That means there's no global connection limit, and you can have much more than 600 active connections across all active instances.

When troubleshooting, make sure that you have enabled Application Insights for your function app. Application Insights lets you view metrics for your function apps like executions. For more information, see [View telemetry in Application Insights](#).

Static clients

To avoid holding more connections than necessary, reuse client instances rather than creating new ones with each function invocation. We recommend reusing client connections for any language that you might write your function in. For example, .NET

clients like the [HttpClient](#), [DocumentClient](#), and Azure Storage clients can manage connections if you use a single, static client.

Here are some guidelines to follow when you're using a service-specific client in an Azure Functions application:

- *Do not* create a new client with every function invocation.
- *Do* create a single, static client that every function invocation can use.
- *Consider* creating a single, static client in a shared helper class if different functions use the same service.

Client code examples

This section demonstrates best practices for creating and using clients from your function code.

HTTP requests

C#

Here's an example of C# function code that creates a static [HttpClient](#) instance:

```
C#  
  
// Create a single, static HttpClient  
private static HttpClient httpClient = new HttpClient();  
  
public static async Task Run(string input)  
{  
    var response = await httpClient.GetAsync("https://example.com");  
    // Rest of function  
}
```

A common question about [HttpClient](#) in .NET is "Should I dispose of my client?" In general, you dispose of objects that implement [IDisposable](#) when you're done using them. But you don't dispose of a static client because you aren't done using it when the function ends. You want the static client to live for the duration of your application.

Azure Cosmos DB clients

C#

[CosmosClient](#) connects to an Azure Cosmos DB instance. The Azure Cosmos DB documentation recommends that you [use a singleton Azure Cosmos DB client for the lifetime of your application](#). The following example shows one pattern for doing that in a function:

C#

```
#r "Microsoft.Azure.Cosmos"
using Microsoft.Azure.Cosmos;

private static Lazy<CosmosClient> lazyClient = new Lazy<CosmosClient>
(InitializeCosmosClient);
private static CosmosClient cosmosClient => lazyClient.Value;

private static CosmosClient InitializeCosmosClient()
{
    // Perform any initialization here
    var uri = "https://youraccount.documents.azure.com:443";
    var authKey = "authKey";

    return new CosmosClient(uri, authKey);
}

public static async Task Run(string input)
{
    Container container = cosmosClient.GetContainer("database",
"collection");
    MyItem item = new MyItem{ id = "myId", partitionKey =
"myPartitionKey", data = "example" };
    await container.UpsertItemAsync(item);

    // Rest of function
}
```

Also, create a file named "function.proj" for your trigger and add the below content :

C#

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>netcoreapp3.1</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="Microsoft.Azure.Cosmos"
Version="3.23.0" />
    </ItemGroup>
</Project>
```

SqlClient connections

Your function code can use the .NET Framework Data Provider for SQL Server ([SqlClient](#)) to make connections to a SQL relational database. This is also the underlying provider for data frameworks that rely on ADO.NET, such as [Entity Framework](#). Unlike [HttpClient](#) and [DocumentClient](#) connections, ADO.NET implements connection pooling by default. But because you can still run out of connections, you should optimize connections to the database. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

Tip

Some data frameworks, such as Entity Framework, typically get connection strings from the **ConnectionStrings** section of a configuration file. In this case, you must explicitly add SQL database connection strings to the **Connection strings** collection of your function app settings and in the **local.settings.json** file in your local project. If you're creating an instance of **SqlConnection** in your function code, you should store the connection string value in **Application settings** with your other connections.

Next steps

For more information about why we recommend static clients, see [Improper instantiation antipattern](#).

For more Azure Functions performance tips, see [Optimize the performance and reliability of Azure Functions](#).

Azure Functions error handling and retries

Article • 04/26/2024

Handling errors in Azure Functions is important to help you avoid lost data, avoid missed events, and monitor the health of your application. It's also an important way to help you understand the retry behaviors of event-based triggers.

This article describes general strategies for error handling and the available retry strategies.

ⓘ Important

Preview retry policy support for certain triggers was removed in December 2022.

Retry policies for supported triggers are now generally available (GA). For a list of extensions that currently support retry policies, see the [Retries](#) section.

Handling errors

Errors that occur in an Azure function can come from:

- Use of built-in Functions [triggers and bindings](#).
- Calls to APIs of underlying Azure services.
- Calls to REST endpoints.
- Calls to client libraries, packages, or third-party APIs.

To avoid loss of data or missed messages, it's important to practice good error handling. This table describes some recommended error-handling practices and provides links to more information.

[] [Expand table](#)

Recommendation	Details
Enable Application Insights	Azure Functions integrates with Application Insights to collect error data, performance data, and runtime logs. You should use Application Insights to discover and better understand errors that occur in your function executions. To learn more, see Monitor Azure Functions .
Use structured error handling	Capturing and logging errors is critical to monitoring the health of your application. The top-most level of any function code should include a

Recommendation	Details
	try/catch block. In the catch block, you can capture and log errors. For information about what errors might be raised by bindings, see Binding error codes . Depending on your specific retry strategy, you might also raise a new exception to run the function again.
Plan your retry strategy	Several Functions bindings extensions provide built-in support for retries and others let you define retry policies, which are implemented by the Functions runtime. For triggers that don't provide retry behaviors, you should consider implementing your own retry scheme. For more information, see Retries .
Design for idempotency	The occurrence of errors when you're processing data can be a problem for your functions, especially when you're processing messages. It's important to consider what happens when the error occurs and how to avoid duplicate processing. To learn more, see Designing Azure Functions for identical input .

Retries

There are two kinds of retries available for your functions:

- Built-in retry behaviors of individual trigger extensions
- Retry policies provided by the Functions runtime

The following table indicates which triggers support retries and where the retry behavior is configured. It also links to more information about errors that come from the underlying services.

[] [Expand table](#)

Trigger/binding	Retry source	Configuration
Azure Cosmos DB	Retry policies	Function-level
Blob Storage	Binding extension	host.json
Event Grid	Binding extension	Event subscription
Event Hubs	Retry policies	Function-level
Kafka	Retry policies	Function-level
Queue Storage	Binding extension	host.json
RabbitMQ	Binding extension	Dead letter queue ↗

Trigger/binding	Retry source	Configuration
Service Bus	Binding extension	host.json*
Timer	Retry policies	Function-level

*Requires version 5.x of the Azure Service Bus extension. In older extension versions, retry behaviors are implemented by the [Service Bus dead letter queue](#).

Retry policies

Azure Functions lets you define retry policies for specific trigger types, which are enforced by the runtime. These trigger types currently support retry policies:

- [Azure Cosmos DB](#)
- [Event Hubs](#)
- [Kafka](#)
- [Timer](#)

Retry policies aren't supported in version 1.x of the Functions runtime.

The retry policy tells the runtime to rerun a failed execution until either successful completion occurs or the maximum number of retries is reached.

A retry policy is evaluated when a function executed by a supported trigger type raises an uncaught exception. As a best practice, you should catch all exceptions in your code and raise new exceptions for any errors that you want to result in a retry.

Important

Event Hubs checkpoints aren't written until after the retry policy for the execution has completed. Because of this behavior, progress on the specific partition is paused until the current batch is done processing.

The version 5.x of the Event Hubs extension supports additional retry capabilities for interactions between the Functions host and the event hub. For more information, see `clientRetryOptions` in the [Event Hubs host.json reference](#).

Retry strategies

You can configure two retry strategies that are supported by policy:

Fixed delay

A specified amount of time is allowed to elapse between each retry.

When running in a Consumption plan, you are only billed for time your function code is executing. You aren't billed for the wait time between executions in either of these retry strategies.

Max retry counts

You can configure the maximum number of times that a function execution is retried before eventual failure. The current retry count is stored in memory of the instance.

It's possible for an instance to have a failure between retry attempts. When an instance fails during a retry policy, the retry count is lost. When there are instance failures, the Event Hubs trigger is able to resume processing and retry the batch on a new instance, with the retry count reset to zero. The timer trigger doesn't resume on a new instance.

This behavior means that the maximum retry count is a best effort. In some rare cases, an execution could be retried more than the requested maximum number of times. For Timer triggers, the retries can be less than the maximum number requested.

Retry examples

Examples are provided for both fixed delay and exponential backoff strategies. To see examples for a specific strategy, you must first select that strategy in the previous tab.

Isolated worker model

Function-level retries are supported with the following NuGet packages:

- [Microsoft.Azure.Functions.Worker.Sdk](#) >= 1.9.0
- [Microsoft.Azure.Functions.Worker.Extensions.EventHubs](#) >= 5.2.0
- [Microsoft.Azure.Functions.Worker.Extensions.Kafka](#) >= 3.8.0
- [Microsoft.Azure.Functions.Worker.Extensions.Timer](#) >= 4.2.0

C#

```
[Function(nameof(TimerFunction))]
[FixedDelayRetry(5, "00:00:10")]
public static void Run([TimerTrigger("0 */5 * * *")] TimerInfo
timerInfo,
    FunctionContext context)
```

```
{  
    var logger = context.GetLogger(nameof(TimerFunction));  
    logger.LogInformation($"Function Ran. Next timer schedule =  
{timerInfo.ScheduleStatus.Next}");  
}
```

[+] [Expand table](#)

Property	Description
MaxRetryCount	Required. The maximum number of retries allowed per function execution. -1 means to retry indefinitely.
DelayInterval	The delay used between retries. Specify it as a string with the format HH:mm:ss.

Binding error codes

When you're integrating with Azure services, errors might originate from the APIs of the underlying services. Information that relates to binding-specific errors is available in the "Exceptions and return codes" sections of the following articles:

- [Azure Cosmos DB](#)
- [Blob Storage](#)
- [Event Grid](#)
- [Event Hubs](#)
- [IoT Hub](#)
- [Notification Hubs](#)
- [Queue Storage](#)
- [Service Bus](#)
- [Table Storage](#)

Next steps

- [Azure Functions triggers and bindings concepts](#)
- [Best practices for reliable Azure functions](#)

Manually run a non HTTP-triggered function

Article • 08/21/2024

This article demonstrates how to manually run a non HTTP-triggered function via specially formatted HTTP request.

In some contexts, such as during development and troubleshooting, you might need to run "on-demand" an Azure Function that is indirectly triggered. Examples of indirect triggers include [functions on a schedule](#) or functions that run as the [result of events](#).

The procedure described in this article is equivalent to using the **Test/Run** functionality of a function's **Code + Test** tab in the Azure portal. You can also use Visual Studio Code to [manually run functions](#).

Prerequisites

The examples in this article use an HTTP test tool. Make sure to choose a tool that keeps your data secure. For more information, see [HTTP test tools](#).

Define the request location

To run a non HTTP-triggered function, you need a way to send a request to Azure to run the function. The URL used to make this request takes a specific form.

https://myfunctionsdemos.azurewebsites.net/admin/functions/QueueTrigger	HOST NAME	FOLDER PATH	FUNCTION NAME
---	-----------	-------------	---------------

- **Host name:** The function app's public location that is made up from the function app's name plus `azurewebsites.net` or your custom domain. When you work with [deployment slots](#) used for staging, the host name portion is the production host name with `-<slotname>` appended to it. In the previous example, the URL would be `myfunctiondemos-staging.azurewebsites.net` for a slot named `staging`.
- **Folder path:** To access non HTTP-triggered functions via an HTTP request, you have to send the request through the path `admin/functions`. APIs under the `/admin/` path are only accessible with authorization.
- **Function name:** The name of the function you want to run.

The following considerations apply when making requests to administrator endpoints in your function app:

- When making requests to any endpoint under the `/admin/` path, you must supply your app's master key in the `x-functions-key` header of the request.
- When you run locally, authorization isn't enforced and the function's master key isn't required. You can directly [call the function](#) omitting the `x-functions-key` header.
- When accessing function app endpoints in a [deployment slot](#), make sure you use the slot-specific host name in the request URL, along with the slot-specific master key.

Get the master key

You can get the master key from either the Azure portal or by using the Azure CLI.

⊗ Caution

Due to the elevated permissions in your function app granted by the master key, you should not share this key with third parties or distribute it in an application. The key should only be sent to an HTTPS endpoint.

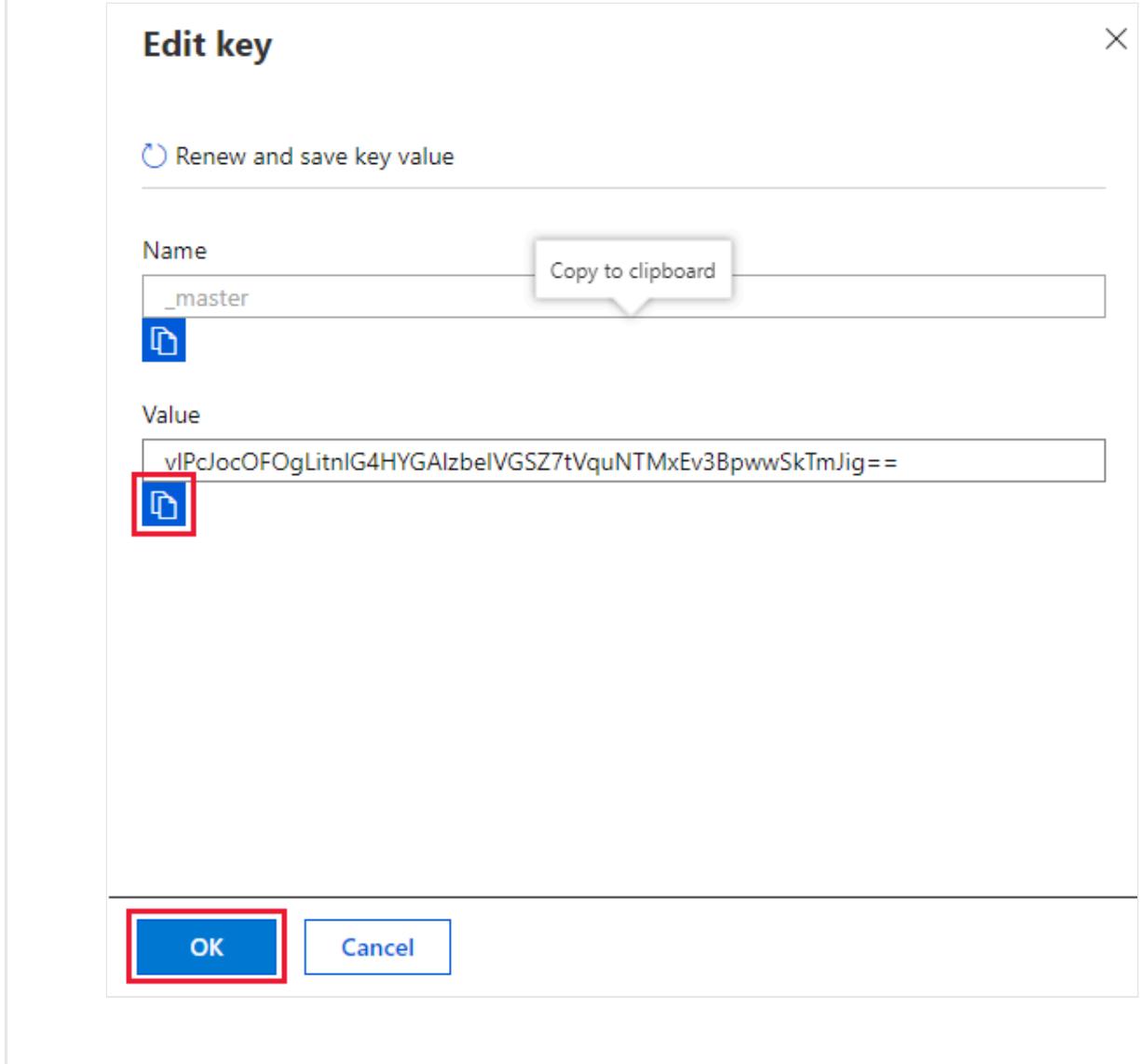
Azure portal

1. Navigate to your function app in the [Azure portal](#), select **App Keys**, and then the `_master` key.

The screenshot shows the Azure portal interface for a function app named "myFunctionApp-dma". The left sidebar has a "Functions" section with "App keys" highlighted. The main content area shows the "Host keys (all functions)" section. A table lists two keys: "_master" and "default". The row for "_master" is highlighted with a red border. The "Value" column for both rows contains the placeholder text "Hidden value. Click to show value". There are "Renew" links at the bottom of each row. A "System keys" section is also visible at the bottom.

Name	Value	Renew
_master	Hidden value. Click to show value	Renew
default	Hidden value. Click to show value	Renew

2. In the **Edit key** section, copy the key value to your clipboard, and then select **OK**.



Call the function

1. In the Azure portal, navigate top your function app and choose your function.
2. Select **Code + Test**, and then select **Logs**. You see messages from the function logged here when you manually run the function from your HTTP test tool.

The screenshot shows the Azure Functions portal interface for a function named 'QueueTrigger1'. The left sidebar has a 'Developer' section with tabs: 'Overview' (selected), 'Code + Test' (highlighted with a red box), 'Integration', 'Monitor', and 'Function Keys'. The main area displays the C# code for the 'run.csx' file:

```
1  using System;
2
3  public static void Run(string myQueueItem, ILogger log)
4  {
5      log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
6  }
7
```

At the bottom of the code editor, there is a 'Logs' button, which is also highlighted with a red box.

3. In your HTTP test tool, use the request location you defined as the request URL, make sure that the HTTP request method is POST, and include these two request headers:

[\[+\] Expand table](#)

Key	Value
x-functions-key	The master key value pasted from the clipboard.
Content-Type	application/json

4. Make sure that the POST request payload/body is `{ "input": "<TRIGGER_INPUT>" }`. The specific `<TRIGGER_INPUT>` you supply depends on the type of trigger, but it can only be a string, numeric, or boolean value. For services that use JSON payloads, such as Azure Service Bus, the test JSON payload should be escaped and serialized as a string.

If you don't want to pass input data to the function, you must still supply an empty dictionary `{}` as the body of the POST request. For more information, see the reference article for the specific non-HTTP trigger.

5. Send the HTTP POST request. The response should be an HTTP 202 (Accepted) response.

6. Next, return to your function in the Azure portal. Review the logs and you see messages coming from the manual call to the function.

The screenshot shows the Azure Functions portal interface. The left sidebar has 'QueueTrigger1 | Code + Test' selected under 'Developer'. The main area displays the C# code for 'run.csx':

```
1  using System;
2
3  public static void Run(string myQueueItem, ILogger log)
4  {
5      log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
6  }
7
```

Below the code editor is a 'Logs' section. The logs show:

- Connected!
- 2020-04-26T05:41:40Z [Information] Executed 'Functions.QueueTrigger1' (Succeeded, Id=433a1df4-4d86-40f7-abc1-0e6438215af9)
- 2020-04-26T05:41:40Z [Information] Queue trigger function processed a request.
- 2020-04-26T05:41:40Z [Information] Executing 'Functions.QueueTrigger1' (Reason='This function was programmatically called via the host APIs.', Id=433a1df4-4d86-40f7-abc1-0e6438215af9)

The way that you access data sent to the trigger depends on the type of trigger and your function language. For more information, see the reference examples for your specific trigger.

Next steps

[Event Grid local testing with viewer web app](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

Bring dependencies or third party library to Azure Functions

Article • 03/24/2022

In this article, you learn to bring in third-party dependencies into your functions apps. Examples of third-party dependencies are json files, binary files and machine learning models.

In this article, you learn how to:

- ✓ Bring in dependencies via Functions Code project
- ✓ Bring in dependencies via mounting Azure Fileshare

Bring in dependencies from the project directory

One of the simplest ways to bring in dependencies is to put the files/artifact together with the functions app code in Functions project directory structure. Here's an example of the directory samples in a Python functions project:

```
<project_root>/  
| - my_first_function/  
| | - __init__.py  
| | - function.json  
| | - example.py  
| - dependencies/  
| | - dependency1  
| - .funcignore  
| - host.json  
| - local.settings.json
```

By putting the dependencies in a folder inside functions app project directory, the dependencies folder will get deployed together with the code. As a result, your function code can access the dependencies in the cloud via file system api.

Accessing the dependencies in your code

Here's an example to access and execute `ffmpeg` dependency that is put into `<project_root>/ffmpeg_lib` directory.

Python

```
import logging

import azure.functions as func
import subprocess

FFMPEG_RELATIVE_PATH = "../ffmpeg_lib/ffmpeg"

def main(req: func.HttpRequest,
         context: func.Context) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    command = req.params.get('command')
    # If no command specified, set the command to help
    if not command:
        command = "-h"

    # context.function_directory returns the current directory in which
    # functions is executed
    ffmpeg_path = "/".join([str(context.function_directory),
                           FFMPEG_RELATIVE_PATH])

    try:
        byte_output = subprocess.check_output([ffmpeg_path, command])
        return func.HttpResponse(byte_output.decode('UTF-
8').rstrip(), status_code=200)
    except Exception as e:
        return func.HttpResponse("Unexpected exception happened when
executing ffmpeg. Error message:" + str(e), status_code=200)
```

ⓘ Note

You may need to use `chmod` to provide `Execute` rights to the `ffmpeg` binary in a Linux environment

Bring dependencies by mounting a file share

When running your function app on Linux, there's another way to bring in third-party dependencies. Functions lets you mount a file share hosted in Azure Files. Consider this approach when you want to decouple dependencies or artifacts from your application code.

First, you need to create an Azure Storage Account. In the account, you also need to create file share in Azure files. To create these resources, follow this [guide](#)

After you created the storage account and file share, use the [az webapp config storage-account add](#) command to attach the file share to your functions app, as shown in the following example.

```
Azure CLI

az webapp config storage-account add \
--name < Function-App-Name > \
--resource-group < Resource-Group > \
--subscription < Subscription-Id > \
--custom-id < Unique-Custom-Id > \
--storage-type AzureFiles \
--account-name < Storage-Account-Name > \
--share-name < File-Share-Name > \
--access-key < Storage-Account-AccessKey > \
--mount-path </path/to/mount>
```

Flag	Value
custom-id	Any unique string
storage-type	Only AzureFiles is supported currently
share-name	Pre-existing share
mount-path	Path at which the share will be accessible inside the container. Value has to be of the format /dir-name and it can't start with /home

More commands to modify/delete the file share configuration can be found [here](#)

Uploading the dependencies to Azure Files

One option to upload your dependency into Azure Files is through Azure portal. Refer to this [guide](#) for instruction to upload dependencies using portal. Other options to upload your dependencies into Azure Files are through [Azure CLI](#) and [PowerShell](#).

Accessing the dependencies in your code

After your dependencies are uploaded in the file share, you can access the dependencies from your code. The mounted share is available at the specified *mount-path*, such as /path/to/mount. You can access the target directory by using file system APIs.

The following example shows HTTP trigger code that accesses the `ffmpeg` library, which is stored in a mounted file share.

Python

```
import logging

import azure.functions as func
import subprocess

FILE_SHARE_MOUNT_PATH = os.environ['FILE_SHARE_MOUNT_PATH']
FFMPEG = "ffmpeg"

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    command = req.params.get('command')
    # If no command specified, set the command to help
    if not command:
        command = "-h"

    try:
        byte_output =
subprocess.check_output(["/".join(FILE_SHARE_MOUNT_PATH, FFMPEG), command])
        return func.HttpResponse(byte_output.decode('UTF-
8').rstrip(),status_code=200)
    except Exception as e:
        return func.HttpResponse("Unexpected exception happened when
executing ffmpeg. Error message:" + str(e),status_code=200)
```

When you deploy this code to a function app in Azure, you need to [create an app setting](#) with a key name of `FILE_SHARE_MOUNT_PATH` and value of the mounted file share path, which for this example is `/azure-files-share`. To do local debugging, you need to populate the `FILE_SHARE_MOUNT_PATH` with the file path where your dependencies are stored in your local machine. Here's an example to set `FILE_SHARE_MOUNT_PATH` using `local.settings.json`:

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "FILE_SHARE_MOUNT_PATH" : "PATH_TO_LOCAL_FFMPEG_DIR"
  }
}
```

Next steps

- [Azure Functions Python developer guide](#)
- [Azure Functions Java developer guide](#)
- [Azure Functions developer reference](#)

Develop Python worker extensions for Azure Functions

Article • 04/25/2023

Azure Functions lets you integrate custom behaviors as part of Python function execution. This feature enables you to create business logic that customers can easily use in their own function apps. To learn more, see the [Python developer reference](#). Worker extensions are supported in both the v1 and v2 Python programming models.

In this tutorial, you'll learn how to:

- ✓ Create an application-level Python worker extension for Azure Functions.
- ✓ Consume your extension in an app the way your customers do.
- ✓ Package and publish an extension for consumption.

Prerequisites

Before you start, you must meet these requirements:

- [Python 3.7 or above](#). To check the full list of supported Python versions in Azure Functions, see the [Python developer guide](#).
- The [Azure Functions Core Tools](#), version 4.0.5095 or later, which supports using the extension with the [v2 Python programming model](#). Check your version with `func -version`.
- [Visual Studio Code](#) installed on one of the [supported platforms](#).

Create the Python Worker extension

The extension you create reports the elapsed time of an HTTP trigger invocation in the console logs and in the HTTP response body.

Folder structure

The folder for your extension project should be like the following structure:

```
<python_worker_extension_root>/  
| - .venv/
```

```
| - python_worker_extension_timer/
| | - __init__.py
| - setup.py
| - readme.md
```

Folder/file	Description
.venv/	(Optional) Contains a Python virtual environment used for local development.
python_worker_extension/	Contains the source code of the Python worker extension. This folder contains the main Python module to be published into PyPI.
setup.py	Contains the metadata of the Python worker extension package.
readme.md	Contains the instruction and usage of your extension. This content is displayed as the description in the home page in your PyPI project.

Configure project metadata

First you create `setup.py`, which provides essential information about your package. To make sure that your extension is distributed and integrated into your customer's function apps properly, confirm that `'azure-functions >= 1.7.0, < 2.0.0'` is in the `install_requires` section.

In the following template, you should change `author`, `author_email`, `install_requires`, `license`, `packages`, and `url` fields as needed.

Python

```
from setuptools import find_packages, setup
setup(
    name='python-worker-extension-timer',
    version='1.0.0',
    author='Your Name Here',
    author_email='your@email.here',
    classifiers=[
        'Intended Audience :: End Users/Desktop',
        'Development Status :: 5 - Production/Stable',
        'Intended Audience :: End Users/Desktop',
        'License :: OSI Approved :: Apache Software License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 3.7',
        'Programming Language :: Python :: 3.8',
        'Programming Language :: Python :: 3.9',
        'Programming Language :: Python :: 3.10',
    ],
    description='Python Worker Extension Demo',
    include_package_data=True,
```

```
long_description=open('readme.md').read(),
install_requires=[
    'azure-functions >= 1.7.0, < 2.0.0',
    # Any additional packages that will be used in your extension
],
extras_require={},
license='MIT',
packages=find_packages(where='.'),
url='https://your-github-or-pypi-link',
zip_safe=False,
)
```

Next, you'll implement your extension code in the application-level scope.

Implement the timer extension

Add the following code in `python_worker_extension_timer/__init__.py` to implement the application-level extension:

Python

```
import typing
from logging import Logger
from time import time
from azure.functions import AppExtensionBase, Context, HttpResponse
class TimerExtension(AppExtensionBase):
    """A Python worker extension to record elapsed time in a function
invocation
"""

@classmethod
def init(cls):
    # This records the starttime of each function
    cls.start_timestamps: typing.Dict[str, float] = {}

@classmethod
def configure(cls, *args, append_to_http_response:bool=False, **kwargs):
    # Customer can use
TimerExtension.configure(append_to_http_response=)
    # to decide whether the elapsed time should be shown in HTTP
response
    cls.append_to_http_response = append_to_http_response

@classmethod
def pre_invocation_app_level(
    cls, logger: Logger, context: Context,
    func_args: typing.Dict[str, object],
    *args, **kwargs
) -> None:
    logger.info(f'Recording start time of {context.function_name}')
    cls.start_timestamps[context.invocation_id] = time()
```

```

@classmethod
def post_invocation_app_level(
    cls, logger: Logger, context: Context,
    func_args: typing.Dict[str, object],
    func_ret: typing.Optional[object],
    *args, **kwargs
) -> None:
    if context.invocation_id in cls.start_timestamps:
        # Get the start_time of the invocation
        start_time: float =
    cls.start_timestamps.pop(context.invocation_id)
    end_time: float = time()
    # Calculate the elapsed time
    elapsed_time = end_time - start_time
    logger.info(f'Time taken to execute {context.function_name} is
{elapsed_time} sec')
    # Append the elapsed time to the end of HTTP response
    # if the append_to_http_response is set to True
    if cls.append_to_http_response and isinstance(func_ret,
    HttpResponse):
        func_ret._HttpResponse__body += f' (TimeElapsed:
{elapsed_time} sec)'.encode()

```

This code inherits from [AppExtensionBase](#) so that the extension applies to every function in the app. You could have also implemented the extension on a function-level scope by inheriting from [FuncExtensionBase](#).

The `init` method is a class method that's called by the worker when the extension class is imported. You can do initialization actions here for the extension. In this case, a hash map is initialized for recording the invocation start time for each function.

The `configure` method is customer-facing. In your readme file, you can tell your customers when they need to call `Extension.configure()`. The readme should also document the extension capabilities, possible configuration, and usage of your extension. In this example, customers can choose whether the elapsed time is reported in the `HttpResponse`.

The `pre_invocation_app_level` method is called by the Python worker before the function runs. It provides the information from the function, such as function context and arguments. In this example, the extension logs a message and records the start time of an invocation based on its `invocation_id`.

Similarly, the `post_invocation_app_level` is called after function execution. This example calculates the elapsed time based on the start time and current time. It also overwrites the return value of the HTTP response.

Create a readme.md

Create a `readme.md` file in the root of your extension project. This file contains the instructions and usage of your extension. The `readme.md` content is displayed as the description in the home page in your PyPI project.

markdown

```
# Python Worker Extension Timer
```

In this file, tell your customers when they need to call
``Extension.configure()``.

The `readme` should also document the extension capabilities, possible configuration, and usage of your extension.

Consume your extension locally

Now that you've created an extension, you can use it in an app project to verify it works as intended.

Create an HTTP trigger function

1. Create a new folder for your app project and navigate to it.
2. From the appropriate shell, such as Bash, run the following command to initialize the project:

Bash

```
func init --python
```

3. Use the following command to create a new HTTP trigger function that allows anonymous access:

Bash

```
func new -t HttpTrigger -n HttpTrigger -a anonymous
```

Activate a virtual environment

1. Create a Python virtual environment, based on OS as follows:

Windows

Console

```
py -m venv .venv
```

2. Activate the Python virtual environment, based on OS as follows:

Windows

Console

```
.venv\Scripts\Activate.ps1
```

Configure the extension

1. Install remote packages for your function app project using the following command:

Bash

```
pip install -r requirements.txt
```

2. Install the extension from your local file path, in editable mode as follows:

Bash

```
pip install -e <PYTHON_WORKER_EXTENSION_ROOT>
```

In this example, replace `<PYTHON_WORKER_EXTENSION_ROOT>` with the root file location of your extension project.

When a customer uses your extension, they'll instead add your extension package location to the requirements.txt file, as in the following examples:

PyPI

Python

```
# requirements.txt
python_worker_extension_timer==1.0.0
```

3. Open the local.settings.json project file and add the following field to `Values`:

JSON

```
"PYTHON_ENABLE_WORKER_EXTENSIONS": "1"
```

When running in Azure, you instead add `PYTHON_ENABLE_WORKER_EXTENSIONS=1` to the [app settings in the function app](#).

4. Add following two lines before the `main` function in `__init__.py` file for the v1 programming model, or in the `function_app.py` file for the v2 programming model:

Python

```
from python_worker_extension_timer import TimerExtension
TimerExtension.configure(append_to_http_response=True)
```

This code imports the `TimerExtension` module and sets the `append_to_http_response` configuration value.

Verify the extension

1. From your app project root folder, start the function host using `func host start -verbose`. You should see the local endpoint of your function in the output as `https://localhost:7071/api/HttpTrigger`.
2. In the browser, send a GET request to `https://localhost:7071/api/HttpTrigger`. You should see a response like the following, with the `TimeElapsed` data for the request appended.

```
This HTTP triggered function executed successfully. Pass a name in the
query string or in the request body for a personalized response.
(TimeElapsed: 0.0009996891021728516 sec)
```

Publish your extension

After you've created and verified your extension, you still need to complete these remaining publishing tasks:

- ✓ Choose a license.
- ✓ Create a `readme.md` and other documentation.
- ✓ Publish the extension library to a Python package registry or a version control system (VCS).

PyPI

To publish your extension to PyPI:

1. Run the following command to install `twine` and `wheel` in your default Python environment or a virtual environment:

Bash

```
pip install twine wheel
```

2. Remove the old `dist/` folder from your extension repository.

3. Run the following command to generate a new package inside `dist/`:

Bash

```
python setup.py sdist bdist_wheel
```

4. Run the following command to upload the package to PyPI:

Bash

```
twine upload dist/*
```

You may need to provide your PyPI account credentials during upload. You can also test your package upload with `twine upload -r testpypi dist/*`. For more information, see the [Twine documentation](#).

After these steps, customers can use your extension by including your package name in their `requirements.txt`.

For more information, see the [official Python packaging tutorial](#).

Examples

- You can view completed sample extension project from this article in the [python_worker_extension_timer](#) sample repository.
- OpenCensus integration is an open-source project that uses the extension interface to integrate telemetry tracing in Azure Functions Python apps. See the [opencensus-python-extensions-azure](#) repository to review the implementation of this Python worker extension.

Next steps

For more information about Azure Functions Python development, see the following resources:

- [Azure Functions Python developer guide](#)
- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)

Continuous deployment for Azure Functions

Article • 05/05/2024

Azure Functions enables you to continuously deploy the changes made in a source control repository to a connected function app. This [source control integration](#) enables a workflow in which a code update triggers build, packaging, and deployment from your project to Azure.

You should always configure continuous deployment for a staging slot and not for the production slot. When you use the production slot, code updates are pushed directly to production without being verified in Azure. Instead, enable continuous deployment to a staging slot, verify updates in the staging slot, and after everything runs correctly you can [swap the staging slot code into production](#). If you connect to a production slot, make sure that only production-quality code makes it into the integrated code branch.

Steps in this article show you how to configure continuous code deployments to your function app in Azure by using the Deployment Center in the Azure portal. You can also [configure continuous integration using the Azure CLI](#). These steps can target either a staging or a production slot.

Functions supports these sources for continuous deployment to your app:

Azure Repos

Maintain your project code in [Azure Repos](#), one of the services in Azure DevOps. Supports both Git and Team Foundation Version Control. Used with the [Azure Pipelines build provider](#). For more information, see [What is Azure Repos?](#)

You can also connect your function app to an external Git repository, but this requires a manual synchronization. For more information about deployment options, see [Deployment technologies in Azure Functions](#).

ⓘ Note

Continuous deployment options covered in this article are specific to code-only deployments. For containerized function app deployments, see [Enable continuous deployment of containers to Azure](#).

Requirements

The unit of deployment for functions in Azure is the function app. For continuous deployment to succeed, the directory structure of your project must be compatible with the basic folder structure that Azure Functions expects. When you create your code project using Azure Functions Core Tools, Visual Studio Code, or Visual Studio, the Azure Functions templates are used to create code projects with the correct directory structure. All functions in a function app are deployed at the same time and in the same package.

After you enable continuous deployment, access to function code in the Azure portal is configured as *read-only* because the *source of truth* is known to reside elsewhere.

ⓘ Note

The Deployment Center doesn't support enabling continuous deployment for a function app with [inbound network restrictions](#). You need to instead configure the build provider workflow directly in GitHub or Azure Pipelines. These workflows also require you to use a virtual machine in the same virtual network as the function app as either a [self-hosted agent \(Pipelines\)](#) or a [self-hosted runner \(GitHub\)](#).

Select a build provider

Building your code project is part of the deployment process. The specific build process depends on your specific language stack, operating system, and hosting plan. Builds can be done locally or remotely, again depending on your specific hosting. For more information, see [Remote build](#).

ⓘ Important

For increased security, consider using a build provider that supports managed identities, including Azure Pipelines and Gitub Actions. The App Service (Kudu) service requires you to [enable basic authentication](#) and work with text-based credentials.

Functions supports these build providers:

Azure Pipelines

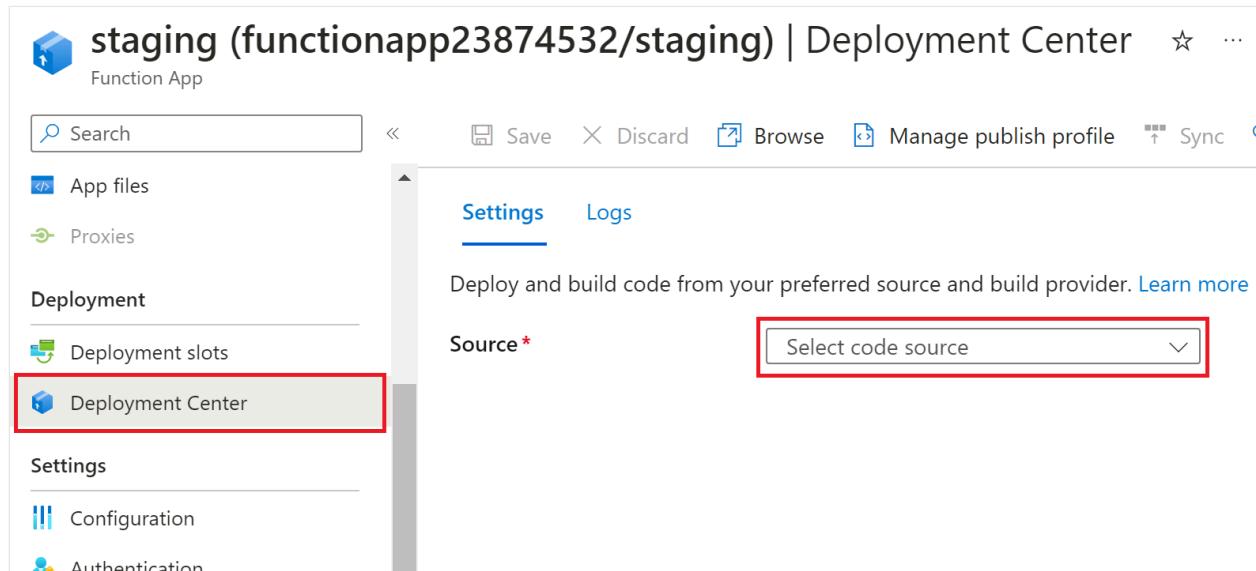
Azure Pipelines is one of the services in Azure DevOps and the default build provider for Azure Repos projects. You can also use Pipelines to build projects from GitHub. In Pipelines, there's an [AzureFunctionApp](#) task designed specifically for deploying to Azure Functions. This task provides you with control over how the project gets built, packaged, and deployed. Supports managed identities.

Keep the strengths and limitations of these providers in mind when you enable source control integration. You might need to change your repository source type to take advantage of a specific provider.

Configure continuous deployment

The [Azure portal](#) provides a Deployment center for your function apps, which makes it easier to configure continuous deployment. The specific way you configure continuous deployment depends both on the type of source control repository in which your code resides and the [build provider](#) you choose.

In the [Azure portal](#), browse to your function app page and select **Deployment Center** under **Deployment** in the left pane.



The screenshot shows the Azure Function App Deployment Center settings page. The left sidebar has sections for 'App files', 'Proxies', 'Deployment', 'Deployment slots', and 'Deployment Center'. 'Deployment Center' is highlighted with a red box. The main area has tabs for 'Settings' (which is active) and 'Logs'. Below the tabs, it says 'Deploy and build code from your preferred source and build provider.' with a 'Learn more' link. A 'Source*' dropdown menu is open, showing 'Select code source' with a red box around it.

Select the **Source** repository type where your project code is being maintained from one of these supported options:

Azure Repos

Deployments from Azure Repos that use Azure Pipelines are defined in the [Azure DevOps portal](#) and not from your function app. For a step-by-step guide for

creating a Pipelines-based deployment from Azure Repos, see [Continuous delivery with Azure Pipelines](#).

After deployment completes, all code from the specified source is deployed to your app. At that point, changes in the deployment source trigger a deployment of those changes to your function app in Azure.

Enable continuous deployment during app creation

Currently, you can configure continuous deployment from GitHub using GitHub Actions when you create your function app in the Azure portal. You can do this on the **Deployment** tab in the [Create Function App](#) page.

If you want to use a different deployment source or build provider for continuous integration, first create your function app and then return to the portal and [set up continuous integration in the Deployment Center](#).

Enable basic authentication for deployments

By default, your function app is created with basic authentication access to the `scm` endpoint disabled. This blocks publishing by all methods that can't use managed identities to access the `scm` endpoint. The publishing impacts of having the `scm` endpoint disabled are detailed in [Deployment without basic authentication](#).

Important

When you use basic authentication, credentials are sent in clear text. To protect these credentials, you must only access the `scm` endpoint over an encrypted connection (HTTPS) when using basic authentication. For more information, see [Secure deployment](#).

To enable basic authentication to the `scm` endpoint:

Azure portal

1. In the [Azure portal](#), navigate to your function app.
2. In the app's left menu, select **Configuration > General settings**.

3. Set SCM Basic Auth Publishing Credentials to On, then select Save.

Next steps

[Best practices for Azure Functions](#)

Azure Functions deployment slots

Article • 03/05/2024

Azure Functions deployment slots allow your function app to run different instances called *slots*. Slots are different environments exposed via a publicly available endpoint. One app instance is always mapped to the production slot, and you can swap instances assigned to a slot on demand. Function apps running in a [Consumption plan](#) have a single extra slot for staging. You can obtain more staging slots by running your app in a [Premium plan](#) or [Dedicated \(App Service\) plan](#). For more information, see [Service limits](#).

The following reflect how functions are affected by swapping slots:

- Traffic redirection is seamless; no requests are dropped because of a swap. This seamless behavior occurs because the next function trigger is routed to the swapped slot.
- Currently executing function are terminated during the swap. To learn how to write stateless and defensive functions, see [Improve the performance and reliability of Azure Functions](#).

Why use slots?

There are many advantages to using deployment slots, including:

- **Different environments for different purposes:** Using different slots gives you the opportunity to differentiate app instances before swapping to production or a staging slot.
- **Prewarming:** Deploying to a slot instead of directly to production allows the app to warm up before going live. Additionally, using slots reduces latency for HTTP-triggered workloads. Instances are warmed up before deployment, which reduces the cold start for newly deployed functions.
- **Easy fallbacks:** After a swap with production, the slot with a previously staged app now has the previous production app. If the changes swapped into the production slot aren't as you expect, you can immediately reverse the swap to get your "last known good instance" back.
- **Minimize restarts:** Changing app settings in a production slot requires a restart of the running app. You can instead change settings in a staging slot and swap the settings change into production with a prewarmed instance. Slots are the recommended way to migrate between Functions runtime versions while maintaining the highest availability. To learn more, see [Minimum downtime update](#).

Swap operations

During a swap, one slot is considered the source and the other is the target. The source slot has the instance of the application that is applied to the target slot. The following steps ensure the target slot doesn't experience downtime during a swap:

1. **Apply settings:** Settings from the target slot are applied to all instances of the source slot. For example, the production settings are applied to the staging instance. The applied settings include the following categories:
 - [Slot-specific app settings and connection strings \(if applicable\)](#)
 - [Continuous deployment settings \(if enabled\)](#)
 - [App Service authentication settings \(if enabled\)](#)
2. **Wait for restarts and availability:** The swap waits for every instance in the source slot to complete its restart and to be available for requests. If any instance fails to restart, the swap operation reverts all changes to the source slot and stops the operation.
3. **Update routing:** If all instances on the source slot are warmed up successfully, the two slots complete the swap by switching routing rules. After this step, the target slot (for example, the production slot) has the app that was previously warmed up in the source slot.
4. **Repeat operation:** Now that the source slot has the preswap app previously in the target slot, complete the same operation by applying all settings and restarting the instances for the source slot.

Keep in mind the following points:

- At any point of the swap operation, initialization of the swapped apps happens on the source slot. The target slot remains online while the source slot is prepared, whether the swap succeeds or fails.
- To swap a staging slot with the production slot, make sure that the production slot is *always* the target slot. This way, the swap operation doesn't affect your production app.
- Settings related to event sources and bindings must be configured as [deployment slot settings before you start a swap](#). Marking them as "sticky" ahead of time ensures events and outputs are directed to the proper instance.

Manage settings

Some configuration settings are slot-specific. The following lists detail which settings change when you swap slots, and which remain the same.

Slot-specific settings:

- Publishing endpoints
- Custom domain names
- Nonpublic certificates and TLS/SSL settings
- Scale settings
- IP restrictions
- Always On
- Diagnostic settings
- Cross-origin resource sharing (CORS)
- Private endpoints

Non slot-specific settings:

- General settings, such as framework version, 32/64-bit, web sockets
- App settings (can be configured to stick to a slot)
- Connection strings (can be configured to stick to a slot)
- Handler mappings
- Public certificates
- Hybrid connections *
- Virtual network integration *
- Service endpoints *
- Azure Content Delivery Network *

Features marked with an asterisk (*) don't get swapped, by design.

ⓘ Note

Certain app settings that apply to unswapped settings are also not swapped. For example, since diagnostic settings are not swapped, related app settings like `WEBSITE_HTTPLOGGING_RETENTION_DAYS` and `DIAGNOSTICS_AZUREBLOBRETENTIONDAYS` are also not swapped, even if they don't show up as slot settings.

Create a deployment setting

You can mark settings as a deployment setting, which makes it *sticky*. A sticky setting doesn't swap with the app instance.

If you create a deployment setting in one slot, make sure to create the same setting with a unique value in any other slot that is involved in a swap. This way, while a setting's value doesn't change, the setting names remain consistent among slots. This name consistency ensures your code doesn't try to access a setting that is defined in one slot but not another.

Use the following steps to create a deployment setting:

1. Navigate to **Deployment slots** in the function app, and then select the slot name.

The screenshot shows the Azure portal interface for managing deployment slots. The URL in the address bar is `Home > myFunctionApp-dma > myFunctionApp-dma | Deployment slots > slot-dma (myfunctionapp-dma/slot-dma) | Deployment slots`. The left sidebar has a tree view with 'Deployment slots' selected. The main content area is titled 'Deployment Slots' and contains a message: 'You have reached the slots quota limit (2) for the current plan.' Below this, it says 'Deployment slots are live apps with their own hostnames. App content and configurations elements can be swapped between two deployment slots, including the production slot.' A table lists the slots:

NAME	STATUS	APP SERVICE PLAN
myfunctionapp-dma	Running	ASP-myResourceGroupdma-82be
myfunctionapp-dma-slot-dma	Running	ASP-myResourceGroupdma-82be

2. Select **Configuration**, and then select the setting name you want to stick with the current slot.

Home > myFunctionApp-dma > myFunctionApp-dma | Deployment slots > slot-dma (myfunctionapp-dma/slot-dma) | Configuration

slot-dma (myfunctionapp-dma/slot-dma) | Configuration

App Service (Slot)

Search (Ctrl+ /) Refresh Save Discard

Application settings Function runtime settings General settings

Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to display them in plain text in your browser by using the controls below. Application Settings are exposed as environment variables for access by your application at runtime. [Learn more](#)

New application setting Show values Advanced edit Filter

Name	Value	Deployment slot setting
APPINSIGHTS_INSTRUMENTATIONKEY	Hidden value. Click to show value	
APPLICATIONINSIGHTS_CONNECTION_STRING	Hidden value. Click to show value	
AzureWebJobsStorage	Hidden value. Click to show value	
FUNCTIONS_EXTENSION_VERSION	Hidden value. Click to show value	
FUNCTIONS_WORKER_RUNTIME	Hidden value. Click to show value	
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING	Hidden value. Click to show value	
WEBSITE_CONTENTSHARE	Hidden value. Click to show value	

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel. Connection strings should only be used with a function app if you are using entity framework. For other scenarios use App Settings. [Learn more](#)

New connection string Show values Advanced edit Filter

Name	Value	Type	Deployment
------	-------	------	------------

3. Select **Deployment slot setting**, and then select **OK**.

Add/Edit application setting

X

Name

APPINSIGHTS_INSTRUMENTATIONKEY



Value

feb69891-99dd-4141-beb4-41e830c6ff0a



Deployment slot setting

OK

Cancel

4. Once setting section disappears, select Save to keep the changes

The screenshot shows the 'slot-dma (myfunctionapp-dma/slot-dma)' blade in the Azure portal. The left sidebar includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Security', 'Functions (preview)', 'Functions', and 'App keys'. The 'Application settings' tab is selected. A note states: 'Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to display them in plain text in your browser by using the controls below. Application Settings are exposed as environment variables for access by your application at runtime.' Below this are buttons for 'New application setting', 'Show values', 'Advanced edit', and 'Filter'. A table lists one setting:

Name	Value	Deployment slot setting
APPINSIGHTS_INSTRUMENTATIONKEY	Hidden value. Click to show value	✓

Deployment

Slots are empty when you create a slot. You can use any of the [supported deployment technologies](#) to deploy your application to a slot.

Scaling

All slots scale to the same number of workers as the production slot.

- For Consumption plans, the slot scales as the function app scales.
- For App Service plans, the app scales to a fixed number of workers. Slots run on the same number of workers as the app plan.

View slots

You can view information about existing slots using either the [Azure CLI](#) or through the [Azure portal](#).

Azure portal

Use these steps to create a new slot in the portal:

1. Navigate to your function app.
2. Select **Deployment slots** and the existing slots are shown.

Add a slot

You can add a slot using either the [Azure CLI](#) or through the [Azure portal](#).

Azure portal

Use these steps to create a slot in the portal:

1. Navigate to your function app.
2. Select **Deployment slots**, and then select **+ Add Slot**.

The screenshot shows the 'Deployment slots' page for a function app named 'functions-ggailey777-7'. The left sidebar has 'Deployment slots' selected. At the top, there's a search bar and a '+ Add Slot' button. A message says 'You haven't added any deployment slots. Click here to get started.' Below is a table with one row:

NAME	STATUS	APP SERVICE PLAN
functions-ggailey777-7 PRODUCTION	Running	ASP-myResourceGroupDKS-b9c2

3. Type the name of the slot and select **Add**.

The screenshot shows the 'Add a slot' dialog box. It has a 'Name' input field containing 'functions-dma-slot' and two buttons at the bottom: 'Add' and 'Close'. The 'Add' button is highlighted with a red box.

Access slot resources

You access resources (HTTP triggers and administrator endpoints) in a staging slot in the same way as the production slot. However, instead of the function app host name you use the slot-specific host name in the request URL, along with any slot-specific keys.

Because staging slots are live apps, you must [secure your functions](#) in a staging slot as you would in the production slot.

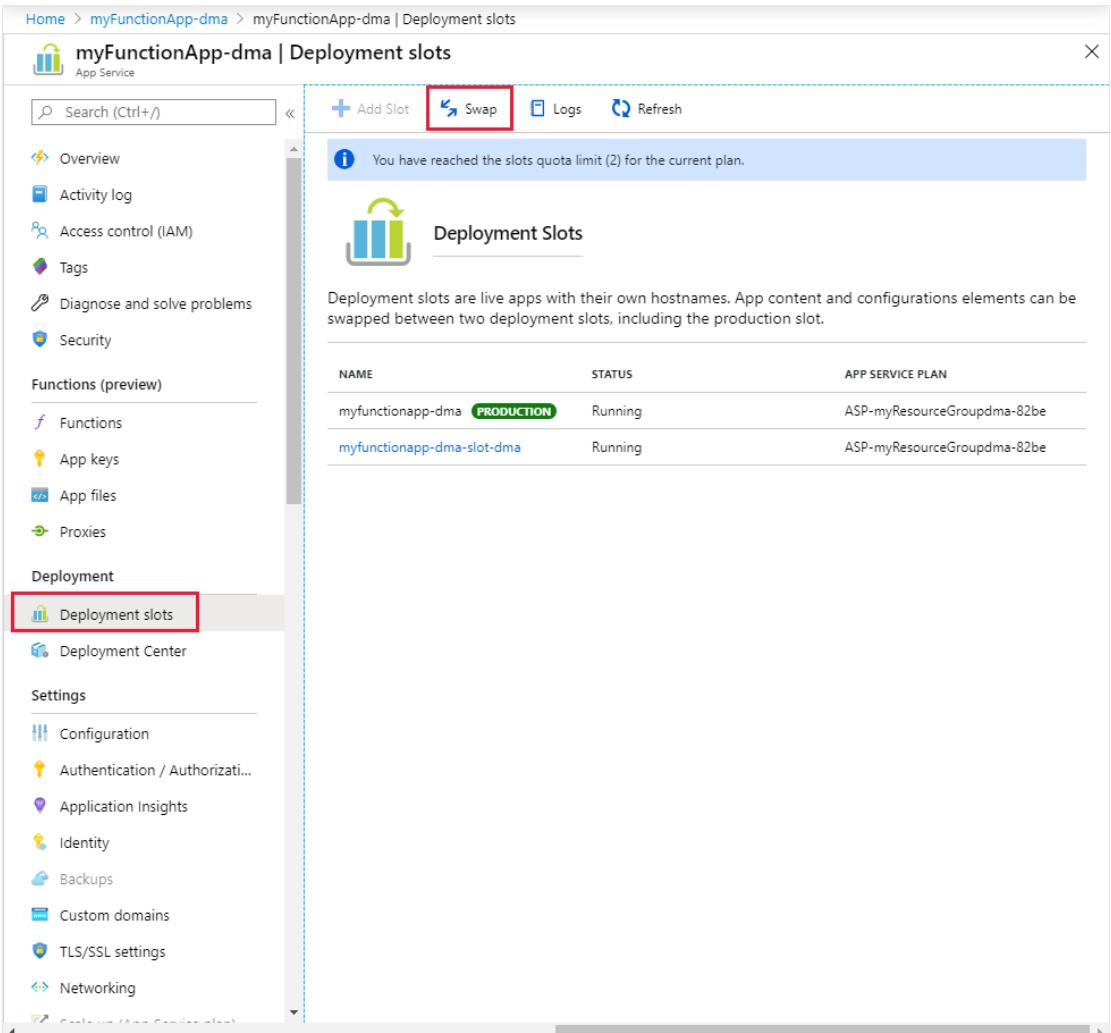
Swap slots

You can swap slots in and out of production using either the [Azure CLI](#) or through the [Azure portal](#).

Azure portal

Use these steps to swap a staging slot into production:

1. Navigate to the function app.
2. Select **Deployment slots**, and then select **Swap**.



NAME	STATUS	APP SERVICE PLAN
myfunctionapp-dma	PRODUCTION	ASP-myResourceGroupdma-82be
myfunctionapp-dma-slot-dma	Running	ASP-myResourceGroupdma-82be

3. Verify the configuration settings for your swap and select **Swap**

Swap

Source
myslottedfunctionapp-staging

Target **PRODUCTION**
myslottedfunctionapp

Perform swap with preview

Config Changes

This is a summary of the final set of configuration changes on the source and target deployment slots after the swap has completed.

Source Changes		Target Changes	
SETTING	TYPE	OLD VALUE	NEW VALUE
WEBSITE_CONTENTS...	AppSetting	myslottedfunctionapp...	myslottedfunctionapp-97782dc9

Swap **Close**

The screenshot shows the 'Swap' dialog box in the Azure portal. It displays two dropdown menus for 'Source' (set to 'myslottedfunctionapp-staging') and 'Target' (set to 'PRODUCTION' which is highlighted in green). There is also a checkbox for 'Perform swap with preview'. Below this is a section titled 'Config Changes' with a summary table. The table has four columns: 'SETTING', 'TYPE', 'OLD VALUE', and 'NEW VALUE'. It shows one change: 'WEBSITE_CONTENTS...' (AppSetting) where the old value is 'myslottedfunctionapp...' and the new value is 'myslottedfunctionapp-97782dc9'. At the bottom are two buttons: 'Swap' (which is highlighted with a red box) and 'Close'.

The swap operation can take a few seconds.

Roll back a swap

If a swap results in an error or you simply want to "undo" a swap, you can roll back to the initial state. To return to the preswapped state, do another swap to reverse the swap.

Remove a slot

You can remove a slot using either the [Azure CLI](#) or through the [Azure portal](#).

Azure portal

Use these steps to remove a slot from your app in the portal:

1. Navigate to **Deployment slots** in the function app, and then select the slot name.

slot-dma (myfunctionapp-dma) | Deployment slots

Add Slot Swap Logs Refresh

You have reached the slots quota limit (2) for the current plan.

Deployment Slots

Deployment slots are live apps with their own hostnames. App content and configurations elements can be swapped between two deployment slots, including the production slot.

NAME	STATUS	APP SERVICE PLAN
myfunctionapp-dma	PRODUCTION Running	ASP-myResourceGroupdma-82be
myfunctionapp-dma-slot-dma	Running	ASP-myResourceGroupdma-82be

Functions (preview)

- Functions
- App keys
- App files
- Proxies

Deployment

- Deployment slots **selected**
- Deployment Center

Settings

- Configuration
- Authentication / Authorizati...
- Application Insights
- Identity
- Backups
- Custom domains
- TLS/SSL settings

2. Select Delete.

slot-dma (myfunctionapp/dma)

Browse Refresh Stop Restart Swap Get publish profile Reset publish profile Delete

Overview

Resource group (change)
myresourcegroup

Status
Running

Location
Central US

Subscription (change)
Vendor Subscriptions

Subscription ID

Tags (change)
Click here to add tags

Metrics Features (9) Notifications (0) Quickstart

Memory working set

1008
908
808
708
608
508

Function Execution Count

100
90
80
70
60
50

Functions (preview)

- Functions
- App keys
- App files
- Proxies

Deployment

- Deployment slots **selected**
- Deployment Center

Settings

- Configuration
- Authentication / Authorizati...

3. Type the name of the deployment slot you want to delete, and then select Delete.



Are you sure you want to delete "slot-dma"?



Warning! Deleting "slot-dma" is irreversible. The action you are about to take cannot be undone. Going further will delete the deployment slots.

TYPE THE DEPLOYMENT SLOT NAME

slot-dma



Affected resources

There are 1 resources that will be deleted

Name	Type
 slot-dma	Slot

Delete

4. Close the confirmation pane.

Deleted



This resource has been removed.

[?](#) Get support

Summary 

Session ID

e84e85ffccdd8464387ed96e835347166

Resource ID

/subscriptions/316e8102-0662-41cb-...

Extension

WebsitesExtension

Content

AppDeleteBlade

Error code

410

[Delete](#)

Change App Service plan

With a function app that is running under an App Service plan, you can change the underlying App Service plan for a slot.

① Note

You can't change a slot's App Service plan under the Consumption plan.

Use the following steps to change a slot's App Service plan:

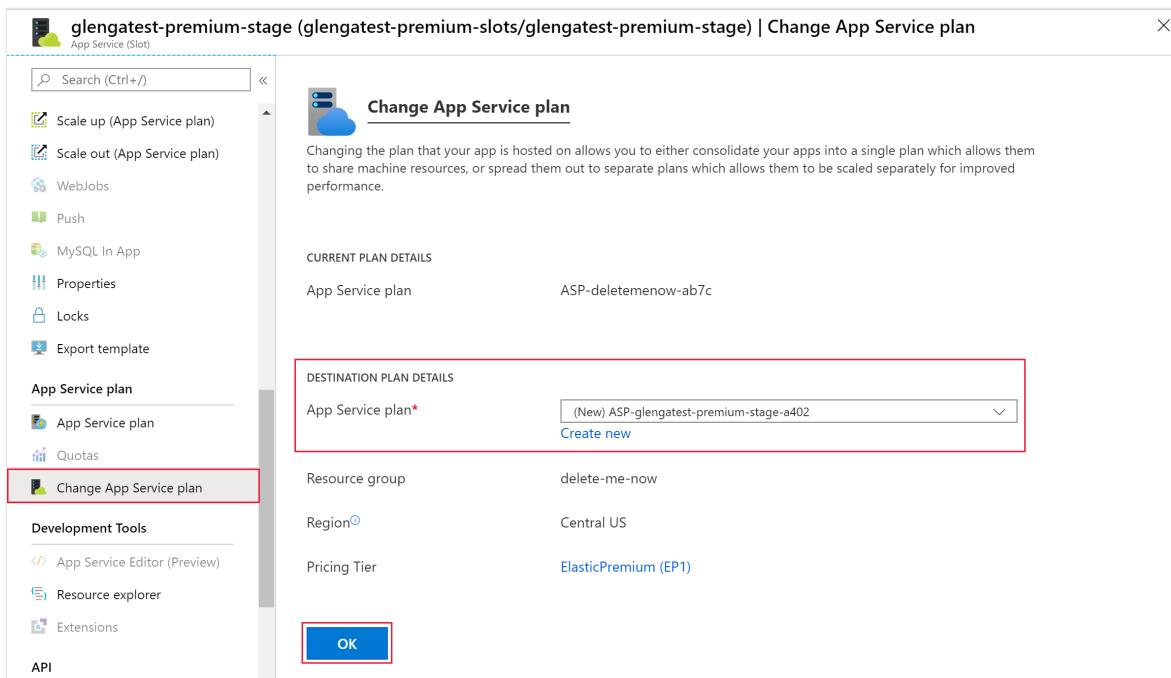
1. Navigate to **Deployment slots** in the function app, and then select the slot name.

The screenshot shows the Azure portal interface for managing deployment slots. The left sidebar navigation bar is visible, with the 'Deployment slots' option highlighted by a red box. The main content area displays the 'Deployment Slots' section, which includes a summary message about reaching the slots quota limit (2) and a table listing two slots: 'myfunctionapp-dma' (Production) and 'myfunctionapp-dma-slot-dma'. The 'myfunctionapp-dma-slot-dma' row is also highlighted with a red box. The table has columns for NAME, STATUS, and APP SERVICE PLAN.

NAME	STATUS	APP SERVICE PLAN
myfunctionapp-dma	Running	ASP-myResourceGroupdma-82be
myfunctionapp-dma-slot-dma	Running	ASP-myResourceGroupdma-82be

2. Under **App Service plan**, select **Change App Service plan**.

3. Select the plan you want to upgrade to, or create a new plan.



4. Select OK.

Considerations

Azure Functions deployment slots have the following considerations:

- The number of slots available to an app depends on the plan. The Consumption plan is only allowed one deployment slot. More slots are available for apps running under other plans. For details, see [Service limits](#).
- Swapping a slot resets keys for apps that have an `AzureWebJobsSecretStorageType` app setting equal to `files`.
- When slots are enabled, your function app is set to read-only mode in the portal.
- Slot swaps might fail when your function app is using a [secured storage account](#) as its default storage account (set in `AzureWebJobsStorage`). For more information, see the [WEBSITE_OVERRIDE_STICKY_DIAGNOSTICS_SETTINGS](#) reference.
- Use function app names shorter than 32 characters. Names longer than 32 characters are at risk of causing [host ID collisions](#).

Next steps

- [Deployment technologies in Azure Functions](#)

Working with containers and Azure Functions

Article • 07/30/2024

This article demonstrates the support that Azure Functions provides for working with containerized function apps running in an Azure Container Apps environment. For more information, see [Azure Container Apps hosting of Azure Functions](#).

Choose the hosting environment for your containerized function app at the top of the article.

If you want to jump right in, the following article shows you how to create your first function running in a Linux container and deploy the image from a container registry to a supported Azure hosting service:

[Create your first containerized Azure Functions on Azure Container Apps](#)

To learn more about deployments to Azure Container Apps, see [Azure Container Apps hosting of Azure Functions](#).

Creating containerized function apps

Functions makes it easy to deploy and run your function apps as Linux containers, which you create and maintain. Functions maintains a set of [language-specific base images](#) that you can use when creating containerized function apps.

Important

When creating your own containers, you are required to keep the base image of your container updated to the latest supported base image. Supported base images for Azure Functions are language-specific and are found in the [Azure Functions base image repos](#).

The Functions team is committed to publishing monthly updates for these base images. Regular updates include the latest minor version updates and security fixes for both the Functions runtime and languages. You should regularly update your container from the latest base image and redeploy the updated version of your container.

For a complete example of how to create the local containerized function app from the command line and publish the image to a container registry, see [Create a function app in a local container](#).

Generate the Dockerfile

Functions tooling provides a Docker option that generates a Dockerfile with your functions code project. You can use this file with Docker to create your functions in a container that derives from the correct base image (language and version).

The way you create a Dockerfile depends on how you create your project.

Command line

- When you create a Functions project using [Azure Functions Core Tools](#), include the `--docker` option when you run the `func init` command, as in the following example:

Console

```
func init --docker
```

- You can also add a Dockerfile to an existing project by using the `--docker-only` option when you run the `func init` command in an existing project folder, as in the following example:

Console

```
func init --docker-only
```

For a complete example, see [Create a function app in a local container](#).

Create your function app in a container

With a Functions-generated Dockerfile in your code project, you can use Docker to create the containerized function app on your local computer. The following `docker build` command creates an image of your containerized functions from the project in the local directory:

Console

```
docker build --tag <DOCKER_ID>/<IMAGE_NAME>:v1.0.0 .
```

For an example of how to create the container, see [Build the container image and verify locally](#).

Update an image in the registry

When you make changes to your functions code project or need to update to the latest base image, you need to rebuild the container locally and republish the updated image to your chosen container registry. The following command rebuilds the image from the root folder with an updated version number and pushes it to your registry:

Azure Container Registry

Azure CLI

```
az acr build --registry <REGISTRY_NAME> --image  
<LOGIN_SERVER>/azurefunctionsimage:v1.0.1 .
```

Replace `<REGISTRY_NAME>` with your Container Registry instance and `<LOGIN_SERVER>` with the login server name.

At this point, you need to update an existing deployment to use the new image. You can update the function app to use the new image either by using the Azure CLI or in the [Azure portal](#):

Azure CLI

Azure CLI

```
az functionapp config container set --image <IMAGE_NAME> --registry-  
password <SECURE_PASSWORD> --registry-username <USER_NAME> --name  
<APP_NAME> --resource-group <RESOURCE_GROUP>
```

In this example, `<IMAGE_NAME>` is the full name of the new image with version. Private registries require you to supply a username and password. Store these credentials securely.

Azure portal create using containers

When you create a Container Apps-hosted function app in the [Azure portal](#), you can choose to deploy your function app from an image in a container registry. To learn how to create a containerized function app in a container registry, see [Create your function app in a container](#).

The following steps create and deploy an existing containerized function app from a container registry.

1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. Under **Select a hosting option**, choose **Container Apps environment > Select**.
4. On the **Basics** page, use the function app settings as specified in the following table:

[] [Expand table](#)

Setting	Suggested value	Description
Subscription	Your subscription	The subscription in which you create your function app.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which you create your function app. You should create a resource group because there are known limitations when creating new function apps in an existing resource group .
Function App name	Unique name*	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Region	Preferred region	Select a region that's near you or near other services that your functions can access.

*App name must be unique within the Azure Container Apps environment.

5. Still on the **Basics** page, accept the suggested new environment for **Azure Container Apps environment**. To minimize costs, the new default environment is created in the **Consumption + Dedicated** with the default workload profile and without zone redundancy. For more information, see [Azure Container Apps hosting of Azure Functions](#).

You can also choose to use an existing Container Apps environment. To create a custom environment, instead select **Create new**. In the **Create Container Apps Environment** page, you can add nondefault workload profiles or enable zone

redundancy. To learn about environments, see [Azure Container Apps environments](#).

6. Select the **Deployment** tab and unselect **Use quickstart image**. Otherwise, the function app is deployed from the base image for your function app language.
7. Choose your **Image type**, public or private. Choose **Private** if you're using Azure Container Registry or some other private registry. Supply the **Image name**, including the registry prefix. If you're using a private registry, provide the image registry authentication credentials. The **Public** setting only supports images stored publicly in Docker Hub.
8. Under **Container resource allocation**, select your desired number of CPU cores and available memory. If your environment has other workload profiles added, you can select a nondefault **Workload profile**. Choices on this page affect the cost of hosting your app. See the [Container Apps pricing page](#) to estimate your potential costs.
9. Select **Review + create** to review the app configuration selections.
10. On the **Review + create** page, review your settings, and then select **Create** to provision the function app and deploy your container image from the registry.

Work with images in Azure Functions

When your function app container is deployed from a registry, Functions maintains information about the source image.

Azure CLI

Use the following commands to get data about the image or change the deployment image used:

- [az functionapp config container show](#): returns information about the image used for deployment.
- [az functionapp config container set](#): change registry settings or update the image used for deployment, as shown in the previous example.

Container Apps workload profiles

Workload profiles are feature of Container Apps that let you better control your deployment resources. Azure Functions on Azure Container Apps also supports workload profiles. For more information, see [Workload profiles in Azure Container Apps](#).

You can also set the amount of CPU and memory resources allocated to your app.

You can create and manage both workload profiles and resource allocations using the Azure CLI or in the Azure portal.

Azure CLI

You enable workload profiles when you create your container app environment. For an example, see [Create a container app in a profile](#).

You can add, edit, and delete profiles in your environment. For an example, see [Add profiles](#).

When you create a containerized function app in an environment that has workload profiles enabled, you should also specify the profile in which to run. You specify the profile by using the `--workload-profile-name` parameter of the [az functionapp create](#) command, like in this example:

Azure CLI

```
az functionapp create --name <APP_NAME> --storage-account <STORAGE_NAME>
--environment MyContainerappEnvironment --resource-group
AzureFunctionsContainers-rg --functions-version 4 --runtime
<LANGUAGE_STACK> --image <IMAGE_URI> --workload-profile-name
<PROFILE_NAME> --cpu <CPU_COUNT> --memory <MEMORY_SIZE>
```

In the [az functionapp create](#) command, the `--environment` parameter specifies the Container Apps environment and the `--image` parameter specifies the image to use for the function app. In this example, replace `<STORAGE_NAME>` with the name you used in the previous section for the storage account. Also, replace `<APP_NAME>` with a globally unique name appropriate to you.

To set the resources allocated to your app, replace `<CPU_COUNT>` with your desired number of virtual CPUs, with a minimum of 0.5 up to the maximum allowed by the profile. For `<MEMORY_SIZE>`, choose a dedicated memory amount from 1 GB up to the maximum allowed by the profile.

You can use the [az functionapp container set](#) command to manage the allocated resources and the workload profile used by your app.

Azure CLI

```
az functionapp container set --name <APP_NAME> --resource-group  
AzureFunctionsContainers-rg --workload-profile-name <PROFILE_NAME> --  
cpu <CPU_COUNT> --memory <MEMORY_SIZE>
```

Application settings

Azure Functions lets you work with application settings for containerized function apps in the standard way. For more information, see [Use application settings](#).

Enable continuous deployment to Azure

When you host your containerized function app on Azure Container Apps, there are two ways to set up continuous deployment from a source code repository:

- [Azure Pipelines](#)
- [GitHub Actions](#)

You aren't currently able to continuously deploy containers based on image changes in a container registry. You must instead use these source-code based continuous deployment pipelines.

Related articles

The following articles provide more information about deploying and managing containers:

- [Azure Container Apps hosting of Azure Functions](#)
- [Scale and hosting options](#)
- [Kubernetes-based serverless hosting](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

Create a function app in a local Linux container

Article • 07/19/2023

This article shows you how to use Azure Functions Core tools to create your first function in a Linux container on your local computer, verify the function locally, and then publish the containerized function to a container registry. From a container registry, you can easily deploy your containerized functions to Azure.

For a complete example of deploying containerized functions to Azure, which include the steps in this article, see one of the following articles:

- [Create your first containerized Azure Functions on Azure Container Apps](#)
- [Create your first containerized Azure Functions](#)
- [Create your first containerized Azure Functions on Azure Arc \(preview\)](#)

You can also create a function app in the Azure portal by using an existing containerized function app from a container registry. For more information, see [Azure portal create using containers](#).

Choose your development language

First, you use Azure Functions tools to create your project code as a function app in a Docker container using a language-specific Linux base image. Make sure to select your language of choice at the top of the article.

Core Tools automatically generates a Dockerfile for your project that uses the most up-to-date version of the correct base image for your functions language. You should regularly update your container from the latest base image and redeploy from the updated version of your container. For more information, see [Creating containerized function apps](#).

Prerequisites

Before you begin, you must have the following requirements in place:

- Install the [.NET 8.0 SDK](#).
- Install [Azure Functions Core Tools](#) version 4.0.5198, or a later version.
- [Azure CLI](#) version 2.4 or a later version.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

To publish the containerized function app image you create to a container registry, you need a Docker ID and [Docker](#) running on your local computer. If you don't have a Docker ID, you can [create a Docker account](#).

Azure Container Registry

You also need to complete the [Create a container registry](#) section of the Container Registry quickstart to create a registry instance. Make a note of your fully qualified login server name.

Create and test the local functions project

In a terminal or command prompt, run the following command for your chosen language to create a function app project in the current folder:

Console

```
func init --worker-runtime dotnet-isolated --docker
```

The `--docker` option generates a *Dockerfile* for the project, which defines a suitable container for use with Azure Functions and the selected runtime.

Use the following command to add a function to your project, where the `--name` argument is the unique name of your function and the `--template` argument specifies the function's trigger. `func new` creates a C# code file in your project.

Console

```
func new --name HttpExample --template "HTTP trigger"
```

To test the function locally, start the local Azure Functions runtime host in the root of the project folder.

Console

```
func start
```

After you see the `HttpExample` endpoint written to the output, navigate to that endpoint. You should see a welcome message in the response output.

Press **Ctrl+C** (**Command+C** on macOS) to stop the host.

Build the container image and verify locally

(Optional) Examine the *Dockerfile* in the root of the project folder. The *Dockerfile* describes the required environment to run the function app on Linux. The complete list of supported base images for Azure Functions can be found in the [Azure Functions base image page](#).

In the root project folder, run the [docker build](#) command, provide a name as `azurefunctionsimage`, and tag as `v1.0.0`. Replace `<DOCKER_ID>` with your Docker Hub account ID. This command builds the Docker image for the container.

Console

```
docker build --tag <DOCKER_ID>/azurefunctionsimage:v1.0.0 .
```

When the command completes, you can run the new container locally.

To verify the build, run the image in a local container using the [docker run](#) command, replace `<DOCKER_ID>` again with your Docker Hub account ID, and add the ports argument as `-p 8080:80`:

Console

```
docker run -p 8080:80 -it <DOCKER_ID>/azurefunctionsimage:v1.0.0
```

After the image starts in the local container, browse to `http://localhost:8080/api/HttpExample`, which must display the same greeting message as before. Because the HTTP triggered function you created uses anonymous authorization, you can call the function running in the container without having to obtain an access key. For more information, see [authorization keys](#).

After verifying the function app in the container, press **Ctrl+C** (**Command+C** on macOS) to stop execution.

Publish the container image to a registry

To make your container image available for deployment to a hosting environment, you must push it to a container registry.

Azure Container Registry

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts. You should use a private registry service for publishing your containers to Azure services.

1. Use the following command to sign in to your registry instance:

Azure CLI

```
az acr login --name <REGISTRY_NAME>
```

In the previous command, replace `<REGISTRY_NAME>` with the name of your Container Registry instance.

2. Use the following command to tag your image with the fully qualified name of your registry login server:

docker

```
docker tag <DOCKER_ID>/azurefunctionsimage:v1.0.0  
<LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

Replace `<LOGIN_SERVER>` with the fully qualified name of your registry login server and `<DOCKER_ID>` with your Docker ID.

3. Use the following command to push the container to your registry instance:

docker

```
docker push <LOGIN_SERVER>/azurefunctionsimage:v1.0.0
```

4. Use the following command to enable the built-in admin account so that Functions can connect to the registry with a username and password:

Azure CLI

```
az acr update -n <REGISTRY_NAME> --admin-enabled true
```

1. Use the following command to retrieve the admin username and password, which Functions needs to connect to the registry:

```
Azure CLI
```

```
az acr credential show -n <REGISTRY_NAME> --query "[username,  
passwords[0].value]" -o tsv
```

 **Important**

The admin account username and password are important credentials. Make sure to store them securely and never in an accessible location like a public repository.

Next steps

[Working with containers and Azure Functions](#)

Azure Functions on Kubernetes with KEDA

Article • 08/20/2024

The Azure Functions runtime provides flexibility in hosting where and how you want. [KEDA](#) (Kubernetes-based Event Driven Autoscaling) pairs seamlessly with the Azure Functions runtime and tooling to provide event driven scale in Kubernetes.

ⓘ Important

Running your containerized function apps on Kubernetes, either by using KEDA or by direct deployment, is an open-source effort that you can use free of cost. Best-effort support is provided by contributors and from the community by using [GitHub issues in the Azure Functions repository](#). Please use these issues to report bugs and raise feature requests.

For fully-supported Kubernetes deployments, instead consider [Azure Container Apps hosting of Azure Functions](#).

How Kubernetes-based functions work

The Azure Functions service is made up of two key components: a runtime and a scale controller. The Functions runtime runs and executes your code. The runtime includes logic on how to trigger, log, and manage function executions. The Azure Functions runtime can run *anywhere*. The other component is a scale controller. The scale controller monitors the rate of events that are targeting your function, and proactively scales the number of instances running your app. To learn more, see [Azure Functions scale and hosting](#).

Kubernetes-based Functions provides the Functions runtime in a [Docker container](#) with event-driven scaling through KEDA. KEDA can scale in to zero instances (when no events are occurring) and out to n instances. It does this by exposing custom metrics for the Kubernetes autoscaler (Horizontal Pod Autoscaler). Using Functions containers with KEDA makes it possible to replicate serverless function capabilities in any Kubernetes cluster. These functions can also be deployed using [Azure Kubernetes Services \(AKS\)](#) [virtual nodes](#) feature for serverless infrastructure.

Managing KEDA and functions in Kubernetes

To run Functions on your Kubernetes cluster, you must install the KEDA component. You can install this component in one of the following ways:

- Azure Functions Core Tools: using the [func kubernetes install command](#).
- Helm: there are various ways to install KEDA in any Kubernetes cluster, including Helm. Deployment options are documented on the [KEDA site ↗](#).

Deploying a function app to Kubernetes

You can deploy any function app to a Kubernetes cluster running KEDA. Since your functions run in a Docker container, your project needs a Dockerfile. You can create a Dockerfile by using the [--docker option](#) when calling `func init` to create the project. If you forgot to create your Dockerfile, you can always call `func init` again from the root of your code project.

1. (Optional) If you need to create your Dockerfile, use the [func init](#) command with the `--docker-only` option:

```
command
```

```
func init --docker-only
```

To learn more about Dockerfile generation, see the [func init](#) reference.

2. Use the [func kubernetes deploy](#) command to build your image and deploy your containerized function app to Kubernetes:

```
command
```

```
func kubernetes deploy --name <name-of-function-deployment> --registry <container-registry-username>
```

In this example, replace `<name-of-function-deployment>` with the name of your function app. The deploy command performs these tasks:

- The Dockerfile created earlier is used to build a local image for your containerized function app.
- The local image is tagged and pushed to the container registry where the user is logged in.
- A manifest is created and applied to the cluster that defines a Kubernetes `Deployment` resource, a `ScaledObject` resource, and `Secrets`, which includes environment variables imported from your `local.settings.json` file.

Deploying a function app from a private registry

The previous deployment steps work for private registries as well. If you're pulling your container image from a private registry, include the `--pull-secret` flag that references the Kubernetes secret holding the private registry credentials when running `func kubernetes deploy`.

Removing a function app from Kubernetes

After deploying you can remove a function by removing the associated `Deployment`, `ScaledObject`, an `Secrets` created.

command

```
kubectl delete deploy <name-of-function-deployment>
kubectl delete ScaledObject <name-of-function-deployment>
kubectl delete secret <name-of-function-deployment>
```

Uninstalling KEDA from Kubernetes

You can remove KEDA from your cluster in one of the following ways:

- Azure Functions Core Tools: using the [func kubernetes remove command](#).
- Helm: see the uninstall steps [on the KEDA site](#).

Supported triggers in KEDA

KEDA has support for the following Azure Function triggers:

- [Azure Storage Queues](#)
- [Azure Service Bus](#)
- [Azure Event / IoT Hubs](#)
- [Apache Kafka ↗](#)
- [RabbitMQ Queue ↗](#)

HTTP Trigger support

You can use Azure Functions that expose HTTP triggers, but KEDA doesn't directly manage them. You can use the KEDA prometheus trigger to [scale HTTP Azure Functions from one to n instances](#).

Next Steps

For more information, see the following resources:

- [Working with containers and Azure Functions](#)
 - [Code and test Azure Functions locally](#)
 - [How the Azure Function Consumption plan works](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Continuous delivery with Azure Pipelines

Article • 04/15/2024

Use [Azure Pipelines](#) to automatically deploy to Azure Functions. Azure Pipelines lets you build, test, and deploy with continuous integration (CI) and continuous delivery (CD) using [Azure DevOps](#).

YAML pipelines are defined using a YAML file in your repository. A step is the smallest building block of a pipeline and can be a script or task (prepackaged script). [Learn about the key concepts and components that make up a pipeline](#).

You'll use the `AzureFunctionApp` task to deploy to Azure Functions. There are now two versions of the `AzureFunctionApp` task ([AzureFunctionApp@1](#), [AzureFunctionApp@2](#)). `AzureFunctionApp@2` includes enhanced validation support that makes pipelines less likely to fail because of errors.

Choose your task version at the top of the article. YAML pipelines aren't available for Azure DevOps 2019 and earlier.

Prerequisites

- An Azure DevOps organization. If you don't have one, you can [create one for free](#). If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a [parallel job](#) or you can request a free tier.
- If you plan to use GitHub instead of Azure Repos, you also need a GitHub repository. If you don't have a GitHub account, you can [create one for free](#).
- An existing function app in Azure that has its source code in a supported repository. If you don't yet have an Azure Functions code project, you can create one by completing the following language-specific article:

C#

[Quickstart: Create a C# function in Azure using Visual Studio Code](#)

Remember to upload the local code project to your GitHub or Azure Repos repository after you publish it to your function app.

Build your app

1. Sign in to your Azure DevOps organization and navigate to your project.
2. In your project, navigate to the **Pipelines** page. Then select **New pipeline**.
3. Select one of these options for **Where is your code?**:
 - **GitHub**: You might be redirected to GitHub to sign in. If so, enter your GitHub credentials. When this is the first connection to GitHub, the wizard also walks you through the process of connecting DevOps to your GitHub accounts.
 - **Azure Repos Git**: You are immediately able to choose a repository in your current DevOps project.
4. When the list of repositories appears, select your sample app repository.
5. Azure Pipelines analyzes your repository and in **Configure your pipeline** provides a list of potential templates. Choose the appropriate **function app** template for your language. If you don't see the correct template select **Show more**.
6. Select **Save and run**, then select **Commit directly to the main branch**, and then choose **Save and run** again.
7. A new run is started. Wait for the run to finish.

Example YAML build pipelines

The following language-specific pipelines can be used for building apps.

C#

You can use the following sample to create a YAML file to build a .NET app.

If you see errors when building your app, verify that the version of .NET that you use matches your Azure Functions version. For more information, see [Azure Functions runtime versions overview](#).

YAML

```
pool:  
  vmImage: 'windows-latest'  
steps:  
- script: |  
    dotnet restore  
    dotnet build --configuration Release  
- task: DotNetCoreCLI@2
```

```

inputs:
  command: publish
  arguments: '--configuration Release --output publish_output'
  projects: '*.*.csproj'
  publishWebProjects: false
  modifyOutputPath: false
  zipAfterPublish: false
- task: ArchiveFiles@2
  displayName: "Archive files"
  inputs:
    rootFolderOrFile: "$(System.DefaultWorkingDirectory)/publish_output"
    includeRootFolder: false
    archiveFile:
      "$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip"
- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish:
      '$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip'
    artifactName: 'drop'

```

Deploy your app

You'll deploy with the [Azure Function App Deploy](#) task. This task requires an [Azure service connection](#) as an input. An Azure service connection stores the credentials to connect from Azure Pipelines to Azure.

To deploy to Azure Functions, add the following snippet at the end of your `azure-pipelines.yml` file. The default `appType` is Windows. You can specify Linux by setting the `appType` to `functionAppLinux`.

YAML

```

trigger:
- main

variables:
  # Azure service connection established during pipeline creation
  azureSubscription: <Name of your Azure subscription>
  appName: <Name of the function app>
  # Agent VM image name
  vmImageName: 'ubuntu-latest'

- task: AzureFunctionApp@1 # Add this at the end of your file
  inputs:
    azureSubscription: <Azure service connection>
    appType: functionAppLinux # default is functionApp
    appName: $(appName)
    package: $(System.ArtifactsDirectory)/**/*.zip
    # Uncomment the next lines to deploy to a deployment slot

```

```
#Note that deployment slots is not supported for Linux Dynamic SKU
#deployToSlotOrASE: true
#resourceGroupName: '<Resource Group Name>'
#slotName: '<Slot name>'
```

The snippet assumes that the build steps in your YAML file produce the zip archive in the `$(System.ArtifactsDirectory)` folder on your agent.

Deploy a container

You can automatically deploy your code to Azure Functions as a custom container after every successful build. To learn more about containers, see [Create a function on Linux using a custom container](#).

Deploy with the Azure Function App for Container task

The simplest way to deploy to a container is to use the [Azure Function App on Container Deploy task](#).

To deploy, add the following snippet at the end of your YAML file:

YAML

```
trigger:
- main

variables:
# Container registry service connection established during pipeline creation
dockerRegistryServiceConnection: <Docker registry service connection>
imageRepository: <Name of your image repository>
containerRegistry: <Name of the Azure container registry>
dockerfilePath: '$(Build.SourcesDirectory)/Dockerfile'
tag: '$(Build.BuildId)'

# Agent VM image name
vmImageName: 'ubuntu-latest'

- task: AzureFunctionAppContainer@1 # Add this at the end of your file
inputs:
  azureSubscription: '<Azure service connection>'
  appName: '<Name of the function app>'
  imageName: $(containerRegistry)/$(imageRepository):$(tag)
```

The snippet pushes the Docker image to your Azure Container Registry. The [Azure Function App on Container Deploy task](#) pulls the appropriate Docker image

corresponding to the `BuildId` from the repository specified, and then deploys the image.

Deploy to a slot

You can configure your function app to have multiple slots. Slots allow you to safely deploy your app and test it before making it available to your customers.

The following YAML snippet shows how to deploy to a staging slot, and then swap to a production slot:

```
YAML

- task: AzureFunctionApp@1
  inputs:
    azureSubscription: <Azure service connection>
    appType: functionAppLinux
    appName: <Name of the Function app>
    package: $(System.ArtifactsDirectory)/**/*.zip
    deployToSlotOrASE: true
    resourceGroupName: <Name of the resource group>
    slotName: staging

- task: AzureAppServiceManage@0
  inputs:
    azureSubscription: <Azure service connection>
    WebAppName: <name of the Function app>
    ResourceGroupName: <name of resource group>
    SourceSlot: staging
    SwapWithProduction: true
```

Create a pipeline with Azure CLI

To create a build pipeline in Azure, use the [az functionapp devops-pipeline create command](#). The build pipeline is created to build and release any code changes that are made in your repo. The command generates a new YAML file that defines the build and release pipeline and then commits it to your repo. The prerequisites for this command depend on the location of your code.

- If your code is in GitHub:
 - You must have **write** permissions for your subscription.
 - You must be the project administrator in Azure DevOps.

- You must have permissions to create a GitHub personal access token (PAT) that has sufficient permissions. For more information, see [GitHub PAT permission requirements](#).
- You must have permissions to commit to the main branch in your GitHub repository so you can commit the autogenerated YAML file.
- If your code is in Azure Repos:
 - You must have **write** permissions for your subscription.
 - You must be the project administrator in Azure DevOps.

Next steps

- Review the [Azure Functions overview](#).
- Review the [Azure DevOps overview](#).

Continuous delivery by using GitHub Actions

Article • 03/16/2024

You can use a [GitHub Actions workflow](#) to define a workflow to automatically build and deploy code to your function app in Azure Functions.

A YAML file (.yml) that defines the workflow configuration is maintained in the `/.github/workflows/` path in your repository. This definition contains the actions and parameters that make up the workflow, which is specific to the development language of your functions. A GitHub Actions workflow for Functions performs the following tasks, regardless of language:

1. Set up the environment.
2. Build the code project.
3. Deploy the package to a function app in Azure.

The Azure Functions action handles the deployment to an existing function app in Azure.

You can create a workflow configuration file for your deployment manually. You can also generate the file from a set of language-specific templates in one of these ways:

- In the Azure portal
- Using the Azure CLI
- From your GitHub repository

If you don't want to create your YAML file by hand, select a different method at the top of the article.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- A GitHub account. If you don't have one, sign up for [free](#).
- A working function app hosted on Azure with source code in a GitHub repository.

Generate deployment credentials

Since GitHub Actions uses your publish profile to access your function app during deployment, you first need to get your publish profile and store it securely as a [GitHub secret](#).

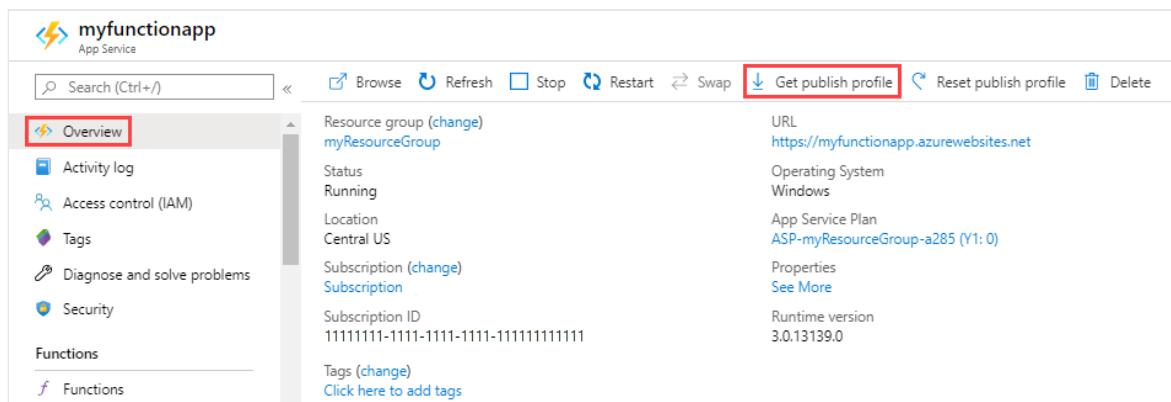
ⓘ Important

The publish profile is a valuable credential that allows access to Azure resources. Make sure you always transport and store it securely. In GitHub, the publish profile must only be stored in GitHub secrets.

Download your publish profile

To download the publishing profile of your function app:

1. Select the function app's **Overview** page, and then select **Get publish profile**.



2. Save and copy the contents of the file.

Add the GitHub secret

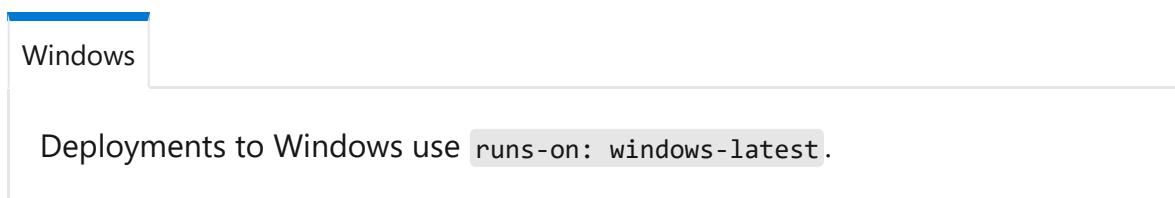
1. In [GitHub](#), go to your repository.
2. Go to **Settings**.
3. Select **Secrets and variables > Actions**.
4. Select **New repository secret**.
5. Add a new secret with the name `AZURE_FUNCTIONAPP_PUBLISH_PROFILE` and the value set to the contents of the publishing profile file.
6. Select **Add secret**.

GitHub can now authenticate to your function app in Azure.

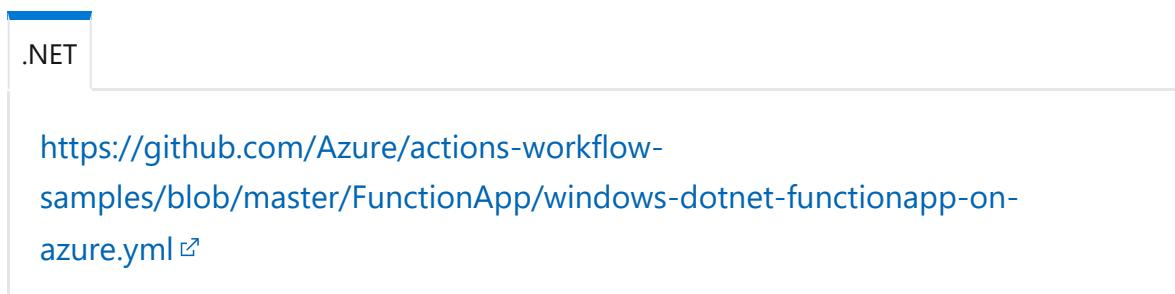
Create the workflow from a template

The best way to manually create a workflow configuration is to start from the officially supported template.

1. Choose either **Windows** or **Linux** to make sure that you get the template for the correct operating system.



2. Copy the language-specific template from the Azure Functions actions repository using the following link:



3. Update the `env.AZURE_FUNCTIONAPP_NAME` parameter with the name of your function app resource in Azure. You may optionally need to update the parameter that sets the language version used by your app, such as `DOTNET_VERSION` for C#.
4. Add this new YAML file in the `/.github/workflows/` path in your repository.

Update a workflow configuration

If for some reason, you need to update or change an existing workflow configuration, just navigate to the `/.github/workflows/` location in your repository, open the specific YAML file, make any needed changes, and then commit the updates to the repository.

Example: workflow configuration file

The following template example uses version 1 of the `functions-action` and a `publish profile` for authentication. The template depends on your chosen language and the operating system on which your function app is deployed:

Windows

If your function app runs on Linux, select [Linux](#).

.NET

yml

```
name: Deploy DotNet project to Azure Function App

on:
  [push]

env:
  AZURE_FUNCTIONAPP_NAME: 'your-app-name'    # set this to your function
app name on Azure
  AZURE_FUNCTIONAPP_PACKAGE_PATH: '.'        # set this to the path to
your function app project, defaults to the repository root
  DOTNET_VERSION: '6.0.x'                    # set this to the dotnet
version to use (e.g. '2.1.x', '3.1.x', '5.0.x')

jobs:
  build-and-deploy:
    runs-on: windows-latest
    environment: dev
    steps:
      - name: 'Checkout GitHub Action'
        uses: actions/checkout@v3

      - name: Setup DotNet ${{ env.DOTNET_VERSION }} Environment
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: ${{ env.DOTNET_VERSION }}

      - name: 'Resolve Project Dependencies Using Dotnet'
        shell: pwsh
        run:
          pushd './${{ env.AZURE_FUNCTIONAPP_PACKAGE_PATH }}'
          dotnet build --configuration Release --output ./output
          popd

      - name: 'Run Azure Functions Action'
        uses: Azure/functions-action@v1
        id: fa
        with:
          app-name: ${{ env.AZURE_FUNCTIONAPP_NAME }}
          package: '${{ env.AZURE_FUNCTIONAPP_PACKAGE_PATH }}/output'
          publish-profile: ${{ secrets.AZURE_FUNCTIONAPP_PUBLISH_PROFILE
}}}
```

Azure Functions action

The Azure Functions action (`Azure/azure-functions`) defines how your code is published to an existing function app in Azure, or to a specific slot in your app.

Parameters

The following parameters are most commonly used with this action:

[+] [Expand table](#)

Parameter	Explanation
<code>app-name</code>	(Mandatory) The name of your function app.
<code>slot-name</code>	(Optional) The name of a specific deployment slot you want to deploy to. The slot must already exist in your function app. When not specified, the code is deployed to the active slot.
<code>publish-profile</code>	(Optional) The name of the GitHub secret that contains your publish profile.

The following parameters are also supported, but are used only in specific cases:

[+] [Expand table](#)

Parameter	Explanation
<code>package</code>	(Optional) Sets a subpath in your repository from which to publish. By default, this value is set to <code>.</code> , which means all files and folders in the GitHub repository are deployed.
<code>respect-pom-xml</code>	(Optional) Used only for Java functions. Whether it's required for your app's deployment artifact to be derived from the pom.xml file. When deploying Java function apps, you should set this parameter to <code>true</code> and set <code>package</code> to <code>.</code> . By default, this parameter is set to <code>false</code> , which means that the <code>package</code> parameter must point to your app's artifact location, such as <code>./target/azure-functions/</code>
<code>respect-funcignore</code>	(Optional) Whether GitHub Actions honors your <code>.funcignore</code> file to exclude files and folders defined in it. Set this value to <code>true</code> when your repository has a <code>.funcignore</code> file and you want to use it to exclude paths and files, such as text editor configurations, <code>.vscode/</code> , or a Python virtual environment (<code>.venv/</code>). The default setting is <code>false</code> .
<code>scm-do-build-during-deployment</code>	(Optional) Whether the App Service deployment site (Kudu) performs predeployment operations. The deployment site for your function app can be found at <code>https://<APP_NAME>.scm.azurewebsites.net/</code> . Change this setting to

Parameter	Explanation
	<code>true</code> when you need to control the deployments in Kudu rather than resolving the dependencies in the GitHub Actions workflow. The default value is <code>false</code> . For more information, see the SCM_DO_BUILD_DURING_DEPLOYMENT setting.
<code>enable-oryx-build</code>	(Optional) Whether the Kudu deployment site resolves your project dependencies by using Oryx. Set to <code>true</code> when you want to use Oryx to resolve your project dependencies by using a remote build instead of the GitHub Actions workflow. When <code>true</code> , you should also set <code>scm-do-build-during-deployment</code> to <code>true</code> . The default value is <code>false</code> .

Considerations

Keep the following considerations in mind when using the Azure Functions action:

- When using GitHub Actions, the code is deployed to your function app using [Zip deployment for Azure Functions](#).
- The credentials required by GitHub to connection to Azure for deployment are stored as Secrets in your GitHub repository and accessed in the deployment as `secrets.<SECRET_NAME>`.
- The easiest way for GitHub Actions to authenticate with Azure Functions for deployment is by using a publish profile. You can also authenticate using a service principal. To learn more, see [this GitHub Actions repository ↗](#).
- The actions for setting up the environment and running a build are generated from the templates, and are language specific.
- The templates use `env` elements to define settings unique to your build and deployment.

Next steps

[Learn more about Azure and GitHub integration](#)

Zip deployment for Azure Functions

Article • 07/17/2024

This article describes how to deploy your function app project files to Azure from a .zip (compressed) file. You learn how to do a push deployment, both by using Azure CLI and by using the REST APIs. [Azure Functions Core Tools](#) also uses these deployment APIs when publishing a local project to Azure.

Zip deployment is also an easy way to run your functions from the deployment package. To learn more, see [Run your functions from a package file in Azure](#).

Azure Functions has the full range of continuous deployment and integration options that are provided by Azure App Service. For more information, see [Continuous deployment for Azure Functions](#).

To speed up development, you might find it easier to deploy your function app project files directly from a .zip file. The .zip deployment API takes the contents of a .zip file and extracts the contents into the `wwwroot` folder of your function app. This .zip file deployment uses the same Kudu service that powers continuous integration-based deployments, including:

- Deletion of files that were left over from earlier deployments.
- Deployment customization, including running deployment scripts.
- Deployment logs.
- Syncing function triggers in a [Consumption plan](#) function app.

For more information, see the [.zip deployment reference](#).

ⓘ Important

When you use .zip deployment, any files from an existing deployment that aren't found in the .zip file are deleted from your function app.

Deployment .zip file requirements

The zip archive you deploy must contain all of the files needed to run your function app. You can manually create a zip archive from the contents of a Functions project folder using built-in .zip compression functionality or third-party tools.

The archive must include the [host.json](#) file at the root of the extracted folder. The selected language stack for the function app creates additional requirements:

- .NET (isolated worker model)
- .NET (in-process model)
- Java
- JavaScript
- TypeScript
- PowerShell
- Python

ⓘ Important

For languages that generate compiled output for deployment, make sure to compress the contents of the output folder you plan to publish and not the entire project folder. When Functions extracts the contents of the zip archive, the `host.json` file must exist in the root of the package.

A zip deployment process extracts the zip archive's files and folders in the `wwwroot` directory. If you include the parent directory when creating the archive, the system will not find the files it expects to see in `wwwroot`.

Deploy by using Azure CLI

You can use Azure CLI to trigger a push deployment. Push deploy a .zip file to your function app by using the [az functionapp deployment source config-zip](#) command. To use this command, you must use Azure CLI version 2.0.21 or later. To see what Azure CLI version you are using, use the `az --version` command.

In the following command, replace the `<zip_file_path>` placeholder with the path to the location of your .zip file. Also, replace `<app_name>` with the unique name of your function app and replace `<resource_group>` with the name of your resource group.

Azure CLI

```
az functionapp deployment source config-zip -g <resource_group> -n \  
<app_name> --src <zip_file_path>
```

This command deploys project files from the downloaded .zip file to your function app in Azure. It then restarts the app. To view the list of deployments for this function app, you must use the REST APIs.

When you're using Azure CLI on your local computer, `<zip_file_path>` is the path to the .zip file on your computer. You can also run Azure CLI in [Azure Cloud Shell](#). When you

use Cloud Shell, you must first upload your deployment .zip file to the Azure Files account that's associated with your Cloud Shell. In that case, <zip_file_path> is the storage location that your Cloud Shell account uses. For more information, see [Persist files in Azure Cloud Shell](#).

Deploy ZIP file with REST APIs

You can use the [deployment service REST APIs](#) to deploy the .zip file to your app in Azure. To deploy, send a POST request to

`https://<app_name>.scm.azurewebsites.net/api/zipdeploy`. The POST request must contain the .zip file in the message body. The deployment credentials for your app are provided in the request by using HTTP BASIC authentication. For more information, see the [.zip push deployment reference](#).

For the HTTP BASIC authentication, you need your App Service deployment credentials. To see how to set your deployment credentials, see [Set and reset user-level credentials](#).

With cURL

The following example uses the cURL tool to deploy a .zip file. Replace the placeholders <deployment_user>, <zip_file_path>, and <app_name>. When prompted by cURL, type in the password.

Bash

```
curl -X POST -u <deployment_user> --data-binary "@<zip_file_path>"  
https://<app_name>.scm.azurewebsites.net/api/zipdeploy
```

This request triggers push deployment from the uploaded .zip file. You can review the current and past deployments by using the

`https://<app_name>.scm.azurewebsites.net/api/deployments` endpoint, as shown in the following cURL example. Again, replace <app_name> with the name of your app and <deployment_user> with the username of your deployment credentials.

Bash

```
curl -u <deployment_user>  
https://<app_name>.scm.azurewebsites.net/api/deployments
```

Asynchronous zip deployment

While deploying synchronously you may receive errors related to connection timeouts. Add `?isAsync=true` to the URL to deploy asynchronously. You will receive a response as soon as the zip file is uploaded with a `Location` header pointing to the pollable deployment status URL. When polling the URL provided in the `Location` header, you will receive a HTTP 202 (Accepted) response while the process is ongoing and a HTTP 200 (OK) response once the archive has been expanded and the deployment has completed successfully.

Microsoft Entra authentication

An alternative to using HTTP BASIC authentication for the zip deployment is to use a Microsoft Entra identity. Microsoft Entra identity may be needed if [HTTP BASIC authentication is disabled for the SCM site](#).

A valid Microsoft Entra access token for the user or service principal performing the deployment will be required. An access token can be retrieved using the Azure CLI's `az account get-access-token` command. The access token will be used in the Authentication header of the HTTP POST request.

Bash

```
curl -X POST \
--data-binary "@<zip_file_path>" \
-H "Authorization: Bearer <access_token>" \
"https://<app_name>.scm.azurewebsites.net/api/zipdeploy"
```

With PowerShell

The following example uses [Publish-AzWebapp](#) upload the .zip file. Replace the placeholders `<group-name>`, `<app-name>`, and `<zip-file-path>`.

PowerShell

```
Publish-AzWebapp -ResourceGroupName <group-name> -Name <app-name> - 
ArchivePath <zip-file-path>
```

This request triggers push deployment from the uploaded .zip file.

To review the current and past deployments, run the following commands. Again, replace the `<deployment-user>`, `<deployment-password>`, and `<app-name>` placeholders.

Bash

```

$username = "<deployment-user>"
$password = "<deployment-password>"
$apiUrl = "https://<app-name>.scm.azurewebsites.net/api/deployments"
$base64AuthInfo =
[Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{}:{}" -f
$username, $password)))
$userAgent = "powershell/1.0"
Invoke-RestMethod -Uri $apiUrl -Headers @{Authorization=("Basic {0}" -f
$base64AuthInfo)} -UserAgent $userAgent -Method GET

```

Deploy by using ARM Template

You can use [ZipDeploy ARM template extension](#) to push your .zip file to your function app.

Example ZipDeploy ARM Template

This template includes both a production and staging slot and deploys to one or the other. Typically, you would use this template to deploy to the staging slot and then swap to get your new zip package running on the production slot.

JSON

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "appServiceName": {
      "type": "string"
    },
    "deployToProduction": {
      "type": "bool",
      "defaultValue": false
    },
    "slot": {
      "type": "string",
      "defaultValue": "staging"
    },
    "packageUri": {
      "type": "secureString"
    }
  },
  "resources": [
    {
      "condition": "[parameters('deployToProduction')]",
      "type": "Microsoft.Web/sites/extensions",
      "apiVersion": "2021-02-01",
      "name": "[concat(parameters('appServiceName'), '/functionextension')]"
    }
  ]
}
```

```
        "name": "[format('{0}/ZipDeploy', parameters('appServiceName'))]",  
        "properties": {  
            "packageUri": "[parameters('packageUri')]",  
            "appOffline": true  
        }  
    },  
    {  
        "condition": "[not(parameters('deployToProduction'))]",  
        "type": "Microsoft.Web/sites/slots/extensions",  
        "apiVersion": "2021-02-01",  
        "name": "[format('{0}/{1}/ZipDeploy', parameters('appServiceName'),  
parameters('slot'))]",  
        "properties": {  
            "packageUri": "[parameters('packageUri')]",  
            "appOffline": true  
        }  
    }  
]
```

For the initial deployment, you would deploy directly to the production slot. For more information, see [Slot deployments](#).

Run functions from the deployment package

You can also choose to run your functions directly from the deployment package file. This method skips the deployment step of copying files from the package to the `wwwroot` directory of your function app. Instead, the package file is mounted by the Functions runtime, and the contents of the `wwwroot` directory become read-only.

Zip deployment integrates with this feature, which you can enable by setting the function app setting `WEBSITE_RUN_FROM_PACKAGE` to a value of `1`. For more information, see [Run your functions from a deployment package file](#).

Deployment customization

The deployment process assumes that the .zip file that you push contains a ready-to-run app. By default, no customizations are run. To enable the same build processes that you get with continuous integration, add the following to your application settings:

```
SCM_DO_BUILD_DURING_DEPLOYMENT=true
```

When you use .zip push deployment, this setting is `false` by default. The default is `true` for continuous integration deployments. When set to `true`, your deployment-related settings are used during deployment. You can configure these settings either as app

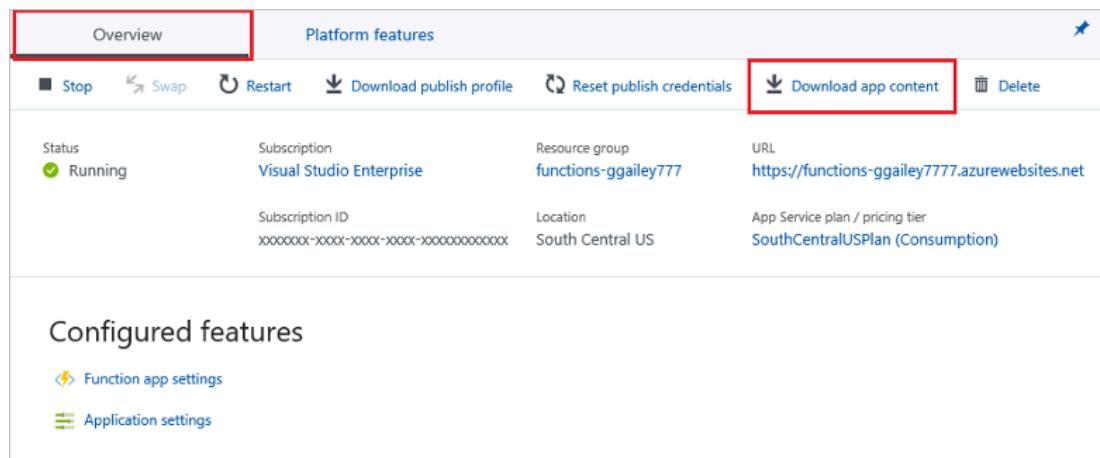
settings or in a .deployment configuration file that's located in the root of your .zip file. For more information, see [Repository and deployment-related settings](#) in the deployment reference.

Download your function app files

If you created your functions by using the editor in the Azure portal, you can download your existing function app project as a .zip file in one of these ways:

- From the Azure portal:

1. Sign in to the [Azure portal](#), and then go to your function app.
2. On the Overview tab, select **Download app content**. Select your download options, and then select **Download**.



The screenshot shows the Azure portal's Overview tab for a function app named 'functions-ggailey777'. The 'Overview' tab is selected. At the top, there are several buttons: Stop, Swap, Restart, Download publish profile, Reset publish credentials, Download app content (which is highlighted with a red box), and Delete. Below the buttons, there are sections for Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggailey777), and URL (https://functions-ggailey777.azurewebsites.net). There are also sections for Subscription ID and Location. In the bottom section, titled 'Configured features', there are links for Function app settings and Application settings.

The downloaded .zip file is in the correct format to be republished to your function app by using .zip push deployment. The portal download can also add the files needed to open your function app directly in Visual Studio.

- Using REST APIs:

Use the following deployment GET API to download the files from your <function_app> project:

HTTP
<code>https://<function_app>.scm.azurewebsites.net/api/zip/site/wwwroot/</code>

Including /site/wwwroot/ makes sure your zip file includes only the function app project files and not the entire site. If you are not already signed in to Azure, you will be asked to do so.

You can also download a .zip file from a GitHub repository. When you download a GitHub repository as a .zip file, GitHub adds an extra folder level for the branch. This extra folder level means that you can't deploy the .zip file directly as you downloaded it from GitHub. If you're using a GitHub repository to maintain your function app, you should use [continuous integration](#) to deploy your app.

Next steps

[Run your functions from a package file in Azure](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Run your functions from a package file in Azure

Article • 07/17/2024

In Azure, you can run your functions directly from a deployment package file in your function app. The other option is to deploy your files in the `c:\home\site\wwwroot` (Windows) or `/home/site/wwwroot` (Linux) directory of your function app.

This article describes the benefits of running your functions from a package. It also shows how to enable this functionality in your function app.

Benefits of running from a package file

There are several benefits to running functions from a package file:

- Reduces the risk of file copy locking issues.
- Can be deployed to a production app (with restart).
- Verifies the files that are running in your app.
- Improves the performance of [Azure Resource Manager deployments](#).
- Reduces cold-start times, particularly for JavaScript functions with large npm package trees.

For more information, see [this announcement](#).

Enable functions to run from a package

To enable your function app to run from a package, add a `WEBSITE_RUN_FROM_PACKAGE` app setting to your function app. The `WEBSITE_RUN_FROM_PACKAGE` app setting can have one of the following values:

[] [Expand table](#)

Value	Description
1	Indicates that the function app runs from a local package file deployed in the <code>c:\home\data\SitePackages</code> (Windows) or <code>/home/data/SitePackages</code> (Linux) folder of your function app.
<code><URL></code>	Sets a URL that is the remote location of the specific package file you want to run. Required for functions apps running on Linux in a Consumption plan.

The following table indicates the recommended `WEBSITE_RUN_FROM_PACKAGE` values for deployment to a specific operating system and hosting plan:

[Expand table](#)

Hosting plan	Windows	Linux
Consumption	1 is highly recommended.	Only <URL> is supported.
Premium	1 is recommended.	1 is recommended.
Dedicated	1 is recommended.	1 is recommended.

General considerations

- The package file must be .zip formatted. Tar and gzip formats aren't supported.
- [Zip deployment](#) is recommended.
- When deploying your function app to Windows, you should set `WEBSITE_RUN_FROM_PACKAGE` to 1 and publish with zip deployment.
- When you run from a package, the `wwwroot` folder is read-only and you receive an error if you write files to this directory. Files are also read-only in the Azure portal.
- The maximum size for a deployment package file is 1 GB.
- You can't use the local cache when running from a deployment package.
- If your project needs to use remote build, don't use the `WEBSITE_RUN_FROM_PACKAGE` app setting. Instead, add the `SCM_DO_BUILD_DURING_DEPLOYMENT=true` deployment customization app setting. For Linux, also add the `ENABLE_ORYX_BUILD=true` setting. For more information, see [Remote build](#).

Note

The `WEBSITE_RUN_FROM_PACKAGE` app setting does not work with MSDeploy as described in [MSDeploy VS. ZipDeploy](#). You will receive an error during deployment, such as `ARM-MSDeploy Deploy Failed`. To resolve this error, change `/MSDeploy` to `/ZipDeploy`.

Add the `WEBSITE_RUN_FROM_PACKAGE` setting

There are several ways that you can add, update, and delete function app settings:

- [In the Azure portal](#).
- [By using the Azure CLI](#).

- By using Azure PowerShell.

Changes to function app settings require your function app to be restarted.

Creating the zip archive

The zip archive you deploy must contain all of the files needed to run your function app. You can manually create a zip archive from the contents of a Functions project folder using built-in .zip compression functionality or third-party tools.

The archive must include the [host.json](#) file at the root of the extracted folder. The selected language stack for the function app creates additional requirements:

- [.NET \(isolated worker model\)](#)
- [.NET \(in-process model\)](#)
- [Java](#)
- [JavaScript](#)
- [TypeScript](#)
- [PowerShell](#)
- [Python](#)

Important

For languages that generate compiled output for deployment, make sure to compress the contents of the output folder you plan to publish and not the entire project folder. When Functions extracts the contents of the zip archive, the `host.json` file must exist in the root of the package.

Use WEBSITE_RUN_FROM_PACKAGE = 1

This section provides information about how to run your function app from a local package file.

Considerations for deploying from an on-site package

- Using an on-site package is the recommended option for running from the deployment package, except when running on Linux hosted in a Consumption plan.
- [Zip deployment](#) is the recommended way to upload a deployment package to your site.

- When not using zip deployment, make sure the `c:\home\data\SitePackages` (Windows) or `/home/data/SitePackages` (Linux) folder has a file named `packagename.txt`. This file contains only the name, without any whitespace, of the package file in this folder that's currently running.

Integration with zip deployment

Zip deployment is a feature of Azure App Service that lets you deploy your function app project to the `wwwroot` directory. The project is packaged as a .zip deployment file. The same APIs can be used to deploy your package to the `c:\home\data\SitePackages` (Windows) or `/home/data/SitePackages` (Linux) folder.

When you set the `WEBSITE_RUN_FROM_PACKAGE` app setting value to `1`, the zip deployment APIs copy your package to the `c:\home\data\SitePackages` (Windows) or `/home/data/SitePackages` (Linux) folder instead of extracting the files to `c:\home\site\wwwroot` (Windows) or `/home/site/wwwroot` (Linux). It also creates the `packagename.txt` file. After your function app is automatically restarted, the package is mounted to `wwwroot` as a read-only filesystem. For more information about zip deployment, see [Zip deployment for Azure Functions](#).

ⓘ Note

When a deployment occurs, a restart of the function app is triggered. Function executions currently running during the deploy are terminated. For information about how to write stateless and defensive functions, see [Write functions to be stateless](#).

Use `WEBSITE_RUN_FROM_PACKAGE = URL`

This section provides information about how to run your function app from a package deployed to a URL endpoint. This option is the only one supported for running from a Linux-hosted package with a Consumption plan.

Considerations for deploying from a URL

- Function apps running on Windows experience a slight increase in [cold-start time](#) when the application package is deployed to a URL endpoint via `WEBSITE_RUN_FROM_PACKAGE = <URL>`.

- When you specify a URL, you must also [manually sync triggers](#) after you publish an updated package.
- The Functions runtime must have permissions to access the package URL.
- Don't deploy your package to Azure Blob Storage as a public blob. Instead, use a private container with a [shared access signature \(SAS\)](#) or [use a managed identity](#) to enable the Functions runtime to access the package.
- You must maintain any SAS URLs used for deployment. When an SAS expires, the package can no longer be deployed. In this case, you must generate a new SAS and update the setting in your function app. You can eliminate this management burden by [using a managed identity](#).
- When running on a Premium plan, make sure to [eliminate cold starts](#).
- When you're running on a Dedicated plan, ensure you enable [Always On](#).
- You can use [Azure Storage Explorer](#) to upload package files to blob containers in your storage account.

Manually uploading a package to Blob Storage

To deploy a zipped package when using the URL option, you must create a .zip compressed deployment package and upload it to the destination. The following procedure deploys to a container in Blob Storage:

1. Create a .zip package for your project using the utility of your choice.
2. In the [Azure portal](#), search for your storage account name or browse for it in the storage accounts list.
3. In the storage account, select **Containers** under **Data storage**.
4. Select **+ Container** to create a new Blob Storage container in your account.
5. In the **New container** page, provide a **Name** (for example, *deployments*), ensure the **Anonymous access level** is **Private**, and then select **Create**.
6. Select the container you created, select **Upload**, browse to the location of the .zip file you created with your project, and then select **Upload**.
7. After the upload completes, choose your uploaded blob file, and copy the URL. If you aren't [using a managed identity](#), you might need to generate a SAS URL.
8. Search for your function app or browse for it in the **Function App** page.
9. In your function app, expand **Settings**, and then select **Environment variables**.
10. In the **App settings** tab, select **+ Add**.

11. Enter the value `WEBSITE_RUN_FROM_PACKAGE` for the **Name**, and paste the URL of your package in Blob Storage for the **Value**.

12. Select **Apply**, and then select **Apply** and **Confirm** to save the setting and restart the function app.

Now you can run your function in Azure to verify that deployment of the deployment package .zip file was successful.

Fetch a package from Azure Blob Storage using a managed identity

You can configure Azure Blob Storage to [authorize requests with Microsoft Entra ID](#). This configuration means that instead of generating a SAS key with an expiration, you can instead rely on the application's [managed identity](#). By default, the app's system-assigned identity is used. If you wish to specify a user-assigned identity, you can set the `WEBSITE_RUN_FROM_PACKAGE_BLOB_MI_RESOURCE_ID` app setting to the resource ID of that identity. The setting can also accept `SystemAssigned` as a value, which is equivalent to omitting the setting.

To enable the package to be fetched using the identity:

1. Ensure that the blob is [configured for private access](#).
2. Grant the identity the [Storage Blob Data Reader](#) role with scope over the package blob. See [Assign an Azure role for access to blob data](#) for details on creating the role assignment.
3. Set the `WEBSITE_RUN_FROM_PACKAGE` application setting to the blob URL of the package. This URL is usually of the form `https://{{storage-account-name}}.blob.core.windows.net/{{container-name}}/{{path-to-package}}` or similar.

Related content

- [Continuous deployment for Azure Functions](#)

Feedback

Was this page helpful?



Yes



No

Automate resource deployment for your function app in Azure Functions

Article • 08/22/2024

You can use a Bicep file or an Azure Resource Manager (ARM) template to automate the process of deploying your function app. During the deployment, you can use existing Azure resources or create new ones. Automation help's you with these scenarios:

- Integrating your resource deployments with your source code in Azure Pipelines and GitHub Actions-based deployments.
- Restoring a function app and related resources from a backup.
- Deploying an app topology multiple times.

This article shows you how to automate the creation of resources and deployment for Azure Functions. Depending on the [triggers and bindings](#) used by your functions, you might need to deploy other resources, which is outside of the scope of this article.

The template code required depends on the desired hosting options for your function app. This article supports the following hosting options:

[+] [Expand table](#)

Hosting option	Deployment type	To learn more, see...
Azure Functions Consumption plan	Code-only	Consumption plan
Azure Functions Flex Consumption plan	Code-only	Flex Consumption plan
Azure Functions Elastic Premium plan	Code Container	Premium plan
Azure Functions Dedicated (App Service) plan	Code Container	Dedicated plan
Azure Container Apps	Container-only	Container Apps hosting of Azure Functions
Azure Arc	Code Container	App Service, Functions, and Logic Apps on Azure Arc (Preview)

 **Important**

The [Flex Consumption plan](#) is currently in preview.

When using this article, keep these considerations in mind:

- There's no canonical way to structure an ARM template.
- A Bicep deployment can be modularized into multiple Bicep files.
- This article assumes that you have a basic understanding of [creating Bicep files](#) or [authoring Azure Resource Manager templates](#).
- Examples are shown as individual sections for specific resources. For a broad set of complete Bicep file and ARM template examples, see [these function app deployment examples](#).

Required resources

You must create or configure these resources for an Azure Functions-hosted deployment:

[+] [Expand table](#)

Resource	Requirement	Syntax and properties reference
A storage account	Required	Microsoft.Storage/storageAccounts
An Application Insights component	Recommended	Microsoft.Insights/components [*]
A function app	Required	Microsoft.Web/sites

^{*}If you don't already have a Log Analytics Workspace that can be used by your Application Insights instance, you also need to create this resource.

When you deploy multiple resources in a single Bicep file or ARM template, the order in which resources are created is important. This requirement is a result of dependencies between resources. For such dependencies, make sure to use the `dependsOn` element to define the dependency in the dependent resource. For more information, see either [Define the order for deploying resources in ARM templates](#) or [Resource dependencies in Bicep](#).

Prerequisites

- The examples are designed to execute in the context of an existing resource group.

- Both Application Insights and storage logs require you to have an existing [Azure Log Analytics workspace](#). Workspaces can be shared between services, and as a rule of thumb you should create a workspace in each geographic region to improve performance. For an example of how to create a Log Analytics workspace, see [Create a Log Analytics workspace](#). You can find the fully qualified workspace resource ID in a workspace page in the [Azure portal](#) under **Settings > Properties > Resource ID**.

Create storage account

All function apps require an Azure storage account. You need a general purpose account that supports blobs, tables, queues, and files. For more information, see [Azure Functions storage account requirements](#).

Important

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

This example section creates a Standard general purpose v2 storage account:

Bicep

Bicep

```
resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' =  
{  
    name: storageAccountName  
    location: location  
    kind: 'StorageV2'  
    sku: {  
        name: 'Standard_LRS'  
    }  
    properties: {  
        supportsHttpsTrafficOnly: true  
        defaultToOAuthAuthentication: true  
        allowBlobPublicAccess: false  
    }  
}
```

For more context, see the complete [main.bicep](#) file in the templates repository.

You need to set the connection string of this storage account as the `AzureWebJobsStorage` app setting, which Functions requires. The templates in this article construct this connection string value based on the created storage account, which is a best practice. For more information, see [Application configuration](#).

Enable storage logs

Because the storage account is used for important function app data, you should monitor the account for modification of that content. To monitor your storage account, you need to configure Azure Monitor resource logs for Azure Storage. In this example section, a Log Analytics workspace named `myLogAnalytics` is used as the destination for these logs.



```
resource blobService
'Microsoft.Storage/storageAccounts/blobServices@2021-09-01' existing = {
  name:'default'
  parent:storageAccountName
}

resource storageDataPlaneLogs
'Microsoft.Insights/diagnosticSettings@2021-05-01-preview' = {
  name: '${storageAccountName}-logs'
  scope: blobService
  properties: {
    workspaceId: myLogAnalytics.id
    logs: [
      {
        category: 'StorageWrite'
        enabled: true
      }
    ]
    metrics: [
      {
        category: 'Transaction'
        enabled: true
      }
    ]
  }
}
```

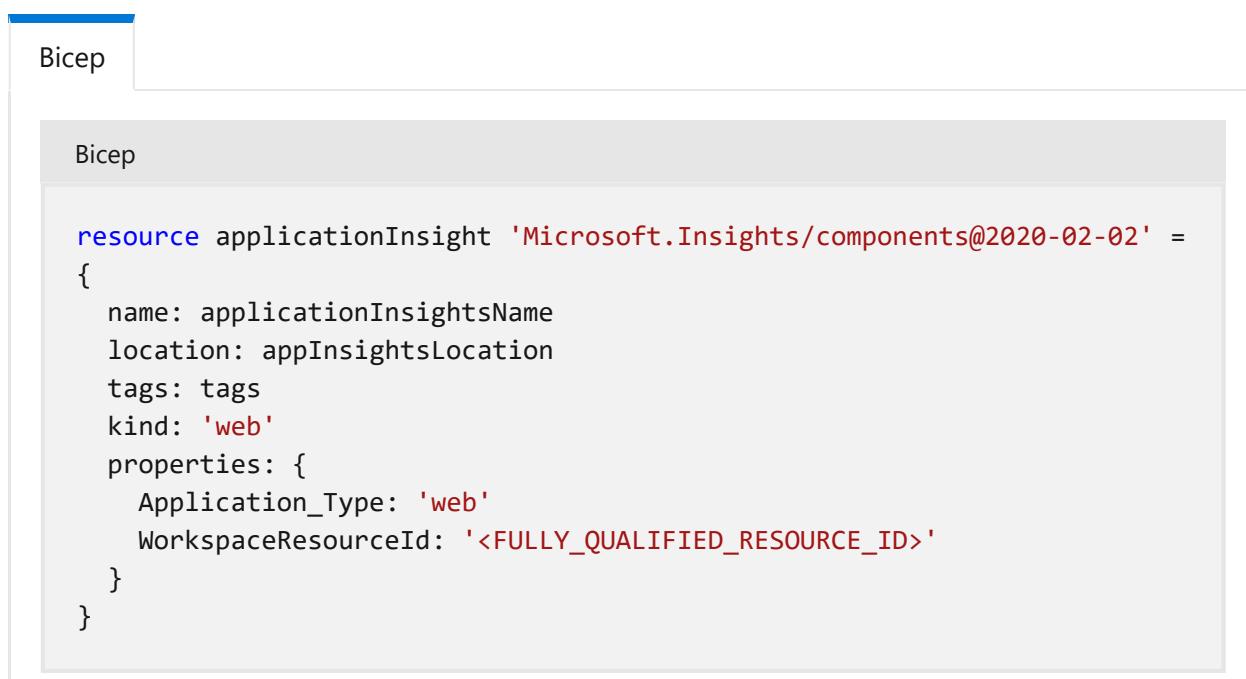
This same workspace can be used for the Application Insights resource defined later. For more information, including how to work with these logs, see [Monitoring Azure Storage](#).

Create Application Insights

You should be using Application Insights for monitoring your function app executions. Application Insights now requires an Azure Log Analytics workspace, which can be shared. These examples assume you're using an existing workspace and have the fully qualified resource ID for the workspace. For more information, see [Azure Log Analytics workspace](#).

In this example section, the Application Insights resource is defined with the type

`Microsoft.Insights/components` and the kind `web`:



```
resource applicationInsight 'Microsoft.Insights/components@2020-02-02' =
{
    name: applicationInsightsName
    location: appInsightsLocation
    tags: tags
    kind: 'web'
    properties: {
        Application_Type: 'web'
        WorkspaceResourceId: '<FULLY_QUALIFIED_RESOURCE_ID>'
    }
}
```

For more context, see the complete [main.bicep](#) file in the templates repository.

The connection must be provided to the function app using the `APPLICATIONINSIGHTS_CONNECTION_STRING` application setting. For more information, see [Application configuration](#).

The examples in this article obtain the connection string value for the created instance. Older versions might instead use `APPINSIGHTS_INSTRUMENTATIONKEY` to set the instrumentation key, which is no longer recommended.

Create the hosting plan

You don't need to explicitly define a Consumption hosting plan resource. When you skip this resource definition, a plan is automatically either created or selected on a per-region basis when you create the function app resource itself.

You can explicitly define a Consumption plan as a special type of `serverfarm` resource, which you specify using the value `Dynamic` for the `computeMode` and `sku` properties. This example section shows you how to explicitly define a consumption plan. The way that you define a hosting plan depends on whether your function app runs on Windows or on Linux.

Windows

```
Bicep
```

```
resource hostingPlan 'Microsoft.Web/serverfarms@2022-03-01' = {
    name: hostingPlanName
    location: location
    sku: {
        name: 'Y1'
        tier: 'Dynamic'
        size: 'Y1'
        family: 'Y'
        capacity: 0
    }
    properties: {
        computeMode: 'Dynamic'
    }
}
```

For more context, see the complete [main.bicep](#) file in the templates repository.

Create the function app

The function app resource is defined by a resource of type `Microsoft.Web/sites` and `kind` that includes `functionapp`, at a minimum.

The way that you define a function app resource depends on whether you're hosting on Linux or on Windows:

Windows

```
For a list of application settings required when running on Windows, see Application configuration. For a sample Bicep file/Azure Resource Manager template, see the function app hosted on Windows in a Consumption plan template.
```

(!) Note

If you choose to optionally define your Consumption plan, you must set the `serverFarmId` property on the app so that it points to the resource ID of the plan. Make sure that the function app has a `dependsOn` setting that also references the plan. If you didn't explicitly define a plan, one gets created for you.

Windows

Bicep

```
resource functionAppName_resource 'Microsoft.Web/sites@2022-03-01' = {
    name: functionAppName
    location: location
    kind: 'functionapp'
    properties: {
        serverFarmId: hostingPlanName.id
        siteConfig: {
            appSettings: [
                {
                    name: 'APPLICATIONINSIGHTS_CONNECTION_STRING'
                    value: applicationInsightsName.properties.ConnectionString
                }
                {
                    name: 'AzureWebJobsStorage'
                    value:
'DefaultEndpointsProtocol=https;AccountName=${storageAccountName};EndpointSuffix=${environment().suffixes.storage};AccountKey=${storageAccount.listKeys().keys[0].value}'
                }
                {
                    name: 'WEBSITE_CONTENTAZUREFILECONNECTIONSTRING'
                    value:
'DefaultEndpointsProtocol=https;AccountName=${storageAccountName};EndpointSuffix=${environment().suffixes.storage};AccountKey=${storageAccount.listKeys().keys[0].value}'
                }
                {
                    name: 'WEBSITE_CONTENTSHARE'
                    value: toLower(functionAppName)
                }
                {
                    name: 'FUNCTIONS_EXTENSION_VERSION'
                    value: '~4'
                }
                {
                    name: 'FUNCTIONS_WORKER_RUNTIME'
                    value: 'node'
                }
            ]
        }
    }
}
```

```
        name: 'WEBSITE_NODE_DEFAULT_VERSION'
        value: '~14'
    }
]
}
}
```

For a complete end-to-end example, see this [main.bicep file](#).

Deployment sources

Your Bicep file or ARM template can optionally also define a deployment for your function code using a [zip deployment package](#).

To successfully deploy your application by using Azure Resource Manager, it's important to understand how resources are deployed in Azure. In most examples, top-level configurations are applied by using `siteConfig`. It's important to set these configurations at a top level, because they convey information to the Functions runtime and deployment engine. Top-level information is required before the child `sourcecontrols/web` resource is applied. Although it's possible to configure these settings in the child-level `config/appSettings` resource, in some cases your function app must be deployed *before* `config/appSettings` is applied.

Zip deployment package

Zip deployment is a recommended way to deploy your function app code. By default, functions that use zip deployment run in the deployment package itself. For more information, including the requirements for a deployment package, see [Zip deployment for Azure Functions](#). When using resource deployment automation, you can reference the .zip deployment package in your Bicep or ARM template.

To use zip deployment in your template, set the `WEBSITE_RUN_FROM_PACKAGE` setting in the app to `1` and include the `/zipDeploy` resource definition.

For a Consumption plan on Linux, instead set the URI of the deployment package directly in the `WEBSITE_RUN_FROM_PACKAGE` setting, as shown in [this example template](#).

This example adds a zip deployment source to an existing app:

Bicep

Bicep

```
@description('The name of the function app.')
param functionName string

@description('The location into which the resources should be
deployed.')
param location string = resourceGroup().location

@description('The zip content url.')
param packageUri string

resource functionAppName_ZipDeploy 'Microsoft.Web/sites/extensions@2021-
02-01' = {
    name: '${functionAppName}/ZipDeploy'
    location: location
    properties: {
        packageUri: packageUri
    }
}
```

Keep the following things in mind when including zip deployment resources in your template:

- Consumption plans on Linux don't support `WEBSITE_RUN_FROM_PACKAGE = 1`. You must instead set the URI of the deployment package directly in the `WEBSITE_RUN_FROM_PACKAGE` setting. For more information, see [Function app hosted on Linux in a Consumption plan](#).
- The `packageUri` must be a location that can be accessed by Functions. Consider using Azure blob storage with a shared access signature (SAS). After the SAS expires, Functions can no longer access the share for deployments. When you regenerate your SAS, remember to update the `WEBSITE_RUN_FROM_PACKAGE` setting with the new URI value.
- When setting `WEBSITE_RUN_FROM_PACKAGE` to a URI, you must [manually sync triggers](#).
- Make sure to always set all required application settings in the `appSettings` collection when adding or updating settings. Existing settings not explicitly set are removed by the update. For more information, see [Application configuration](#).
- Functions doesn't support Web Deploy (msdeploy) for package deployments. You must instead use zip deployment in your deployment pipelines and automation. For more information, see [Zip deployment for Azure Functions](#).

Remote builds

The deployment process assumes that the .zip file that you use or a zip deployment contains a ready-to-run app. This means that by default no customizations are run.

There are scenarios that require you to rebuild your app remotely. One such example is when you need to include Linux-specific packages in Python or Node.js apps that you developed on a Windows computer. In this case, you can configure Functions to perform a remote build on your code after the zip deployment.

The way that you request a remote build depends on the operating system to which you're deploying:

Windows

When an app is deployed to Windows, language-specific commands (like `dotnet restore` for C# apps or `npm install` for Node.js apps) are run.

To enable the same build processes that you get with continuous integration, add `SCM_DO_BUILD_DURING_DEPLOYMENT=true` to your application settings in your deployment code and remove the `WEBSITE_RUN_FROM_PACKAGE` entirely.

Application configuration

Functions provides the following options for configuring your function app in Azure:

[\[+\] Expand table](#)

Configuration	Microsoft.Web/sites property
Site settings	<code>siteConfig</code>
Application settings	<code>siteConfig.appSettings</code> collection

These site settings are required on the `siteConfig` property:

Windows

- `netFrameworkVersion`

These application settings are required (or recommended) for a specific operating system and hosting option:

Windows

- [APPLICATIONINSIGHTS_CONNECTION_STRING](#)
- [AzureWebJobsStorage](#)
- [FUNCTIONS_EXTENSION_VERSION](#)
- [FUNCTIONS_WORKER_RUNTIME](#)
- [WEBSITE_CONTENTAZUREFILECONNECTIONSTRING](#)
- [WEBSITE_CONTENTSHARE](#)
- [WEBSITE_RUN_FROM_PACKAGE](#) (recommended)
- [WEBSITE_NODE_DEFAULT_VERSION](#) (Node.js-only)

Keep these considerations in mind when working with site and application settings using Bicep files or ARM templates:

- There are important considerations for when you should set `WEBSITE_CONTENTSHARE` in an automated deployment. For detailed guidance, see the [WEBSITE_CONTENTSHARE](#) reference.
- You should always define your application settings as a `siteConfig/appSettings` collection of the `Microsoft.Web/sites` resource being created, as is done in the examples in this article. This definition guarantees the settings your function app needs to run are available on initial startup.
- When adding or updating application settings using templates, make sure that you include all existing settings with the update. You must do this because the underlying update REST API calls replace the entire `/config/appsettings` resource. If you remove the existing settings, your function app won't run. To programmatically update individual application settings, you can instead use the Azure CLI, Azure PowerShell, or the Azure portal to make these changes. For more information, see [Work with application settings](#).

Slot deployments

Functions lets you deploy different versions of your code to unique endpoints in your function app. This option makes it easier to develop, validate, and deploy functions updates without impacting functions running in production. Deployment slots is a

feature of Azure App Service. The number of slots available depends on your hosting plan. For more information, see [Azure Functions deployment slots](#) functions.

A slot resource is defined in the same way as a function app resource (`Microsoft.Web/sites`), but instead you use the `Microsoft.Web/sites/slots` resource identifier. For an example deployment (in both Bicep and ARM templates) that creates both a production and a staging slot in a Premium plan, see [Azure Function App with a Deployment Slot ↗](#).

To learn about how to swap slots by using templates, see [Automate with Resource Manager templates](#).

Keep the following considerations in mind when working with slot deployments:

- Don't explicitly set the `WEBSITE_CONTENTSHARE` setting in the deployment slot definition. This setting is generated for you when the app is created in the deployment slot.
- When you swap slots, some application settings are considered "sticky," in that they stay with the slot and not with the code being swapped. You can define such a *slot setting* by including `"slotSetting":true` in the specific application setting definition in your template. For more information, see [Manage settings](#).

Function access keys

Host-level [function access keys](#) are defined as Azure resources. This means that you can create and manage host keys in your ARM templates and Bicep files. A host key is defined as a resource of type `Microsoft.Web/sites/host/functionKeys`. This example creates a host-level access key named `my_custom_key` when the function app is created:

Bicep

```
resource functionKey 'Microsoft.Web/sites/host/functionKeys@2022-09-01'
= {
  name: '${parameters('name')}/default/my_custom_key'
  properties: {
    name: 'my_custom_key'
  }
  dependsOn: [
    resourceId('Microsoft.Web/Sites', parameters('name'))
```

```
]  
}
```

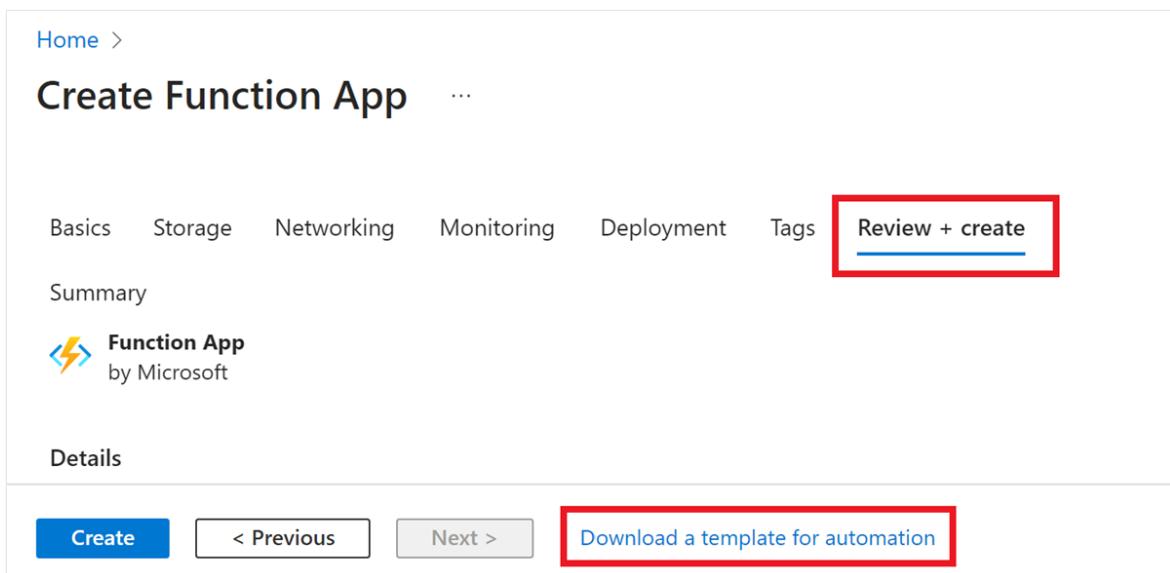
In this example, the `name` parameter is the name of the new function app. You must include a `dependsOn` setting to guarantee that the key is created with the new function app. Finally, the `properties` object of the host key can also include a `value` property that can be used to set a specific key.

When you don't set the `value` property, Functions automatically generates a new key for you when the resource is created, which is recommended. To learn more about access keys, including security best practices for working with access keys, see [Work with access keys in Azure Functions](#).

Create your template

Experts with Bicep or ARM templates can manually code their deployments using a simple text editor. For the rest of us, there are several ways to make the development process easier:

- **Visual Studio Code:** There are extensions available to help you work with both [Bicep files](#) and [ARM templates](#). You can use these tools to help make sure that your code is correct, and they provide some [basic validation](#).
- **Azure portal:** When you [create your function app and related resources in the portal](#), the final **Review + create** screen has a [Download a template for automation](#) link.



This link shows you the ARM template generated based on the options you chose in portal. This template can seem a bit complex when you're creating a function

app with many new resources. However, it can provide a good reference for how your ARM template might look.

Validate your template

When you manually create your deployment template file, it's important to validate your template before deployment. All deployment methods validate your template syntax and raise a `validation failed` error message as shown in the following JSON formatted example:

JSON

```
{"error": {"code": "InvalidTemplate", "message": "Deployment template validation failed: 'The resource 'Microsoft.Web/sites/func-xyz' is not defined in the template. Please see https://aka.ms/arm-template for usage details.'.", "additionalInfo": [{"type": "TemplateViolation", "info": {"lineNumber": 0, "linePosition": 0, "path": ""}}]}}
```

The following methods can be used to validate your template before deployment:

Azure Pipelines

The following [Azure resource group deployment v2 task](#) with `deploymentMode: 'Validation'` instructs Azure Pipelines to validate the template.

yml

```
- task: AzureResourceManagerTemplateDeployment@3
  inputs:
    deploymentScope: 'Resource Group'
    subscriptionId: # Required subscription ID
    action: 'Create Or Update Resource Group'
    resourceGroupName: # Required resource group name
    location: # Required when action == Create Or Update Resource Group
    templateLocation: 'Linked artifact'
    csmFile: # Required when TemplateLocation == Linked Artifact
    csmParametersFile: # Optional
    deploymentMode: 'Validation'
```

You can also create a test resource group to find [preflight](#) and [deployment](#) errors.

Deploy your template

You can use any of the following ways to deploy your Bicep file and template:

Bicep

- [Azure CLI](#)
- [PowerShell](#)

Deploy to Azure button

ⓘ Note

This method doesn't support deploying Bicep files currently.

Replace `<url-encoded-path-to-azuredeploy-json>` with a [URL-encoded ↗](#) version of the raw path of your `azuredeploy.json` file in GitHub.

Here's an example that uses markdown:

markdown

```
[![Deploy to Azure](https://azuredploy.net/deploybutton.png)]  
(https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-  
to-azuredploy-json>)
```

Here's an example that uses HTML:

HTML

```
<a href="https://portal.azure.com/#create/Microsoft.Template/uri/<url-  
encoded-path-to-azuredploy-json>" target="_blank"></a>
```

Deploy using PowerShell

The following PowerShell commands create a resource group and deploy a Bicep file or ARM template that creates a function app with its required resources. To run locally, you must have [Azure PowerShell](#) installed. Run [Connect-AzAccount](#) to sign in.

Bicep

PowerShell

```
# Register Resource Providers if they're not already registered
Register-AzResourceProvider -ProviderNamespace "microsoft.web"
Register-AzResourceProvider -ProviderNamespace "microsoft.storage"

# Create a resource group for the function app
New-AzResourceGroup -Name "MyResourceGroup" -Location 'West Europe'

# Deploy the template
New-AzResourceGroupDeployment -ResourceGroupName "MyResourceGroup" -
TemplateFile main.bicep -Verbose
```

To test out this deployment, you can use a [template like this one](#) that creates a function app on Windows in a Consumption plan.

Next steps

Learn more about how to develop and configure Azure Functions.

- [Azure Functions developer reference](#)
- [How to configure Azure function app settings](#)
- [Create your first Azure function](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Tutorial: Deploy to Azure Functions using Jenkins

Article • 05/30/2023

ⓘ Important

Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article [Jenkins plug-ins for Azure](#).

[Azure Functions](#) is a serverless compute service. Using Azure Functions, you can run code on-demand without provisioning or managing infrastructure. This tutorial shows how to deploy a Java function to Azure Functions using the Azure Functions plug-in.

Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Jenkins server:** If you don't have a Jenkins server installed, refer to the article, [Create a Jenkins server on Azure](#).

View the source code

The source code used for this tutorial is located in the [Visual Studio China GitHub repo](#).

Create a Java function

To create a Java function with the Java runtime stack, use either the [Azure portal](#) or the [Azure CLI](#).

The following steps show how to create a Java function using the Azure CLI:

1. Create a resource group, replacing the <resource_group> placeholder with your resource group name.

Azure CLI

```
az group create --name <resource_group> --location eastus
```

2. Create an Azure storage account, replacing the placeholders with the appropriate values.

Azure CLI

```
az storage account create --name <storage_account> --location eastus --resource-group <resource_group> --sku Standard_LRS
```

3. Create the test function app, replacing the placeholders with the appropriate values.

Azure CLI

```
az functionapp create --resource-group <resource_group> --runtime java --consumption-plan-location eastus --name <function_app> --storage-account <storage_account> --functions-version 2
```

Prepare Jenkins server

The following steps explain how to prepare the Jenkins server:

1. Deploy a [Jenkins server](#) on Azure. If you don't already have an instance of the Jenkins server installed, the article, [Create a Jenkins server on Azure](#) guides you through the process.
2. Sign in to the Jenkins instance with SSH.
3. On the Jenkins instance, install Az CLI, version 2.0.67 or higher.
4. Install maven using the following command:

Bash

```
sudo apt install -y maven
```

5. On the Jenkins instance, install the [Azure Functions Core Tools](#) by issuing the following commands at a terminal prompt:

Bash

```

curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor
> microsoft.gpg
sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg
sudo sh -c 'echo "deb [arch=amd64]
https://packages.microsoft.com/repos/microsoft-ubuntu-$(lsb_release -cs)-prod $(lsb_release -cs) main" >
/etc/apt/sources.list.d/dotnetdev.list'
cat /etc/apt/sources.list.d/dotnetdev.list
sudo apt-get update
sudo apt-get install azure-functions-core-tools-3

```

6. Jenkins needs an Azure service principal to authenticate and access Azure resources. Refer to the [Deploy to Azure App Service](#) for step-by-step instructions.
7. Make sure the [Credentials plug-in](#) is installed.
 - a. From the menu, select **Manage Jenkins**.
 - b. Under **System Configuration**, select **Manage plug-in**.
 - c. Select the **Installed** tab.
 - d. In the **filter** field, enter `credentials`.
 - e. Verify that the **Credentials plug-in** is installed. If not, you'll need to install it from the **Available** tab.

Enabled		Name	Version	Previously installed version	Uninstall
<input checked="" type="checkbox"/>	Credentials Binding Plugin		1.24		Uninstall
<input checked="" type="checkbox"/>	Credentials plugin	This plugin allows you to store credentials in Jenkins.	2.3.14		Uninstall
<input checked="" type="checkbox"/>	Git	This plugin integrates Git with Jenkins.	4.5.1		Uninstall
<input checked="" type="checkbox"/>	Git client plugin	Utility plugin for Git support in Jenkins	3.6.0		Uninstall
<input checked="" type="checkbox"/>	Github plugin	This plugin integrates GitHub to Jenkins.	1.32.0		Uninstall
<input checked="" type="checkbox"/>	JSch dependency plugin	Jenkins plugin that brings the JSch library as a plugin dependency, and provides an SSHAuthenticatorFactory for using JSch with the ssh-credentials plugin.	0.1.55.2		Uninstall

8. From the menu, select **Manage Jenkins**.
9. Under **Security**, select **Manage Credentials**.
10. Under **Credentials**, select **(global)**.
11. From the menu, select **Add Credentials**.
12. Enter the following values for your **Microsoft Azure service principal**:

- **Kind:** Select the value: *Username with password*.
- **Username:** Specify the `appId` of the service principal created.
- **Password:** Specify the `password` (secret) of the service principal.
- **ID:** Specify the credential identifier, such as `azuresp`.

13. Select OK.

Fork the sample GitHub repo

1. [Sign in to the GitHub repo for the odd or even sample app ↗](#).
2. In the upper-right corner in GitHub, choose **Fork**.
3. Follow the prompts to select your GitHub account and finish forking.

Create a Jenkins Pipeline

In this section, you create the [Jenkins Pipeline ↗](#).

1. In the Jenkins dashboard, create a Pipeline.
2. Enable **Prepare an environment for the run**.
3. In the **Pipeline->Definition** section, select **Pipeline script from SCM**.
4. Enter your GitHub fork's URL and script path ("doc/resources/jenkins/JenkinsFile") to use in the [JenkinsFile example ↗](#).

Node.js

```
node {
  withEnv(['AZURE_SUBSCRIPTION_ID=99999999-9999-9999-9999-999999999999',
           'AZURE_TENANT_ID=99999999-9999-9999-9999-999999999999']) {
    stage('Init') {
      cleanWs()
      checkout scm
    }

    stage('Build') {
      sh 'mvn clean package'
    }

    stage('Publish') {
      def RESOURCE_GROUP = '<resource_group>'
      def FUNC_NAME = '<function_app>'
      // login Azure
      withCredentials([usernamePassword(credentialsId: 'azuresp',
                                         username: 'azuresp',
                                         password: 'azuresp')])
      azFuncDelete(functionName: FUNC_NAME, resourceGroup: RESOURCE_GROUP)
      azFuncCreateOrUpdate(functionName: FUNC_NAME,
                           resourceGroup: RESOURCE_GROUP,
                           zipFilePath: 'target/*.zip')
    }
  }
}
```

```
passwordVariable: 'AZURE_CLIENT_SECRET', usernameVariable:  
'AZURE_CLIENT_ID')) {  
    sh '''  
        az login --service-principal -u $AZURE_CLIENT_ID -p  
$AZURE_CLIENT_SECRET -t $AZURE_TENANT_ID  
        az account set -s $AZURE_SUBSCRIPTION_ID  
    '''  
}  
}  
}  
}
```

Build and deploy

It's now time to run the Jenkins job.

1. First, obtain the authorization key via the instructions in the [Azure Functions HTTP triggers and bindings](#) article.
2. In your browser, enter the app's URL. Replace the placeholders with the appropriate values and specify a numeric value for <input_number> as input for the Java function.

```
https://<function_app>.azurewebsites.net/api/HttpTrigger-Java?code=<authorization_key>&number=<input_number>
```

3. You'll see results similar to the following example output (where an odd number - 365 - was used as a test):

Output

```
The number 365 is Odd.
```

Clean up resources

If you're not going to continue to use this application, delete the resources you created with the following step:

Azure CLI

```
az group delete -y --no-wait -n <resource_group>
```

Next steps

[Azure Functions](#)

Manage your function app

Article • 07/18/2024

In Azure Functions, a function app provides the execution context for your individual functions. Function app behaviors apply to all functions hosted by a given function app. All functions in a function app must be of the same [language](#).

Individual functions in a function app are deployed together and are scaled together. All functions in the same function app share resources, per instance, as the function app scales.

Connection strings, environment variables, and other application settings are defined separately for each function app. Any data that must be shared between function apps should be stored externally in a persisted store.

Get started in the Azure portal

Note

Because of limitations on editing function code in the [Azure portal](#), you should develop your functions locally and publish your code project to a function app in Azure. For more information, see [Development limitations in the Azure portal](#)

To view the app settings in your function app, follow these steps:

1. Sign in to the [Azure portal](#) using your Azure account. Search for your function app and select it.
2. In the left pane of your function app, expand **Settings**, select **Environment variables**, and then select the **App settings** tab.

The screenshot shows the Azure portal interface for managing environment variables of a function app named "function-app-1". The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Better Together (preview), Resource visualizer, Functions, Deployment, Performance, Settings, Environment variables (which is selected and highlighted with a red box), Configuration, Authentication, and Application Insights. The main content area is titled "Environment variables" and contains two tabs: "App settings" (selected) and "Connection strings". The "App settings" tab has a search bar, an "Add" button, and buttons for Refresh, Show values, Advanced edit, and Pull reference values. A table lists several environment variables with their names, values, deployment slot settings, sources, and delete actions. The variables listed are: APPLICATIONINSIGHTS_CONNECTION... (Value: Show value, Source: App Service), AzureWebJobsStorage (Value: Show value, Source: App Service), FUNCTIONS_EXTENSION_VERSION (Value: Show value, Source: App Service), FUNCTIONS_WORKER_RUNTIME (Value: Show value, Source: App Service), WEBSITE_CONTENTAZUREFILECONNE... (Value: Show value, Source: App Service), WEBSITE_CONTENTSHARE (Value: Show value, Source: App Service), and WEBSITE_USE_PLACEHOLDER_DOTNE... (Value: Show value, Source: App Service). At the bottom of the table are "Apply" and "Discard" buttons, and a "Send us your feedback" link.

Work with application settings

In addition to the predefined app settings used by Azure Functions, you can create any number of app settings, as required by your function code. For more information, see [App settings reference for Azure Functions](#).

These settings are stored encrypted. For more information, see [App settings security](#).

You can manage app settings from the [Azure portal](#), and by using the [Azure CLI](#) and [Azure PowerShell](#). You can also manage app settings from [Visual Studio Code](#) and from [Visual Studio](#).

The screenshot shows the Azure portal interface for managing app settings of a function app. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Better Together (preview), Resource visualizer, Functions, Deployment, Performance, Settings, Environment variables (selected and highlighted with a red box), Configuration, Authentication, and Application Insights. The main content area is titled "App settings" and contains a table listing environment variables. The variables listed are: APPINSIGHTS_INSTRUMENTATIONKEY (Value: Show value, Source: App Service), AZURE_FUNCTIONS_VERSION (Value: Show value, Source: App Service), and WEBSITE_CONTENTAZUREFILECONNE... (Value: Show value, Source: App Service). At the bottom of the table are "Apply" and "Discard" buttons, and a "Send us your feedback" link.

To view your app settings, see [Get started in the Azure portal](#).

The **App settings** tab maintains settings that are used by your function app:

1. To see the values of the app settings, select **Show values**.
2. To add a setting, select **+ Add**, and then enter the **Name** and **Value** of the new key-value pair.

Name	Value	Deployment slot setting	Source	Delete
APPLICATIONINSIGHTS_CONNECTION_STRING	Show value		App Service	Delete
AzureWebJobsStorage	Show value		App Service	Delete
FUNCTIONS_EXTENSION_VERSION	Show value		App Service	Delete
FUNCTIONS_WORKER_RUNTIME	Show value		App Service	Delete
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING	Show value		App Service	Delete
WEBSITE_CONTENTSHARE	Show value		App Service	Delete
WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED	Show value		App Service	Delete

Apply Discard [Send us your feedback](#)

Use application settings

The function app settings values can also be read in your code as environment variables. For more information, see the Environment variables section of these language-specific reference articles:

- [C# precompiled](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

When you develop a function app locally, you must maintain local copies of these values in the `local.settings.json` project file. For more information, see [Local settings file](#).

FTPS deployment settings

Azure Functions supports deploying project code to your function app by using FTPS. Because this deployment method requires you to [sync triggers](#), it isn't recommended. To securely transfer project files, always use FTPS and not FTP.

To get the credentials required for FTPS deployment, use one of these methods:

Azure portal

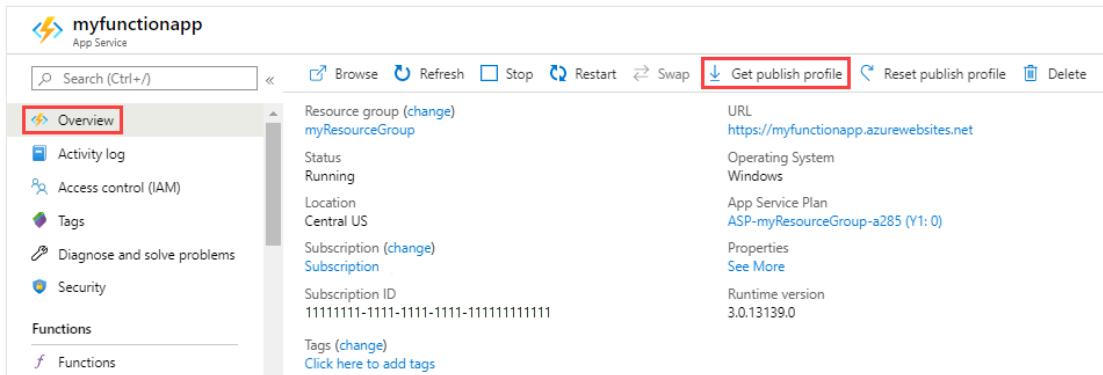
You can get the FTPS publishing credentials in the Azure portal by downloading the publishing profile for your function app.

ⓘ Important

The publishing profile contains important security credentials. Always secure the downloaded file on your local computer.

To download the publishing profile of your function app:

1. Select the function app's **Overview** page, and then select **Get publish profile**.



The screenshot shows the Azure portal interface for an App Service named 'myfunctionapp'. The left sidebar has 'Overview' selected. The main content area displays various details about the function app, including its resource group ('myResourceGroup'), status ('Running'), location ('Central US'), subscription information, and tags. At the top right, there are several buttons: 'Browse', 'Refresh', 'Stop', 'Restart', 'Swap', 'Get publish profile' (which is highlighted with a red box), 'Reset publish profile', and 'Delete'.

2. Save and copy the contents of the file.

3. In the file, locate the `publishProfile` element with the attribute

`publishMethod="FTP"`. In this element, the `publishUrl`, `userName`, and `userPWD` attributes contain the target URL and credentials for FTPS publishing.

Hosting plan type

When you create a function app, you also create a hosting plan in which the app runs. A plan can have one or more function apps. The functionality, scaling, and pricing of your functions depend on the type of plan. For more information, see [Azure Functions hosting options](#).

You can determine the type of plan being used by your function app from the Azure portal, or by using the Azure CLI or Azure PowerShell APIs.

The following values indicate the plan type:

[+] Expand table

Plan type	Azure portal	Azure CLI/PowerShell
Consumption	Consumption	Dynamic

Plan type	Azure portal	Azure CLI/PowerShell
Premium	ElasticPremium	ElasticPremium
Dedicated (App Service)	Various	Various

Azure portal

- To determine the type of plan used by your function app, see the [App Service Plan](#) in the [Overview](#) page of the function app in the [Azure portal](#).

- To see the pricing tier, select the name of the **App Service Plan**, and then select **Settings > Properties** from the left pane.

Plan migration

You can migrate a function app between a Consumption plan and a Premium plan on Windows. When migrating between plans, keep in mind the following considerations:

- Direct migration to a Dedicated (App Service) plan isn't supported.
- Migration isn't supported on Linux.
- The source plan and the target plan must be in the same resource group and geographical region. For more information, see [Move an app to another App Service plan](#).
- The specific CLI commands depend on the direction of the migration.
- Downtime in your function executions occurs as the function app is migrated between plans.
- State and other app-specific content is maintained, because the same Azure Files share is used by the app both before and after migration.

You can migrate your plan using these tools:

Azure portal

You can use the [Azure portal](#) to switch to a different plan.

Choose the direction of the migration for your app on Windows.

Consumption-to-Premium

1. In the Azure portal, navigate to your Consumption plan app and choose **Change App Service plan** under **App Service plan**.
2. Select **Premium** under **Plan type**, create a new Premium plan, and select **OK**.

For more information, see [Move an app to another App Service plan](#).

Development limitations in the Azure portal

Consider these limitations when you develop your functions in the [Azure portal](#):

- In-portal editing is supported only for functions that were created or last modified in the Azure portal.
- In-portal editing is supported only for JavaScript, PowerShell, Python, and C# Script functions.
- In-portal editing isn't supported in the preview release of the [Flex Consumption plan](#).
- When you deploy code to a function app from outside the Azure portal, you can no longer edit any of the code for that function app in the portal. In this case, just continue using [local development](#).
- For compiled C# functions and Java functions, you can create the function app and related resources in the portal. However, you must create the functions code project locally and then publish it to Azure.

When possible, develop your functions locally and publish your code project to a function app in Azure. For more information, see [Code and test Azure Functions locally](#).

Manually install extensions

C# class library functions can include the NuGet packages for [binding extensions](#) directly in the class library project. For other non-.NET languages and C# script, you should [use extension bundles](#). If you must manually install extensions, you can do so by [using Azure Functions Core Tools](#) locally. If you can't use extension bundles and are only able to work in the portal, you need to use [Advanced Tools \(Kudu\)](#) to manually create the extensions.csproj file directly in the site. Make sure to first remove the `extensionBundle` element from the `host.json` file.

This same process works for any other file you need to add to your app.

Important

When possible, don't edit files directly in your function app in Azure. We recommend [downloading your app files locally](#), using [Core Tools to install extensions](#) and other packages, validating your changes, and then [republishing your app using Core Tools](#) or one of the other [supported deployment methods](#).

The Functions editor built into the Azure portal lets you update your function code and configuration files directly in the portal:

1. Select your function app, then under **Functions**, select **Functions**.
2. Choose your function and select **Code + test** under **Developer**.
3. Choose your file to edit and select **Save** when you finish.

Files in the root of the app, such as function.proj or extensions.csproj need to be created and edited by using the [Advanced Tools \(Kudu\)](#):

1. Select your function app, expand **Development tools**, and then select **Advanced tools > Go**.
2. If prompted, sign in to the Source Control Manager (SCM) site with your Azure credentials.
3. From the **Debug console** menu, choose **CMD**.
4. Navigate to `.\site\wwwroot`, select the plus (+) button at the top, and select **New file**.
5. Give the file a name, such as `extensions.csproj`, and then press Enter.
6. Select the edit button next to the new file, add or update code in the file, and then select **Save**.

7. For a project file like `extensions.csproj`, run the following command to rebuild the extensions project:

```
Bash
```

```
dotnet build extensions.csproj
```

Platform features

Function apps run in the Azure App Service platform, which maintains them. As such, your function apps have access to most of the features of Azure's core web hosting platform. When you use the [Azure portal](#), the left pane is where you access the many features of the App Service platform that you can use in your function apps.

The following matrix indicates Azure portal feature support by hosting plan and operating system:

[Expand table](#)

Feature	Consumption plan	Flex Consumption plan	Premium plan	Dedicated plan
Advanced tools (Kudu)	Windows: ✓ Linux: X	X	✓	✓
App Service editor	Windows: ✓ Linux: X	X	Windows: ✓ Linux: X	Windows: ✓ Linux: X
Backups	X	X	X	✓
Console	Windows: command-line Linux: X	X	Windows: command-line Linux: SSH	Windows: command-line Linux: SSH

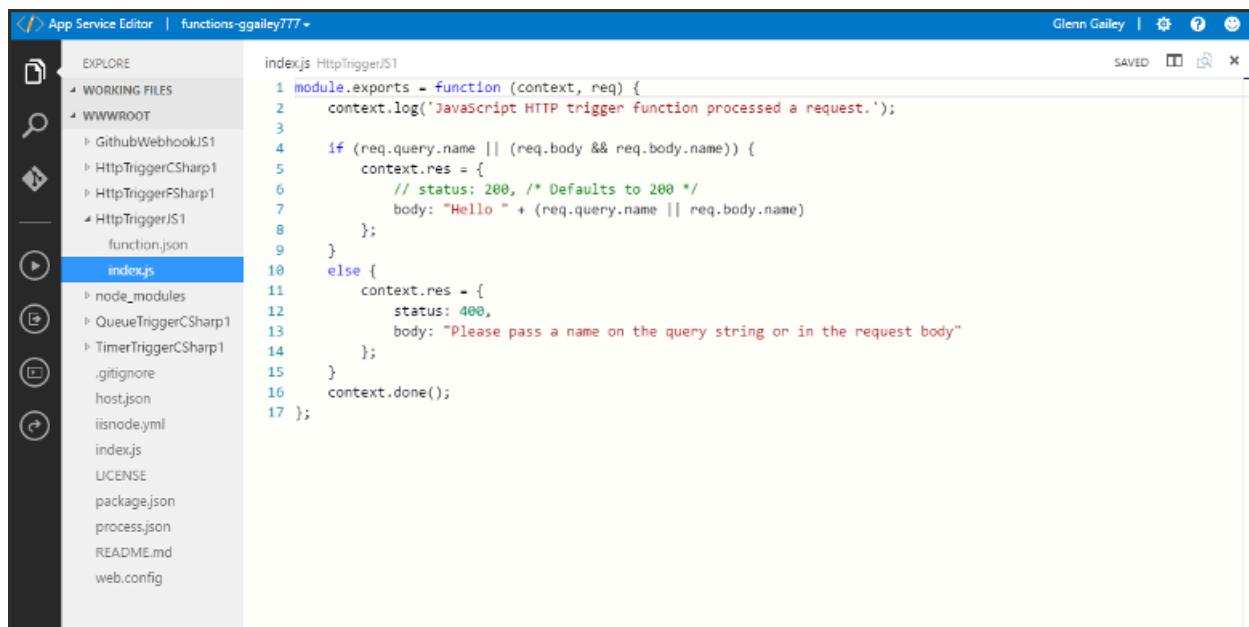
The rest of this article focuses on the following features in the portal that are useful for your function apps:

- [App Service editor](#)
- [Console](#)
- [Advanced tools \(Kudu\)](#)
- [Deployment options](#)
- [CORS](#)
- [Authentication](#)

For more information about how to work with App Service settings, see [Configure Azure App Service Settings](#).

App Service editor

The App Service editor is an advanced in-portal editor that you can use to modify JSON configuration files and code files alike. Choosing this option launches a separate browser tab with a basic editor. This editor enables you to integrate with the Git repository, run and debug code, and modify function app settings. This editor provides an enhanced development environment for your functions compared with the built-in function editor.



The screenshot shows the Azure App Service Editor interface. On the left, there's a sidebar with icons for file operations like New, Open, Save, and Delete. Below that is a tree view of the project structure:

- EXPLORE
- WORKING FILES (selected)
- WWWROOT
 - GithubWebhookJS1
 - HttpTriggerCSharp1
 - HttpTriggerFSharp1
 - HttpTriggerJS1
 - function.json
 - index.js (highlighted)
 - node_modules
 - QueueTriggerCSharp1
 - TimerTriggerCSharp1
 - .gitignore
 - host.json
 - iisnode.yml
 - index.js
 - LICENSE
 - package.json
 - process.json
 - README.md
 - web.config

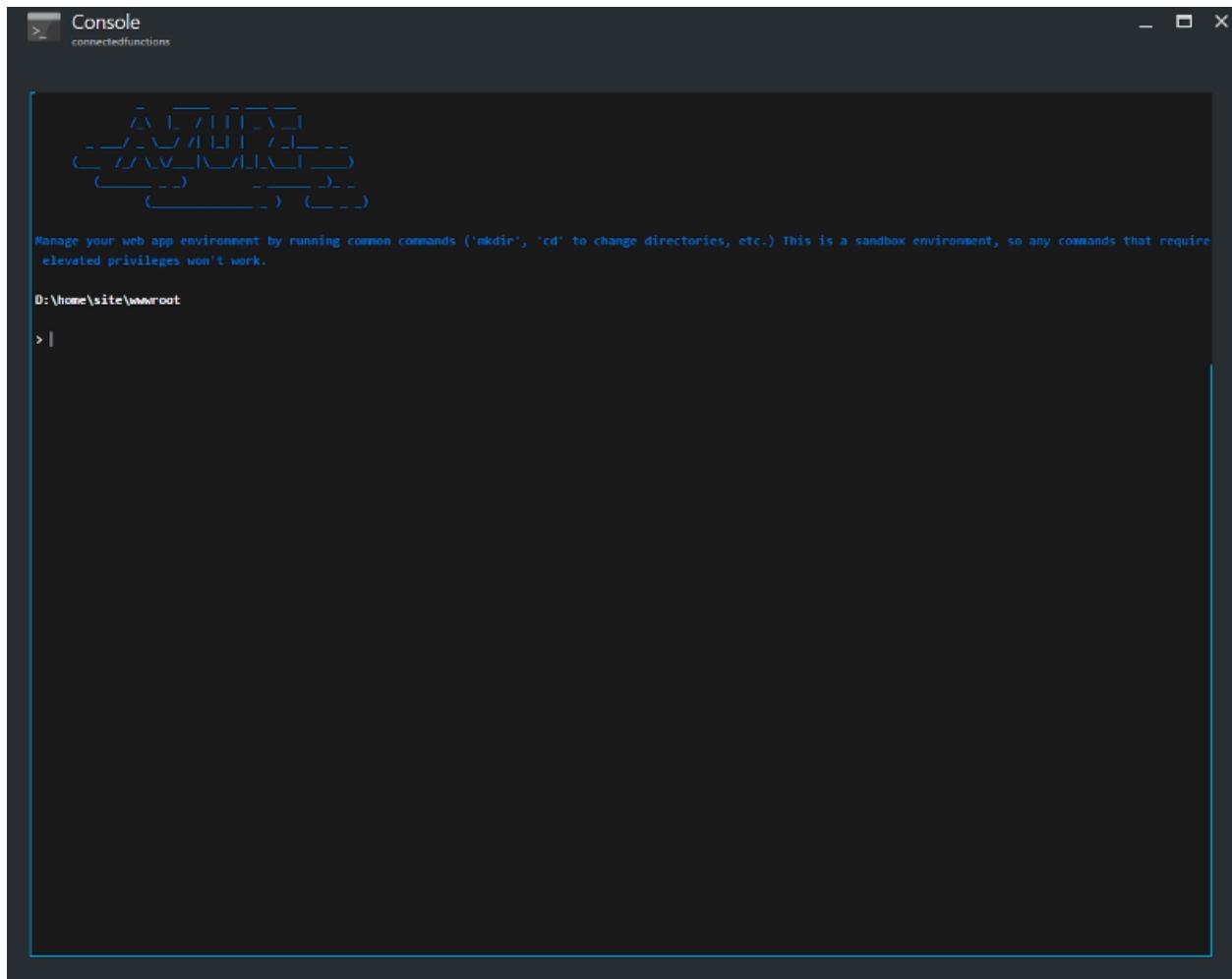
The main area displays the content of the selected file, index.js:

```
1 module.exports = function (context, req) {
2     context.log('JavaScript HTTP trigger function processed a request.');
3
4     if (req.query.name || (req.body && req.body.name)) {
5         context.res = {
6             // status: 200, /* Defaults to 200 */
7             body: "Hello " + (req.query.name || req.body.name)
8         };
9     }
10    else {
11        context.res = {
12            status: 400,
13            body: "Please pass a name on the query string or in the request body"
14        };
15    }
16    context.done();
17 }
```

We recommend that you consider developing your functions on your local computer. When you develop locally and publish to Azure, your project files are read-only in the Azure portal. For more information, see [Code and test Azure Functions locally](#).

Console

The in-portal console is an ideal developer tool when you prefer to interact with your function app from the command line. Common commands include directory and file creation and navigation, as well as executing batch files and scripts.



The screenshot shows a terminal window titled "Console" with the path "connectedfunctions". It features a blue decorative header at the top. The main area contains a message about managing the web app environment via common commands like "mkdir" and "cd", noting it's a sandbox environment where elevated privileges won't work. The current directory is listed as "D:\home\site\wwwroot". A command prompt is shown with a cursor at the end of the line.

When developing locally, we recommend using the [Azure Functions Core Tools](#) and the [Azure CLI](#).

Advanced tools (Kudu)

The advanced tools for App Service (also known as Kudu) provide access to advanced administrative features of your function app. From Kudu, you manage system information, app settings, environment variables, site extensions, HTTP headers, and server variables. You can also launch **Kudu** by browsing to the SCM endpoint for your function app, for example: <https://<myfunctionapp>.scm.azurewebsites.net/>.

The screenshot shows the Kudu interface. At the top, there's a navigation bar with links for Kudu, Environment, Debug console, Process explorer, Tools, and Site extensions. Below the navigation bar is a file explorer window titled '/ | 3 items'. It lists three items: 'data' (modified 10/26/2016, 7:15:25 PM), 'LogFiles' (modified 10/26/2016, 7:15:17 PM), and 'site' (modified 10/26/2016, 7:15:17 PM). To the right of the file explorer is a large blacked-out area representing a terminal or remote execution console. At the top right of this area, there's a link labeled 'Use old console'. The bottom left of the blacked-out area shows some command-line output from a Windows CMD process.

Name	Modified	Size
data	10/26/2016, 7:15:25 PM	
LogFiles	10/26/2016, 7:15:17 PM	
site	10/26/2016, 7:15:17 PM	

Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new CMD process.
Type 'cls' to clear the console

Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

D:\home>

Deployment Center

When you use a source control solution to develop and maintain your functions code, Deployment Center lets you build and deploy from source control. Your project is built and deployed to Azure when you make updates. For more information, see [Deployment technologies in Azure Functions](#).

Cross-origin resource sharing

To prevent malicious code execution on the client, modern browsers block requests from web applications to resources running in a separate domain. [Cross-origin resource sharing \(CORS\)](#) lets an `Access-Control-Allow-Origin` header declare which origins are allowed to call endpoints on your function app.

Azure portal

When you configure the **Allowed origins** list for your function app, the `Access-Control-Allow-Origin` header is automatically added to all responses from HTTP endpoints in your function app.

The screenshot shows the CORS configuration page for a function. At the top, there's a header with a save and discard button. Below the header is a blue-bordered information box containing text about CORS. Underneath the box, there's a section titled "ALLOWED ORIGINS" with three listed entries: "https://functions.azure.com", "https://functions-staging.azure.com", and "https://functions-next.azure.com". Each entry has a "..." button to its right.

CORS
ConnectedFunctions

Save Discard

ALLOWED ORIGINS

https://functions.azure.com ...
https://functions-staging.azure.com ...
https://functions-next.azure.com ...

If there's another domain entry, the wildcard (*) is ignored.

Authentication

When functions use an HTTP trigger, you can require calls to first be authenticated. App Service supports Microsoft Entra authentication and sign-in with social providers, such as Facebook, Microsoft, and Twitter. For information about configuring specific authentication providers, see [Azure App Service authentication overview](#).

The screenshot shows two side-by-side windows from the Azure portal.

Left Window: Authentication / Authorization

- App Service Authentication:** Set to **On**.
- Action to take when request is not authenticated:** Set to **Log in with Azure Active Directory**.
- Authentication Providers:**
 - Azure Active Directory:** Not Configured.
 - Facebook:** Not Configured.
 - Google:** Not Configured.
 - Twitter:** Not Configured.
 - Microsoft Account:** Not Configured.
- Advanced Settings:** Token Store set to **On**.

Right Window: Microsoft Account Authentication Settings

These settings allow users to sign in with Microsoft Account. Click here to learn more.

SCOPE	DESCRIPTION
wl.basic	Read access to a user's basic profile info. Also enables read access to...
wl.offline_access	The ability of an app to read and update a user's info at any time.
wl.signin	Single sign-in behavior.
wl.birthday	Read access to a user's birthday info including birth day, month, and...
wl.calendars	Read access to a user's calendars and events.
wl.calendars_update	Read and write access to a user's calendars and events.
wl.contacts_birthday	Read access to the birth day and birth month of a user's contacts.
wl.contacts_create	Creation of new contacts in the user's address book.
wl.contacts_calendars	Read access to a user's calendars and events.
wl.contacts_photos	Read access to a user's albums, photos, videos, and audio, and their...
wl.contacts_skydrive	Read access to Microsoft OneDrive files that other users have shared...
wlemails	Read access to a user's personal, preferred, and business email address...
wlevents_create	Read access to Microsoft OneDrive files that other users have shared...
wlimap	Read and write access to a user's email using IMAP, and send access...
wl.phone_numbers	Read access to a user's personal, business, and mobile phone number...

OK

Related content

- Configure an App Service app
- Continuous deployment for Azure Functions

Feedback

Was this page helpful?

Yes

No

Provide product feedback | Get help at Microsoft Q&A

Create and manage function apps in the Flex Consumption plan

Article • 08/21/2024

This article shows you how to create function apps hosted in the [Flex Consumption plan](#) in Azure Functions. It also shows you how to manage certain features of a Flex Consumption plan hosted app.

Function app resources are language-specific. Make sure to choose your preferred code development language at the beginning of the article.

ⓘ Important

The [Flex Consumption plan](#) is currently in preview.

Prerequisites

- An Azure account with an active subscription. If you don't already have one, you can [create an account for free](#).
- **Azure CLI:** used to create and manage resources in Azure. When using the Azure CLI on your local computer, make sure to use version 2.60.0, or a later version. You can also use [Azure Cloud Shell](#), which has the correct Azure CLI version.
- **Visual Studio Code:** used to create and develop apps, create Azure resources, and deploy code projects to Azure. When using Visual Studio Code, make sure to also install the latest [Azure Functions extension](#). You can also install the [Azure Tools extension pack](#).
- While not required to create a Flex Consumption plan app, you need a code project to be able to deploy to and validate a new function app. Complete the first part of one of these quickstart articles, where you create a code project with an HTTP triggered function:
 - [Create an Azure Functions project from the command line](#)
 - [Create an Azure Functions project using Visual Studio Code](#)

Return to this article after you create and run the local project, but before you're asked to create Azure resources. You create the function app and other Azure resources in the next section.

Create a Flex Consumption app

This section shows you how to create a function app in the Flex Consumption plan by using either the Azure CLI, Azure portal, or Visual Studio Code. For an example of creating an app in a Flex Consumption plan using Bicep/ARM templates, see the [Flex Consumption repository](#).

To support your function code, you need to create three resources:

- A [resource group](#), which is a logical container for related resources.
- A [Storage account](#), which is used to maintain state and other information about your functions.
- A function app in the Flex Consumption plan, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources in the Flex Consumption plan.

Azure CLI

```
az login
```

1. If you haven't done so already, sign in to Azure:

Azure CLI

```
az login
```

The [az login](#) command signs you into your Azure account.

2. Use the `az functionapp list-fxconsumption-locations` command to review the list of regions that currently support Flex Consumption.

Azure CLI

```
az functionapp list-fxconsumption-locations --output table
```

3. Create a resource group in one of the [currently supported regions](#):

Azure CLI

```
az group create --name <RESOURCE_GROUP> --location <REGION>
```

In the above command, replace `<RESOURCE_GROUP>` with a value that's unique in your subscription and `<REGION>` with one of the [currently supported regions](#).

The `az group create` command creates a resource group.

4. Create a general-purpose storage account in your resource group and region:

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location <REGION>
--resource-group <RESOURCE_GROUP> --sku Standard_LRS --allow-blob-
public-access false
```

In the previous example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#). The `az storage account create` command creates the storage account.

 **Important**

The storage account is used to store important app data, sometimes including the application code itself. You should limit access from other apps and users to the storage account.

5. Create the function app in Azure:

Azure CLI

```
az functionapp create --resource-group <RESOURCE_GROUP> --name
<APP_NAME> --storage-account <STORAGE_NAME> --flexconsumption-
location <REGION> --runtime dotnet-isolated --runtime-version 8.0
```

[C# apps that run in-process](#) aren't currently supported when running in a Flex Consumption plan.

In this example, replace both `<RESOURCE_GROUP>` and `<STORAGE_NAME>` with the resource group and the name of the account you used in the previous step, respectively. Also replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default domain name server (DNS) domain for the function app. The `az functionapp create` command creates the function app in Azure.

This command creates a function app running in the Flex Consumption plan. The specific language runtime version used is one that is currently supported in the preview.

Because you created the app without specifying [always ready instances](#), your app only incurs costs when actively executing functions. The command also creates an associated Azure Application Insights instance in the same resource group, with which you can monitor your function app and view logs. For more information, see [Monitor Azure Functions](#).

Deploy your code project

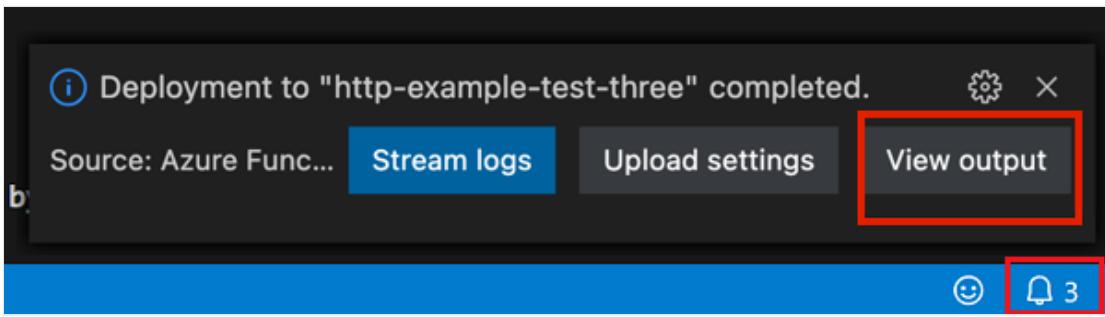
You can choose to deploy your project code to an existing function app using various tools:

Visual Studio Code

ⓘ Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower-right corner to see it again.



Enable virtual network integration

You can enable [virtual network integration](#) for your app in a Flex Consumption plan. The examples in this section assume that you already have [created a virtual network with subnet](#) in your account. You can enable virtual network integration when you create your app or at a later time.

ⓘ Important

The Flex Consumption plan currently doesn't support subnets with names that contain underscore (_) characters.

To enable virtual networking when you create your app:

Azure CLI

You can enable virtual network integration by running the [az functionapp create](#) command and including the `--vnet` and `--subnet` parameters.

1. [Create the virtual network and subnet](#), if you haven't already done so.
2. Complete steps 1-4 in [Create a Flex Consumption app](#) to create the resources required by your app.
3. Run the [az functionapp create](#) command, including the `--vnet` and `--subnet` parameters, as in this example:

Azure CLI

```
az functionapp create --resource-group <RESOURCE_GROUP> --name <APP_NAME> --storage-account <STORAGE_NAME> --flexconsumption-location <REGION> --runtime <RUNTIME_NAME> --runtime-version <RUNTIME_VERSION> --vnet <VNET_RESOURCE_ID> --subnet <SUBNET_NAME>
```

The <VNET_RESOURCE_ID> value is the resource ID for the virtual network, which is in the format:

```
/subscriptions/<SUBSCRIPTION_ID>/resourceGroups/<RESOURCE_GROUP>/providers/Microsoft.Network/virtualNetworks/<VNET_NAME>. You can use this command to get a list of virtual network IDs, filtered by <RESOURCE_GROUP>: az network vnet list --resource-group <RESOURCE_GROUP> --output tsv --query "[].id".
```

For end-to-end examples of how to create apps in Flex Consumption with virtual network integration see these resources:

- [Flex Consumption: HTTP to Event Hubs using VNET Integration ↗](#)
- [Flex Consumption: triggered from Service Bus using VNET Integration ↗](#)

To modify or delete virtual network integration in an existing app:

Azure CLI

Use the [az functionapp vnet-integration add](#) command to enable virtual network integration to an existing function app:

Azure CLI

```
az functionapp vnet-integration add --resource-group <RESOURCE_GROUP> --name <APP_NAME> --vnet <VNET_RESOURCE_ID> --subnet <SUBNET_NAME>
```

Use the [az functionapp vnet-integration remove](#) command to disable virtual network integration in your app:

Azure CLI

```
az functionapp vnet-integration remove --resource-group <RESOURCE_GROUP> --name <APP_NAME>
```

Use the [az functionapp vnet-integration list](#) command to list the current virtual network integrations for your app:

Azure CLI

```
az functionapp vnet-integration list --resource-group <RESOURCE_GROUP> --name <APP_NAME>
```

When choosing a subnet, these considerations apply:

- The subnet you choose can't already be used for other purposes, such as with private endpoints or service endpoints, or be delegated to any other hosting plan or service.
- You can share the same subnet with more than one app running in a Flex Consumption plan. Because the networking resources are shared across all apps, one function app might impact the performance of others on the same subnet.
- In a Flex Consumption plan, a single function app might use up to 40 IP addresses, even when the app scales beyond 40 instances. While this rule of thumb is helpful when estimating the subnet size you need, it's not strictly enforced.

Configure deployment settings

In the Flex Consumption plan, the deployment package that contains your app's code is maintained in an Azure Blob Storage container. By default, deployments use the same storage account (`AzureWebJobsStorage`) and connection string value used by the Functions runtime to maintain your app. The connection string is stored in the `DEPLOYMENT_STORAGE_CONNECTION_STRING` application setting. However, you can instead designate a blob container in a separate storage account as the deployment source for your code. You can also change the authentication method used to access the container.

A customized deployment source should meet this criteria:

- The storage account must already exist.
- The container to use for deployments must also exist.
- When more than one app uses the same storage account, each should have its own deployment container. Using a unique container for each app prevents the deployment packages from being overwritten, which would happen if apps shared the same container.

When configuring deployment storage authentication, keep these considerations in mind:

- When you use a connection string to connect to the deployment storage account, the application setting that contains the connection string must already exist.
- When you use a user-assigned managed identity, the provided identity gets linked to the function app. The `Storage Blob Data Contributor` role scoped to the deployment storage account also gets assigned to the identity.
- When you use a system-assigned managed identity, an identity gets created when a valid system-assigned identity doesn't already exist in your app. When a system-

assigned identity does exists, the `Storage Blob Data Contributor` role scoped to the deployment storage account also gets assigned to the identity.

To configure deployment settings when you create your function app in the Flex Consumption plan:

Azure CLI

Use the `az functionapp create` command and supply these additional options that customize deployment storage:

[Expand table](#)

Parameter	Description
<code>--deployment-storage-name</code>	The name of the deployment storage account.
<code>--deployment-storage-container-name</code>	The name of the container in the account to contain your app's deployment package.
<code>--deployment-storage-auth-type</code>	The authentication type to use for connecting to the deployment storage account. Accepted values include <code>StorageAccountConnectionString</code> , <code>UserAssignedIdentity</code> , and <code>SystemAssignedIdentity</code> .
<code>--deployment-storage-auth-value</code>	When using <code>StorageAccountConnectionString</code> , this parameter is set to the name of the application setting that contains the connection string to the deployment storage account. When using <code>UserAssignedIdentity</code> , this parameter is set to the name of the resource ID of the identity you want to use.

This example creates a function app in the Flex Consumption plan with a separate deployment storage account and user assigned identity:

Azure CLI

```
az functionapp create --resource-group <RESOURCE_GROUP> --name <APP_NAME> --storage <STORAGE_NAME> --runtime dotnet-isolated --runtime-version 8.0 --flexconsumption-location "<REGION>" --deployment-storage-name <DEPLOYMENT_ACCOUNT_NAME> --deployment-storage-container-name <DEPLOYMENT_CONTAINER_NAME> --deployment-storage-auth-type UserAssignedIdentity --deployment-storage-auth-value <MI_RESOURCE_ID>
```

You can also modify the deployment storage configuration for an existing app.

Azure CLI

Use the `az functionapp deployment config set` command to modify the deployment storage configuration:

Azure CLI

```
az functionapp deployment config set --resource-group <RESOURCE_GROUP> --name <APP_NAME> --deployment-storage-name <DEPLOYMENT_ACCOUNT_NAME> --deployment-storage-container-name <DEPLOYMENT_CONTAINER_NAME>
```

Configure instance memory

The instance memory size used by your Flex Consumption plan can be explicitly set when you create your app. For more information about supported sizes, see [Instance memory](#).

To set an instance memory size that's different from the default when creating your app:

Azure CLI

Specify the `--instance-memory` parameter in your `az functionapp create` command.

This example creates a C# app with an instance size of `4096`:

Azure CLI

```
az functionapp create --instance-memory 4096 --resource-group <RESOURCE_GROUP> --name <APP_NAME> --storage-account <STORAGE_NAME> --flexconsumption-location <REGION> --runtime dotnet-isolated --runtime-version 8.0
```

At any point, you can change the instance memory size setting used by your app.

Azure CLI

This example uses the `az functionapp scale config set` command to change the instance memory size setting to 4,096 MB:

Azure CLI

```
az functionapp scale config set --resource-group <resourceGroup> --name
```

```
<APP_NAME> --instance-memory 4096
```

Set always ready instance counts

You can set a number of always ready instances for the [Per-function scaling](#) groups or individual functions, to keep your functions loaded and ready to execute. There are three special groups, as in per-function scaling:

- `http` - all the HTTP triggered functions in the app scale together into their own instances.
- `durable` - all the Durable triggered functions (Orchestration, Activity, Entity) in the app scale together into their own instances.
- `blob` - all the blob (Event Grid) triggered functions in the app scale together into their own instances.

Use `http`, `durable` or `blob` as the name for the name value pair setting to configure always ready counts for these groups. For all other functions in the app you need to configure always ready for each individual function using the format `function:`

```
<FUNCTION_NAME>=n.
```

Azure CLI

Use the `--always-ready-instances` parameter with the [az functionapp create](#) command to define one or more always ready instance designations. This example sets the always ready instance count for all HTTP triggered functions to `5`:

Azure CLI

```
az functionapp create --resource-group <RESOURCE_GROUP> --name
<APP_NAME> --storage <STORAGE_NAME> --runtime <LANGUAGE_RUNTIME> --
runtime-version <RUNTIME_VERSION> --flexconsumption-location <REGION> --
always-ready-instances http=10
```

This example sets the always ready instance count for all Durable trigger functions to `3` and sets the always ready instance count to `2` for a service bus triggered function named `function5`:

Azure CLI

```
az functionapp create --resource-group <RESOURCE_GROUP> --name
<APP_NAME> --storage <STORAGE_NAME> --runtime <LANGUAGE_RUNTIME> --
```

```
runtime-version <RUNTIME_VERSION> --flexconsumption-location <REGION> --  
always-ready-instances durable=3 function:function5=2
```

You can also modify always ready instances on an existing app by adding or removing instance designations or by changing existing instance designation counts.

Azure CLI

This example uses the [az functionapp scale config always-ready set](#) command to change the always ready instance count for the HTTP triggers group to 10:

Azure CLI

```
az functionapp scale config always-ready set --resource-group  
<RESOURCE_GROUP> --name <APP_NAME> --settings http=10
```

To remove always ready instances, use the [az functionapp scale config always-ready delete](#) command, as in this example that removes all always ready instances from both the HTTP triggers group and also a function named `hello_world`:

Azure CLI

```
az functionapp scale config always-ready delete --resource-group  
<RESOURCE_GROUP> --name <APP_NAME> --setting-names http  
function:hello_world
```

Set HTTP concurrency limits

Unless you set specific limits, HTTP concurrency defaults for Flex Consumption plan apps are determined based on your instance size setting. For more information, see [HTTP trigger concurrency](#).

Here's how you can set HTTP concurrency limits for an existing app:

Azure CLI

Use the [az functionapp scale config set](#) command to set specific HTTP concurrency limits for your app, regardless of instance size.

Azure CLI

```
az functionapp scale config set --resource-group <RESOURCE_GROUP> --name <APP_NAME> --trigger-type http --trigger-settings perInstanceConcurrency=10
```

This example sets the HTTP trigger concurrency level to `10`. After you specifically set an HTTP concurrency value, that value is maintained despite any changes in your app's instance size setting.

View currently supported regions

During the preview, you're only able to run on the Flex Consumption plan only in selected regions. To view the list of regions that currently support Flex Consumption plans:

1. If you haven't done so already, sign in to Azure:

```
Azure CLI
```

```
az login
```

The `az login` command signs you into your Azure account.

2. Use the `az functionapp list-fxconsumption-locations` command to review the list of regions that currently support Flex Consumption.

```
Azure CLI
```

```
az functionapp list-fxconsumption-locations --output table
```

When you create an app in the [Azure portal](#) or by using [Visual Studio Code](#), currently unsupported regions are filtered out of the region list.

Related content

- [Azure Functions Flex Consumption plan hosting](#)
- [Azure Functions hosting options](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Use App Configuration references for App Service and Azure Functions (preview)

Article • 03/29/2023

This topic shows you how to work with configuration data in your App Service or Azure Functions application without requiring any code changes. [Azure App Configuration](#) is a service to centrally manage application configuration. Additionally, it's an effective audit tool for your configuration values over time or releases.

Granting your app access to App Configuration

To get started with using App Configuration references in App Service, you'll first need an App Configuration store, and provide your app permission to access the configuration key-values in the store.

1. Create an App Configuration store by following the [App Configuration quickstart](#).

 Note

App Configuration references do not yet support network-restricted configuration stores.

2. Create a [managed identity](#) for your application.

App Configuration references will use the app's system assigned identity by default, but you can [specify a user-assigned identity](#).

3. Enable the newly created identity to have the right set of access permissions on the App Configuration store. Update the [role assignments for your store](#). You'll be assigning `App Configuration Data Reader` role to this identity, scoped over the resource.

Access App Configuration Store with a user-assigned identity

Some apps might need to reference configuration at creation time, when a system-assigned identity wouldn't yet be available. In these cases, a user-assigned identity can

be created and given access to the App Configuration store, in advance. Follow these steps to [create user-assigned identity for App Configuration store](#).

Once you have granted permissions to the user-assigned identity, follow these steps:

1. [Assign the identity](#) to your application if you haven't already.
2. Configure the app to use this identity for App Configuration reference operations by setting the `keyVaultReferenceIdentity` property to the resource ID of the user-assigned identity. Though the property has `keyVault` in the name, the identity will apply to App Configuration references as well.

Azure CLI

```
userAssignedIdentityResourceId=$(az identity show -g MyResourceGroupName -n MyUserAssignedIdentityName --query id -o tsv)
appResourceId=$(az webapp show -g MyResourceGroupName -n MyAppName --query id -o tsv)
az rest --method PATCH --uri "${appResourceId}?api-version=2021-01-01"
--body "{'properties':
{'keyVaultReferenceIdentity':'${userAssignedIdentityResourceId}'}}"
```

This configuration will apply to all references from this App.

Granting your app access to referenced key vaults

In addition to storing raw configuration values, Azure App Configuration has its own format for storing [Key Vault references](#). If the value of an App Configuration reference is a Key Vault reference in App Configuration store, your app will also need to have permission to access the key vault being specified.

Note

The Azure App Configuration Key Vault references concept should not be confused with [the App Service and Azure Functions Key Vault references concept](#). Your app may use any combination of these, but there are some important differences to note. If your vault needs to be network restricted or you need the app to periodically update to latest versions, consider using the App Service and Azure Functions direct approach instead of using an App Configuration reference.

1. Identify the identity that you used for the App Configuration reference. Access to the vault must be granted to that same identity.
2. Create an [access policy in Key Vault](#) for that identity. Enable the "Get" secret permission on this policy. Do not configure the "authorized application" or `applicationId` settings, as this is not compatible with a managed identity.

Reference syntax

An App Configuration reference is of the form

`@Microsoft.AppConfiguration({referenceString})`, where `{referenceString}` is replaced by below:

Reference	Description
string parts	
<code>Endpoint=endpoint;</code>	Endpoint is the required part of the reference string. The value for Endpoint should have the url of your App Configuration resource.
<code>Key=keyName;</code>	Key forms the required part of the reference string. Value for Key should be the name of the Key that you want to assign to the App setting.
<code>Label=label</code>	The Label part is optional in reference string. Label should be the value of Label for the Key specified in Key

For example, a complete reference with `Label` would look like the following,

```
@Microsoft.AppConfiguration(Endpoint=https://myAppConfigStore.azureconfig.io;
Key=myAppConfigKey; Label=myKeysLabel)
```

Alternatively without any `Label`:

```
@Microsoft.AppConfiguration(Endpoint=https://myAppConfigStore.azureconfig.io;
Key=myAppConfigKey)
```

Any configuration change to the app that results in a site restart causes an immediate refetch of all referenced key-values from the App Configuration store.

Source Application Settings from App Config

App Configuration references can be used as values for [Application Settings](#), allowing you to keep configuration data in App Configuration instead of the site config. Application Settings and App Configuration key-values both are securely encrypted at rest. If you need centralized configuration management capabilities, then configuration data should go into App Config.

To use an App Configuration reference for an [app setting](#), set the reference as the value of the setting. Your app can reference the Configuration value through its key as usual. No code changes are required.

💡 Tip

Most application settings using App Configuration references should be marked as slot settings, as you should have separate stores or labels for each environment.

Considerations for Azure Files mounting

Apps can use the `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` application setting to mount Azure Files as the file system. This setting has additional validation checks to ensure that the app can be properly started. The platform relies on having a content share within Azure Files, and it assumes a default name unless one is specified via the `WEBSITE_CONTENTSHARE` setting. For any requests that modify these settings, the platform will attempt to validate if this content share exists, and it will attempt to create it if not. If it can't locate or create the content share, the request is blocked.

If you use App Configuration references for this setting, this validation check will fail by default, as the connection itself can't be resolved while processing the incoming request. To avoid this issue, you can skip the validation by setting

`WEBSITE_SKIP_CONTENTSHARE_VALIDATION` to "1". This setting will bypass all checks, and the content share won't be created for you. You should ensure it's created in advance.

✖️ Caution

If you skip validation and either the connection string or content share are invalid, the app will be unable to start properly and will only serve HTTP 500 errors.

As part of creating the site, it's also possible that attempted mounting of the content share could fail due to managed identity permissions not being propagated or the virtual network integration not being set up. You can defer setting up Azure Files until later in the deployment template to accommodate for the required setup. See [Azure](#)

[Resource Manager deployment](#) to learn more. App Service will use a default file system until Azure Files is set up, and files aren't copied over so make sure that no deployment attempts occur during the interim period before Azure Files is mounted.

Azure Resource Manager deployment

When automating resource deployments through Azure Resource Manager templates, you may need to sequence your dependencies in a particular order to make this feature work. Of note, you'll need to define your application settings as their own resource, rather than using a `siteConfig` property in the site definition. This is because the site needs to be defined first so that the system-assigned identity is created with it and can be used in the access policy.

Below is an example pseudo-template for a function app with App Configuration references:

JSON

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "roleNameGuid": {
            "type": "string",
            "defaultValue": "[newGuid()]",
            "metadata": {
                "description": "A new GUID used to identify the role assignment"
            }
        }
    },
    "variables": {
        "functionAppName": "DemoMBFunc",
        "appConfigStoreName": "DemoMBAppConfig",
        "resourcesRegion": "West US2",
        "appConfigSku": "standard",
        "FontNameKey": "FontName",
        "FontColorKey": "FontColor",
        "myLabel": "Test",
        "App Configuration Data Reader": "[concat('/subscriptions/',
subscription().subscriptionId,
'/providers/Microsoft.Authorization/roleDefinitions/', '516239f1-63e1-4d78-a4de-a74fb236a071')]"
    },
    "resources": [
        {
            "type": "Microsoft.Web/sites",
            "name": "[variables('functionAppName')]"
        }
    ]
}
```

```
    "apiVersion": "2021-03-01",
    "location": "[variables('resourcesRegion')]",
    "identity": {
        "type": "SystemAssigned"
    },
    //...
    "resources": [
        {
            "type": "config",
            "name": "appsettings",
            "apiVersion": "2021-03-01",
            //...
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites',
variables('functionAppName'))]",
                "[resourceId('Microsoft.AppConfiguration/configurationStores',
variables('appConfigStoreName'))]"
            ],
            "properties": {
                "WEBSITE_FONTNAME": "[concat('@Microsoft.AppConfiguration(Endpoint=',
reference(resourceId('Microsoft.AppConfiguration/configurationStores',
variables('appConfigStoreName'))).endpoint,';
Key=',variables('FontNameKey'),'; Label=',variables('myLabel'), ')')]",
                "WEBSITE_FONTCOLOR": "[concat('@Microsoft.AppConfiguration(Endpoint=',
reference(resourceId('Microsoft.AppConfiguration/configurationStores',
variables('appConfigStoreName'))).endpoint,';
Key=',variables('FontColorKey'),'; Label=',variables('myLabel'), ')')]",
                "WEBSITE_ENABLE_SYNC_UPDATE_SITE": "true"
            }
        },
        {
            "type": "sourcecontrols",
            "name": "web",
            "apiVersion": "2021-03-01",
            //...
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites',
variables('functionAppName'))]",
                "[resourceId('Microsoft.Web/sites/config',
variables('functionAppName'), 'appsettings')]"
            ]
        }
    ],
    {
        "type": "Microsoft.AppConfiguration/configurationStores",
        "name": "[variables('appConfigStoreName')]",
        "apiVersion": "2019-10-01",
        "location": "[variables('resourcesRegion')]",
        "sku": {
            "name": "[variables('appConfigSku')]"
        }
    }
]
```

```

},
//...
"dependsOn": [
    "[resourceId('Microsoft.Web/sites',
variables('functionAppName'))]"
],
"properties": {
},
"resources": [
{
    "type": "keyValues",
    "name": "[variables('FontNameKey')]",
    "apiVersion": "2021-10-01-preview",
    //...
    "dependsOn": [
        ""
    ],
    "properties": {
        "value": "Calibri",
        "contentType": "application/json"
    }
},
{
    "type": "keyValues",
    "name": "[variables('FontColorKey')]",
    "apiVersion": "2021-10-01-preview",
    //...
    "dependsOn": [
        ""
    ],
    "properties": {
        "value": "Blue",
        "contentType": "application/json"
    }
}
]
},
{
    "scope": "
[resourceId('Microsoft.AppConfiguration/configurationStores',
variables('appConfigStoreName'))]"
},
{
    "type": "Microsoft.Authorization/roleAssignments",
    "apiVersion": "2020-04-01-preview",
    "name": "[parameters('roleNameGuid')]",
    "properties": {
        "roleDefinitionId": "[variables('App Configuration Data
Reader')]",
        "principalId": "
[reference(resourceId('Microsoft.Web/sites/',
variables('functionAppName')),"

```

```
'2020-12-01', 'Full').identity.principalId],
    "principalType": "ServicePrincipal"
}
]
}
```

ⓘ Note

In this example, the source control deployment depends on the application settings. This is normally unsafe behavior, as the app setting update behaves asynchronously. However, because we have included the `WEBSITE_ENABLE_SYNC_UPDATE_SITE` application setting, the update is synchronous. This means that the source control deployment will only begin once the application settings have been fully updated. For more app settings, see [Environment variables and app settings in Azure App Service](#).

Troubleshooting App Configuration References

If a reference isn't resolved properly, the reference value will be used instead. For the application settings, an environment variable would be created whose value has the `@Microsoft.AppConfiguration(...)` syntax. It may cause an error, as the application was expecting a configuration value instead.

Most commonly, this error could be due to a misconfiguration of the [App Configuration access policy](#). However, it could also be due to a syntax error in the reference or the Configuration key-value not existing in the store.

Next steps

[Reference Key vault secrets from App Service](#)

How to target Azure Functions runtime versions

Article • 07/05/2024

A function app runs on a specific version of the Azure Functions runtime. By default, function apps are created in the latest 4.x version of the Functions runtime. Your function apps are supported only when they run on a [supported major version](#). This article explains how to configure a function app in Azure to target, or *pin* to, a specific version when required.

The way that you target a specific version depends on whether you're running Windows or Linux. This version of the article supports Linux. Choose your operating system at the top of the article.

Important

When possible, always run your functions on the latest supported version of the Azure Functions runtime. You should only pin your app to a specific version if you're instructed to do so due to an issue with the latest version. Always move up to the latest runtime version as soon as your functions can run correctly.

During local development, your installed version of Azure Functions Core Tools must match the major runtime version used by the function app in Azure. For more information, see [Core Tools versions](#).

Update your runtime version

When possible, you should always run your function apps on the latest supported version of the Azure Functions runtime. If your function app is currently running on an older version of the runtime, you should migrate your app to version 4.x.

When your app has existing functions, you must take precautions before moving to a later major runtime version. The following articles detail breaking changes between major versions, including language-specific breaking changes. They also provide you with step-by-step instructions for a successful migration of your existing function app.

- [Migrate from runtime version 3.x to version 4.x](#)
- [Migrate from runtime version 1.x to version 4.x](#)

To determine your current runtime version, see [View the current runtime version](#).

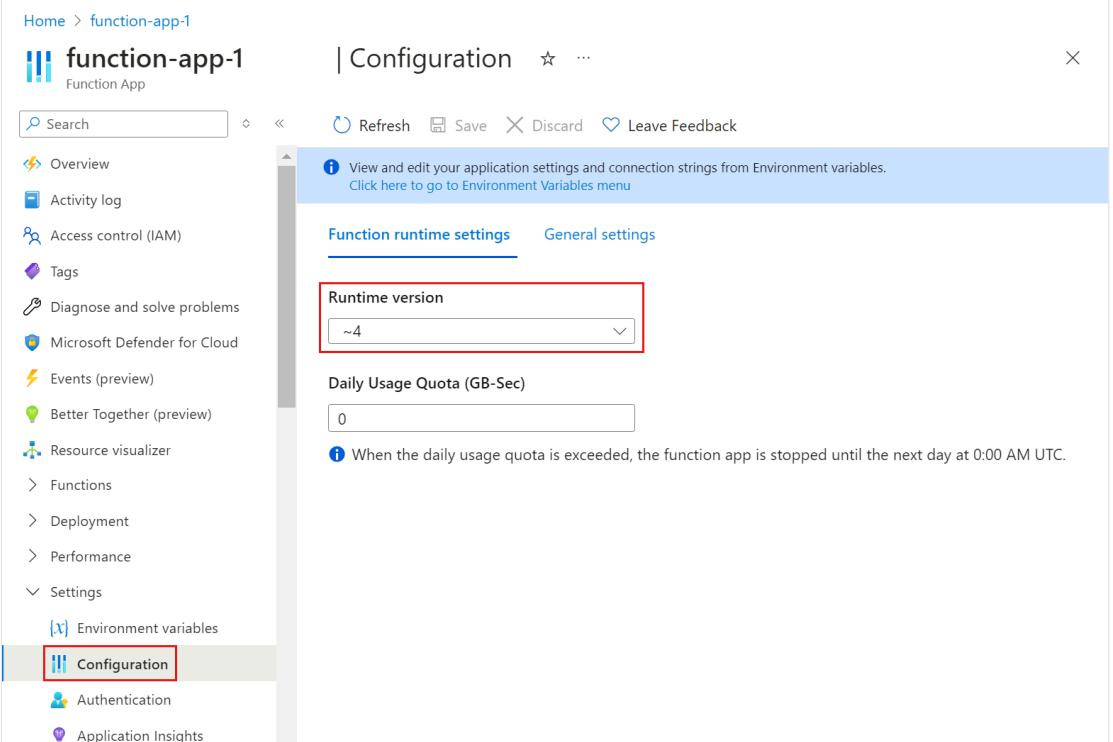
View the current runtime version

You can view the current runtime version of your function app in one of these ways:

Azure portal

To view and update the runtime version currently used by a function app, follow these steps:

1. In the [Azure portal](#), browse to your function app.
2. Expand **Settings**, and then select **Configuration**.
3. In the **Function runtime settings** tab, note the **Runtime version**. In this example, the version is set to `~4`.



Pin to a specific version

Azure Functions lets you use the `FUNCTIONS_EXTENSION_VERSION` app setting to target the runtime version used by a given function app. If you specify only the major version (`~4`), the function app is automatically updated to new minor versions of the runtime as they become available. Minor version updates are done automatically because new minor versions aren't likely to introduce changes that would break your functions.

Linux apps use the [linuxFxVersion site setting](#) along with `FUNCTIONS_EXTENSION_VERSION` to determine the correct Linux base image in which to run your functions. When you create a new function app on Linux, the runtime automatically chooses the correct base image for you based on the runtime version of your language stack.

Pinning to a specific runtime version causes your function app to restart.

To pin your function app to a specific runtime version on Linux, you set a version-specific base image URL in the [linuxFxVersion site setting](#) in the format `DOCKER|<PINNED_VERSION_IMAGE_URI>`.

Important

Pinned function apps on Linux don't receive regular security and host functionality updates. Unless recommended by a support professional, use the [`FUNCTIONS_EXTENSION_VERSION`](#) setting and a standard [linuxFxVersion](#) value for your language and version, such as `Python|3.9`. For valid values, see the [linuxFxVersion reference article](#).

Pinning to a specific runtime isn't currently supported for Linux function apps running in a Consumption plan.

The following example shows the [linuxFxVersion](#) value required to pin a Node.js 16 function app to a specific runtime version of 4.14.0.3:

```
DOCKER|mcr.microsoft.com/azure-functions/node:4.14.0.3-node16
```

When needed, a support professional can provide you with a valid base image URI for your application.

Use the following Azure CLI commands to view and set the [linuxFxVersion](#). You can't currently set [linuxFxVersion](#) in the portal or by using Azure PowerShell:

- To view the current runtime version, use the [az functionapp config show](#) command:

Azure CLI

```
az functionapp config show --name <function_app> \
--resource-group <my_resource_group> --query 'linuxFxVersion' -o tsv
```

In this code, replace `<function_app>` with the name of your function app. Also, replace `<my_resource_group>` with the name of the resource group for your function app. The current value of [linuxFxVersion](#) is returned.

- To update the `linuxFxVersion` setting in the function app, use the [az functionapp config set](#) command:

```
Azure CLI
```

```
az functionapp config set --name <FUNCTION_APP> \
--resource-group <RESOURCE_GROUP> \
--linux-fx-version <LINUX_FX_VERSION>
```

Replace `<FUNCTION_APP>` with the name of your function app. Also, replace `<RESOURCE_GROUP>` with the name of the resource group for your function app. Finally, replace `<LINUX_FX_VERSION>` with the value of a specific image provided to you by a support professional.

You can run these commands from the [Azure Cloud Shell](#) by choosing **Open Cloud Shell** in the preceding code examples. You can also use the [Azure CLI locally](#) to execute this command after executing `az login` to sign in.

The function app restarts after the change is made to the site config.

Next steps

[Release notes for runtime versions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

How to disable functions in Azure Functions

Article • 03/14/2024

This article explains how to disable a function in Azure Functions. To *disable* a function means to make the runtime ignore the event intended to trigger the function. This ability lets you prevent a specific function from running without having to modify and republish the entire function app.

You can disable a function in place by creating an app setting in the format

`AzureWebJobs.<FUNCTION_NAME>.Disabled` set to `true`. You can create and modify this application setting in several ways, including by using the [Azure CLI](#), [Azure PowerShell](#), and from your function's **Overview** tab in the [Azure portal](#).

Changes to application settings cause your function app to restart. For more information, see [App settings reference for Azure Functions](#).

Disable a function

Use one of these modes to create an app setting that disables an example function named `QueueTrigger`:

Portal

Use the **Enable** and **Disable** buttons on the function's **Overview** page. These buttons work by changing the value of the `AzureWebJobs.QueueTrigger.Disabled` app setting. The function-specific app setting is created the first time a function is disabled.

The screenshot shows the Azure portal's overview page for a function named 'HttpTrigger1' within a function app named 'myfunctionapp'. The 'Overview' tab is selected. At the top, there are buttons for 'Enable' (with a checkmark) and 'Disable'. To the right of these are 'Delete', 'Get Function Url', and 'Refresh' buttons. On the left, there's a sidebar with links: 'Developer' (selected), 'Code / Test', 'Integration', 'Monitor', and 'Function Keys'. The main content area displays the function app name, status ('Enabled'), and Application Insights information ('myfunctionapp').

Even when you publish to your function app from a local project, you can still use the portal to disable functions in the function app.

➊ Note

Disabled functions can still be run by calling the REST endpoint using a master key. To learn more, see [Run a disabled function](#). This means that a disabled function still runs when started from the **Test/Run** window in the portal using the **master (Host key)**.

Disable functions in a slot

By default, app settings also apply to apps running in deployment slots. You can, however, override the app setting used by the slot by setting a slot-specific app setting. For example, you might want a function to be active in production but not during deployment testing. It's common to disable timer triggered functions in slots to prevent simultaneous executions.

To disable a function only in the staging slot:

Portal

Navigate to the slot instance of your function app by selecting **Deployment slots** under **Deployment**, choosing your slot, and selecting **Functions** in the slot instance. Choose your function, then use the **Enable** and **Disable** buttons on the function's **Overview** page. These buttons work by changing the value of the `AzureWebJobs`.

`<FUNCTION_NAME>.Disabled` app setting. This function-specific setting is created the first time you disable the function.

You can also directly add the app setting named `AzureWebJobs`.

`<FUNCTION_NAME>.Disabled` with value of `true` in the **Configuration** for the slot instance. When you add a slot-specific app setting, make sure to check the **Deployment slot setting** box. This option maintains the setting value with the slot during swaps.

To learn more, see [Azure Functions Deployment slots](#).

Run a disabled function

You can still cause a disabled function to run by supplying the **master key** in a REST request to the endpoint URL of the disabled function. In this way, you can develop and validate functions in Azure in a disabled state while preventing them from being accessed by others. Using any other type of key in the request returns an HTTP 404 response.

⊗ Caution

Due to the elevated permissions in your function app granted by the master key, you shouldn't share this key with third parties or distribute it in native client applications. Use caution when choosing the admin access level.

To learn more about the master key, see [Obtaining keys](#). To learn more about calling non-HTTP triggered functions, see [Manually run a non HTTP-triggered function](#).

Disable functions locally

Functions can be disabled in the same way when running locally. To disable a function named `QueueTrigger`, add an entry to the `Values` collection in the `local.settings.json` file, as follows:

JSON

```
{  
  "IsEncrypted": false,  
  "Values": {  
    "FUNCTIONS_WORKER_RUNTIME": "python",  
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
  }  
}
```

```
        "AzureWebJobs.QueueTrigger.Disabled": true  
    }  
}
```

Considerations

Keep the following considerations in mind when you disable functions:

- When you disable an HTTP triggered function by using the methods described in this article, the endpoint can still be accessed when running on your local computer and [in the portal](#).
- At this time, function names that contain a hyphen (-) can't be disabled when running on Linux. If you plan to disable your functions when running on Linux, don't use hyphens in your function names.

Next steps

This article is about disabling automatic triggers. For more information about triggers, see [Triggers and bindings](#).

How to use a secured storage account with Azure Functions

Article • 07/02/2024

This article shows you how to connect your function app to a secured storage account. For an in-depth tutorial on how to create your function app with inbound and outbound access restrictions, see the [Integrate with a virtual network](#) tutorial. To learn more about Azure Functions and networking, see [Azure Functions networking options](#).

Restrict your storage account to a virtual network

When you create a function app, you either create a new storage account or link to an existing one. Currently, only the Azure portal, [ARM template deployments](#), and [Bicep deployments](#) support function app creation with an existing secured storage account.

Note

Secured storage accounts are supported for all tiers of the [Dedicated \(App Service\) plan](#) and the [Elastic Premium plan](#). They're also supported by the [Flex Consumption plan](#). The [Consumption plan](#) doesn't support virtual networks.

For a list of all restrictions on storage accounts, see [Storage account requirements](#).

Important

The [Flex Consumption plan](#) is currently in preview.

Secure storage during function app creation

You can create a function app, along with a new storage account that is secured behind a virtual network. The following sections show you how to create these resources by using either the Azure portal or by using deployment templates.

Azure portal

Complete the steps in [Create a function app in a Premium plan](#). This section of the virtual networking tutorial shows you how to create a function app that connects to storage over private endpoints.

Note

When you create your function app in the Azure portal, you can also choose an existing secured storage account in the **Storage** tab. However, you must configure the appropriate networking on the function app so that it can connect through the virtual network used to secure the storage account. If you don't have permissions to configure networking or you haven't fully prepared your network, select **Configure networking after creation** in the **Networking** tab. You can configure networking for your new function app in the portal under **Settings > Networking**.

Secure storage for an existing function app

When you have an existing function app, you can directly configure networking on the storage account being used by the app. However, this process results in your function app being down while you configure networking and while your function app restarts.

To minimize downtime, you can instead swap-out an existing storage account for a new, secured storage account.

1. Enable virtual network integration

As a prerequisite, you need to enable virtual network integration for your function app:

1. Choose a function app with a storage account that doesn't have service endpoints or private endpoints enabled.
2. [Enable virtual network integration](#) for your function app.

2. Create a secured storage account

Set up a secured storage account for your function app:

1. [Create a second storage account](#). This storage account is the secured storage account for your function app to use instead of its original unsecured storage

account. You can also use an existing storage account not already being used by Functions.

2. Save the connection string for this storage account to use later.
3. [Create a file share](#) in the new storage account. For your convenience, you can use the same file share name from your original storage account. Otherwise, if you use a new file share name, you must update your app setting.
4. Secure the new storage account in one of the following ways:
 - [Create a private endpoint](#). As you set up your private endpoint connection, create private endpoints for the `file` and `blob` subresources. For Durable Functions, you must also make `queue` and `table` subresources accessible through private endpoints. If you're using a custom or on-premises Domain Name System (DNS) server, [configure your DNS server](#) to resolve to the new private endpoints.
 - [Restrict traffic to specific subnets](#). Ensure your function app is network integrated with an allowed subnet and that the subnet has a service endpoint to `Microsoft.Storage`.
5. Copy the file and blob content from the current storage account used by the function app to the newly secured storage account and file share. [AzCopy](#) and [Azure Storage Explorer](#) are common methods. If you use Azure Storage Explorer, you might need to allow your client IP address access to your storage account's firewall.

Now you're ready to configure your function app to communicate with the newly secured storage account.

3. Enable application and configuration routing

Note

These configuration steps are required only for the [Elastic Premium](#) and [Dedicated \(App Service\)](#) hosting plans. The [Flex Consumption plan](#) doesn't require site settings to configure networking.

You're now ready to route your function app's traffic to go through the virtual network:

1. Enable [application routing](#) to route your app's traffic to the virtual network:

- a. In your function app, expand **Settings**, and then select **Networking**. In the **Networking** page, under **Outbound traffic configuration**, select the subnet associated with your virtual network integration.
 - b. In the new page, under **Application routing**, select **Outbound internet traffic**.
2. Enable **content share routing** to enable your function app to communicate with your new storage account through its virtual network. In the same page as the previous step, under **Configuration routing**, select **Content storage**.

4. Update application settings

Finally, you need to update your application settings to point to the new secure storage account:

1. In your function app, expand **Settings**, and then select **Environment variables**.
2. In the **App settings** tab, update the following settings by selecting each setting, editing it, and then selecting **Apply**:

[] Expand table

Setting name	Value	Comment
AzureWebJobsStorage	Storage connection string	Use the connection string for your new secured storage account, which you saved earlier.
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING	Storage connection string	Use the connection string for your new secured storage account, which you saved earlier.
WEBSITE_CONTENTSHARE	File share	Use the name of the file share created in the secured storage account where the project deployment files reside.

3. Select **Apply**, and then **Confirm** to save the new application settings in the function app.

The function app restarts.

After the function app finishes restarting, it connects to the secured storage account.

Next steps

[Azure Functions networking options](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Relocate your function app to another Azure region

Article • 09/13/2024

This article describes how to move an Azure Functions-hosted function app to another Azure region.

There are various reasons why you may want to move your existing Azure resources from one region to another. You may want to:

- Take advantage of a new Azure region.
- Deploy features or services available in specific regions only.
- Meet internal policy and governance requirements.
- Align with company mergers and acquisitions
- Meet capacity planning requirements.

The Azure resources that host your function app are region-specific and can't be moved across regions. Instead, you must create a copy of your existing function app resources in the target region, and then redeploy your functions code over to the new app.

You can move these same resources to another resource group or subscription, as long as they remain in the same region. For more information, see [Move App Service resources to a new resource group or subscription](#).

Prerequisites

- Make sure that the target region supports Azure Functions and any related service whose resources you want to move.
- Make sure that you have privileges to create the resources needed in the new region.

Prepare

Identify all the function app resources used on the source region, which potentially includes:

- Function app
- [Hosting plan](#)
- [Deployment slots](#)
- [Custom domains purchased in Azure](#)

- TLS/SSL certificates and settings
- Configured networking options
- Managed identities
- Configured application settings
- Scaling configurations

When preparing to move your app to a new region, there are a few parts of the architecture that require special consideration and planning.

Function app name

Function app names must be globally unique across all Azure apps. This means that your new function app can't have the same name and URL as the original one. This is even true when using a custom DNS, because the underlying

`<APP_NAME>.azurewebsites.net` must still be unique. You might need to update any clients that connect to HTTP endpoints on your function app. These clients need to use the new URL when making requests.

Source code

Ideally, you maintain your source code in a code repository of some kind, or in a container repository if running in a Linux container. If you're using continuous deployment, plan to switch the repository or container deployment connection to the new function app address. If, for some reason, you no longer have the source code, you can [download the currently running package](#) from the original function app. We recommend storing your source files in a code repository and [using continuous deployment for updates](#).

Default storage account

The Functions host requires an Azure Storage account. For more information, see [Storage account requirements](#). For best performance, your function app should [use a storage account in the same region](#). When you create a new app with a new storage account in your new region, your app gets a new set of [function access keys](#) and the state of any triggers (such as timer triggers) is reset.

Persisted local storage

Functions executions are [intended to be stateless](#). However, we don't prevent you from writing data to the local file system. It's possible to store data generated and used by

your application on the `%HOME%\site` virtual drive, but this data shouldn't be state-related. If your scenario requires you to maintain state between function executions, consider instead using [Durable Functions](#).

If your application persists data to the app's shared storage path, make sure to plan how you're going to manage that state during a resource move. Keep in mind that for Dedicated (App Service) plan apps, the share is part of the site. For Consumption and Premium plans, the share is, by default, an Azure Files share in the default storage account. Apps running on Linux might be using an [explicitly mounted share](#) for persisted storage.

Connected services

Your functions might connect to Azure Services and other resources using either a service SDK or triggers and bindings. Any connected service might be negatively impacted when the app moves to a new region. If latency or throughout are issues, consider moving any connected service to the new region as well. To learn how to move those resources across regions, see the documentation for the respective services. When moving an app with connected services, you might want to consider a [cross-region disaster recovery and business continuity](#) strategy during the move.

Changes to connected services might require you to update the values stored in your application settings, which are used to connect to those services.

Configuration

- You can capture a snapshot of the existing app settings and connection strings from the Azure portal. Expand **Settings > Environment variables**, select **Advanced edit** under either **App settings** or **Connections strings** and save the JSON output that contains the existing settings or connections. You need to recreate these settings in the new region, but the values themselves are likely to change as a result of subsequent region changes in the connected services.
- Existing [Key Vault references](#) can't be exported across an Azure geographical boundary. You must recreate any required references in the new region.
- Your app configuration might be managed by [Azure App Configuration](#) or from some other central (downstream) database dependency. Review any App Configuration store or similar stores for environment and region-specific settings that might require modifications.

Custom domains

If your function app uses a custom domain, [bind it preemptively to the target app](#). Verify and [enable the domain in the target app](#). After the move, you must remap the domain name.

Virtual networks

Azure Functions lets you integrate your apps with virtual network resources, and even run them in a virtual network. For more information, see [Azure Functions networking options](#). When moving to a new region, you must first move or recreate all required virtual network and subnet resources before deploying your app. This includes moving or recreating any private endpoints and service endpoints.

Identities

- You need to recreate any system assigned managed identities along with your app in the new target region. Typically, an automatically created Microsoft Entra ID app, used by EasyAuth, defaults to the app resource name.
- User-assigned managed identities also can't be moved across regions. To keep user-assigned managed identities in the same resource group with your app, you must recreate them in the new region. For more information, see [Relocate managed identities for Azure resources to another region](#).
- Grant the managed identities the same permissions in your relocated services as the original identities that they're replacing, including Group memberships.

Certificates

App Service certificate resources can be moved to a new resource group or subscription but not across regions. Certificates that can be exported can also be imported into the app or into Key Vault in the new region. This export and import process is equivalent to a move between regions.

There are different types of certificates that need to be taken into consideration as you plan your service relocation:

[] [Expand table](#)

Certificate type	Exportable	Comments
App Service managed	No	Recreate these certificates in the new region.
Azure Key Vault managed	Yes	These certificates can be exported from Key Vault and then imported into Key Vault in the new region.
Private key (self-managed)	Yes	Certificates you acquired outside of Azure can be exported from App Service and then imported either into the new app or into Key Vault in the new region.
Public key	No	Your app might have certificates with only a public key and no secret, which are used to access other secured endpoints. Obtain the required public key certificate files and import them into the app in the new region.

Access keys

Functions uses access keys to make it more difficult to access HTTP endpoints in your function app. These keys are maintained encrypted in the default storage account. When you create a new app in the new region, a new set of keys get created. You must update any existing clients that use access keys to use the new keys in the new region. While you should use the new keys, it's possible to recreate the old keys in the new app. For more information, see [Work with access keys in Azure Functions](#).

Downtime

If minimal downtime is a requirement, consider running your function app in both regions as recommended to implement a disaster recovery architecture. The specific architecture you implement depends on the trigger types in your function app. For more information, see [Reliability in Azure Functions](#).

Durable Functions

The Durable Functions extension lets you define orchestrations, where state is maintained in your function executions using stateful entities. Ideally, you should allow running orchestrations to complete before migrating your Durable Functions app, especially when you plan to switch to a new storage account in the new region. When migrating your Durable Functions apps, consider using one of these [disaster recovery](#) and [geo-distribution strategies](#).

Relocate

Recreating your function app in a new region requires you to first recreate the Azure infrastructure of an App Service plan, function app instance, and related resources, such as virtual networks, identities, and slots. You also must reconnect or, in the new region, recreate the Azure resources required by the app. These resources might include the default Azure Storage account and the Application Insights instance.

Then you can package and redeploy the actual application source code or container to the function app running in the new region.

Recreate your Azure infrastructure

There are several ways to create a function app and related resources in Azure at the target region:

- **Deployment templates:** If you originally deployed your function app using infrastructure-as-code (IaC) files (Bicep, ARM templates, or Terraform), you can update those previous deployments to target the new region and use them to recreate resources in the new region. If you no longer have these deployment files, you can always [download an ARM template for your existing resource group from the Azure portal](#).
- **Azure CLI/PowerShell scripts:** If you originally deployed your function app using Azure CLI or Azure PowerShell scripts, you can update these scripts to instead target the new region and run them again. If you no longer have these scripts, then you can also [download an ARM template for your existing resource group from the Azure portal](#).
- **Azure portal:** If you created your function app in the portal originally or don't feel comfortable using scripts or IaC files, you can just recreate everything in the portal. Make sure to use the same [hosting plan](#), [language runtime](#), and language version as your original app.

Review configured resources

Review and configure the resources identified in the [Prepare](#) step above in the target region if they weren't configured during the deploy. If you're using continuous deployment with managed identity authentication, make sure the required identities and role mappings exist in the new function app.

Redeploy your source code

Now that you have the infrastructure in place, you can repackaging and redeploy the source code to the function app. This is a good time to move your source code or container image to a repository and [enable continuous deployment](#) from that repository.

You can also use any other publishing method supported by Functions. Most tool-based publishing requires you to [enable basic authentication on the scm endpoint](#), which isn't recommended for production apps.

Relocation considerations

- Remember to verify your configuration and test your functions in the target region.
- If you had custom domain configured, [remap the domain name](#).
- For a function app running in a Dedicated (App Service) plan, also review the [App Service Migration Plan](#) when the plan is shared with one or more web apps.

Clean up

After the move is complete, delete the function app and hosting plan from the source region. You pay for function apps in Premium or Dedicated plans, even when the app itself isn't running. If you have recreated other services in the new region, you should also delete the older services after you're certain that they're no longer needed.

Related resources

Review the [Azure Architecture Center](#) for examples of function apps running in multiple regions as part of more advanced and geo-redundant solution architectures.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Work with legacy proxies

Article • 11/20/2022

ⓘ Important

Azure Functions proxies is a legacy feature for [versions 1.x through 3.x](#) of the Azure Functions runtime. Support for proxies can be re-enabled in version 4.x for you to successfully upgrade your function apps to the latest runtime version. As soon as possible, you should switch to integrating your function apps with Azure API Management. API Management lets you take advantage of a more complete set of features for defining, securing, managing, and monetizing your Functions-based APIs. For more information, see [API Management integration](#).

To learn how to re-enable proxies support in Functions version 4.x, see [Re-enable proxies in Functions v4.x](#).

To help make it easier to migrate from existing proxy implemetations, this article links to equivalent API Management content, when available.

This article explains how to configure and work with Azure Functions Proxies. With this feature, you can specify endpoints on your function app that are implemented by another resource. You can use these proxies to break a large API into multiple function apps (as in a microservice architecture), while still presenting a single API surface for clients.

Standard Functions billing applies to proxy executions. For more information, see [Azure Functions pricing](#).

Re-enable proxies in Functions v4.x

After [migrating your function app to version 4.x of the Functions runtime](#), you'll need to specifically reenable proxies. You should still switch to integrating your function apps with [Azure API Management](#) as soon as possible, and not just rely on proxies.

Re-enabling proxies requires you to set a flag in the `AzureWebJobsFeatureFlags` application setting in one of the following ways:

- If the `AzureWebJobsFeatureFlags` setting doesn't already exists, add this setting to your function app with a value of `EnableProxies`.

- If this setting already exists, add `,EnableProxies` to the end of the existing value.

`AzureWebJobsFeatureFlags` is a comma-delimited array of flags used to enable preview and other temporary features.

To learn more about how to create and modify application settings, see [Work with application settings](#).

Create a proxy

Important

For equivalent content using API Management, see [Expose serverless APIs from HTTP endpoints using Azure API Management](#).

Proxies are defined in the `proxies.json` file in the root of your function app. The steps in this section show you how to use the Azure portal to create this file in your function app. Not all languages and operating system combinations support in-portal editing. If you can't modify your function app files in the portal, you can instead create and deploy the equivalent `proxies.json` file from the root of your local project folder. To learn more about portal editing support, see [Language support details](#).

1. Open the [Azure portal](#), and then go to your function app.
2. In the left pane, select **Proxies** and then select **+Add**.
3. Provide a name for your proxy.
4. Configure the endpoint that's exposed on this function app by specifying the **route template** and **HTTP methods**. These parameters behave according to the rules for [HTTP triggers](#).
5. Set the **backend URL** to another endpoint. This endpoint could be a function in another function app, or it could be any other API. The value doesn't need to be static, and it can reference [application settings](#) and [parameters from the original client request](#).
6. Select **Create**.

Your proxy now exists as a new endpoint on your function app. From a client perspective, it's the same as an `HttpTrigger` in Functions. You can try out your new proxy by copying the **Proxy URL** and testing it with your favorite HTTP client.

Modify requests and responses

Important

API Management lets you change API behavior through configuration using policies. Policies are a collection of statements that are run sequentially on the request or response of an API. For more information about API Management policies, see [Policies in Azure API Management](#).

With proxies, you can modify requests to and responses from the back-end. These transformations can use variables as defined in [Use variables](#).

Modify the back-end request

By default, the back-end request is initialized as a copy of the original request. In addition to setting the back-end URL, you can make changes to the HTTP method, headers, and query string parameters. The modified values can reference [application settings](#) and [parameters from the original client request](#).

Back-end requests can be modified in the portal by expanding the *request override* section of the proxy detail page.

Modify the response

By default, the client response is initialized as a copy of the back-end response. You can make changes to the response's status code, reason phrase, headers, and body. The modified values can reference [application settings](#), [parameters from the original client request](#), and [parameters from the back-end response](#).

Back-end responses can be modified in the portal by expanding the *response override* section of the proxy detail page.

Use variables

The configuration for a proxy doesn't need to be static. You can condition it to use variables from the original client request, the back-end response, or application settings.

Reference local functions

You can use `localhost` to reference a function inside the same function app directly, without a roundtrip proxy request.

"backendUri": "https://localhost/api/httptriggerC#1" will reference a local HTTP triggered function at the route /api/httptriggerC#1

ⓘ Note

If your function uses *function*, *admin* or *sys* authorization levels, you will need to provide the code and clientId, as per the original function URL. In this case the reference would look like: "backendUri": "https://localhost/api/httptriggerC#1?code=<keyvalue>&clientId=<keyname>" We recommend storing these keys in **application settings** and referencing those in your proxies. This avoids storing secrets in your source code.

Reference request parameters

You can use request parameters as inputs to the back-end URL property or as part of modifying requests and responses. Some parameters can be bound from the route template that's specified in the base proxy configuration, and others can come from properties of the incoming request.

Route template parameters

Parameters that are used in the route template are available to be referenced by name. The parameter names are enclosed in braces ({}).

For example, if a proxy has a route template, such as /pets/{petId}, the back-end URL can include the value of {petId}, as in

<https://<AnotherApp>.azurewebsites.net/api/pets/{petId}>. If the route template terminates in a wildcard, such as /api/*restOfPath, the value {restOfPath} is a string representation of the remaining path segments from the incoming request.

Additional request parameters

In addition to the route template parameters, the following values can be used in config values:

- {request.method}: The HTTP method that's used on the original request.
- {request.headers.<HeaderName>}: A header that can be read from the original request. Replace <HeaderName> with the name of the header that you want to read. If the header isn't included on the request, the value will be the empty string.

- `{request.querystring.<ParameterName>}`: A query string parameter that can be read from the original request. Replace `<ParameterName>` with the name of the parameter that you want to read. If the parameter isn't included on the request, the value will be the empty string.

Reference back-end response parameters

Response parameters can be used as part of modifying the response to the client. The following values can be used in config values:

- `{backend.response.statusCode}`: The HTTP status code that's returned on the back-end response.
- `{backend.response.statusReason}`: The HTTP reason phrase that's returned on the back-end response.
- `{backend.response.headers.<HeaderName>}`: A header that can be read from the back-end response. Replace `<HeaderName>` with the name of the header you want to read. If the header isn't included on the response, the value will be the empty string.

Reference application settings

You can also reference [application settings defined for the function app](#) by surrounding the setting name with percent signs (%).

For example, a back-end URL of `https://%ORDER_PROCESSING_HOST%/api/orders` would have "%ORDER_PROCESSING_HOST%" replaced with the value of the ORDER_PROCESSING_HOST setting.

Tip

Use application settings for back-end hosts when you have multiple deployments or test environments. That way, you can make sure that you are always talking to the right back-end for that environment.

Troubleshoot Proxies

By adding the flag `"debug":true` to any proxy in your `proxies.json`, you'll enable debug logging. Logs are stored in `D:\home\LogFiles\Application\Proxies\DetailedTrace` and accessible through the advanced tools (kudu). Any HTTP responses will also contain a `Proxy-Trace-Location` header with a URL to access the log file.

You can debug a proxy from the client side by adding a `Proxy-Trace-Enabled` header set to `true`. This will also log a trace to the file system, and return the trace URL as a header in the response.

Block proxy traces

For security reasons you may not want to allow anyone calling your service to generate a trace. They won't be able to access the trace contents without your sign-in credentials, but generating the trace consumes resources and exposes that you're using Function Proxies.

Disable traces altogether by adding `"debug":false` to any particular proxy in your `proxies.json`.

Advanced configuration

The proxies that you configure are stored in a `proxies.json` file, which is located in the root of a function app directory. You can manually edit this file and deploy it as part of your app when you use any of the [deployment methods](#) that Functions supports.

Tip

If you have not set up one of the deployment methods, you can also work with the `proxies.json` file in the portal. Go to your function app, select **Platform features**, and then select **App Service Editor**. By doing so, you can view the entire file structure of your function app and then make changes.

`Proxies.json` is defined by a `proxies` object, which is composed of named proxies and their definitions. Optionally, if your editor supports it, you can reference a [JSON schema](#) for code completion. An example file might look like the following:

JSON

```
{  
  "$schema": "http://json.schemastore.org/proxies",  
  "proxies": {  
    "proxy1": {  
      "matchCondition": {  
        "methods": [ "GET" ],  
        "route": "/api/{test}"  
      },  
      "backendUri":  
        "https://<AnotherApp>.azurewebsites.net/api/<FunctionName>"  
    }  
  }  
}
```

```
        }
    }
}
```

Each proxy has a friendly name, such as `proxy1` in the preceding example. The corresponding proxy definition object is defined by the following properties:

- **matchCondition**: Required--an object defining the requests that trigger the execution of this proxy. It contains two properties that are shared with [HTTP triggers](#):
 - *methods*: An array of the HTTP methods that the proxy responds to. If it isn't specified, the proxy responds to all HTTP methods on the route.
 - *route*: Required--defines the route template, controlling which request URLs your proxy responds to. Unlike in HTTP triggers, there's no default value.
- **backendUri**: The URL of the back-end resource to which the request should be proxied. This value can reference application settings and parameters from the original client request. If this property isn't included, Azure Functions responds with an HTTP 200 OK.
- **requestOverrides**: An object that defines transformations to the back-end request.
See [Define a requestOverrides object](#).
- **responseOverrides**: An object that defines transformations to the client response.
See [Define a responseOverrides object](#).

ⓘ Note

The *route* property in Azure Functions Proxies does not honor the *routePrefix* property of the Function App host configuration. If you want to include a prefix such as `/api`, it must be included in the *route* property.

Disable individual proxies

You can disable individual proxies by adding `"disabled": true` to the proxy in the `proxies.json` file. This will cause any requests meeting the `matchCondition` to return 404.

JSON

```
{
    "$schema": "http://json.schemastore.org/proxies",
    "proxies": {
        "Root": {
            "disabled":true,
```

```

        "matchCondition": {
            "route": "/example"
        },
        "backendUri":
        "https://<AnotherApp>.azurewebsites.net/api/<FunctionName>"
    }
}

```

Application Settings

The proxy behavior can be controlled by several app settings. They're all outlined in the [Functions App Settings reference](#)

- [AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL](#)
- [AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES](#)

Reserved Characters (string formatting)

Proxies read all strings out of a JSON file, using \ as an escape symbol. Proxies also interpret curly braces. See a full set of examples below.

Character	Escaped Character	Example
{ or }	\{ or \}	\{\{ example \}\} --> \{ example \}
\	\\	example.com\\text.html --> example.com\text.html
"	\"	\\"example\\" --> "example"

Define a requestOverrides object

The `requestOverrides` object defines changes made to the request when the back-end resource is called. The object is defined by the following properties:

- `backend.request.method`: The HTTP method that's used to call the back-end.
- `backend.request.querystring.<ParameterName>`: A query string parameter that can be set for the call to the back-end. Replace `<ParameterName>` with the name of the parameter that you want to set. If an empty string is provided, the parameter is still included on the back-end request.
- `backend.request.headers.<HeaderName>`: A header that can be set for the call to the back-end. Replace `<HeaderName>` with the name of the header that you want to set. If an empty string is provided, the parameter is still included on the back-end request.

Values can reference application settings and parameters from the original client request.

An example configuration might look like the following:

JSON

```
{  
    "$schema": "http://json.schemastore.org/proxies",  
    "proxies": {  
        "proxy1": {  
            "matchCondition": {  
                "methods": [ "GET" ],  
                "route": "/api/{test}"  
            },  
            "backendUri":  
                "https://<AnotherApp>.azurewebsites.net/api/<FunctionName>",  
            "requestOverrides": {  
                "backend.request.headers.Accept": "application/xml",  
                "backend.request.headers.x-functions-key":  
                    "%ANOTHERAPP_API_KEY%"  
            }  
        }  
    }  
}
```

Define a responseOverrides object

The `requestOverrides` object defines changes that are made to the response that's passed back to the client. The object is defined by the following properties:

- `response.statusCode`: The HTTP status code to be returned to the client.
- `response.statusReason`: The HTTP reason phrase to be returned to the client.
- `response.body`: The string representation of the body to be returned to the client.
- `response.headers.<HeaderName>`: A header that can be set for the response to the client. Replace `<HeaderName>` with the name of the header that you want to set. If you provide the empty string, the header isn't included on the response.

Values can reference application settings, parameters from the original client request, and parameters from the back-end response.

An example configuration might look like the following:

JSON

```
{  
    "$schema": "http://json.schemastore.org/proxies",  
    "proxies": {
```

```
"proxy1": {  
    "matchCondition": {  
        "methods": [ "GET" ],  
        "route": "/api/{test}"  
    },  
    "responseOverrides": {  
        "response.body": "Hello, {test}",  
        "response.headers.Content-Type": "text/plain"  
    }  
}  
}
```

ⓘ Note

In this example, the response body is set directly, so no `backendUri` property is needed. The example shows how you might use Azure Functions Proxies for mocking APIs.

Migrate .NET apps from the in-process model to the isolated worker model

Article • 09/04/2024

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you migrate your apps to the isolated worker model by following the instructions in this article.

This article walks you through the process of safely migrating your .NET function app from the [in-process model](#) to the [isolated worker model](#). To learn about the high-level differences between these models, see the [execution mode comparison](#).

This guide assumes that your app is running on version 4.x of the Functions runtime. If not, you should instead follow the guides for upgrading your host version:

- Migrate apps from Azure Functions version 2.x and 3.x to version 4.x
- Migrate apps from Azure Functions version 1.x to version 4.x

These host version migration guides also help you migrate to the isolated worker model as you work through them.

Identify function apps to migrate

Use the following Azure PowerShell script to generate a list of function apps in your subscription that currently use the in-process model.

The script uses subscription that Azure PowerShell is currently configured to use. You can change the subscription by first running `Set-AzContext -Subscription '<YOUR SUBSCRIPTION ID>'` and replacing `<YOUR SUBSCRIPTION ID>` with the ID of the subscription you would like to evaluate.

Azure PowerShell

```
$FunctionApps = Get-AzFunctionApp

$appInfo = @{}

foreach ($app in $FunctionApps)
{
    if ($app.Runtime -eq 'dotnet')
```

```

    {
        $AppInfo.Add($App.Name, $App.Runtime)
    }
}

$AppInfo

```

Choose your target .NET version

On version 4.x of the Functions runtime, your .NET function app targets .NET 6 when using the in-process model.

When you migrate your function app, you have the opportunity to choose the target version of .NET. You can update your C# project to one of the following versions of .NET that are supported by Functions version 4.x:

[\[+\] Expand table](#)

.NET version	.NET Official Support Policy <small>↗</small> release type	Functions process model ^{1,2}
.NET 9	Preview ³	Isolated worker model
.NET 8	LTS (end of support November 10, 2026)	Isolated worker model, In-process model ²
.NET 6	LTS (end of support November 12, 2024)	Isolated worker model, In-process model ²
.NET Framework 4.8	See policy <small>↗</small>	Isolated worker model

¹ The [isolated worker model](#) supports Long Term Support (LTS) and Standard Term Support (STS) versions of .NET, as well as .NET Framework. The [in-process model](#) only supports LTS releases of .NET, ending with .NET 8. For a full feature and functionality comparison between the two models, see [Differences between in-process and isolate worker process .NET Azure Functions](#).

² Support ends for the in-process model on November 10, 2026. For more information, see [this support announcement ↗](#). For continued full support, you should [migrate your apps to the isolated worker model](#).

³ See [Preview .NET versions in the isolated worker model](#) for details on support, current restrictions, and instructions for using the preview version.

💡 Tip

We recommend upgrading to .NET 8 on the isolated worker model. This provides a quick migration path to the fully released version with the longest support window from .NET.

This guide doesn't present specific examples for .NET 9 (Preview) or .NET 6. If you need to target these versions, you can adapt the .NET 8 examples.

Prepare for migration

If you haven't already, identify the list of apps that need to be migrated in your current Azure Subscription by using the [Azure PowerShell](#).

Before you migrate an app to the isolated worker model, you should thoroughly review the contents of this guide. You should also familiarize yourself with the features of the [isolated worker model](#) and the [differences between the two models](#).

To migrate the application, you will:

1. Migrate your local project to the isolated worker model by following the steps in [Migrate your local project](#).
2. After migrating your project, fully test the app locally using version 4.x of the [Azure Functions Core Tools](#).
3. [Update your function app in Azure](#) to the isolated model.

Migrate your local project

The section outlines the various changes that you need to make to your local project to move it to the isolated worker model. Some of the steps change based on your target version of .NET. Use the tabs to select the instructions that match your desired version. These steps assume a local C# project, and if your app is instead using C# script (`.csx` files), you should [convert to the project model](#) before continuing.

💡 Tip

If you are moving to an LTS or STS version of .NET, the [.NET Upgrade Assistant](#) can be used to automatically make many of the changes mentioned in the following sections.

First, convert the project file and update your dependencies. As you do, you will see build errors for the project. In subsequent steps, you'll make the corresponding changes to remove these errors.

Project file

The following example is a `.csproj` project file that uses .NET 6 on version 4.x:

```
XML

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
    <RootNamespace>My.Namespace</RootNamespace>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Sdk.Functions" Version="4.1.1" />
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
</Project>
```

Use one of the following procedures to update this XML file to run in the isolated worker model:

.NET 8

These steps assume a local C# project, and if your app is instead using C# script (`.csx` files), you should [convert to the project model](#) before continuing.

The following changes are required in the `.csproj` XML project file:

1. Set the value of `PropertyGroup.TargetFramework` to `net8.0`.
2. Set the value of `PropertyGroup.AzureFunctionsVersion` to `v4`.
3. Add the following `OutputType` element to the `PropertyGroup`:

XML

```
<OutputType>Exe</OutputType>
```

4. In the `ItemGroup.PackageReference` list, replace the package reference to `Microsoft.NET.Sdk.Functions` with the following references:

XML

```
<FrameworkReference Include="Microsoft.AspNetCore.App" />
<PackageReference Include="Microsoft.Azure.Functions.Worker"
Version="1.21.0" />
<PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
Version="1.17.2" />
<PackageReference
Include="Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore"
Version="1.2.1" />
<PackageReference
Include="Microsoft.ApplicationInsights.WorkerService"
Version="2.22.0" />
<PackageReference
Include="Microsoft.Azure.Functions.Worker.ApplicationInsights"
Version="1.2.0" />
```

Make note of any references to other packages in the `Microsoft.Azure.WebJobs.*` namespaces. You'll replace these packages in a later step.

5. Add the following new `ItemGroup`:

XML

```
<ItemGroup>
<Using Include="System.Threading.ExecutionContext"
Alias="ExecutionContext"/>
</ItemGroup>
```

After you make these changes, your updated project should look like the following example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<TargetFramework>net8.0</TargetFramework>
<AzureFunctionsVersion>v4</AzureFunctionsVersion>
<RootNamespace>My.Namespace</RootNamespace>
<OutputType>Exe</OutputType>
```

```
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
</PropertyGroup>
<ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker"
Version="1.21.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
Version="1.17.2" />
    <PackageReference
Include="Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore"
Version="1.2.1" />
    <PackageReference
Include="Microsoft.ApplicationInsights.WorkerService" Version="2.22.0"
/>
    <PackageReference
Include="Microsoft.Azure.Functions.Worker.ApplicationInsights"
Version="1.2.0" />
    <!-- Other packages may also be in this list -->
</ItemGroup>
<ItemGroup>
    <None Update="host.json">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
        <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
</ItemGroup>
<ItemGroup>
    <Using Include="System.Threading.ExecutionContext"
Alias="ExecutionContext"/>
</ItemGroup>
</Project>
```

Changing your project's target framework might also require changes to parts of your toolchain, outside of project code. For example, in VS Code, you might need to update the `azureFunctions.deploySubpath` extension setting through user settings or your project's `.vscode/settings.json` file. Check for any dependencies on the framework version that may exist outside of your project code, as part of build steps or a CI/CD pipeline.

Package references

When migrating to the isolated worker model, you need to change the packages your application references.

If you haven't already, update your project to reference the latest stable versions of:

- Microsoft.Azure.Functions.Worker ↗
- Microsoft.Azure.Functions.Worker.Sdk ↗

Depending on the triggers and bindings your app uses, your app might need to reference a different set of packages. The following table shows the replacements for some of the most commonly used extensions:

[] [Expand table](#)

Scenario	Changes to package references
Timer trigger	Add Microsoft.Azure.Functions.Worker.Extensions.Timer ↗
Storage bindings	Replace Microsoft.Azure.WebJobs.Extensions.Storage with Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs ↗, Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues ↗, and Microsoft.Azure.Functions.Worker.Extensions.Tables ↗
Blob bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.Storage.Blobs with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs ↗
Queue bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.Storage.Queues with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues ↗
Table bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.Tables with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Tables ↗
Cosmos DB bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.CosmosDB and/or Microsoft.Azure.WebJobs.Extensions.DocumentDB with the latest version of Microsoft.Azure.Functions.Worker.Extensions.CosmosDB ↗
Service Bus bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.ServiceBus with the latest version of Microsoft.Azure.Functions.Worker.Extensions.ServiceBus ↗

Scenario	Changes to package references
Event Hubs bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.EventHubs</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.EventHubs
Event Grid bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.EventGrid</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.EventGrid
SignalR Service bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.SignalRService</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.SignalRService
Durable Functions	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.DurableTask</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask
Durable Functions (SQL storage provider)	Replace references to <code>Microsoft.DurableTask.SqlServer.AzureFunctions</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask.SqlServer
Durable Functions (Netherite storage provider)	Replace references to <code>Microsoft.Azure.DurableTask.Netherite.AzureFunctions</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask.Netherite
SendGrid bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.SendGrid</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.SendGrid
Kafka bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Kafka</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Kafka
RabbitMQ bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.RabbitMQ</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.RabbitMQ
Dependency injection and startup config	Remove references to <code>Microsoft.Azure.Functions.Extensions</code> (The isolated worker model provides this functionality by default.)

See [Supported bindings](#) for a complete list of extensions to consider, and consult each extension's documentation for full installation instructions for the isolated process model. Be sure to install the latest stable version of any packages you are targeting.

💡 Tip

Any changes to extension versions during this process might require you to update your `host.json` file as well. Be sure to read the documentation of each extension that you use. For example, the Service Bus extension has breaking changes in the structure between versions 4.x and 5.x. For more information, see [Azure Service Bus bindings for Azure Functions](#).

Your isolated worker model application should not reference any packages in the `Microsoft.Azure.WebJobs.*` namespaces or `Microsoft.Azure.Functions.Extensions`. If you have any remaining references to these, they should be removed.

💡 Tip

Your app might also depend on Azure SDK types, either as part of your triggers and bindings or as a standalone dependency. You should take this opportunity to update these as well. The latest versions of the Functions extensions work with the latest versions of the [Azure SDK for .NET](#), almost all of the packages for which are the form `Azure.*`.

Program.cs file

When migrating to run in an isolated worker process, you must add a `Program.cs` file to your project with the following contents:

.NET 8

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWebApplication()
    .ConfigureServices(services => {
        services.AddApplicationInsightsTelemetryWorkerService();
        services.ConfigureFunctionsApplicationInsights();
```

```
    })  
    .Build();  
  
    host.Run();
```

This example includes [ASP.NET Core integration](#) to improve performance and provide a familiar programming model when your app uses HTTP triggers. If you do not intend to use HTTP triggers, you can replace the call to `ConfigureFunctionsWebApplication` with a call to `ConfigureFunctionsWorkerDefaults`. If you do so, you can remove the reference to `Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore` from your project file. However, for the best performance, even for functions with other trigger types, you should keep the `FrameworkReference` to ASP.NET Core.

The `Program.cs` file will replace any file that has the `FunctionsStartup` attribute, which is typically a `Startup.cs` file. In places where your `FunctionsStartup` code would reference `IFunctionsHostBuilder.Services`, you can instead add statements within the `.ConfigureServices()` method of the `HostBuilder` in your `Program.cs`. To learn more about working with `Program.cs`, see [Start-up and configuration](#) in the isolated worker model guide.

The default `Program.cs` examples above include setup of [Application Insights integration for the isolated worker model](#). In your `Program.cs`, you must also configure any log filtering that should apply to logs coming from code in your project. In the isolated worker model, the `host.json` file only controls events emitted by the Functions host runtime. If you don't configure filtering rules in `Program.cs`, you may see differences in the log levels present for various categories in your telemetry.

Although you can register custom configuration sources as part of the `HostBuilder`, note that these similarly apply only to code in your project. Trigger and binding configuration is also needed by the platform, and this should be provided through the [application settings](#), [Key Vault references](#), or [App Configuration references](#) features.

Once you have moved everything from any existing `FunctionsStartup` to the `Program.cs` file, you can delete the `FunctionsStartup` attribute and the class it was applied to.

Function signature changes

Some key types change between the in-process model and the isolated worker model. Many of these relate to the attributes, parameters, and return types that make up the function signature. For each of your functions, you must make changes to:

- The function attribute (which also sets the function's name)
- How the function obtains an `ILogger`/`ILogger<T>`
- Trigger and binding attributes and parameters

The rest of this section walks you through each of these steps.

Function attributes

The `Function` attribute in the isolated worker model replaces the `FunctionName` attribute. The new attribute has the same signature, and the only difference is in the name. You can therefore just perform a string replacement across your project.

Logging

In the in-process model, you could include an optional `ILogger` parameter to your function, or you could use dependency injection to get an `ILogger<T>`. If your app already used dependency injection, the same mechanisms work in the isolated worker model.

However, for any Functions that relied on the `ILogger` method parameter, you need to make a change. It is recommended that you use dependency injection to obtain an `ILogger<T>`. Use the following steps to migrate the function's logging mechanism:

1. In your function class, add a `private readonly ILogger<MyFunction> _logger;` property, replacing `MyFunction` with the name of your function class.
2. Create a constructor for your function class that takes in the `ILogger<T>` as a parameter:

C#

```
public MyFunction(ILogger<MyFunction> logger) {
    _logger = logger;
}
```

Replace both instances of `MyFunction` in the preceding code snippet with the name of your function class.

3. For logging operations in your function code, replace references to the `ILogger` parameter with `_logger`.
4. Remove the `ILogger` parameter from your function signature.

To learn more, see [Logging in the isolated worker model](#).

Trigger and binding changes

When you [changed your package references in a previous step](#), you introduced errors for your triggers and bindings that you will now fix:

1. Remove any `using Microsoft.Azure.WebJobs;` statements.
2. Add a `using Microsoft.Azure.Functions.Worker;` statement.
3. For each binding attribute, change the attribute's name as specified in its reference documentation, which you can find in the [Supported bindings](#) index. In general, the attribute names change as follows:
 - Triggers typically remain named the same way. For example, `QueueTrigger` is the attribute name for both models.
 - Input bindings typically need "Input" added to their name. For example, if you used the `CosmosDB` input binding attribute in the in-process model, the attribute would now be `CosmosDBInput`.
 - Output bindings typically need "Output" added to their name. For example, if you used the `Queue` output binding attribute in the in-process model, this attribute would now be `QueueOutput`.
4. Update the attribute parameters to reflect the isolated worker model version, as specified in the binding's reference documentation.

For example, in the in-process model, a blob output binding is represented by a `[Blob(...)]` attribute that includes an `Access` property. In the isolated worker model, the blob output attribute would be `[BlobOutput(...)]`. The binding no longer requires the `Access` property, so that parameter can be removed. So

```
[Blob("sample-images-sm/{fileName}", FileAccess.Write, Connection =
"MyStorageConnection")]
```

would become `[BlobOutput("sample-images-
sm/{fileName}", Connection = "MyStorageConnection")]`.

5. Move output bindings out of the function parameter list. If you have just one output binding, you can apply this to the return type of the function. If you have multiple outputs, create a new class with properties for each output, and apply the attributes to those properties. To learn more, see [Multiple output bindings](#).
6. Consult each binding's reference documentation for the types it allows you to bind to. In some cases, you might need to change the type. For output bindings, if the

in-process model version used an `IAsyncCollector<T>`, you can replace this with binding to an array of the target type: `T[]`. You can also consider replacing the output binding with a client object for the service it represents, either as the binding type for an input binding if available, or by [injecting a client yourself](#).

7. If your function includes an `IBinder` parameter, remove it. Replace the functionality with a client object for the service it represents, either as the binding type for an input binding if available, or by [injecting a client yourself](#).
8. Update the function code to work with any new types.

local.settings.json file

The local.settings.json file is only used when running locally. For information, see [Local settings file](#).

When migrating from running in-process to running in an isolated worker process, you need to change the `FUNCTIONS_WORKER_RUNTIME` value to "dotnet-isolated". Make sure that your local.settings.json file has at least the following elements:

```
JSON

{
    "IsEncrypted": false,
    "Values": {
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
    }
}
```

The value you have for `AzureWebJobsStorage` might be different. You do not need to change its value as part of the migration.

host.json file

No changes are required to your `host.json` file. However, if your Application Insights configuration in this file from your in-process model project, you might want to make additional changes in your `Program.cs` file. The `host.json` file only controls logging from the Functions host runtime, and in the isolated worker model, some of these logs come from your application directly, giving you more control. See [Managing log levels in the isolated worker model](#) for details on how to filter these logs.

Other code changes

This section highlights other code changes to consider as you work through the migration. These changes are not needed by all applications, but you should evaluate if any are relevant to your scenarios.

JSON serialization

By default, the isolated worker model uses `System.Text.Json` for JSON serialization. To customize serializer options or switch to JSON.NET (`Newtonsoft.Json`), see [these instructions](#).

Application Insights log levels and filtering

Logs can be sent to Application Insights from both the Functions host runtime and code in your project. The `host.json` allows you to configure rules for host logging, but to control logs coming from your code, you'll need to configure filtering rules as part of your `Program.cs`. See [Managing log levels in the isolated worker model](#) for details on how to filter these logs.

Example function migrations

HTTP trigger example

An HTTP trigger for the in-process model might look like the following example:

```
C#  
  
using Microsoft.AspNetCore.Http;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.Extensions.Logging;  
  
namespace Company.Function  
{  
    public static class HttpTriggerCSharp  
    {  
        [FunctionName("HttpTriggerCSharp")]  
        public static IActionResult Run(  
            [HttpTrigger(AuthorizationLevel.Function, "get", Route = null)]  
            HttpRequest req,  
            ILogger log)  
        {  
            log.LogInformation("C# HTTP trigger function processed a  
request.");  
        }  
    }  
}
```

```
        return new OkObjectResult($"Welcome to Azure Functions,
{req.Query["name"]}!");
    }
}
```

An HTTP trigger for the migrated version might look like the following example:

.NET 8

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace Company.Function
{
    public class HttpTriggerCSharp
    {
        private readonly ILogger<HttpTriggerCSharp> _logger;

        public HttpTriggerCSharp(ILogger<HttpTriggerCSharp> logger)
        {
            _logger = logger;
        }

        [Function("HttpTriggerCSharp")]
        public IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Function, "get")]
            HttpRequest req)
        {
            _logger.LogInformation("C# HTTP trigger function processed a
request.");

            return new OkObjectResult($"Welcome to Azure Functions,
{req.Query["name"]}!");
        }
    }
}
```

Update your function app in Azure

Updating your function app to the isolated model involves two changes that should be completed together, because if you only complete one, the app is in an error state. Both of these changes also cause the app process to restart. For these reasons, you should perform the update using a [staging slot](#). Staging slots help minimize downtime for your

app and allow you to test and verify your migrated code with your updated configuration in Azure. You can then deploy your fully migrated app to the production slot through a swap operation.

Important

[When an app's deployed payload doesn't match the configured runtime, it will be in an error state.](#) During the migration process, you will put the app into this state, ideally only temporarily. Deployment slots help mitigate the impact of this, because the error state will be resolved in your staging (non-production) environment before the changes are applied as single update to your production environment. Slots also defend against any mistakes and allow you to detect any other issues before reaching production.

During the process, you might still see errors in logs coming from your staging (non-production) slot. This is expected, though these should go away as you proceed through the steps. Before you perform the slot swap operation, you should confirm that these errors stop being raised and that your application is working as expected.

Use the following steps to use deployment slots to update your function app to the isolated worker model:

1. [Create a deployment slot](#) if you haven't already. You might also want to familiarize yourself with the slot swap process and ensure that you can make updates to the existing application with minimal disruption.
2. Change the configuration of the staging (non-production) slot to use the isolated worker model by setting the `FUNCTIONS_WORKER_RUNTIME` application setting to `dotnet-isolated`. `FUNCTIONS_WORKER_RUNTIME` should **not** be marked as a "slot setting".

If you are also targeting a different version of .NET as part of your update, you should also change the stack configuration. To do so, see the [instructions to update the stack configuration for the isolated worker model](#). You will use the same instructions for any future .NET version updates you make.

If you have any automated infrastructure provisioning such as a CI/CD pipeline, make sure that the automations are also updated to keep `FUNCTIONS_WORKER_RUNTIME` set to `dotnet-isolated` and to target the correct .NET version.

3. Publish your migrated project to the staging (non-production) slot of your function app.

If you use Visual Studio to publish an isolated worker model project to an existing app or slot that uses the in-process model, it can also complete the previous step for you at the same time. If you did not complete the previous step, Visual Studio prompts you to update the function app during deployment. Visual Studio presents this as a single operation, but these are still two separate operations. You might still see errors in your logs from the staging (non-production) slot during the interim state.

4. Confirm that your application is working as expected within the staging (non-production) slot.
5. Perform a [slot swap operation](#). This applies the changes you made in your staging (non-production) slot to the production slot. A slot swap happens as a single update, which avoids introducing the interim error state in your production environment.
6. Confirm that your application is working as expected within the production slot.

Once you complete these steps, the migration is complete, and your app runs on the isolated model. Congratulations! Repeat the steps from this guide as necessary for [any other apps needing migration](#).

Next steps

[Learn more about the isolated worker model](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Migrate apps from Azure Functions version 3.x to version 4.x

Article • 03/29/2024

Azure Functions version 4.x is highly backwards compatible to version 3.x. Most apps should safely migrate to 4.x without requiring significant code changes. For more information about Functions runtime versions, see [Azure Functions runtime versions overview](#).

Important

As of December 13, 2022, function apps running on versions 2.x and 3.x of the Azure Functions runtime have reached the end of extended support. For more information, see [Retired versions](#).

This article walks you through the process of safely migrating your function app to run on version 4.x of the Functions runtime. Because project migration instructions are language dependent, make sure to choose your development language from the selector at the [top of the article](#).

Identify function apps to migrate

Use the following PowerShell script to generate a list of function apps in your subscription that currently target versions 2.x or 3.x:

PowerShell

```
$Subscription = '<YOUR SUBSCRIPTION ID>'  
  
Set-AzContext -Subscription $Subscription | Out-Null  
  
$FunctionApps = Get-AzFunctionApp  
  
$AppInfo = @{}  
  
foreach ($App in $FunctionApps)  
{  
    if ($App.ApplicationSettings["FUNCTIONS_EXTENSION_VERSION"] -like  
'*3*')  
    {  
        $AppInfo.Add($App.Name,  
$App.ApplicationSettings["FUNCTIONS_EXTENSION_VERSION"])  
    }  
}
```

```
}
```

```
$AppInfo
```

Choose your target .NET version

On version 3.x of the Functions runtime, your C# function app targets .NET Core 3.1 using the in-process model or .NET 5 using the isolated worker model.

When you migrate your function app, you have the opportunity to choose the target version of .NET. You can update your C# project to one of the following versions of .NET that are supported by Functions version 4.x:

[+] Expand table

.NET version	.NET Official Support Policy <small>↗</small> release type	Functions process model ^{1,3}
.NET 8 ²	LTS	Isolated worker model
.NET 7	STS (end of support May 14, 2024)	Isolated worker model
.NET 6	LTS (end of support November 12, 2024)	Isolated worker model, In-process model ³
.NET Framework 4.8	See policy <small>↗</small>	Isolated worker model

¹ The [isolated worker model](#) supports Long Term Support (LTS) and Standard Term Support (STS) versions of .NET, as well as .NET Framework. The [in-process model](#) only supports LTS releases of .NET. For a full feature and functionality comparison between the two models, see [Differences between in-process and isolate worker process .NET Azure Functions](#).

² .NET 8 is not yet supported on the in-process model, though it is available on the isolated worker model. For information about .NET 8 plans, including future options for the in-process model, see the [Azure Functions Roadmap Update post](#) ↗.

³ Support ends for the in-process model on November 10, 2026. For more information, see [this support announcement](#) ↗. For continued full support, you should [migrate your apps to the isolated worker model](#).

 Tip

We recommend updating to .NET 8 on the isolated worker model. .NET 8 is the fully released version with the longest support window from .NET.

Although you can choose to instead use the in-process model, this is not recommended if it can be avoided. [Support will end for the in-process model on November 10, 2026](#), so you'll need to move to the isolated worker model before then. Doing so while migrating to version 4.x will decrease the total effort required, and the isolated worker model will give your app [additional benefits](#), including the ability to more easily target future versions of .NET. If you are moving to the isolated worker model, the [.NET Upgrade Assistant](#) can also handle many of the necessary code changes for you.

This guide doesn't present specific examples for .NET 7 or .NET 6 on the isolated worker model. If you need to target these versions, you can adapt the .NET 8 isolated worker model examples.

Prepare for migration

If you haven't already, identify the list of apps that need to be migrated in your current Azure Subscription by using the [Azure PowerShell](#).

Before you migrate an app to version 4.x of the Functions runtime, you should do the following tasks:

1. Review the list of [breaking changes between 3.x and 4.x](#).
2. Complete the steps in [Migrate your local project](#) to migrate your local project to version 4.x.
3. After migrating your project, fully test the app locally using version 4.x of the [Azure Functions Core Tools](#).
4. [Run the pre-upgrade validator](#) on the app hosted in Azure, and resolve any identified issues.
5. Update your function app in Azure to the new version. If you need to minimize downtime, consider using a [staging slot](#) to test and verify your migrated app in Azure on the new runtime version. You can then deploy your app with the updated version settings to the production slot. For more information, see [Update using slots](#).
6. Publish your migrated project to the updated function app.

When you use Visual Studio to publish a version 4.x project to an existing function app at a lower version, you're prompted to let Visual Studio update the function app to

version 4.x during deployment. This update uses the same process defined in [Update without slots](#).

Migrate your local project

Upgrading instructions are language dependent. If you don't see your language, choose it from the selector at the [top of the article](#).

Choose the tab that matches your target version of .NET and the desired process model (in-process or isolated worker process).

Tip

If you are moving to an LTS or STS version of .NET using the isolated worker model, the [.NET Upgrade Assistant](#) can be used to automatically make many of the changes mentioned in the following sections.

Project file

The following example is a `.csproj` project file that uses .NET Core 3.1 on version 3.x:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <AzureFunctionsVersion>v3</AzureFunctionsVersion>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Sdk.Functions" Version="3.0.13" />
  </ItemGroup>
  <ItemGroup>
    <Reference Include="Microsoft.CSharp" />
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
</Project>
```

Use one of the following procedures to update this XML file to run in Functions version 4.x:

.NET 8

These steps assume a local C# project, and if your app is instead using C# script (`.csx` files), you should [convert to the project model](#) before continuing.

The following changes are required in the `.csproj` XML project file:

1. Set the value of `PropertyGroup.TargetFramework` to `net8.0`.
2. Set the value of `PropertyGroup.AzureFunctionsVersion` to `v4`.
3. Add the following `OutputType` element to the `PropertyGroup`:

XML

```
<OutputType>Exe</OutputType>
```

4. In the `ItemGroup.PackageReference` list, replace the package reference to `Microsoft.NET.Sdk.Functions` with the following references:

XML

```
<FrameworkReference Include="Microsoft.AspNetCore.App" />
<PackageReference Include="Microsoft.Azure.Functions.Worker"
Version="1.21.0" />
<PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
Version="1.17.2" />
<PackageReference
Include="Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore"
Version="1.2.1" />
<PackageReference
Include="Microsoft.ApplicationInsights.WorkerService"
Version="2.22.0" />
<PackageReference
Include="Microsoft.Azure.Functions.Worker.ApplicationInsights"
Version="1.2.0" />
```

Make note of any references to other packages in the `Microsoft.Azure.WebJobs.*` namespaces. You'll replace these packages in a later step.

5. Add the following new `ItemGroup`:

XML

```
<ItemGroup>
  <Using Include="System.Threading.ExecutionContext"
    Alias="ExecutionContext"/>
</ItemGroup>
```

After you make these changes, your updated project should look like the following example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
    <RootNamespace>My.Namespace</RootNamespace>
    <OutputType>Exe</OutputType>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker"
      Version="1.21.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
      Version="1.17.2" />
    <PackageReference
      Include="Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore"
      Version="1.2.1" />
    <PackageReference
      Include="Microsoft.ApplicationInsights.WorkerService" Version="2.22.0"
      />
    <PackageReference
      Include="Microsoft.Azure.Functions.Worker.ApplicationInsights"
      Version="1.2.0" />
    <!-- Other packages may also be in this list -->
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
  <ItemGroup>
    <Using Include="System.Threading.ExecutionContext"
      Alias="ExecutionContext"/>
```

```
</ItemGroup>  
</Project>
```

Package and namespace changes

Based on the model you are migrating to, you might need to update or change the packages your application references. When you adopt the target packages, you then need to update the namespace of using statements and some types you reference. You can see the effect of these namespace changes on [using statements in the HTTP trigger template examples](#) later in this article.

.NET 8

If you haven't already, update your project to reference the latest stable versions of:

- [Microsoft.Azure.Functions.Worker](#)
- [Microsoft.Azure.Functions.Worker.Sdk](#)

Depending on the triggers and bindings your app uses, your app might need to reference a different set of packages. The following table shows the replacements for some of the most commonly used extensions:

[\[+\] Expand table](#)

Scenario	Changes to package references
Timer trigger	Add Microsoft.Azure.Functions.Worker.Extensions.Timer
Storage bindings	Replace Microsoft.Azure.WebJobs.Extensions.Storage with Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs , Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues , and Microsoft.Azure.Functions.Worker.Extensions.Tables
Blob bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.Storage.Blobs with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs
Queue bindings	Replace references to Microsoft.Azure.WebJobs.Extensions.Storage.Queues with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues

Scenario	Changes to package references
Table bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Tables</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Tables
Cosmos DB bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.CosmosDB</code> and/or <code>Microsoft.Azure.WebJobs.Extensions.DocumentDB</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.CosmosDB
Service Bus bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.ServiceBus</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.ServiceBus
Event Hubs bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.EventHubs</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.EventHubs
Event Grid bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.EventGrid</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.EventGrid
SignalR Service bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.SignalRService</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.SignalRService
Durable Functions	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.DurableTask</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask
Durable Functions (SQL storage provider)	Replace references to <code>Microsoft.DurableTask.SqlServer.AzureFunctions</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask.SqlServer
Durable Functions (Netherite storage provider)	Replace references to <code>Microsoft.Azure.DurableTask.Netherite.AzureFunctions</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask.Netherite

Scenario	Changes to package references
SendGrid bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.SendGrid</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.SendGrid
Kafka bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Kafka</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Kafka
RabbitMQ bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.RabbitMQ</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.RabbitMQ
Dependency injection and startup config	Remove references to <code>Microsoft.Azure.Functions.Extensions</code> (The isolated worker model provides this functionality by default.)

See [Supported bindings](#) for a complete list of extensions to consider, and consult each extension's documentation for full installation instructions for the isolated process model. Be sure to install the latest stable version of any packages you are targeting.

Your isolated worker model application should not reference any packages in the `Microsoft.Azure.WebJobs.*` namespaces OR `Microsoft.Azure.Functions.Extensions`.

If you have any remaining references to these, they should be removed.

💡 Tip

Your app might also depend on Azure SDK types, either as part of your triggers and bindings or as a standalone dependency. You should take this opportunity to update these as well. The latest versions of the Functions extensions work with the latest versions of the [Azure SDK for .NET](#), almost all of the packages for which are the form `Azure.*`.

Program.cs file

When migrating to run in an isolated worker process, you must add the following program.cs file to your project:

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWebApplication()
    .ConfigureServices(services => {
        services.AddApplicationInsightsTelemetryWorkerService();
        services.ConfigureFunctionsApplicationInsights();
    })
    .Build();

host.Run();
```

This example includes [ASP.NET Core integration](#) to improve performance and provide a familiar programming model when your app uses HTTP triggers. If you do not intend to use HTTP triggers, you can replace the call to `ConfigureFunctionsWebApplication` with a call to `ConfigureFunctionsWorkerDefaults`. If you do so, you can remove the reference to `Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore` from your project file. However, for the best performance, even for functions with other trigger types, you should keep the `FrameworkReference` to ASP.NET Core.

The `Program.cs` file will replace any file that has the `FunctionsStartup` attribute, which is typically a `Startup.cs` file. In places where your `FunctionsStartup` code would reference `IFunctionsHostBuilder.Services`, you can instead add statements within the `.ConfigureServices()` method of the `HostBuilder` in your `Program.cs`. To learn more about working with `Program.cs`, see [Start-up and configuration](#) in the isolated worker model guide.

The default `Program.cs` examples above include setup of [Application Insights integration for the isolated worker model](#). In your `Program.cs`, you must also configure any log filtering that should apply to logs coming from code in your project. In the isolated worker model, the `host.json` file only controls events emitted by the Functions host runtime. If you don't configure filtering rules in `Program.cs`, you may see differences in the log levels present for various categories in your telemetry.

Although you can register custom configuration sources as part of the `HostBuilder`, note that these similarly apply only to code in your project. Trigger and binding configuration is also needed by the platform, and this should be provided through the [application settings](#), [Key Vault references](#), or [App Configuration references](#) features.

Once you have moved everything from any existing `FunctionsStartup` to the `Program.cs` file, you can delete the `FunctionsStartup` attribute and the class it was applied to.

local.settings.json file

The local.settings.json file is only used when running locally. For information, see [Local settings file](#).

When you migrate to version 4.x, make sure that your local.settings.json file has at least the following elements:

.NET 8

JSON

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage":  
            "AzureWebJobsConnectionStringValue",  
            "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"  
    }  
}
```

ⓘ Note

When migrating from running in-process to running in an isolated worker process, you need to change the `FUNCTIONS_WORKER_RUNTIME` value to "dotnet-isolated".

host.json file

.NET 8 (isolated)

No changes are required to your `host.json` file. However, if your Application Insights configuration in this file from your in-process model project, you might want to make additional changes in your `Program.cs` file. The `host.json` file only controls logging from the Functions host runtime, and in the isolated worker model, some of these logs come from your application directly, giving you more control. See [Managing log levels in the isolated worker model](#) for details on how to filter these logs.

Class name changes

Some key classes changed names between versions. These changes are a result either of changes in .NET APIs or in differences between in-process and isolated worker process. The following table indicates key .NET classes used by Functions that could change when migrating:

.NET 8

[Expand table](#)

.NET Core 3.1	.NET 5	.NET 8
<code>FunctionName</code> (attribute)	<code>Function</code> (attribute)	<code>Function</code> (attribute)
<code>ILogger</code>	<code>ILogger</code>	<code>ILogger</code> , <code>ILogger<T></code>
<code>HttpRequest</code>	<code>HttpRequestData</code>	<code>HttpRequestData</code> , <code>HttpRequest</code> (using ASP.NET Core integration)
<code>IActionResult</code>	<code>HttpResponseData</code>	<code>HttpResponseData</code> , <code>IActionResult</code> (using ASP.NET Core integration)
<code>FunctionsStartup</code> (attribute)	Uses Program.cs instead	Uses Program.cs instead

There might also be class name differences in bindings. For more information, see the reference articles for the specific bindings.

Other code changes

This section highlights other code changes to consider as you work through the migration. These changes are not needed by all applications, but you should evaluate if any are relevant to your scenarios. Make sure to check [Breaking changes between 3.x and 4.x](#) for additional changes you might need to make to your project.

JSON serialization

By default, the isolated worker model uses `System.Text.Json` for JSON serialization. To customize serializer options or switch to JSON.NET (`Newtonsoft.Json`), see [these instructions](#).

Application Insights log levels and filtering

Logs can be sent to Application Insights from both the Functions host runtime and code in your project. The `host.json` allows you to configure rules for host logging, but to control logs coming from your code, you'll need to configure filtering rules as part of your `Program.cs`. See [Managing log levels in the isolated worker model](#) for details on how to filter these logs.

HTTP trigger template

The differences between in-process and isolated worker process can be seen in HTTP triggered functions. The HTTP trigger template for version 3.x (in-process) looks like the following example:

C#

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

namespace Company.Function
{
    public static class HttpTriggerCSharp
    {
```

```

    [FunctionName("HttpTriggerCSharp")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.AuthLevelValue, "get", "post",
Route = null)] HttpRequest req,
        ILogger log)
    {
        log.LogInformation("C# HTTP trigger function processed a
request.");

        string name = req.Query["name"];

        string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
        dynamic data = JsonConvert.DeserializeObject(requestBody);
        name = name ?? data?.name;

        string responseMessage = string.IsNullOrEmpty(name)
            ? "This HTTP triggered function executed successfully. Pass
a name in the query string or in the request body for a personalized
response."
            : $"Hello, {name}. This HTTP triggered function executed
successfully.";

        return new OkObjectResult(responseMessage);
    }
}
}

```

The HTTP trigger template for the migrated version looks like the following example:

.NET 8

C#

```

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace Company.Function
{
    public class HttpTriggerCSharp
    {
        private readonly ILogger<HttpTriggerCSharp> _logger;

        public HttpTriggerCSharp(ILogger<HttpTriggerCSharp> logger)
        {
            _logger = logger;
        }

        [Function("HttpTriggerCSharp")]
        public IActionResult Run(

```

```
[HttpTrigger(AuthorizationLevel.Function, "get")]
HttpRequest req
{
    _logger.LogInformation("C# HTTP trigger function processed a
    request.");
    return new OkObjectResult($"Welcome to Azure Functions,
    {req.Query["name"]}!");
}
}
```

Run the pre-upgrade validator

Azure Functions provides a pre-upgrade validator to help you identify potential issues when migrating your function app to 4.x. To run the pre-upgrade validator:

1. In the [Azure portal](#), navigate to your function app.
2. Open the [Diagnose and solve problems](#) page.
3. In **Function App Diagnostics**, start typing `Functions 4.x Pre-Upgrade Validator` and then choose it from the list.
4. After validation completes, review the recommendations and address any issues in your app. If you need to make changes to your app, make sure to validate the changes against version 4.x of the Functions runtime, either [locally using Azure Functions Core Tools v4](#) or by [using a staging slot](#).

Update your function app in Azure

You need to update the runtime of the function app host in Azure to version 4.x before you publish your migrated project. The runtime version used by the Functions host is controlled by the `FUNCTIONS_EXTENSION_VERSION` application setting, but in some cases other settings must also be updated. Both code changes and changes to application settings require your function app to restart.

The easiest way is to [update without slots](#) and then republish your app project. You can also minimize the downtime in your app and simplify rollback by [updating using slots](#).

Update without slots

The simplest way to update to v4.x is to set the `FUNCTIONS_EXTENSION_VERSION` application setting to `~4` on your function app in Azure. You must follow a [different procedure](#) on a site with slots.

Azure CLI

```
az functionapp config appsettings set --settings  
FUNCTIONS_EXTENSION_VERSION=~4 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>
```

You must also set another setting, which differs between Windows and Linux.

Windows

When running on Windows, you also need to enable .NET 6.0, which is required by version 4.x of the runtime.

Azure CLI

```
az functionapp config set --net-framework-version v6.0 -g  
<RESOURCE_GROUP_NAME> -n <APP_NAME>
```

.NET 6 is required for function apps in any language running on Windows.

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group.

You can now republish your app project that has been migrated to run on version 4.x.

Update using slots

Using [deployment slots](#) is a good way to update your function app to the v4.x runtime from a previous version. By using a staging slot, you can run your app on the new runtime version in the staging slot and switch to production after verification. Slots also provide a way to minimize downtime during the update. If you need to minimize downtime, follow the steps in [Minimum downtime update](#).

After you've verified your app in the updated slot, you can swap the app and new version settings into production. This swap requires setting

`WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the production slot. How you add this setting affects the amount of downtime required for the update.

Standard update

If your slot-enabled function app can handle the downtime of a full restart, you can update the `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS` setting directly in the production slot. Because changing this setting directly in the production slot causes a restart that impacts availability, consider doing this change at a time of reduced traffic. You can then swap in the updated version from the staging slot.

The [Update-AzFunctionAppSetting](#) PowerShell cmdlet doesn't currently support slots. You must use Azure CLI or the Azure portal.

1. Use the following command to set `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the production slot:

Azure CLI

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME>
```

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group. This command causes the app running in the production slot to restart.

2. Use the following command to also set

`WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS` in the staging slot:

Azure CLI

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME> --slot <SLOT_NAME>
```

3. Use the following command to change `FUNCTIONS_EXTENSION_VERSION` and update the staging slot to the new runtime version:

Azure CLI

```
az functionapp config appsettings set --settings  
FUNCTIONS_EXTENSION_VERSION=~4 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>
```

```
--slot <SLOT_NAME>
```

4. Version 4.x of the Functions runtime requires .NET 6 in Windows. On Linux, .NET apps must also update to .NET 6. Use the following command so that the runtime can run on .NET 6:

Windows

When running on Windows, you also need to enable .NET 6.0, which is required by version 4.x of the runtime.

Azure CLI

```
az functionapp config set --net-framework-version v6.0 -g  
<RESOURCE_GROUP_NAME> -n <APP_NAME>
```

.NET 6 is required for function apps in any language running on Windows.

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group.

5. If your code project required any updates to run on version 4.x, deploy those updates to the staging slot now.
6. Confirm that your function app runs correctly in the updated staging environment before swapping.
7. Use the following command to swap the updated staging slot to production:

Azure CLI

```
az functionapp deployment slot swap -g <RESOURCE_GROUP_NAME> -n  
<APP_NAME> --slot <SLOT_NAME> --target-slot production
```

Minimum downtime update

To minimize the downtime in your production app, you can swap the `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS` setting from the staging slot into production. After that, you can swap in the updated version from a prewarmed staging slot.

1. Use the following command to set `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the staging slot:

```
Azure CLI
```

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME> --slot <SLOT_NAME>
```

2. Use the following commands to swap the slot with the new setting into production, and at the same time restore the version setting in the staging slot.

```
Azure CLI
```

```
az functionapp deployment slot swap -g <RESOURCE_GROUP_NAME> -n  
<APP_NAME> --slot <SLOT_NAME> --target-slot production  
az functionapp config appsettings set --settings  
FUNCTIONS_EXTENSION_VERSION=~3 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>  
--slot <SLOT_NAME>
```

You may see errors from the staging slot during the time between the swap and the runtime version being restored on staging. This error can happen because having `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` only in staging during a swap removes the `FUNCTIONS_EXTENSION_VERSION` setting in staging. Without the version setting, your slot is in a bad state. Updating the version in the staging slot right after the swap should put the slot back into a good state, and you can roll back your changes if needed. However, any rollback of the swap also requires you to directly remove `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` from production before the swap back to prevent the same errors in production seen in staging. This change in the production setting would then cause a restart.

3. Use the following command to again set

`WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the staging slot:

```
Azure CLI
```

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME> --slot <SLOT_NAME>
```

At this point, both slots have `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` set.

4. Use the following command to change `FUNCTIONS_EXTENSION_VERSION` and update the staging slot to the new runtime version:

Azure CLI

```
az functionapp config appsettings set --settings  
FUNCTIONS_EXTENSION_VERSION=~4 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>  
--slot <SLOT_NAME>
```

5. Version 4.x of the Functions runtime requires .NET 6 in Windows. On Linux, .NET apps must also update to .NET 6. Use the following command so that the runtime can run on .NET 6:

Windows

When running on Windows, you also need to enable .NET 6.0, which is required by version 4.x of the runtime.

Azure CLI

```
az functionapp config set --net-framework-version v6.0 -g  
<RESOURCE_GROUP_NAME> -n <APP_NAME>
```

.NET 6 is required for function apps in any language running on Windows.

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group.

6. If your code project required any updates to run on version 4.x, deploy those updates to the staging slot now.
7. Confirm that your function app runs correctly in the updated staging environment before swapping.
8. Use the following command to swap the updated and prewarmed staging slot to production:

Azure CLI

```
az functionapp deployment slot swap -g <RESOURCE_GROUP_NAME> -n  
<APP_NAME> --slot <SLOT_NAME> --target-slot production
```

Breaking changes between 3.x and 4.x

The following are key breaking changes to be aware of before upgrading a 3.x app to 4.x, including language-specific breaking changes. For a full list, see Azure Functions GitHub issues labeled [Breaking Change: Approved](#).

If you don't see your programming language, go select it from the [top of the page](#).

Runtime

- Azure Functions Proxies is a legacy feature for versions 1.x through 3.x of the Azure Functions runtime. Support for Functions Proxies can be re-enabled in version 4.x so that you can successfully update your function apps to the latest runtime version. As soon as possible, you should instead switch to integrating your function apps with Azure API Management. API Management lets you take advantage of a more complete set of features for defining, securing, managing, and monetizing your Functions-based APIs. For more information, see [API Management integration](#). To learn how to re-enable Proxies support in Functions version 4.x, see [Re-enable Proxies in Functions v4.x](#).
- Logging to Azure Storage using `AzureWebJobsDashboard` is no longer supported in 4.x. You should instead use [Application Insights](#). ([#1923](#))
- Azure Functions 4.x now enforces [minimum version requirements for extensions](#). Update to the latest version of affected extensions. For non-.NET languages, [update](#) to extension bundle version 2.x or later. ([#1987](#))
- Default and maximum timeouts are now enforced in 4.x for function apps running on Linux in a Consumption plan. ([#1915](#))
- Azure Functions 4.x uses `Azure.Identity` and `Azure.Security.KeyVault.Secrets` for the Key Vault provider and has deprecated the use of `Microsoft.Azure.KeyVault`. For more information about how to configure function app settings, see the Key Vault option in [Secret Repositories](#). ([#2048](#))
- Function apps that share storage accounts now fail to start when their host IDs are the same. For more information, see [Host ID considerations](#). ([#2049](#))
- Azure Functions 4.x supports .NET 6 in-process and isolated apps.
- `InvalidHostServicesException` is now a fatal error. ([#2045](#))
- `EnableEnhancedScopes` is enabled by default. ([#1954](#))
- Remove `HttpClient` as a registered service. ([#1911](#))

Next steps

[Learn more about Functions versions](#)

Migrate apps from Azure Functions version 1.x to version 4.x

Article • 03/29/2024

ⓘ Important

[Support will end for version 1.x of the Azure Functions runtime on September 14, 2026](#). We highly recommend that you migrate your apps to version 4.x by following the instructions in this article.

This article walks you through the process of safely migrating your function app to run on version 4.x of the Functions runtime. Because project migration instructions are language dependent, make sure to choose your development language from the selector at the [top of the article](#).

If you are running version 1.x of the runtime in Azure Stack Hub, see [Considerations for Azure Stack Hub](#) first.

Identify function apps to migrate

Use the following PowerShell script to generate a list of function apps in your subscription that currently target version 1.x:

PowerShell

```
$Subscription = '<YOUR SUBSCRIPTION ID>'  
  
Set-AzContext -Subscription $Subscription | Out-Null  
  
$FunctionApps = Get-AzFunctionApp  
  
$AppInfo = @{}  
  
foreach ($App in $FunctionApps)  
{  
    if ($App.ApplicationSettings["FUNCTIONS_EXTENSION_VERSION"] -like  
'*1*')  
    {  
        $AppInfo.Add($App.Name,  
$App.ApplicationSettings["FUNCTIONS_EXTENSION_VERSION"])  
    }  
}
```

Choose your target .NET version

On version 1.x of the Functions runtime, your C# function app targets .NET Framework.

When you migrate your function app, you have the opportunity to choose the target version of .NET. You can update your C# project to one of the following versions of .NET that are supported by Functions version 4.x:

[Expand table](#)

.NET version	.NET Official Support Policy <small>↗</small> release type	Functions process model ^{1,3}
.NET 8 ²	LTS	Isolated worker model
.NET 7	STS (end of support May 14, 2024)	Isolated worker model
.NET 6	LTS (end of support November 12, 2024)	Isolated worker model, In-process model ³
.NET Framework 4.8	See policy ↗	Isolated worker model

¹ The [isolated worker model](#) supports Long Term Support (LTS) and Standard Term Support (STS) versions of .NET, as well as .NET Framework. The [in-process model](#) only supports LTS releases of .NET. For a full feature and functionality comparison between the two models, see [Differences between in-process and isolate worker process .NET Azure Functions](#).

² .NET 8 is not yet supported on the in-process model, though it is available on the isolated worker model. For information about .NET 8 plans, including future options for the in-process model, see the [Azure Functions Roadmap Update post ↗](#).

³ Support ends for the in-process model on November 10, 2026. For more information, see [this support announcement ↗](#). For continued full support, you should [migrate your apps to the isolated worker model](#).

Tip

Unless your app depends on a library or API only available to .NET Framework, we recommend updating to .NET 8 on the isolated worker model. Many apps on

version 1.x target .NET Framework only because that is what was available when they were created. Additional capabilities are available to more recent versions of .NET, and if your app is not forced to stay on .NET Framework due to a dependency, you should target a more recent version. .NET 8 is the fully released version with the longest support window from .NET.

Although you can choose to instead use the in-process model, this is not recommended if it can be avoided. [Support will end for the in-process model on November 10, 2026](#), so you'll need to move to the isolated worker model before then. Doing so while migrating to version 4.x will decrease the total effort required, and the isolated worker model will give your app [additional benefits](#), including the ability to more easily target future versions of .NET. If you are moving to the isolated worker model, the [.NET Upgrade Assistant](#) can also handle many of the necessary code changes for you.

This guide doesn't present specific examples for .NET 7 or .NET 6 on the isolated worker model. If you need to target these versions, you can adapt the .NET 8 isolated worker model examples.

Prepare for migration

If you haven't already, identify the list of apps that need to be migrated in your current Azure Subscription by using the [Azure PowerShell](#).

Before you migrate an app to version 4.x of the Functions runtime, you should do the following tasks:

1. Review the list of [behavior changes after version 1.x](#). Migrating from version 1.x to version 4.x also can affect bindings.
2. Complete the steps in [Migrate your local project](#) to migrate your local project to version 4.x.
3. After migrating your project, fully test the app locally using version 4.x of the [Azure Functions Core Tools](#).
4. Update your function app in Azure to the new version. If you need to minimize downtime, consider using a [staging slot](#) to test and verify your migrated app in Azure on the new runtime version. You can then deploy your app with the updated version settings to the production slot. For more information, see [Update using slots](#).
5. Publish your migrated project to the updated function app.

When you use Visual Studio to publish a version 4.x project to an existing function app at a lower version, you're prompted to let Visual Studio update the function app to version 4.x during deployment. This update uses the same process defined in [Update without slots](#).

Migrate your local project

The following sections describes the updates you must make to your C# project files to be able to run on one of the supported versions of .NET in Functions version 4.x. The updates shown are ones common to most projects. Your project code could require updates not mentioned in this article, especially when using custom NuGet packages.

Migrating a C# function app from version 1.x to version 4.x of the Functions runtime requires you to make changes to your project code. Many of these changes are a result of changes in the C# language and .NET APIs.

Choose the tab that matches your target version of .NET and the desired process model (in-process or isolated worker process).

Tip

If you are moving to an LTS or STS version of .NET using the isolated worker model, the [.NET Upgrade Assistant](#) can be used to automatically make many of the changes mentioned in the following sections.

Project file

The following example is a `.csproj` project file that runs on version 1.x:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net48</TargetFramework>
    <AzureFunctionsVersion>v1</AzureFunctionsVersion>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Sdk.Functions" Version="1.0.24" />
  </ItemGroup>
  <ItemGroup>
    <Reference Include="Microsoft.CSharp" />
  </ItemGroup>
  <ItemGroup>
```

```
<None Update="host.json">
  <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</None>
<None Update="local.settings.json">
  <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  <CopyToPublishDirectory>Never</CopyToPublishDirectory>
</None>
</ItemGroup>
</Project>
```

Use one of the following procedures to update this XML file to run in Functions version 4.x:

.NET 8

These steps assume a local C# project, and if your app is instead using C# script (.csx files), you should [convert to the project model](#) before continuing.

The following changes are required in the .csproj XML project file:

1. Set the value of `PropertyGroup.TargetFramework` to `net8.0`.
2. Set the value of `PropertyGroup.AzureFunctionsVersion` to `v4`.
3. Add the following `OutputType` element to the `PropertyGroup`:

XML

```
<OutputType>Exe</OutputType>
```

4. In the `ItemGroup.PackageReference` list, replace the package reference to `Microsoft.NET.Sdk.Functions` with the following references:

XML

```
<FrameworkReference Include="Microsoft.AspNetCore.App" />
<PackageReference Include="Microsoft.Azure.Functions.Worker"
Version="1.21.0" />
<PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
Version="1.17.2" />
<PackageReference
Include="Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore"
Version="1.2.1" />
<PackageReference
Include="Microsoft.ApplicationInsights.WorkerService"
Version="2.22.0" />
<PackageReference
```

```
Include="Microsoft.Azure.Functions.Worker.ApplicationInsights"
Version="1.2.0" />
```

Make note of any references to other packages in the `Microsoft.Azure.WebJobs.*` namespaces. You'll replace these packages in a later step.

5. Add the following new `ItemGroup`:

XML

```
<ItemGroup>
  <Using Include="System.Threading.ExecutionContext"
    Alias="ExecutionContext"/>
</ItemGroup>
```

After you make these changes, your updated project should look like the following example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
    <RootNamespace>My.Namespace</RootNamespace>
    <OutputType>Exe</OutputType>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker"
      Version="1.21.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
      Version="1.17.2" />
    <PackageReference
      Include="Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore"
      Version="1.2.1" />
    <PackageReference
      Include="Microsoft.ApplicationInsights.WorkerService" Version="2.22.0"
      />
    <PackageReference
      Include="Microsoft.Azure.Functions.Worker.ApplicationInsights"
      Version="1.2.0" />
    <!-- Other packages may also be in this list -->
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
  </ItemGroup>
</Project>
```

```

    </None>
<None Update="local.settings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <CopyToPublishDirectory>Never</CopyToPublishDirectory>
</None>
</ItemGroup>
<ItemGroup>
    <Using Include="System.Threading.ExecutionContext"
Alias="ExecutionContext"/>
</ItemGroup>
</Project>

```

Package and namespace changes

Based on the model you are migrating to, you might need to update or change the packages your application references. When you adopt the target packages, you then need to update the namespace of using statements and some types you reference. You can see the effect of these namespace changes on `using` statements in the [HTTP trigger template examples](#) later in this article.

.NET 8

If you haven't already, update your project to reference the latest stable versions of:

- [Microsoft.Azure.Functions.Worker](#)
- [Microsoft.Azure.Functions.Worker.Sdk](#)

Depending on the triggers and bindings your app uses, your app might need to reference a different set of packages. The following table shows the replacements for some of the most commonly used extensions:

[Expand table](#)

Scenario	Changes to package references
Timer trigger	Add Microsoft.Azure.Functions.Worker.Extensions.Timer
Storage bindings	Replace <code>Microsoft.Azure.WebJobs.Extensions.Storage</code> with Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs , Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues , and Microsoft.Azure.Functions.Worker.Extensions.Tables

Scenario	Changes to package references
Blob bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Storage.Blobs</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs
Queue bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Storage.Queues</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues
Table bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Tables</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Tables
Cosmos DB bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.CosmosDB</code> and/or <code>Microsoft.Azure.WebJobs.Extensions.DocumentDB</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.CosmosDB
Service Bus bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.ServiceBus</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.ServiceBus
Event Hubs bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.EventHubs</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.EventHubs
Event Grid bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.EventGrid</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.EventGrid
SignalR Service bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.SignalRService</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.SignalRService
Durable Functions	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.DurableTask</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask

Scenario	Changes to package references
Durable Functions (SQL storage provider)	Replace references to <code>Microsoft.DurableTask.SqlServer.AzureFunctions</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask.SqlServer
Durable Functions (Netherite storage provider)	Replace references to <code>Microsoft.Azure.DurableTask.Netherite.AzureFunctions</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.DurableTask.Netherite
SendGrid bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.SendGrid</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.SendGrid
Kafka bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.Kafka</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.Kafka
RabbitMQ bindings	Replace references to <code>Microsoft.Azure.WebJobs.Extensions.RabbitMQ</code> with the latest version of Microsoft.Azure.Functions.Worker.Extensions.RabbitMQ
Dependency injection and startup config	Remove references to <code>Microsoft.Azure.Functions.Extensions</code> (The isolated worker model provides this functionality by default.)

See [Supported bindings](#) for a complete list of extensions to consider, and consult each extension's documentation for full installation instructions for the isolated process model. Be sure to install the latest stable version of any packages you are targeting.

Your isolated worker model application should not reference any packages in the `Microsoft.Azure.WebJobs.*` namespaces or `Microsoft.Azure.Functions.Extensions`. If you have any remaining references to these, they should be removed.

💡 Tip

Your app might also depend on Azure SDK types, either as part of your triggers and bindings or as a standalone dependency. You should take this opportunity to update these as well. The latest versions of the Functions extensions work

with the latest versions of the [Azure SDK for .NET](#), almost all of the packages for which are the form `Azure.*`.

The [Notification Hubs](#) and [Mobile Apps](#) bindings are supported only in version 1.x of the runtime. When upgrading to version 4.x of the runtime, you need to remove these bindings in favor of working with these services directly using their SDKs.

Program.cs file

In most cases, migrating requires you to add the following program.cs file to your project:

```
.NET 8

C#  
  
using Microsoft.Azure.Functions.Worker;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
  
var host = new HostBuilder()  
    .ConfigureFunctionsWebApplication()  
    .ConfigureServices(services => {  
        services.AddApplicationInsightsTelemetryWorkerService();  
        services.ConfigureFunctionsApplicationInsights();  
    })  
    .Build();  
  
host.Run();
```

This example includes [ASP.NET Core integration](#) to improve performance and provide a familiar programming model when your app uses HTTP triggers. If you do not intend to use HTTP triggers, you can replace the call to `ConfigureFunctionsWebApplication` with a call to `ConfigureFunctionsWorkerDefaults`. If you do so, you can remove the reference to `Microsoft.Azure.Functions.Worker.Extensions.Http.AspNetCore` from your project file. However, for the best performance, even for functions with other trigger types, you should keep the `FrameworkReference` to ASP.NET Core.

The `Program.cs` file will replace any file that has the `FunctionsStartup` attribute, which is typically a `Startup.cs` file. In places where your `FunctionsStartup` code would reference `IFunctionsHostBuilder.Services`, you can instead add statements

within the `.ConfigureServices()` method of the `HostBuilder` in your `Program.cs`. To learn more about working with `Program.cs`, see [Start-up and configuration](#) in the isolated worker model guide.

The default `Program.cs` examples above include setup of [Application Insights integration for the isolated worker model](#). In your `Program.cs`, you must also configure any log filtering that should apply to logs coming from code in your project. In the isolated worker model, the `host.json` file only controls events emitted by the Functions host runtime. If you don't configure filtering rules in `Program.cs`, you may see differences in the log levels present for various categories in your telemetry.

Although you can register custom configuration sources as part of the `HostBuilder`, note that these similarly apply only to code in your project. Trigger and binding configuration is also needed by the platform, and this should be provided through the [application settings](#), [Key Vault references](#), or [App Configuration references](#) features.

Once you have moved everything from any existing `FunctionsStartup` to the `Program.cs` file, you can delete the `FunctionsStartup` attribute and the class it was applied to.

host.json file

Settings in the `host.json` file apply at the function app level, both locally and in Azure. In version 1.x, your `host.json` file is either empty or it contains some settings that apply to all functions in the function app. For more information, see [Host.json v1](#). If your `host.json` file has setting values, review the [host.json v2 format](#) for any changes.

To run on version 4.x, you must add `"version": "2.0"` to the `host.json` file. You should also consider adding `logging` to your configuration, as in the following examples:

.NET 8

JSON

```
{  
    "version": "2.0",  
    "logging": {  
        "applicationInsights": {  
            "samplingSettings": {  
                "isEnabled": true,  
                "maxTelemetryItemsPerSecond": 1000  
            }  
        }  
    }  
}
```

```
        "excludedTypes": "Request"
    },
    "enableLiveMetricsFilters": true
}
}
```

The `host.json` file only controls logging from the Functions host runtime, and in the isolated worker model, some of these logs come from your application directly, giving you more control. See [Managing log levels in the isolated worker model](#) for details on how to filter these logs.

local.settings.json file

The `local.settings.json` file is only used when running locally. For information, see [Local settings file](#). In version 1.x, the `local.settings.json` file has only two required values:

JSON

```
{
    "IsEncrypted": false,
    "Values": {
        "AzureWebJobsStorage": "AzureWebJobsStorageConnectionStringValue",
        "AzureWebJobsDashboard": "AzureWebJobsStorageConnectionStringValue"
    }
}
```

When you migrate to version 4.x, make sure that your `local.settings.json` file has at least the following elements:

.NET 8

JSON

```
{
    "IsEncrypted": false,
    "Values": {
        "AzureWebJobsStorage":
            "AzureWebJobsStorageConnectionStringValue",
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
    }
}
```

 Note

When migrating from running in-process to running in an isolated worker process, you need to change the `FUNCTIONS_WORKER_RUNTIME` value to "dotnet-isolated".

Class name changes

Some key classes changed names between version 1.x and version 4.x. These changes are a result either of changes in .NET APIs or in differences between in-process and isolated worker process. The following table indicates key .NET classes used by Functions that could change when migrating:

.NET 8

 Expand table

Version 1.x	.NET 8
<code>FunctionName</code> (attribute)	<code>Function</code> (attribute)
<code>TraceWriter</code>	<code>ILogger<T></code> , <code>ILogger</code>
<code>HttpRequestMessage</code>	<code>HttpRequestData</code> , <code>HttpRequest</code> (using ASP.NET Core integration)
<code>HttpResponseMessage</code>	<code>HttpResponseData</code> , <code>IActionResult</code> (using ASP.NET Core integration)

There might also be class name differences in bindings. For more information, see the reference articles for the specific bindings.

Other code changes

.NET 8 (isolated)

This section highlights other code changes to consider as you work through the migration. These changes are not needed by all applications, but you should evaluate if any are relevant to your scenarios. Make sure to check [Behavior changes after version 1.x](#) for additional changes you might need to make to your project.

JSON serialization

By default, the isolated worker model uses `System.Text.Json` for JSON serialization. To customize serializer options or switch to JSON.NET (`Newtonsoft.Json`), see [these instructions](#).

Application Insights log levels and filtering

Logs can be sent to Application Insights from both the Functions host runtime and code in your project. The `host.json` allows you to configure rules for host logging, but to control logs coming from your code, you'll need to configure filtering rules as part of your `Program.cs`. See [Managing log levels in the isolated worker model](#) for details on how to filter these logs.

HTTP trigger template

Most of the code changes between version 1.x and version 4.x can be seen in HTTP triggered functions. The HTTP trigger template for version 1.x looks like the following example:

C#

```
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;

namespace Company.Function
{
    public static class HttpTriggerCSharp
    {
        [FunctionName("HttpTriggerCSharp")]
        public static async Task<HttpResponseMessage>
            Run([HttpTrigger(AuthorizationLevel.AuthHeaderValue, "get",
"post",
                    Route = null)]HttpRequestMessage req, TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");

            // parse query parameter
            string name = req.GetQueryNameValuePairs()
                .FirstOrDefault(q => string.Compare(q.Key, "name", true) ==
0)
                .Value;
```

```

    if (name == null)
    {
        // Get request body
        dynamic data = await req.Content.ReadAsAsync<object>();
        name = data?.name;
    }

    return name == null
        ? req.CreateResponse(HttpStatusCode.BadRequest,
            "Please pass a name on the query string or in the
request body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
    }
}

```

In version 4.x, the HTTP trigger template looks like the following example:

.NET 8

C#

```

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace Company.Function
{
    public class HttpTriggerCSharp
    {
        private readonly ILogger<HttpTriggerCSharp> _logger;

        public HttpTriggerCSharp(ILogger<HttpTriggerCSharp> logger)
        {
            _logger = logger;
        }

        [Function("HttpTriggerCSharp")]
        public IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Function, "get")]
HttpRequest req)
        {
            _logger.LogInformation("C# HTTP trigger function processed a
request.");

            return new OkObjectResult($"Welcome to Azure Functions,
{req.Query["name"]}!");
        }
    }
}

```

```
}
```

Update your function app in Azure

You need to update the runtime of the function app host in Azure to version 4.x before you publish your migrated project. The runtime version used by the Functions host is controlled by the `FUNCTIONS_EXTENSION_VERSION` application setting, but in some cases other settings must also be updated. Both code changes and changes to application settings require your function app to restart.

The easiest way is to [update without slots](#) and then republish your app project. You can also minimize the downtime in your app and simplify rollback by [updating using slots](#).

Update without slots

The simplest way to update to v4.x is to set the `FUNCTIONS_EXTENSION_VERSION` application setting to `~4` on your function app in Azure. You must follow a [different procedure](#) on a site with slots.

Azure CLI

```
Azure CLI
```

```
az functionapp config appsettings set --settings
FUNCTIONS_EXTENSION_VERSION=~4 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>
```

You must also set another setting, which differs between Windows and Linux.

Windows

When running on Windows, you also need to enable .NET 6.0, which is required by version 4.x of the runtime.

Azure CLI

```
az functionapp config set --net-framework-version v6.0 -g
<RESOURCE_GROUP_NAME> -n <APP_NAME>
```

.NET 6 is required for function apps in any language running on Windows.

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group.

You can now republish your app project that has been migrated to run on version 4.x.

Update using slots

Using [deployment slots](#) is a good way to update your function app to the v4.x runtime from a previous version. By using a staging slot, you can run your app on the new runtime version in the staging slot and switch to production after verification. Slots also provide a way to minimize downtime during the update. If you need to minimize downtime, follow the steps in [Minimum downtime update](#).

After you've verified your app in the updated slot, you can swap the app and new version settings into production. This swap requires setting `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the production slot. How you add this setting affects the amount of downtime required for the update.

Standard update

If your slot-enabled function app can handle the downtime of a full restart, you can update the `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS` setting directly in the production slot. Because changing this setting directly in the production slot causes a restart that impacts availability, consider doing this change at a time of reduced traffic. You can then swap in the updated version from the staging slot.

The [Update-AzFunctionAppSetting](#) PowerShell cmdlet doesn't currently support slots. You must use Azure CLI or the Azure portal.

1. Use the following command to set `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the production slot:

Azure CLI

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME>
```

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group. This command

causes the app running in the production slot to restart.

2. Use the following command to also set

```
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS
```

 in the staging slot:

Azure CLI

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME> --slot <SLOT_NAME>
```

3. Use the following command to change `FUNCTIONS_EXTENSION_VERSION` and update the staging slot to the new runtime version:

Azure CLI

```
az functionapp config appsettings set --settings  
FUNCTIONS_EXTENSION_VERSION=~4 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>  
--slot <SLOT_NAME>
```

4. Version 4.x of the Functions runtime requires .NET 6 in Windows. On Linux, .NET apps must also update to .NET 6. Use the following command so that the runtime can run on .NET 6:

Windows

When running on Windows, you also need to enable .NET 6.0, which is required by version 4.x of the runtime.

Azure CLI

```
az functionapp config set --net-framework-version v6.0 -g  
<RESOURCE_GROUP_NAME> -n <APP_NAME>
```

.NET 6 is required for function apps in any language running on Windows.

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group.

5. If your code project required any updates to run on version 4.x, deploy those updates to the staging slot now.
6. Confirm that your function app runs correctly in the updated staging environment before swapping.

7. Use the following command to swap the updated staging slot to production:

Azure CLI

```
az functionapp deployment slot swap -g <RESOURCE_GROUP_NAME> -n <APP_NAME> --slot <SLOT_NAME> --target-slot production
```

Minimum downtime update

To minimize the downtime in your production app, you can swap the `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS` setting from the staging slot into production. After that, you can swap in the updated version from a prewarmed staging slot.

1. Use the following command to set `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` in the staging slot:

Azure CLI

```
az functionapp config appsettings set --settings WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME> -n <APP_NAME> --slot <SLOT_NAME>
```

2. Use the following commands to swap the slot with the new setting into production, and at the same time restore the version setting in the staging slot.

Azure CLI

```
az functionapp deployment slot swap -g <RESOURCE_GROUP_NAME> -n <APP_NAME> --slot <SLOT_NAME> --target-slot production
az functionapp config appsettings set --settings FUNCTIONS_EXTENSION_VERSION=~3 -g <RESOURCE_GROUP_NAME> -n <APP_NAME> --slot <SLOT_NAME>
```

You may see errors from the staging slot during the time between the swap and the runtime version being restored on staging. This error can happen because having `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` only in staging during a swap removes the `FUNCTIONS_EXTENSION_VERSION` setting in staging. Without the version setting, your slot is in a bad state. Updating the version in the staging slot right after the swap should put the slot back into a good state, and you can roll back your changes if needed. However, any rollback of the swap also requires you to directly remove `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` from

production before the swap back to prevent the same errors in production seen in staging. This change in the production setting would then cause a restart.

3. Use the following command to again set

```
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0
```

 in the staging slot:

Azure CLI

```
az functionapp config appsettings set --settings  
WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0 -g <RESOURCE_GROUP_NAME>  
-n <APP_NAME> --slot <SLOT_NAME>
```

At this point, both slots have `WEBSITE_OVERRIDE_STICKY_EXTENSION_VERSIONS=0` set.

4. Use the following command to change `FUNCTIONS_EXTENSION_VERSION` and update the staging slot to the new runtime version:

Azure CLI

```
az functionapp config appsettings set --settings  
FUNCTIONS_EXTENSION_VERSION=~4 -g <RESOURCE_GROUP_NAME> -n <APP_NAME>  
--slot <SLOT_NAME>
```

5. Version 4.x of the Functions runtime requires .NET 6 in Windows. On Linux, .NET apps must also update to .NET 6. Use the following command so that the runtime can run on .NET 6:

Windows

When running on Windows, you also need to enable .NET 6.0, which is required by version 4.x of the runtime.

Azure CLI

```
az functionapp config set --net-framework-version v6.0 -g  
<RESOURCE_GROUP_NAME> -n <APP_NAME>
```

.NET 6 is required for function apps in any language running on Windows.

In this example, replace `<APP_NAME>` with the name of your function app and `<RESOURCE_GROUP_NAME>` with the name of the resource group.

6. If your code project required any updates to run on version 4.x, deploy those updates to the staging slot now.

7. Confirm that your function app runs correctly in the updated staging environment before swapping.
8. Use the following command to swap the updated and prewarmed staging slot to production:

```
Azure CLI
```

```
az functionapp deployment slot swap -g <RESOURCE_GROUP_NAME> -n <APP_NAME> --slot <SLOT_NAME> --target-slot production
```

Behavior changes after version 1.x

This section details changes made after version 1.x in both trigger and binding behaviors as well as in core Functions features and behaviors.

Changes in triggers and bindings

Starting with version 2.x, you must install the extensions for specific triggers and bindings used by the functions in your app. The only exception for this HTTP and timer triggers, which don't require an extension. For more information, see [Register and install binding extensions](#).

There are also a few changes in the *function.json* or attributes of the function between versions. For example, the Event Hubs `path` property is now `eventHubName`. See the [existing binding table](#) for links to documentation for each binding.

Changes in features and functionality

A few features were removed, updated, or replaced after version 1.x. This section details the changes you see in later versions after having used version 1.x.

In version 2.x, the following changes were made:

- Keys for calling HTTP endpoints are always stored encrypted in Azure Blob storage. In version 1.x, keys were stored in Azure Files by default. When you migrate an app from version 1.x to version 2.x, existing secrets that are in Azure Files are reset.
- The version 2.x runtime doesn't include built-in support for webhook providers. This change was made to improve performance. You can still use HTTP triggers as endpoints for webhooks.

- To improve monitoring, the WebJobs dashboard in the portal, which used the [AzureWebJobsDashboard](#) setting is replaced with Azure Application Insights, which uses the [APPINSIGHTS_INSTRUMENTATIONKEY](#) setting. For more information, see [Monitor Azure Functions](#).
- All functions in a function app must share the same language. When you create a function app, you must choose a runtime stack for the app. The runtime stack is specified by the [FUNCTIONS_WORKER_RUNTIME](#) value in application settings. This requirement was added to improve footprint and startup time. When developing locally, you must also include this setting in the [local.settings.json file](#).
- The default timeout for functions in an App Service plan is changed to 30 minutes. You can manually change the timeout back to unlimited by using the [functionTimeout](#) setting in host.json.
- HTTP concurrency throttles are implemented by default for Consumption plan functions, with a default of 100 concurrent requests per instance. You can change this behavior in the [maxConcurrentRequests](#) setting in the host.json file.
- Because of [.NET Core limitations](#), support for F# script (.fsx files) functions has been removed. Compiled F# functions (.fs) are still supported.
- The URL format of Event Grid trigger webhooks has been changed to follow this pattern: `https://{{app}}/runtime/webhooks/{{triggerName}}`.
- The names of some [pre-defined custom metrics](#) were changed after version 1.x. `Duration` was replaced with `MaxDurationMs`, `MinDurationMs`, and `AvgDurationMs`. `Success Rate` was also renamed to `Success Rate`.

Considerations for Azure Stack Hub

[App Service on Azure Stack Hub](#) does not support version 4.x of Azure Functions. When you are planning a migration off of version 1.x in Azure Stack Hub, you can choose one of the following options:

- Migrate to version 4.x hosted in public cloud Azure Functions using the instructions in this article. Instead of upgrading your existing app, you would create a new app using version 4.x and then deploy your modified project to it.
- Switch to [WebJobs](#) hosted on an App Service plan in Azure Stack Hub.

Next steps

[Learn more about Functions versions](#)

Migrate to version 4 of the Node.js programming model for Azure Functions

Article • 01/18/2024

This article discusses the differences between version 3 and version 4 of the Node.js programming model and how to upgrade an existing v3 app. If you want to create a new v4 app instead of upgrading an existing v3 app, see the tutorial for either [Visual Studio Code \(VS Code\)](#) or [Azure Functions Core Tools](#). This article uses "tip" alerts to highlight the most important concrete actions that you should take to upgrade your app.

Version 4 is designed to provide Node.js developers with the following benefits:

- Provide a familiar and intuitive experience to Node.js developers.
- Make the file structure flexible with support for full customization.
- Switch to a code-centric approach for defining function configuration.

Considerations

- The Node.js programming model shouldn't be confused with the Azure Functions runtime:
 - **Programming model:** Defines how you author your code and is specific to JavaScript and TypeScript.
 - **Runtime:** Defines underlying behavior of Azure Functions and is shared across all languages.
- The version of the programming model is strictly tied to the version of the [@azure/functions](#) npm package. It's versioned independently of the [runtime](#). Both the runtime and the programming model use the number 4 as their latest major version, but that's a coincidence.
- You can't mix the v3 and v4 programming models in the same function app. As soon as you register one v4 function in your app, any v3 functions registered in *function.json* files are ignored.

Requirements

Version 4 of the Node.js programming model requires the following minimum versions:

- [@azure/functions](#) npm package v4.0.0

- [Node.js](#) v18+
- [Azure Functions Runtime](#) v4.25+
- [Azure Functions Core Tools](#) v4.0.5382+ (if running locally)

Include the npm package

In v4, the [@azure/functions](#) npm package contains the primary source code that backs the Node.js programming model. In previous versions, that code shipped directly in Azure and the npm package had only the TypeScript types. You now need to include this package for both TypeScript and JavaScript apps. You *can* include the package for existing v3 apps, but it isn't required.

Tip

Make sure the `@azure/functions` package is listed in the `dependencies` section (not `devDependencies`) of your `package.json` file. You can install v4 by using the following command:

```
npm install @azure/functions
```

Set your app entry point

In v4 of the programming model, you can structure your code however you want. The only files that you need at the root of your app are `host.json` and `package.json`.

Otherwise, you define the file structure by setting the `main` field in your `package.json` file. You can set the `main` field to a single file or multiple files by using a [glob pattern](#). The following table shows example values for the `main` field:

[\[+\] Expand table](#)

Example	Description
<code>src/index.js</code>	Register functions from a single root file.
<code>src/functions/*.js</code>	Register each function from its own file.
<code>src/{index.js,functions/*.js}</code>	A combination where you register each function from its own file, but you still have a root file for general app-level code.

💡 Tip

Make sure you define a `main` field in your `package.json` file.

Switch the order of arguments

The trigger input, instead of the invocation context, is now the first argument to your function handler. The invocation context, now the second argument, is simplified in v4 and isn't as required as the trigger input. You can leave it off if you aren't using it.

💡 Tip

Switch the order of your arguments. For example, if you're using an HTTP trigger, switch `(context, request)` to either `(request, context)` or just `(request)` if you aren't using the context.

Define your function in code

You no longer have to create and maintain those separate `function.json` configuration files. You can now fully define your functions directly in your TypeScript or JavaScript files. In addition, many properties now have defaults so that you don't have to specify them every time.

v4

JavaScript

```
const { app } = require('@azure/functions');

app.http('httpTrigger1', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    handler: async (request, context) => {
        context.log(`Http function processed request for url
        ${request.url}`);
        const name = request.query.get('name') || (await request.text())
        || 'world';

        return { body: `Hello, ${name}!` };
    }
});
```

```
  },
});
```

💡 Tip

Move the configuration from your `function.json` file to your code. The type of the trigger corresponds to a method on the `app` object in the new model. For example, if you use an `httpTrigger` type in `function.json`, call `app.http()` in your code to register the function. If you use `timerTrigger`, call `app.timer()`.

Review your usage of context

In v4, the `context` object is simplified to reduce duplication and to make writing unit tests easier. For example, we streamlined the primary input and output so that they're accessed only as the argument and return value of your function handler.

You can't access the primary input and output on the `context` object anymore, but you must still access *secondary* inputs and outputs on the `context` object. For more information about secondary inputs and outputs, see the [Node.js developer guide](#).

Get the primary input as an argument

The primary input is also called the *trigger* and is the only required input or output. You must have one (and only one) trigger.

v4

Version 4 supports only one way of getting the trigger input, as the first argument:

JavaScript

```
async function httpTrigger1(request, context) {
  const onlyOption = request;
```

💡 Tip

Make sure you aren't using `context.req` or `context.bindings` to get the input.

Set the primary output as your return value

v4

Version 4 supports only one way of setting the primary output, through the return value:

JavaScript

```
return {  
  body: `Hello, ${name}!`  
};
```



Make sure you always return the output in your function handler, instead of setting it with the `context` object.

Context logging

In v4, logging methods were moved to the root `context` object as shown in the following example. For more information about logging, see the [Node.js developer guide](#).

v4

JavaScript

```
context.log('This is an info log');  
context.error('This is an error');  
context.warn('This is an error');
```

Create a test context

Version 3 doesn't support creating an invocation context outside the Azure Functions runtime, so authoring unit tests can be difficult. Version 4 allows you to create an instance of the invocation context, although the information during tests isn't detailed unless you add it yourself.

v4

JavaScript

```
const testInvocationContext = new InvocationContext({
  functionName: 'testFunctionName',
  invocationId: 'testInvocationId'
});
```

Review your usage of HTTP types

The HTTP request and response types are now a subset of the [fetch standard](#). They're no longer unique to Azure Functions.

The types use the [undici](#) package in Node.js. This package follows the fetch standard and is [currently being integrated](#) into Node.js core.

HttpRequest

v4

- *Body*. You can access the body by using a method specific to the type that you want to receive:

JavaScript

```
const body = await request.text();
const body = await request.json();
const body = await request.formData();
const body = await request.arrayBuffer();
const body = await request.blob();
```

- *Header*:

JavaScript

```
const header = request.headers.get('content-type');
```

- *Query parameter*:

JavaScript

```
const name = request.query.get('name');
```

HttpResponse

v4

- *Status*:

JavaScript

```
return { status: 200 };
```

- *Body*:

Use the `body` property to return most types like a `string` or `Buffer`:

JavaScript

```
return { body: "Hello, world!" };
```

Use the `jsonBody` property for the easiest way to return a JSON response:

JavaScript

```
return { jsonBody: { hello: "world" } };
```

- *Header*. You can set the header in two ways, depending on whether you're using the `HttpResponse` class or the `HttpResponseInit` interface:

JavaScript

```
const response = new HttpResponse();
response.headers.set('content-type', 'application/json');
return response;
```

JavaScript

```
return {  
  headers: { 'content-type': 'application/json' }  
};
```

💡 Tip

Update any logic by using the HTTP request or response types to match the new methods.

Troubleshoot

See the [Node.js Troubleshoot guide](#).

Update language stack versions in Azure Functions

Article • 07/05/2024

The support for any given language stack in Azure Functions is limited to [specific versions](#). As new versions become available, you might want to update your apps to take advantage of their features. Support in Functions may also end for older versions, typically aligned to the community end-of-support timelines. See the [Language runtime support policy](#) for details. To ensure your apps continue to receive support, you should follow the instructions outlined in this article to update them to the latest available versions.

The way that you update your function app depends on:

- The language you use to author your functions; make sure to choose your programming language at the [top](#) of the article.
- The operating system on which your app runs in Azure: Windows or Linux.
- The [hosting plan](#).

Note

This article shows you how to update the .NET version of an app using the [isolated worker model](#). Apps that run on older versions of .NET with [the in-process model](#) can [update to target .NET 8](#), or they can [migrate from the in-process model to the isolated worker model](#).

Prepare to update

Before you update the stack configuration for your function app in Azure, you should complete these tasks:

1. Verify your functions locally

Make sure that you test and verify your function code locally on the new target version.

Use these steps to update the project on your local computer:

1. Ensure you have [installed the target version of the .NET SDK](#).

If you are targeting a preview version, consult the [Functions guidance for preview .NET versions](#) to ensure that the version is supported. Additional steps may be required for .NET previews.

2. Update your references to the latest versions of:

[Microsoft.Azure.Functions.Worker](#) and [Microsoft.Azure.Functions.Worker.Sdk](#).

3. Update your project's target framework to the new version. For C# projects, you must update the `<TargetFramework>` element in the `.csproj` file. See [Target frameworks](#) for specifics related to the chosen version.

Changing your project's target framework might also require changes to parts of your toolchain, outside of project code. For example, in VS Code, you might need to update the `azureFunctions.deploySubpath` extension setting through user settings or your project's `.vscode/settings.json` file. Check for any dependencies on the framework version that may exist outside of your project code, as part of build steps or a CI/CD pipeline.

4. Make any updates to your project code that are required by the new .NET version. Check the version's release notes for specifics. You can also use the [.NET Upgrade Assistant](#) to help you update your code in response to changes across major versions.

After you've made those changes, rebuild your project and test it to confirm your app runs as expected.

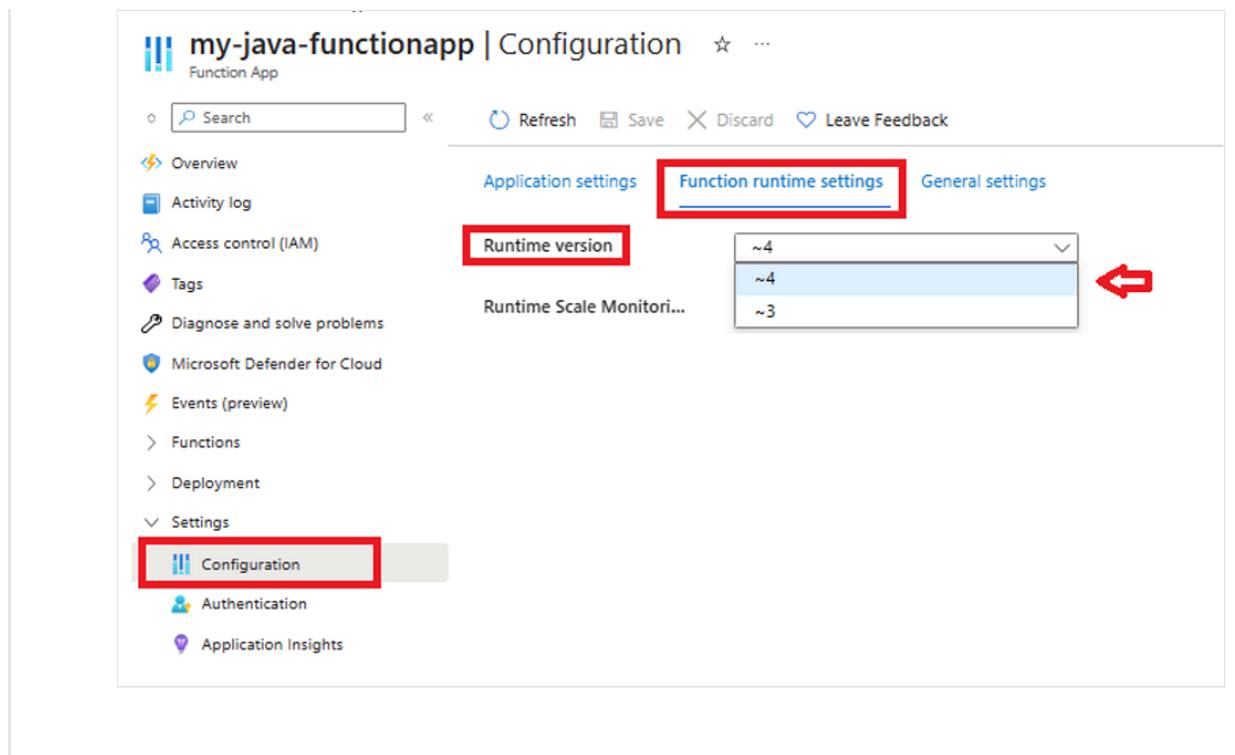
2. Move to the latest Functions runtime

Make sure your function app is running on the latest version of the Functions runtime (version 4.x). You can determine the runtime version either in the Azure portal or by using the Azure CLI.

Azure portal

Use these steps to determine your Functions runtime version:

1. In the [Azure portal](#), locate your function app and select **Configuration** on the left-hand side under **Settings**.
2. Select the **Function runtime settings** tab and check the **Runtime version** value to see if your function app is running on version 4.x of the Functions runtime (~4).



If you need to first update your function app to version 4.x, see [Migrate apps from Azure Functions version 1.x to version 4.x](#) or [Migrate apps from Azure Functions version 3.x to version 4.x](#). You should follow the instructions in those articles rather than just changing the `FUNCTIONS_EXTENSION_VERSION` setting.

Publish app updates

If you updated your app to run correctly on the new version, publish the app updates before you update the stack configuration for your function app.

Tip

To simplify the update process, minimize downtime for your functions, and provide a potential for rollback, you should publish your updated app to a staging slot. For more information, see [Azure Functions deployment slots](#).

When publishing your updated app to a staging slot, make sure to follow the slot-specific update instructions in the rest of this article. You later swap the updated staging slot into production.

Update the stack configuration

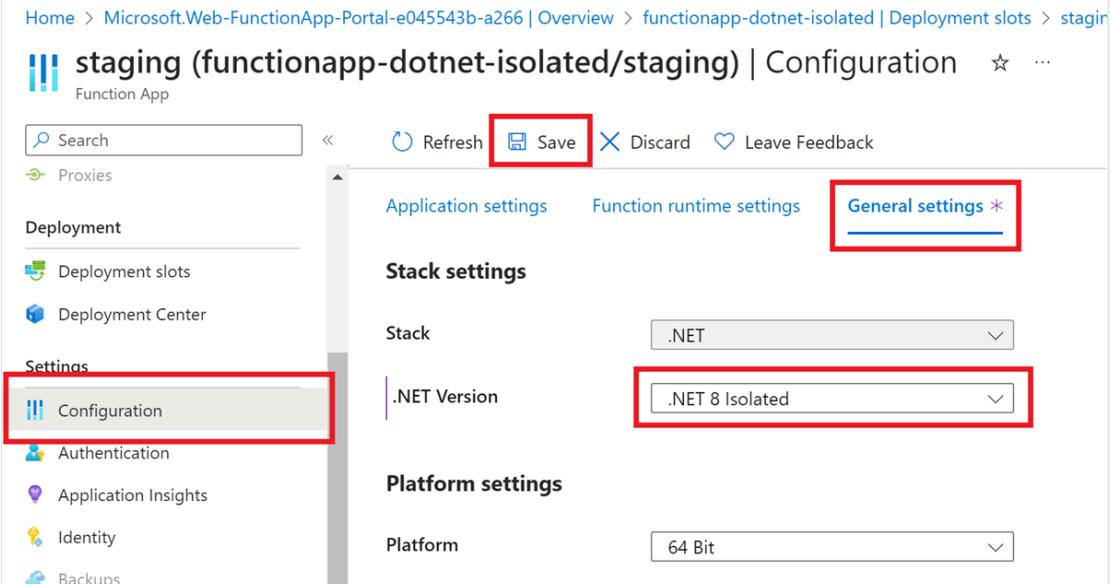
The way that you update the stack configuration depends on whether you're running on Windows or on Linux in Azure.

When using a [staging slot](#), make sure to target your updates to the correct slot.

Windows

Use the following steps to update the .NET version:

1. In the [Azure portal](#), locate your function app and select **Configuration** on the left-hand side. When using a staging slot, make sure to first select the specific slot.
2. In the **General settings** tab, update **.NET version** to the desired version.



3. Select **Save** and when notified about a restart select **Continue**.

Your function app restarts after you update the version.

Swap slots

If you have been performing your code project deployment and updating settings in a staging slot, you finally need to swap the staging slot into production. For more information, see [Swap slots](#).

Next steps

[C# isolated worker process guide](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Monitor Azure Functions

Article • 03/11/2024

This article describes:

- The types of monitoring data you can collect for this service.
- How to analyze that data.

Note

If you're already familiar with this service and/or Azure Monitor and just want to know how to analyze monitoring data, see the [Analyze](#) section near the end of this article.

When you have critical applications and business processes that rely on Azure resources, you need to monitor and get alerts for your system. The Azure Monitor service collects and aggregates metrics and logs from every component of your system. Azure Monitor provides you with a view of availability, performance, and resilience, and notifies you of issues. You can use the Azure portal, PowerShell, Azure CLI, REST API, or client libraries to set up and view monitoring data.

- For more information on Azure Monitor, see the [Azure Monitor overview](#).
- For more information on how to monitor Azure resources in general, see [Monitor Azure resources with Azure Monitor](#).

Insights

Some services in Azure have a built-in monitoring dashboard in the Azure portal that provides a starting point for monitoring your service. These dashboards are called *insights*, and you can find them in the **Insights Hub** of Azure Monitor in the Azure portal.

Application Insights

Azure Functions offers built-in integration with Application Insights to monitor functions executions. For detailed information about how to integrate, configure, and use Application Insights to monitor Azure Functions, see the following articles:

- [Monitor executions in Azure Functions](#)
- [How to configure monitoring for Azure Functions](#)

- [Analyze Azure Functions telemetry in Application Insights.](#)
- [Monitor Azure Functions with Application Insights](#)

Resource types

Azure uses the concept of resource types and IDs to identify everything in a subscription. Azure Monitor similarly organizes core monitoring data into metrics and logs based on resource types, also called *namespaces*. Different metrics and logs are available for different resource types. Your service might be associated with more than one resource type.

Resource types are also part of the resource IDs for every resource running in Azure. For example, one resource type for a virtual machine is `Microsoft.Compute/virtualMachines`. For a list of services and their associated resource types, see [Resource providers](#).

For more information about the resource types for Azure Functions, see [Azure Functions monitoring data reference](#).

Data storage

For Azure Monitor:

- Metrics data is stored in the Azure Monitor metrics database.
- Log data is stored in the Azure Monitor logs store. Log Analytics is a tool in the Azure portal that can query this store.
- The Azure activity log is a separate store with its own interface in the Azure portal.
- You can optionally route metric and activity log data to the Azure Monitor logs database store so you can query the data and correlate it with other log data using Log Analytics.

For detailed information on how Azure Monitor stores data, see [Azure Monitor data platform](#).

Azure Monitor platform metrics

Azure Monitor provides platform metrics for most services. These metrics are:

- Individually defined for each namespace.
- Stored in the Azure Monitor time-series metrics database.
- Lightweight and capable of supporting near real-time alerting.
- Used to track the performance of a resource over time.

Collection: Azure Monitor collects platform metrics automatically. No configuration is required.

Routing: You can also usually route platform metrics to Azure Monitor logs / Log Analytics so you can query them with other log data. For more information, see the [Metrics diagnostic setting](#). For how to configure diagnostic settings for a service, see [Create diagnostic settings in Azure Monitor](#).

For a list of all metrics it's possible to gather for all resources in Azure Monitor, see [Supported metrics in Azure Monitor](#).

For a list of available metrics for Azure Functions, see [Azure Functions monitoring data reference](#).

Azure Monitor resource logs

Resource logs provide insight into operations that were done by an Azure resource. Logs are generated automatically, but you must route them to Azure Monitor logs to save or query them. Logs are organized by category. A given namespace might have multiple resource log categories.

Collection: Resource logs aren't collected and stored until you create a *diagnostic setting* and route the logs to one or more locations. When you create a diagnostic setting, you specify which categories of logs to collect. There are multiple ways to create and maintain diagnostic settings, including the Azure portal, programmatically, and through Azure Policy.

Routing: The suggested default is to route resource logs to Azure Monitor Logs so you can query them with other log data. Other locations such as Azure Storage, Azure Event Hubs, and certain Microsoft monitoring partners are also available. For more information, see [Azure resource logs](#) and [Resource log destinations](#).

For detailed information about collecting, storing, and routing resource logs, see [Diagnostic settings in Azure Monitor](#).

For a list of all available resource log categories in Azure Monitor, see [Supported resource logs in Azure Monitor](#).

All resource logs in Azure Monitor have the same header fields, followed by service-specific fields. The common schema is outlined in [Azure Monitor resource log schema](#).

Azure Functions integrates with Azure Monitor Logs to monitor functions. For detailed instructions on how to set up diagnostic settings to configure and route resource logs, see [Create diagnostic settings in Azure Monitor](#).

Diagnostics settings

Save Discard Delete

A diagnostic setting specifies a list of categories of platform logs and/or metrics that you want to collect from a resource, and one or more destinations that you would stream them to. Normal usage charges for the destination will occur. [Learn more about the different log categories and contents of those logs](#)

Diagnostic settings name *

Category details

log

FunctionAppLogs

metric

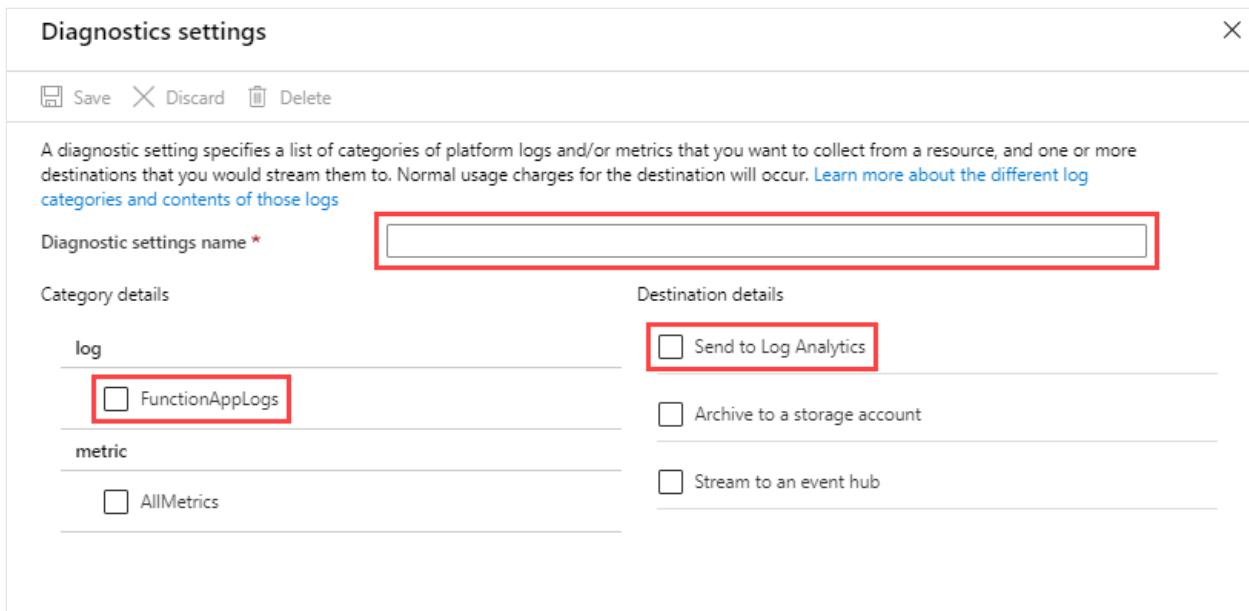
AllMetrics

Destination details

Send to Log Analytics

Archive to a storage account

Stream to an event hub



For the available resource log categories, their associated Log Analytics tables, and the logs schemas for Azure Functions, see [Azure Functions monitoring data reference](#).

Azure activity log

The activity log contains subscription-level events that track operations for each Azure resource as seen from outside that resource; for example, creating a new resource or starting a virtual machine.

Collection: Activity log events are automatically generated and collected in a separate store for viewing in the Azure portal.

Routing: You can send activity log data to Azure Monitor Logs so you can analyze it alongside other log data. Other locations such as Azure Storage, Azure Event Hubs, and certain Microsoft monitoring partners are also available. For more information on how to route the activity log, see [Overview of the Azure activity log](#).

Other logs

Azure Functions also offers the ability to collect more than Azure Monitor resource logs. To view a near real time stream of application log files generated by your function running in Azure, you can connect to Application Insights and use Live Metrics Stream. Or, you can use the App Service platform built-in log streaming to view a stream of application log files. For more information, see [Enable streaming execution logs in Azure Functions](#).

Analyze monitoring data

There are many tools for analyzing monitoring data.

Azure Monitor tools

Azure Monitor supports the following basic tools:

- [Metrics explorer](#), a tool in the Azure portal that allows you to view and analyze metrics for Azure resources. For more information, see [Analyze metrics with Azure Monitor metrics explorer](#).
- [Log Analytics](#), a tool in the Azure portal that allows you to query and analyze log data by using the [Kusto query language \(KQL\)](#). For more information, see [Get started with log queries in Azure Monitor](#).
- The [activity log](#), which has a user interface in the Azure portal for viewing and basic searches. To do more in-depth analysis, you have to route the data to Azure Monitor logs and run more complex queries in Log Analytics.

Tools that allow more complex visualization include:

- [Dashboards](#) that let you combine different kinds of data into a single pane in the Azure portal.
- [Workbooks](#), customizable reports that you can create in the Azure portal. Workbooks can include text, metrics, and log queries.
- [Grafana](#), an open platform tool that excels in operational dashboards. You can use Grafana to create dashboards that include data from multiple sources other than Azure Monitor.
- [Power BI](#), a business analytics service that provides interactive visualizations across various data sources. You can configure Power BI to automatically import log data from Azure Monitor to take advantage of these visualizations.

Azure Monitor export tools

You can get data out of Azure Monitor into other tools by using the following methods:

- **Metrics:** Use the [REST API for metrics](#) to extract metric data from the Azure Monitor metrics database. The API supports filter expressions to refine the data retrieved. For more information, see [Azure Monitor REST API reference](#).
- **Logs:** Use the REST API or the [associated client libraries](#).
- Another option is the [workspace data export](#).

To get started with the REST API for Azure Monitor, see [Azure monitoring REST API walkthrough](#).

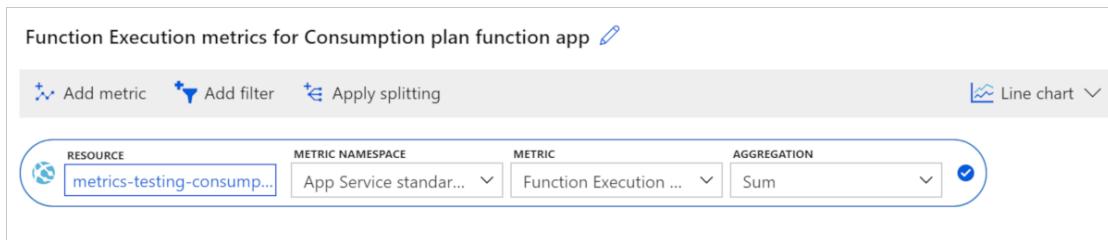
Analyze metrics for Azure Functions

The following examples use Azure Monitor metrics to help estimate the cost of running your function app on a Consumption plan. To learn more about estimating Consumption plan costs, see [Estimating Consumption plan costs](#).

Portal

Use [Azure Monitor metrics explorer](#) to view cost-related data for your Consumption plan function apps in a graphical format.

1. In the [Azure portal](#), navigate to your function app.
2. In the left panel, scroll down to **Monitoring** and choose **Metrics**.
3. From **Metric**, choose **Function Execution Count** and **Sum** for **Aggregation**.
This adds the sum of the execution counts during chosen period to the chart.



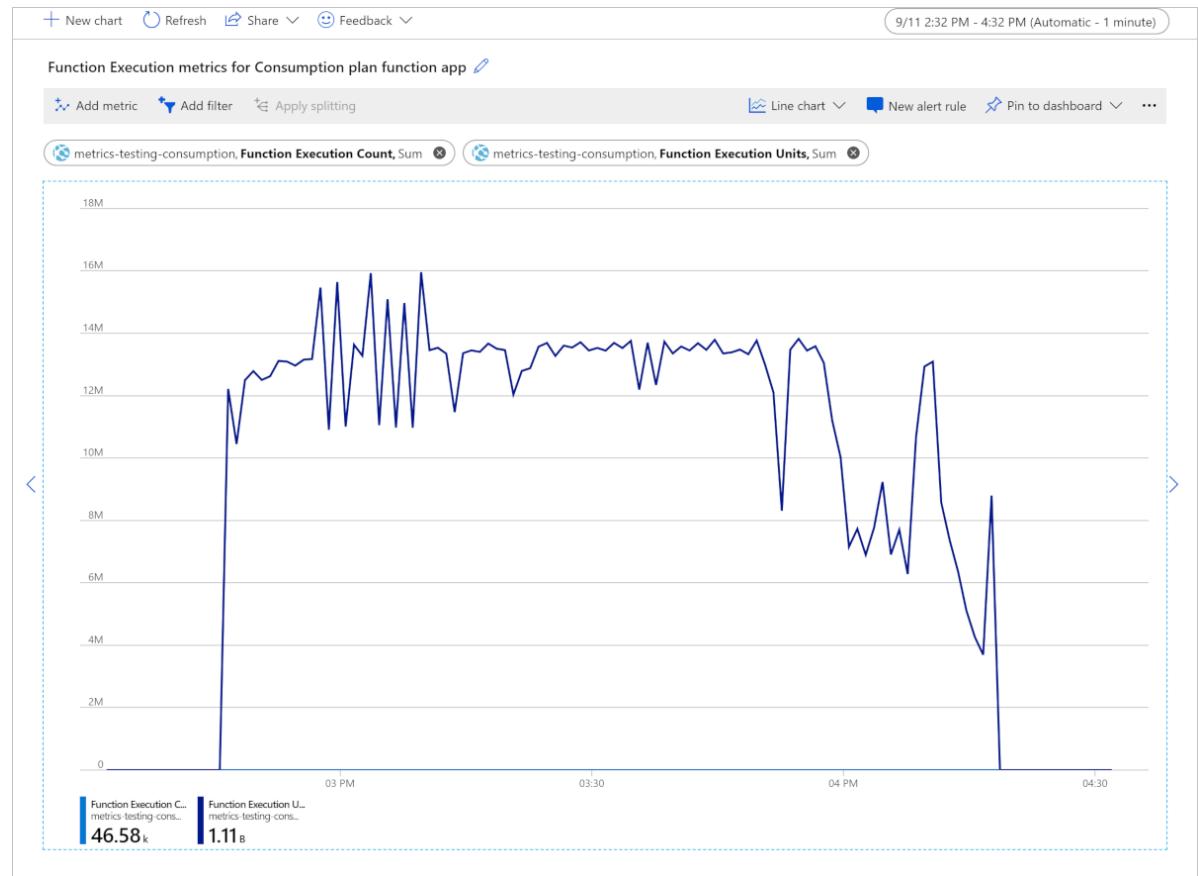
Function Execution metrics for Consumption plan function app

Add metric Add filter Apply splitting Line chart

RESOURCE metrics-testing-consum...	METRIC NAMESPACE App Service standar...	METRIC Function Execution ...	AGGREGATION Sum
---------------------------------------	--	----------------------------------	--------------------

4. Select **Add metric** and repeat steps 2-4 to add **Function Execution Units** to the chart.

The resulting chart contains the totals for both execution metrics in the chosen time range, which in this case is two hours.



As the number of execution units is so much greater than the execution count, the chart just shows execution units.

This chart shows a total of 1.11 billion Function Execution Units consumed in a two-hour period, measured in MB-milliseconds. To convert to GB-seconds, divide by 1024000. In this example, the function app consumed $1110000000 / 1024000 = 1083.98$ GB-seconds. You can take this value and multiply by the current price of execution time on the [Functions pricing page](#), which gives you the cost of these two hours, assuming you've already used any free grants of execution time.

Analyze logs for Azure Functions

Azure Functions writes all logs to the **FunctionAppLogs** table under **LogManagement** in the Log Analytics workspace where you send the data. You can use Kusto queries to query the data.

The screenshot shows the Azure Log Analytics workspace interface. On the left, there's a navigation menu with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Locks, Export template, Network isolation, Advanced settings), General (Quick Start, Workspace summary, View Designer, Workbooks, Logs, Solutions). The 'Logs' item under General is highlighted with a red box. The main area has tabs for Tables, Queries, and Filter. A search bar at the top says 'Search (Ctrl+ /)'. Below it, a 'New Query 1' button and a 'Run' button are visible. The 'Time range : Last 24 hours' is set. There's a 'Select Scope' dropdown showing 'DefaultWorkspace-316e8102-0662-41cb-b95e-2e2dbdabf52b-EUS | Logs'. A search bar in the center says 'Type your query here or click one of the example queries to start'. To the right, there's a 'Queries History' section with a 'Clear history' link and a 'Learn more' link. Below that is a 'Start Querying' button with a grid icon. A note says 'You have no recent queries. Start querying using example queries or type your own queries in the query editor.' On the far right, there are links for Getting started, Online course, Language reference, Community, and What's new.

Kusto queries

You can analyze monitoring data in the Azure Monitor Logs / Log Analytics store by using the Kusto query language (KQL).

Important

When you select **Logs** from the service's menu in the portal, Log Analytics opens with the query scope set to the current service. This scope means that log queries will only include data from that type of resource. If you want to run a query that includes data from other Azure services, select **Logs** from the **Azure Monitor** menu. See [Log query scope and time range in Azure Monitor Log Analytics](#) for details.

For a list of common queries for any service, see the [Log Analytics queries interface](#).

The following sample queries can help you monitor all your functions app logs:

```
Kusto

FunctionAppLogs
| order by TimeGenerated desc
```

```
Kusto

FunctionAppLogs
| project TimeGenerated, HostInstanceId, Message, _ResourceId
| order by TimeGenerated desc
```

The following sample query can help you monitor a specific functions app's logs:

Kusto

```
FunctionAppLogs  
| where FunctionName == "<Function name>"  
| order by TimeGenerated desc
```

The following sample query can help you monitor exceptions on all your functions app logs:

Kusto

```
FunctionAppLogs  
| where ExceptionDetails != ""  
| order by TimeGenerated asc
```

The following sample query can help you monitor exceptions on a specific functions app's logs:

Kusto

```
FunctionAppLogs  
| where ExceptionDetails != ""  
| where FunctionName == "<Function name>"  
| order by TimeGenerated desc
```

Alerts

Azure Monitor alerts proactively notify you when specific conditions are found in your monitoring data. Alerts allow you to identify and address issues in your system before your customers notice them. For more information, see [Azure Monitor alerts](#).

There are many sources of common alerts for Azure resources. For examples of common alerts for Azure resources, see [Sample log alert queries](#). The [Azure Monitor Baseline Alerts \(AMBA\)](#) site provides key alert metrics, dashboards, and guidelines for Azure Landing Zone (ALZ) scenarios.

The common alert schema standardizes the consumption of Azure Monitor alert notifications. For more information, see [Common alert schema](#).

Types of alerts

You can alert on any metric or log data source in the Azure Monitor data platform. There are many different types of alerts depending on the services you're monitoring and the monitoring data you're collecting. Different types of alerts have various benefits and drawbacks. For more information, see [Choose the right monitoring alert type](#).

The following list describes the types of Azure Monitor alerts you can create:

- [Metric alerts](#) evaluate resource metrics at regular intervals. Metrics can be platform metrics, custom metrics, logs from Azure Monitor converted to metrics, or Application Insights metrics. Metric alerts can also apply multiple conditions and dynamic thresholds.
- [Log alerts](#) allow users to use a Log Analytics query to evaluate resource logs at a predefined frequency.
- [Activity log alerts](#) trigger when a new activity log event occurs that matches defined conditions. Resource Health alerts and Service Health alerts are activity log alerts that report on your service and resource health.

Some Azure services also support [smart detection alerts](#), [Prometheus alerts](#), or [recommended alert rules](#).

For some services, you can monitor at scale by applying the same metric alert rule to multiple resources of the same type that exist in the same Azure region. Individual notifications are sent for each monitored resource. For supported Azure services and clouds, see [Monitor multiple resources with one alert rule](#).

 **Note**

If you're creating or running an application that runs on your service, [Azure Monitor application insights](#) might offer more types of alerts.

Azure Functions alert rules

The following table lists common and recommended alert rules for Azure Functions. These are just recommended alerts. You can set alerts for any metric, log entry, or activity log entry listed in the [Monitoring data reference for Azure Functions](#).

 [Expand table](#)

Alert type	Condition	Description
Metric	Average connections	When number of connections exceed a set value

Alert type	Condition	Description
Metric	HTTP 404	When HTTP 404 responses exceed a set value
Metric	HTTP Server Errors	When HTTP 5xx errors exceed a set value
Activity Log	Create or update function app	When app is created or updated
Activity Log	Delete function app	When app is deleted
Activity Log	Restart function app	When app is restarted
Activity Log	Stop function app	When app is stopped

Advisor recommendations

For some services, if critical conditions or imminent changes occur during resource operations, an alert displays on the service [Overview](#) page in the portal. You can find more information and recommended fixes for the alert in **Advisor recommendations** under **Monitoring** in the left menu. During normal operations, no advisor recommendations display.

For more information on Azure Advisor, see [Azure Advisor overview](#).

Related content

For more information about monitoring Azure Functions, see the following articles:

- [Azure Functions monitoring data reference](#) provides a reference of the metrics, logs, and other important values available for your function app.
- [Monitor Azure resources with Azure Monitor](#) gives general details about monitoring Azure resources.
- [Monitor executions in Azure Functions](#) details how to monitor a function app.
- [How to configure monitoring for Azure Functions](#) describes how to configure monitoring.
- [Analyze Azure Functions telemetry in Application Insights](#) describes how to view and query the data being collected from a function app.

Monitor executions in Azure Functions

Article • 03/28/2023

Azure Functions offers built-in integration with [Azure Application Insights](#) to monitor functions executions. This article provides an overview of the monitoring capabilities provided by Azure for monitoring Azure Functions.

Application Insights collects log, performance, and error data. By automatically detecting performance anomalies and featuring powerful analytics tools, you can more easily diagnose issues and better understand how your functions are used. These tools are designed to help you continuously improve performance and usability of your functions. You can even use Application Insights during local function app project development. For more information, see [What is Application Insights?](#).

As Application Insights instrumentation is built into Azure Functions, you need a valid instrumentation key to connect your function app to an Application Insights resource. The instrumentation key is added to your application settings as you create your function app resource in Azure. If your function app doesn't already have this key, you can [set it manually](#).

You can also monitor the function app itself by using Azure Monitor. To learn more, see [Monitoring Azure Functions with Azure Monitor](#).

Application Insights pricing and limits

You can try out Application Insights integration with Azure Functions for free featuring a daily limit to how much data is processed for free.

If you enable Applications Insights during development, you might hit this limit during testing. Azure provides portal and email notifications when you're approaching your daily limit. If you miss those alerts and hit the limit, new logs won't appear in Application Insights queries. Be aware of the limit to avoid unnecessary troubleshooting time. For more information, see [Application Insights billing](#).

Important

Application Insights has a [sampling](#) feature that can protect you from producing too much telemetry data on completed executions at times of peak load. Sampling is enabled by default. If you appear to be missing data, you might need to adjust the sampling settings to fit your particular monitoring scenario. To learn more, see [Configure sampling](#).

The full list of Application Insights features available to your function app is detailed in [Application Insights for Azure Functions supported features](#).

Application Insights integration

Typically, you create an Application Insights instance when you create your function app. In this case, the instrumentation key required for the integration is already set as an application setting named `APPINSIGHTS_INSTRUMENTATIONKEY`. If for some reason your function app doesn't have the instrumentation key set, you need to [enable Application Insights integration](#).

ⓘ Important

Sovereign clouds, such as Azure Government, require the use of the Application Insights connection string (`APPLICATIONINSIGHTS_CONNECTION_STRING`) instead of the instrumentation key. To learn more, see the [APPLICATIONINSIGHTS_CONNECTION_STRING reference](#).

The following table details the supported features of Application Insights available for monitoring your function apps:

Azure Functions runtime version	1.x	2.x+
Automatic collection of		
• Requests	✓	✓
• Exceptions	✓	✓
• Performance Counters	✓	✓
• Dependencies		
— HTTP	✓	
— Service Bus		✓
— Event Hubs		✓
— SQL*		✓
Supported features		

Azure Functions runtime version	1.x	2.x+
• QuickPulse/LiveMetrics	Yes	Yes
— Secure Control Channel		Yes
• Sampling	Yes	Yes
• Heartbeats		Yes
Correlation		
• Service Bus		Yes
• Event Hubs		Yes
Configurable		
• Fully configurable		Yes

* To enable the collection of SQL query string text, see [Enable SQL query collection](#).

Collecting telemetry data

With Application Insights integration enabled, telemetry data is sent to your connected Application Insights instance. This data includes logs generated by the Functions host, traces written from your functions code, and performance data.

ⓘ Note

In addition to data from your functions and the Functions host, you can also collect data from the [Functions scale controller](#).

Log levels and categories

When you write traces from your application code, you should assign a log level to the traces. Log levels provide a way for you to limit the amount of data that is collected from your traces.

A *log level* is assigned to every log. The value is an integer that indicates relative importance:

LogLevel	Code	Description
----------	------	-------------

LogLevel	Code	Description
Trace	0	Logs that contain the most detailed messages. These messages might contain sensitive application data. These messages are disabled by default and should never be enabled in a production environment.
Debug	1	Logs that are used for interactive investigation during development. These logs should primarily contain information useful for debugging and have no long-term value.
Information	2	Logs that track the general flow of the application. These logs should have long-term value.
Warning	3	Logs that highlight an abnormal or unexpected event in the application flow, but don't otherwise cause the application execution to stop.
Error	4	Logs that highlight when the current flow of execution is stopped because of a failure. These errors should indicate a failure in the current activity, not an application-wide failure.
Critical	5	Logs that describe an unrecoverable application or system crash, or a catastrophic failure that requires immediate attention.
None	6	Disables logging for the specified category.

The [host.json file](#) configuration determines how much logging a functions app sends to Application Insights.

To learn more about log levels, see [Configure log levels](#).

By assigning logged items to a category, you have more control over telemetry generated from specific sources in your function app. Categories make it easier to run analytics over collected data. Traces written from your function code are assigned to individual categories based on the function name. To learn more about categories, see [Configure categories](#).

Custom telemetry data

In [C#](#), [JavaScript](#), and [Python](#), you can use an Application Insights SDK to write custom telemetry data.

Dependencies

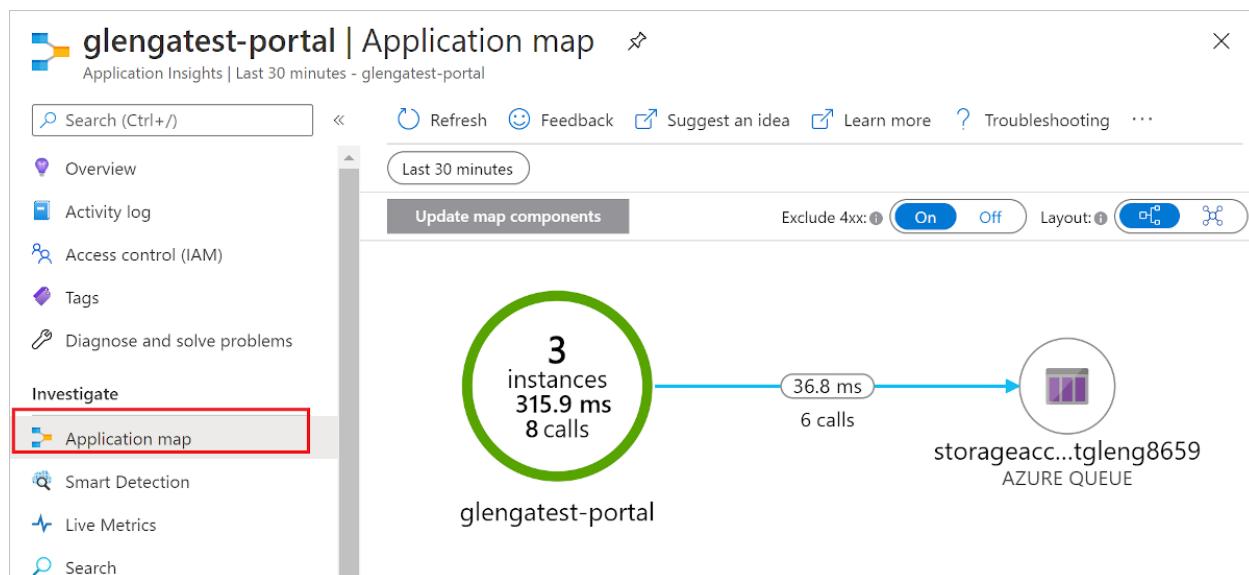
Starting with version 2.x of Functions, Application Insights automatically collects data on dependencies for bindings that use certain client SDKs. Application Insights distributed tracing and dependency tracking aren't currently supported for C# apps running in an

[isolated worker process](#). Application Insights collects data on the following dependencies:

- Azure Cosmos DB
- Azure Event Hubs
- Azure Service Bus
- Azure Storage services (Blob, Queue, and Table)

HTTP requests and database calls using `SqlClient` are also captured. For the complete list of dependencies supported by Application Insights, see [automatically tracked dependencies](#).

Application Insights generates an *application map* of collected dependency data. The following is an example of an application map of an HTTP trigger function with a Queue storage output binding.



Dependencies are written at the `Information` level. If you filter at `Warning` or above, you won't see the dependency data. Also, automatic collection of dependencies happens at a non-user scope. To capture dependency data, make sure the level is set to at least `Information` outside the user scope (`Function.<YOUR_FUNCTION_NAME>.User`) in your host.

In addition to automatic dependency data collection, you can also use one of the language-specific Application Insights SDKs to write custom dependency information to the logs. For an example how to write custom dependencies, see one of the following language-specific examples:

- [Log custom telemetry in C# functions](#)
- [Log custom telemetry in JavaScript functions](#)
- [Log custom telemetry in Python functions](#)

Performance Counters

Automatic collection of Performance Counters isn't supported when running on Linux.

Writing to logs

The way that you write to logs and the APIs you use depend on the language of your function app project.

See the developer guide for your language to learn more about writing logs from your functions.

- [C# \(.NET class library\)](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

Analyze data

By default, the data collected from your function app is stored in Application Insights. In the [Azure portal](#), Application Insights provides an extensive set of visualizations of your telemetry data. You can drill into error logs and query events and metrics. To learn more, including basic examples of how to view and query your collected data, see [Analyze Azure Functions telemetry in Application Insights](#).

Streaming Logs

While developing an application, you often want to see what's being written to the logs in near real time when running in Azure.

There are two ways to view a stream of the log data being generated by your function executions.

- **Built-in log streaming:** the App Service platform lets you view a stream of your application log files. This stream is equivalent to the output seen when you debug your functions during [local development](#) and when you use the **Test** tab in the portal. All log-based information is displayed. For more information, see [Stream logs](#). This streaming method supports only a single instance, and can't be used with an app running on Linux in a Consumption plan.

- **Live Metrics Stream:** when your function app is [connected to Application Insights](#), you can view log data and other metrics in near real time in the Azure portal using [Live Metrics Stream](#). Use this method when monitoring functions running on multiple-instances or on Linux in a Consumption plan. This method uses [sampled data](#).

Log streams can be viewed both in the portal and in most local development environments. To learn how to enable log streams, see [Enable streaming execution logs in Azure Functions](#).

Diagnostic logs

Application Insights lets you export telemetry data to long-term storage or other analysis services.

Because Functions also integrates with Azure Monitor, you can also use diagnostic settings to send telemetry data to various destinations, including Azure Monitor logs. To learn more, see [Monitoring Azure Functions with Azure Monitor Logs](#).

Scale controller logs

The [Azure Functions scale controller](#) monitors instances of the Azure Functions host on which your app runs. This controller makes decisions about when to add or remove instances based on current performance. You can have the scale controller emit logs to Application Insights to better understand the decisions the scale controller is making for your function app. You can also store the generated logs in Blob storage for analysis by another service.

To enable this feature, you add an application setting named `SCALE_CONTROLLER_LOGGING_ENABLED` to your function app settings. To learn how, see [Configure scale controller logs](#).

Azure Monitor metrics

In addition to log-based telemetry data collected by Application Insights, you can also get data about how the function app is running from [Azure Monitor Metrics](#). To learn more, see [Monitoring with Azure Monitor](#).

Report issues

To report an issue with Application Insights integration in Functions, or to make a suggestion or request, [create an issue in GitHub](#).

Next steps

For more information, see the following resources:

- [Application Insights](#)
- [ASP.NET Core logging](#)

How to configure monitoring for Azure Functions

Article • 08/06/2024

Azure Functions integrates with Application Insights to better enable you to monitor your function apps. Application Insights, a feature of Azure Monitor, is an extensible Application Performance Management (APM) service that collects data generated by your function app, including information your app writes to logs. Application Insights integration is typically enabled when your function app is created. If your app doesn't have the instrumentation key set, you must first [enable Application Insights integration](#).

You can use Application Insights without any custom configuration. However, the default configuration can result in high volumes of data. If you're using a Visual Studio Azure subscription, you might hit your data cap for Application Insights. For information about Application Insights costs, see [Application Insights billing](#). For more information, see [Solutions with high-volume of telemetry](#).

In this article, you learn how to configure and customize the data that your functions send to Application Insights. You can set common logging configurations in the `host.json` file. By default, these settings also govern custom logs emitted by your code. However, in some cases this behavior can be disabled in favor of options that give you more control over logging. For more information, see [Custom application logs](#).

ⓘ Note

You can use specially configured application settings to represent specific settings in a `host.json` file for a particular environment. Doing so lets you effectively change `host.json` settings without needing to republish the `host.json` file in your project. For more information, see [Override host.json values](#).

Custom application logs

By default, custom application logs you write are sent to the Functions host, which then sends them to Application Insights under the [Worker category](#). Some language stacks allow you to instead send the logs directly to Application Insights, which gives you full control over how logs you write are emitted. In this case, the logging pipeline changes from `worker -> Functions host -> Application Insights` to `worker -> Application Insights`.

The following table summarizes the configuration options available for each stack:

ⓘ [Expand table](#)

Language stack	Where to configure custom logs
.NET (in-process model)	<code>host.json</code>
.NET (isolated model)	Default (send custom logs to the Functions host): <code>host.json</code> To send logs directly to Application Insights, see: Configure Application Insights in the HostBuilder
Node.js	<code>host.json</code>

Language stack	Where to configure custom logs
Python	<code>host.json</code>
Java	Default (send custom logs to the Functions host): <code>host.json</code> To send logs directly to Application Insights, see: Configure the Application Insights Java agent
PowerShell	<code>host.json</code>

When you configure custom application logs to be sent directly, the host no longer emits them, and `host.json` no longer controls their behavior. Similarly, the options exposed by each stack apply only to custom logs, and they don't change the behavior of the other runtime logs described in this article. In this case, to control the behavior of all logs, you might need to make changes in both configurations.

Configure categories

The Azure Functions logger includes a *category* for every log. The category indicates which part of the runtime code or your function code wrote the log. Categories differ between version 1.x and later versions.

Category names are assigned differently in Functions compared to other .NET frameworks. For example, when you use `ILogger<T>` in ASP.NET, the category is the name of the generic type. C# functions also use `ILogger<T>`, but instead of setting the generic type name as a category, the runtime assigns categories based on the source. For example:

- Entries related to running a function are assigned a category of `Function.<FUNCTION_NAME>`.
- Entries created by user code inside the function, such as when calling `logger.LogInformation()`, are assigned a category of `Function.<FUNCTION_NAME>.User`.

The following table describes the main categories of logs that the runtime creates:

v2.x+

[Expand table](#)

Category	Table	Description
<code>Function</code>	<code>traces</code>	Includes function started and completed logs for all function runs. For successful runs, these logs are at the <code>Information</code> level. Exceptions are logged at the <code>Error</code> level. The runtime also creates <code>Warning</code> level logs, such as when queue messages are sent to the <code>poison queue</code> .
<code>Function.<YOUR_FUNCTION_NAME></code>	<code>dependencies</code>	Dependency data is automatically collected for some services. For successful runs, these logs are at the <code>Information</code> level. For more information, see Dependencies . Exceptions are logged at the <code>Error</code> level. The runtime also creates <code>Warning</code> level logs, such as when queue messages are sent to the <code>poison queue</code> .
<code>Function.<YOUR_FUNCTION_NAME></code>	<code>customMetrics</code> <code>customEvents</code>	C# and JavaScript SDKs lets you collect custom metrics and log custom events. For more information, see Custom telemetry data .
<code>Function.<YOUR_FUNCTION_NAME></code>	<code>traces</code>	Includes function started and completed logs for specific function runs. For successful runs, these logs are at the <code>Information</code> level. Exceptions are logged at the <code>Error</code> level.

Category	Table	Description
		at the <code>Error</code> level. The runtime also creates <code>Warning</code> level logs, such as when queue messages are sent to the <code>poison queue</code> .
<code>Function.<YOUR_FUNCTION_NAME>.User</code>	<code>traces</code>	User-generated logs, which can be any log level. For more information about writing to logs from your functions, see Writing to logs .
<code>Host.Aggregator</code>	<code>customMetrics</code>	These runtime-generated logs provide counts and averages of function invocations over a configurable period of time. The default period is 30 seconds or 1,000 results, whichever comes first. Examples are the number of runs, success rate, and duration. All of these logs are written at the <code>Information</code> level. If you filter at <code>Warning</code> or higher, you don't see any of this data.
<code>Host.Results</code>	<code>requests</code>	These runtime-generated logs indicate success or failure of a function. All of these logs are written at the <code>Information</code> level. If you filter at <code>Warning</code> or higher, you don't see any of this data.
<code>Microsoft</code>	<code>traces</code>	Fully qualified log category that reflects a .NET runtime component invoked by the host.
<code>Worker</code>	<code>traces</code>	Logs generated by the language worker process for non-.NET languages. Language worker logs might also be logged in a <code>Microsoft.*</code> category, such as <code>Microsoft.Azure.WebJobs.Script.Workers.Rpc.RpcFunctionInvocationDispatcher</code> . These logs are written at the <code>Information</code> level.

 **Note**

For .NET class library functions, these categories assume you're using `ILogger` and not `ILogger<T>`.

For more information, see the [Functions ILogger documentation](#).

The **Table** column indicates to which table in Application Insights the log is written.

Configure log levels

A *log level* is assigned to every log. The value is an integer that indicates relative importance:

 [Expand table](#)

LogLevel	Code	Description
Trace	0	Logs that contain the most detailed messages. These messages might contain sensitive application data. These messages are disabled by default and should never be enabled in a production environment.
Debug	1	Logs that are used for interactive investigation during development. These logs should primarily contain information useful for debugging and have no long-term value.
Information	2	Logs that track the general flow of the application. These logs should have long-term value.
Warning	3	Logs that highlight an abnormal or unexpected event in the application flow, but don't otherwise cause the application execution to stop.
Error	4	Logs that highlight when the current flow of execution is stopped because of a failure. These errors should indicate a failure in the current activity, not an application-wide failure.

LogLevel	Code	Description
Critical	5	Logs that describe an unrecoverable application or system crash, or a catastrophic failure that requires immediate attention.
None	6	Disables logging for the specified category.

The [host.json file](#) configuration determines how much logging a functions app sends to Application Insights.

For each category, you indicate the minimum log level to send. The [host.json](#) settings vary depending on the [Functions runtime version](#).

The following examples define logging based on the following rules:

- The default logging level is set to `Warning` to prevent [excessive logging](#) for unanticipated categories.
- `Host.Aggregator` and `Host.Results` are set to lower levels. Setting logging levels too high (especially higher than `Information`) can result in loss of metrics and performance data.
- Logging for function runs is set to `Information`. If necessary, you can [override](#) this setting in local development to `Debug` or `Trace`.

v2.x+

JSON

```
{
  "logging": {
    "fileLoggingMode": "debugOnly",
    "logLevel": {
      "default": "Warning",
      "Host.Aggregator": "Trace",
      "Host.Results": "Information",
      "Function": "Information"
    }
  }
}
```

If [host.json](#) includes multiple logs that start with the same string, the more defined logs ones are matched first. Consider the following example that logs everything in the runtime, except `Host.Aggregator`, at the `Error` level:

v2.x+

JSON

```
{
  "logging": {
    "fileLoggingMode": "debugOnly",
    "logLevel": {
      "default": "Information",
      "Host": "Error",
      "Function": "Error",
      "Host.Aggregator": "Information"
    }
  }
}
```

```
}
```

You can use a log level setting of `None` to prevent any logs from being written for a category.

⊗ Caution

Azure Functions integrates with Application Insights by storing telemetry events in Application Insights tables. If you set a category log level to any value different from `Information`, it prevents the telemetry from flowing to those tables, and you won't be able to see related data in the **Application Insights** and **Function Monitor** tabs.

For example, for the previous samples:

- If you set the `Host.Results` category to the `Error` log level, Azure gathers only host execution telemetry events in the `requests` table for failed function executions, preventing the display of host execution details of successful executions in both the **Application Insights** and **Function Monitor** tabs.
- If you set the `Function` category to the `Error` log level, it stops gathering function telemetry data related to `dependencies`, `customMetrics`, and `customEvents` for all the functions, preventing you from viewing any of this data in Application Insights. Azure gathers only `traces` logged at the `Error` level.

In both cases, Azure continues to collect errors and exceptions data in the **Application Insights** and **Function Monitor** tabs. For more information, see [Solutions with high-volume of telemetry](#).

Configure the aggregator

As noted in the previous section, the runtime aggregates data about function executions over a period of time. The default period is 30 seconds or 1,000 runs, whichever comes first. You can configure this setting in the `host.json` file. For example:

```
JSON
```

```
{
  "aggregator": {
    "batchSize": 1000,
    "flushTimeout": "00:00:30"
  }
}
```

Configure sampling

Application Insights has a [sampling](#) feature that can protect you from producing too much telemetry data on completed executions at times of peak load. When the rate of incoming executions exceeds a specified threshold, Application Insights starts to randomly ignore some of the incoming executions. The default setting for maximum number of executions per second is 20 (five in version 1.x). You can configure sampling in `host.json`. Here's an example:

v2.x+

JSON

```
{  
  "logging": {  
    "applicationInsights": {  
      "samplingSettings": {  
        "isEnabled": true,  
        "maxTelemetryItemsPerSecond" : 20,  
        "excludedTypes": "Request;Exception"  
      }  
    }  
  }  
}
```

You can exclude certain types of telemetry from sampling. In this example, data of type `Request` and `Exception` is excluded from sampling. It ensures that *all* function executions (requests) and exceptions are logged while other types of telemetry remain subject to sampling.

If your project uses a dependency on the Application Insights SDK to do manual telemetry tracking, you might experience unusual behavior if your sampling configuration differs from the sampling configuration in your function app. In such cases, use the same sampling configuration as the function app. For more information, see [Sampling in Application Insights](#).

Enable SQL query collection

Application Insights automatically collects data on dependencies for HTTP requests, database calls, and for several bindings. For more information, see [Dependencies](#). For SQL calls, the name of the server and database is always collected and stored, but SQL query text isn't collected by default. You can use `dependencyTrackingOptions.enableSqlCommandTextInstrumentation` to enable SQL query text logging by using the following settings (at a minimum) in your [host.json file](#):

JSON

```
"logging": {  
  "applicationInsights": {  
    "enableDependencyTracking": true,  
    "dependencyTrackingOptions": {  
      "enableSqlCommandTextInstrumentation": true  
    }  
  }  
}
```

For more information, see [Advanced SQL tracking to get full SQL query](#).

Configure scale controller logs

This feature is in preview.

You can have the [Azure Functions scale controller](#) emit logs to either Application Insights or to Blob storage to better understand the decisions the scale controller is making for your function app.

To enable this feature, add an application setting named `SCALE_CONTROLLER_LOGGING_ENABLED` to your function app settings. The following value of the setting must be in the format `<DESTINATION>:<VERBOSITY>`. For more information, see the following table:

[+] Expand table

Property	Description
<code><DESTINATION></code>	The destination to which logs are sent. Valid values are <code>AppInsights</code> and <code>Blob</code> . When you use <code>AppInsights</code> , ensure that the Application Insights is enabled in your function app . When you set the destination to <code>Blob</code> , logs are created in a blob container named <code>azure-functions-scale-controller</code> in the default storage account set in the <code>AzureWebJobsStorage</code> application setting.
<code><VERBOSITY></code>	Specifies the level of logging. Supported values are <code>None</code> , <code>Warning</code> , and <code>Verbose</code> . When set to <code>Verbose</code> , the scale controller logs a reason for every change in the worker count, and information about the triggers that factor into those decisions. Verbose logs include trigger warnings and the hashes used by the triggers before and after the scale controller runs.

💡 Tip

Keep in mind that while you leave scale controller logging enabled, it impacts the [potential costs of monitoring your function app](#). Consider enabling logging until you have collected enough data to understand how the scale controller is behaving, and then disabling it.

For example, the following Azure CLI command turns on verbose logging from the scale controller to Application Insights:

Azure CLI

```
az functionapp config appsettings set --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--settings SCALE_CONTROLLER_LOGGING_ENABLED=AppInsights:Verbose
```

In this example, replace `<FUNCTION_APP_NAME>` and `<RESOURCE_GROUP_NAME>` with the name of your function app and the resource group name, respectively.

The following Azure CLI command disables logging by setting the verbosity to `None`:

Azure CLI

```
az functionapp config appsettings set --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--settings SCALE_CONTROLLER_LOGGING_ENABLED=AppInsights:None
```

You can also disable logging by removing the `SCALE_CONTROLLER_LOGGING_ENABLED` setting using the following Azure CLI command:

Azure CLI

```
az functionapp config appsettings delete --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--setting-names SCALE_CONTROLLER_LOGGING_ENABLED
```

With scale controller logging enabled, you're now able to [query your scale controller logs](#).

Enable Application Insights integration

For a function app to send data to Application Insights, it needs to connect to the Application Insights resource using **only one** of these application settings:

[+] Expand table

Setting name	Description
<code>APPLICATIONINSIGHTS_CONNECTION_STRING</code>	This setting is recommended and is required when your Application Insights instance runs in a sovereign cloud. The connection string supports other new capabilities .
<code>APPINSIGHTS_INSTRUMENTATIONKEY</code>	Legacy setting, which Application Insights has deprecated in favor of the connection string setting.

When you create your function app in the [Azure portal](#) from the command line by using [Azure Functions Core Tools](#) or [Visual Studio Code](#), Application Insights integration is enabled by default. The Application Insights resource has the same name as your function app, and is created either in the same region or in the nearest region.

Require Microsoft Entra authentication

You can use the `APPLICATIONINSIGHTS_AUTHENTICATION_STRING` setting to enable connections to Application Insights using Microsoft Entra authentication. This creates a consistent authentication experience across all Application Insights pipelines, including Profiler and Snapshot Debugger, as well as from the Functions host and language-specific agents.

① Note

There's no Entra authentication support for local development.

The value contains either `Authorization=AAD` for a system-assigned managed identity or `clientId=<YOUR_CLIENT_ID>;Authorization=AAD` for a user-assigned managed identity. The managed identity must already be available to the function app, with an assigned role equivalent to [Monitoring Metrics Publisher](#). For more information, see [Microsoft Entra authentication for Application Insights](#).

The `APPLICATIONINSIGHTS_CONNECTION_STRING` setting is still required.

① Note

When using `APPLICATIONINSIGHTS_AUTHENTICATION_STRING` to connect to Application Insights using Microsoft Entra authentication, you should also [Disable local authentication for Application Insights](#). This configuration requires Microsoft Entra authentication in order for telemetry to be ingested into your workspace.

New function app in the portal

To review the Application Insights resource being created, select it to expand the **Application Insights** window. You can change the **New resource name** or select a different **Location** in an [Azure geography](#) where you want to store your data.

The screenshot shows the Azure portal interface for creating a new Function App. On the left, the 'Function App' configuration pane is open, showing fields for App name (functionapp0921), Subscription (Visual Studio Enterprise), Resource Group (Create new, functionapp0921), OS (Windows selected), Hosting Plan (Consumption Plan), Location (West Europe), Runtime Stack (.NET), Storage (Create new, functionapp09218f22), and Application Insights (selected). A red box highlights the 'Application Insights' section. At the bottom of this pane are 'Create' and 'Automation options' buttons, and an 'Apply' button in a red box. On the right, the 'functionapp0921 - Application Insights' blade is displayed, showing the Application Insights site extensions section with an 'Enable' button (also in a red box) and a 'Link to an Application Insights resource' section. This section includes a summary message about connecting to an auto-created Application Insights resource named 'functionapp0921'. Below this is a 'Change your resource' section with a 'Create new resource' form (New resource name: functionapp0921, Location: West Europe) and a 'Select existing resource' search bar. A table lists existing resources: functions-ggailey777-vs (Resource Group: functions-ggailey777-vs, Location: West Europe). The 'Apply' button is also present here in a red box.

When you select **Create**, an Application Insights resource is created with your function app, which has the `APPLICATIONINSIGHTS_CONNECTION_STRING` set in application settings. Everything is ready to go.

Add to an existing function app

If an Application Insights resource wasn't created with your function app, use the following steps to create the resource. You can then add the connection string from that resource as an [application setting](#) in your function app.

1. In the [Azure portal](#), search for and select **function app**, and then select your function app.
2. Select the **Application Insights is not configured** banner at the top of the window. If you don't see this banner, then your app might already have Application Insights enabled.

The screenshot shows the Azure Functions Overview page for a resource named "functions-1". The left sidebar lists various management options like Activity log, Access control (IAM), Tags, etc. The main pane displays resource details: Resource group (change) is "myResourceGroup", Status is "Stopped", Location is "Central US", Subscription is "Subscription", Subscription ID is "11111111-1111-1111-1111-111111111111", and Tags (change) is "Click here to add tags". A prominent red box highlights a warning message: "Application Insights is not configured. Configure Application Insights to capture function logs." Below the details are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

3. Expand **Change your resource** and create an Application Insights resource by using the settings specified in the following table:

[Expand table](#)

Setting	Suggested value	Description
New resource name	Unique app name	It's easiest to use the same name as your function app, which must be unique in your subscription.
Location	West Europe	If possible, use the same region as your function app, or the one that's close to that region.

The screenshot shows the "Application Insights" configuration window. It includes sections for enabling monitoring, linking to existing resources, and creating new resources. The "Create new resource" section is active, with fields for "New resource name *" (set to "functions-rg") and "Location *" (set to "West Europe"). Both fields are highlighted with a red border. Below these fields are search and filter options, and a table of top 5 relevant resources. The "Apply" button at the bottom of the table is also highlighted with a red border.

4. Select **Apply**.

The Application Insights resource is created in the same resource group and subscription as your function app. After the resource is created, close the **Application Insights** window.

5. In your function app, expand **Settings**, and then select **Environment variables**. In the **App settings** tab, if you see an app setting named `APPLICATIONINSIGHTS_CONNECTION_STRING`, Application Insights integration is enabled for your function app running in Azure. If this setting doesn't exist, add it by using your Application Insights connection string as the value.

ⓘ Note

Older function apps might use `APPINSIGHTS_INSTRUMENTATIONKEY` instead of `APPLICATIONINSIGHTS_CONNECTION_STRING`. When possible, update your app to use the connection string instead of the instrumentation key.

Disable built-in logging

Early versions of Functions used built-in monitoring, which is no longer recommended. When you enable Application Insights, disable the built-in logging that uses Azure Storage. The built-in logging is useful for testing with light workloads, but isn't intended for high-load production use. For production monitoring, we recommend Application Insights. If you use built-in logging in production, the logging record might be incomplete because of throttling on Azure Storage.

To disable built-in logging, delete the `AzureWebJobsDashboard` app setting. For more information about how to delete app settings in the Azure portal, see the **Application settings** section of [How to manage a function app](#). Before you delete the app setting, ensure that no existing functions in the same function app use the setting for Azure Storage triggers or bindings.

Solutions with high volume of telemetry

Function apps are an essential part of solutions that can cause high volumes of telemetry, such as IoT solutions, rapid event driven solutions, high load financial systems, and integration systems. In this case, you should consider extra configuration to reduce costs while maintaining observability.

The generated telemetry can be consumed in real-time dashboards, alerting, detailed diagnostics, and so on. Depending on how the generated telemetry is consumed, you need to define a strategy to reduce the volume of data generated. This strategy allows you to properly monitor, operate, and diagnose your function apps in production. Consider the following options:

- **Use sampling:** As mentioned [previously](#), sampling helps to dramatically reduce the volume of telemetry events ingested while maintaining a statistically correct analysis. It could happen that even using sampling you still get a high volume of telemetry. Inspect the options that [adaptive sampling](#) provides to you. For example, set the `maxTelemetryItemsPerSecond` to a value that balances the volume generated with your monitoring needs. Keep in mind that the telemetry sampling is applied per host executing your function app.
- **Default log level:** Use `Warning` or `Error` as the default value for all telemetry categories. Later, you can decide which [categories](#) you want to set at the `Information` level, so that you can monitor and diagnose your functions properly.
- **Tune your functions telemetry:** With the default log level set to `Error` or `Warning`, no detailed information from each function is gathered (dependencies, custom metrics, custom events, and traces). For those functions that are key for production monitoring, define an explicit entry for the `Function`.

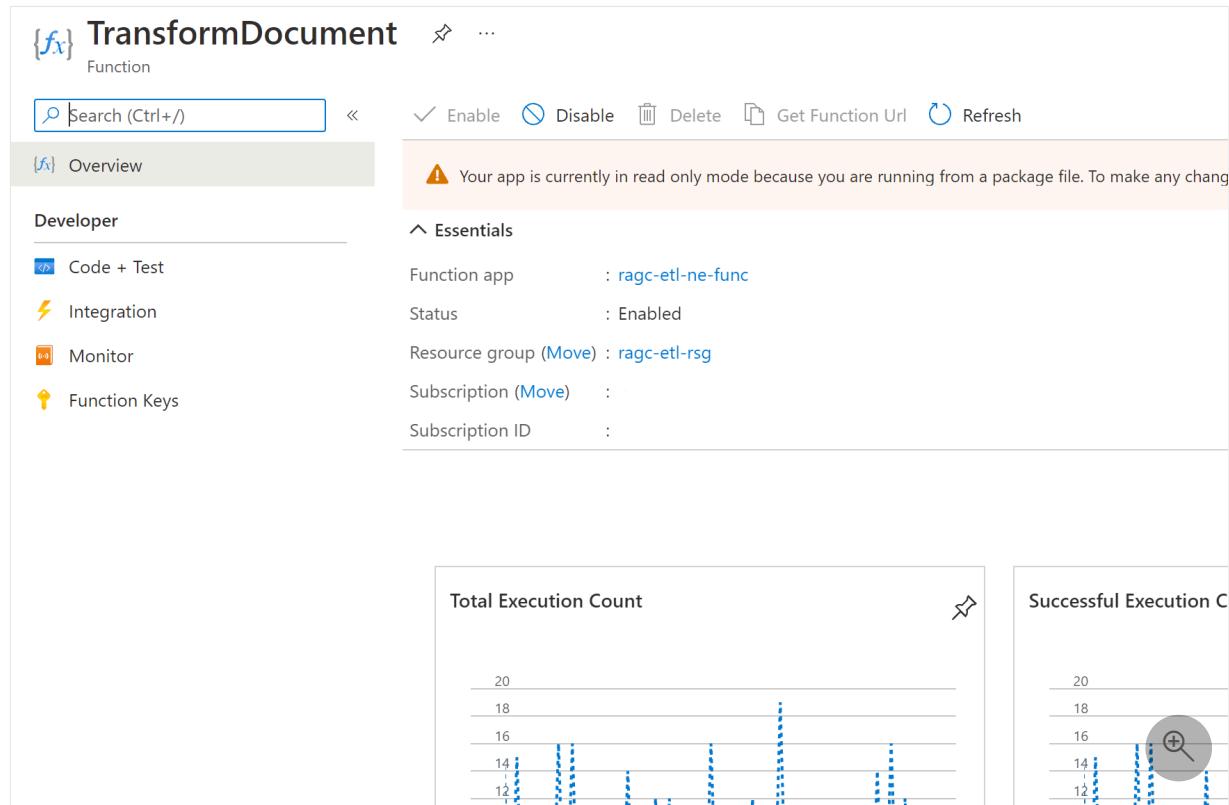
`<YOUR_FUNCTION_NAME>` category and set it to `Information`, so that you can gather detailed information.

To avoid gathering [user-generated logs](#) at the `Information` level, set the `Function`.

`<YOUR_FUNCTION_NAME>.User` category to the `Error` or `Warning` log level.

- **Host.Aggregator category:** As described in [configure categories](#), this category provides aggregated information of function invocations. The information from this category is gathered in the Application Insights `customMetrics` table, and is shown in the function **Overview** tab in the Azure portal. Depending on how you configure the aggregator, consider that there can be a delay, determined by the `flushTimeout` setting, in the telemetry gathered. If you set this category to a value different from `Information`, you stop gathering the data in the `customMetrics` table and don't display metrics in the function **Overview** tab.

The following screenshot shows `Host.Aggregator` telemetry data displayed in the function **Overview** tab:



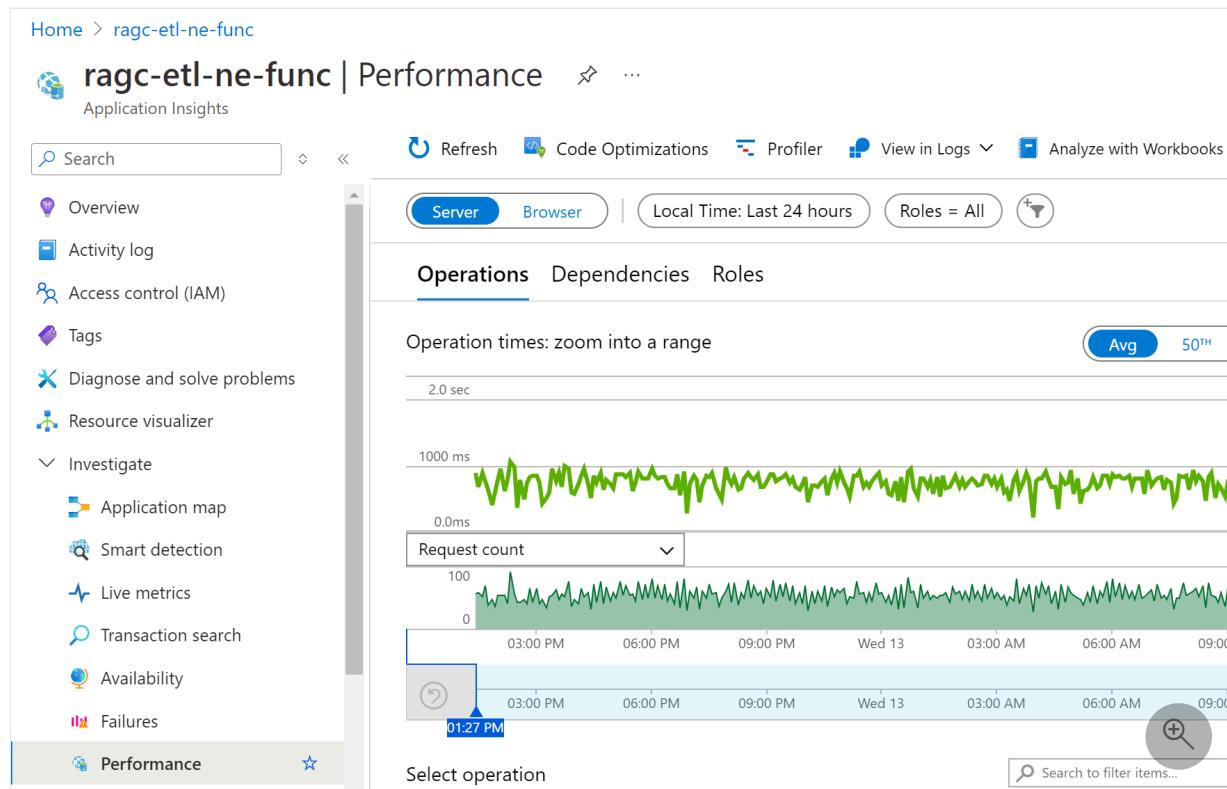
The following screenshot shows `Host.Aggregator` telemetry data in Application Insights `customMetrics` table:

- **Host.Results category:** As described in [configure categories](#), this category provides the runtime-generated logs indicating the success or failure of a function invocation. The information from this category is gathered in the Application Insights `requests` table, and is shown in the function **Monitor** tab and in different Application Insights dashboards (Performance, Failures, and so on). If you set this category to a value different than `Information`, you gather only telemetry generated at the log level defined (or higher). For example, setting it to `error` results in tracking requests data only for failed executions.

The following screenshot shows the `Host.Results` telemetry data displayed in the function **Monitor** tab:

Date (UTC)	Success	Result Code
2021-10-13 11:30:08.761	✓ Success	0
2021-10-13 11:30:07.451	✓ Success	0
2021-10-13 11:30:06.123	✓ Success	0
2021-10-13 11:30:04.746	✓ Success	0
2021-10-13 11:28:18.380	✓ Success	0

The following screenshot shows `Host.Results` telemetry data displayed in Application Insights Performance dashboard:



- **Host.Aggregator vs Host.Results:** Both categories provide good insights about function executions. If needed, you can remove the detailed information from one of these categories, so that you can use the other for monitoring and alerting. Here's a sample:

```
v2.x+  
  
JSON  
  
{  
  "version": "2.0",  
  "logging": {  
    "logLevel": {  
      "default": "Warning",  
      "Function": "Error",  
      "Host.Aggregator": "Error",  
      "Host.Results": "Information",  
      "Function.Function1": "Information",  
      "Function.Function1.User": "Error"  
    },  
    "applicationInsights": {  
      "samplingSettings": {  
        "isEnabled": true,  
        "maxTelemetryItemsPerSecond": 1,  
        "excludedTypes": "Exception"  
      }  
    }  
  }  
}
```

With this configuration:

- The default value for all functions and telemetry categories is set to `Warning` (including Microsoft and Worker categories). So, by default, all errors and warnings generated by runtime and custom logging are gathered.
- The `Function` category log level is set to `Error`, so for all functions, by default, only exceptions and error logs are gathered. Dependencies, user-generated metrics, and user-generated events are skipped.
- For the `Host.Aggregator` category, as it's set to the `Error` log level, aggregated information from function invocations isn't gathered in the `customMetrics` Application Insights table, and information about executions counts (total, successful, and failed) aren't shown in the function overview dashboard.
- For the `Host.Results` category, all the host execution information is gathered in the `requests` Application Insights table. All the invocations results are shown in the function Monitor dashboard and in Application Insights dashboards.
- For the function called `Function1`, we set the log level to `Information`. So, for this concrete function, all the telemetry is gathered (dependency, custom metrics, and custom events). For the same function, we set the `Function1.User` category (user-generated traces) to `Error`, so only custom error logging is gathered.

 **Note**

Configuration per function isn't supported in v1.x of the Functions runtime.

- Sampling is configured to send one telemetry item per second per type, excluding the exceptions. This sampling happens for each server host running our function app. So, if we have four instances, this configuration emits four telemetry items per second per type and all the exceptions that might occur.

 **Note**

Metric counts such as request rate and exception rate are adjusted to compensate for the sampling rate, so that they show approximately correct values in Metric Explorer.

 **Tip**

Experiment with different configurations to ensure that you cover your requirements for logging, monitoring, and alerting. Also, ensure that you have detailed diagnostics in case of unexpected errors or malfunctioning.

Overriding monitoring configuration at runtime

Finally, there could be situations where you need to quickly change the logging behavior of a certain category in production, and you don't want to make a whole deployment just for a change in the `host.json` file. For such cases, you can override the [host.json values](#).

To configure these values at App settings level (and avoid redeployment on just `host.json` changes), you should override specific `host.json` values by creating an equivalent value as an application setting. When the runtime finds an application setting in the format `AzureFunctionsJobHost__path_to_setting`, it overrides the

equivalent `host.json` setting located at `path.to.setting` in the JSON. When expressed as an application setting, a double underscore (`__`) replaces the dot (`.`) used to indicate JSON hierarchy. For example, you can use the following app settings to configure individual function log levels in `host.json`.

[Expand table](#)

Host.json path	App setting
logging.logLevel.default	AzureFunctionsJobHost__logging__logLevel__default
logging.logLevel.Host.Aggregator	AzureFunctionsJobHost__logging__logLevel__Host__Aggregator
logging.logLevel.Function	AzureFunctionsJobHost__logging__logLevel__Function
logging.logLevel.Function.Function1	AzureFunctionsJobHost__logging__logLevel__Function__Function1
logging.logLevel.Function.Function1.User	AzureFunctionsJobHost__logging__logLevel__Function__Function1__User

You can override the settings directly at the Azure portal Function App Configuration pane or by using an Azure CLI or PowerShell script.

Azure CLI

Azure CLI

```
az functionapp config appsettings set --name MyFunctionApp --resource-group MyResourceGroup  
--settings "AzureFunctionsJobHost__logging__logLevel__Host__Aggregator=Information"
```

Note

Overriding the `host.json` through changing app settings will restart your function app. App settings that contain a period aren't supported when running on Linux in an Elastic Premium plan or a Dedicated (App Service) plan. In these hosting environments, you should continue to use the `host.json` file.

Monitor function apps using Health check

You can use the Health Check feature to monitor function apps on the Premium (Elastic Premium) and Dedicated (App Service) plans. Health check isn't an option for the Consumption plan. To learn how to configure it, see [Monitor App Service instances using Health check](#). Your function app should have an HTTP trigger function that responds with an HTTP status code of 200 on the same endpoint as configured on the `Path` parameter of the health check. You can also have that function perform extra checks to ensure that dependent services are reachable and working.

Related content

For more information about monitoring, see:

- [Monitor Azure Functions](#)
- [Analyze Azure Functions telemetry data in Application Insights](#)
- [Application Insights overview](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Analyze Azure Functions telemetry in Application Insights

Article • 01/09/2023

Azure Functions integrates with Application Insights to better enable you to monitor your function apps. Application Insights collects telemetry data generated by your function app, including information your app writes to logs. Application Insights integration is typically enabled when your function app is created. If your function app doesn't have the instrumentation key set, you must first [enable Application Insights integration](#).

By default, the data collected from your function app is stored in Application Insights. In the [Azure portal](#), Application Insights provides an extensive set of visualizations of your telemetry data. You can drill into error logs and query events and metrics. This article provides basic examples of how to view and query your collected data. To learn more about exploring your function app data in Application Insights, see [What is Application Insights?](#).

To be able to view Application Insights data from a function app, you must have at least Contributor role permissions on the function app. You also need to have the [Monitoring Reader permission](#) on the Application Insights instance. You have these permissions by default for any function app and Application Insights instance that you create.

To learn more about data retention and potential storage costs, see [Data collection, retention, and storage in Application Insights](#).

Viewing telemetry in Monitor tab

With [Application Insights integration enabled](#), you can view telemetry data in the **Monitor** tab.

1. In the function app page, select a function that has run at least once after Application Insights was configured. Then, select **Monitor** from the left pane. Select **Refresh** periodically, until the list of function invocations appears.

HttpTrigger1 (myfunctionapp/HttpTrigger1) | Monitor

Search (Ctrl+ /) < Refresh Live app metrics

Overview Developer Monitor Function Keys

Application Insights Instance myfunctionapp Success count in last 30 days 16 Error count in last 30 days 0 Query returned 16 items Run in Application Insights

Troubleshoot your app Diagnose and solve problems

DATE (UTC) ▾	SUCCESS ▾	RESULT CODE ▾	DURATION (MS) ▾
2020-04-02 02:21:54.865	✓	200	20.767
2020-04-02 02:15:06.467	✓	200	211.77380000000002
2020-04-01 01:16:42.620	✓	200	2.5682
2020-04-01 01:16:41.936	✓	200	2.7289000000000003
2020-04-01 01:16:41.104	✓	200	9.6564
2020-04-01 01:16:39.455	✓	200	150.5889

ⓘ Note

It can take up to five minutes for the list to appear while the telemetry client batches data for transmission to the server. The delay doesn't apply to the **Live Metrics Stream**. That service connects to the Functions host when you load the page, so logs are streamed directly to the page.

2. To see the logs for a particular function invocation, select the **Date (UTC)** column link for that invocation. The logging output for that invocation appears in a new page.

Refresh Live app metrics

Application Insights Instance FunctionMonitoringExamples Success count in last 30 days 6

Run in Application Insights

DATE (UTC) ▾	SUCCESS ▾
2020-02-04 16:58:33.944	✓
2020-02-04 16:56:09.088	✓
2020-02-04 16:55:45.908	✓
2020-02-04 16:54:59.215	✓
2020-02-04 16:54:43.857	✓
2020-02-04 16:54:27.492	✓

Invocation Details

Run in Application Insights

DATE (UTC)	MESSAGE	LOG LEVEL
2020-02-04 16:58:33.963	Executing 'CustomTelemetry' (Reason='This function was... Information	
2020-02-04 16:58:33.963	C# HTTP trigger function processed a request. Information	
2020-02-04 16:58:33.964	Executed 'CustomTelemetry' (Succeeded, Id=04d3510d-b... Information	

3. Choose **Run in Application Insights** to view the source of the query that retrieves the Azure Monitor log data in Azure Log. If this is your first time using Azure Log Analytics in your subscription, you're asked to enable it.
4. After you enable Log Analytics, the following query is displayed. You can see that the query results are limited to the last 30 days (`where timestamp > ago(30d)`), and the results show no more than 20 rows (`take 20`). In contrast, the invocation details list for your function is for the last 30 days with no limit.

The screenshot shows the Azure Application Insights Analytics interface. On the left, the schema browser lists 'functions-ggailey777' under 'Active'. Under 'APPLICATION INSIGHTS', 'requests' is selected. The main area displays a table of query results for completed requests. The table has columns: timestamp [UTC], id, operation_Name, success, resultCode, duration, and operation_Id. The data shows multiple entries for 'HttpTriggerCSharp1' requests, each with a timestamp between April 2019 and May 2019, a unique ID, and a duration ranging from approximately 0.5 seconds to over 100 seconds.

timestamp [UTC]	id	operation_Name	success	resultCode	duration	operation_Id
2019-04-08T07:16:10.763	f0hzN/V0p2c=.7fc2b698_	HttpTriggerCSharp1	True	0	6.4631	f0hzN/V0p2c=
2019-04-08T07:15:19.398	rFjoQUPKePU=.7fc2b67c_	HttpTriggerCSharp1	True	0	2.1411	rFjoQUPKePU=
2019-04-08T07:15:05.407	ucHluRhA950=.7fc2b675_	HttpTriggerCSharp1	True	0	10.5045	ucHluRhA950=
2019-04-08T07:05:19.357	IYZcQGz0TOY=.7fc2b558_	HttpTriggerCSharp1	True	0	8.0953	IYZcQGz0TOY=
2019-04-08T07:04:46.803	49UmDByca80=.7fc2b5..._	HttpTriggerCSharp1	True	0	8.1741	49UmDByca80=
2019-04-08T07:04:40.637	h4buunHlwDE=.7fc2b5..._	HttpTriggerCSharp1	True	0	133.798	h4buunHlwDE=
2019-04-08T06:55:59.997	chph7dbHTpU=.5220de..._	HttpTriggerCSharp1	True	0	4.2124	chph7dbHTpU=
2019-04-08T06:55:54.849	QvSRNIPm1CA=.5220de..._	HttpTriggerCSharp1	True	0	2.4622	QvSRNIPm1CA=
2019-04-08T06:55:44.079	9ozxDVdijYc=.5220de23_	HttpTriggerCSharp1	True	0	2.0653	9ozxDVdijYc=
2019-04-08T06:53:19.047	85i2g+dRH4=.5220dd..._	HttpTriggerCSharp1	True	0	1.8892	85i2g+dRH4=
2019-04-08T06:53:00.107	9Q7RJ2ISXzs=.5220ddd2_	HttpTriggerCSharp1	True	0	2.1717	9Q7RJ2ISXzs=
2019-04-08T06:52:21.397	Trvet9BXYt8=.5220ddbf_	HttpTriggerCSharp1	True	0	6.7988	Trvet9BXYt8=

For more information, see [Query telemetry data](#) later in this article.

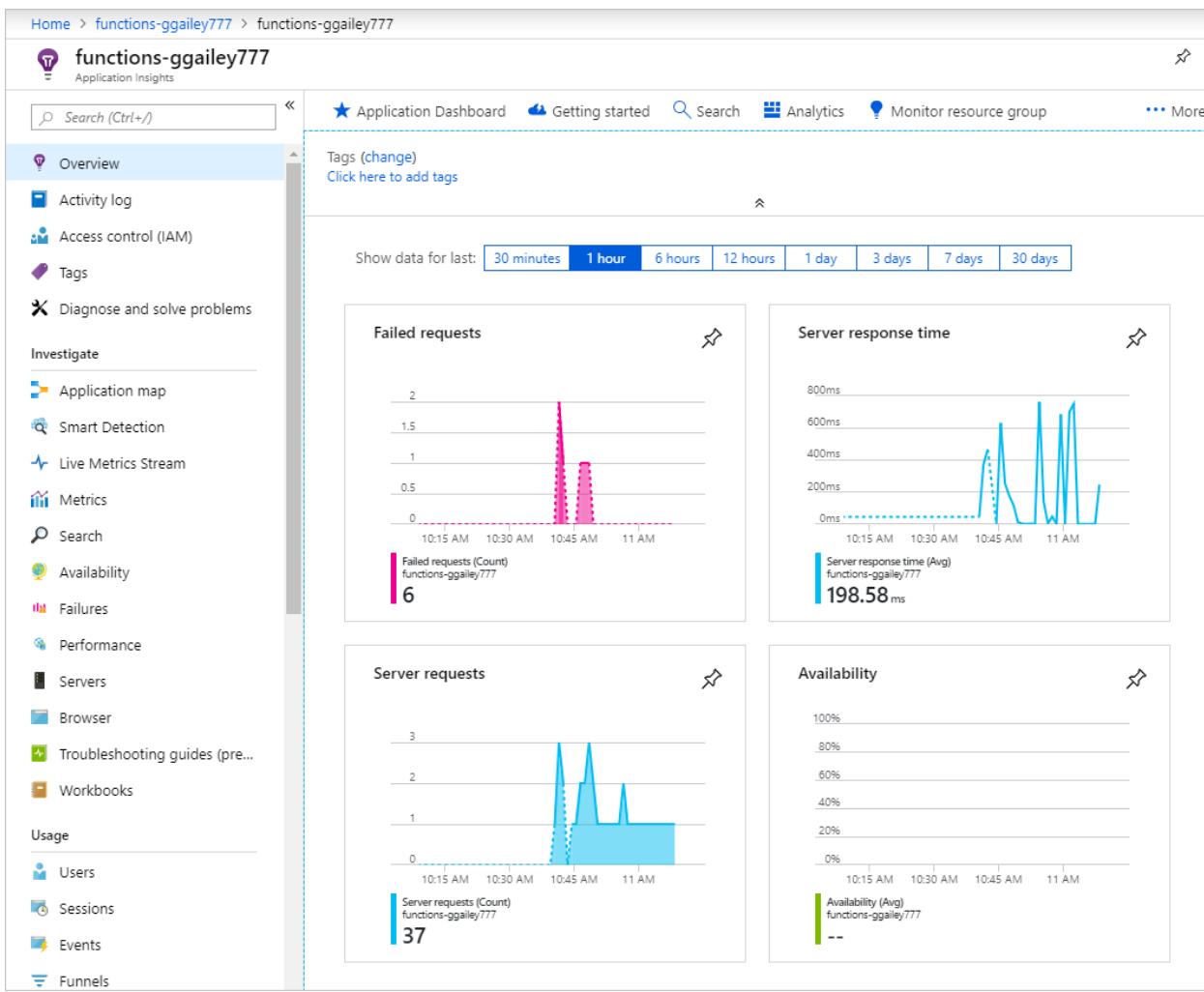
View telemetry in Application Insights

To open Application Insights from a function app in the [Azure portal](#):

1. Browse to your function app in the portal.
2. Select **Application Insights** under **Settings** in the left page.
3. If this is your first time using Application Insights with your subscription, you'll be prompted to enable it. To do this, select **Turn on Application Insights**, and then select **Apply** on the next page.

The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar contains navigation links: Functions, App keys, App files, Deployment slots, Deployment Center, Configuration, Authentication / Authorizati..., Application Insights (which is highlighted with a red box), and Identity. The main content area displays a status message: "Application Insights is linked through Instrumentation Key in app settings" with a green checkmark icon. Below it is a blue button labeled "Turn on Application Insights". A large red box highlights the "Turn on Application Insights" button.

For information about how to use Application Insights, see the [Application Insights documentation](#). This section shows some examples of how to view data in Application Insights. If you're already familiar with Application Insights, you can go directly to [the sections about how to configure and customize the telemetry data](#).



The following areas of Application Insights can be helpful when evaluating the behavior, performance, and errors in your functions:

Investigate	Description
Failures	Create charts and alerts based on function failures and server exceptions. The Operation Name is the function name. Failures in dependencies aren't shown unless you implement custom telemetry for dependencies.
Performance	Analyze performance issues by viewing resource utilization and throughput per Cloud role instances . This performance data can be useful for debugging scenarios where functions are bogging down your underlying resources.
Metrics	Create charts and alerts that are based on metrics. Metrics include the number of function invocations, execution time, and success rates.
Live Metrics	View metrics data as it's created in near real time.

Query telemetry data

[Application Insights Analytics](#) gives you access to all telemetry data in the form of tables in a database. Analytics provides a query language for extracting, manipulating, and

visualizing the data.

Choose **Logs** to explore or query for logged events.

The screenshot shows the Azure Application Insights Logs blade. On the left, there's a navigation menu with sections like Dashboard, FunctionMonitoringExamples | Logs, Monitoring, Usage, Configure, and Properties. The 'Logs' section is highlighted with a red box. In the center, there's a query editor with a search bar, a 'Run' button, and a 'Time range: Set in query' dropdown. A red box highlights the query itself:

```
requests
| where timestamp > ago(30m)
| summarize count() by cloud_RoleInstance, bin(timestamp, 1m)
| render timechart
```

Below the query, the 'Results' tab is selected, showing a chart titled 'Completed'. The chart displays the count of requests over time, grouped by worker instance. A specific data point is highlighted with a blue box, showing the following details:

timestamp [UTC]: 3/15/2020, 6:47:00.000 PM
count.: 3
cloud.RoleInstance: dc0afb88b06f8acee5b9f72ac727a3520876ed062d9dd904b77e346206763032

Here's a query example that shows the distribution of requests per worker over the last 30 minutes.

The screenshot shows a Kusto query editor with the title 'Kusto'. The query is:

```
requests
| where timestamp > ago(30m)
| summarize count() by cloud_RoleInstance, bin(timestamp, 1m)
| render timechart
```

The tables that are available are shown in the **Schema** tab on the left. You can find data generated by function invocations in the following tables:

Table	Description
traces	Logs created by the runtime, scale controller, and traces from your function code.
requests	One request for each function invocation.
exceptions	Any exceptions thrown by the runtime.
customMetrics	The count of successful and failing invocations, success rate, and duration.

Table	Description
customEvents	Events tracked by the runtime, for example: HTTP requests that trigger a function.
performanceCounters	Information about the performance of the servers that the functions are running on.

The other tables are for availability tests, and client and browser telemetry. You can implement custom telemetry to add data to them.

Within each table, some of the Functions-specific data is in a `customDimensions` field. For example, the following query retrieves all traces that have log level `Error`.

```
Kusto
traces
| where customDimensions.LogLevel == "Error"
```

The runtime provides the `customDimensions.LogLevel` and `customDimensions.Category` fields. You can provide additional fields in logs that you write in your function code. For an example in C#, see [Structured logging](#) in the .NET class library developer guide.

Query function invocations

Every function invocation is assigned a unique ID. `InvocationId` is included in the custom dimension and can be used to correlate all the logs from a particular function execution.

```
Kusto
traces
| project customDimensions["InvocationId"], message
```

Telemetry correlation

Logs from different functions can be correlated using `operation_Id`. Use the following query to return all the logs for a specific logical operation.

```
Kusto
traces
| where operation_Id == '45fa5c4f8097239efe14a2388f8b4e29'
```

```
| project timestamp, customDimensions["InvocationId"], message  
| order by timestamp
```

Sampling percentage

Sampling configuration can be used to reduce the volume of telemetry. Use the following query to determine if sampling is operational or not. If you see that `RetainedPercentage` for any type is less than 100, then that type of telemetry is being sampled.

Kusto

```
union requests,dependencies,pageViews,browserTimings,exceptions,traces  
| where timestamp > ago(1d)  
| summarize RetainedPercentage = 100/avg(itemCount) by bin(timestamp, 1h),  
itemType
```

Query scale controller logs

This feature is in preview.

After enabling both [scale controller logging](#) and [Application Insights integration](#), you can use the Application Insights log search to query for the emitted scale controller logs. Scale controller logs are saved in the `traces` collection under the `ScaleControllerLogs` category.

The following query can be used to search for all scale controller logs for the current function app within the specified time period:

Kusto

```
traces  
| extend CustomDimensions = todynamic(tostring(customDimensions))  
| where CustomDimensions.Category == "ScaleControllerLogs"
```

The following query expands on the previous query to show how to get only logs indicating a change in scale:

Kusto

```
traces  
| extend CustomDimensions = todynamic(tostring(customDimensions))  
| where CustomDimensions.Category == "ScaleControllerLogs"
```

```
| where message == "Instance count changed"
| extend Reason = CustomDimensions.Reason
| extend PreviousInstanceCount = CustomDimensions.PreviousInstanceCount
| extend NewInstanceCount = CustomDimensions.CurrentInstanceCount
```

Consumption plan-specific metrics

When running in a [Consumption plan](#), the execution cost of a single function execution is measured in *GB-seconds*. Execution cost is calculated by combining its memory usage with its execution time. To learn more, see [Estimating Consumption plan costs](#).

The following telemetry queries are specific to metrics that impact the cost of running functions in the Consumption plan.

Determine memory usage

Under **Monitoring**, select **Logs (Analytics)**, then copy the following telemetry query and paste it into the query window and select **Run**. This query returns the total memory usage at each sampled time.

```
performanceCounters
| where name == "Private Bytes"
| project timestamp, name, value
```

The results look like the following example:

timestamp [UTC]	name	value
9/12/2019, 1:05:14.947 AM	Private Bytes	209,932,288
9/12/2019, 1:06:14.994 AM	Private Bytes	212,189,184
9/12/2019, 1:06:30.010 AM	Private Bytes	231,714,816
9/12/2019, 1:07:15.040 AM	Private Bytes	210,591,744
9/12/2019, 1:12:16.285 AM	Private Bytes	216,285,184
9/12/2019, 1:12:31.376 AM	Private Bytes	235,806,720

Determine duration

Azure Monitor tracks metrics at the resource level, which for Functions is the function app. Application Insights integration emits metrics on a per-function basis. Here's an example analytics query to get the average duration of a function:

```
customMetrics  
| where name contains "Duration"  
| extend averageDuration = valueSum / valueCount  
| summarize averageDurationMilliseconds=avg(averageDuration) by name
```

name	averageDurationMilliseconds
QueueTrigger AvgDurationMs	16.087
QueueTrigger MaxDurationMs	90.249
QueueTrigger MinDurationMs	8.522

Next steps

Learn more about monitoring Azure Functions:

- [Monitor Azure Functions](#)
- [How to configure monitoring for Azure Functions](#)

Enable streaming execution logs in Azure Functions

Article • 08/24/2023

While developing an application, you often want to see what's being written to the logs in near real time when running in Azure.

There are two ways to view a stream of log files being generated by your function executions.

- **Built-in log streaming:** the App Service platform lets you view a stream of your application log files. This is equivalent to the output seen when you debug your functions during [local development](#) and when you use the **Test** tab in the portal. All log-based information is displayed. For more information, see [Stream logs](#). This streaming method supports only a single instance, and can't be used with an app running on Linux in a Consumption plan.
- **Live Metrics Stream:** when your function app is [connected to Application Insights](#), you can view log data and other metrics in near real-time in the Azure portal using [Live Metrics Stream](#). Use this method when monitoring functions running on multiple-instances and supports all plan types. This method uses [sampled data](#).

Log streams can be viewed both in the portal and in most local development environments.

Portal

You can view both types of log streams in the portal.

To view streaming logs in the portal, select the **Platform features** tab in your function app. Then, under **Monitoring**, choose **Log streaming**.

The screenshot shows the 'Platform features' section of the Azure portal. At the top, there are tabs for 'Overview', 'Platform features' (which is highlighted with a red box), and 'Log streaming'. Below the tabs is a search bar labeled 'Search features'. The main area is divided into several sections: 'General Settings' (Function app settings, Application settings, Properties, Backups, All settings), 'Code Deployment' (Deployment Center), 'Development tools' (Logic Apps, Console (CMD / PowerShell), Advanced tools (Kudu), App Service Editor, Resource Explorer, Site Extensions), 'Networking' (Networking, SSL, Custom domains, Authentication / Authorization, Identity, Push notifications), 'Monitoring' (Diagnostic logs, Log streaming, Process explorer, Metrics), 'API' (API definition, CORS), 'App Service plan' (App Service plan, Scale up, Scale out, Quotas), and 'Resource management' (Diagnose and solve problems, Activity log, Access control (IAM), Tags, Locks, Automation script). The 'Log streaming' link under the Monitoring section is also highlighted with a red box.

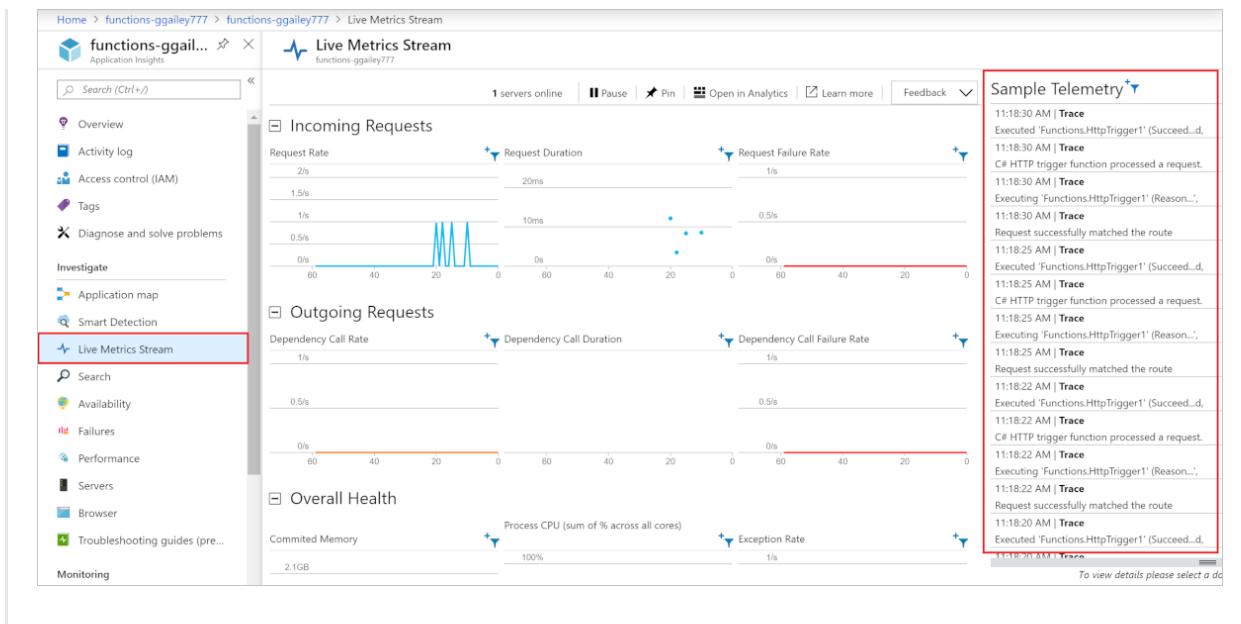
This connects your app to the log streaming service and application logs are displayed in the window. You can toggle between **Application logs** and **Web server logs**.

The screenshot shows the Azure Functions Log streaming interface. At the top, there are tabs for 'Overview', 'Platform features', and 'Log streaming'. Below these are buttons for 'Reconnect', 'Copy logs', 'Pause', and 'Clear'. A tab for 'Application logs' is selected. The main area displays log entries in a monospaced font. The logs show the execution of an 'HttpTriggerCSharp1' function, including requests from the host APIs and timer triggers. The log entries are timestamped and include details like reason codes and IDs.

```
2019-04-08T07:15:19.398 [Information] Executing 'Functions.HttpTriggerCSharp1'  
(Reason='This function was programmatically called via the host APIs.', Id=62b1969f-b568-4291-83ef-691e11d0d2e8)  
2019-04-08T07:15:19.399 [Information] C# HTTP trigger function processed a request.  
2019-04-08T07:15:19.399 [Information] Executed 'Functions.HttpTriggerCSharp1'  
(Succeeded, Id=62b1969f-b568-4291-83ef-691e11d0d2e8)  
2019-04-08T07:15:19.398 [Information] Executing 'Functions.HttpTriggerCSharp1'  
(Reason='This function was programmatically called via the host APIs.', Id=62b1969f-b568-4291-83ef-691e11d0d2e8)  
2019-04-08T07:15:19.399 [Information] C# HTTP trigger function processed a request.  
2019-04-08T07:15:19.399 [Information] Executed 'Functions.HttpTriggerCSharp1'  
(Succeeded, Id=62b1969f-b568-4291-83ef-691e11d0d2e8)  
2019-04-08T07:16:05.008 [Information] Executing 'Functions.TimerTriggerCSharp1'  
(Reason='Timer fired at 2019-04-08T00:16:05.0074271-07:00', Id=3f5f11b1-f243-454f-9e58-6ab7916d68dc)  
2019-04-08T07:16:05.009 [Information] C# Timer trigger function executed at: 4/8/2019  
12:16:05 AM  
2019-04-08T07:16:05.010 [Information] Executed 'Functions.TimerTriggerCSharp1'  
(Succeeded, Id=3f5f11b1-f243-454f-9e58-6ab7916d68dc)  
2019-04-08T07:16:05.008 [Information] Executing 'Functions.TimerTriggerCSharp1'  
(Reason='Timer fired at 2019-04-08T00:16:05.0074271-07:00', Id=3f5f11b1-f243-454f-9e58-6ab7916d68dc)  
2019-04-08T07:16:05.009 [Information] C# Timer trigger function executed at: 4/8/2019  
12:16:05 AM  
2019-04-08T07:16:05.010 [Information] Executed 'Functions.TimerTriggerCSharp1'  
(Succeeded, Id=3f5f11b1-f243-454f-9e58-6ab7916d68dc)  
2019-04-08T07:16:10.768 [Information] Executing 'Functions.HttpTriggerCSharp1'  
(Reason='This function was programmatically called via the host APIs.', Id=7be06fb2-cb0f-4825-8e76-500dc7766891)  
2019-04-08T07:16:10.769 [Information] C# HTTP trigger function processed a request.  
2019-04-08T07:16:10.769 [Information] Executed 'Functions.HttpTriggerCSharp1'  
(Succeeded, Id=7be06fb2-cb0f-4825-8e76-500dc7766891)  
2019-04-08T07:16:10.768 [Information] Executing 'Functions.HttpTriggerCSharp1'
```

To view the Live Metrics Stream for your app, select the **Overview** tab of your function app. When you have Application Insights enabled, you see an **Application Insights** link under **Configured features**. This link takes you to the Application Insights page for your app.

In Application Insights, select **Live Metrics Stream**. **Sampled log entries** are displayed under **Sample Telemetry**.



Next steps

- Monitor Azure Functions
- Analyze Azure Functions telemetry in Application Insights

Use OpenTelemetry with Azure Functions

Article • 05/21/2024

ⓘ Important

OpenTelemetry support for Azure Functions is currently in preview, and your app must be hosted in a [Flex Consumption plan](#) to use OpenTelemetry.

This article shows you how to configure your function app to export log and trace data in an OpenTelemetry format. Azure Functions generates telemetry data on your function executions from both the Functions host process and the language-specific worker process in which your function code runs. By default, this telemetry data is sent to Application Insights using the Application Insights SDK. However, you can choose to export this data using OpenTelemetry semantics. While you can still use an OpenTelemetry format to send your data to Application Insights, you can now also export the same data to any other OpenTelemetry-compliant endpoint.

ⓘ Tip

Because this article is targeted at your development language of choice, remember to choose the correct language at the top of the article.

OpenTelemetry currently isn't supported for C# in-process apps.

You can obtain these benefits by enabling OpenTelemetry in your function app:

- Correlation across traces and logs being generated both at the host and in your application code.
- Consistent, standards-based generation of exportable telemetry data.
- Integrates with other providers that can consume OpenTelemetry-compliant data.

OpenTelemetry is enabled at the function app level, both in host configuration (`host.json`) and in your code project. Functions also provides a client optimized experience for exporting OpenTelemetry data from your function code that's running in a language-specific worker process.

1. Enable OpenTelemetry in the Functions host

When you enable OpenTelemetry output in the function app's host.json file, your host exports OpenTelemetry output regardless of the language stack used by your app.

To enable OpenTelemetry output from the Functions host, update the [host.json file](#) in your code project to add a `"telemetryMode": "openTelemetry"` element to the root collection. With OpenTelemetry enabled, your host.json file might look like this:

```
JSON

{
    "version": "2.0",
    "logging": {
        "applicationInsights": {
            "samplingSettings": {
                "isEnabled": true,
                "excludedTypes": "Request"
            },
            "enableLiveMetricsFilters": true
        }
    },
    "telemetryMode": "openTelemetry"
}
```

2. Configure application settings

When OpenTelemetry is enabled in the host.json file, the endpoints to which data is sent is determined based on which OpenTelemetry-supported application settings are available in your app's environment variables.

Create specific application settings in your function app based on the OpenTelemetry output destination. When connection settings are provided for both Application Insights and an OpenTelemetry protocol (OTLP) exporter, OpenTelemetry data is sent to both endpoints.

Application Insights

`APPLICATIONINSIGHTS_CONNECTION_STRING`: the connection string for an Application Insights workspace. When this setting exists, OpenTelemetry data is sent to that workspace. This setting is the same one used to connect to Application Insights without OpenTelemetry enabled. If your app doesn't already have this setting, you might need to [Enable Application Insights integration](#).

3. Enable OpenTelemetry in your app

With the Functions host configured to use OpenTelemetry, you should also update your application code to output OpenTelemetry data. Enabling OpenTelemetry in both the host and your application code enables better correlation between traces and logs emitted both by the Functions host process and from your language worker process.

The way that you instrument your application to use OpenTelemetry depends on your target OpenTelemetry endpoint:

1. Run these commands to install the required assemblies in your app:

Application Insights

Windows Command Prompt

```
dotnet add package Microsoft.Azure.Functions.Worker.OpenTelemetry -  
-version 1.0.0-preview1  
dotnet add package OpenTelemetry.Extensions.Hosting  
dotnet add package Azure.Monitor.OpenTelemetry.AspNetCore
```

2. In your Program.cs project file, add this `using` statement:

Application Insights

C#

```
using Azure.Monitor.OpenTelemetry.AspNetCore;
```

3. In the `ConfigureServices` delegate, add this service configuration:

Application Insights

C#

```
services.AddOpenTelemetry()  
.UseFunctionsWorkerDefaults()  
.UseAzureMonitor();
```

To export to both OpenTelemetry endpoints, call both `UseAzureMonitor` and `UseOtlpExporter`.

Considerations for OpenTelemetry

When you export your data using OpenTelemetry, keep these current considerations in mind.

- When the host is configured to use OpenTelemetry, only logs and traces are exported. Host metrics aren't currently exported.
- You can't currently run your app project locally using Core Tools when you have OpenTelemetry enabled in the host. You currently need to deploy your code to Azure to validate your OpenTelemetry-related updates.
- At this time, only HTTP trigger and Azure SDK-based triggers are supported with OpenTelemetry outputs.

Related content

[Monitor Azure Functions Flex Consumption plan](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Configure your App Service or Azure Functions app to use Microsoft Entra sign-in

Article • 05/21/2024

ⓘ Note

Beginning June 1, 2024, all newly created App Service apps will have the option to create a unique default hostname with a naming convention of <app-name>-<random-hash>. <region>.azurewebsites.net. The names of existing apps will not change.

Example: myapp-ds27dh7271aah175.westus-01.azurewebsites.net

For more information, refer to [Unique Default Hostname for App Service Resource](#).

Select another authentication provider to jump to it.

This article shows you how to configure authentication for Azure App Service or Azure Functions so that your app signs in users with the [Microsoft identity platform](#) (Microsoft Entra) as the authentication provider.

Choose a tenant for your application and its users

Before your application can sign in users, you first need to register it in a workforce or external tenant. If you're making your app available to employee or business guests, register your app in a workforce tenant. If your app is for consumers and business customers, register it in an external tenant.

1. Sign in to the [Azure portal](#) and navigate to your app.
2. On your app's left menu, select **Authentication**, and then select **Add identity provider**.
3. In the **Add an identity provider** page, select **Microsoft** as the **Identity provider** to sign in Microsoft and Microsoft Entra identities.

4. For **Tenant type**, select **Workforce configuration (current tenant)** for employees and business guests or select **External configuration** for consumers and business customers.

Choose the app registration

The App Service Authentication feature can automatically create an app registration for you or you can use a registration that you or a directory admin created separately.

Create a new app registration automatically, unless you need to create an app registration separately. You can customize the app registration in the [Microsoft Entra admin center](#) later if you want.

The following situations are the most common cases to use an existing app registration:

- Your account doesn't have permissions to create app registrations in your Microsoft Entra tenant.
- You want to use an app registration from a different Microsoft Entra tenant than the one your app is in.
- The option to create a new registration isn't available for government clouds.

Workforce configuration

[Create and use a new app registration](#) or [use an existing registration created separately](#).

Option 1: Create and use a new app registration

Use this option unless you need to create an app registration separately. You can customize the app registration in Microsoft Entra once it's created.

Note

The option to create a new registration automatically isn't available for government clouds. Instead, [define a registration separately](#).

Enter the **Name** for the new app registration.

Select the **Supported account type**:

- **Current tenant - Single tenant.** Accounts in this organizational directory only. All user and guest accounts in your directory can use your application or API.

Use this option if your target audience is internal to your organization.

- **Any Microsoft Entra directory - Multitenant.** Accounts in any organizational directory. All users with a work or school account from Microsoft can use your application or API. This includes schools and businesses that use Office 365. Use this option if your target audience is business or educational customers and to enable multitenancy.
- **Any Microsoft Entra directory & personal Microsoft accounts.** Accounts in any organizational directory and personal Microsoft accounts (for example, Skype, Xbox). All users with a work or school, or personal Microsoft account can use your application or API. It includes schools and businesses that use Office 365 as well as personal accounts that are used to sign in to services like Xbox and Skype. Use this option to target the widest set of Microsoft identities and to enable multitenancy.
- **Personal Microsoft accounts only.** Personal accounts that are used to sign in to services like Xbox and Skype. Use this option to target the widest set of Microsoft identities.

You can change the name of the registration or the supported account types later if you want.

A client secret is created as a slot-sticky [application setting](#) named `MICROSOFT_PROVIDER_AUTHENTICATION_SECRET`. You can update that setting later to use [Key Vault references](#) if you wish to manage the secret in Azure Key Vault.

Option 2: Use an existing registration created separately

Either:

- Select **Pick an existing app registration in this directory** and select an app registration from the drop-down.
- Select **Provide the details of an existing app registration** and provide:
 - Application (client) ID.
 - Client secret (recommended). A secret value that the application uses to prove its identity when requesting a token. This value is saved in your app's configuration as a slot-sticky application setting named `MICROSOFT_PROVIDER_AUTHENTICATION_SECRET`. If the client secret isn't set, sign-in operations from the service use the OAuth 2.0 implicit grant flow, which isn't* recommended.
 - Issuer URL, which takes the form `<authentication-endpoint>/<tenant-id>/v2.0`. Replace `<authentication-endpoint>` with the authentication

endpoint value specific to the cloud environment. For example, a workforce tenant in global Azure would use "https://login.microsoftonline.com" as its authentication endpoint.

If you need to manually create an app registration in a workforce tenant, follow the [register an application](#) quickstart. As you go through the registration process, be sure to note the application (client) ID and client secret values.

During the registration process, in the **Redirect URIs** section, select **Web** for platform and type <app-url>/.auth/login/aad/callback. For example,

`https://contoso.azurewebsites.net/.auth/login/aad/callback`.

After creation, modify the app registration:

1. From the left navigation, select **Expose an API** > **Add** > **Save**. This value uniquely identifies the application when it's used as a resource, allowing tokens to be requested that grant access. It's used as a prefix for scopes you create.

For a single-tenant app, you can use the default value, which is in the form `api://<application-client-id>`. You can also specify a more readable URI like `https://contoso.com/api` based on one of the verified domains for your tenant. For a multitenant app, you must provide a custom URI. To learn more about accepted formats for App ID URIs, see the [app registrations best practices reference](#).

2. Select **Add a scope**.
 - a. In **Scope name**, enter *user_impersonation*.
 - b. In **Who can consent**, select **Admins and users** if you want to allow users to consent to this scope.
 - c. In the text boxes, enter the consent scope name and description you want users to see on the consent page. For example, enter *Access <application-name>*.
 - d. Select **Add scope**.
3. (Recommended) To create a client secret:
 - a. From the left navigation, select **Certificates & secrets** > **Client secrets** > **New client secret**.
 - b. Enter a description and expiration and select **Add**.
 - c. In the **Value** field, copy the client secret value. It won't be shown again once you navigate away from this page.

4. (Optional) To add multiple Reply URLs, select Authentication.

Configure additional checks

Configure Additional checks, which determines which requests are allowed to access your application. You can customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**.

For **Client application requirement**, choose whether to:

- Allow requests only from this application itself
- Allow requests from specific client applications
- Allow requests from any application (Not recommended)

For **Identity requirement**, choose whether to:

- Allow requests from any identity
- Allow requests from specific identities

For **Tenant requirement**, choose whether to:

- Allow requests only from the issuer tenant
- Allow requests from specific tenants
- Use default restrictions based on issuer

Your app may still need to make additional authorization decisions in code. For more information, see [Use a built-in authorization policy](#).

Configure authentication settings

These options determine how your application responds to unauthenticated requests, and the default selections will redirect all requests to sign in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

For **Restrict access**, decide whether to:

- Require authentication
- Allow unauthenticated access

For **Unauthenticated requests**

- HTTP 302 Found redirect: recommended for websites
- HTTP 401 Unauthorized: recommended for APIs
- HTTP 403 Forbidden
- HTTP 404 Not found

Select **Token store** (recommended). The token store collects, stores, and refreshes tokens for your application. You can disable this later if your app doesn't need tokens or you need to optimize performance.

Add the identity provider

If you selected workforce configuration, you can select **Next: Permissions** and add any Microsoft Graph permissions needed by the application. These will be added to the app registration, but you can also change them later. If you selected external configuration, you can add Microsoft Graph permissions later.

Select **Add**.

You're now ready to use the Microsoft identity platform for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

For an example of configuring Microsoft Entra sign-in for a web app that accesses Azure Storage and Microsoft Graph, see [this tutorial](#).

Authorize requests

By default, App Service Authentication only handles authentication, determining if the caller is who they say they are. Authorization, determining if that caller should have access to some resource, is an extra step beyond authentication. You can learn more about these concepts from [Microsoft identity platform authorization basics](#).

Your app can [make authorization decisions in code](#). App Service Authentication does provide some [built-in checks](#), which can help, but they may not alone be sufficient to cover the authorization needs of your app.

Tip

Multi-tenant applications should validate the issuer and tenant ID of the request as part of this process to make sure the values are allowed. When App Service Authentication is configured for a multi-tenant scenario, it doesn't validate which tenant the request comes from. An app may need to be limited to specific tenants,

based on if the organization has signed up for the service, for example. See the [Microsoft identity platform multi-tenant guidance](#).

Perform validations from application code

When you perform authorization checks in your app code, you can use the [claims information that App Service Authentication makes available](#). The injected `x-ms-client-principal` header contains a Base64-encoded JSON object with the claims asserted about the caller. By default, these claims go through a claims mapping, so the claim names may not always match what you would see in the token. For example, the `tid` claim is mapped to `http://schemas.microsoft.com/identity/claims/tenantid` instead.

You can also work directly with the underlying access token from the injected `x-ms-token-aad-access-token` header.

Use a built-in authorization policy

The created app registration authenticates incoming requests for your Microsoft Entra tenant. By default, it also lets anyone within the tenant to access the application, which is fine for many applications. However, some applications need to restrict access further by making authorization decisions. Your application code is often the best place to handle custom authorization logic. However, for common scenarios, the Microsoft identity platform provides built-in checks that you can use to limit access.

This section shows how to enable built-in checks using the [App Service authentication V2 API](#). Currently, the only way to configure these built-in checks is via [Azure Resource Manager templates](#) or the [REST API](#).

Within the API object, the Microsoft Entra identity provider configuration has a `validation` section that can include a `defaultAuthorizationPolicy` object as in the following structure:

JSON

```
{  
  "validation": {  
    "defaultAuthorizationPolicy": {  
      "allowedApplications": [],  
      "allowedPrincipals": {  
        "identities": []  
      }  
    }  
  }  
}
```

```
    }  
}
```

[+] Expand table

Property	Description
<code>defaultAuthorizationPolicy</code>	A grouping of requirements that must be met in order to access the app. Access is granted based on a logical <code>AND</code> over each of its configured properties. When <code>allowedApplications</code> and <code>allowedPrincipals</code> are both configured, the incoming request must satisfy both requirements in order to be accepted.
<code>allowedApplications</code>	An allowlist of string application client IDs representing the client resource that is calling into the app. When this property is configured as a nonempty array, only tokens obtained by an application specified in the list will be accepted. This policy evaluates the <code>appid</code> or <code>azp</code> claim of the incoming token, which must be an access token. See the Microsoft identity platform claims reference .
<code>allowedPrincipals</code>	A grouping of checks that determine if the principal represented by the incoming request may access the app. Satisfaction of <code>allowedPrincipals</code> is based on a logical <code>OR</code> over its configured properties.
<code>identities</code> (under <code>allowedPrincipals</code>)	An allowlist of string object IDs representing users or applications that have access. When this property is configured as a nonempty array, the <code>allowedPrincipals</code> requirement can be satisfied if the user or application represented by the request is specified in the list. There's a limit of 500 characters total across the list of identities. This policy evaluates the <code>oid</code> claim of the incoming token. See the Microsoft identity platform claims reference .

Additionally, some checks can be configured through an [application setting](#), regardless of the API version being used. The `WEBSITE_AUTH_AAD_ALLOWED_TENANTS` application setting can be configured with a comma-separated list of up to 10 tenant IDs (for example, "559a2f9c-c6f2-4d31-b8d6-5ad1a13f8330,5693f64a-3ad5-4be7-b846-e9d1141bcebc") to require that the incoming token is from one of the specified tenants, as specified by the `tid` claim. The `WEBSITE_AUTH_AAD_REQUIRE_CLIENT_SERVICE_PRINCIPAL` application setting can be configured to "true" or "1" to require the incoming token to include an `oid` claim. This setting is ignored and treated as true if

`allowedPrincipals.identities` has been configured (since the `oid` claim is checked against this provided list of identities).

Requests that fail these built-in checks are given an HTTP `403 Forbidden` response.

Configure client apps to access your App Service

In the prior sections, you registered your App Service or Azure Function to authenticate users. This section explains how to register native clients or daemon apps in Microsoft Entra so that they can request access to APIs exposed by your App Service on behalf of users or themselves, such as in an N-tier architecture. Completing the steps in this section isn't required if you only wish to authenticate users.

Native client application

You can register native clients to request access your App Service app's APIs on behalf of a signed in user.

1. From the portal menu, select **Microsoft Entra**.
2. From the left navigation, select **App registrations > New registration**.
3. In the **Register an application** page, enter a **Name** for your app registration.
4. In **Redirect URI**, select **Public client (mobile & desktop)** and type the URL `<app-url>/auth/login/aad/callback`. For example,
`https://contoso.azurewebsites.net/.auth/login/aad/callback`.
5. Select **Register**.
6. After the app registration is created, copy the value of **Application (client) ID**.

 **Note**

For a Microsoft Store application, use the [package SID](#) as the URI instead.

7. From the left navigation, select **API permissions > Add a permission > My APIs**.
8. Select the app registration you created earlier for your App Service app. If you don't see the app registration, make sure that you've added the `user_impersonation` scope in the app registration.

9. Under **Delegated permissions**, select **user_impersonation**, and then select **Add permissions**.

You have now configured a native client application that can request access your App Service app on behalf of a user.

Daemon client application (service-to-service calls)

In an N-tier architecture, your client application can acquire a token to call an App Service or Function app on behalf of the client app itself (not on behalf of a user). This scenario is useful for non-interactive daemon applications that perform tasks without a logged in user. It uses the standard OAuth 2.0 [client credentials](#) grant.

1. From the portal menu, select **Microsoft Entra**.
2. From the left navigation, select **App registrations > New registration**.
3. In the **Register an application** page, enter a **Name** for your app registration.
4. For a daemon application, you don't need a Redirect URI so you can keep that empty.
5. Select **Register**.
6. After the app registration is created, copy the value of **Application (client) ID**.
7. From the left navigation, select **Certificates & secrets > Client secrets > New client secret**.
8. Enter a description and expiration and select **Add**.
9. In the **Value** field, copy the client secret value. It won't be shown again once you navigate away from this page.

You can now [request an access token using the client ID and client secret](#) by setting the **resource** parameter to the **Application ID URI** of the target app. The resulting access token can then be presented to the target app using the standard [OAuth 2.0 Authorization header](#), and App Service authentication will validate and use the token as usual to now indicate that the caller (an application in this case, not a user) is authenticated.

At present, this allows *any* client application in your Microsoft Entra tenant to request an access token and authenticate to the target app. If you also want to enforce *authorization* to allow only certain client applications, you must perform some extra configuration.

1. [Define an App Role](#) in the manifest of the app registration representing the App Service or Function app you want to protect.
2. On the app registration representing the client that needs to be authorized, select **API permissions > Add a permission > My APIs**.

3. Select the app registration you created earlier. If you don't see the app registration, make sure that you've [added an App Role](#).
4. Under **Application permissions**, select the App Role you created earlier, and then select **Add permissions**.
5. Make sure to select **Grant admin consent** to authorize the client application to request the permission.
6. Similar to the previous scenario (before any roles were added), you can now [request an access token](#) for the same target `resource`, and the access token will include a `roles` claim containing the App Roles that were authorized for the client application.
7. Within the target App Service or Function app code, you can now validate that the expected roles are present in the token (this isn't performed by App Service authentication). For more information, see [Access user claims](#).

You have now configured a daemon client application that can access your App Service app using its own identity.

Best practices

Regardless of the configuration you use to set up authentication, the following best practices keep your tenant and applications more secure:

- Configure each App Service app with its own app registration in Microsoft Entra.
- Give each App Service app its own permissions and consent.
- Avoid permission sharing between environments by using separate app registrations for separate deployment slots. When you're testing new code, this practice can help prevent issues from affecting the production app.

Migrate to the Microsoft Graph

Some older apps may also have been set up with a dependency on the [deprecated Azure AD Graph](#), which is scheduled for full retirement. For example, your app code may have called Azure AD Graph to check group membership as part of an authorization filter in a middleware pipeline. Apps should move to the [Microsoft Graph](#) by following the [guidance provided by Microsoft Entra as part of the Azure AD Graph deprecation process](#). In following those instructions, you may need to make some changes to your configuration of App Service authentication. Once you have added Microsoft Graph permissions to your app registration, you can:

1. Update the **Issuer URL** to include the "/v2.0" suffix if it doesn't already.

2. Remove requests for Azure AD Graph permissions from your sign-in configuration.

The properties to change depend on [which version of the management API you're using](#):

- If you're using the V1 API (`/authsettings`), this would be in the `additionalLoginParams` array.
- If you're using the V2 API (`/authsettingsv2`), this would be in the `loginParameters` array.

You would need to remove any reference to "https://graph.windows.net", for example. This includes the `resource` parameter (which isn't supported by the "/v2.0" endpoint) or any scopes you're specifically requesting that are from the Azure AD Graph.

You would also need to update the configuration to request the new Microsoft Graph permissions you set up for the application registration. You can use the `.default scope` to simplify this setup in many cases. To do so, add a new sign-in parameter `scope=openid profile email https://graph.microsoft.com/.default`.

With these changes, when App Service Authentication attempts to sign in, it will no longer request permissions to the Azure AD Graph, and instead it will get a token for the Microsoft Graph. Any use of that token from your application code would also need to be updated, as per the [guidance provided by Microsoft Entra](#).

Next steps

- [App Service Authentication / Authorization overview](#).
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- [Tutorial: Authenticate and authorize users in a web app that accesses Azure Storage and Microsoft Graph](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Configure your App Service or Azure Functions app to use Facebook login

Article • 03/31/2021

This article shows how to configure Azure App Service or Azure Functions to use Facebook as an authentication provider.

To complete the procedure in this article, you need a Facebook account that has a verified email address and a mobile phone number. To create a new Facebook account, go to facebook.com.

Register your application with Facebook

1. Go to the [Facebook Developers](#) website and sign in with your Facebook account credentials.

If you don't have a Facebook for Developers account, select **Get Started** and follow the registration steps.

2. Select **My Apps > Add New App**.

3. In **Display Name** field:

- a. Type a unique name for your app.
- b. Provide your **Contact Email**.
- c. Select **Create App ID**.
- d. Complete the security check.

The developer dashboard for your new Facebook app opens.

4. Select **Dashboard > Facebook Login > Set up > Web**.

5. In the left navigation under **Facebook Login**, select **Settings**.

6. In the **Valid OAuth redirect URIs** field, enter `https://<app-name>.azurewebsites.net/.auth/login/facebook/callback`. Remember to replace `<app-name>` with the name of your Azure App Service app.

7. Select **Save Changes**.

8. In the left pane, select **Settings > Basic**.

9. In the App Secret field, select Show. Copy the values of App ID and App Secret. You use them later to configure your App Service app in Azure.

ⓘ Important

The app secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

10. The Facebook account that you used to register the application is an administrator of the app. At this point, only administrators can sign in to this application.

To authenticate other Facebook accounts, select **App Review** and enable **Make <your-app-name> public** to enable the general public to access the app by using Facebook authentication.

Add Facebook information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Click **Add identity provider**.
3. Select **Facebook** in the identity provider dropdown. Paste in the App ID and App Secret values that you obtained previously.

The secret will be stored as a slot-sticky [application setting](#) named `FACEBOOK_PROVIDER_AUTHENTICATION_SECRET`. You can update that setting later to use [Key Vault references](#) if you wish to manage the secret in Azure Key Vault.

4. If this is the first identity provider configured for the application, you will also be prompted with an **App Service authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests, and the default selections will redirect all requests to log in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. (Optional) Click **Next: Scopes** and add any scopes needed by the application. These will be requested at login time for browser-based flows.
6. Click **Add**.

You're now ready to use Facebook for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Next steps

- [App Service Authentication / Authorization overview.](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Configure your App Service or Azure Functions app to use GitHub login

Article • 03/01/2022

This article shows how to configure Azure App Service or Azure Functions to use GitHub as an authentication provider.

To complete the procedure in this article, you need a GitHub account. To create a new GitHub account, go to [GitHub](#).

Register your application with GitHub

1. Sign in to the [Azure portal](#) and go to your application. Copy your **URL**. You'll use it to configure your GitHub app.
2. Follow the instructions for [creating an OAuth app on GitHub](#). In the **Authorization callback URL** section, enter the HTTPS URL of your app and append the path `/.auth/login/github/callback`. For example,
`https://contoso.azurewebsites.net/.auth/login/github/callback`.
3. On the application page, make note of the **Client ID**, which you will need later.
4. Under **Client Secrets**, select **Generate a new client secret**.
5. Make note of the client secret value, which you will need later.

Important

The client secret is an important security credential. Do not share this secret with anyone or distribute it with your app.

Add GitHub information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Click **Add identity provider**.
3. Select **GitHub** in the identity provider dropdown. Paste in the **Client ID** and **Client secret** values that you obtained previously.

The secret will be stored as a slot-sticky [application setting](#) named `GITHUB_PROVIDER_AUTHENTICATION_SECRET`. You can update that setting later to use [Key Vault references](#) if you wish to manage the secret in Azure Key Vault.

4. If this is the first identity provider configured for the application, you will also be prompted with an **App Service authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests, and the default selections will redirect all requests to log in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. Click **Add**.

You're now ready to use GitHub for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Configure your App Service or Azure Functions app to use Google login

Article • 03/25/2024

This topic shows you how to configure Azure App Service or Azure Functions to use Google as an authentication provider.

To complete the procedure in this topic, you must have a Google account that has a verified email address. To create a new Google account, go to accounts.google.com.

Register your application with Google

1. Follow the Google documentation at [Sign In with Google for Web - Setup](#) to create a client ID and client secret. There's no need to make any code changes. Just use the following information:
 - For **Authorized JavaScript Origins**, use `https://<app-name>.azurewebsites.net` with the name of your app in `<app-name>`.
 - For **Authorized Redirect URI**, use `https://<app-name>.azurewebsites.net/.auth/login/google/callback`.
2. Copy the App ID and the App secret values.

Important

The App secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

Add Google information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Click **Add identity provider**.
3. Select **Google** in the identity provider dropdown. Paste in the App ID and App Secret values that you obtained previously.

The secret will be stored as a slot-sticky [application setting](#) named `GOOGLE_PROVIDER_AUTHENTICATION_SECRET`. You can update that setting later to use

[Key Vault references](#) if you wish to manage the secret in Azure Key Vault.

4. If this is the first identity provider configured for the application, you will also be prompted with an **App Service authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests, and the default selections will redirect all requests to log in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. Click **Add**.

 **Note**

For adding scope: You can define what permissions your application has in the provider's registration portal. The app can request scopes at login time which leverage these permissions.

You are now ready to use Google for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | Get help at Microsoft Q&A

Configure your App Service or Azure Functions app to use X login

Article • 08/09/2024

This article shows how to configure Azure App Service or Azure Functions to use X as an authentication provider.

To complete the procedure in this article, you need an X account that has a verified email address and phone number. To create a new X account, go to [x.com](#).

Register your application with X

1. Sign in to the [Azure portal](#) and go to your application. Copy your **URL**. You'll use it to configure your X app.
2. Go to the [X Developers](#) website, sign in with your X account credentials, and select **Create an app**.
3. Enter the **App name** and the **Application description** for your new app. Paste your application's **URL** into the **Website URL** field. In the **Callback URLs** section, enter the HTTPS URL of your App Service app and append the path `/auth/login/x/callback`. For example,
`https://contoso.azurewebsites.net/.auth/login/x/callback`.
4. At the bottom of the page, type at least 100 characters in **Tell us how this app will be used**, then select **Create**. Click **Create** again in the pop-up. The application details are displayed.
5. Select the **Keys and Access Tokens** tab.

Make a note of these values:

- API key
- API secret key

Important

The API secret key is an important security credential. Do not share this secret with anyone or distribute it with your app.

Add X information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Click **Add identity provider**.
3. Select **Twitter** in the identity provider dropdown. Paste in the `API key` and `API secret key` values that you obtained previously.

The secret will be stored as a slot-sticky [application setting](#) named `TWITTER_PROVIDER_AUTHENTICATION_SECRET`. You can update that setting later to use [Key Vault references](#) if you wish to manage the secret in Azure Key Vault.

4. If this is the first identity provider configured for the application, you will also be prompted with an **App Service authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests, and the default selections will redirect all requests to log in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. Click **Add**.

You're now ready to use X for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Feedback

Was this page helpful?

 Yes

 No

Configure your App Service or Azure Functions app to sign in using an OpenID Connect provider

Article • 09/04/2023

This article shows you how to configure Azure App Service or Azure Functions to use a custom authentication provider that adheres to the [OpenID Connect specification](#). OpenID Connect (OIDC) is an industry standard used by many identity providers (IDPs). You don't need to understand the details of the specification in order to configure your app to use an adherent IDP.

You can configure your app to use one or more OIDC providers. Each must be given a unique alphanumeric name in the configuration, and only one can serve as the default redirect target.

Register your application with the identity provider

Your provider will require you to register the details of your application with it. One of these steps involves specifying a redirect URI. This redirect URI will be of the form `<app-url>/auth/login/<provider-name>/callback`. Each identity provider should provide more instructions on how to complete these steps. `<provider-name>` will refer to the friendly name you give to the OpenID provider name in Azure.

Note

Some providers may require additional steps for their configuration and how to use the values they provide. For example, Apple provides a private key which is not itself used as the OIDC client secret, and you instead must use it to craft a JWT which is treated as the secret you provide in your app config (see the "Creating the Client Secret" section of the [Sign in with Apple documentation](#))

You'll need to collect a **client ID** and **client secret** for your application.

Important

The client secret is an important security credential. Don't share this secret with anyone or distribute it within a client application.

Additionally, you'll need the OpenID Connect metadata for the provider. This is often exposed via a [configuration metadata document](#), which is the provider's Issuer URL suffixed with `/well-known/openid-configuration`. Gather this configuration URL.

If you're unable to use a configuration metadata document, you'll need to gather the following values separately:

- The issuer URL (sometimes shown as `issuer`)
- The [OAuth 2.0 Authorization endpoint](#) (sometimes shown as `authorization_endpoint`)
- The [OAuth 2.0 Token endpoint](#) (sometimes shown as `token_endpoint`)
- The URL of the [OAuth 2.0 JSON Web Key Set](#) document (sometimes shown as `jwks_uri`)

Add provider information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **OpenID Connect** in the identity provider dropdown.
4. Provide the unique alphanumeric name selected earlier for **OpenID provider name**.
5. If you have the URL for the **metadata document** from the identity provider, provide that value for **Metadata URL**. Otherwise, select the **Provide endpoints separately** option and put each URL gathered from the identity provider in the appropriate field.
6. Provide the earlier collected **Client ID** and **Client Secret** in the appropriate fields.
7. Specify an application setting name for your client secret. Your client secret will be stored as an app setting to ensure secrets are stored in a secure fashion. You can update that setting later to use [Key Vault references](#) if you wish to manage the secret in Azure Key Vault.
8. Press the **Add** button to finish setting up the identity provider.

Note

The OpenID provider name can't contain symbols like `-` because an appsetting will be created based on this and it doesn't support it. Use `_` instead.

 **Note**

Azure requires "openid," "profile," and "email" scopes. Make sure you've configured your App Registration in your ID Provider with at least these scopes.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Configure your App Service or Azure Functions app to sign in using a Sign in with Apple provider (Preview)

Article • 09/23/2021

This article shows you how to configure Azure App Service or Azure Functions to use Sign in with Apple as an authentication provider.

To complete the procedure in this article, you must have enrolled in the Apple developer program. To enroll in the Apple developer program, go to developer.apple.com/programs/enroll.

✖ Caution

Enabling Sign in with Apple will disable management of the App Service Authentication / Authorization feature for your application through some clients, such as the Azure portal, Azure CLI, and Azure PowerShell. The feature relies on a new API surface which, during preview, is not yet accounted for in all management experiences.

Create an application in the Apple Developer portal

You'll need to create an App ID and a service ID in the Apple Developer portal.

1. On the Apple Developer portal, go to **Certificates, Identifiers, & Profiles**.
2. On the **Identifiers** tab, select the (+) button.
3. On the **Register a New Identifier** page, choose **App IDs** and select **Continue**. (App IDs include one or more Service IDs.)

Register a New Identifier

[Continue](#)

App IDs
Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.

Services IDs
For each website that uses Sign In with Apple, register a services identifier (Services ID), configure your domain and return URL, and create an associated private key.

4. On the **Register an App ID** page, provide a description and a bundle ID, and select **Sign in with Apple** from the capabilities list. Then select **Continue**. Take note of

your App ID Prefix (Team ID) from this step, you'll need it later.

Register an App ID

[Back](#) [Continue](#)

Platform	App ID Prefix
<input checked="" type="radio"/> iOS, tvOS, watchOS	E88K2FF6LU (Team ID)
Description	Bundle ID
easy auth test sign in with apple	<input checked="" type="radio"/> Explicit
You cannot use special characters such as @, &, *, ;, "	<input type="radio"/> Wildcard
com.microsoft.easyauthtest.client	
We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).	

Capabilities

ENABLED	NAME
<input type="checkbox"/>	Access WiFi Information

5. Review the app registration information and select **Register**.

6. Again, on the Identifiers tab, select the (+) button.

Certificates, Identifiers & Profiles

Certificates	Identifiers +
Identifiers	NAME IDENTIFIER
Devices	NE [REDACTED] com.mi[REDACTED]
Profiles	Xcode iOS Wildcard App ID *
Keys	
More	

7. On the Register a New Identifier page, choose Services IDs and select Continue.

Register a New Identifier

[Continue](#)

- App IDs
Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.
- Services IDs
For each website that uses Sign In with Apple, register a services identifier (Services ID), configure your domain and return URL, and create an associated private key.

8. On the Register a Services ID page, provide a description and an identifier. The description is what will be shown to the user on the consent screen. The identifier will be your client ID used in configuring the Apple provider with your app service.

Then select **Configure**.

Register a Services ID

[Back](#) [Continue](#)

Description	Identifier
easy auth test sign in with apple	com.microsoft.easyauthtest.client
You cannot use special characters such as @, &, *, ;, "	We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).
ENABLED	NAME
<input checked="" type="checkbox"/>	Sign In with Apple
Configure	

9. On the pop-up window, set the Primary App ID to the App ID you created earlier. Specify your application's domain in the domain section. For the return URL, use the URL `<app-url>/auth/login/apple/callback`. For example,

`https://contoso.azurewebsites.net/.auth/login/apple/callback`. Then select Add

and Save.

The screenshot shows the 'Web Authentication Configuration' page. At the top, there's a brief description: 'Use Sign In with Apple to let your users sign in to your app's accompanying website with their Apple ID. To configure web authentication, group your website with the existing primary App ID that's enabled for Sign In with Apple.' Below this, a 'Primary App ID' dropdown is set to 'Easy Auth test ID (SGGM6D27TK.com.micros)'. The 'Domains' section contains a 'Web Domain' input field with 'easyauth.net' and an 'Add' button next to a 'Return URLs' input field containing 'https://easyauth.net/.auth/login/apple/callback'. At the bottom right are 'Cancel' and 'Save' buttons, with 'Save' being highlighted by an orange box.

10. Review the service registration information and select **Save**.

Generate the client secret

Apple requires app developers to create and sign a JWT token as the client secret value. To generate this secret, first generate and download an elliptic curve private key from the Apple Developer portal. Then, use that key to [sign a JWT](#) with a [specific payload](#).

Create and download the private key

1. On the **Keys** tab in the Apple Developer portal, choose **Create a key** or select the (+) button.
2. On the **Register a New Key** page give the key a name, check the box next to **Sign in with Apple** and select **Configure**.
3. On the **Configure Key** page, link the key to the primary app ID you created previously and select **Save**.
4. Finish creating the key by confirming the information and selecting **Continue** and then reviewing the information and selecting **Register**.

5. On the Download Your Key page, download the key. It will download as a .p8 (PKCS#8) file - you'll use the file contents to sign your client secret JWT.

Structure the client secret JWT

Apple requires the client secret be the base64-encoding of a JWT token. The decoded JWT token should have a payload structured like this example:

```
JSON

{
  "alg": "ES256",
  "kid": "URKEYID001",
}.{
  "sub": "com.yourcompany.app1",
  "nbf": 1560203207,
  "exp": 1560289607,
  "iss": "ABC123DEFG",
  "aud": "https://appleid.apple.com"
}.[Signature]
```

- **sub**: The Apple Client ID (also the service ID)
- **iss**: Your Apple Developer Team ID
- **aud**: Apple is receiving the token, so they're the audience
- **exp**: No more than six months after **nbf**

The base64-encoded version of the above payload looks like this:

```
eyJhbGciOiJFUzI1NiIsImtpZCI6IlVSS0VZSUQwMDEifQ.eyJzdWIiOiJjb20ueW91cmNvbXBhbnkuYXBw
MSIiIm5iZiI6MTU2MDIwMzIwNywiZXhwIjoxNTYwMjg5Nja3LCJpc3MiOiJBQkMxMjNERUZHIIiwiYXVkJ
iaHR0cHM6Ly9hcHBsZWlkLmFwcGx1LmNvbSJ9.ABSXELWuTbgqfrIUz7bLi6nXvkXAz508vt0jb2dSHTQTi
b1x1DSP4_4Ur1KI-pdzNg1sgeocolPNTmDKaz08-
BHAZCsdeeTNlgFEzBytIpMKFFVEQbEtGRkam5Iec1UK7S9oOva4EK4jV4VmDrr-
LGWW03TaAxAvy3_ZoKohvFFkVG
```

Note: Apple doesn't accept client secret JWTs with an expiration date more than six months after the creation (or nbf) date. That means you'll need to rotate your client secret, at minimum, every six months.

More information about generating and validating tokens can be found in [Apple's developer documentation ↗](#).

Sign the client secret JWT

You'll use the `.p8` file you downloaded previously to sign the client secret JWT. This file is a [PCKS#8 file](#) that contains the private signing key in PEM format. There are many libraries that can create and sign the JWT for you.

There are different kinds of open-source libraries available online for creating and signing JWT tokens. For more information about generating JWT tokens, see [JSON Web Token \(JWT\)](#). For example, one way of generating the client secret is by importing the [Microsoft.IdentityModel.Tokens NuGet package](#) and running a small amount of C# code shown below.

C#

```
using Microsoft.IdentityModel.Tokens;

public static string GetAppleClientSecret(string teamId, string clientId,
string keyId, string p8key)
{
    string audience = "https://appleid.apple.com";

    string issuer = teamId;
    string subject = clientId;
    string kid = keyId;

    IList<Claim> claims = new List<Claim> {
        new Claim ("sub", subject)
    };

    CngKey cngKey = CngKey.Import(Convert.FromBase64String(p8key),
CngKeyBlobFormat.Pkcs8PrivateBlob);

    SigningCredentials signingCred = new SigningCredentials(
        new ECDsaSecurityKey(new ECDsaCng(cngKey)),
        SecurityAlgorithms.EcdsaSha256
    );

    JwtSecurityToken token = new JwtSecurityToken(
        issuer,
        audience,
        claims,
        DateTime.Now,
        DateTime.Now.AddDays(180),
        signingCred
    );
    token.Header.Add("kid", kid);
    token.Header.Remove("typ");

    JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();

    return tokenHandler.WriteToken(token);
}
```

- **teamId**: Your Apple Developer Team ID
- **clientId**: The Apple Client ID (also the service ID)
- **p8key**: The PEM format key - you can obtain the key by opening the `.p8` file in a text editor, and copying everything between `-----BEGIN PRIVATE KEY-----` and `-----END PRIVATE KEY-----` without line breaks
- **keyId**: The ID of the downloaded key

This token returned is the client secret value you'll use to configure the Apple provider.

Important

The client secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

Add the client secret as an [application setting](#) for the app, using a setting name of your choice. Make note of this name for later.

Add provider information to your application

Note

The required configuration is in a new API format, currently only supported by [file-based configuration \(preview\)](#). You will need to follow the below steps using such a file.

This section will walk you through updating the configuration to include your new IDP. An example configuration follows.

1. Within the `identityProviders` object, add an `apple` object if one doesn't already exist.
2. Assign an object to that key with a `registration` object within it, and optionally a `login` object:

JSON

```
"apple" : {  
    "registration" : {  
        "clientId": "<client ID>",  
        "clientSecretSettingName":  
        "APP_SETTING_CONTAINING_APPLE_CLIENT_SECRET"  
    },
```

```
    "login": {
        "scopes": []
    }
}
```

- a. Within the `registration` object, set the `clientId` to the client ID you collected.
- b. Within the `registration` object, set `clientSecretSettingName` to the name of the application setting where you stored the client secret.
- c. Within the `login` object, you may choose to set the `scopes` array to include a list of scopes used when authenticating with Apple, such as "name" and "email". If scopes are configured, they'll be explicitly requested on the consent screen when users sign in for the first time.

Once this configuration has been set, you're ready to use your Apple provider for authentication in your app.

A complete configuration might look like the following example (where the `APPLE_GENERATED_CLIENT_SECRET` setting points to an application setting containing a generated JWT):

JSON

```
{
    "platform": {
        "enabled": true
    },
    "globalValidation": {
        "redirectToProvider": "apple",
        "unauthenticatedClientAction": "RedirectLoginPage"
    },
    "identityProviders": {
        "apple": {
            "registration": {
                "clientId": "com.contoso.example.client",
                "clientSecretSettingName": "APPLE_GENERATED_CLIENT_SECRET"
            },
            "login": {
                "scopes": []
            }
        }
    },
    "login": {
        "tokenStore": {
            "enabled": true
        }
    }
}
```

Next steps

- [App Service Authentication / Authorization overview.](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Customize sign-in and sign-out in Azure App Service authentication

Article • 07/08/2024

This article shows you how to customize user sign-ins and sign-outs while using the built-in [authentication and authorization in App Service](#).

Use multiple sign-in providers

The portal configuration doesn't offer a turn-key way to present multiple sign-in providers to your users (such as both Facebook and Twitter). However, it isn't difficult to add the functionality to your app. The steps are outlined as follows:

First, in the **Authentication / Authorization** page in the Azure portal, configure each of the identity provider you want to enable.

In **Action to take when request is not authenticated**, select **Allow Anonymous requests (no action)**.

In the sign-in page, or the navigation bar, or any other location of your app, add a sign-in link to each of the providers you enabled (`/auth/login/<provider>`). For example:

HTML

```
<a href="/.auth/login/aad">Log in with Microsoft Entra</a>
<a href="/.auth/login/facebook">Log in with Facebook</a>
<a href="/.auth/login/google">Log in with Google</a>
<a href="/.auth/login/twitter">Log in with Twitter</a>
<a href="/.auth/login/apple">Log in with Apple</a>
```

When the user clicks on one of the links, the respective sign-in page opens to sign in the user.

To redirect the user post-sign-in to a custom URL, use the `post_login_redirect_uri` query string parameter (not to be confused with the Redirect URI in your identity provider configuration). For example, to navigate the user to `/Home/Index` after sign-in, use the following HTML code:

HTML

```
<a href="/.auth/login/<provider>?post_login_redirect_uri=/Home/Index">Log
```

in

Client-directed sign-in

In a client-directed sign-in, the application signs in the user to the identity provider using a provider-specific SDK. The application code then submits the resulting authentication token to App Service for validation (see [Authentication flow](#)) using an HTTP POST request. This validation itself doesn't actually grant you access to the desired app resources, but a successful validation will give you a session token that you can use to access app resources.

To validate the provider token, App Service app must first be configured with the desired provider. At runtime, after you retrieve the authentication token from your provider, post the token to `/.auth/login/<provider>` for validation. For example:

```
POST https://<appname>.azurewebsites.net/.auth/login/aad HTTP/1.1
Content-Type: application/json
```

```
{"id_token": "<token>", "access_token": "<token>"}
```

The token format varies slightly according to the provider. See the following table for details:

[\[+\] Expand table](#)

Provider value	Required in request body	Comments
aad	<pre>{"access_token": "<access_token>"}</pre>	The <code>id_token</code> , <code>refresh_token</code> , and <code>expires_in</code> properties are optional.
microsoftaccount	<pre>{"access_token": "<access_token>"}</pre> or <pre>{"authentication_token": "<token>"}</pre>	<code>authentication_token</code> is preferred over <code>access_token</code> . The <code>expires_in</code> property is optional. When requesting the token from Live services, always request the <code>wl.basic</code> scope.
google	<pre>{"id_token": "<id_token>"}</pre>	The <code>authorization_code</code> property is optional. Providing an <code>authorization_code</code> value will add an access token and a refresh token to the token store. When specified,

Provider value	Required in request body	Comments
		<code>authorization_code</code> can also optionally be accompanied by a <code>redirect_uri</code> property.
facebook	{"access_token": " <user_access_token>"}	Use a valid user access token from Facebook.
twitter	{"access_token": " <access_token>," "access_token_secret": " <access_token_secret>"}	

ⓘ Note

The GitHub provider for App Service authentication does not support customized sign-in and sign-out.

If the provider token is validated successfully, the API returns with an `authenticationToken` in the response body, which is your session token.

JSON

```
{
  "authenticationToken": "...",
  "user": {
    "userId": "sid:..."
  }
}
```

Once you have this session token, you can access protected app resources by adding the `X-ZUMO-AUTH` header to your HTTP requests. For example:

```
GET https://<appname>.azurewebsites.net/api/products/1
X-ZUMO-AUTH: <authenticationToken_value>
```

Sign out of a session

Users can initiate a sign-out by sending a `GET` request to the app's `/auth/logout` endpoint. The `GET` request does the following:

- Clears authentication cookies from the current session.
- Deletes the current user's tokens from the token store.
- For Microsoft Entra and Google, performs a server-side sign-out on the identity provider.

Here's a simple sign-out link in a webpage:

HTML

```
<a href="/.auth/logout">Sign out</a>
```

By default, a successful sign-out redirects the client to the URL `/auth/logout/complete`.

You can change the post-sign-out redirect page by adding the `post_logout_redirect_uri` query parameter. For example:

```
GET /.auth/logout?post_logout_redirect_uri=/index.html
```

It's recommended that you [encode ↗](#) the value of `post_logout_redirect_uri`.

When using fully qualified URLs, the URL must be either hosted in the same domain or configured as an allowed external redirect URL for your app. In the following example, to redirect to `https://myexternalurl.com` that's not hosted in the same domain:

```
GET /.auth/logout?post_logout_redirect_uri=https%3A%2F%2Fmyexternalurl.com
```

Run the following command in the [Azure Cloud Shell](#):

Azure CLI

```
az webapp auth update --name <app_name> --resource-group <group_name> --  
allowed-external-redirect-urls "https://myexternalurl.com"
```

Preserve URL fragments

After users sign in to your app, they usually want to be redirected to the same section of the same page, such as `/wiki/Main_Page#SectionZ`. However, because [URL fragments ↗](#) (for example, `#SectionZ`) are never sent to the server, they are not preserved by default after the OAuth sign-in completes and redirects back to your app. Users then get a

suboptimal experience when they need to navigate to the desired anchor again. This limitation applies to all server-side authentication solutions.

In App Service authentication, you can preserve URL fragments across the OAuth sign-in. To do this, set an app setting called `WEBSITE_AUTH_PRESERVE_URL_FRAGMENT` to `true`. You can do it in the [Azure portal](#), or simply run the following command in the [Azure Cloud Shell](#):

Azure CLI

```
az webapp config appsettings set --name <app_name> --resource-group <group_name> --settings WEBSITE_AUTH_PRESERVE_URL_FRAGMENT="true"
```

Setting the sign-in accounts domain hint

Both Microsoft Account and Microsoft Entra lets you sign in from multiple domains. For example, Microsoft Account allows `outlook.com`, `live.com`, and `hotmail.com` accounts. Microsoft Entra allows any number of custom domains for the sign-in accounts. However, you may want to accelerate your users straight to your own branded Microsoft Entra sign-in page (such as `contoso.com`). To suggest the domain name of the sign-in accounts, follow these steps.

1. In <https://resources.azure.com>, At the top of the page, select **Read/Write**.
2. In the left browser, navigate to **subscriptions** > `<subscription-name>` > **resourceGroups** > `<resource-group-name>` > **providers** > **Microsoft.Web** > **sites** > `<app-name>` > **config** > **authsettingsV2**.
3. Click **Edit**.
4. Add a `loginParameters` array with a `domain_hint` item.

JSON

```
"identityProviders": {  
    "azureActiveDirectory": {  
        "login": {  
            "loginParameters": ["domain_hint=<domain-name>"],  
        }  
    }  
}
```

5. Click **Put**.

This setting appends the `domain_hint` query string parameter to the login redirect URL.

ⓘ Important

It's possible for the client to remove the `domain_hint` parameter after receiving the redirect URL, and then login with a different domain. So while this function is convenient, it's not a security feature.

Authorize or deny users

While App Service takes care of the simplest authorization case (i.e. reject unauthenticated requests), your app may require more fine-grained authorization behavior, such as limiting access to only a specific group of users. In certain cases, you need to write custom application code to allow or deny access to the signed-in user. In other cases, App Service or your identity provider may be able to help without requiring code changes.

- [Server level](#)
- [Identity provider level](#)
- [Application level](#)

Server level (Windows apps only)

For any Windows app, you can define authorization behavior of the IIS web server, by editing the `Web.config` file. Linux apps don't use IIS and can't be configured through `Web.config`.

1. Navigate to `https://<app-name>.scm.azurewebsites.net/DebugConsole`
2. In the browser explorer of your App Service files, navigate to `site/wwwroot`. If a `Web.config` doesn't exist, create it by selecting `+ > New File`.
3. Select the pencil for `Web.config` to edit it. Add the following configuration code and click **Save**. If `Web.config` already exists, just add the `<authorization>` element with everything in it. Add the accounts you want to allow in the `<allow>` element.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
```

```
<allow users="user1@contoso.com,user2@contoso.com"/>
<deny users="*"/>
</authorization>
</system.web>
</configuration>
```

Identity provider level

The identity provider may provide certain turn-key authorization. For example:

- You can [manage enterprise-level access](#) directly in Microsoft Entra. For instructions, see [How to remove a user's access to an application](#).
- For [Google](#), Google API projects that belong to an [organization](#) can be configured to allow access only to users in your organization (see [Google's Setting up OAuth 2.0 support page](#)).

Application level

If either of the other levels don't provide the authorization you need, or if your platform or identity provider isn't supported, you must write custom code to authorize users based on the [user claims](#).

More resources

- [Tutorial: Authenticate and authorize users end-to-end](#)
- [Environment variables and app settings for authentication](#)

Feedback

Was this page helpful?



[Provide product feedback](#)

Work with user identities in Azure App Service authentication

Article • 07/22/2024

This article shows you how to work with user identities when using the built-in [authentication and authorization in App Service](#).

Access user claims in app code

For all language frameworks, App Service makes the claims in the incoming token (whether from an authenticated end user or a client application) available to your code by injecting them into the request headers. External requests aren't allowed to set these headers, so they're present only if set by App Service. Some example headers include:

[+] [Expand table](#)

Header	Description
X-MS-CLIENT-PRINCIPAL	A Base64 encoded JSON representation of available claims. For more information, see Decoding the client principal header .
X-MS-CLIENT-PRINCIPAL-ID	An identifier for the caller set by the identity provider.
X-MS-CLIENT-PRINCIPAL-NAME	A human-readable name for the caller set by the identity provider, such as email address or user principal name.
X-MS-CLIENT-PRINCIPAL-IDP	The name of the identity provider used by App Service Authentication.

Provider tokens are also exposed through similar headers. For example, Microsoft Entra also sets `X-MS-TOKEN-AAD-ACCESS-TOKEN` and `X-MS-TOKEN-AAD-ID-TOKEN` as appropriate.

Note

Different language frameworks might present these headers to the app code in different formats, such as lowercase or title case.

Code that is written in any language or framework can get the information that it needs from these headers. [Decoding the client principal header](#) covers this process. For some frameworks, the platform also provides extra options that might be more convenient.

Decoding the client principal header

`X-MS-CLIENT-PRINCIPAL` contains the full set of available claims as Base64 encoded JSON. These claims go through a default claims-mapping process, so some might have different names than you would see if processing the token directly. The decoded payload is structured as follows:

JSON
{ "auth_typ": "", "claims": [{ "typ": "", "val": "" }], "name_typ": "", "role_typ": "" }

[+] [Expand table](#)

Property	Type	Description
<code>auth_typ</code>	string	The name of the identity provider used by App Service Authentication.
<code>claims</code>	array of objects	An array of objects representing the available claims. Each object contains <code>typ</code> and <code>val</code> properties.
<code>typ</code>	string	The name of the claim. It might be subject to default claims mapping and could be different from the corresponding claim contained in a token.
<code>val</code>	string	The value of the claim.
<code>name_typ</code>	string	The name claim type, which is typically a URI providing scheme information about the <code>name</code> claim if one is defined.
<code>role_typ</code>	string	The role claim type, which is typically a URI providing scheme information about the <code>role</code> claim if one is defined.

To process this header, your app needs to decode the payload and iterate through the `claims` array to find the claims of interest. It might be convenient to convert them into a representation used by the app's language framework. Here's an example of this process in C# that constructs a `ClaimsPrincipal` type for the app to use:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Text;
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Http;

public static class ClaimsPrincipalParser
{
    private class ClientPrincipalClaim
    {
        [JsonPropertyName("typ")]
        public string Type { get; set; }
        [JsonPropertyName("val")]
        public string Value { get; set; }
    }

    private class ClientPrincipal
    {
        [JsonPropertyName("auth_typ")]
        public string IdentityProvider { get; set; }
        [JsonPropertyName("name_typ")]
        public string NameClaimType { get; set; }
        [JsonPropertyName("role_typ")]
        public string RoleClaimType { get; set; }
        [JsonPropertyName("claims")]
        public IEnumerable<ClientPrincipalClaim> Claims { get; set; }
    }

    public static ClaimsPrincipal Parse(HttpContext req)
    {
        var principal = new ClientPrincipal();

        if (req.Headers.TryGetValue("x-ms-client-principal", out var header))
        {
            var data = header[0];
            var decoded = Convert.FromBase64String(data);
            var json = Encoding.UTF8.GetString(decoded);
            principal = JsonSerializer.Deserialize<ClientPrincipal>(json,
new JsonSerializerOptions { PropertyNameCaseInsensitive = true });
        }

        /**
         * At this point, the code can iterate through `principal.Cclaims` to
         * check claims as part of validation. Alternatively, we can
         * convert
         * it into a standard object with which to perform those checks
         * later
    }
}
```

```

        * in the request pipeline. That object can also be leveraged for
        * associating user data, etc. The rest of this function performs
such
        * a conversion to create a `ClaimsPrincipal` as might be used in
        * other .NET code.
    */

    var identity = new ClaimsIdentity(principal.IdentityProvider,
principal.NameClaimType, principal.RoleClaimType);
    identity.AddClaims(principal.Claims.Select(c => new Claim(c.Type,
c.Value)));

    return new ClaimsPrincipal(identity);
}
}

```

Framework-specific alternatives

For ASP.NET 4.6 apps, App Service populates `ClaimsPrincipal.Current` with the authenticated user's claims, so you can follow the standard .NET code pattern, including the `[Authorize]` attribute. Similarly, for PHP apps, App Service populates the `_SERVER['REMOTE_USER']` variable. For Java apps, the claims are [accessible from the Tomcat servlet](#).

For [Azure Functions](#), `ClaimsPrincipal.Current` isn't populated for .NET code, but you can still find the user claims in the request headers, or get the `ClaimsPrincipal` object from the request context or even through a binding parameter. For more information, see [Working with client identities in Azure Functions](#).

For .NET Core, [Microsoft.Identity.Web](#) supports populating the current user with App Service authentication. To learn more, you can read about it on the [Microsoft.Identity.Web wiki](#), or see it demonstrated in [this tutorial for a web app accessing Microsoft Graph](#).

Note

For claims mapping to work, you must enable the [Token store](#).

Access user claims using the API

If the [token store](#) is enabled for your app, you can also obtain other details on the authenticated user by calling `/.auth/me`.

Next steps

[Tutorial: Authenticate and authorize users end-to-end](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Work with OAuth tokens in Azure App Service authentication

Article • 12/02/2022

This article shows you how to work with OAuth tokens while using the built-in [authentication and authorization in App Service](#).

Retrieve tokens in app code

From your server code, the provider-specific tokens are injected into the request header, so you can easily access them. The following table shows possible token header names:

Provider	Header names
Azure Active Directory	X-MS-TOKEN-AAD-ID-TOKEN X-MS-TOKEN-AAD-ACCESS-TOKEN X-MS-TOKEN-AAD-EXPIRES-ON X-MS-TOKEN-AAD-REFRESH-TOKEN
Facebook Token	X-MS-TOKEN-FACEBOOK-ACCESS-TOKEN X-MS-TOKEN-FACEBOOK-EXPIRES-ON
Google	X-MS-TOKEN-GOOGLE-ID-TOKEN X-MS-TOKEN-GOOGLE-ACCESS-TOKEN X-MS-TOKEN-GOOGLE-EXPIRES-ON X-MS-TOKEN-GOOGLE-REFRESH-TOKEN
Twitter	X-MS-TOKEN-TWITTER-ACCESS-TOKEN X-MS-TOKEN-TWITTER-ACCESS-TOKEN-SECRET

ⓘ Note

Different language frameworks may present these headers to the app code in different formats, such as lowercase or title case.

From your client code (such as a mobile app or in-browser JavaScript), send an HTTP `GET` request to `/auth/me` ([token store](#) must be enabled). The returned JSON has the provider-specific tokens.

ⓘ Note

Access tokens are for accessing provider resources, so they are present only if you configure your provider with a client secret. To see how to get refresh tokens, see Refresh access tokens.

Refresh auth tokens

When your provider's access token (not the [session token](#)) expires, you need to reauthenticate the user before you use that token again. You can avoid token expiration by making a `GET` call to the `/auth/refresh` endpoint of your application. When called, App Service automatically refreshes the access tokens in the [token store](#) for the authenticated user. Subsequent requests for tokens by your app code get the refreshed tokens. However, for token refresh to work, the token store must contain [refresh tokens](#) for your provider. The way to get refresh tokens are documented by each provider, but the following list is a brief summary:

- **Google:** Append an `access_type=offline` query string parameter to your `/auth/login/google` API call. For more information, see [Google Refresh Tokens](#).
- **Facebook:** Doesn't provide refresh tokens. Long-lived tokens expire in 60 days (see [Facebook Expiration and Extension of Access Tokens](#)).
- **Twitter:** Access tokens don't expire (see [Twitter OAuth FAQ](#)).
- **Microsoft:** In <https://resources.azure.com>, do the following steps:
 1. At the top of the page, select **Read/Write**.
 2. In the left browser, navigate to `subscriptions > <subscription_name> > resourceGroups > <resource_group_name> > providers > Microsoft.Web > sites > <app_name> > config > authsettingsV2`.
 3. Click **Edit**.
 4. Modify the following property.

JSON

```
"identityProviders": {  
    "azureActiveDirectory": {  
        "login": {  
            "loginParameters": ["scope=openid profile email  
offline_access"]  
        }  
    }  
}
```

5. Click Put.

ⓘ Note

The scope that gives you a refresh token is `offline_access`. See how it's used in [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#). The other scopes are requested by default by App Service already. For information on these default scopes, see [OpenID Connect Scopes](#).

Once your provider is configured, you can [find the refresh token and the expiration time for the access token](#) in the token store.

To refresh your access token at any time, just call `/.auth/refresh` in any language. The following snippet uses jQuery to refresh your access tokens from a JavaScript client.

JavaScript

```
function refreshTokens() {
  let refreshUrl = "/.auth/refresh";
  $.ajax(refreshUrl) .done(function() {
    console.log("Token refresh completed successfully.");
  }) .fail(function() {
    console.log("Token refresh failed. See application logs for details.");
  });
}
```

If a user revokes the permissions granted to your app, your call to `/.auth/me` may fail with a `403 Forbidden` response. To diagnose errors, check your application logs for details.

Extend session token expiration grace period

The authenticated session expires after 8 hours. After an authenticated session expires, there is a 72-hour grace period by default. Within this grace period, you're allowed to refresh the session token with App Service without reauthenticating the user. You can just call `/.auth/refresh` when your session token becomes invalid, and you don't need to track token expiration yourself. Once the 72-hour grace period is lapses, the user must sign in again to get a valid session token.

If 72 hours isn't enough time for you, you can extend this expiration window. Extending the expiration over a long period could have significant security implications (such as

when an authentication token is leaked or stolen). So you should leave it at the default 72 hours or set the extension period to the smallest value.

To extend the default expiration window, run the following command in the [Cloud Shell](#).

Azure CLI

```
az webapp auth update --resource-group <group_name> --name <app_name> --  
token-refresh-extension-hours <hours>
```

ⓘ Note

The grace period only applies to the App Service authenticated session, not the tokens from the identity providers. There is no grace period for the expired provider tokens.

Next steps

[Tutorial: Authenticate and authorize users end-to-end](#)

Manage the API and runtime versions of App Service authentication

Article • 10/12/2023

This article shows you how to customize the API and runtime versions of the built-in [authentication and authorization in App Service](#).

There are two versions of the management API for App Service authentication. The V2 version is required for the "Authentication" experience in the Azure portal. An app already using the V1 API can upgrade to the V2 version once a few changes have been made. Specifically, secret configuration must be moved to slot-sticky application settings. This can be done automatically from the "Authentication" section of the portal for your app.

Update the configuration version

Warning

Migration to V2 will disable management of the App Service Authentication/Authorization feature for your application through some clients, such as its existing experience in the Azure portal, Azure CLI, and Azure PowerShell. This cannot be reversed.

The V2 API doesn't support creation or editing of Microsoft Account as a distinct provider as was done in V1. Rather, it uses the converged [Microsoft identity platform](#) to sign-in users with both Microsoft Entra ID and personal Microsoft accounts. When switching to the V2 API, the V1 Microsoft Entra configuration is used to configure the Microsoft identity platform provider. The V1 Microsoft Account provider will be carried forward in the migration process and continue to operate as normal, but you should move to the newer Microsoft identity platform model. See [Support for Microsoft Account provider registrations](#) to learn more.

The automated migration process will move provider secrets into application settings and then convert the rest of the configuration into the new format. To use the automatic migration:

1. Navigate to your app in the portal and select the **Authentication** menu option.
2. If the app is configured using the V1 model, you'll see an **Upgrade** button.

3. Review the description in the confirmation prompt. If you're ready to perform the migration, select **Upgrade** in the prompt.

Manually managing the migration

The following steps will allow you to manually migrate the application to the V2 API if you don't wish to use the automatic version mentioned above.

Moving secrets to application settings

1. Get your existing configuration by using the V1 API:

Azure CLI

```
az webapp auth show -g <group_name> -n <site_name>
```

In the resulting JSON payload, make note of the secret value used for each provider you've configured:

- Microsoft Entra ID: `clientSecret`
- Google: `googleClientSecret`
- Facebook: `facebookAppSecret`
- Twitter: `twitterConsumerSecret`
- Microsoft Account: `microsoftAccountClientSecret`

 **Important**

The secret values are important security credentials and should be handled carefully. Do not share these values or persist them on a local machine.

2. Create slot-sticky application settings for each secret value. You may choose the name of each application setting. Its value should match what you obtained in the previous step or [reference a Key Vault secret](#) that you've created with that value.

To create the setting, you can use the Azure portal or run a variation of the following for each provider:

Azure CLI

```
# For Web Apps, Google example
az webapp config appsettings set -g <group_name> -n <site_name> --slot-
settings GOOGLE_PROVIDER_AUTHENTICATION_SECRET=
```

```
<value_from_previous_step>

# For Azure Functions, Twitter example
az functionapp config appsettings set -g <group_name> -n <site_name> --
slot-settings TWITTER_PROVIDER_AUTHENTICATION_SECRET=
<value_from_previous_step>
```

ⓘ Note

The application settings for this configuration should be marked as slot-sticky, meaning that they will not move between environments during a **slot swap** operation. This is because your authentication configuration itself is tied to the environment.

3. Create a new JSON file named `authsettings.json`. Take the output that you received previously and remove each secret value from it. Write the remaining output to the file, making sure that no secret is included. In some cases, the configuration may have arrays containing empty strings. Make sure that `microsoftAccountOAuthScopes` doesn't, and if it does, switch that value to `null`.
4. Add a property to `authsettings.json` that points to the application setting name you created earlier for each provider:

- Microsoft Entra ID: `clientSecretSettingName`
- Google: `googleClientSecretSettingName`
- Facebook: `facebookAppSecretSettingName`
- Twitter: `twitterConsumerSecretSettingName`
- Microsoft Account: `microsoftAccountClientSecretSettingName`

An example file after this operation might look similar to the following, in this case only configured for Microsoft Entra ID:

JSON

```
{
  "id": "/subscriptions/00d563f8-5b89-4c6a-bcec-
c1b9f6d607e0/resourceGroups/myresourcegroup/providers/Microsoft.Web/sites/mywebapp/config/authsettings",
  "name": "authsettings",
  "type": "Microsoft.Web/sites/config",
  "location": "Central US",
  "properties": {
    "enabled": true,
    "runtimeVersion": "~1",
    "unauthenticatedClientAction": "AllowAnonymous",
```

```

        "tokenStoreEnabled": true,
        "allowedExternalRedirectUrls": null,
        "defaultProvider": "AzureActiveDirectory",
        "clientId": "3197c8ed-2470-480a-8fae-58c25558ac9b",
        "clientSecret": "",
        "clientSecretSettingName":
        "MICROSOFT_IDENTITY_AUTHENTICATION_SECRET",
        "clientSecretCertificateThumbprint": null,
        "issuer": "https://sts.windows.net/0b2ef922-672a-4707-9643-
9a5726eec524/",
        "allowedAudiences": [
            "https://mywebapp.azurewebsites.net"
        ],
        "additionalLoginParams": null,
        "isAadAutoProvisioned": true,
        "aadClaimsAuthorization": null,
        "googleClientId": null,
        "googleClientSecret": null,
        "googleClientSecretSettingName": null,
        "googleOAuthScopes": null,
        "facebookAppId": null,
        "facebookAppSecret": null,
        "facebookAppSecretSettingName": null,
        "facebookOAuthScopes": null,
        "gitHubClientId": null,
        "gitHubClientSecret": null,
        "gitHubClientSecretSettingName": null,
        "gitHubOAuthScopes": null,
        "twitterConsumerKey": null,
        "twitterConsumerSecret": null,
        "twitterConsumerSecretSettingName": null,
        "microsoftAccountClientId": null,
        "microsoftAccountClientSecret": null,
        "microsoftAccountClientSecretSettingName": null,
        "microsoftAccountOAuthScopes": null,
        "isAuthFromFile": "false"
    }
}

```

5. Submit this file as the new Authentication/Authorization configuration for your app:

Azure CLI

```

az rest --method PUT --url
"/subscriptions/<subscription_id>/resourceGroups/<group_name>/providers
/Microsoft.Web/sites/<site_name>/config/authsettings?api-version=2020-
06-01" --body @./authsettings.json

```

6. Validate that your app is still operating as expected after this gesture.

7. Delete the file used in the previous steps.

You've now migrated the app to store identity provider secrets as application settings.

Support for Microsoft Account provider registrations

If your existing configuration contains a Microsoft Account provider and doesn't contain a Microsoft Entra provider, you can switch the configuration over to the Microsoft Entra provider and then perform the migration. To do this:

1. Go to [App registrations](#) in the Azure portal and find the registration associated with your Microsoft Account provider. It may be under the "Applications from personal account" heading.
2. Navigate to the "Authentication" page for the registration. Under "Redirect URLs", you should see an entry ending in `/auth/login/microsoftaccount/callback`. Copy this URI.
3. Add a new URI that matches the one you just copied, except instead have it end in `/auth/login/aad/callback`. This will allow the registration to be used by the App Service Authentication / Authorization configuration.
4. Navigate to the App Service Authentication / Authorization configuration for your app.
5. Collect the configuration for the Microsoft Account provider.
6. Configure the Microsoft Entra provider using the "Advanced" management mode, supplying the client ID and client secret values you collected in the previous step. For the Issuer URL, use `<authentication-endpoint>/<tenant-id>/v2.0`, and replace `<authentication-endpoint>` with the [authentication endpoint for your cloud environment](#) (e.g., "https://login.microsoftonline.com" for global Azure), also replacing `<tenant-id>` with your **Directory (tenant) ID**.
7. Once you've saved the configuration, test the login flow by navigating in your browser to the `/auth/login/aad` endpoint on your site and complete the sign-in flow.
8. At this point, you've successfully copied the configuration over, but the existing Microsoft Account provider configuration remains. Before you remove it, make sure that all parts of your app reference the Microsoft Entra provider through login links, etc. Verify that all parts of your app work as expected.
9. Once you've validated that things work against the Microsoft Entra provider, you may remove the Microsoft Account provider configuration.

Warning

It is possible to converge the two registrations by modifying the **supported account types** for the Microsoft Entra app registration. However, this would force a new consent prompt for Microsoft Account users, and those users' identity claims

may be different in structure, notably changing values since a new App ID is being used. This approach is not recommended unless thoroughly understood. You should instead wait for support for the two registrations in the V2 API surface.

Switching to V2

Once the above steps have been performed, navigate to the app in the Azure portal. Select the "Authentication (preview)" section.

Alternatively, you may make a PUT request against the `config/authsettingsv2` resource under the site resource. The schema for the payload is the same as captured in [File-based configuration](#).

Pin your app to a specific authentication runtime version

When you enable authentication/authorization, platform middleware is injected into your HTTP request pipeline as described in the [feature overview](#). This platform middleware is periodically updated with new features and improvements as part of routine platform updates. By default, your web or function app will run on the latest version of this platform middleware. These automatic updates are always backwards compatible. However, in the rare event that this automatic update introduces a runtime issue for your web or function app, you can temporarily roll back to the previous middleware version. This article explains how to temporarily pin an app to a specific version of the authentication middleware.

Automatic and manual version updates

You can pin your app to a specific version of the platform middleware by setting a `runtimeVersion` setting for the app. Your app always runs on the latest version unless you choose to explicitly pin it back to a specific version. There will be a few versions supported at a time. If you pin to an invalid version that is no longer supported, your app will use the latest version instead. To always run the latest version, set `runtimeVersion` to `~1`.

View and update the current runtime version

You can change the runtime version used by your app. The new runtime version should take effect after restarting the app.

View the current runtime version

You can view the current version of the platform authentication middleware either using the Azure CLI or via one of the built-in version HTTP endpoints in your app.

From the Azure CLI

Using the Azure CLI, view the current middleware version with the [az webapp auth show](#) command.

Azure CLI

```
az webapp auth show --name <my_app_name> \
--resource-group <my_resource_group>
```

In this code, replace `<my_app_name>` with the name of your app. Also replace `<my_resource_group>` with the name of the resource group for your app.

You'll see the `runtimeVersion` field in the CLI output. It will resemble the following example output, which has been truncated for clarity:

Output

```
{
  "additionalLoginParams": null,
  "allowedAudiences": null,
  ...
  "runtimeVersion": "1.3.2",
  ...
}
```

From the version endpoint

You can also hit `/.auth/version` endpoint on an app also to view the current middleware version that the app is running on. It will resemble the following example output:

Output

```
{
  "version": "1.3.2"
}
```

Update the current runtime version

Using the Azure CLI, you can update the `runtimeVersion` setting in the app with the [az webapp auth update](#) command.

Azure CLI

```
az webapp auth update --name <my_app_name> \
--resource-group <my_resource_group> \
--runtime-version <version>
```

Replace `<my_app_name>` with the name of your app. Also replace `<my_resource_group>` with the name of the resource group for your app. Also, replace `<version>` with a valid version of the 1.x runtime or `~1` for the latest version. See the [release notes on the different runtime versions](#) to help determine the version to pin to.

You can run this command from the [Azure Cloud Shell](#) by choosing **Try it** in the preceding code sample. You can also use the [Azure CLI locally](#) to execute this command after executing `az login` to sign in.

Next steps

[Tutorial: Authenticate and authorize users end-to-end](#)

File-based configuration in Azure App Service authentication

Article • 02/02/2022

With [App Service authentication](#), the authentication settings can be configured with a file. You may need to use file-based configuration to use certain preview capabilities of App Service authentication / authorization before they are exposed via [Azure Resource Manager APIs](#).

ⓘ Important

Remember that your app payload, and therefore this file, may move between environments, as with [slots](#). It is likely you would want a different app registration pinned to each slot, and in these cases, you should continue to use the standard configuration method instead of using the configuration file.

Enabling file-based configuration

1. Create a new JSON file for your configuration at the root of your project (deployed to D:\home\site\wwwroot in your web / function app). Fill in your desired configuration according to the [file-based configuration reference](#). If modifying an existing Azure Resource Manager configuration, make sure to translate the properties captured in the `authsettings` collection into your configuration file.
2. Modify the existing configuration, which is captured in the [Azure Resource Manager APIs](#) under `Microsoft.Web/sites/<siteName>/config/authsettingsV2`. To modify this, you can use an [Azure Resource Manager template](#) or a tool like [Azure Resource Explorer](#). Within the authsettingsV2 collection, you will need to set two properties (and may remove others):
 - a. Set `platform.enabled` to "true"
 - b. Set `platform.configFilePath` to the name of the file (for example, "auth.json")

ⓘ Note

The format for `platform.configFilePath` varies between platforms. On Windows, both relative and absolute paths are supported. Relative is recommended. For Linux, only absolute paths are supported currently, so the value of the setting should be "/home/site/wwwroot/auth.json" or similar.

Once you have made this configuration update, the contents of the file will be used to define the behavior of App Service Authentication / Authorization for that site. If you ever wish to return to Azure Resource Manager configuration, you can do so by removing changing the setting `platform.configFilePath` to null.

Configuration file reference

Any secrets that will be referenced from your configuration file must be stored as [application settings](#). You may name the settings anything you wish. Just make sure that the references from the configuration file uses the same keys.

The following exhausts possible configuration options within the file:

```
JSON

{
  "platform": {
    "enabled": <true|false>
  },
  "globalValidation": {
    "unauthenticatedClientAction": "RedirectLoginPage|AllowAnonymous|RejectWith401|RejectWith404",
    "redirectToProvider": "<default provider alias>",
    "excludedPaths": [
      "/path1",
      "/path2",
      "/path3/subpath/*"
    ]
  },
  "httpSettings": {
    "requireHttps": <true|false>,
    "routes": {
      "apiPrefix": "<api prefix>"
    },
    "forwardProxy": {
      "convention": "NoProxy|Standard|Custom",
      "customHostHeaderName": "<host header value>",
      "customProtoHeaderName": "<proto header value>"
    }
  },
  "login": {
    "routes": {
      "logoutEndpoint": "<logout endpoint>"
    },
    "tokenStore": {
      "enabled": <true|false>,
      "tokenRefreshExtensionHours": "<double>",
      "fileSystem": {
        "directory": "<directory to store the tokens in if using a"
      }
    }
  }
}
```



```
        "email"
    ]
},
},
"gitHub": {
    "enabled": <true|false>,
    "registration": {
        "clientId": "<client id>",
        "clientSecretSettingName": "APP_SETTING_CONTAINING_GITHUB_SECRET"
    },
    "login": {
        "scopes": [
            "profile",
            "email"
        ]
    }
},
"google": {
    "enabled": true,
    "registration": {
        "clientId": "<client id>",
        "clientSecretSettingName": "APP_SETTING_CONTAINING_GOOGLE_SECRET"
    },
    "login": {
        "scopes": [
            "profile",
            "email"
        ]
    },
    "validation": {
        "allowedAudiences": [
            "audience1",
            "audience2"
        ]
    }
},
"twitter": {
    "enabled": <true|false>,
    "registration": {
        "consumerKey": "<consumer key>",
        "consumerSecretSettingName": "APP_SETTING_CONTAINING_TWITTER_CONSUMER_SECRET"
    }
},
"apple": {
    "enabled": <true|false>,
    "registration": {
        "clientId": "<client id>",
        "clientSecretSettingName": "APP_SETTING_CONTAINING_APPLE_SECRET"
    },
    "login": {
        "scopes": [

```

```

        "profile",
        "email"
    ]
},
},
"openIdConnectProviders": {
    "<providerName>": {
        "enabled": <true|false>,
        "registration": {
            "clientId": "<client id>",
            "clientCredential": {
                "clientSecretSettingName": "<name of app setting
containing client secret>"
            },
            "openIdConnectConfiguration": {
                "authorizationEndpoint": "<url specifying
authorization endpoint>",
                "tokenEndpoint": "<url specifying token endpoint>",
                "issuer": "<url specifying issuer>",
                "certificationUri": "<url specifying jwks
endpoint>",
                "wellKnownOpenIdConfiguration": "<url specifying
.well-known/open-id-configuration endpoint - if this property is set, the
other properties of this object are ignored, and authorizationEndpoint,
tokenEndpoint, issuer, and certificationUri are set to the corresponding
values listed at this endpoint>"
            }
        },
        "login": {
            "nameClaimType": "<name of claim containing name>",
            "scopes": [
                "openid",
                "profile",
                "email"
            ],
            "loginParameterNames": [
                "paramName1=value1",
                "paramName2=value2"
            ],
            ...
        }
    },
    //...
}
}

```

More resources

- Tutorial: Authenticate and authorize users end-to-end
- Environment variables and app settings for authentication

Work with access keys in Azure Functions

Article • 07/18/2024

Azure Functions lets you use secret keys to make it more difficult to access your function endpoints. This article describes the various kinds of access keys supported by Functions, and how to work with access keys.

While access keys provide some mitigation against unwanted access, you should consider other options to secure HTTP endpoints in production. For example, it's not a good practice to distribute shared secrets in a public app. If your function is being called from a public client, you should consider implementing these or other security mechanisms:

- [Enable App Service Authentication/Authorization](#)
- [Use Azure API Management \(APIM\) to authenticate requests](#)
- [Deploy your function app to a virtual network](#)
- [Deploy your function app in isolation](#)

Access keys provide the basis for HTTP authorization in HTTP triggered functions. For more information, see [Authorization level](#).

Understand keys

The scope of an access key and the actions it supports depend on the type of access key.

[] [Expand table](#)

Key type	Key name	HTTP auth level	Description
Function	<code>default</code> or <code>user</code> <code>defined</code>	<code>function</code>	Allows access only to a specific function endpoint.
Host	<code>default</code> or <code>user</code> <code>defined</code>	<code>function</code>	Allows access to all function endpoints in a function app.
Master	<code>_master</code>	<code>admin</code>	Special host key that also provides administrative access to the runtime REST APIs in a function app. This key can't be

Key type	Key name	HTTP auth level	Description
			revoked. Because the master key grants elevated permissions in your function app, you shouldn't share this key with third parties or distribute it in native client applications.
System	Depends on the extension	n/a	<p>Specific extensions might require a system-managed key to access webhook endpoints. System keys are designed for extension-specific function endpoints that get called by internal components. For example, the Event Grid trigger requires that the subscription use a system key when calling the trigger endpoint. Durable Functions also uses system keys to call Durable Task extension APIs.</p> <p>System keys can only be created by specific extensions, and you can't explicitly set their values. Like other keys, you can generate a new value for the key from the portal or by using the key APIs.</p>

Each key is named for reference, and there's a default key (named `default`) at the function and host level. Function keys take precedence over host keys. When two keys are defined with the same name, the function key is always used.

The following table compares the uses for various kinds of access keys:

[+] [Expand table](#)

Action	Scope	Key type
Execute a function	Specific function	Function
Execute a function	Any function	Function or host
Call an <code>admin</code> endpoint	Function app	Master-only
Call Durable Task extension APIs	Function app*	System
Call an extension-specific Webhook (internal)	Function app*	system

*Scope determined by the extension.

Key requirements

In Functions, access keys are randomly generated 32-byte arrays that are encoded as URL-safe base-64 strings. While you can generate your own access keys and use them

with Functions, we strongly recommend that you instead allow Functions to generate all of your access keys for you.

Functions-generated access keys include special signature and checksum values that indicate the type of access key and that it was generated by Azure Functions. Having these extra components in the key itself makes it much easier to determine the source of these kinds of secrets located during security scanning and other automated processes.

To allow Functions to generate your keys for you, don't supply the key `value` to any of the APIs that you can use to generate keys.

Manage key storage

Keys are stored as part of your function app in Azure and are encrypted at rest. By default, keys are stored in a Blob storage container in the account provided by the `AzureWebJobsStorage` setting. You can use the `AzureWebJobsSecretStorageType` setting to override this default behavior and instead store keys in one of these alternate locations:

[+] Expand table

Location	Value	Description
A second storage account	<code>blob</code>	Stores keys in Blob storage in a storage account that's different than the one used by the Functions runtime. The specific account and container used is defined by a shared access signature (SAS) URL set in the <code>AzureWebJobsSecretStorageSas</code> setting. You must maintain the <code>AzureWebJobsSecretStorageSas</code> setting when the SAS URL changes.
Azure Key Vault	<code>keyvault</code>	The key vault set in <code>AzureWebJobsSecretStorageKeyVaultUri</code> is used to store keys.
File system	<code>files</code>	Keys are persisted on the local file system, which is the default in Functions v1.x. File system storage isn't recommended.
Kubernetes Secrets	<code>kubernetes</code>	The resource set in <code>AzureWebJobsKubernetesSecretName</code> is used to store keys. Supported only when your function app is deployed to Kubernetes. The Azure Functions Core Tools generates the values automatically when you use it to deploy your app to a Kubernetes cluster.

When using Key Vault for key storage, the app settings you need depend on the managed identity type, either system-assigned or user-assigned.

Setting name	System-assigned	User-assigned	App registration
AzureWebJobsSecretStorageKeyVaultUri	✓	✓	✓
AzureWebJobsSecretStorageKeyVaultClientId	X	✓	✓
AzureWebJobsSecretStorageKeyVaultClientSecret	X	X	✓
AzureWebJobsSecretStorageKeyVaultTenantId	X	X	✓

Use access keys

HTTP triggered functions can generally be called by using a URL in the format:

`https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>`. When the authorization level of a given function is set a value other than `anonymous`, you must also provide an access key in your request. The access key can either be provided in the URL using the `?code=` query string or in the request header (`x-functions-key`). For more information, see [Access key authorization](#).

To access the runtime REST APIs (under `/admin/`), you must provide the master key (`_master`) in the `x-functions-key` request header. You can [remove the admin endpoints](#) using the `functionsRuntimeAdminIsolationEnabled` site property.

Get your function access keys

You can get function and host keys programmatically by using these Azure Resource Manager APIs:

- [List Function Keys](#)
- [List Host Keys](#)
- [List Function Keys Slot](#)
- [List Host Keys Slot](#).

To learn how to call Azure Resource Manager APIs, see the [Azure REST API reference](#).

You can use these methods to get access keys without having to use the REST APIs.

Azure portal

1. Sign in to the Azure portal, then search for and select **Function App**.
2. Select the function app you want to work with.
3. In the left pane, expand **Functions**, and then select **App keys**.

The **App keys** page appears. On this page the host keys are displayed, which can be used to access any function in the app. The system key is also displayed, which gives anyone administrator-level access to all function app APIs.

You can also practice least privilege by using the key for a specific function. You can get function-specific keys from the **Function keys** tab of a specific HTTP-triggered function.

Renew or create access keys

When you renew or create your access key values, you must manually redistribute the updated key values to all clients that call your function.

You can renew function and host keys programmatically or create new ones by using these Azure Resource Manager APIs:

- [Create Or Update Function Secret](#)
- [Create Or Update Function Secret Slot](#)
- [Create Or Update Host Secret](#)
- [Create Or Update Host Secret Slot](#)

To learn how to call Azure Resource Manager APIs, see the [Azure REST API reference](#).

You can use these methods to get access keys without having to manually create calls to the REST APIs.

Azure portal

1. Sign in to the Azure portal, then search for and select **Function App**.
2. Select the function app you want to work with.
3. In the left pane, expand **Functions**, and then select **App keys**.

The **App keys** page appears. On this page the host keys are displayed, which can be used to access any function in the app. The system key is also displayed, which gives anyone administrator-level access to all function app APIs.

4. Select **Renew key value** next to the key you want to renew, then select **Renew and save**.

You can also renew a function key in the **Function keys** tab of a specific HTTP-triggered function.

Delete access keys

You can delete function and host keys programmatically by using these Azure Resource Manager APIs:

- [Delete Function Secret](#)
- [Delete Function Secret Slot](#)
- [Delete Host Secret](#)
- [Delete Host Secret Slot](#)

To learn how to call Azure Resource Manager APIs, see the [Azure REST API reference](#).

Related content

- [Securing Azure Functions](#)
- [Azure Functions HTTP trigger](#)
- [Manage your function app](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Add and manage TLS/SSL certificates in Azure App Service

Article • 09/19/2024

ⓘ Note

Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention <app-name>-<random-hash>. <region>.azurewebsites.net. Existing app names will remain unchanged.

Example: myapp-ds27dh7271ah175.westus-01.azurewebsites.net

For further details, refer to [Unique Default Hostname for App Service Resource](#).

You can add digital security certificates to [use in your application code](#) or to [help secure custom DNS names in Azure App Service](#), which provides a highly scalable, self-patching web hosting service. Currently called Transport Layer Security (TLS) certificates, also previously known as Secure Socket Layer (SSL) certificates, these private or public certificates help you secure internet connections by encrypting data sent between your browser, websites that you visit, and the website server.

The following table lists the options for you to add certificates in App Service:

[+] Expand table

Option	Description
Create a free App Service managed certificate	A private certificate that's free of charge and easy to use if you just need to improve security for your custom domain in App Service.
Import an App Service certificate	A private certificate that's managed by Azure. It combines the simplicity of automated certificate management and the flexibility of renewal and export options.
Import a certificate from Key Vault	Useful if you use Azure Key Vault to manage your PKCS12 certificates . See Private certificate requirements .
Upload a private certificate	If you already have a private certificate from a third-party provider, you can upload it. See Private certificate requirements .

Option	Description
Upload a public certificate	Public certificates aren't used to secure custom domains, but you can load them into your code if you need them to access remote resources.

Prerequisites

- [Create an App Service app](#). The app's **App Service plan** must be in the **Basic**, **Standard**, **Premium**, or **Isolated** tier. See [Scale up an app](#) to update the tier.
- For a private certificate, make sure that it satisfies all [requirements from App Service](#).
- **Free certificate only:**
 - Map the domain where you want the certificate to App Service. For information, see [Tutorial: Map an existing custom DNS name to Azure App Service](#).
 - For a root domain (like contoso.com), make sure your app doesn't have any [IP restrictions](#) configured. Both certificate creation and its periodic renewal for a root domain depend on your app being reachable from the internet.

Private certificate requirements

The [free App Service managed certificate](#) and the [App Service certificate](#) already satisfy the requirements of App Service. If you choose to upload or import a private certificate to App Service, your certificate must meet the following requirements:

- Exported as a [password-protected PFX file](#), encrypted using triple DES
- Contains private key at least 2048 bits long
- Contains all intermediate certificates and the root certificate in the certificate chain

If you want to help secure a custom domain in a TLS binding, the certificate must meet these additional requirements:

- Contains an [Extended Key Usage](#) for server authentication (OID = 1.3.6.1.5.5.7.3.1)
- Signed by a trusted certificate authority

ⓘ Note

[Elliptic Curve Cryptography \(ECC\) certificates](#) work with App Service but aren't covered by this article. For the exact steps to create ECC certificates, work with your

certificate authority.

ⓘ Note

After you add a private certificate to an app, the certificate is stored in a deployment unit that's bound to the App Service plan's resource group, region, and operating system combination, internally called a *webspace*. That way, the certificate is accessible to other apps in the same resource group, region, and OS combination. Private certificates uploaded or imported to App Service are shared with App Services in the same deployment unit.

You can add up to 1000 private certificates per webspace.

Create a free managed certificate

The free App Service managed certificate is a turn-key solution for helping to secure your custom DNS name in App Service. Without any action from you, this TLS/SSL server certificate is fully managed by App Service and is automatically renewed continuously in six-month increments, 45 days before expiration, as long as the prerequisites that you set up stay the same. All the associated bindings are updated with the renewed certificate. You create and bind the certificate to a custom domain, and let App Service do the rest.

ⓘ Important

Before you create a free managed certificate, make sure you have [met the prerequisites](#) for your app.

Free certificates are issued by DigiCert. For some domains, you must explicitly allow DigiCert as a certificate issuer by creating a [CAA domain record](#) ↗ with the value: `issue digicert.com`.

Azure fully manages the certificates on your behalf, so any aspect of the managed certificate, including the root issuer, can change at anytime. These changes are outside your control. Make sure to avoid hard dependencies and "pinning" practice certificates to the managed certificate or any part of the certificate hierarchy. If you need the certificate pinning behavior, add a certificate to your custom domain using any other available method in this article.

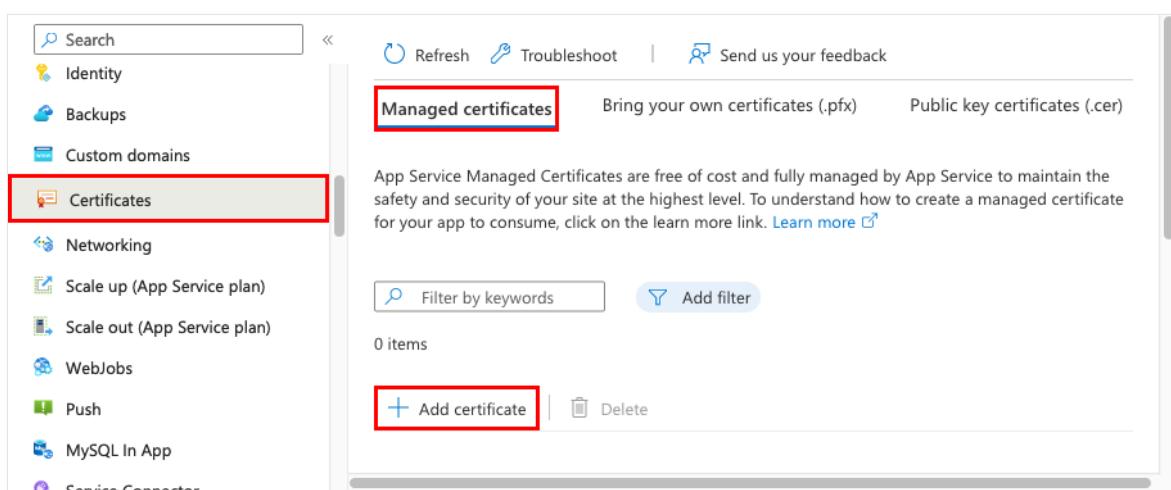
The free certificate comes with the following limitations:

- Doesn't support wildcard certificates.
- Doesn't support usage as a client certificate by using certificate thumbprint, which is planned for deprecation and removal.
- Doesn't support private DNS.
- Isn't exportable.
- Isn't supported in an App Service Environment.
- Only supports alphanumeric characters, dashes (-), and periods (.)
- Only custom domains of length up to 64 characters are supported.

Apex domain

- Must have an A record pointing to your web app's IP address.
- Must be on apps that are publicly accessible.
- Isn't supported with root domains that are integrated with Traffic Manager.
- Must meet all of the above for successful certificate issuances and renewals.

1. In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.
2. On your app's navigation menu, select **Certificates**. In the **Managed certificates** pane, select **Add certificate**.



3. Select the custom domain for the free certificate, and then select **Validate**. When validation completes, select **Add**. You can create only one managed certificate for each supported custom domain.

When the operation completes, the certificate appears in the **Managed certificates** list.

[Managed certificates](#)[Bring your own certificates \(.pfx\)](#)[Public key certificates \(.cer\)](#)

App Service Managed Certificates are free of cost and fully managed by App Service to maintain the safety and security of your site at the highest level. To understand how to create a managed certificate for your app to consume, click on the learn more link. [Learn more](#)

[Filter by keywords](#)[Add filter](#)

1 items

[Add certificate](#)[Delete](#)

Certificate Status ↑	Domain	Certificate Name
<input type="checkbox"/> ✓ No action needed	www.contoso.com	Contoso www

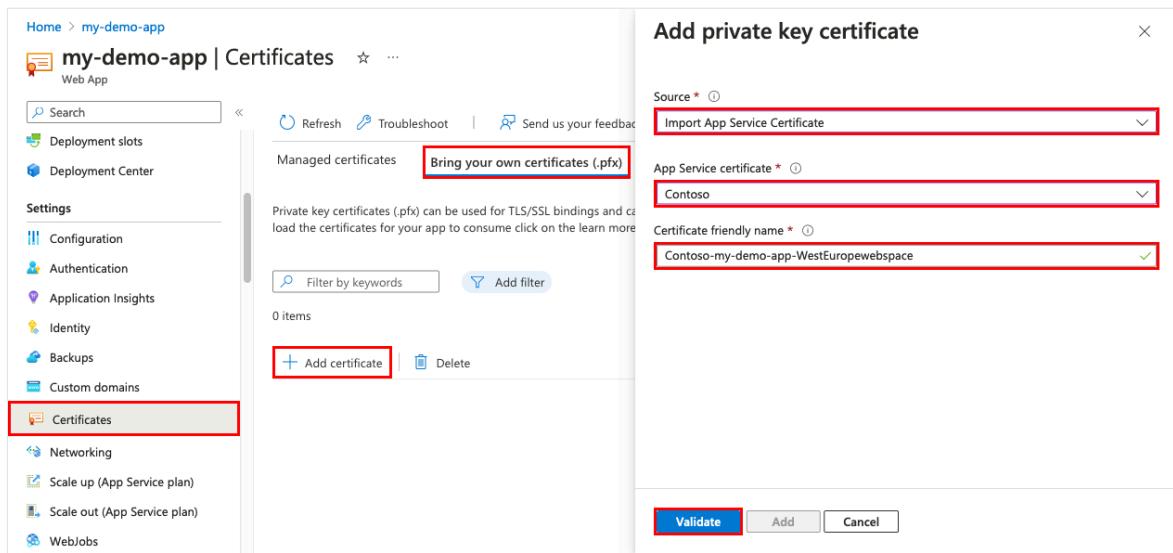
< Previous Items per page: 50 Page 1 of 1 Next >

4. To provide security for a custom domain with this certificate, you still have to create a certificate binding. Follow the steps in [Secure a custom DNS name with a TLS/SSL binding in Azure App Service](#).

Import an App Service certificate

To import an App Service certificate, first [buy and configure an App Service certificate](#), and then follow the steps here.

1. In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.
2. From your app's navigation menu, select **Certificates > Bring your own certificates (.pfx) > Add certificate**.
3. In Source, select **Import App Service Certificate**.
4. In **App Service certificate**, select the certificate you just created.
5. In **Certificate friendly name**, give the certificate a name in your app.
6. Select **Validate**. When validation succeeds, select **Add**.



When the operation completes, the certificate appears in the **Bring your own certificates (.pfx)** list.

Certificate Status ↑	Domain	Certificate Name	Expiration D... ↑	Tl
<input type="checkbox"/> ✓ No action needed	contoso.com,www.contoso.com	Contoso-my-demo-app-WestEurope...	29/06/2024	OF

7. To help secure a custom domain with this certificate, you still have to create a certificate binding. Follow the steps in [Secure a custom DNS name with a TLS/SSL binding in Azure App Service](#).

Import a certificate from Key Vault

If you use Azure Key Vault to manage your certificates, you can import a PKCS12 certificate into App Service from Key Vault if you met the [requirements](#).

Authorize App Service to read from the vault

By default, the App Service resource provider doesn't have access to your key vault. To use a key vault for a certificate deployment, you must authorize read access for the resource provider (App Service) to the key vault. You can grant access either with access policy or RBAC.

 **Note**

Currently, the Azure portal does not allow you to configure an App Service certificate in Key Vault to use the RBAC model. You can, however, use Azure CLI, Azure PowerShell, or an ARM template deployment to perform this configuration.

RBAC permissions

 Expand table

Resource provider	Service principal app ID / assignee	Key vault RBAC role
Microsoft Azure App Service or Microsoft.Azure.WebSites	- abfa0a7c-a6b6-4736-8310-5855508787cd for public Azure cloud environment - 6a02c803-daf3-4136-b4c3-5a6f318b4714 for Azure Government cloud environment	Certificate User

The service principal app ID or assignee value is the ID for the App Service resource provider. To learn how to authorize key vault permissions for the App Service resource provider using an access policy, see the [provide access to Key Vault keys, certificates, and secrets with an Azure role-based access control documentation](#).

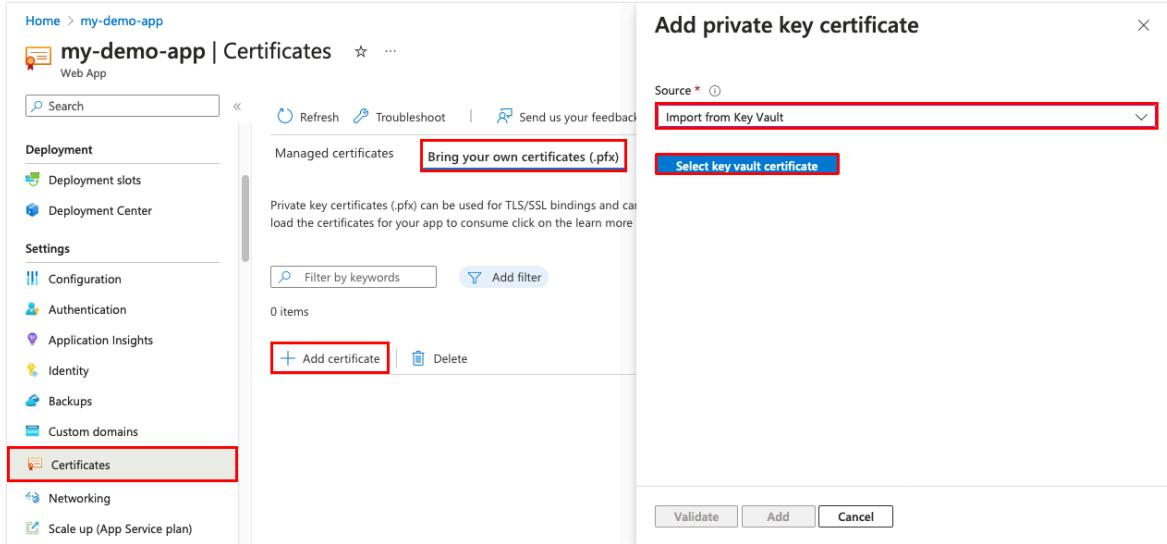
 **Note**

Do not delete these RBAC permissions from key vault. If you do, App Service will not be able to sync your web app with the latest key vault certificate version.

Import a certificate from your vault to your app

1. In the [Azure portal](#), from the left menu, select App Services > <app-name>.

2. From your app's navigation menu, select **Certificates** > **Bring your own certificates (.pfx)** > **Add certificate**.
3. In Source, select **Import from Key Vault**.
4. Select **Select key vault certificate**.



5. To help you select the certificate, use the following table:

[Expand table](#)

Setting	Description
Subscription	The subscription associated with the key vault.
Key vault	The key vault that has the certificate you want to import.
Certificate	From this list, select a PKCS12 certificate that's in the vault. All PKCS12 certificates in the vault are listed with their thumbprints, but not all are supported in App Service.

6. When finished with your selection, select **Select**, **Validate**, and then **Add**.

When the operation completes, the certificate appears in the **Bring your own certificates** list. If the import fails with an error, the certificate doesn't meet the [requirements for App Service](#).

Private key certificates (.pfx) can be used for TLS/SSL bindings and can be loaded to the certificate store for your app to consume. To understand how to load the certificates for your app to consume click on the learn more link. [Learn more](#)

Filter by keywords Add filter

1 items

Add certificate | Delete

Certificate Status ↑	Domain	Certificate Name	Expiration D... ↑	T...
<input type="checkbox"/> ✓ No action needed	contoso.com,www.contoso.com	Contoso-my-demo-app-WestEurope...	29/06/2024	OF

< Previous Items per page: 50 Page 1 of 1 Next >

ⓘ Note

If you update your certificate in Key Vault with a new certificate, App Service automatically syncs your certificate within 24 hours.

7. To help secure custom domain with this certificate, you still have to create a certificate binding. Follow the steps in [Secure a custom DNS name with a TLS/SSL binding in Azure App Service](#).

Upload a private certificate

After you get a certificate from your certificate provider, make the certificate ready for App Service by following the steps in this section.

Merge intermediate certificates

If your certificate authority gives you multiple certificates in the certificate chain, you must merge the certificates following the same order.

1. In a text editor, open each received certificate.
2. To store the merged certificate, create a file named *mergedcertificate.crt*.
3. Copy the content for each certificate into this file. Make sure to follow the certificate sequence specified by the certificate chain, starting with your certificate and ending with the root certificate, for example:

```
-----BEGIN CERTIFICATE-----
<your entire Base64 encoded SSL certificate>
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----
<The entire Base64 encoded intermediate certificate 1>
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----
<The entire Base64 encoded intermediate certificate 2>
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----
<The entire Base64 encoded root certificate>
-----END CERTIFICATE-----
```

Export the merged private certificate to PFX

Now, export your merged TLS/SSL certificate with the private key that was used to generate your certificate request. If you generated your certificate request using OpenSSL, then you created a private key file.

ⓘ Note

OpenSSL v3 changed the default cipher from 3DES to AES256, but this can be overridden on the command line: -keypbe PBE-SHA1-3DES -certpbe PBE-SHA1-3DES -macalg SHA1. OpenSSL v1 uses 3DES as the default, so the PFX files generated are supported without any special modifications.

1. To export your certificate to a PFX file, run the following command, but replace the placeholders <*private-key-file*> and <*merged-certificate-file*> with the paths to your private key and your merged certificate file.

Bash

```
openssl pkcs12 -export -out myserver.pfx -inkey <private-key-file> -in
<merged-certificate-file>
```

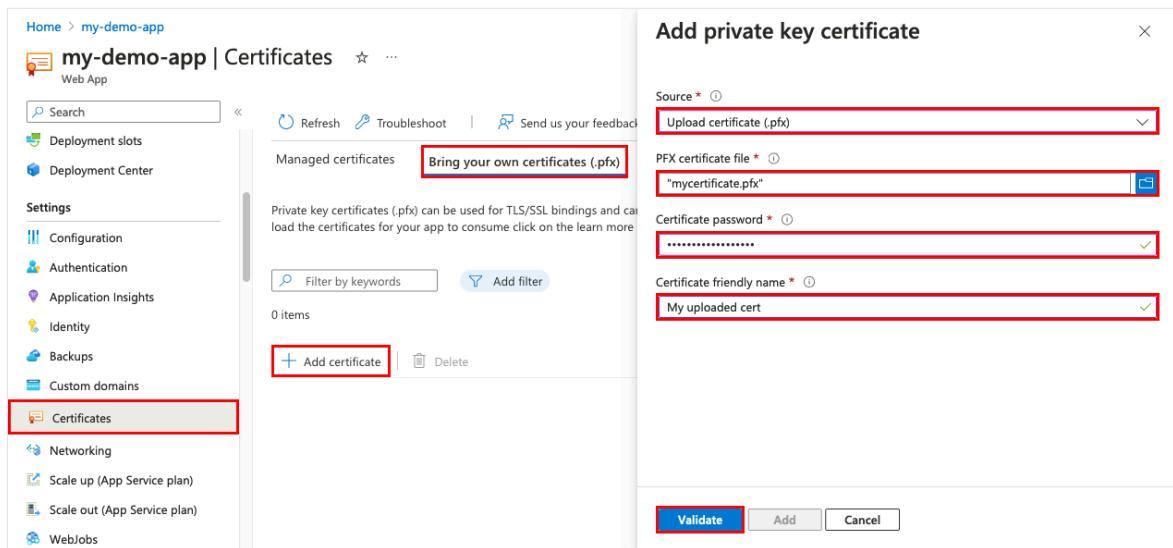
2. When you're prompted, specify a password for the export operation. When you upload your TLS/SSL certificate to App Service later, you must provide this password.

3. If you used IIS or *Certreq.exe* to generate your certificate request, install the certificate to your local computer, and then [export the certificate to a PFX file](#).

Upload the certificate to App Service

You're now ready upload the certificate to App Service.

1. In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.
2. From your app's navigation menu, select **Certificates > Bring your own certificates (.pfx) > Upload Certificate**.



3. To help you upload the .pfx certificate, use the following table:

[+] [Expand table](#)

Setting	Description
PFX certificate file	Select your .pfx file.
Certificate password	Enter the password that you created when you exported the PFX file.
Certificate friendly name	The certificate name that will be shown in your web app.

4. When finished with your selection, select **Select, Validate, and then Add**.

When the operation completes, the certificate appears in the **Bring your own certificates** list.

Private key certificates (.pfx) can be used for TLS/SSL bindings and can be loaded to the certificate store for your app to consume. To understand how to load the certificates for your app to consume click on the learn more link. [Learn more](#)

Filter by keywords Add filter

1 items

Add certificate Delete

Certificate Status ↑	Domain	Certificate Name	Expiration D... ↑	T...
<input type="checkbox"/> ✓ No action needed	contoso.com,www.contoso.com	Contoso-my-demo-app-WestEurope...	29/06/2024	OF

< Previous Items per page: 50 Page 1 of 1 Next >

5. To provide security for a custom domain with this certificate, you still have to create a certificate binding. Follow the steps in [Secure a custom DNS name with a TLS/SSL binding in Azure App Service](#).

Upload a public certificate

Public certificates are supported in the .cer format.

ⓘ Note

After you upload a public certificate to an app, it's only accessible by the app it's uploaded to. Public certificates must be uploaded to each individual web app that needs access. For App Service Environment specific scenarios, refer to [the documentation for certificates and the App Service Environment](#).

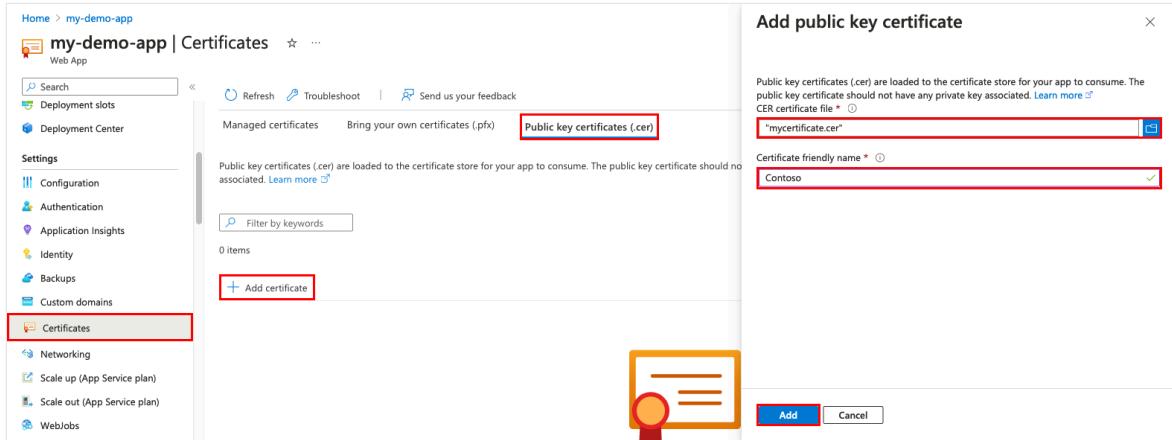
You can upload up to 1000 public certificates per App Service Plan.

1. In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.
2. From your app's navigation menu, select **Certificates > Public key certificates (.cer) > Add certificate**.
3. To help you upload the .cer certificate, use the following table:

[Expand table](#)

Setting	Description
CER certificate file	Select your .cer file.
Certificate friendly name	The certificate name that will be shown in your web app.

4. When you're done, select **Add**.



5. After the certificate is uploaded, copy the certificate thumbprint, and then review [Make the certificate accessible](#).

Renew an expiring certificate

Before a certificate expires, make sure to add the renewed certificate to App Service, and update any certificate bindings where the process depends on the certificate type. For example, a [certificate imported from Key Vault](#), including an [App Service certificate](#), automatically syncs to App Service every 24 hours and updates the TLS/SSL binding when you renew the certificate. For an [uploaded certificate](#), there's no automatic binding update. Based on your scenario, review the corresponding section:

- [Renew an uploaded certificate](#)
- [Renew an App Service certificate](#)
- [Renew a certificate imported from Key Vault](#)

Renew an uploaded certificate

When you replace an expiring certificate, the way you update the certificate binding with the new certificate might adversely affect the user experience. For example, your inbound IP address might change when you delete a binding, even if that binding is IP-based. This result is especially impactful when you renew a certificate that's already in an IP-based binding. To avoid a change in your app's IP address, and to avoid downtime for your app due to HTTPS errors, follow these steps in the specified sequence:

1. Upload the new certificate.
2. Go to the **Custom domains** page for your app, select the ... button, and then select **Update binding**.
3. Select the new certificate and then select **Update**.
4. Delete the existing certificate.

Renew a certificate imported from Key Vault

ⓘ Note

To renew an App Service certificate, see [Renew an App Service certificate](#).

To renew a certificate that you imported into App Service from Key Vault, review [Renew your Azure Key Vault certificate](#).

After the certificate renews in your key vault, App Service automatically syncs the new certificate and updates any applicable certificate binding within 24 hours. To sync manually, follow these steps:

1. Go to your app's **Certificate** page.
2. Under **Bring your own certificates (.pfx)**, select the ... button for the imported key vault certificate, and then select **Sync**.

Frequently asked questions

How can I automate adding a bring-your-own certificate to an app?

- [Azure CLI: Bind a custom TLS/SSL certificate to a web app](#)
- [Azure PowerShell: Bind a custom TLS/SSL certificate to a web app using PowerShell](#)

Can I use a private CA (certificate authority) certificate for inbound TLS on my app?

You can use a private CA certificate for inbound TLS in [App Service Environment version 3](#). This isn't possible in App Service (multi-tenant). For more information on App Service

multi-tenant vs. single-tenant, see [App Service Environment v3](#) and [App Service public multitenant comparison](#).

Can I make outbound calls using a private CA client certificate from my app?

This is only supported for Windows container apps in multi-tenant App Service. In addition, you can make outbound calls using a private CA client certificate with both code-based and container-based apps in [App Service Environment version 3](#). For more information on App Service multi-tenant vs. single-tenant, see [App Service Environment v3](#) and [App Service public multitenant comparison](#).

Can I load a private CA certificate in my App Service Trusted Root Store?

You can load your own CA certificate into the Trusted Root Store in [App Service Environment version 3](#). You can't modify the list of Trusted Root Certificates in App Service (multi-tenant). For more information on App Service multi-tenant vs. single-tenant, see [App Service Environment v3](#) and [App Service public multitenant comparison](#).

More resources

- [Secure a custom DNS name with a TLS/SSL binding in Azure App Service](#)
- [Enforce HTTPS](#)
- [Enforce TLS 1.1/1.2](#)
- [Use a TLS/SSL certificate in your code in Azure App Service](#)
- [FAQ: App Service Certificates](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Set up Azure App Service access restrictions

Article • 08/21/2024

ⓘ Note

Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention <app-name>-<random-hash>. <region>.azurewebsites.net. Existing app names will remain unchanged.

Example: myapp-ds27dh7271aah175.westus-01.azurewebsites.net

For further details, refer to [Unique Default Hostname for App Service Resource ↗](#).

By setting up access restrictions, you can define a priority-ordered allow/deny list that controls network access to your app. The list can include IP addresses or Azure Virtual Network subnets. When there are one or more entries, an implicit *deny all* exists at the end of the list. To learn more about access restrictions, go to the [access restrictions overview](#).

The access restriction capability works with all Azure App Service-hosted workloads. The workloads can include web apps, API apps, Linux apps, Linux custom containers and Functions.

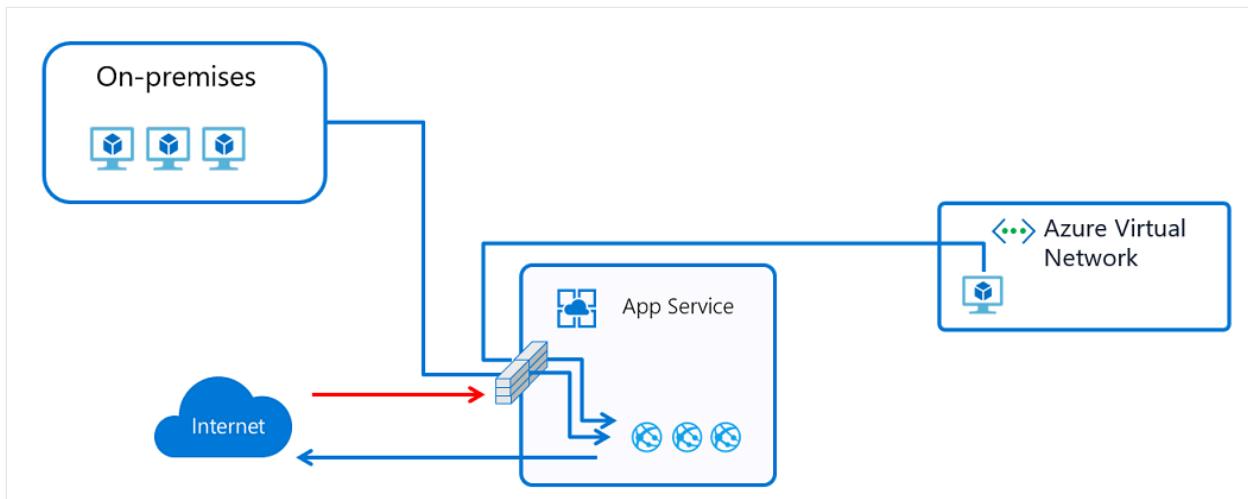
When a request is made to your app, the FROM address is evaluated against the rules in your access restriction list. If the FROM address is in a subnet configured with service endpoints to Microsoft.Web, the source subnet is compared against the virtual network rules in your access restriction list. If the address isn't allowed access based on the rules in the list, the service replies with an [HTTP 403 ↗](#) status code.

The access restriction capability is implemented in the App Service front-end roles, which are upstream of the worker hosts where your code runs. Therefore, access restrictions are effectively network access-control lists (ACLs).

The ability to restrict access to your web app from an Azure virtual network uses [service endpoints](#). With service endpoints, you can restrict access to a multitenant service from selected subnets. It doesn't work to restrict traffic to apps that are hosted in an App Service Environment. If you're in an App Service Environment, you can control access to your app by applying IP address rules.

⚠ Note

The service endpoints must be enabled both on the networking side and for the Azure service that they're being enabled with. For a list of Azure services that support service endpoints, see [Virtual Network service endpoints](#).



Manage access restriction rules in the portal

To add an access restriction rule to your app, do the following steps:

1. Sign in to the Azure portal.
2. Select the app that you want to add access restrictions to.
3. On the left menu, select **Networking**.
4. On the **Networking** page, under **Inbound traffic configuration**, select the **Public network access** setting.

Home > network-integration-webapp

network-integration-webapp | Networking

Web App

Search Refresh Troubleshoot Send us your feedback

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Microsoft Defender for Cloud Events (preview) Log stream

Deployment Deployment slots Deployment Center

Settings Environment variables

Check your network configuration. Select any of the features listed below to change your network setup. [Learn more](#)

Inbound traffic configuration

Public network access	Enabled with no access restrictions
App assigned address	Not configured
Private endpoints	0 private endpoints
Inbound addresses	20.105.232.6

Optional inbound services

Azure Front Door	View details
------------------	------------------------------

5. On the **Access Restrictions** page, review the list of access restriction rules that are defined for your app.

Access Restrictions

Save Refresh

App access

Public access is applied to both main site and advanced tool site. Deny public network access will block all incoming traffic except that comes from private endpoints. [Learn more](#)

Public network access Enabled from select virtual networks and IP addresses Enabled from all networks (This will clear all current access restrictions) Disabled

Site access and rules

Main site Advanced tool site

You can define lists of allow/deny rules to control traffic to your site. Rules are evaluated in priority order. If no created rule is matched to the traffic, the "Unmatched rule action" will control how the traffic is handled. [Learn more](#)

Unmatched rule action Deny Allow

[+ Add](#) [Delete](#)

Filter rules Action : All

Priority ↑	Name	Source	Action	HTTP headers
80	Deny example	122.133.144.32/28	Deny	Not configured
100	IP example rule	122.133.144.0/24	Allow	Not configured
2147483647	Deny all	Any	Deny	Not configured

The list displays all the current restrictions that are applied to the app. If you have a virtual network restriction on your app, the table shows whether the service endpoints are enabled for Microsoft.Web. If no restrictions are defined on your app and your unmatched rule isn't set to Deny, the app is accessible from anywhere.

Permissions

The following Role-based access control permissions on the subnet or at a higher level are required to configure access restrictions through Azure portal, CLI or when setting the site config properties directly:

[+] Expand table

Action	Description
Microsoft.Web/sites/config/read	Get Web App configuration settings
Microsoft.Web/sites/config/write	Update Web App's configuration settings
Microsoft.Network/virtualNetworks/subnets/joinViaServiceEndpoint/action*	Joins resource such as storage account or SQL database to a subnet
Microsoft.Web/sites/write**	Update Web App settings

*only required when adding a virtual network (service endpoint) rule.

**only required if you're updating access restrictions through Azure portal.

If you're adding a service endpoint-based rule and the virtual network is in a different subscription than the app, you must ensure that the subscription with the virtual network is registered for the `Microsoft.Web` resource provider. You can explicitly register the provider [by following this documentation](#), but also automatically registered when creating the first web app in a subscription.

Add an access restriction rule

To add an access restriction rule to your app, on the **Access Restrictions** page, select **Add**. The rule is only effective after saving.

Rules are enforced in priority order, starting from the lowest number in the **Priority** column. If you don't configure unmatched rule, an implicit *deny all* is in effect after you add even a single rule.

On the **Add Access Restriction** pane, when you create a rule, do the following:

1. Under Action, select either Allow or Deny.

The screenshot shows the 'Add Access Restriction' dialog box. It has the following fields:

- Name**: A text input field containing "Enter name for the IpAddress rule" with a green checkmark icon to its right.
- Action**: A button group with two options: "Allow" (highlighted in blue) and "Deny".
- Priority ***: A text input field containing "Ex. 300".
- Description**: An empty text input field with a green checkmark icon to its right.
- Type**: A dropdown menu set to "IPv4".
- IP Address Block ***: A text input field containing "Enter an IPv4 CIDR. Ex: 208.130.0.0/16".
- Add rule**: A blue button at the bottom left of the form.

2. Optionally, enter a name and description of the rule.

3. In the **Priority** box, enter a priority value.

4. In the **Type** drop-down list, select the type of rule. The different types of rules are described in the following sections.

5. Select **Add rule** after typing in the rule specific input to add the rule to the list.

Finally select **Save** back in the **Access Restrictions** page.

Note

- There is a limit of 512 access restriction rules. If you require more than 512 access restriction rules, we suggest that you consider installing a standalone security product, such as Azure Front Door, Azure App Gateway, or an alternative WAF.

Set an IP address-based rule

Follow the procedure as outlined in the preceding section, but with the following addition:

- For step 4, in the **Type** drop-down list, select **IPv4 or IPv6**.

Specify the **IP Address Block** in Classless Inter-Domain Routing (CIDR) notation for both the IPv4 and IPv6 addresses. To specify an address, you can use something like **1.2.3.4/32**, where the first four octets represent your IP address and **/32** is the mask. The IPv4 CIDR notation for all addresses is **0.0.0.0/0**. To learn more about CIDR notation, see [Classless Inter-Domain Routing](#).

Note

IP-based access restriction rules only handle virtual network address ranges when your app is in an App Service Environment. If your app is in the multi-tenant service, you need to use **service endpoints** to restrict traffic to select subnets in your virtual network.

Set a service endpoint-based rule

- For step 4, in the **Type** drop-down list, select **Virtual Network**.

Add Access Restriction

X

Name ⓘ

Enter name for the ipAddress rule



Action

Allow

Deny

Priority *

Ex. 300

Description



Type

Virtual Network



Subscription *

Purple Demo Subscription



Virtual Network *

networking-demos-vnet



Subnet *

nat-gw-subnet



Selected subnet 'networking-demos-vnet/nat-gw-subnet' does not have service endpoint enabled for Microsoft.Web. Enabling access may take up to 15 minutes to complete.



Ignore missing Microsoft.Web service endpoints

Add rule

Specify the **Subscription**, **Virtual Network**, and **Subnet** drop-down lists, matching what you want to restrict access to.

By using service endpoints, you can restrict access to selected Azure virtual network subnets. If service endpoints aren't already enabled with **Microsoft.Web** for the subnet that you selected, they're automatically enabled unless you select the **Ignore missing Microsoft.Web service endpoints** check box. The scenario where you might want to

enable service endpoints on the app but not the subnet depends mainly on whether you have the permissions to enable them on the subnet.

If you need someone else to enable service endpoints on the subnet, select the **Ignore missing Microsoft.Web service endpoints** check box. Your app is configured for service endpoints in anticipation of having them enabled later on the subnet.

You can't use service endpoints to restrict access to apps that run in an App Service Environment. When your app is in an App Service Environment, you can control access to it by applying IP access rules.

With service endpoints, you can configure your app with application gateways or other web application firewall (WAF) devices. You can also configure multi-tier applications with secure back ends. For more information, see [Networking features and App Service](#) and [Application Gateway integration with service endpoints](#).

 **Note**

- Service endpoints aren't supported for web apps that use IP-based TLS/SSL bindings with a virtual IP (VIP).

Set a service tag-based rule

- For step 4, in the **Type** drop-down list, select **Service Tag**.

Add Access Restriction X

General settings

Name (i)

 ✓

Action

Allow Deny

Priority *

 ✓

Description

 ✓

Source settings

Type

 ▼

Service Tag *

 ^

ActionGroup

ApplicationInsightsAvailability

AzureCloud

AzureCognitiveSearch

AzureEventGrid

AzureFrontDoor.Backend

AzureMachineLearning

AzureTrafficManager

LogicApps

All publicly available service tags are supported in access restriction rules. Each service tag represents a list of IP ranges from Azure services. A list of these services and links to the specific ranges can be found in the [service tag documentation](#). Use Azure Resource Manager templates or scripting to configure more advanced rules like regional scoped rules.

(i) Note

When creating service tag-based rules through Azure portal or Azure CLI you will need read access at the subscription level to get the full list of service tags for selection/validation. In addition, the `Microsoft.Network` resource provider needs to be registered on the subscription.

Edit a rule

1. To begin editing an existing access restriction rule, on the **Access Restrictions** page, select the rule you want to edit.
2. On the **Edit Access Restriction** pane, make your changes, and then select **Update rule**.
3. Select **Save** to save the changes.

Edit rule ×

General settings

Name

Priority *

Action
 Allow Deny

Description

Source settings

IP Address Block *

Update rule

! **Note**

When you edit a rule, you can't switch between rule types.

Delete a rule

1. To delete a rule, on the **Access Restrictions** page, check the rule or rules you want to delete, and then select **Delete**.
2. Select **Save** to save the changes.

The screenshot shows the 'Access Restrictions' page in the Azure portal. At the top, there are 'Save' and 'Refresh' buttons, with 'Save' highlighted with a red box. Below this is a section titled 'App access' with a note about public access. Under 'Public network access', the 'Enabled from select virtual networks and IP addresses' option is selected. In the 'Site access and rules' section, the 'Main site' tab is active. It shows an 'Unmatched rule action' set to 'Deny'. Below this is a table of rules:

Priority ↑	Name	Source	Action	HTTP headers
80	Deny example	122.133.144.32/28	Deny	Not configured
100	IP example rule	122.133.144.0/24	Allow	Not configured
2147483647	Deny all	Any	Deny	Not configured

At the bottom of the table, there is a 'Delete' button highlighted with a red box.

Access restriction advanced scenarios

The following sections describe some advanced scenarios using access restrictions.

Filter by http header

As part of any rule, you can add http header filters. The following http header names are supported:

- X-Forwarded-For
- X-Forwarded-Host
- X-Azure-FDID
- X-FD-HealthProbe

For each header name, you can add up to eight values separated by comma. The http header filters are evaluated after the rule itself and both conditions must be true for the

rule to apply.

Multi-source rules

Multi-source rules allow you to combine up to eight IP ranges or eight Service Tags in a single rule. You use multi-source rules if you have more than 512 IP ranges or you want to create logical rules. Logical rules could be where multiple IP ranges are combined with a single http header filter.

Multi-source rules are defined the same way you define single-source rules, but with each range separated with comma.

PowerShell example:

Azure PowerShell

```
Add-AzWebAppAccessRestrictionRule -ResourceGroupName "ResourceGroup" -  
WebAppName "AppName"  
-Name "Multi-source rule" -IpAddress  
"192.168.1.0/24,192.168.10.0/24,192.168.100.0/24"  
-Priority 100 -Action Allow
```

Block a single IP address

For a scenario where you want to explicitly block a single IP address or a block of IP addresses, but allow access to everything else, add a **Deny** rule for the specific IP address and configure the unmatched rule action to **Allow**.

The screenshot shows the 'Access Restrictions' section of the Azure App Service configuration. It includes settings for 'App access' (Public network access: Enabled from select virtual networks and IP addresses), 'Site access and rules' (Main site selected), and a table of rules:

Priority ↑	Name	Source	Action	HTTP headers
80	Block Me	122.133.144.32/32	Deny	Not configured
2147483647	Allow all	Any	Allow	Not configured

Restrict access to an SCM site

In addition to being able to control access to your app, you can restrict access to the SCM (Advanced tool) site used by your app. The SCM site is both the web deploy endpoint and the Kudu console. You can assign access restrictions to the SCM site from the app separately or use the same set of restrictions for both the app and the SCM site. When you select the **Use main site rules** check box, the rules list is hidden, and it uses the rules from the main site. If you clear the check box, your SCM site settings appear again.

The screenshot shows the 'Access Restrictions' page in the Azure portal. At the top, there are 'Save' and 'Refresh' buttons. Below them is a section titled 'App access' with a note about public access being applied to both main site and advanced tool site. It includes a 'Public network access' dropdown with three options: 'Enabled from all networks' (radio button), 'Enabled from select virtual networks and IP addresses' (selected radio button), and 'Disabled'.

Below this is a section titled 'Site access and rules'. It has tabs for 'Main site' and 'Advanced tool site' (which is selected). A note says you can define lists of allow/deny rules to control traffic to your site. Rules are evaluated in priority order. If no created rule is matched to the traffic, the "Unmatched rule action" will control how the traffic is handled. The 'Unmatched rule action' dropdown shows 'Deny' selected (radio button).

There is a checkbox for 'Use main site rules' which is unchecked. Below these are 'Add' and 'Delete' buttons, and a search bar labeled 'Filter rules'.

The main area displays a table of rules:

Priority ↑	Name	Source	Action	HTTP headers
200	Deployment server	200.100.50.0/32	Allow	Not configured
2147483647	Deny all	Any	Deny	Not configured

Restrict access to a specific Azure Front Door instance

Traffic from Azure Front Door to your application originates from a well known set of IP ranges defined in the `AzureFrontDoor.Backend` service tag. Using a service tag restriction rule, you can restrict traffic to only originate from Azure Front Door. To ensure traffic only originates from your specific instance, you need to further filter the incoming requests based on the unique http header that Azure Front Door sends.

Add Access Restriction

X

General settings

Name ⓘ

MyAzureFrontDoorRule



Action

Allow Deny

Priority *

100



Description



Source settings

Type

Service Tag



Service Tag *

AzureFrontDoor.Backend



HTTP headers filter settings

X-Forwarded-Host ⓘ

Ex. exampleOne.com, exampleTwo.com

X-Forwarded-For ⓘ

Enter IPv4 or IPv6 CIDR addresses.

X-Azure-FDID ⓘ

xxxxxxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx



X-FD-HealthProbe ⓘ

Ex. 1

PowerShell example:

Azure PowerShell

```
$afd = Get-AzFrontDoor -Name "MyFrontDoorInstanceName"  
Add-AzWebAppAccessRestrictionRule -ResourceGroupName "ResourceGroup" -  
WebAppName "AppName"  
-Name "Front Door example rule" -Priority 100 -Action Allow -ServiceTag
```

```
AzureFrontDoor.Backend  
-HTTPHeader @{'x-azure-fdid' = $afd.FrontDoorId}
```

Manage access restriction programmatically

You can manage access restriction programmatically, below you can find examples of how to add rules to access restrictions and how to change *Unmatched rule action* for both *Main site* and *Advanced tool site*.

Add access restrictions rules for main site

You can add access restrictions rules for *Main site* programmatically by choosing one of the following options:

Azure CLI

You can run the following command in the [Cloud Shell](#). For more information about `az webapp config access-restriction` command, visit [this page](#).

Azure CLI

```
az webapp config access-restriction add --resource-group ResourceGroup -  
-name AppName \  
--rule-name 'IP example rule' --action Allow --ip-address  
122.133.144.0/24 --priority 100  
  
az webapp config access-restriction add --resource-group ResourceGroup -  
-name AppName \  
--rule-name "Azure Front Door example" --action Allow --priority 200 -  
-service-tag AzureFrontDoor.Backend \  
--http-header x-azure-fdid=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

Add access restrictions rules for advanced tool site

You can add access restrictions rules for *Advanced tool site* programmatically by choosing one of the following options:

Azure CLI

You can run the following command in the [Cloud Shell](#). For more information about `az webapp config access-restriction` command, visit [this page](#).

Azure CLI

```
az webapp config access-restriction add --resource-group ResourceGroup --name AppName \
--rule-name 'IP example rule' --action Allow --ip-address 122.133.144.0/24 --priority 100 --scm-site true
```

Change unmatched rule action for main site

You can change *Unmatched rule action* for *Main site* programmatically by choosing one of the following options:

Azure CLI

You can run the following command in the [Cloud Shell](#). For more information about `az resource` command, visit [this page](#). Accepted values for `ipSecurityRestrictionsDefaultAction` are `Allow` or `Deny`.

Azure CLI

```
az resource update --resource-group ResourceGroup --name AppName --resource-type "Microsoft.Web/sites" \
--set properties.siteConfig.ipSecurityRestrictionsDefaultAction=Allow
```

Change unmatched rule action for advanced tool site

You can change *Unmatched rule action* for *Advanced tool site* programmatically by choosing one of the following options:

Azure CLI

You can run the following command in the [Cloud Shell](#). For more information about `az resource` command, visit [this page](#). Accepted values for `scmIpSecurityRestrictionsDefaultAction` are `Allow` or `Deny`.

Azure CLI

```
az resource update --resource-group ResourceGroup --name AppName --resource-type "Microsoft.Web/sites" \
```

```
--set  
properties.siteConfig.scmIpSecurityRestrictionsDefaultAction=Allow
```

Set up Azure Functions access restrictions

Access restrictions are also available for function apps with the same functionality as App Service plans. When you enable access restrictions, you also disable the Azure portal code editor for any disallowed IPs.

Next steps

[Access restrictions for Azure Functions](#)

[Application Gateway integration with service endpoints](#)

[Advanced access restriction scenarios in Azure App Service - blog post ↗](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

How to use managed identities for App Service and Azure Functions

Article • 09/30/2024

ⓘ Note

Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention `<app-name>-<random-hash>. <region>.azurewebsites.net`. Existing app names will remain unchanged.

Example: `myapp-ds27dh7271ah175.westus-01.azurewebsites.net`

For further details, refer to [Unique Default Hostname for App Service Resource ↗](#).

This article shows you how to create a managed identity for App Service and Azure Functions applications and how to use it to access other resources.

ⓘ Important

Because [managed identities don't support cross-directory scenarios](#), they won't behave as expected if your app is migrated across subscriptions or tenants. To recreate the managed identities after such a move, see [Will managed identities be recreated automatically if I move a subscription to another directory?](#).

Downstream resources also need to have access policies updated to use the new identity.

ⓘ Note

Managed identities are not available for [apps deployed in Azure Arc](#).

A managed identity from Microsoft Entra ID allows your app to easily access other Microsoft Entra protected resources such as Azure Key Vault. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. For more about managed identities in Microsoft Entra ID, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to the app and is deleted if the app is deleted.
An app can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your app. An app can have multiple user-assigned identities, and one user-assigned identity can be assigned to multiple Azure resources, such as two App Service apps.

The managed identity configuration is specific to the slot. To configure a managed identity for a deployment slot in the portal, navigate to the slot first. To find the managed identity for your web app or deployment slot in your Microsoft Entra tenant from the Azure portal, search for it directly from the **Overview** page of your tenant. Usually, the slot name is similar to <app-name>/slots/<slot-name>.

This video shows you how to use managed identities for App Service.

[https://learn-video.azurefd.net/vod/player?id=4fdf7a78-b3ce-48df-b3ce-cd7796d0ad5a&locale=en-us&embedUrl=%2Fazure%2Fapp-service%2Foverview-managed-identity ↗](https://learn-video.azurefd.net/vod/player?id=4fdf7a78-b3ce-48df-b3ce-cd7796d0ad5a&locale=en-us&embedUrl=%2Fazure%2Fapp-service%2Foverview-managed-identity)

The steps in the video are also described in the following sections.

Add a system-assigned identity

Azure portal

1. Access your app's settings in the [Azure portal](#) ↗ under the **Settings** group in the left navigation pane.
2. Select **Identity**.
3. Within the **System assigned** tab, switch **Status** to **On**. Click **Save**.

The screenshot shows the Azure portal interface for managing a resource's identity. On the left, there's a navigation menu with various options like Deployment, Settings, and Identity. The 'Identity' option under Settings is highlighted with a red box. The main content area shows two tabs at the top: 'System assigned' (which is selected) and 'User assigned'. Below this, there's a note about system assigned managed identities being restricted to one per resource. There are buttons for Save, Discard, Refresh, and Got feedback?.

Add a user-assigned identity

Creating an app with a user-assigned identity requires that you create the identity and then add its resource identifier to your app config.

The screenshot shows the Azure portal for a specific application named 'my-demo-app'. In the left sidebar, the 'Identity' option is highlighted with a red box. The main page shows the 'User assigned' tab selected. A modal window titled 'Add user assigned managed i...' is open, prompting the user to select a subscription (Visual Studio Enterprise Subscription) and a user-assigned managed identity (test). The 'test' identity is selected and highlighted with a red box. At the bottom of the modal, there's an 'Add' button highlighted with a red box.

Once you select **Add**, the app restarts.

Configure target resource

You may need to configure the target resource to allow access from your app or function. For example, if you [request a token](#) to access Key Vault, you must also add an access policy that includes the managed identity of your app or function. Otherwise, your calls to Key Vault will be rejected, even if you use a valid token. The same is true for Azure SQL Database. To learn more about which resources support Microsoft Entra tokens, see [Azure services that support Microsoft Entra authentication](#).

Important

The back-end services for managed identities maintain a cache per resource URI for around 24 hours. If you update the access policy of a particular target resource and immediately retrieve a token for that resource, you may continue to get a cached token with outdated permissions until that token expires. There's currently no way to force a token refresh.

Connect to Azure services in app code

With its managed identity, an app can obtain tokens for Azure resources that are protected by Microsoft Entra ID, such as Azure SQL Database, Azure Key Vault, and Azure Storage. These tokens represent the application accessing the resource, and not any specific user of the application.

App Service and Azure Functions provide an internally accessible [REST endpoint](#) for token retrieval. The REST endpoint can be accessed from within the app with a standard HTTP GET, which can be implemented with a generic HTTP client in every language. For .NET, JavaScript, Java, and Python, the Azure Identity client library provides an abstraction over this REST endpoint and simplifies the development experience. Connecting to other Azure services is as simple as adding a credential object to the service-specific client.

HTTP GET

A raw HTTP GET request looks like the following example:

HTTP

```
GET /MSI/token?resource=https://vault.azure.net&api-version=2019-08-01
HTTP/1.1
Host: localhost:4141
X-IDENTITY-HEADER: 853b9a84-5bfa-4b22-a3f3-0b9a43d9ad8a
```

And a sample response might look like the following:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "access_token": "eyJ0eXAi...",
    "expires_on": "1586984735",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer",
    "client_id": "00001111-aaaa-2222-bbbb-3333cccc4444"
}
```

This response is the same as the [response for the Microsoft Entra service-to-service access token request](#). To access Key Vault, you will then add the value of `access_token` to a client connection with the vault.

For more information on the REST endpoint, see [REST endpoint reference](#).

Remove an identity

When you remove a system-assigned identity, it's deleted from Microsoft Entra ID. System-assigned identities are also automatically removed from Microsoft Entra ID when you delete the app resource itself.

Azure portal

1. In the left navigation of your app's page, scroll down to the **Settings** group.
2. Select **Identity**. Then follow the steps based on the identity type:
 - **System-assigned identity**: Within the **System assigned** tab, switch **Status** to **Off**. Click **Save**.
 - **User-assigned identity**: Select the **User assigned** tab, select the checkbox for the identity, and select **Remove**. Select **Yes** to confirm.

Note

There is also an application setting that can be set, WEBSITE_DISABLE_MSI, which just disables the local token service. However, it leaves the identity in place, and tooling will still show the managed identity as "on" or "enabled." As a result, use of this setting is not recommended.

REST endpoint reference

An app with a managed identity makes this endpoint available by defining two environment variables:

- **IDENTITY_ENDPOINT** - the URL to the local token service.
- **IDENTITY_HEADER** - a header used to help mitigate server-side request forgery (SSRF) attacks. The value is rotated by the platform.

The **IDENTITY_ENDPOINT** is a local URL from which your app can request tokens. To get a token for a resource, make an HTTP GET request to this endpoint, including the following parameters:

 Expand table

Parameter	In	Description
name		
resource	Query	The Microsoft Entra resource URI of the resource for which a token should be obtained. This could be one of the Azure services that support Microsoft Entra authentication or any other resource URI.
api-version	Query	The version of the token API to be used. Use <code>2019-08-01</code> .
X-IDENTITY-HEADER	Header	The value of the IDENTITY_HEADER environment variable. This header is used to help mitigate server-side request forgery (SSRF) attacks.
client_id	Query	(Optional) The client ID of the user-assigned identity to be used. Cannot be used on a request that includes <code>principal_id</code> , <code>mi_res_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.
principal_id	Query	(Optional) The principal ID of the user-assigned identity to be used. <code>object_id</code> is an alias that may be used instead. Cannot be used on a request that includes <code>client_id</code> , <code>mi_res_id</code> , or <code>object_id</code> . If all ID

Parameter	In	Description
name		parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.
mi_res_id	Query	(Optional) The Azure resource ID of the user-assigned identity to be used. Cannot be used on a request that includes <code>principal_id</code> , <code>client_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.

ⓘ Important

If you are attempting to obtain tokens for user-assigned identities, you must include one of the optional properties. Otherwise the token service will attempt to obtain a token for a system-assigned identity, which may or may not exist.

Next steps

- [Tutorial: Connect to SQL Database from App Service without secrets using a managed identity](#)
- [Access Azure Storage securely using a managed identity](#)
- [Call Microsoft Graph securely using a managed identity](#)
- [Connect securely to services with Key Vault secrets](#)

ⓘ Note: The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Use Key Vault references as app settings in Azure App Service and Azure Functions

Article • 09/30/2024

ⓘ Note

Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention <app-name>-<random-hash>. <region>.azurewebsites.net. Existing app names will remain unchanged.

Example: myapp-ds27dh7271aah175.westus-01.azurewebsites.net

For further details, refer to [Unique Default Hostname for App Service Resource ↗](#).

This article shows you how to use secrets from Azure Key Vault as values of [app settings](#) or [connection strings](#) in your App Service or Azure Functions apps.

[Azure Key Vault](#) is a service that provides centralized secrets management, with full control over access policies and audit history. When an app setting or connection string is a key vault reference, your application code can use it like any other app setting or connection string. This way, you can maintain secrets apart from your app's configuration. App settings are securely encrypted at rest, but if you need secret management capabilities, they should go into a key vault.

Grant your app access to a key vault

In order to read secrets from a key vault, you need to have a vault created and give your app permission to access it.

1. Create a key vault by following the [Key Vault quickstart](#).
2. Create a [managed identity](#) for your application.

Key vault references use the app's system-assigned identity by default, but you can [specify a user-assigned identity](#).

3. Authorize [read access to secrets in your key vault](#) for the managed identity you created earlier. How you do it depends on the permissions model of your key

vault:

- **Azure role-based access control:** Assign the **Key Vault Secrets User** role to the managed identity. For instructions, see [Provide access to Key Vault keys, certificates, and secrets with an Azure role-based access control](#).
- **Vault access policy:** Assign the **Get** secrets permission to the managed identity. For instructions, see [Assign a Key Vault access policy](#).

Access network-restricted vaults

If your vault is configured with [network restrictions](#), ensure that the application has network access. Vaults shouldn't depend on the app's public outbound IPs because the origin IP of the secret request could be different. Instead, the vault should be configured to accept traffic from a virtual network used by the app.

1. Make sure the application has outbound networking capabilities configured, as described in [App Service networking features](#) and [Azure Functions networking options](#).

Linux applications that connect to private endpoints must be explicitly configured to route all traffic through the virtual network. This requirement will be removed in a forthcoming update. To configure this setting, run the following command:



```
az webapp config set --subscription <sub> -g <group-name> -n <app-name> --generic-configurations '{"vnetRouteAllEnabled": true}'
```

2. Make sure that the vault's configuration allows the network or subnet that your app uses to access it.

Access vaults with a user-assigned identity

Some apps need to reference secrets at creation time, when a system-assigned identity isn't available yet. In these cases, a user-assigned identity can be created and given access to the vault in advance.

Once you have granted permissions to the user-assigned identity, follow these steps:

1. [Assign the identity](#) to your application if you haven't already.

2. Configure the app to use this identity for key vault reference operations by setting the `keyVaultReferenceIdentity` property to the resource ID of the user-assigned identity.

```
Azure CLI

identityResourceId=$(az identity show --resource-group <group-name>
--name <identity-name> --query id -o tsv)
az webapp update --resource-group <group-name> --name <app-name> --
set keyVaultReferenceIdentity=${identityResourceId}
```

This setting applies to all key vault references for the app.

Rotation

If the secret version isn't specified in the reference, the app uses the latest version that exists in the key vault. When newer versions become available, such as with a rotation event, the app automatically updates and begins using the latest version within 24 hours. The delay is because App Service caches the values of the key vault references and refetches it every 24 hours. Any configuration change to the app causes an app restart and an immediate refetch of all referenced secrets.

Source app settings from key vault

To use a key vault reference, set the reference as the value of the setting. Your app can reference the secret through its key as normal. No code changes are required.

Tip

Most app settings using key vault references should be marked as slot settings, as you should have separate vaults for each environment.

A key vault reference is of the form `@Microsoft.KeyVault({referenceString})`, where `{referenceString}` is in one of the following formats:

[+] Expand table

Reference string	Description
SecretUri= <i>secretUri</i>	The SecretUri should be the full data-plane URI of a secret in the vault, optionally including a version, e.g., <code>https://myvault.vault.azure.net/secrets/mysecret/</code> or <code>https://myvault.vault.azure.net/secrets/mysecret/ec96f02080254f109c51a1f14cdb1931</code>
VaultName= <i>vaultName</i> ;SecretName= <i>secretName</i> ;SecretVersion= <i>secretVersion</i>	The VaultName is required and is the vault name. The SecretName is required and is the secret name. The SecretVersion is optional but if present indicates the version of the secret to use.

For example, a complete reference would look like the following string:

```
@Microsoft.KeyVault(SecretUri=https://myvault.vault.azure.net/secrets/mysecret/)
```

Alternatively:

```
@Microsoft.KeyVault(VaultName=myvault;SecretName=mysecret)
```

Considerations for Azure Files mounting

Apps can use the `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` application setting to mount [Azure Files](#) as the file system. This setting has validation checks to ensure that the app can be properly started. The platform relies on having a content share within Azure Files, and it assumes a default name unless one is specified via the `WEBSITE_CONTENTSHARE` setting. For any requests that modify these settings, the platform validates if this content share exists, and attempts to create it if not. If it can't locate or create the content share, it blocks the request.

When you use key vault references in this setting, the validation check fails by default, because the secret itself can't be resolved while processing the incoming request. To avoid this issue, you can skip the validation by setting

`WEBSITE_SKIP_CONTENTSHARE_VALIDATION` to "1". This setting tells App Service to bypass all checks, and doesn't create the content share for you. You should ensure that it's created in advance.

 **Caution**

If you skip validation and either the connection string or content share are invalid, the app will be unable to start properly and will only serve HTTP 500 errors.

As part of creating the app, attempted mounting of the content share could fail due to managed identity permissions not being propagated or the virtual network integration not being set up. You can defer setting up Azure Files until later in the deployment template to accommodate this. See [Azure Resource Manager deployment](#) to learn more. In this case, App Service uses a default file system until Azure Files is set up, and files aren't copied over. You must ensure that no deployment attempts occur during the interim period before Azure Files is mounted.

Considerations for Application Insights instrumentation

Apps can use the `APPINSIGHTS_INSTRUMENTATIONKEY` or `APPLICATIONINSIGHTS_CONNECTION_STRING` application settings to integrate with [Application Insights](#). The portal experiences for App Service and Azure Functions also use these settings to surface telemetry data from the resource. If these values are referenced from Key Vault, these experiences aren't available, and you instead need to work directly with the Application Insights resource to view the telemetry. However, these values are [not considered secrets](#), so you might alternatively consider configuring them directly instead of using key vault references.

Azure Resource Manager deployment

When automating resource deployments through Azure Resource Manager templates, you may need to sequence your dependencies in a particular order to make this feature work. Be sure to define your app settings as their own resource, rather than using a `siteConfig` property in the app definition. This is because the app needs to be defined first so that the system-assigned identity is created with it and can be used in the access policy.

The following pseudo-template is an example of what a function app might look like:

JSON

```
{  
  //...  
  "resources": [  
    {  
      "type": "Microsoft.Storage/storageAccounts",  
      "name": "[variables('storageAccountName')]",  
      //...  
    },
```

```

    },
    "type": "Microsoft.Insights/components",
    "name": "[variables('appInsightsName')]",
    //...
},
{
    "type": "Microsoft.Web/sites",
    "name": "[variables('functionAppName')]",
    "identity": {
        "type": "SystemAssigned"
    },
    //...
    "resources": [
        {
            "type": "config",
            "name": "appsettings",
            //...
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites',
variables('functionAppName'))]",
                "[resourceId('Microsoft.KeyVault/vaults/',
variables('keyVaultName'))]",
                "[resourceId('Microsoft.KeyVault/vaults/secrets',
variables('keyVaultName'), variables('storageConnectionStringName'))]",
                "[resourceId('Microsoft.KeyVault/vaults/secrets',
variables('keyVaultName'), variables('appInsightsKeyName'))]"
            ],
            "properties": {
                "AzureWebJobsStorage": "[concat('@Microsoft.KeyVault(SecretUri=',
reference(variables('storageConnectionStringName')).secretUriWithVersion,
'))]]",
                "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING": "[concat('@Microsoft.KeyVault(SecretUri=',
reference(variables('storageConnectionStringName')).secretUriWithVersion,
'))]]",
                "APPINSIGHTS_INSTRUMENTATIONKEY": "[concat('@Microsoft.KeyVault(SecretUri=',
reference(variables('appInsightsKeyName')).secretUriWithVersion, '))]]",
                "WEBSITE_ENABLE_SYNC_UPDATE_SITE": "true"
            }
        },
        {
            "type": "sourcecontrols",
            "name": "web",
            //...
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites',
variables('functionAppName'))]",
                "[resourceId('Microsoft.Web/sites/config',
variables('functionAppName'), 'appsettings')]"
            ],
        }
    ]
}

```

```

},
{
  "type": "Microsoft.KeyVault/vaults",
  "name": "[variables('keyVaultName')]",
  //...
  "dependsOn": [
    "[resourceId('Microsoft.Web/sites',
variables('functionAppName'))]"
  ],
  "properties": {
    //...
    "accessPolicies": [
      {
        "tenantId": "
[reference(resourceId('Microsoft.Web/sites/'), variables('functionAppName')),
'2020-12-01', 'Full').identity.tenantId]",
        "objectId": "
[reference(resourceId('Microsoft.Web/sites/'), variables('functionAppName')),
'2020-12-01', 'Full').identity.principalId]",
        "permissions": {
          "secrets": [ "get" ]
        }
      }
    ]
  },
  "resources": [
    {
      "type": "secrets",
      "name": "[variables('storageConnectionStringName')]",
      //...
      "dependsOn": [
        "[resourceId('Microsoft.KeyVault/vaults/',
variables('keyVaultName'))]",
        "[resourceId('Microsoft.Storage/storageAccounts',
variables('storageAccountName'))]"
      ],
      "properties": {
        "value": "
[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=',
listKeys(variables('storageAccountResourceId'),'2019-09-01').key1)]"
      }
    },
    {
      "type": "secrets",
      "name": "[variables('appInsightsKeyName')]",
      //...
      "dependsOn": [
        "[resourceId('Microsoft.KeyVault/vaults/',
variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components',
variables('appInsightsName'))]"
      ],
      "properties": {
        "value": "

```

```
[reference(resourceId('microsoft.insights/components/'),
variables('appInsightsName')), '2019-09-01').InstrumentationKey]"
    }
]
}
]
```

ⓘ Note

In this example, the source control deployment depends on the application settings. This is normally unsafe behavior, as the app setting update behaves asynchronously. However, because we have included the `WEBSITE_ENABLE_SYNC_UPDATE_SITE` application setting, the update is synchronous. This means that the source control deployment will only begin once the application settings have been fully updated. For more app settings, see [Environment variables and app settings in Azure App Service](#).

Troubleshooting key vault references

If a reference isn't resolved properly, the reference string is used instead (for example, `@Microsoft.KeyVault(...)`). It may cause the application to throw errors, because it's expecting a secret of a different value.

Failure to resolve is commonly due to a misconfiguration of the [Key Vault access policy](#). However, it could also be due to a secret no longer existing or a syntax error in the reference itself.

If the syntax is correct, you can view other causes for error by checking the current resolution status in the portal. Navigate to Application Settings and select "Edit" for the reference in question. The edit dialog shows status information, including any errors. If you don't see the status message, it means that the syntax is invalid and not recognized as a key vault reference.

You can also use one of the built-in detectors to get additional information.

Using the detector for App Service

1. In the portal, navigate to your app.
2. Select **Diagnose and solve problems**.
3. Choose **Availability and Performance** and select **Web app down**.

4. In the search box, search for and select **Key Vault Application Settings Diagnostics**.

Using the detector for Azure Functions

1. In the portal, navigate to your app.
 2. Navigate to **Platform features**.
 3. Select **Diagnose and solve problems**.
 4. Choose **Availability and Performance** and select **Function app down or reporting errors**.
 5. Select **Key Vault Application Settings Diagnostics**.
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Encrypt your application data at rest using customer-managed keys

Article • 03/24/2022

Encrypting your function app's application data at rest requires an Azure Storage Account and an Azure Key Vault. These services are used when you run your app from a deployment package.

- [Azure Storage provides encryption at rest](#). You can use system-provided keys or your own, customer-managed keys. This is where your application data is stored when it's not running in a function app in Azure.
- [Running from a deployment package](#) is a deployment feature of App Service. It allows you to deploy your site content from an Azure Storage Account using a Shared Access Signature (SAS) URL.
- [Key Vault references](#) are a security feature of App Service. It allows you to import secrets at runtime as application settings. Use this to encrypt the SAS URL of your Azure Storage Account.

Set up encryption at rest

Create an Azure Storage account

First, [create an Azure Storage account](#) and [encrypt it with customer managed keys](#). Once the storage account is created, use the [Azure Storage Explorer](#) to upload package files.

Next, use the Storage Explorer to [generate an SAS](#).

 Note

Save this SAS URL, this is used later to enable secure access of the deployment package at runtime.

Configure running from a package from your storage account

Once you upload your file to Blob storage and have an SAS URL for the file, set the `WEBSITE_RUN_FROM_PACKAGE` application setting to the SAS URL. The following example does it by using Azure CLI:

```
az webapp config appsettings set --name <app-name> --resource-group <resource-group-name> --settings WEBSITE_RUN_FROM_PACKAGE="<your-SAS-URL>"
```

Adding this application setting causes your function app to restart. After the app has restarted, browse to it and make sure that the app has started correctly using the deployment package. If the application didn't start correctly, see the [Run from package troubleshooting guide](#).

Encrypt the application setting using Key Vault references

Now you can replace the value of the `WEBSITE_RUN_FROM_PACKAGE` application setting with a Key Vault reference to the SAS-encoded URL. This keeps the SAS URL encrypted in Key Vault, which provides an extra layer of security.

1. Use the following [az keyvault create](#) command to create a Key Vault instance.

Azure CLI

```
az keyvault create --name "Contoso-Vault" --resource-group <group-name> --location eastus
```

2. Follow [these instructions to grant your app access](#) to your key vault:

3. Use the following [az keyvault secret set](#) command to add your external URL as a secret in your key vault:

Azure CLI

```
az keyvault secret set --vault-name "Contoso-Vault" --name "external-url" --value "<SAS-URL>"
```

4. Use the following [az webapp config appsettings set](#) command to create the `WEBSITE_RUN_FROM_PACKAGE` application setting with the value as a Key Vault reference to the external URL:

Azure CLI

```
az webapp config appsettings set --settings WEBSITE_RUN_FROM_PACKAGE="@Microsoft.KeyVault(SecretUri=https://Contoso-Vault.vault.azure.net/secrets/external-url/<secret-version>")"
```

The `<secret-version>` will be in the output of the previous `az keyvault secret set` command.

Updating this application setting causes your function app to restart. After the app has restarted, browse to it make sure it has started correctly using the Key Vault reference.

How to rotate the access token

It is best practice to periodically rotate the SAS key of your storage account. To ensure the function app does not inadvertently lose access, you must also update the SAS URL in Key Vault.

1. Rotate the SAS key by navigating to your storage account in the Azure portal.
Under **Settings > Access keys**, click the icon to rotate the SAS key.
2. Copy the new SAS URL, and use the following command to set the updated SAS URL in your key vault:

Azure CLI

```
az keyvault secret set --vault-name "Contoso-Vault" --name "external-url" --value "<SAS-URL>"
```

3. Update the key vault reference in your application setting to the new secret version:

Azure CLI

```
az webapp config appsettings set --settings  
WEBSITE_RUN_FROM_PACKAGE="@Microsoft.KeyVault(SecretUri=https://Contoso  
-Vault.vault.azure.net/secrets/external-url/<secret-version>")
```

The `<secret-version>` will be in the output of the previous `az keyvault secret set` command.

How to revoke the function app's data access

There are two methods to revoke the function app's access to the storage account.

Rotate the SAS key for the Azure Storage account

If the SAS key for the storage account is rotated, the function app will no longer have access to the storage account, but it will continue to run with the last downloaded version of the package file. Restart the function app to clear the last downloaded version.

Remove the function app's access to Key Vault

You can revoke the function app's access to the site data by disabling the function app's access to Key Vault. To do this, remove the access policy for the function app's identity. This is the same identity you created earlier while configuring key vault references.

Summary

Your application files are now encrypted at rest in your storage account. When your function app starts, it retrieves the SAS URL from your key vault. Finally, the function app loads the application files from the storage account.

If you need to revoke the function app's access to your storage account, you can either revoke access to the key vault or rotate the storage account keys, which invalidates the SAS URL.

Frequently Asked Questions

Is there any additional charge for running my function app from the deployment package?

Only the cost associated with the Azure Storage Account and any applicable egress charges.

How does running from the deployment package affect my function app?

- Running your app from the deployment package makes `wwwroot/` read-only. Your app receives an error when it attempts to write to this directory.
- TAR and GZIP formats are not supported.
- This feature is not compatible with local cache.

Next steps

- Key Vault references for App Service
- Azure Storage encryption for data at rest

Store unstructured data using Azure Functions and Azure Cosmos DB

Article • 10/12/2022

Azure Cosmos DB [↗](#) is a great way to store unstructured and JSON data. Combined with Azure Functions, Azure Cosmos DB makes storing data quick and easy with much less code than required for storing data in a relational database.

ⓘ Note

At this time, the Azure Cosmos DB trigger, input bindings, and output bindings work with SQL API and Graph API accounts only.

In Azure Functions, input and output bindings provide a declarative way to connect to external service data from your function. In this article, learn how to update an existing function to add an output binding that stores unstructured data in an Azure Cosmos DB document.

Prerequisites

To complete this tutorial:

This topic uses as its starting point the resources created in [Create your first function from the Azure portal](#). If you haven't already done so, please complete these steps now to create your function app.

Create an Azure Cosmos DB account

You must have an Azure Cosmos DB account that uses the SQL API before you create the output binding.

1. From the Azure portal menu or the [Home page](#), select **Create a resource**.
2. Search for **Azure Cosmos DB**. Select **Create > Azure Cosmos DB**.
3. On the **Create an Azure Cosmos DB account** page, select the **Create** option within the **Azure Cosmos DB for NoSQL** section.

Azure Cosmos DB provides several APIs:

- NoSQL, for document data
- PostgreSQL
- MongoDB, for document data
- Apache Cassandra
- Table
- Apache Gremlin, for graph data

To learn more about the API for NoSQL, see [Welcome to Azure Cosmos DB](#).

4. In the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos DB account.

Setting	Value	Description
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Resource group name	Select a resource group, or select Create new , then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos DB account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URL, use a unique name. The name can contain only lowercase letters, numbers, and the hyphen (-) character. It must be 3-44 characters.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select Provisioned throughput to create an account in provisioned throughput mode. Select Serverless to create an account in serverless mode.
Apply Azure Cosmos DB free tier discount	Apply or Do not apply	With Azure Cosmos DB free tier, you get the first 1000 RU/s and 25 GB of storage for free in an account. Learn more about free tier .
Limit total account throughput	Selected or not	Limit the total amount of throughput that can be provisioned on this account. This limit prevents unexpected charges related to provisioned throughput. You can update or remove this limit after your account is created.

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt in when creating the account. If you don't see the option to apply

the free tier discount, another account in the subscription has already been enabled with free tier.

Home > Marketplace > Create an Azure Cosmos DB account >

Create Azure Cosmos DB Account - Azure Cosmos DB for NoSQL

Basics Global Distribution Networking Backup Policy Encryption Tags Review + create

Azure Cosmos DB is a fully managed NoSQL and relational database service for building scalable, high performance applications. Try it for free, for 30 days with unlimited renewals. Go to production starting at \$24/month per database, multiple containers included. [Learn more](#)

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Contoso Subscription

Resource Group * myResourceGroup [Create new](#)

Instance Details

Account Name * mysqlaccount

Location * (US) East US 2

Capacity mode ⓘ Provisioned throughput Serverless [Learn more about capacity mode](#)

With Azure Cosmos DB free tier, you will get the first 1000 RU/s and 25 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$64/month discount per account.

Apply Free Tier Discount Apply Do Not Apply

Limit total account throughput Limit the total amount of throughput that can be provisioned on this account
 ⓘ This limit will prevent unexpected charges related to provisioned throughput. You can update or remove this limit after your account is created.

[Review + create](#) [Previous](#) [Next: Global Distribution](#)

ⓘ Note

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Limit total account throughput

5. In the **Global Distribution** tab, configure the following details. You can leave the default values for this quickstart:

Setting	Value	Description
Geo-Redundancy	Disable	Enable or disable global distribution on your account by pairing your region with a pair region. You can add more regions to your account later.

Setting	Value	Description
Multi-region Writes	Disable	Multi-region writes capability allows you to take advantage of the provisioned throughput for your databases and containers across the globe.
Availability Zones	Disable	Availability Zones help you further improve availability and resiliency of your application.

① Note

The following options are not available if you select **Serverless** as the **Capacity mode** in the previous **Basics** page:

- Geo-redundancy
- Multi-region Writes

6. Optionally, you can configure more details in the following tabs:

- **Networking**. Configure [access from a virtual network](#).
- **Backup Policy**. Configure either [periodic](#) or [continuous](#) backup policy.
- **Encryption**. Use either service-managed key or a [customer-managed key](#).
- **Tags**. Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups.

7. Select **Review + create**.

8. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

The screenshot shows the Azure Deployment Overview page for a deployment named "Microsoft.Azure.CosmosDB-20230302095414". The status is "Your deployment is complete". Deployment details include the name, subscription ("Contoso Subscription"), resource group ("myResourceGroup"), start time ("3/2/2023, 9:54:17 AM"), and correlation ID ("d7e2b2c3-580b-4e8b-ba8d-cc5d5dd8cd3b"). A "Go to resource" button is visible at the bottom.

9. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the Azure Cosmos DB account Quick start page for "mysqlaccount". It displays a message: "Congratulations! Your Azure Cosmos DB account was created." It provides instructions to "Choose a platform" (.NET, Java, Node.js, Python) and lists two steps: "Step 1: Add a container" and "Step 2: Download and run your .NET app".

Add an output binding

1. In the Azure portal, navigate to and select the function app you created previously.
2. Select **Functions**, and then select the HttpTrigger function.

Home > Function App > myFunctionApp-dma > myFunctionApp-dma | Functions

myFunctionApp-dma | Functions

App Service

+ Add Develop Locally Refresh Enable Disable Delete

Search (Ctrl+/
)

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Security

Functions (preview)

Functions (highlighted)

App keys App files

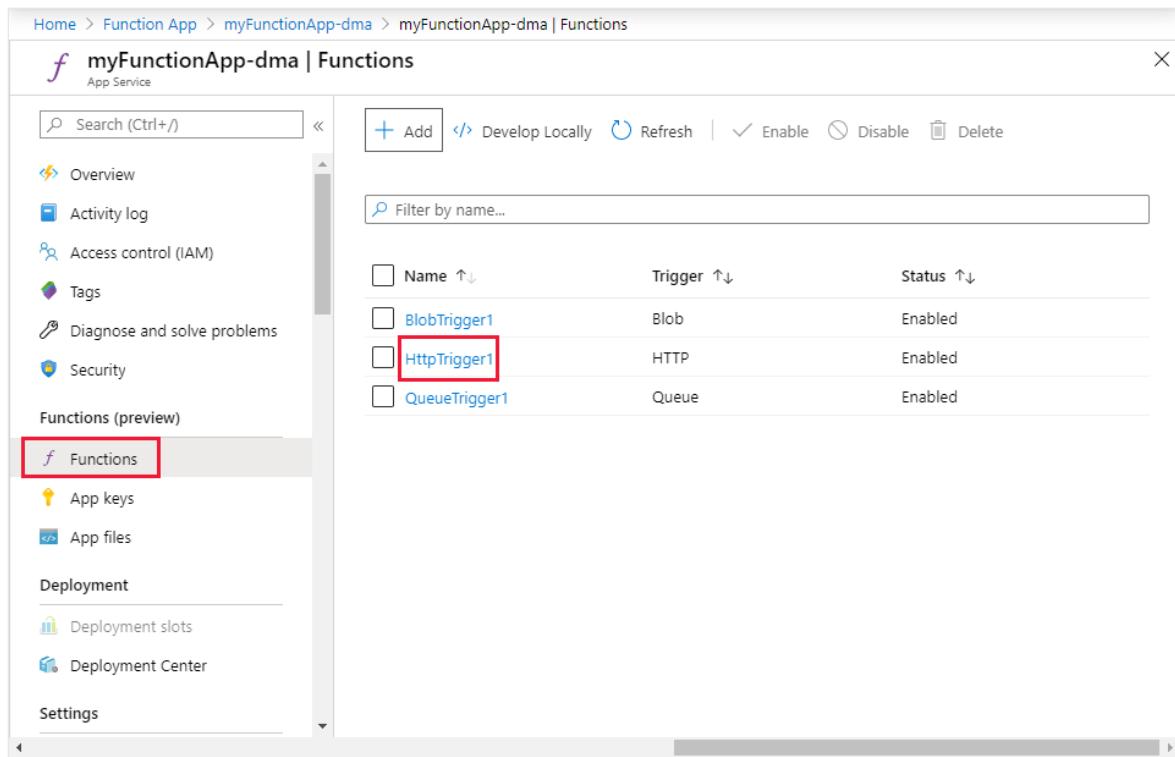
Deployment

Deployment slots Deployment Center

Settings

Name ↑↓	Trigger ↑↓	Status ↑↓
BlobTrigger1	Blob	Enabled
HttpTrigger1	HTTP	Enabled
QueueTrigger1	Queue	Enabled

Filter by name...



3. Select Integration and + Add output.

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

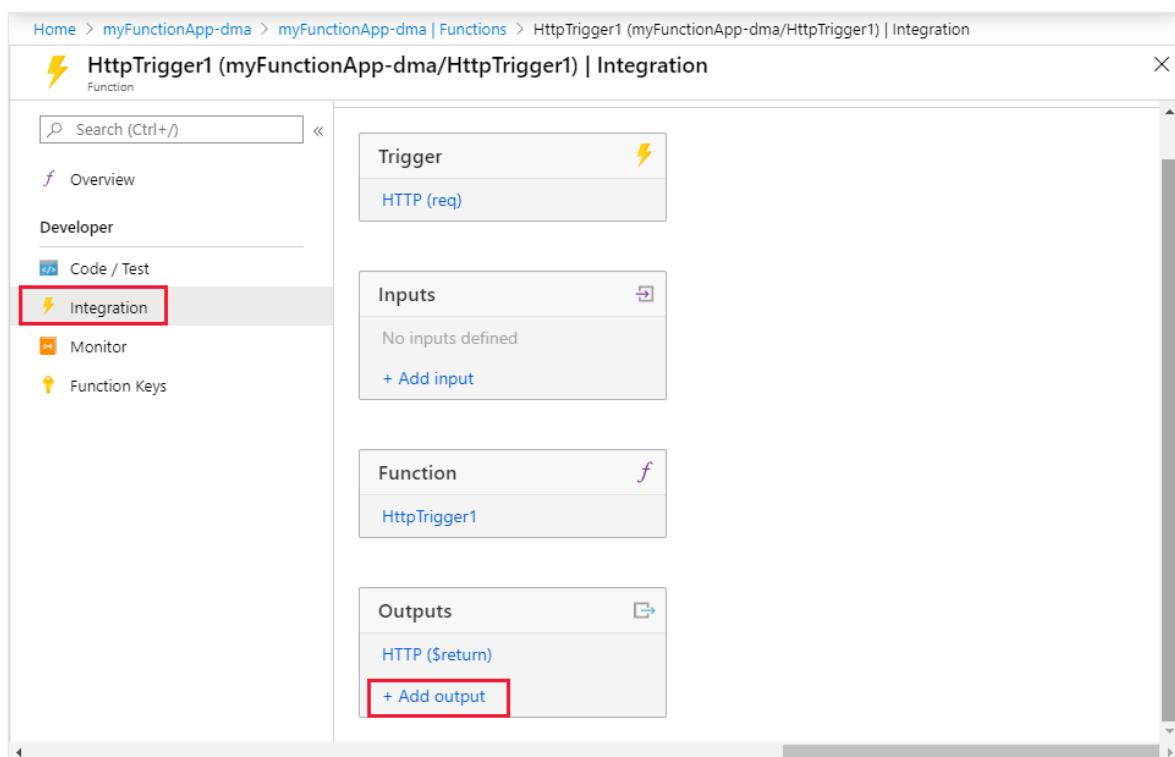
HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

Trigger  HTTP (req)

Inputs 
No inputs defined
+ Add input

Function 
HttpTrigger1

Outputs 
HTTP (\$return)
+ Add output (highlighted with a red box)



4. Use the Create Output settings as specified in the table:

Create Output

X

Start by selecting the type of output binding you want to add.

Binding Type

Azure Cosmos DB

Azure Cosmos DB details

Document parameter name* ⓘ

taskDocument

Database name* ⓘ

taskDatabase

Collection Name* ⓘ

taskCollection

If true, creates the Cosmos DB databas...* ⓘ

Yes

Cosmos DB account connection* ⓘ

dma-cosmosdb_DOCUMENTDB

New

Partition key (optional) ⓘ

OK

Cancel

Setting	Suggested value	Description
Binding Type	Azure Cosmos DB	Name of the binding type to select to create the output binding to Azure Cosmos DB.
Document parameter name	taskDocument	Name that refers to the Azure Cosmos DB object in code.
Database name	taskDatabase	Name of database to save documents.

Setting	Suggested value	Description
Collection name	taskCollection	Name of the database collection.
If true, creates the Azure Cosmos DB database and collection	Yes	The collection doesn't already exist, so create it.
Azure Cosmos DB account connection	New setting	Select New , then choose Azure Cosmos DB Account and the Database account you created earlier, and then select OK . Creates an application setting for your account connection. This setting is used by the binding to connection to the database.

5. Select **OK** to create the binding.

Update the function code

Replace the existing function code with the following code, in your chosen language:

C#

Replace the existing C# function with the following code:

```
C#  
  

#r "Newtonsoft.Json"  
  

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;  
  

public static IActionResult Run(HttpContext req, out object taskDocument, ILogger log)
{
    string name = req.Query["name"];
    string task = req.Query["task"];
    string duedate = req.Query["duedate"];  
  

    // We need both name and task parameters.
    if (!string.IsNullOrEmpty(name) && !string.IsNullOrEmpty(task))
    {
        taskDocument = new
        {
            name,
```

```

        duedate,
        task
    };

    return (ActionResult)new OkResult();
}
else
{
    taskDocument = null;
    return (ActionResult)new BadRequestResult();
}
}

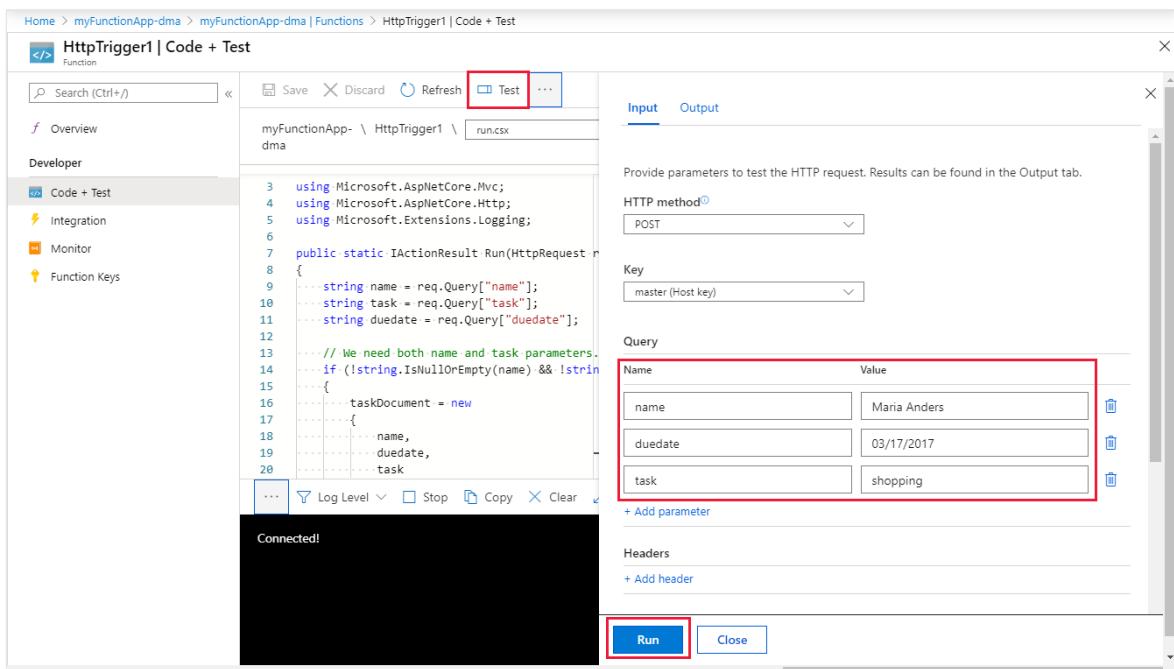
```

This code sample reads the HTTP Request query strings and assigns them to fields in the `taskDocument` object. The `taskDocument` binding sends the object data from this binding parameter to be stored in the bound document database. The database is created the first time the function runs.

Test the function and database

1. Select **Test/Run**. Under **Query**, select **+ Add parameter** and add the following parameters to the query string:

- `name`
- `task`
- `duedate`



2. Select **Run** and verify that a 200 status is returned.

```

9     string name = req.Query["name"];
10    string task = req.Query["task"];
11    string duedate = req.Query["duedate"];
12
13    // We need both name and task parameters.
14    if (!string.IsNullOrEmpty(name) && !string.IsNullOrEmpty(task))
15    {
16        taskDocument = new
17        {
18            name,
19            duedate,
20            task
21        };
22
23        return (ActionResult)new OkResult();
24    }
25    else

```

Connected!
2020-04-14T05:10:00Z [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=ce811ba9-dfba-42d1-a0cb-a7caed0efc5)
2020-04-14T05:09:59Z [Information] Executing 'Functions.HttpTrigger1' (Reason="This function was programmatically called via the host APIs.", Id=ce811ba9-dfba-42d1-a0cb-a7caed0efc5)

3. In the Azure portal, search for and select Azure Cosmos DB.

Azure Cosmos DB

Services See all

- Azure Cosmos DB**
- Azure Database for MySQL servers
- Azure Arc
- Azure Databricks
- Azure DevOps
- Azure Lighthouse
- Azure Migrate
- Azure Sentinel
- Azure SQL
- Azure Database for MariaDB servers

Resources

No results were found.

4. Choose your Azure Cosmos DB account, then select Data Explorer.

5. Expand the **TaskCollection** nodes, select the new document, and confirm that the document contains your query string values, along with some additional metadata.

1	"name": "Marie",
2	"dueDate": "4/20/20",
3	"task": "shopping",
4	"_id": "32bede08a-2b0c-4f1e-8fc2-eae87f36b5a3",
5	"_rid": "272baPhKu48BA==AAAAAAA=",
6	"_self": "dbs/272baPhKu48=/colls/272baPhKu48=/docs/272baPhKu48=/",
7	"_etag": "\\"1b0114c8-0000-0500-0000-5e95426b0000\\\"",
8	"_attachments": "attachments/",
9	"_ts": 1586840171
10	
11	

You've successfully added a binding to your HTTP trigger to store unstructured data in an Azure Cosmos DB instance.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

For more information about binding to an Azure Cosmos DB instance, see [Azure Functions Azure Cosmos DB bindings](#).

- [Azure Functions triggers and bindings concepts](#)
Learn how Functions integrates with other services.

- [Azure Functions developer reference](#)

Provides more technical information about the Functions runtime and a reference for coding functions and defining triggers and bindings.

- [Code and test Azure Functions locally](#)

Describes the options for developing your functions locally.

Tutorial: Get started with Azure Functions triggers and bindings in Azure Cache for Redis

Article • 04/14/2024

This tutorial shows how to implement basic triggers with Azure Cache for Redis and Azure Functions. It guides you through using Visual Studio Code (VS Code) to write and deploy an Azure function in C#.

In this tutorial, you learn how to:

- ✓ Set up the necessary tools.
- ✓ Configure and connect to a cache.
- ✓ Create an Azure function and deploy code to it.
- ✓ Confirm the logging of triggers.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#).
- [Visual Studio Code](#).

Set up an Azure Cache for Redis instance

Create a new Azure Cache for Redis instance by using the Azure portal or your preferred CLI tool. This tutorial uses a *Standard C1* instance, which is a good starting point. Use the [quickstart guide](#) to get started.

Azure Cache for Redis helps your application stay responsive even as user load increases. It does so by leveraging the low latency, high-throughput capabilities of the Redis engine. [Learn more](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * CacheTeam

Resource group * contoso-azfunc

DNS name * contosoasfunc12

Location * West US

Cache type (View full pricing details) * Standard C1 (1 GB Cache, Replication)

See which features are available on each Azure Cache for Redis tier [Learn more](#)

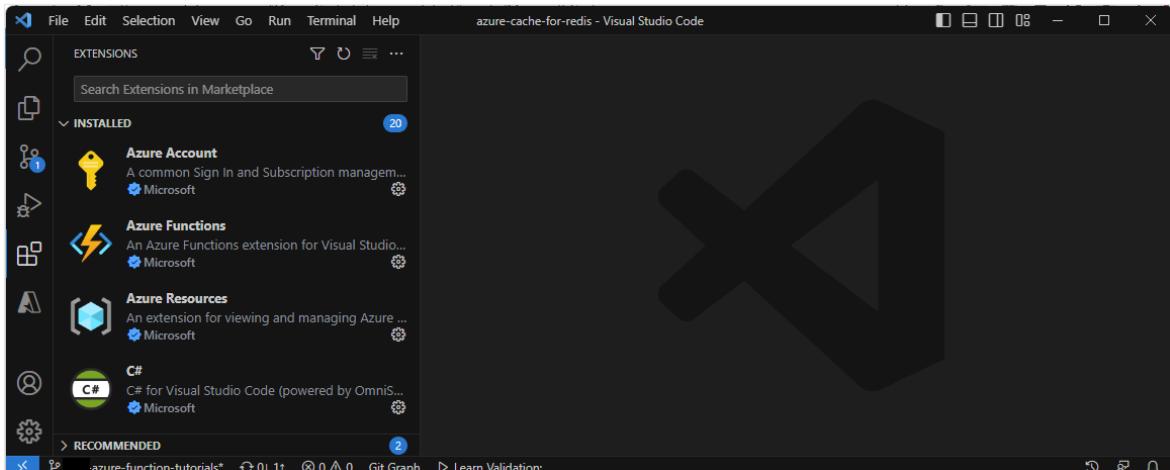
Review + create < Previous Next : Networking >

The default settings should suffice. This tutorial uses a public endpoint for demonstration, but we recommend that you use a private endpoint for anything in production.

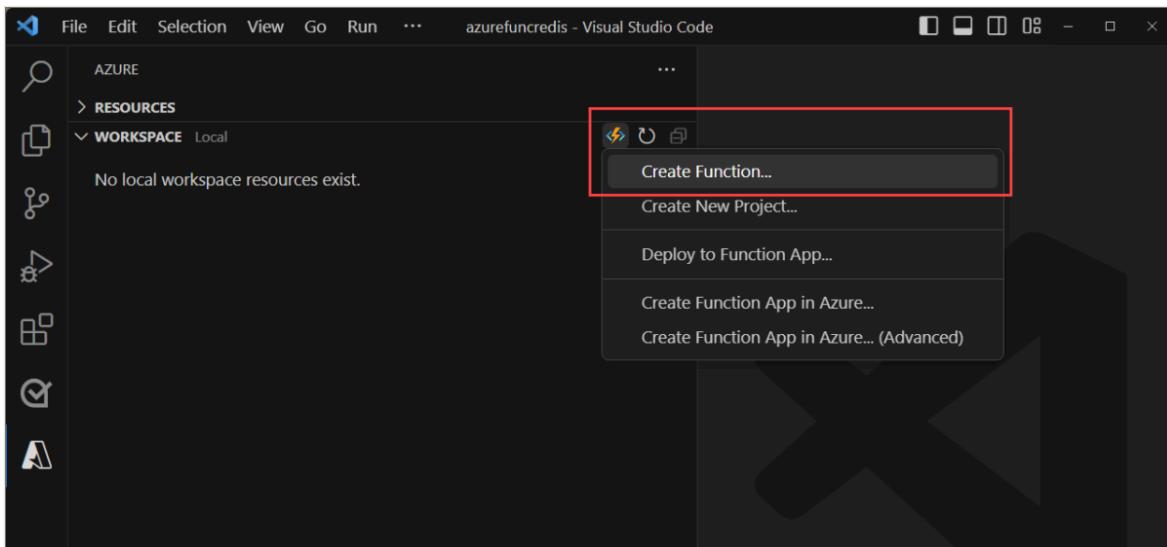
Creating the cache can take a few minutes. You can move to the next section while the process finishes.

Set up Visual Studio Code

1. If you didn't install the Azure Functions extension for VS Code yet, search for **Azure Functions** on the **EXTENSIONS** menu, and then select **Install**. If you don't have the C# extension installed, install it, too.



2. Go to the **Azure** tab. Sign in to your Azure account.
3. To store the project that you're building, create a new local folder on your computer. This tutorial uses *RedisAzureFunctionDemo* as an example.
4. On the **Azure** tab, create a new function app by selecting the lightning bolt icon in the upper right of the **Workspace** tab.
5. Select **Create function....**



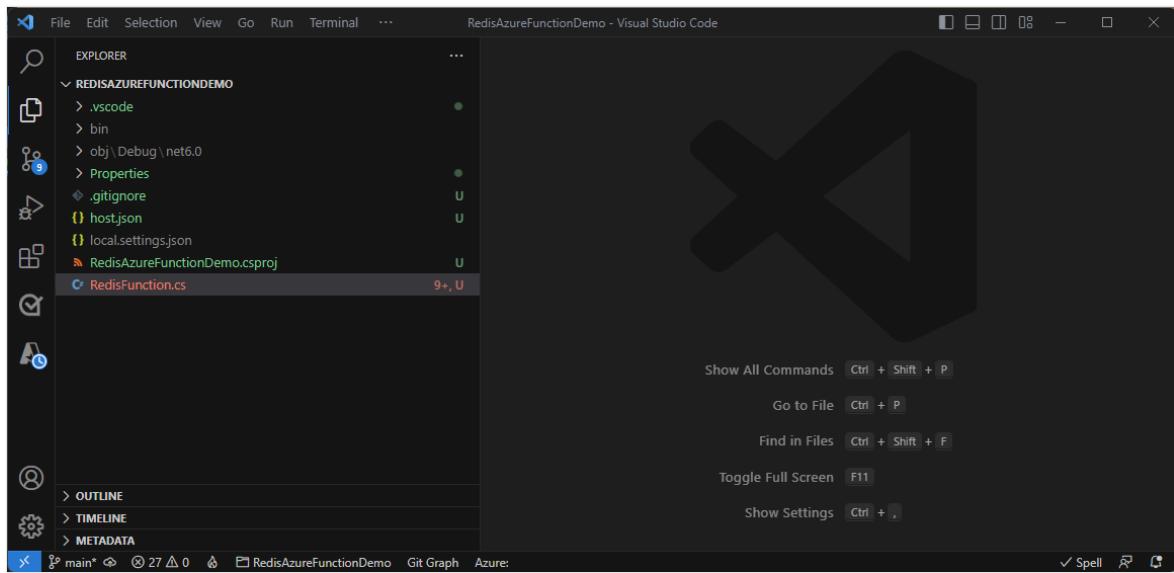
6. Select the folder that you created to start the creation of a new Azure Functions project. You get several on-screen prompts. Select:
 - C# as the language.
 - .NET 8.0 Isolated LTS as the .NET runtime.
 - Skip for now as the project template.

If you don't have the .NET Core SDK installed, you're prompted to do so.

ⓘ Important

For .NET functions, using the *isolated worker model* is recommended over the *in-process* model. For a comparison of the in-process and isolated worker models, see [differences between the isolated worker model and the in-process model for .NET on Azure Functions](#). This sample uses the *isolated worker model*.

7. Confirm that the new project appears on the **EXPLORER** pane.



Install the necessary NuGet package

You need to install `Microsoft.Azure.Functions.Worker.Extensions.Redis`, the NuGet package for the Redis extension that allows Redis keyspace notifications to be used as triggers in Azure Functions.

Install this package by going to the **Terminal** tab in VS Code and entering the following command:

```
terminal
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Redis --prerelease
```

ⓘ Note

The `Microsoft.Azure.Functions.Worker.Extensions.Redis` package is used for .NET isolated worker process functions. .NET in-process functions and all other languages will use the `Microsoft.Azure.WebJobs.Extensions.Redis` package instead.

Configure the cache

1. Go to your newly created Azure Cache for Redis instance.
2. Go to your cache in the Azure portal, and then:
 - a. On the resource menu, select **Advanced settings**.

b. Scroll down to the **notify-keyspace-events** box and enter KEA.

KEA is a configuration string that enables keyspace notifications for all keys and events. For more information on keyspace configuration strings, see the [Redis documentation](#).

c. Select **Save** at the top of the window.

The screenshot shows the 'Advanced settings' page for an Azure Cache for Redis resource named 'azrefunc13'. The left sidebar lists various settings like Overview, Activity log, and Access control (IAM). The 'Access keys' section is selected. In the main pane, there are fields for 'Non-SSL Port' (disabled), 'SSL Port' (6380), 'TLS 1.0 and 1.1 support' (warning), 'Minimum TLS version' (Default), 'Maxmemory policy' (volatile-lru), and 'maxmemory-reserved' (125). The 'notify-keyspace-events' field is highlighted with a red box and contains the value 'KEA'. A 'Save' button is also highlighted with a red box at the top right of the form.

3. Locate **Access keys** on the Resource menu, and then write down or copy the contents of the **Primary connection string** box. This string is used to connect to the cache.

The screenshot shows the 'Access keys' page for an Azure Cache for Redis resource named 'contososazfunc17'. The left sidebar lists various settings like Overview, Activity log, and Access control (IAM). The 'Access keys' section is selected. It shows two connection strings: 'Primary' and 'Secondary'. The 'Primary connection string' box is highlighted with a red box and contains the value 'contososazfunc17.redis.cache.windows.net:6380,password=7n7Qk6llaDXazB2h5ZWcbbq0oQncqbCYjAzCaOldOyE='.

Set up the example code for Redis triggers

1. In VS Code, add a file called *Common.cs* to the project. This class is used to help parse the JSON serialized response for the PubSubTrigger.
2. Copy and paste the following code into the *Common.cs* file:

```
C#  
  
public class Common  
{  
    public const string connectionString = "redisConnectionString";  
  
    public class ChannelMessage  
    {  
        public string SubscriptionChannel { get; set; }  
        public string Channel { get; set; }  
        public string Message { get; set; }  
    }  
}
```

3. Add a file called *RedisTriggers.cs* to the project.
4. Copy and paste the following code sample into the new file:

```
C#  
  
using Microsoft.Extensions.Logging;  
using Microsoft.Azure.Functions.Worker;  
using Microsoft.Azure.Functions.Worker.Extensions.Redis;  
  
public class RedisTriggers  
{  
    private readonly ILogger<RedisTriggers> logger;  
  
    public RedisTriggers(ILogger<RedisTriggers> logger)  
    {  
        this.logger = logger;  
    }  
  
    // PubSubTrigger function listens to messages from the 'pubsubTest'  
    // channel.  
    [Function("PubSubTrigger")]  
    public void PubSub(  
        [RedisPubSubTrigger(connectionString, "pubsubTest")]  
        Common.ChannelMessage channelMessage)  
    {  
        logger.LogInformation($"Function triggered on pub/sub message  
'{channelMessage.Message}' from channel '{channelMessage.Channel}'.");  
    }  
}
```

```

    // KeyeventTrigger function listens to key events from the 'del'
    operation.
    [Function("KeyeventTrigger")]
    public void Keyevent(
        [RedisPubSubTrigger(Common.connectionString,
    "__keyevent@0__:del")] Common.ChannelMessage channelMessage)
    {
        logger.LogInformation($"Key '{channelMessage.Message}' deleted.");
    }

    // KeyspaceTrigger function listens to key events on the
    'keyspaceTest' key.
    [Function("KeyspaceTrigger")]
    public void Keyspace(
        [RedisPubSubTrigger(Common.connectionString,
    "__keyspace@0__:keyspaceTest")] Common.ChannelMessage channelMessage)
    {
        logger.LogInformation($"Key 'keyspaceTest' was updated with
operation '{channelMessage.Message}'");
    }

    // ListTrigger function listens to changes to the 'listTest' list.
    [Function("ListTrigger")]
    public void List(
        [RedisListTrigger(Common.connectionString, "listTest")] string
response)
    {
        logger.LogInformation(response);
    }

    // StreamTrigger function listens to changes to the 'streamTest'
    stream.
    [Function("StreamTrigger")]
    public void Stream(
        [RedisStreamTrigger(Common.connectionString, "streamTest")]
string response)
    {
        logger.LogInformation(response);
    }
}

```

5. This tutorial shows multiple ways to trigger on Redis activity:

- `PubSubTrigger`, which is triggered when an activity is published to the Pub/Sub channel named `pubsubTest`.
- `KeyspaceTrigger`, which is built on the Pub/Sub trigger. Use it to look for changes to the `keyspaceTest` key.
- `KeyeventTrigger`, which is also built on the Pub/Sub trigger. Use it to look for any use of the `DEL` command.
- `ListTrigger`, which looks for changes to the `listTest` list.

- `StreamTrigger`, which looks for changes to the `streamTest` stream.

Connect to your cache

1. To trigger on Redis activity, you need to pass in the connection string of your cache instance. This information is stored in the `local.settings.json` file that was automatically created in your folder. We recommend that you use the [local settings file](#) as a security best practice.
2. To connect to your cache, add a `ConnectionStrings` section in the `local.settings.json` file, and then add your connection string by using the `redisConnectionString` parameter. The section should look like this example:

JSON

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "",  
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated",  
        "redisConnectionString": "<your-connection-string>"  
    }  
}
```

The code in `Common.cs` looks to this value when it's running locally:

C#

```
public const string connectionString = "redisConnectionString";
```

ⓘ Important

This example is simplified for the tutorial. For production use, we recommend that you use [Azure Key Vault](#) to store connection string information or [authenticate to the Redis instance using EntraID](#).

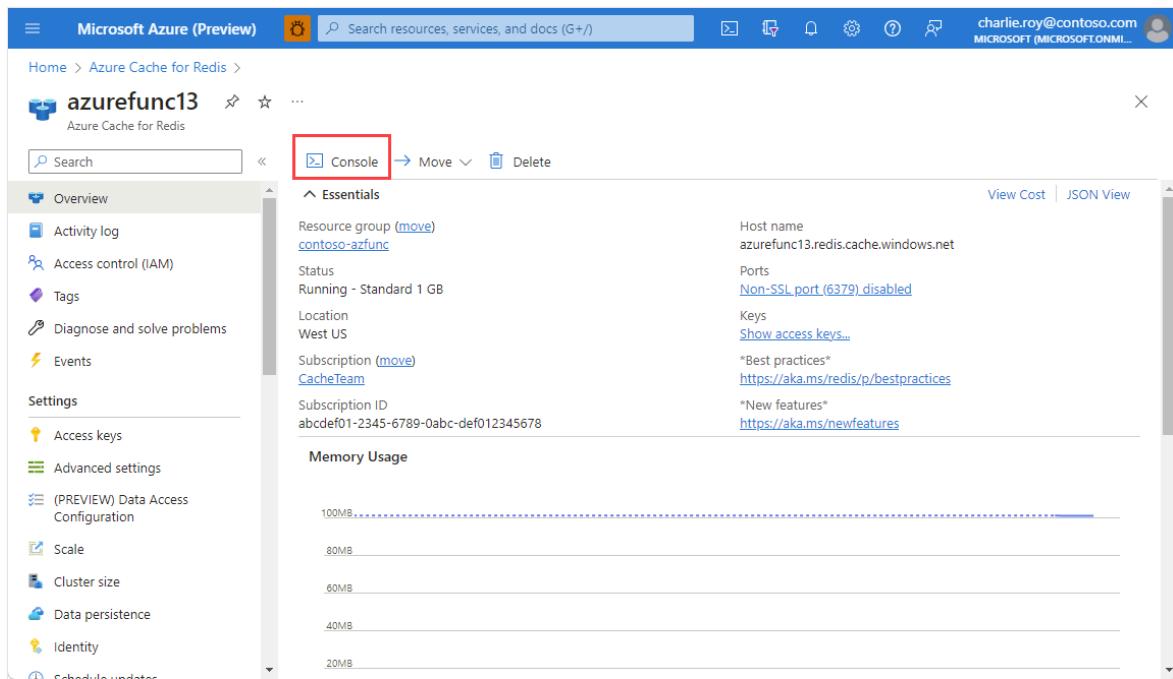
Build and run the code locally

1. Switch to the **Run and debug** tab in VS Code and select the green arrow to debug the code locally. If you don't have Azure Functions core tools installed, you're prompted to do so. In that case, you'll need to restart VS Code after installing.

2. The code should build successfully. You can track its progress in the terminal output.

3. To test the trigger functionality, try creating and deleting the `keyspaceTest` key.

You can use any way you prefer to connect to the cache. An easy way is to use the built-in console tool in the Azure Cache for Redis portal. Go to the cache instance in the Azure portal, and then select **Console** to open it.



The screenshot shows the Azure Cache for Redis blade for a resource named 'azrefunc13'. The 'Console' button in the top navigation bar is highlighted with a red box. The blade displays various details about the cache, including its resource group, status, location, and subscription information. It also shows memory usage metrics at the bottom.

After the console is open, try the following commands:

- `SET keyspaceTest 1`
- `SET keyspaceTest 2`
- `DEL keyspaceTest`
- `PUBLISH pubsubTest testMessage`
- `LPUSH listTest test`
- `XADD streamTest * name Clippy`

 **(PREVIEW) Redis Console** ⚡ ...

RedisFunctionsTriggerCache

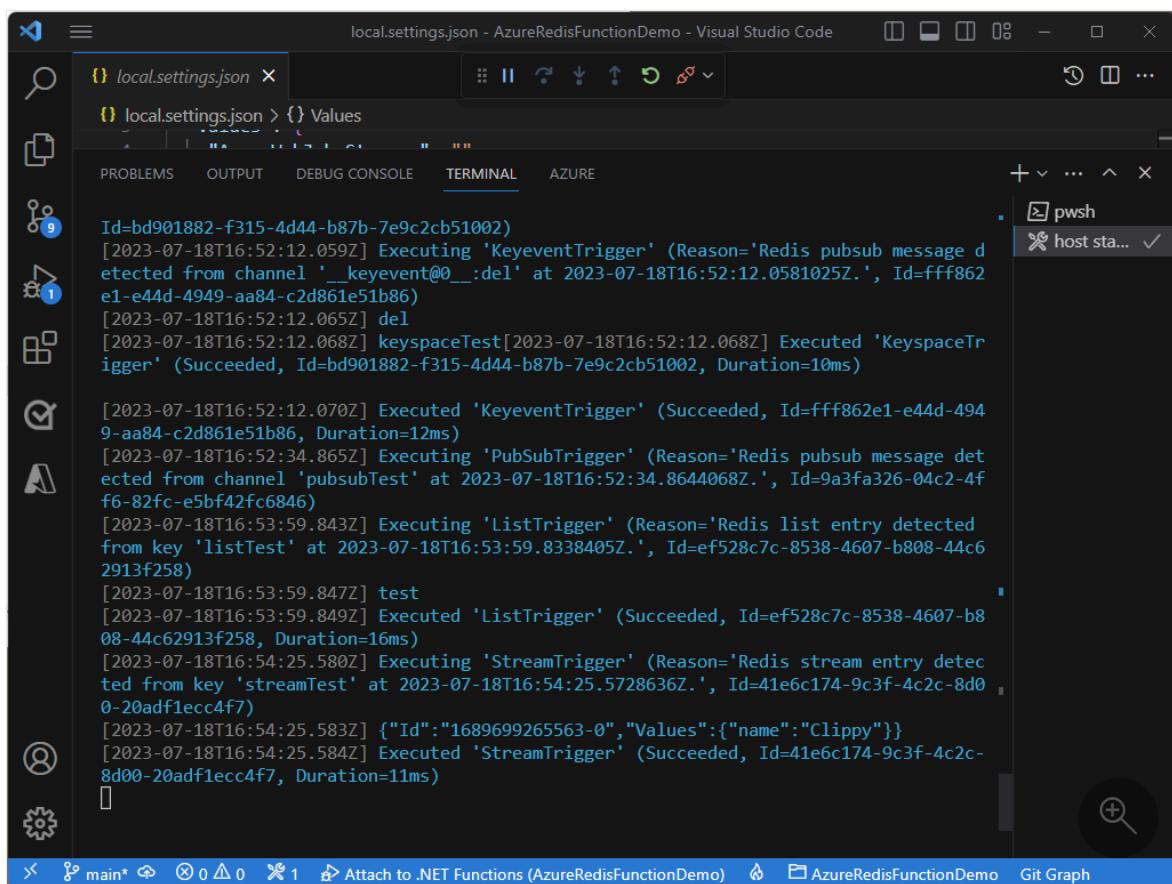
```
Welcome to secure redis console!

This console connects to your live redis server and all commands are run on the server.

WARNING: Use expensive commands with caution as they can impact your server load!

>set keyspaceTest 1
OK
>set keyspaceTest 2
OK
>del keyspaceTest
(integer) 1
>publish pubsubTest testMessage
(integer) 1
>LPUSH listTest test
(integer) 1
>XADD streamTest * name Clippy
"1676673107882-0"
>
```

4. Confirm that the triggers are being activated in the terminal.



The screenshot shows the Visual Studio Code interface with the terminal tab active. The terminal window displays a log of Redis trigger executions:

```
Id=bd901882-f315-4d44-b87b-7e9c2cb51002
[2023-07-18T16:52:12.059Z] Executing 'KeyeventTrigger' (Reason='Redis pubsub message detected from channel '_keyevent@0__:del' at 2023-07-18T16:52:12.0581025Z.', Id=fff862e1-e44d-4949-aa84-c2d861e51b86)
[2023-07-18T16:52:12.065Z] del
[2023-07-18T16:52:12.068Z] keyspaceTest[2023-07-18T16:52:12.068Z] Executed 'KeyspaceTrigger' (Succeeded, Id=bd901882-f315-4d44-b87b-7e9c2cb51002, Duration=10ms)

[2023-07-18T16:52:12.070Z] Executed 'KeyeventTrigger' (Succeeded, Id=fff862e1-e44d-4949-aa84-c2d861e51b86, Duration=12ms)
[2023-07-18T16:52:34.865Z] Executing 'PubSubTrigger' (Reason='Redis pubsub message detected from channel 'pubsubTest' at 2023-07-18T16:52:34.8644068Z.', Id=9a3fa326-04c2-4f6-82fc-e5bf42fc6846)
[2023-07-18T16:53:59.843Z] Executing 'ListTrigger' (Reason='Redis list entry detected from key 'listTest' at 2023-07-18T16:53:59.8338405Z.', Id=ef528c7c-8538-4607-b808-44c62913f258)
[2023-07-18T16:53:59.847Z] test
[2023-07-18T16:53:59.849Z] Executed 'ListTrigger' (Succeeded, Id=ef528c7c-8538-4607-b808-44c62913f258, Duration=16ms)
[2023-07-18T16:54:25.580Z] Executing 'StreamTrigger' (Reason='Redis stream entry detected from key 'streamTest' at 2023-07-18T16:54:25.5728636Z.', Id=41e6c174-9c3f-4c2c-8d00-20adf1ecc4f7)
[2023-07-18T16:54:25.583Z] {"Id":"1689699265563-0","Values": {"name": "Clippy"}}
[2023-07-18T16:54:25.584Z] Executed 'StreamTrigger' (Succeeded, Id=41e6c174-9c3f-4c2c-8d00-20adf1ecc4f7, Duration=11ms)
```

Add Redis bindings

Bindings add a streamlined way to read or write data stored on your Redis instance. To demonstrate the benefit of bindings, we add two other functions. One is called `SetGetter`, which triggers each time a key is set and returns the new value of the key using an *input binding*. The other is called `StreamSetter`, which triggers when a new item is added to the stream `myStream` and uses an *output binding* to write the value `true` to the key `newStreamEntry`.

1. Add a file called `RedisBindings.cs` to the project.
2. Copy and paste the following code sample into the new file:

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Redis;

public class RedisBindings
{
    private readonly ILogger<RedisBindings> logger;

    public RedisBindings(ILogger<RedisBindings> logger)
    {
        this.logger = logger;
    }

    //This example uses the PubSub trigger to listen to key events on
    //the 'set' operation. A Redis Input binding is used to get the value of
    //the key being set.
    [Function("SetGetter")]
    public void SetGetter(
        [RedisPubSubTrigger(Common.ConnectionString,
    "__keyevent@0__:set")] Common.ChannelMessage channelMessage,
        [RedisInput(Common.ConnectionString, "GET {Message}")] string
    value)
    {
        logger.LogInformation($"Key '{channelMessage.Message}' was set
    to value '{value}'");
    }

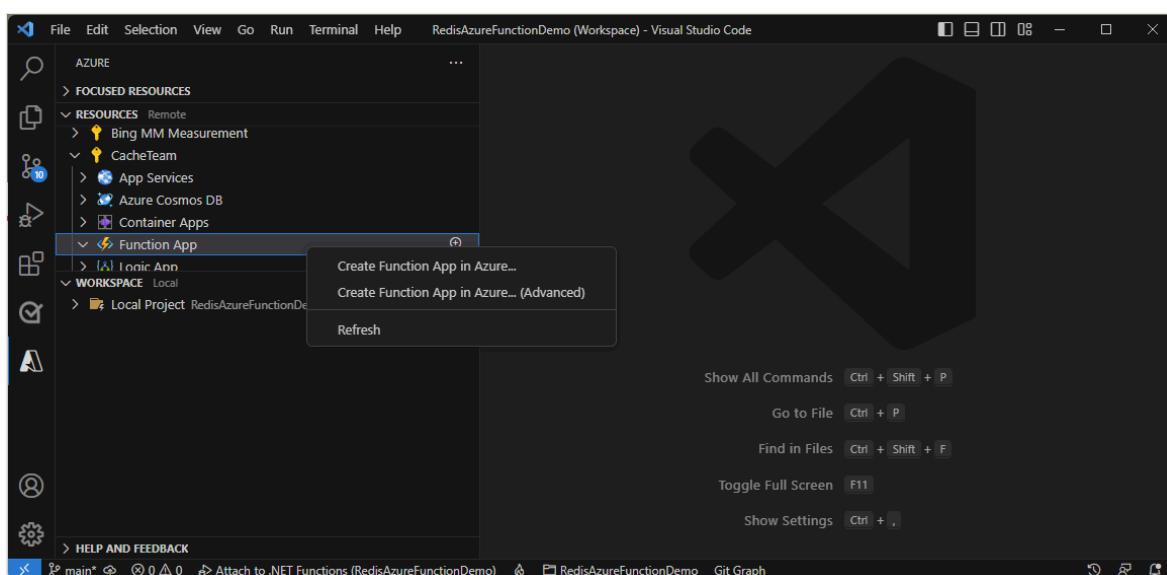
    //This example uses the PubSub trigger to listen to key events to
    //the key 'key1'. When key1 is modified, a Redis Output binding is used
    //to set the value of the 'key1modified' key to 'true'.
    [Function("SetSetter")]
    [RedisOutput(Common.ConnectionString, "SET")]
    public string SetSetter(
        [RedisPubSubTrigger(Common.ConnectionString,
    "__keyspace@0__:key1")] Common.ChannelMessage channelMessage)
    {
        logger.LogInformation($"Key '{channelMessage.Message}' was
    updated. Setting the value of 'key1modified' to 'true'");
    }
}
```

```
        return $"key1modified true";
    }
}
```

3. Switch to the **Run and debug** tab in VS Code and select the green arrow to debug the code locally. The code should build successfully. You can track its progress in the terminal output.
4. To test the input binding functionality, try setting a new value for any key, for instance using the command `SET hello world`. You should see that the `SetGetter` function triggers and returns the updated value.
5. To test the output binding functionality, try adding a new item to the stream `myStream` using the command `XADD myStream * item Order1`. Notice that the `StreamSetter` function triggered on the new stream entry and set the value `true` to another key called `newStreamEntry`. This `set` command also triggers the `SetGetter` function.

Deploy code to an Azure function

1. Create a new Azure function:
 - a. Go back to the **Azure** tab and expand your subscription.
 - b. Right-click **Function App**, and then select **Create Function App in Azure (Advanced)**.



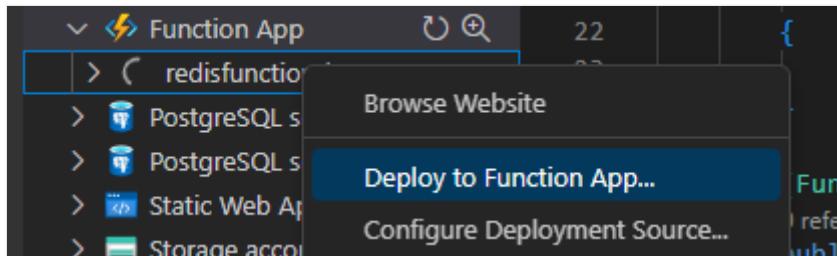
2. You get several prompts for information to configure the new function app:
 - Enter a unique name.
 - Select **.NET 8 Isolated** as the runtime stack.

- Select either **Linux** or **Windows** (either works).
- Select an existing or new resource group to hold the function app.
- Select the same region as your cache instance.
- Select **Premium** as the hosting plan.
- Create a new Azure App Service plan.
- Select the **EP1** pricing tier.
- Select an existing storage account or create a new one.
- Create a new Application Insights resource. You use the resource to confirm that the trigger is working.

ⓘ Important

Redis triggers aren't currently supported on consumption functions.

3. Wait a few minutes for the new function app to be created. It appears under **Function App** in your subscription. Right-click the new function app, and then select **Deploy to Function App**.



4. The app builds and starts deploying. You can track its progress in the output window.

Add connection string information

1. In the Azure portal, go to your new function app and select **Environment variables** from the resource menu.
2. On the working pane, go to **App settings**.
3. For **Name**, enter **redisConnectionString**.
4. For **Value**, enter your connection string.
5. Select **Apply** on the page to confirm.
6. Navigate to the **Overview** pane and select **Restart** to reboot the functions app with the connection string information.

Test your triggers and bindings

1. After deployment is complete and the connection string information is added, open your function app in the Azure portal. Then select **Log Stream** from the resource menu.
2. Wait for Log Analytics to connect, and then use the Redis console to activate any of the triggers. Confirm that triggers are being logged here.

The screenshot shows the Azure portal interface with the URL [https://portal.azure.com/#blade/HubsBlade](#). The left sidebar shows the navigation path: Home > Function App > redisfunction19. The main content area is titled "redisfunction19 | Log stream". The "Log stream" section is selected in the sidebar. The log entries are as follows:

```
19722:56:06-2023:06:06Z : [Information] set
2023-07-19T22:56:06Z [Information] Executed 'KeypspaceTrigger' (Succeeded, Id=23332ca3-3fe6-44aa-88bc-723091e42c59, Duration=0ms)
19722:56:06-2023:06:06Z : [Information] Executing 'KeypspaceTrigger' (Reason='Redis pubsub message detected from channel '_keyspace@0:_keyspaceTest' at 2023-07-19T22:56:06-06:06')
2023-07-19T22:56:06Z : [Information] Executed 'KeypspaceTrigger' (Succeeded, Id=98a12dfc-821c-4450-8b41-806016b74974, Duration=0ms)
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing 'KeypspaceTrigger' (Reason='Redis pubsub message detected from channel '_keyevent@0:_del' at 2023-07-19T22:56:13-06:06')
2023-07-19T22:56:06Z : [Information] Executed 'KeypspaceTrigger' (Reason='Redis pubsub message detected from channel '_keyspace@0:_keyspaceTest' at 2023-07-19T22:56:13-06:06)
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing 'KeypspaceTrigger' (Reason='Redis pubsub message detected from channel '_keyevent@0:_del' at 2023-07-19T22:56:13-06:06)
2023-07-19T22:56:06Z : [Information] Executed 'KeypspaceTrigger' (Succeeded, Id=5b2f57ed-952f-4db7-8c7a-2b5688a7256a, Duration=0ms)
2023-07-19T22:56:06Z : [Information] Executing KeypspaceTest
2023-07-19T22:56:06Z : [Information] Executed 'keyeventTrigger' (Succeeded, Id=07a438ec-94ad-4d99-9a32-3848540781fe, Duration=0ms)
2023-07-19T22:56:06Z : [Information] Executing 'KeypspaceTrigger' (Reason='Redis pubsub message detected from channel '_keyspace@0:_keyspaceTest' at 2023-07-19T22:56:06-06:06)
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing 'KeypspaceTrigger' (Reason='Redis pubsub message detected from channel '_keyevent@0:_del' at 2023-07-19T22:56:13-06:06)
2023-07-19T22:56:06Z : [Information] Executed 'KeypspaceTrigger' (Succeeded, Id=e8b8cce4-11bd-404d-b99e-14245b322a56, Duration=0ms)
2023-07-19T22:56:06Z : [Information] del
2023-07-19T22:56:06Z : [Information] Executed 'KeypspaceTrigger' (Succeeded, Id=278906508-660f-44ae-b979-9b286931b680, Duration=1ms)
2023-07-19T22:56:06Z : [Information] Executing KeypspaceTest
2023-07-19T22:56:06Z : [Information] Executed 'keyeventTrigger' (Succeeded, Id=e8b8cce4-11bd-404d-b99e-14245b322a56, Duration=0ms)
2023-07-19T22:56:06Z : [Information] host Status: {
    "id": "redisfunction19",
    "state": "redisfunction19",
    "version": "2023-07-19T22:56:06Z-20888",
    "versionDetails": "4.23.0+eeacf1e92eefab538714da022f010810a252f36d",
    "platformVersion": "100.0.7.252",
    "instanceId": "0dbddffcc08e2edcbea5a98ded4c819ed63fcc0939e49c487429c6add6272a",
    "computerName": "pd1sdw00005w",
    "processUptime": 1501225,
    "functionAppContentId": "Unknown"
}
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
2023-07-19T22:56:06Z : [Information] Executing StatusCodeResult, setting HTTP status code 200
```

Clean up resources

If you want to continue to use the resources you created in this article, keep the resource group.

Otherwise, if you're finished with the resources, you can delete the Azure resource group that you created to avoid charges.

Important

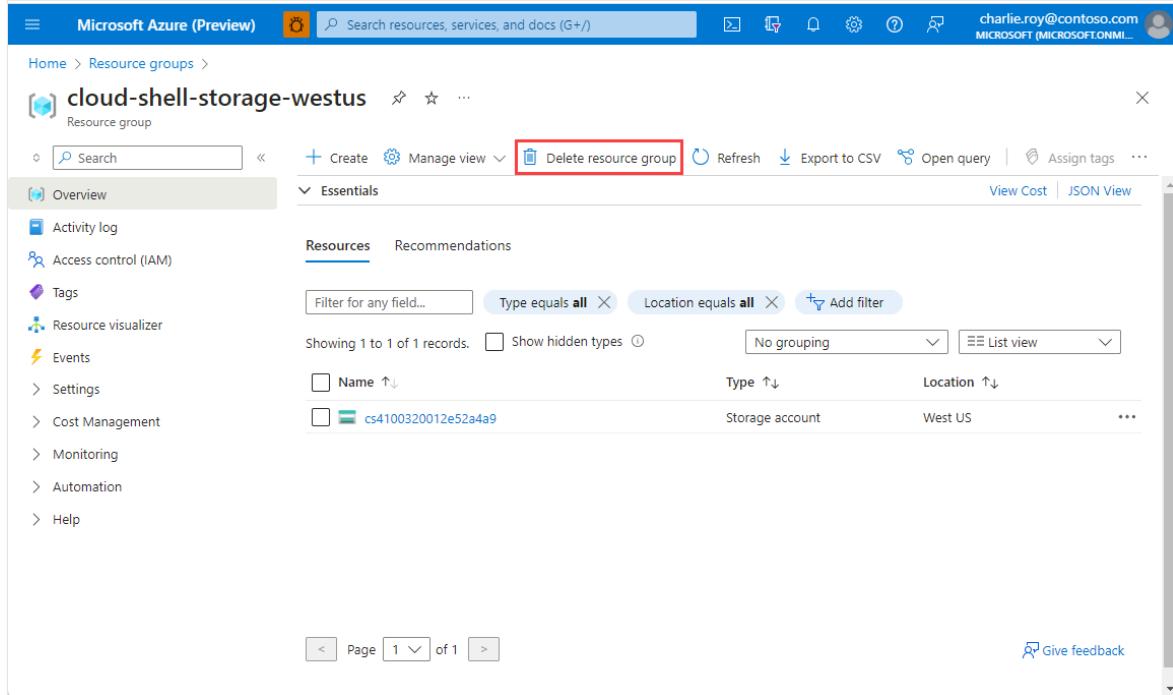
Deleting a resource group is irreversible. When you delete a resource group, all the resources in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the resources inside an existing resource group that contains resources you want to keep, you can delete each resource individually instead of deleting the resource group.

To delete a resource group

1. Sign in to the [Azure portal](#), and then select **Resource groups**.

2. Select the resource group you want to delete.

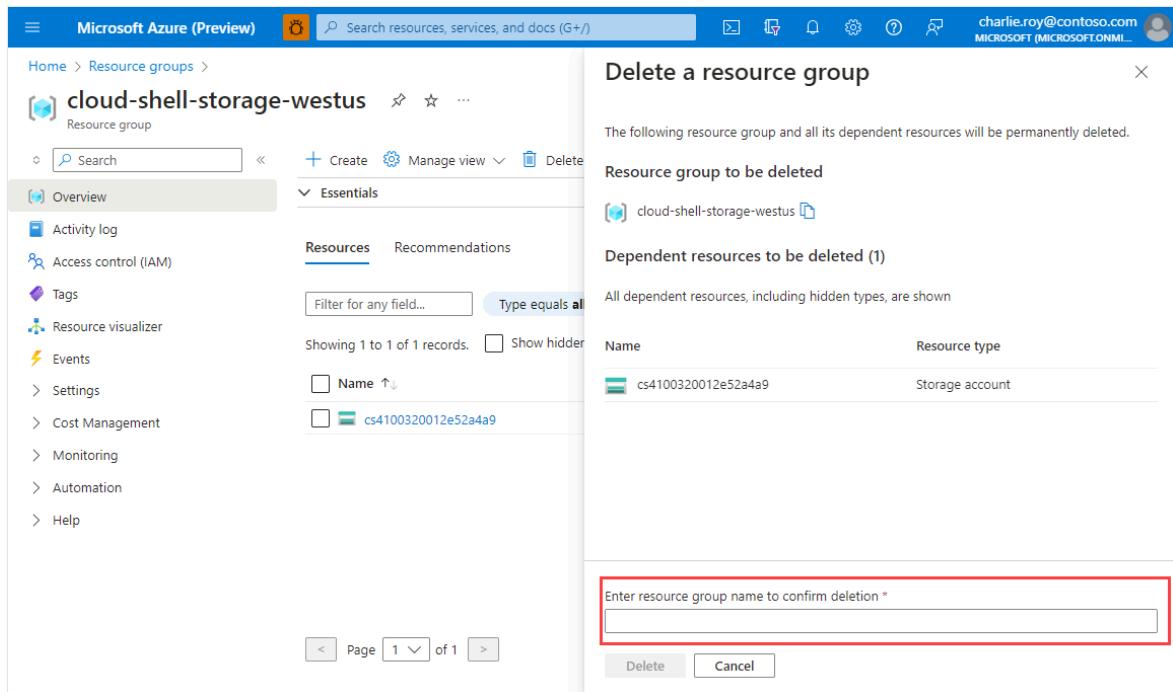
If there are many resource groups, use the **Filter for any field...** box, type the name of your resource group you created for this article. Select the resource group in the results list.



The screenshot shows the Microsoft Azure Resource Groups blade. The left sidebar lists various options like Overview, Activity log, Access control (IAM), Tags, Resource visualizer, Events, Settings, Cost Management, Monitoring, Automation, and Help. The main area displays the 'cloud-shell-storage-westus' resource group. A search bar at the top has 'cloud-shell-storage-westus' typed into it. Below the search bar, there are buttons for '+ Create', 'Manage view', and 'Delete resource group'. The 'Delete resource group' button is highlighted with a red box. To its right are 'Refresh', 'Export to CSV', 'Open query', 'Assign tags', and more. The 'Essentials' section shows a single record: 'cs4100320012e52a4a9' (Storage account) located in 'West US'. At the bottom, there are navigation buttons for 'Page' (1 of 1) and a 'Give feedback' link.

3. Select Delete resource group.

4. You're asked to confirm the deletion of the resource group. Type the name of your resource group to confirm, and then select Delete.



The screenshot shows the Microsoft Azure Resource Groups blade with the 'Delete a resource group' dialog open. The dialog title is 'Delete a resource group'. It contains the message: 'The following resource group and all its dependent resources will be permanently deleted.' Below this, it says 'Resource group to be deleted' and shows 'cloud-shell-storage-westus'. It also lists 'Dependent resources to be deleted (1)' and 'All dependent resources, including hidden types, are shown'. A table shows one resource: 'Name' 'cs4100320012e52a4a9' and 'Resource type' 'Storage account'. At the bottom of the dialog, there is an input field labeled 'Enter resource group name to confirm deletion *' which is highlighted with a red box. Below the input field are 'Delete' and 'Cancel' buttons.

After a few moments, the resource group and all of its resources are deleted.

Related content

- [Overview of Azure functions for Azure Cache for Redis](#)
 - [Build a write-behind cache by using Azure Functions](#)
-

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Create a write-behind cache by using Azure Functions and Azure Cache for Redis

Article • 04/12/2024

The objective of this tutorial is to use an Azure Cache for Redis instance as a [write-behind cache](#). The write-behind pattern in this tutorial shows how writes to the cache trigger corresponding writes to a SQL database (an instance of the Azure SQL Database service).

You use the [Redis trigger for Azure Functions](#) to implement this functionality. In this scenario, you see how to use Azure Cache for Redis to store inventory and pricing information, while backing up that information in a SQL database.

Every new item or new price written to the cache is then reflected in a SQL table in the database.

In this tutorial, you learn how to:

- ✓ Configure a database, trigger, and connection strings.
- ✓ Validate that triggers are working.
- ✓ Deploy code to a function app.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#).
- Completion of the previous tutorial, [Get started with Azure Functions triggers in Azure Cache for Redis](#), with these resources provisioned:
 - Azure Cache for Redis instance
 - Azure Functions instance
 - A working knowledge of using Azure SQL
 - Visual Studio Code (VS Code) environment set up with NuGet packages installed

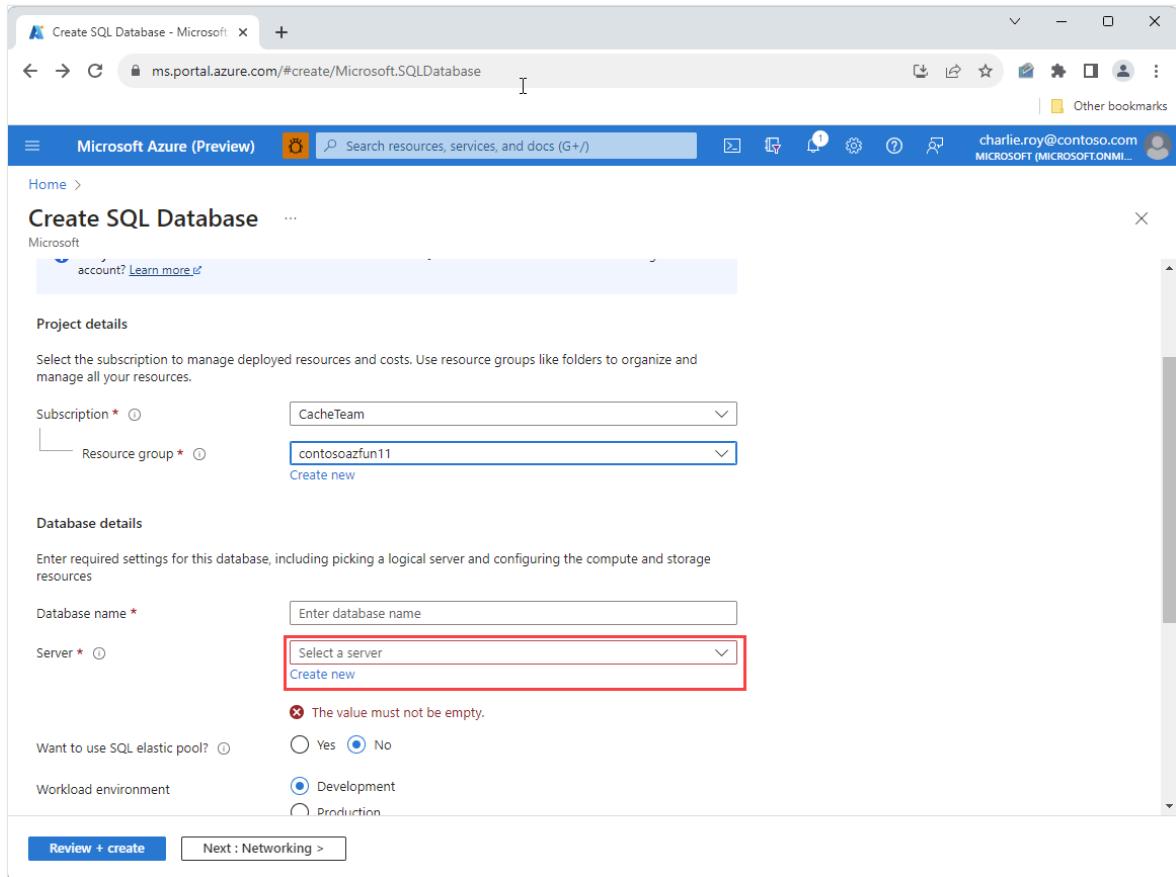
Create and configure a new SQL database

The SQL database is the backing database for this example. You can create a SQL database through the Azure portal or through your preferred method of automation.

For more information on creating a SQL database, see [Quickstart: Create a single database - Azure SQL Database](#).

This example uses the portal:

1. Enter a database name and select **Create new** to create a new server to hold the database.



2. Select **Use SQL authentication** and enter an admin sign-in and password. Be sure to remember these credentials or write them down. When you're deploying a server in production, use Microsoft Entra authentication instead.

Microsoft Azure (Preview) Search resources, services, and docs (G+/)

Home > Create SQL Database > Create SQL Database Server

Server details

Enter required settings for this server, including providing a name and location. This server will be created in the same subscription and resource group as your database.

Server name * contososql14 .database.windows.net

Location * (US) East US

Authentication

Select your preferred authentication methods for accessing this server. Create a server admin login and password to access your server with SQL authentication, select only Azure AD authentication [Learn more](#) using an existing Azure AD user, group, or application as Azure AD admin [Learn more](#), or select both SQL and Azure AD authentication.

Authentication method Use only Azure Active Directory (Azure AD) authentication Use both SQL and Azure AD authentication Use SQL authentication

Server admin login * contoso.charlie.roy

Password *

Confirm password *

OK

3. Go to the **Networking** tab and choose **Public endpoint** as a connection method. Select **Yes** for both firewall rules that appear. This endpoint allows access from your Azure function app.

Microsoft Azure (Preview) Search resources, services, and docs (G+/)

Home > Create SQL Database

Networking

Configure network access and connectivity for your server. The configuration selected below will apply to the selected server 'contososql14' and all databases it manages. [Learn more](#)

Network connectivity

Choose an option for configuring connectivity to your server via public endpoint or private endpoint. Choosing no access creates with defaults and you can configure connection method after server creation. [Learn more](#)

Connectivity method * No access Public endpoint Private endpoint

Firewall rules

Setting 'Allow Azure services and resources to access this server' to Yes allows communications from all resources inside the Azure boundary, that may or may not be part of your subscription. [Learn more](#)
Setting 'Add current client IP address' to Yes will add an entry for your client IP address to the server firewall.

Allow Azure services and resources to access this server * No Yes

Add current client IP address * No Yes

Connection policy

Configure how clients communicate with your SQL database server. [Learn more](#)

Connection policy Default - Uses Redirect policy for all client connections originating inside of Azure (except Private Endpoint connections) and Proxy for all client connections originating outside Azure Proxy - All connections are proxied via the Azure SQL Database gateways

Cost summary

General Purpose (GP_S_Gen5_1)
Cost per GB (in USD) 0.12
Max storage selected (in GB) x 41.6
ESTIMATED STORAGE COST / MONTH 4.78 USD
COMPUTE COST / VCORE SECOND 0.000145 USD

NOTES

1 Serverless databases are billed in vCore seconds based on a combination of CPU and memory utilization. [Learn more about serverless billing](#)

Review + create < Previous Next : Security >

4. After validation finishes, select **Review + create** and then **Create**. The SQL database starts to deploy.

5. After deployment finishes, go to the resource in the Azure portal and select the **Query editor** tab. Create a new table called *inventory* that holds the data you'll write to it. Use the following SQL command to make a new table with two fields:

- `ItemName` lists the name of each item.
- `Price` stores the price of the item.

The screenshot shows the Microsoft Azure Query editor interface. At the top, there is a code editor window containing the following SQL command:

```
CREATE TABLE inventory (
    ItemName varchar(255),
    Price decimal(18,2)
);
```

Below the code editor is a toolbar with several buttons. The "Run" button is highlighted with a red box. The toolbar also includes "Cancel query", "Save query", "Export data as", and "Show only Editor". To the right of the toolbar, there is a user profile and a "MICROSOFT (MICROSOFT.ONMIL)" link. The main workspace shows a sidebar with "Tables", "Views", and "Stored Procedures". The "Tables" section is expanded, showing "sys.database_firewall_rules". The main area is titled "Query 1" and contains the SQL command. Below the command, there are tabs for "Results" and "Messages", and a search bar. A green banner at the bottom says "Ready".

6. After the command finishes running, expand the *Tables* folder and verify that the new table was created.

Configure the Redis trigger

First, make a copy of the same VS Code project that you used in the previous [tutorial](#).

Copy the folder from the previous tutorial under a new name, such as

RedisWriteBehindTrigger, and open it in VS Code.

Second, delete the *RedisBindings.cs* and *RedisTriggers.cs* files.

In this example, you use the [pub/sub trigger](#) to trigger on `keyevent` notifications. The goals of the example are:

- Trigger every time a `SET` event occurs. A `SET` event happens when either new keys are written to the cache instance or the value of a key is changed.
- After a `SET` event is triggered, access the cache instance to find the value of the new key.
- Determine if the key already exists in the *inventory* table in the SQL database.
 - If so, update the value of that key.
 - If not, write a new row with the key and its value.

To configure the trigger:

1. Import the `System.Data.SqlClient` NuGet package to enable communication with the SQL database. Go to the VS Code terminal and use the following command:

terminal

```
dotnet add package System.Data.SqlClient
```

2. Create a new file called *RedisFunction.cs*. Make sure you've deleted the *RedisBindings.cs* and *RedisTriggers.cs* files.
3. Copy and paste the following code in *RedisFunction.cs* to replace the existing code:

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Redis;
using System.Data.SqlClient;

public class WriteBehindDemo
{
    private readonly ILogger<WriteBehindDemo> logger;

    public WriteBehindDemo(ILogger<WriteBehindDemo> logger)
    {
        this.logger = logger;
    }

    public string SQLAddress =
        System.Environment.GetEnvironmentVariable("SQLConnectionString");

    //This example uses the PubSub trigger to listen to key events on the
    'set' operation. A Redis Input binding is used to get the value of the key
    being set.
    [Function("WriteBehind")]
}
```

```

public void WriteBehind(
    [RedisPubSubTrigger(Common.connectionString, "__keyevent@0__:set")]
Common.ChannelMessage channelMessage,
    [RedisInput(Common.connectionString, "GET {Message}")] string
setValue)
{
    var key = channelMessage.Message; //The name of the key that was set
    var value = 0.0;

    //Check if the value is a number. If not, log an error and return.
    if (double.TryParse(setValue, out double result))
    {
        value = result; //The value that was set. (i.e. the price.)
        logger.LogInformation($"Key '{channelMessage.Message}' was set
to value '{value}'");
    }
    else
    {
        logger.LogInformation($"Invalid input for key '{key}'. A number
is expected.");
        return;
    }

    // Define the name of the table you created and the column names.
    String tableName = "dbo.inventory";
    String column1Value = "ItemName";
    String column2Value = "Price";

    logger.LogInformation($" '{SQLAddress}'");
    using (SqlConnection connection = new SqlConnection(SQLAddress))
    {
        connection.Open();
        using (SqlCommand command = new SqlCommand())
        {
            command.Connection = connection;

            //Form the SQL query to update the database. In
            practice, you would want to use a parameterized query to prevent SQL
            injection attacks.
            //An example query would be something like "UPDATE
            dbo.inventory SET Price = 1.75 WHERE ItemName = 'Apple'".
            command.CommandText = "UPDATE " + tableName + " SET " +
            column2Value + " = " + value + " WHERE " + column1Value + " = '" + key +
            "'";

            int rowsAffected = command.ExecuteNonQuery(); //The
            query execution returns the number of rows affected by the query. If the key
            doesn't exist, it will return 0.

            if (rowsAffected == 0) //If key doesn't exist, add it to
            the database
            {
                //Form the SQL query to update the database. In
                practice, you would want to use a parameterized query to prevent SQL
                injection attacks.
                //An example query would be something like "INSERT

```

```

        INTO dbo.inventory (ItemName, Price) VALUES ('Bread', '2.55').
        command.CommandText = "INSERT INTO " + tableName + "
        (" + column1Value + ", " + column2Value + ") VALUES ('" + key + "', '" +
        value + "')";
        command.ExecuteNonQuery();

        logger.LogInformation($"Item {key} has been
added to the database with price {value}");
    }

    else {
        logger.LogInformation($"Item {key} has been
updated to price {value}");
    }
}
connection.Close();
}

//Log the time that the function was executed.
logger.LogInformation($"C# Redis trigger function executed at:
{DateTime.Now}");
}
}

```

ⓘ Important

This example is simplified for the tutorial. For production use, we recommend that you use parameterized SQL queries to prevent SQL injection attacks.

Configure connection strings

You need to update the `local.settings.json` file to include the connection string for your SQL database. Add an entry in the `Values` section for `SQLConnectionString`. Your file should look like this example:

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated",
    "redisConnectionString": "<redis-connection-string>",
    "SQLConnectionString": "<sql-connection-string>"
  }
}
```

To find the Redis connection string, go to the resource menu in the Azure Cache for Redis resource. Locate the string is in the **Access Keys** area on the Resource menu.

To find the SQL database connection string, go to the resource menu in the SQL database resource. Under **Settings**, select **Connection strings**, and then select the **ADO.NET** tab. The string is in the **ADO.NET (SQL authentication)** area.

You need to manually enter the password for your SQL database connection string, because the password isn't pasted automatically.

Important

This example is simplified for the tutorial. For production use, we recommend that you use [Azure Key Vault](#) to store connection string information or [use Azure EntraID for SQL authentication](#).

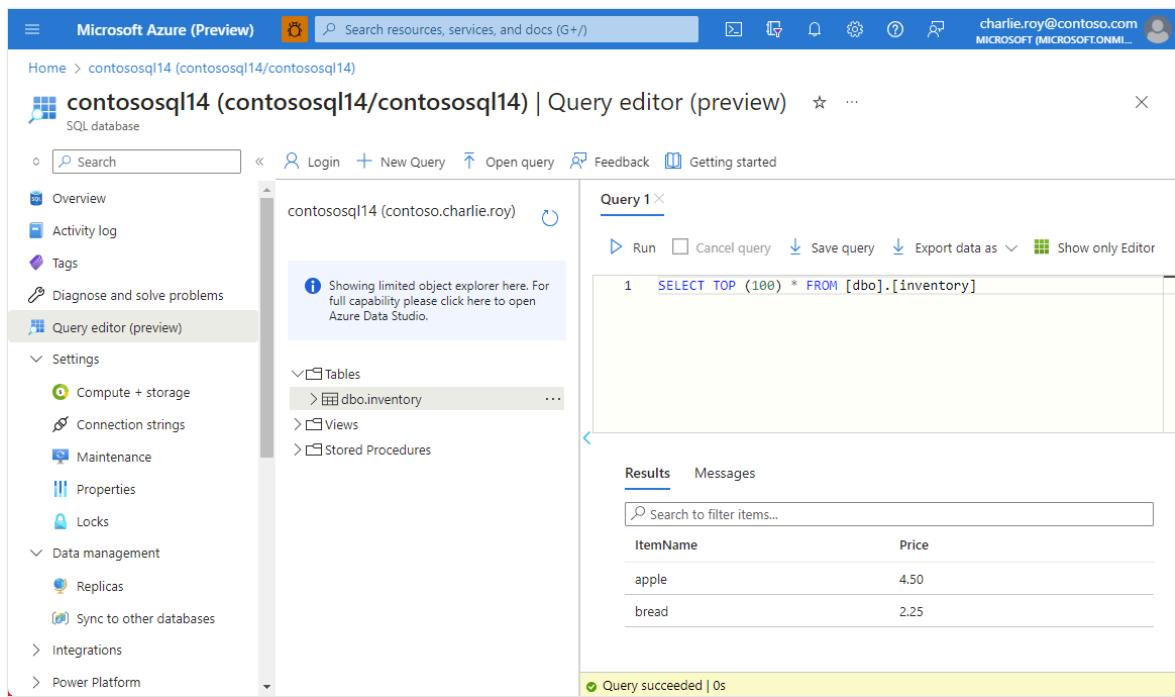
Build and run the project

1. In VS Code, go to the **Run and debug tab** and run the project.
2. Go back to your Azure Cache for Redis instance in the Azure portal, and select the **Console** button to enter the Redis console. Try using some `SET` commands:
 - `SET apple 5.25`
 - `SET bread 2.25`
 - `SET apple 4.50`
3. Back in VS Code, the triggers are being registered. To validate that the triggers are working:
 - a. Go to the SQL database in the Azure portal.
 - b. On the resource menu, select **Query editor**.
 - c. For **New Query**, create a query with the following SQL command to view the top 100 items in the inventory table:

```
SQL
```

```
SELECT TOP (100) * FROM [dbo].[inventory]
```

Confirm that the items written to your Azure Cache for Redis instance appear here.



Deploy the code to your function app

This tutorial builds on the previous tutorial. For more information, see [Deploy code to an Azure function](#).

1. In VS Code, go to the **Azure** tab.
2. Find your subscription and expand it. Then, find the **Function App** section and expand that.
3. Select and hold (or right-click) your function app, and then select **Deploy to Function App**.

Add connection string information

This tutorial builds on the previous tutorial. For more information on the `redisConnectionString`, see [Add connection string information](#).

1. Go to your function app in the Azure portal. On the resource menu, select **Environment variables**.
2. In the **App Settings** pane, enter **SQLConnectionString** as a new field. For **Value**, enter your connection string.
3. Select **Apply**.
4. Go to the **Overview** blade and select **Restart** to restart the app with the new connection string information.

Verify deployment

After the deployment finishes, go back to your Azure Cache for Redis instance and use `SET` commands to write more values. Confirm that they also appear in your SQL database.

If you want to confirm that your function app is working properly, go to the app in the portal and select **Log stream** from the resource menu. You should see the triggers running there, and the corresponding updates being made to your SQL database.

If you ever want to clear the SQL database table without deleting it, you can use the following SQL query:

SQL

```
TRUNCATE TABLE [dbo].[inventory]
```

Clean up resources

If you want to continue to use the resources you created in this article, keep the resource group.

Otherwise, if you're finished with the resources, you can delete the Azure resource group that you created to avoid charges.

i Important

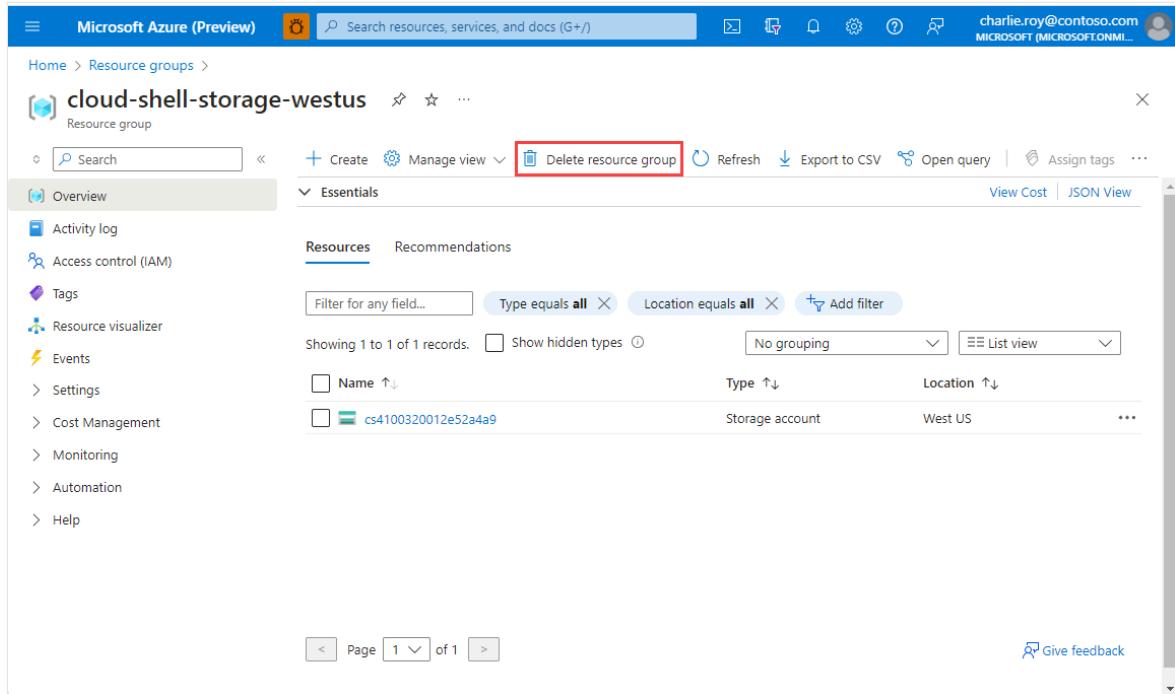
Deleting a resource group is irreversible. When you delete a resource group, all the resources in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the resources inside an existing resource group that contains resources you want to keep, you can delete each resource individually instead of deleting the resource group.

To delete a resource group

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the resource group you want to delete.

If there are many resource groups, use the **Filter for any field...** box, type the name of your resource group you created for this article. Select the resource group in the

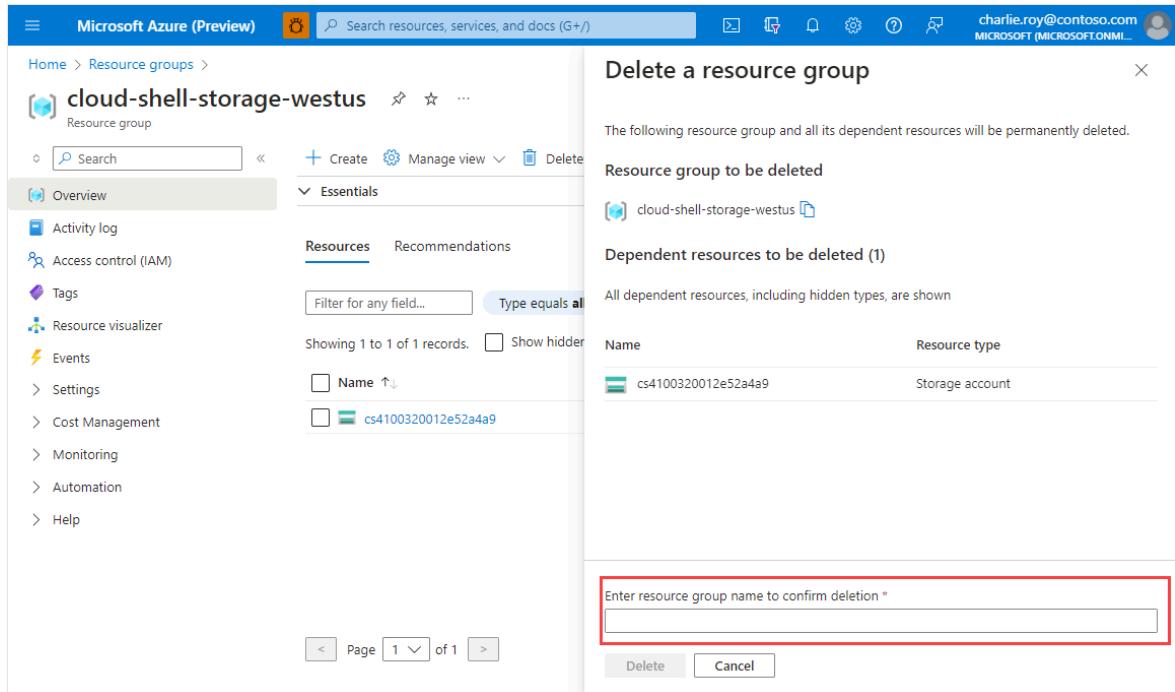
results list.



The screenshot shows the Azure portal interface for a resource group named 'cloud-shell-storage-westus'. The left sidebar contains navigation links like Home, Resource groups, Activity log, Access control (IAM), Tags, Resource visualizer, Events, Settings, Cost Management, Monitoring, Automation, and Help. The main content area has tabs for Overview and Essentials. Under Overview, there are sections for Resources and Recommendations. A search bar at the top allows filtering by field, type, location, and adding filters. A table lists one record: 'cs4100320012e52a4a9' (Storage account, West US). At the top right, there are buttons for Refresh, Export to CSV, Open query, Assign tags, View Cost, and JSON View. The 'Delete resource group' button in the top right is highlighted with a red box.

3. Select Delete resource group.

4. You're asked to confirm the deletion of the resource group. Type the name of your resource group to confirm, and then select **Delete**.



The screenshot shows the 'Delete a resource group' dialog box. It starts with a message: 'The following resource group and all its dependent resources will be permanently deleted.' Below this, it says 'Resource group to be deleted' and shows 'cloud-shell-storage-westus'. It then lists 'Dependent resources to be deleted (1)' as 'All dependent resources, including hidden types, are shown'. A table shows one entry: 'Name' 'cs4100320012e52a4a9' and 'Resource type' 'Storage account'. At the bottom, there is an input field labeled 'Enter resource group name to confirm deletion *' which is highlighted with a red box. Below the input field are 'Delete' and 'Cancel' buttons.

After a few moments, the resource group and all of its resources are deleted.

Summary

This tutorial and [Get started with Azure Functions triggers in Azure Cache for Redis](#) show how to use Azure Cache for Redis to trigger Azure function apps. They also show how to use Azure Cache for Redis as a write-behind cache with Azure SQL Database. Using Azure Cache for Redis with Azure Functions is a powerful combination that can solve many integration and performance problems.

Related content

- [Overview of Azure functions for Azure Cache for Redis](#)
 - [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Add messages to an Azure Storage queue using Functions

Article • 07/02/2024

In Azure Functions, input and output bindings provide a declarative way to make data from external services available to your code. In this article, you use an output binding to create a message in a queue when an HTTP request triggers a function. You use Azure storage container to view the queue messages that your function creates.

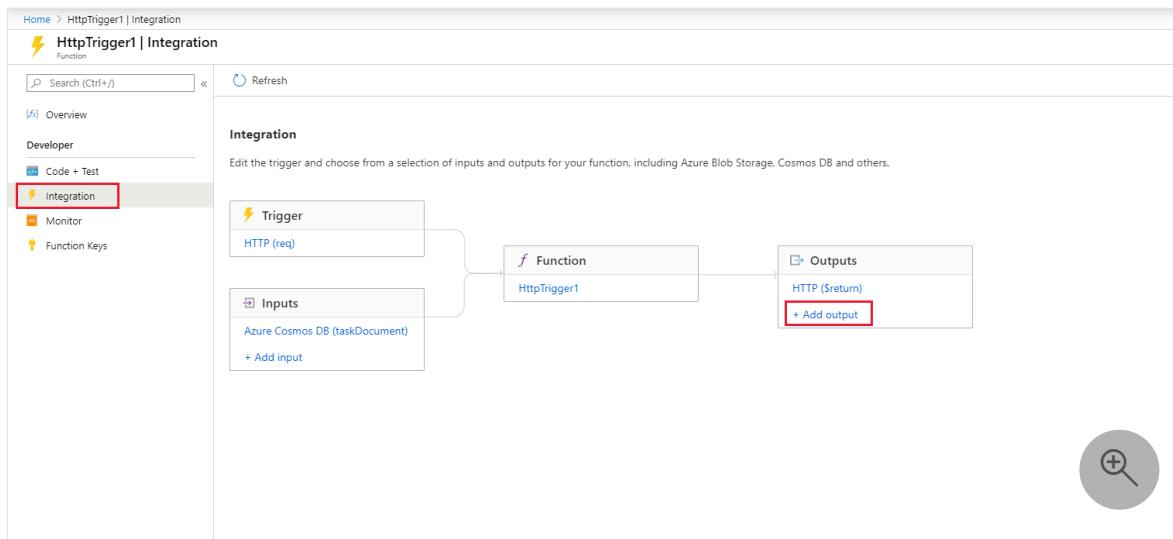
Prerequisites

- An Azure subscription. If you don't have one, create a [free account](#) before you begin.
- Follow the directions in [Create your first function in the Azure portal](#), omitting the **Clean up resources** step, to create the function app and function to use in this article.

Add an output binding

In this section, you use the portal UI to add an Azure Queue Storage output binding to the function you created in the prerequisites. This binding makes it possible to write minimal code to create a message in a queue. You don't need to write code for such tasks as opening a storage connection, creating a queue, or getting a reference to a queue. The Azure Functions runtime and queue output binding take care of those tasks for you.

1. In the Azure portal, search for and select the function app that you created in [Create your first function from the Azure portal](#).
2. In your function app, select the function that you created.
3. Select **Integration**, and then select **+ Add output**.



4. Select the **Azure Queue Storage** binding type and add the settings as specified in the table that follows this screenshot:

Create Output

Start by selecting the type of output binding you want to add.

Binding Type

Azure Queue Storage

Azure Queue Storage details

Message parameter name* (i)

outputQueueItem

Queue name* (i)

outqueue

Storage account connection* (i)

AzureWebJobsStorage

New

OK **Cancel**

Setting	Suggested value	description
Message parameter name	outputQueueItem	The name of the output binding parameter.
Queue name	outqueue	The name of the queue to connect to in your storage account.
Storage account connection	AzureWebJobsStorage	You can use the existing storage account connection used by your function app or create a new one.

5. Select **OK** to add the binding.

Now that you have an output binding defined, you need to update the code to use the binding to add messages to a queue.

Add code that uses the output binding

In this section, you add code that writes a message to the output queue. The message includes the value passed to the HTTP trigger in the query string. For example, if the query string includes `name=Azure`, the queue message is *Name passed to the function: Azure*.

1. In your function, select **Code + Test** to display the function code in the editor.
2. Update the function code, according to your function language:

C#

Add an `outputQueueItem` parameter to the method signature as shown in the following example:

cs

```
public static async Task<IActionResult> Run(HttpContext req,
      ICollector<string> outputQueueItem, ILogger log)
{
    ...
}
```

In the body of the function, just before the `return` statement, add code that uses the parameter to create a queue message:

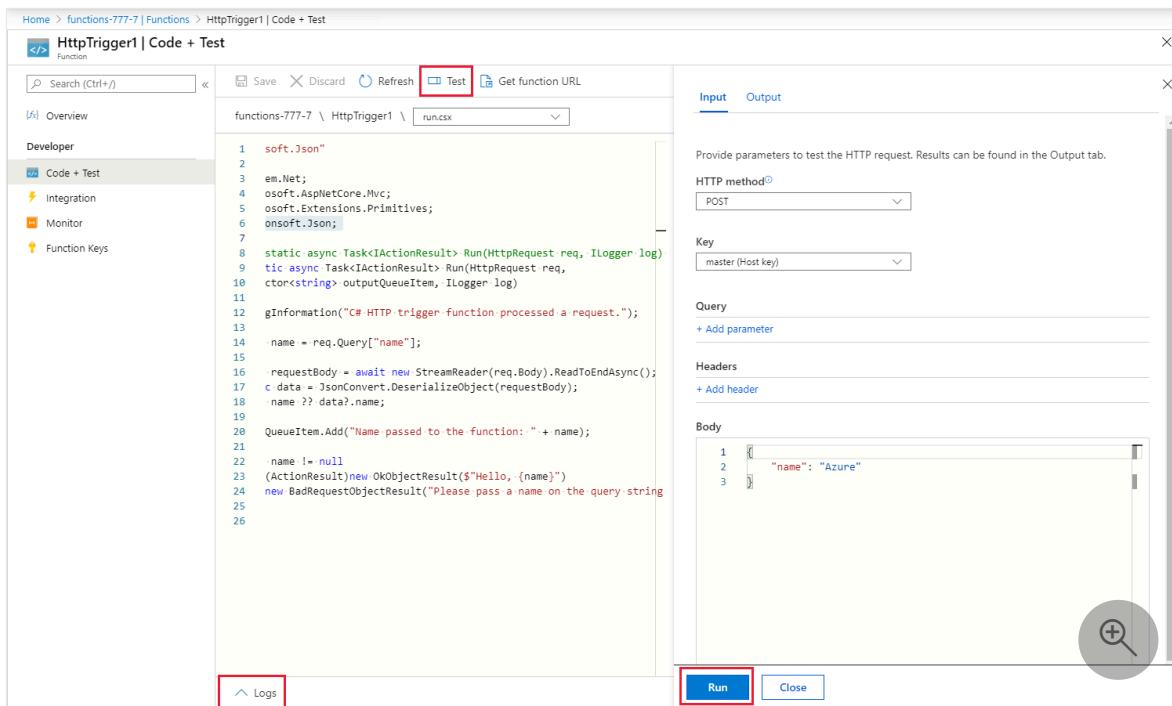
CS

```
outputQueueItem.Add("Name passed to the function: " + name);
```

3. Select **Save** to save your changes.

Test the function

1. After the code changes are saved, select **Test**.
2. Confirm that your test matches this screenshot, and then select **Run**.



Notice that the **Request body** contains the `name` value *Azure*. This value appears in the queue message created when the function is invoked.

As an alternative to selecting **Run**, you can call the function by entering a URL in a browser and specifying the `name` value in the query string. This browser method is shown in [Create your first function from the Azure portal](#).

3. Check the logs to make sure that the function succeeded.

A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You use storage account to verify that the queue and a message in it were created.

Find the storage account connected to AzureWebJobsStorage

1. In your function app, expand **Settings**, and then select **Environment variables**.
2. In the **App settings** tab, select **AzureWebJobsStorage**.

The screenshot shows the Azure portal interface for a function app named "function-app-py". On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, and Functions. The "Settings" section is expanded, and "Environment variables" is selected, highlighted with a red box. The main content area is titled "App settings" and shows a table of environment variables:

Name	Value	Deployment slot setting	Source	Delete
APPLICATIONINSIGHTS_CONN...	(Show value)		App Service	[Delete]
AzureWebJobsFeatureFlags	(Show value)		App Service	[Delete]
AzureWebJobsStorage	(Show value)		App Service	[Delete]
DEPLOYMENT_STORAGE_CON...	(Show value)		App Service	[Delete]
FUNCTIONS_EXTENSION_VERS...	(Show value)		App Service	[Delete]

At the bottom, there are "Apply" and "Discard" buttons, and a "Send us your feedback" link.

3. Locate and make note of the account name.

The screenshot shows the "Add/Edit application setting" dialog. The "Name" field contains "AzureWebJobsStorage" and the "Value" field contains "DefaultEndpointsProtocol=https;AccountName=storageaccountmyresbcb9;AccountKey=/fLt9Wtr+z/30JESevURLqh4aiO8...". A red box highlights the "AccountName" part of the value. There's a checkbox for "Deployment slot setting" which is unchecked. A search icon is visible at the bottom right.

Examine the output queue

1. In the resource group for your function app, select the storage account that you're using.
2. Under Queue service, select **Queues**, and select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name**

value of *Azure*, the queue message is *Name passed to the function: Azure*.

3. Run the function again.

A new message appears in the queue.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Related content

In this article, you added an output binding to an existing function. For more information about binding to Queue Storage, see [Queue Storage trigger and bindings](#).

- [Azure Functions triggers and bindings concepts](#)
Learn how Functions integrates with other services.
- [Azure Functions developer reference](#)
Provides more technical information about the Functions runtime and a reference for coding functions and defining triggers and bindings.
- [Code and test Azure Functions locally](#)
Describes the options for developing your functions locally.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Install [.NET Core CLI tools](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings...`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Except for HTTP and timer triggers, bindings are implemented as extension packages. Run the following [dotnet add package](#) command in the Terminal window to add the Storage extension package to your project.

```
Isolated process

Bash

dotnet add package
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues --prerelease
```

Now, you can add the storage output binding to your project.

Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

Isolated worker model

Open the `HttpExample.cs` project file and add the following `MultiResponse` class:

C#

```
public class MultiResponse
{
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Isolated worker model

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
```

```

    {
        var logger = ExecutionContext.GetLogger("HttpExample");
        logger.LogInformation("C# HTTP trigger function processed a
request.");
    }

        var message = "Welcome to Azure Functions!";

        var response = req.CreateResponse(HttpStatusCode.OK);
        response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
        response.WriteString(message);

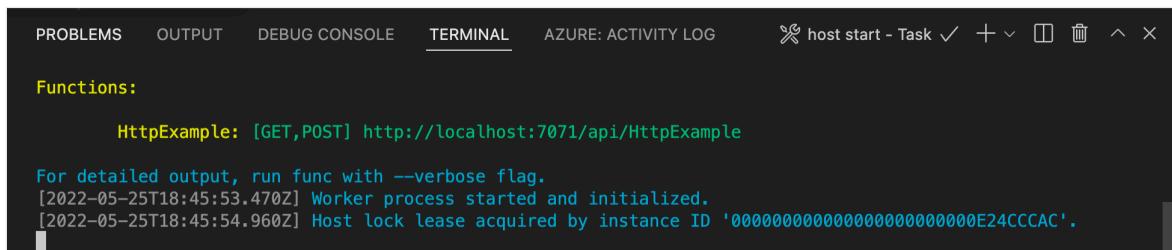
        // Return a response to both HTTP trigger and storage output
binding.
        return new MultiResponse()
    {
        // Write a single message.
        Messages = new string[] { message },
        HttpResponseMessage = response
    };
}
}

```

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure. If you don't already have Core Tools installed locally, you are prompted to install it the first time you run your project.

1. To call your function, press **F5** to start the function app project. The **Terminal** panel displays the output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    AZURE: ACTIVITY LOG
✖ host start - Task ✓ + ▾ □ ▢ ▲ ×

Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

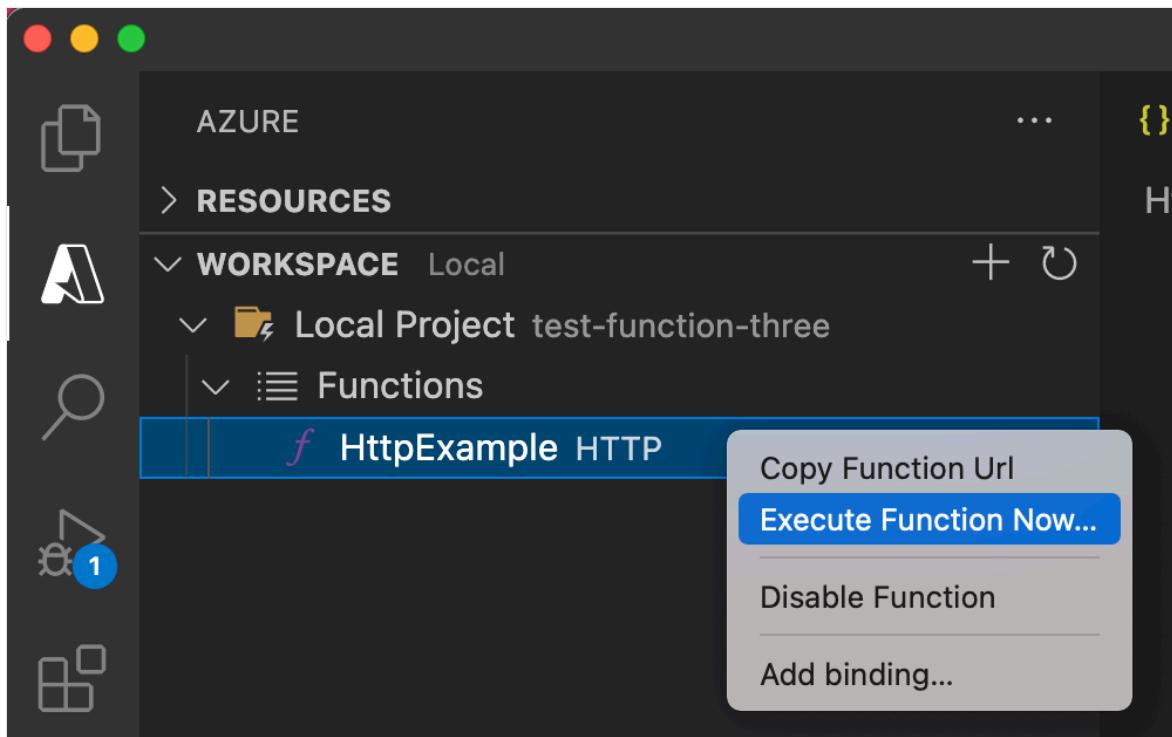
For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.

```

If you don't already have Core Tools installed, select **Install** to install Core Tools when prompted to do so.

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

- With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Windows) or `ctrl - click` (macOS) the `HttpExample` function and choose **Execute Function Now....**

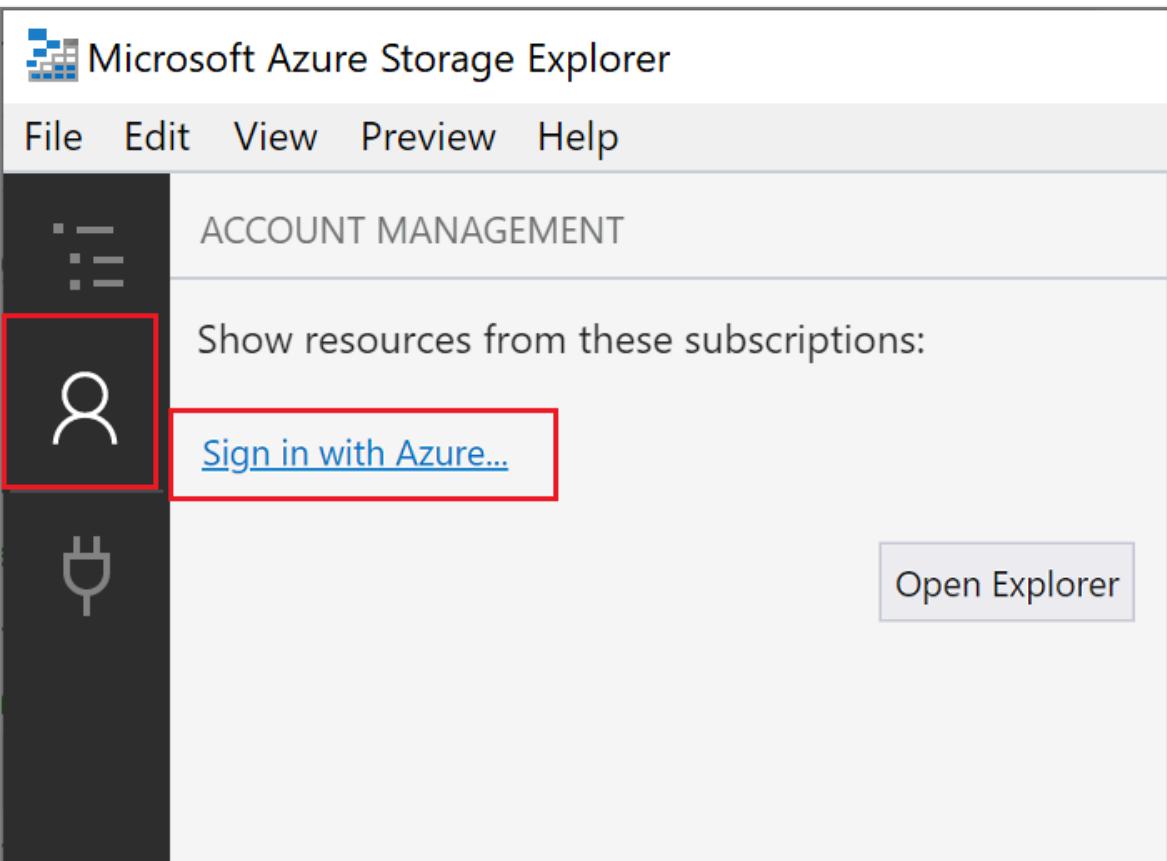


- In the **Enter request body**, press `Enter` to send a request message to your function.
- When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in the **Terminal** panel.
- Press `Ctrl + C` to stop Core Tools and disconnect the debugger.

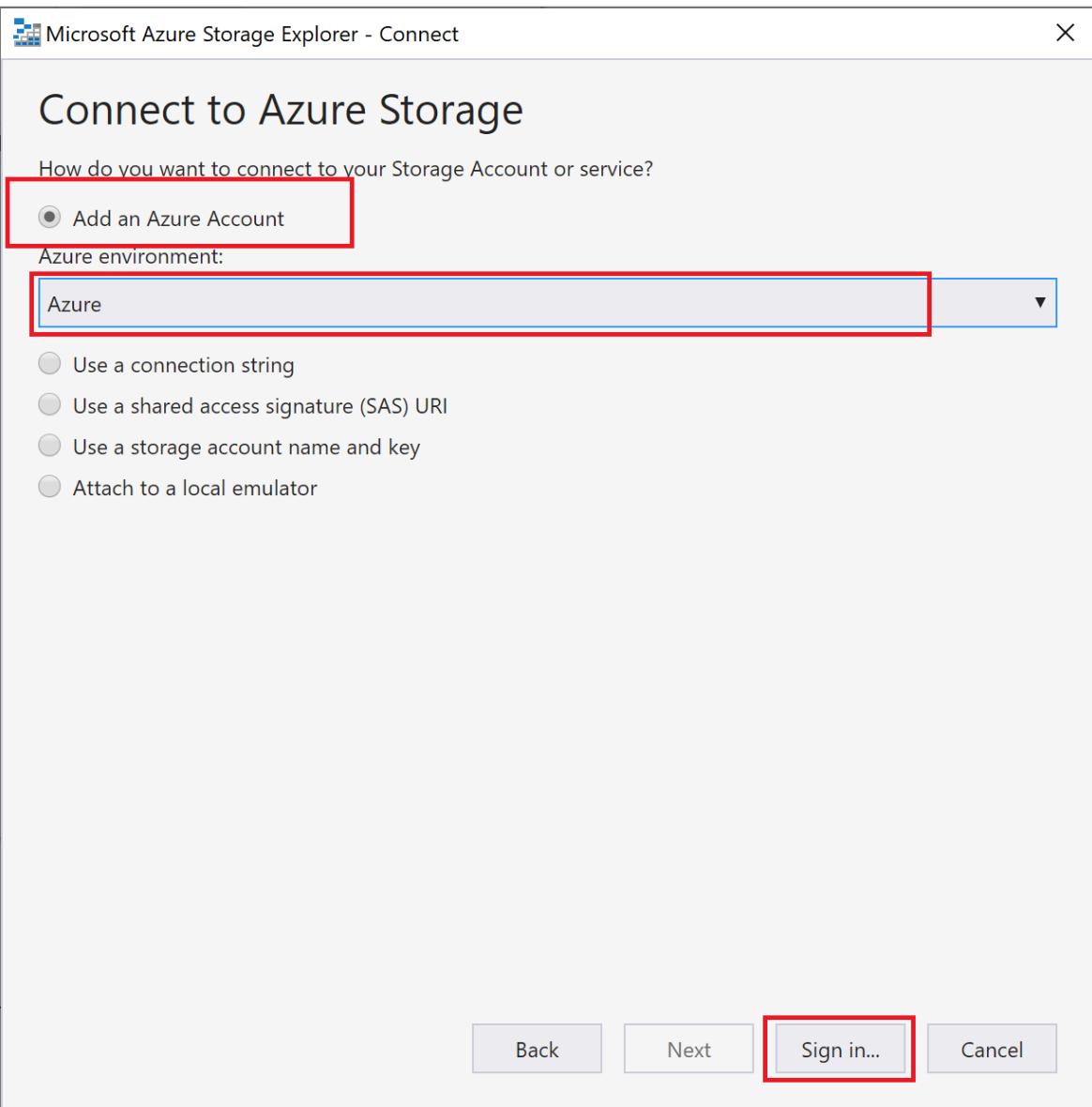
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

- Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

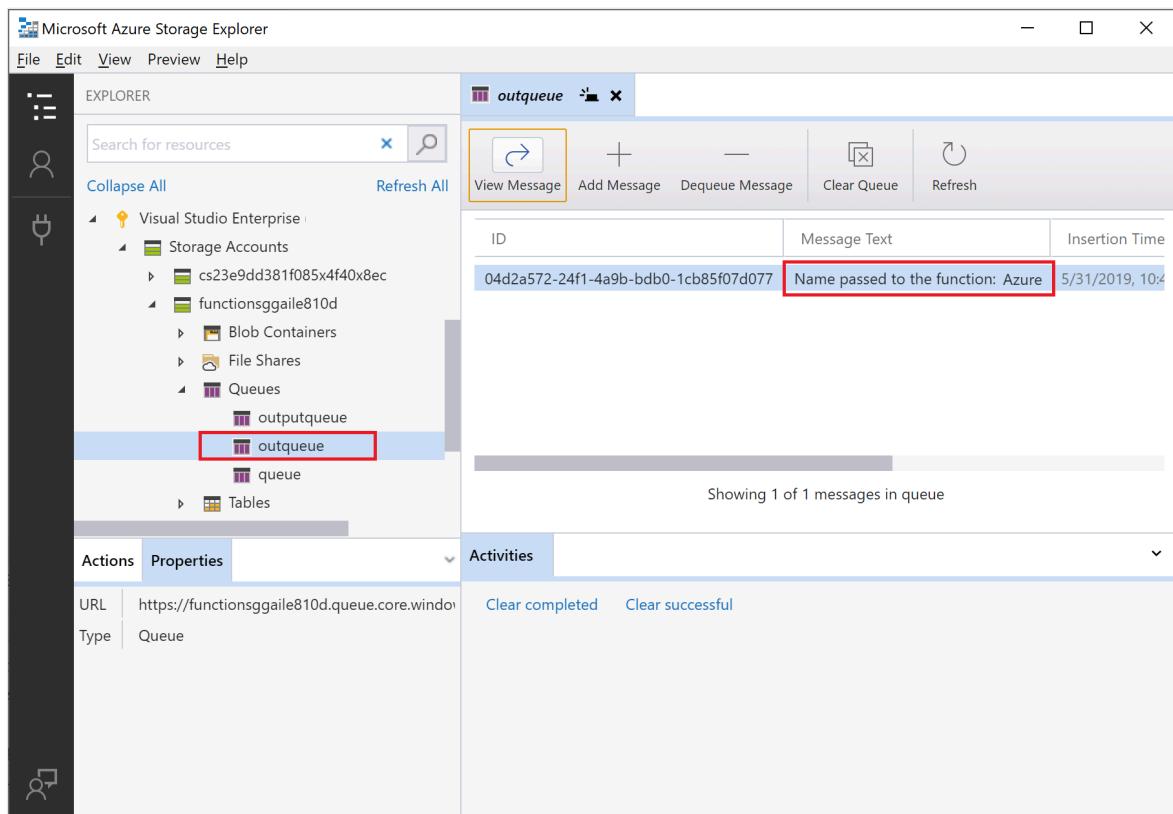


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

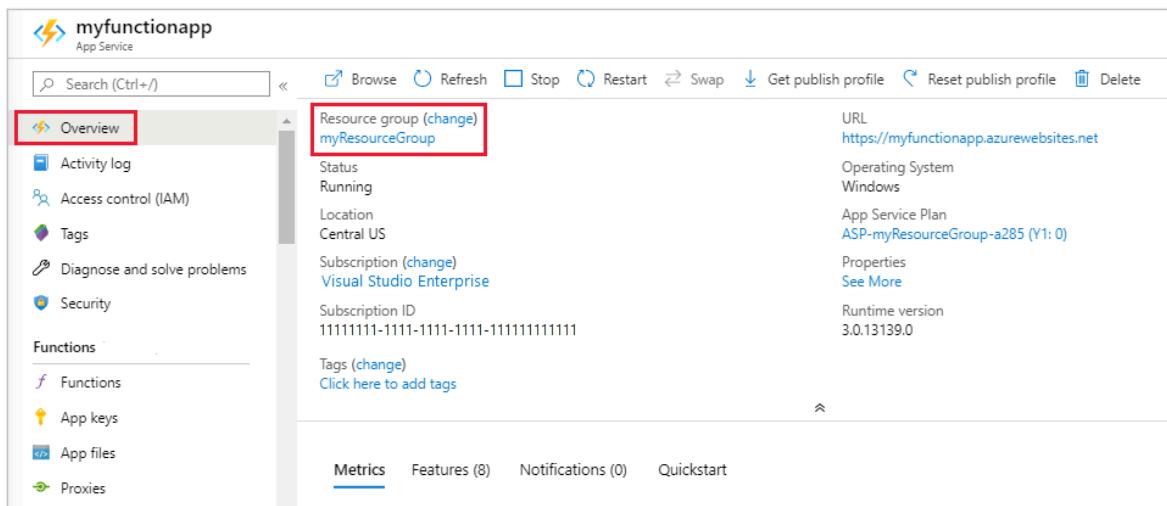
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the "Overview" tab, which displays details about the app: Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), Subscription ID (11111111-1111-1111-1111-111111111111), and Tags (with a link to add tags). A red box highlights the "Resource group (change)" link, which points to "myResourceGroup". To the right of the resource group details, there are sections for URL (<https://myfunctionapp.azurewebsites.net>), Operating System (Windows), App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)), Properties, and Runtime version (3.0.13139.0).

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in C#.](#)
- [Azure Functions C# developer reference](#)

Connect functions to Azure Storage using Visual Studio

Article • 03/31/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

This article shows you how to use Visual Studio to connect the function you created in the [previous quickstart article](#) to Azure Storage. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the Storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Prerequisites

Before you start this article, you must:

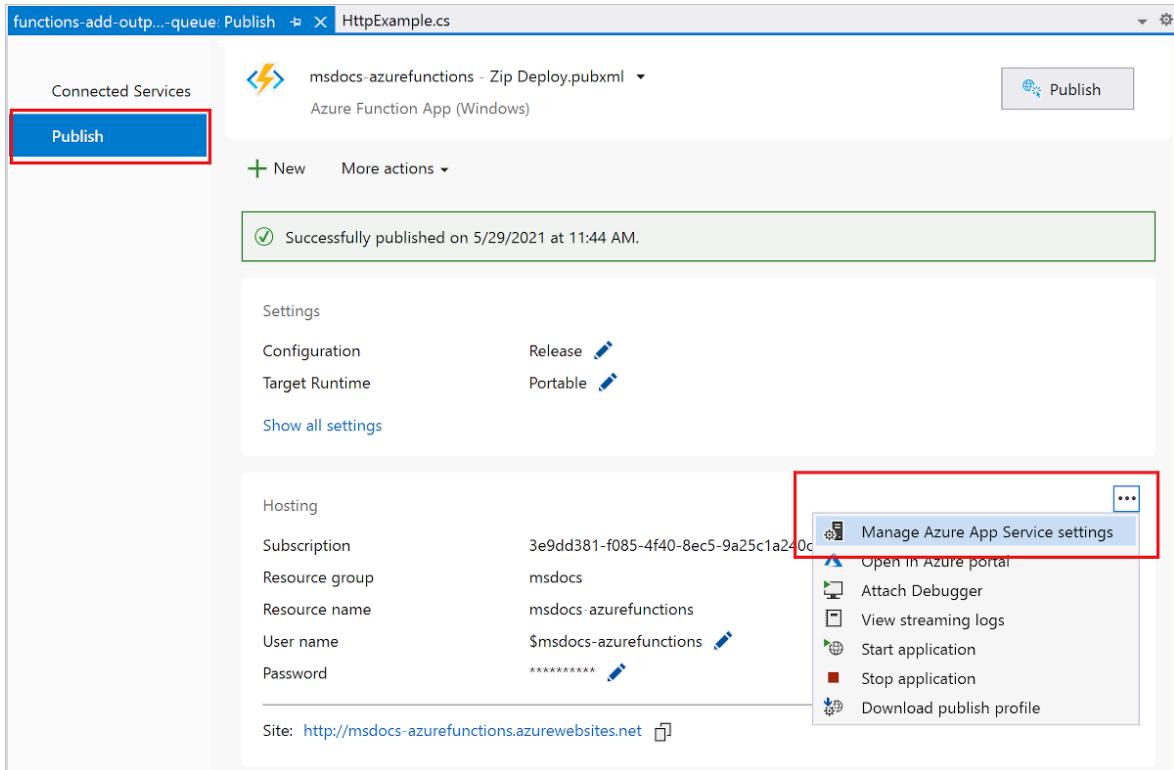
- Complete [part 1 of the Visual Studio quickstart](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Sign in to your Azure subscription from Visual Studio.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required Storage account. The connection string for this account is stored securely in app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your Storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. In **Solution Explorer**, right-click the project and select **Publish**.

2. In the Publish tab under **Hosting**, expand the three dots (...) and select **Manage Azure App Service settings**.



3. Under **AzureWebJobsStorage**, copy the **Remote** string value to **Local**, and then select **OK**.

The storage binding, which uses the `AzureWebJobsStorage` setting for the connection, can now connect to your Queue storage when running locally.

Register binding extensions

Because you're using a Queue storage output binding, you need the Storage bindings extension installed before you run the project. Except for HTTP and timer triggers, bindings are implemented as extension packages.

1. From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.
2. In the console, run the following `Install-Package` command to install the Storage extensions:

Isolated worker model

Bash

```
Install-Package  
Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues
```

Now, you can add the storage output binding to your project.

Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.

Isolated worker model

Open the *HttpExample.cs* project file and add the following `MultiResponse` class:

C#

```
public class MultiResponse  
{  
    [QueueOutput("outqueue", Connection = "AzureWebJobsStorage")]  
    public string[] Messages { get; set; }  
    public HttpResponseMessage HttpResponseMessage { get; set; }  
}
```

The `MultiResponse` class allows you to write to a storage queue named `outqueue` and an HTTP success message. Multiple messages could be sent to the queue because the `QueueOutput` attribute is applied to a string array.

The `Connection` property sets the connection string for the storage account. In this case, you could omit `Connection` because you're already using the default storage account.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Replace the existing `HttpExample` class with the following code:

C#

```
[Function("HttpExample")]
public static MultiResponse
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpExample");
    logger.LogInformation("C# HTTP trigger function processed a
request.");

    var message = "Welcome to Azure Functions!";

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-
8");
    response.WriteString(message);

    // Return a response to both HTTP trigger and storage output
binding.
    return new MultiResponse()
    {
        // Write a single message.
        Messages = new string[] { message },
        HttpResponseMessage = response
    };
}
```

Run the function locally

1. To run your function, press `F5` in Visual Studio. You might need to enable a firewall exception so that the tools can handle HTTP requests. Authorization levels are never enforced when you run a function locally.
2. Copy the URL of your function from the Azure Functions runtime output.

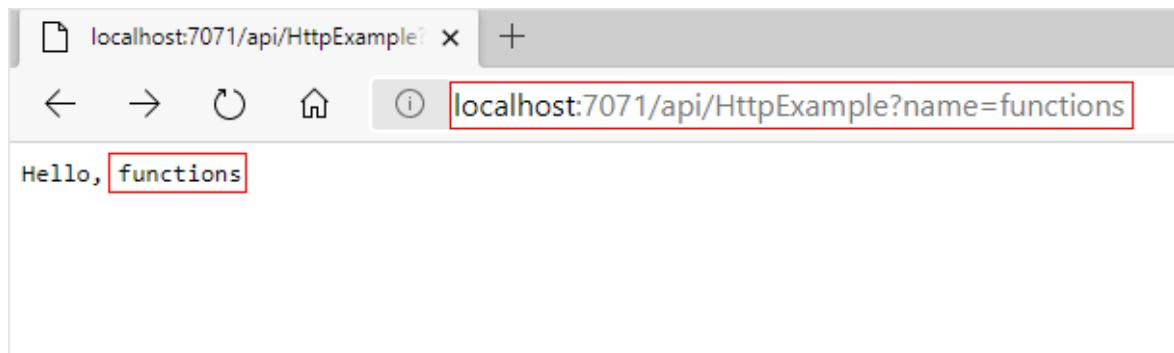
```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.49.0\cli_x64\func.exe
[5/27/2020 7:53:39 AM] Loading functions metadata
[5/27/2020 7:53:39 AM] 1 functions loaded
[5/27/2020 7:53:39 AM] Generating 1 job function(s)
[5/27/2020 7:53:40 AM] Found the following functions:
[5/27/2020 7:53:40 AM] FunctionApp2.HttpExample.Run
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Initializing function HTTP routes
[5/27/2020 7:53:40 AM] Mapped function route 'api/HttpExample' [get,post] to 'HttpExample'
[5/27/2020 7:53:40 AM]
[5/27/2020 7:53:40 AM] Host initialized (691ms)
[5/27/2020 7:53:40 AM] Host started (712ms)
[5/27/2020 7:53:40 AM] Job host started
Hosting environment: Development
Content root path: C:\source\repos\FunctionApp\FunctionApp\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[5/27/2020 7:53:47 AM] Host lock lease acquired by instance ID '00000000000000000000000000000000FB2CECE'.
```

- Paste the URL for the HTTP request into your browser's address bar and run the request. The following image shows the response in the browser to the local GET request returned by the function:



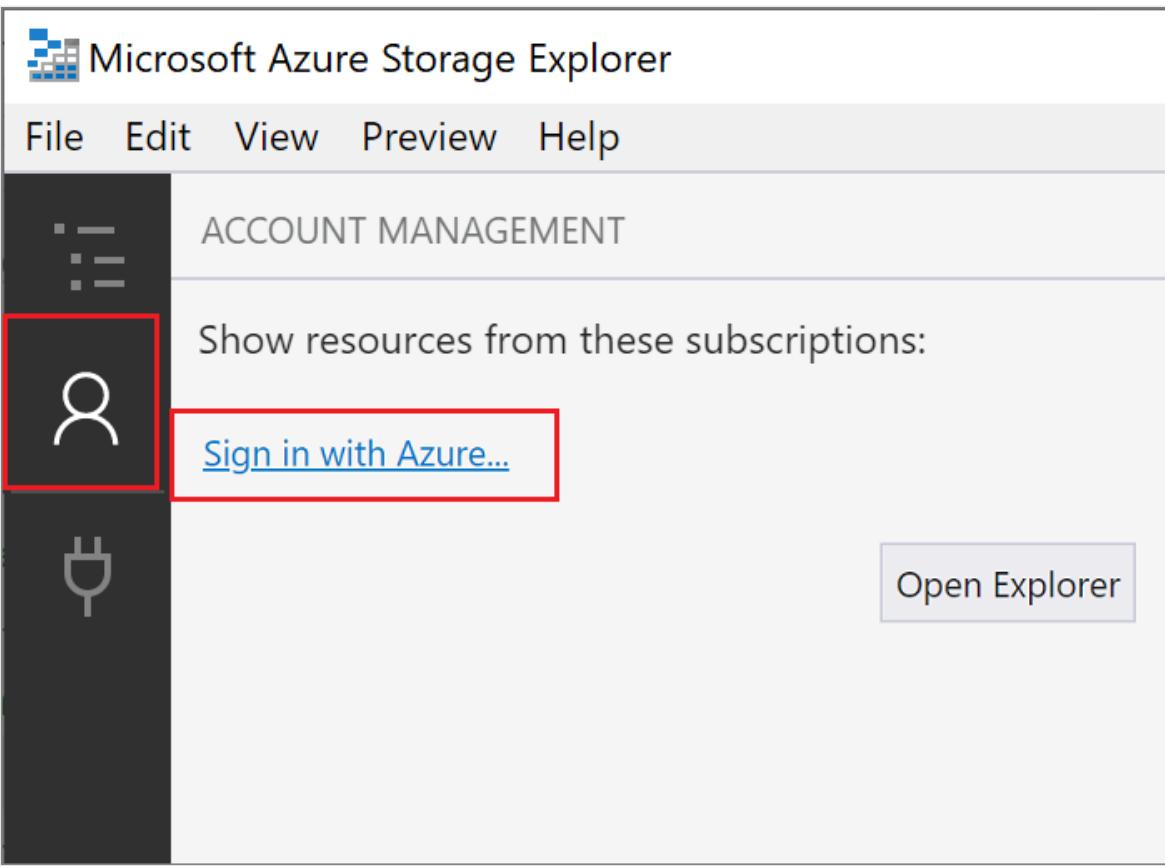
- To stop debugging, press **Shift + F5** in Visual Studio.

A new queue named `outqueue` is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

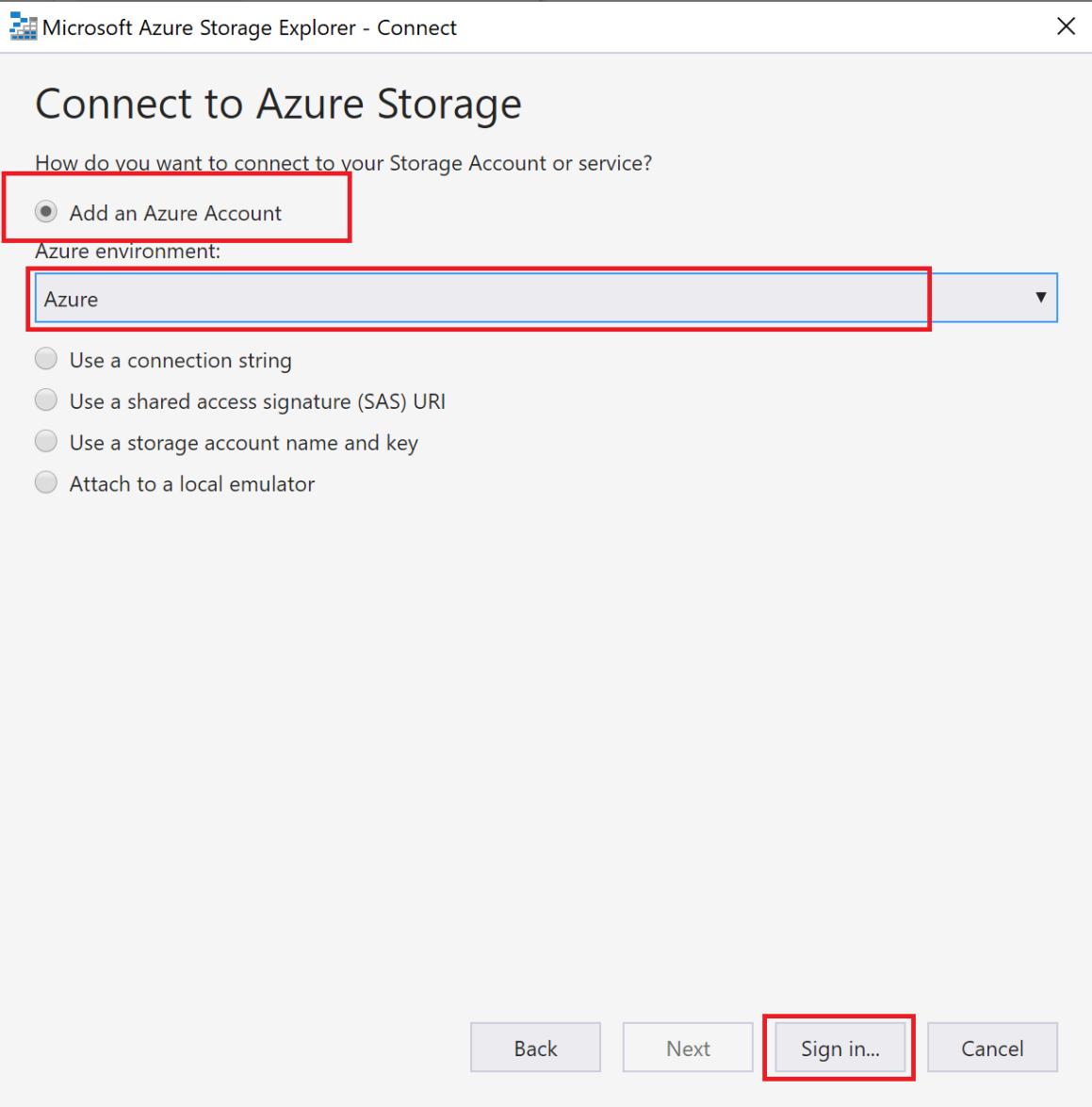
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

- Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

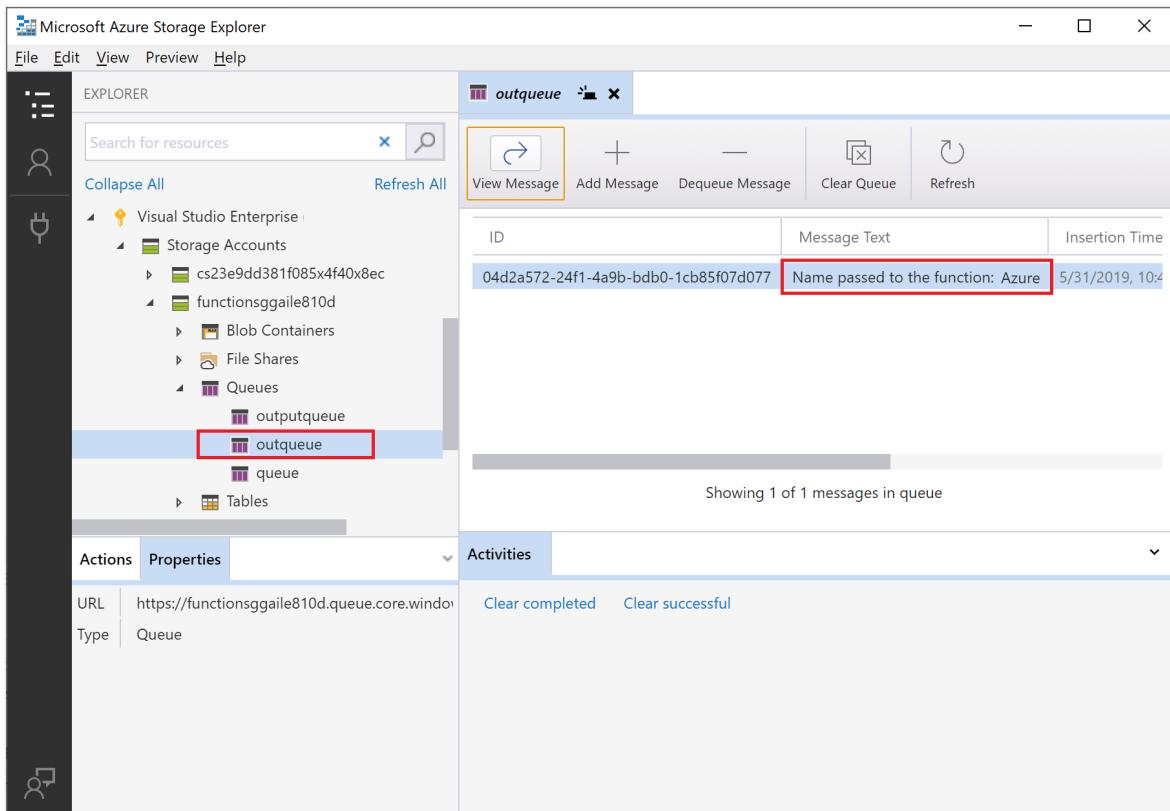


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Storage Explorer, expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default `name` value of *Azure*, the queue message is *Name passed to the function: Azure*.



- Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

- In **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
- After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL.
- Again **view the message in the storage queue** to verify that the output binding again generates a new message in the queue.

Clean up resources

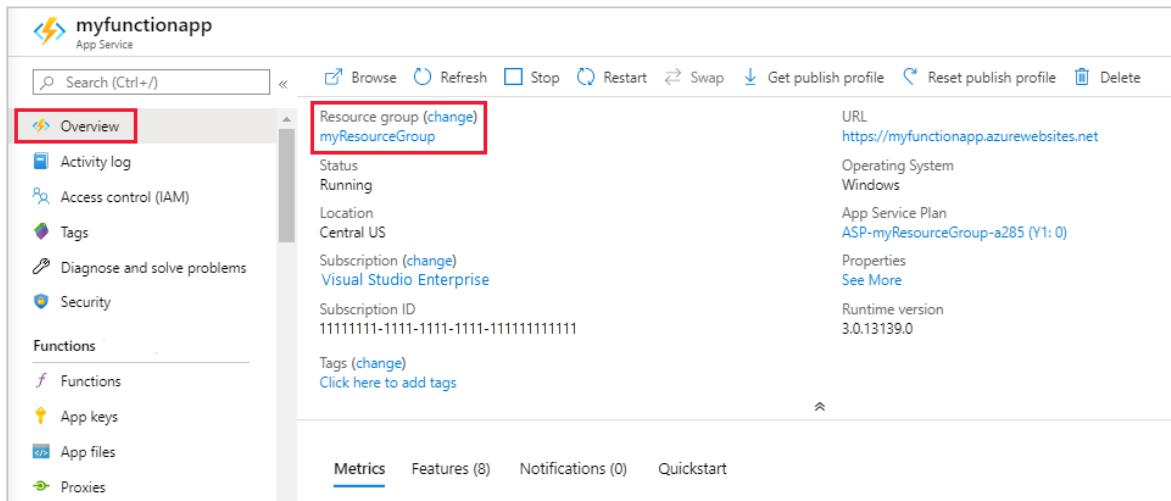
Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you've created in this quickstart, don't clean up the resources.

Resources in Azure refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You might be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In the Azure portal, go to the **Resource group** page.

To get to that page from the function app page, select the **Overview** tab, and then select the link under **Resource group**.



The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The left sidebar has a "Functions" section with "myfunctionapp" selected. The main content area is the "Overview" tab, which displays the following details:

Setting	Value
URL	https://myfunctionapp.azurewebsites.net
Operating System	Windows
App Service Plan	ASP-myResourceGroup-a285 (Y1: 0)
Properties	See More
Runtime version	3.0.13139.0

Below these details, there is a "Resource group (change)" section with "myResourceGroup" highlighted. The "Subscription" section shows "Visual Studio Enterprise" and the "Subscription ID" is listed as "1111111-1111-1111-1111-111111111111". There is also a "Tags (change)" section with a link "Click here to add tags".

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this article.

2. In the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
3. Select **Delete resource group** and follow the instructions.

Deletion might take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. To learn more about developing Functions, see [Develop Azure Functions using Visual Studio](#).

Next, you should enable Application Insights monitoring for your function app:

[Enable Application Insights integration](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 04/25/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no additional costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replace `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open `local.settings.json` file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the `local.settings.json` file contains secrets downloaded from Azure, always exclude this file from source control. The `.gitignore` file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

When using the [Python v2 programming model](#), binding attributes are defined directly in the `function_app.py` file as decorators. From the previous quickstart, your `function_app.py` file already contains one decorator-based binding:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="hello", auth_level=func.AuthLevel.ANONYMOUS)
```

The `route` decorator adds `HttpTrigger` and `HttpOutput` binding to the function, which enables your function be triggered when http requests hit the specified route.

To write to an Azure Storage queue from this function, add the `queue_output` decorator to your function code:

Python

```
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
```

In the decorator, `arg_name` identifies the binding parameter referenced in your code, `queue_name` is name of the queue that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Storage account. In quickstarts you use the same storage account as the function app, which is in the `AzureWebJobsStorage` setting (from `local.settings.json` file). When the `queue_name` doesn't exist, the binding creates it on first use.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Update `HttpExample\function_app.py` to match the following code, add the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="HttpExample")
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
def HttpExample(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) ->
func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        msg.set(name)
        return func.HttpResponse(f"Hello, {name}. This HTTP triggered
function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
            status_code=200
        )
```

The `msg` parameter is an instance of the `azure.functions.Out class`. The `set` method writes a string message to the queue. In this case, it's the `name` passed to the function in the URL query string.

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the *LocalFunctionProj* folder.

```
Console  
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools  
Core Tools Version: 4.0.5049 Commit hash: N/A (64-bit)  
Function Runtime Version: 4.15.2.20177  
  
[2023-03-17T03:27:12.372Z] Worker process started and initialized.  
  
Functions:  
  
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.
3. When you're done, press `Ctrl + C` and type `y` to stop the functions host.

💡 Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[1.*, 2.0.0)"  
    }  
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in *host.json*.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's *local.setting.json* file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

bash

Bash

```
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>"
```

2. (Optional) Use the [az storage queue list](#) command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

Azure CLI

```
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

bash

Azure CLI

```
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}')` | base64 --decode`
```

Because the message body is stored [base64 encoded](#), the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

Now that you've verified locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the `LocalFunctionsProj` folder, use the `func azure functionapp publish` command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`.

The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you've finished, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)

Quickstart: Create an app showing GitHub star count with Azure Functions and SignalR Service via C#

Article • 05/16/2024

In this article, you learn how to use SignalR Service and Azure Functions to build a serverless application with C# to broadcast messages to clients.



Prerequisites

The following prerequisites are needed for this quickstart:

- Visual Studio Code, or other code editor. If you don't already have Visual Studio Code installed, [download Visual Studio Code here](#).
- An Azure subscription. If you don't have an Azure subscription, [create one for free](#) before you begin.
- [Azure Functions Core Tools](#)
- [.NET Core SDK](#)

Create an Azure SignalR Service instance

In this section, you create a basic Azure SignalR instance to use for your app. The following steps use the Azure portal to create a new instance, but you can also use the Azure CLI. For more information, see the [az signalr create](#) command in the [Azure SignalR Service CLI Reference](#).

1. Sign in to the [Azure portal](#).
2. In the upper-left side of the page, select **+ Create a resource**.
3. On the **Create a resource** page, in the **Search services and marketplace** text box, enter **signalr** and then select **SignalR Service** from the list.
4. On the **SignalR Service** page, select **Create**.

5. On the **Basics** tab, you enter the essential information for your new SignalR Service instance. Enter the following values:

[+] Expand table

Field	Suggested Value	Description
Subscription	Choose your subscription	Select the subscription you want to use to create a new SignalR Service instance.
Resource group	Create a resource group named <i>SignalRTTestResources</i>	Select or create a resource group for your SignalR resource. It's useful to create a new resource group for this tutorial instead of using an existing resource group. To free resources after completing the tutorial, delete the resource group. Deleting a resource group also deletes all of the resources that belong to the group. This action can't be undone. Before you delete a resource group, make certain that it doesn't contain resources you want to keep. For more information, see Using resource groups to manage your Azure resources .
Resource name	<i>testsignalr</i>	Enter a unique resource name to use for the SignalR resource. If <i>testsignalr</i> is already taken in your region, add a digit or character until the name is unique. The name must be a string of 1 to 63 characters and contain only numbers, letters, and the hyphen (-) character. The name can't start or end with the hyphen character, and consecutive hyphen characters aren't valid.
Region	Choose your region	Select the appropriate region for your new SignalR Service instance. Azure SignalR Service isn't currently available in all regions. For more information, see Azure SignalR Service region availability
Pricing tier	Select Change and then choose Free (Dev/Test Only) . Choose Select to confirm your choice of pricing tier.	Azure SignalR Service has three pricing tiers: Free, Standard, and Premium. Tutorials use the Free tier, unless noted otherwise in the prerequisites. For more information about the functionality

Field	Suggested Value	Description
		differences between tiers and pricing, see Azure SignalR Service pricing ↗
Service mode	Choose the appropriate service mode	<p>Use Default when you host the SignalR hub logic in your web apps and use SignalR service as a proxy. Use Serverless when you use Serverless technologies such as Azure Functions to host the SignalR hub logic.</p> <p>Classic mode is only for backward compatibility and isn't recommended to use.</p> <p>For more information, see Service mode in Azure SignalR Service.</p>

You don't need to change the settings on the **Networking** and **Tags** tabs for the SignalR tutorials.

6. Select the **Review + create** button at the bottom of the **Basics** tab.
7. On the **Review + create** tab, review the values and then select **Create**. It takes a few moments for deployment to complete.
8. When the deployment is complete, select the **Go to resource** button.
9. On the SignalR resource page, select **Keys** from the menu on the left, under **Settings**.
10. Copy the **Connection string** for the primary key. You need this connection string to configure your app later in this tutorial.

Setup and run the Azure Function locally

You need the Azure Functions Core Tools for this step.

1. Create an empty directory and change to the directory with the command line.
2. Initialize a new project.

```

In-process

Bash

# Initialize a function project
func init --worker-runtime dotnet

# Add SignalR Service package reference to the project

```

```
dotnet add package  
Microsoft.Azure.WebJobs.Extensions.SignalRService
```

3. Using your code editor, create a new file with the name *Function.cs*. Add the following code to *Function.cs*:

In-process

C#

```
using System;  
using System.IO;  
using System.Linq;  
using System.Net.Http;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Http;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.Azure.WebJobs.Extensions.SignalRService;  
using Newtonsoft.Json;  
  
namespace CSharp  
{  
    public static class Function  
    {  
        private static HttpClient httpClient = new HttpClient();  
        private static string Etag = string.Empty;  
        private static string StarCount = "0";  
  
        [FunctionName("index")]  
        public static IActionResult  
GetHomePage([HttpTrigger(AuthorizationLevel.Anonymous)]HttpRequest  
req, ExecutionContext context)  
        {  
            var path = Path.Combine(context.FunctionAppDirectory,  
"content", "index.html");  
            return new ContentResult  
            {  
                Content = File.ReadAllText(path),  
                ContentType = "text/html",  
            };  
        }  
  
        [FunctionName("negotiate")]  
        public static SignalRConnectionInfo Negotiate(  
            [HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequest  
req,  
            [SignalRConnectionInfo(HubName = "serverless")]  
            SignalRConnectionInfo connectionInfo)  
        {
```

```

        return connectionInfo;
    }

    [FunctionName("broadcast")]
    public static async Task Broadcast([TimerTrigger("*/5 * * * *")]
        TimerInfo myTimer,
        [SignalR(HubName = "serverless")]
        IAsyncCollector<SignalRMessage> signalRMessages)
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
"https://api.github.com/repos/azure/azure-signalr");
        request.Headers.UserAgent.ParseAdd("Serverless");
        request.Headers.Add("If-None-Match", Etag);
        var response = await httpClient.SendAsync(request);
        if (response.Headers.Contains("Etag"))
        {
            Etag = response.Headers.GetValues("Etag").First();
        }
        if (response.StatusCode ==
System.Net.HttpStatusCode.OK)
        {
            var result =
JsonConvert.DeserializeObject<GitResult>(await
response.Content.ReadAsStringAsync());
            StarCount = result.StarCount;
        }

        await signalRMessages.AddAsync(
            new SignalRMessage
            {
                Target = "newMessage",
                Arguments = new[] { $"Current star count of
https://github.com/Azure/azure-signalr is: {StarCount}" }
            });
    }

    private class GitResult
    {
        [JsonProperty("stargazers_count")]
        public string StarCount { get; set; }
    }
}

```

The code in *Function.cs* has three functions:

- `GetHomePage` is used to get a website as client.
- `Negotiate` is used by the client to get an access token.
- `Broadcast` is periodically called to get the star count from GitHub and then broadcast messages to all clients.

4. The client interface for this sample is a web page. We render the web page using the `GetHomePage` function by reading HTML content from file `content/index.html`. Now let's create this `index.html` under the `content` subdirectory with the following content:

```
HTML

<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div id="messages"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
    signalr/3.1.7/signalr.min.js"></script>
    <script>
        let messages = document.querySelector('#messages');
        const apiUrl = window.location.origin;
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>

</html>
```

5. Update your `*.csproj` to make the content page in the build output folder.

```
HTML

<ItemGroup>
    <None Update="content/index.html">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
</ItemGroup>
```

6. Azure Functions requires a storage account to work. You can install and run the [Azure Storage Emulator](#). Or you can update the setting to use your real storage account with the following command:

```
Bash
```

```
func settings add AzureWebJobsStorage "<storage-connection-string>"
```

7. It's almost done now. The last step is to set a connection string of the SignalR Service to Azure Function settings.

- Confirm the SignalR Service instance was successfully created by searching for its name in the search box at the top of the portal. Select the instance to open it.

serverlesschat

RESOURCES

- serverlesschat Azure Cosmos DB account
- serverlesschat SignalR
- serverlesschat Storage account
- serverlesschat App Service
- serverlesschat2 Storage account

RESOURCE GROUPS

- serverlesschat Resource group

SERVICES

MARKETPLACE

DOCUMENTATION

Searching all subscriptions. [Change](#)

- Select Keys to view the connection strings for the SignalR Service instance.

serverlesschat - Keys

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Events

Settings

Keys

Quickstart

Host name

serverlesschat.service.signalr.net

Primary

KEY

PRIMARY-KEY

CONNECTION STRING

PRIMARY-CONNECTION-STRING

- Copy the primary connection string, and then run the following command:

```
Bash
```

```
func settings add AzureSignalRConnectionString "<signalr-connection-string>"
```

8. Run the Azure function locally:

Bash

```
func start
```

After the Azure function is running locally, open <http://localhost:7071/api/index>, and you can see the current star count. If you star or unstar in the GitHub, you get a star count refreshing every few seconds.

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know ↗](#).

Next steps

In this quickstart, you built and ran a real-time serverless application locally. Next, learn more about bi-directional communication between clients and Azure Functions with Azure SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Azure Functions Bi-directional communicating sample](#)

[Azure Functions Bi-directional communicating sample for isolated process](#)

Deploy to Azure Function App using Visual Studio

Quickstart: Use Java to create an App showing GitHub star count with Azure Functions and SignalR Service

Article • 01/04/2023

In this article, you'll use Azure SignalR Service, Azure Functions, and Java to build a serverless application to broadcast messages to clients.

ⓘ Note

The code in this article is available on [GitHub](#).

Prerequisites

- A code editor, such as [Visual Studio Code](#).
- An Azure account with an active subscription. If you don't already have an account, [create an account for free](#).
- [Azure Functions Core Tools](#). Used to run Azure Function apps locally.
 - The required SignalR Service bindings in Java are only supported in Azure Function Core Tools version 2.4.419 (host version 2.0.12332) or above.
 - To install extensions, Azure Functions Core Tools requires the [.NET Core SDK](#) installed. However, no knowledge of .NET is required to build Java Azure Function apps.
- [Java Developer Kit](#), version 11
- [Apache Maven](#), version 3.0 or above.

This quickstart can be run on macOS, Windows, or Linux.

Create an Azure SignalR Service instance

In this section, you'll create a basic Azure SignalR instance to use for your app. The following steps use the Azure portal to create a new instance, but you can also use the Azure CLI. For more information, see the [az signalr create](#) command in the [Azure SignalR Service CLI Reference](#).

1. Sign in to the [Azure portal](#).
2. In the upper-left side of the page, select **+ Create a resource**.
3. On the **Create a resource** page, in the **Search services and marketplace** text box, enter **signalr** and then select **SignalR Service** from the list.
4. On the **SignalR Service** page, select **Create**.
5. On the **Basics** tab, you'll enter the essential information for your new SignalR Service instance. Enter the following values:

Field	Suggested Value	Description
Subscription	Choose your subscription	Select the subscription you want to use to create a new SignalR Service instance.
Resource group	Create a resource group named <i>SignalRTTestResources</i>	Select or create a resource group for your SignalR resource. It's useful to create a new resource group for this tutorial instead of using an existing resource group. To free resources after completing the tutorial, delete the resource group. Deleting a resource group also deletes all of the resources that belong to the group. This action can't be undone. Before you delete a resource group, make certain that it doesn't contain resources you want to keep. For more information, see Using resource groups to manage your Azure resources .
Resource name	<i>testsignalr</i>	Enter a unique resource name to use for the SignalR resource. If <i>testsignalr</i> has already been used in your region, add a digit or character until the name is unique. The name must be a string of 1 to 63 characters and contain only numbers, letters, and the hyphen (-) character. The name can't start or end with the hyphen character, and consecutive hyphen characters aren't valid.
Region	Choose your region	Select the appropriate region for your new SignalR Service instance. Azure SignalR Service isn't currently available in all regions. For more information, see Azure SignalR Service region availability

Field	Suggested Value	Description
Pricing tier	Select Change and then choose Free (Dev/Test Only) . Choose Select to confirm your choice of pricing tier.	Azure SignalR Service has three pricing tiers: Free, Standard, and Premium. Tutorials use the Free tier, unless noted otherwise in the prerequisites. For more information about the functionality differences between tiers and pricing, see Azure SignalR Service pricing
Service mode	Choose the appropriate service mode for this tutorial	Use Default for ASP.NET. Use Serverless for Azure Functions or REST API. Classic mode isn't used in the Azure SignalR tutorials. For more information, see Service mode in Azure SignalR Service .

You don't need to change the settings on the **Networking** and **Tags** tabs for the SignalR tutorials.

6. Select the **Review + create** button at the bottom of the **Basics** tab.
7. On the **Review + create** tab, review the values and then select **Create**. It will take a few moments for deployment to complete.
8. When the deployment is complete, select the **Go to resource** button.
9. On the SignalR resource page, select **Keys** from the menu on the left, under **Settings**.
10. Copy the **Connection string** for the primary key. You'll need this connection string to configure your app later in this tutorial.

Configure and run the Azure Function app

Make sure you have Azure Function Core Tools, Java (version 11 in the sample), and Maven installed.

1. Initialize the project using Maven:

Bash

```
mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure -DarchetypeArtifactId=azure-functions-archetype -DjavaVersion=11
```

Maven asks you for values needed to finish generating the project. Provide the following values:

Prompt	Value	Description
groupId	com.signalr	A value that uniquely identifies your project across all projects, following the package naming rules for Java.
artifactId	java	A value that is the name of the jar, without a version number.
version	1.0-SNAPSHOT	Choose the default value.
package	com.signalr	A value that is the Java package for the generated function code. Use the default.

2. Go to the folder `src/main/java/com/signalr` and copy the following code to `Function.java`:

Java

```
package com.signalr;

import com.google.gson.Gson;
import com.microsoft.azure.functions.ExecutionContext;
import com.microsoft.azure.functions.HttpMethod;
import com.microsoft.azure.functions.HttpRequestMessage;
import com.microsoft.azure.functions HttpResponseMessage;
import com.microsoft.azure.functions.HttpStatus;
import com.microsoft.azure.functions.annotation.AuthorizationLevel;
import com.microsoft.azure.functions.annotation.FunctionName;
import com.microsoft.azure.functions.annotation.HttpTrigger;
import com.microsoft.azure.functions.annotation.TimerTrigger;
import com.microsoft.azure.functions.signalr.*;
import com.microsoft.azure.functions.signalr.annotation.*;

import org.apache.commons.io.IOUtils;

import java.io.IOException;
import java.io.InputStream;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpResponse.BodyHandlers;
import java.nio.charset.StandardCharsets;
import java.util.Optional;

public class Function {
    private static String Etag = "";
    private static String StarCount;

    @FunctionName("index")
    public HttpResponseMessage run(
```

```

        @HttpTrigger(
            name = "req",
            methods = {HttpMethod.GET},
            authLevel =
AuthorizationLevel.ANONYMOUS)HttpRequestMessage<Optional<String>>
request,
        final ExecutionContext context) throws IOException {

    InputStream inputStream =
getClass().getClassLoader().getResourceAsStream("content/index.html");
    String text = IOUtils.toString(inputStream,
StandardCharsets.UTF_8.name());
    return
request.createResponseBuilder(HttpStatus.OK).header("Content-Type",
"text/html").body(text).build();
}

@FunctionName("negotiate")
public SignalRConnectionInfo negotiate(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS)
HttpRequestMessage<Optional<String>> req,
    @SignalRConnectionInfoInput(
        name = "connectionInfo",
        hubName = "serverless") SignalRConnectionInfo
connectionInfo) {

    return connectionInfo;
}

@FunctionName("broadcast")
@SignalROutput(name = "$return", hubName = "serverless")
public SignalRMessage broadcast(
    @TimerTrigger(name = "timeTrigger", schedule = "*/* * * * *")
String timerInfo) throws IOException, InterruptedException {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest req =
HttpRequest.newBuilder().uri(URI.create("https://api.github.com/repos/a
zure/azure-signalr")).header("User-Agent", "serverless").header("If-
None-Match", Etag).build();
    HttpResponse<String> res = client.send(req,
BodyHandlers.ofString());
    if (res.headers().firstValue("Etag").isPresent())
    {
        Etag = res.headers().firstValue("Etag").get();
    }
    if (res.statusCode() == 200)
    {
        Gson gson = new Gson();
        GitResult result = gson.fromJson(res.body(),
GitResult.class);
        StarCount = result.stargazers_count;
    }
}

```

```

        return new SignalRMessage("newMessage", "Current start count of
https://github.com/Azure/azure-signalr is:".concat(StarCount));
    }

    class GitResult {
        public String stargazers_count;
    }
}

```

3. Some dependencies need to be added. Open *pom.xml* and add the following dependencies used in the code:

XML

```

<dependency>
    <groupId>com.microsoft.azure.functions</groupId>
    <artifactId>azure-functions-java-library-signalr</artifactId>
    <version>1.0.0</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.7</version>
</dependency>

```

4. The client interface for this sample is a web page. We read HTML content from *content/index.html* in the `index` function, and then create a new file *content/index.html* in the `resources` directory. Your directory tree should look like this:

nsProject

```

| - src
| | - main
| | | - java
| | | | - com
| | | | | - signalr
| | | | | | - Function.java
| | | - resources
| | | | - content
| | | | | - index.html
| - pom.xml
| - host.json
| - local.settings.json

```

5. Open *index.html* and copy the following content:

```
HTML

<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div id="messages"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
signalr/3.1.7/signalr.min.js"></script>
    <script>
        let messages = document.querySelector('#messages');
        const apiUrl = window.location.origin;
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>

</html>
```

6. Azure Functions requires a storage account to work. You can install and run the [Azure Storage Emulator](#).

7. You're almost done now. The last step is to set a connection string of the SignalR Service to Azure Function settings.

a. Search for the Azure SignalR instance you deployed earlier using the **Search** box in Azure portal. Select the instance to open it.

RESOURCES

All 5 results

serverlesschat Azure Cosmos DB account

serverlesschat SignalR

serverlesschat Storage account

serverlesschat App Service

serverlesschat2 Storage account

RESOURCE GROUPS

All 1 results

serverlesschat Resource group

SERVICES

0 results

MARKETPLACE

0 results

DOCUMENTATION

0 results

Searching all subscriptions. [Change](#)

b. Select Keys to view the connection strings for the SignalR Service instance.

serverlesschat - Keys

SignalR

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Events

Settings

Keys

Quickstart

Host name

serverlesschat.service.signalr.net

Primary

KEY

PRIMARY-KEY

CONNECTION STRING

PRIMARY-CONNECTION-STRING

c. Copy the primary connection string, and then run the following command:

Bash

```
func settings add AzureSignalRConnectionString "<signalr-connection-string>"  
# Also we need to set AzureWebJobsStorage as Azure Function's requirement  
func settings add AzureWebJobsStorage "UseDevelopmentStorage=true"
```

8. Run the Azure Function in local:

Bash

```
mvn clean package  
mvn azure-functions:run
```

After Azure Function is running locally, go to <http://localhost:7071/api/index> and you'll see the current star count. If you star or "unstar" in the GitHub, you'll get a star count refreshing every few seconds.

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know](#)↗.

Next steps

In this quickstart, you built and ran a real-time serverless application in the local host. Next, learn more about how to bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Bi-directional communicating in Serverless](#)

[Create your first function with Java and Maven](#)

Quickstart: Create a serverless app with Azure Functions and SignalR Service using JavaScript

Article • 04/24/2024

In this article, you use Azure SignalR Service, Azure Functions, and JavaScript to build a serverless application to broadcast messages to clients.

Prerequisites

This quickstart can be run on macOS, Windows, or Linux.

[] Expand table

Prerequisite	Description
An Azure subscription	If you don't have a subscription, create an Azure free account .
A code editor	You need a code editor such as Visual Studio Code .
Azure Functions Core Tools	Requires version 4.0.5611 or higher to run Node.js v4 programming model.
Node.js LTS	See supported node.js versions in the Azure Functions JavaScript developer guide .
Azurite	SignalR binding needs Azure Storage. You can use a local storage emulator when a function is running locally.
Azure CLI	Optionally, you can use the Azure CLI to create an Azure SignalR Service instance.

Create an Azure SignalR Service instance

In this section, you create a basic Azure SignalR instance to use for your app. The following steps use the Azure portal to create a new instance, but you can also use the Azure CLI. For more information, see the [az signalr create](#) command in the [Azure SignalR Service CLI Reference](#).

1. Sign in to the [Azure portal](#).
2. In the upper-left side of the page, select + **Create a resource**.

3. On the **Create a resource** page, in the **Search services and marketplace** text box, enter **signalr** and then select **SignalR Service** from the list.
4. On the **SignalR Service** page, select **Create**.
5. On the **Basics** tab, you enter the essential information for your new SignalR Service instance. Enter the following values:

[+] [Expand table](#)

Field	Suggested Value	Description
Subscription	Choose your subscription	Select the subscription you want to use to create a new SignalR Service instance.
Resource group	Create a resource group named <i>SignalRTTestResources</i>	<p>Select or create a resource group for your SignalR resource. It's useful to create a new resource group for this tutorial instead of using an existing resource group. To free resources after completing the tutorial, delete the resource group.</p> <p>Deleting a resource group also deletes all of the resources that belong to the group. This action can't be undone. Before you delete a resource group, make certain that it doesn't contain resources you want to keep.</p> <p>For more information, see Using resource groups to manage your Azure resources.</p>
Resource name	<i>testsignalr</i>	<p>Enter a unique resource name to use for the SignalR resource. If <i>testsignalr</i> is already taken in your region, add a digit or character until the name is unique.</p> <p>The name must be a string of 1 to 63 characters and contain only numbers, letters, and the hyphen (-) character. The name can't start or end with the hyphen character, and consecutive hyphen characters aren't valid.</p>
Region	Choose your region	<p>Select the appropriate region for your new SignalR Service instance.</p> <p>Azure SignalR Service isn't currently available in all regions. For more information, see Azure SignalR Service region availability</p>
Pricing tier	Select Change and then choose Free (Dev/Test Only) . Choose	Azure SignalR Service has three pricing tiers: Free, Standard, and Premium. Tutorials use the

Field	Suggested Value	Description
	Select to confirm your choice of pricing tier.	<p>Free tier, unless noted otherwise in the prerequisites.</p> <p>For more information about the functionality differences between tiers and pricing, see Azure SignalR Service pricing</p>
Service mode	Choose the appropriate service mode	<p>Use Default when you host the SignalR hub logic in your web apps and use SignalR service as a proxy. Use Serverless when you use Serverless technologies such as Azure Functions to host the SignalR hub logic.</p> <p>Classic mode is only for backward compatibility and isn't recommended to use.</p> <p>For more information, see Service mode in Azure SignalR Service.</p>

You don't need to change the settings on the **Networking** and **Tags** tabs for the SignalR tutorials.

6. Select the **Review + create** button at the bottom of the **Basics** tab.
7. On the **Review + create** tab, review the values and then select **Create**. It takes a few moments for deployment to complete.
8. When the deployment is complete, select the **Go to resource** button.
9. On the SignalR resource page, select **Keys** from the menu on the left, under **Settings**.
10. Copy the **Connection string** for the primary key. You need this connection string to configure your app later in this tutorial.

Setup function project

Make sure you have Azure Functions Core Tools installed.

1. Open a command line.
2. Create project directory and then change into it.
3. Run the Azure Functions `func init` command to initialize a new project.

Bash

```
func init --worker-runtime javascript --language javascript --model V4
```

Create the project functions

After you initialize a project, you need to create functions. This project requires three functions:

- `index`: Hosts a web page for a client.
- `negotiate`: Allows a client to get an access token.
- `broadcast`: Uses a time trigger to periodically broadcast messages to all clients.

When you run the `func new` command from the root directory of the project, the Azure Functions Core Tools creates the function source files storing them in a folder with the function name. You edit the files as necessary replacing the default code with the app code.

Create the index function

1. Run the following command to create the `index` function.

```
Bash
```

```
func new -n index -t HttpTrigger
```

2. Edit `src/functions/httpTrigger.js` and replace the contents with the following json code:

```
JavaScript
```

```
const { app } = require('@azure/functions');
const fs = require('fs').promises;
const path = require('path');

app.http('index', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    handler: async (request, context) => {

        try {

            context.log(`Http function processed request for url
"${request.url}"`);

            const filePath =
path.join(__dirname, '../content/index.html');
            const html = await fs.readFile(filePath);

            return {
                body: html,
        }
    }
})
```

```
        headers: {
            'Content-Type': 'text/html'
        }
    };

} catch (error) {
    context.log(error);
    return {
        status: 500,
        jsonBody: error
    }
}
});

});
```

Create the negotiate function

1. Run the following command to create the `negotiate` function.

Bash

```
func new -n negotiate -t HttpTrigger
```

2. Edit `src/functions/negotiate.js` and replace the contents with the following json code:

JavaScript

```
const { app, input } = require('@azure/functions');

const inputSignalR = input.generic({
    type: 'signalRConnectionInfo',
    name: 'connectionInfo',
    hubName: 'serverless',
    connectionStringSetting: 'SIGNALR_CONNECTION_STRING',
});

app.post('negotiate', {
    authLevel: 'anonymous',
    handler: (request, context) => {
        try {
            return { body:
                JSON.stringify(context.extraInputs.get(inputSignalR)) }
        } catch (error) {
            context.log(error);
            return {
                status: 500,
                jsonBody: error
            }
        }
    }
})
```

```
        },
        route: 'negotiate',
        extraInputs: [inputSignalR],
    });
}
```

Create a broadcast function.

1. Run the following command to create the `broadcast` function.

Bash

```
func new -n broadcast -t TimerTrigger
```

2. Edit `src/functions/broadcast.js` and replace the contents with the following code:

JavaScript

```
const { app, output } = require('@azure/functions');
const getStars = require('../getStars');

var etag = '';
var star = 0;

const goingOutToSignalR = output.generic({
    type: 'signalR',
    name: 'signalR',
    hubName: 'serverless',
    connectionStringSetting: 'SIGNALR_CONNECTION_STRING',
});

app.timer('sendMessasge', {
    schedule: '0 * * * *',
    extraOutputs: [goingOutToSignalR],
    handler: async (myTimer, context) => {

        try {
            const response = await getStars(etag);

            if(response.etag === etag){
                console.log(`Same etag: ${response.etag}, no need to
broadcast message`);
                return;
            }

            etag = response.etag;
            const message = `${response.stars}`;

            context.extraOutputs.set(goingOutToSignalR,
            {
                'target': 'newMessage',
            });
        }
    }
});
```

```
        'arguments': [message]
    });
} catch (error) {
    context.log(error);
}

});
});
```

Create the index.html file

The client interface for this app is a web page. The `index` function reads HTML content from the `content/index.html` file.

1. Create a folder called `content` in your project root folder.
2. Create the file `content/index.html`.
3. Copy the following content to the `content/index.html` file and save it:

```
HTML

<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div>Instructions: Goto <a href="https://github.com/Azure/azure-signalr">GitHub repo</a> and star the repository.</div>
    <hr>
    <div>Star count: <div id="messages"></div></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/8.0.0/signalr.min.js"></script>
    <script>
        let messages = document.querySelector('#messages');
        const apiBaseUrl = window.location.origin;
        console.log(`apiBaseUrl: ${apiBaseUrl}`);
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiBaseUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            console.log(`message: ${message}`);
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>
```

```
</html>
```

Setup Azure Storage

Azure Functions requires a storage account to work. Choose either of the two following options:

- Run the free [Azure Storage Emulator](#).
- Use the Azure Storage service. This may incur costs if you continue to use it.

Local emulation

1. Start the Azurite storage emulator:

Bash

```
azurite -l azurite -d azurite\debug.log
```

2. Make sure the `AzureWebJobsStorage` in `local.settings.json` set to

```
UseDevelopmentStorage=true
```

Add the SignalR Service connection string to the function app settings

You're almost done now. The last step is to set the SignalR Service connection string in Azure Function app settings.

1. In the Azure portal, go to the SignalR instance you deployed earlier.
2. Select **Keys** to view the connection strings for the SignalR Service instance.

The screenshot shows the Azure portal interface for managing access keys. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Keys (which is currently selected), Quickstart, and Create. The main panel has a heading 'Use access keys to authenticate your SignalR clients when making requests to this Azure SignalR service. Store your access keys securely - for example, using Azure Key Vault - and don't share them. We recommend regenerating your access keys regularly. You are provided two access keys so that you can maintain connections using one key while regenerating the other.' Below this, it says 'When you regenerate your access keys, you must update your SignalR clients to use the new keys. [Learn more](#)'. There are fields for 'Host name' (set to 'serverlesschat.service.signalr.net') and 'KEY' (set to 'PRIMARY-KEY'). At the bottom, there's a 'CONNECTION STRING' field containing 'PRIMARY-CONNECTION-STRING', which is highlighted with a red border.

3. Copy the primary connection string, and execute the command:

```
Bash
```

```
func settings add AzureSignalRConnectionString "<signalr-connection-string>"
```

Run the Azure Function app locally

Run the Azure Function app in the local environment:

```
Bash
```

```
func start
```

After the Azure Function is running locally, go to <http://localhost:7071/api/index>. The page displays the current star count for the GitHub Azure/azure-signalr repository. When you star or unstar the repository in GitHub, you'll see the refreshed count every few seconds.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.

2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Sample code

You can get all code used in the article from GitHub repository:

- [aspnet/AzureSignalR-samples ↗](#).

Next steps

In this quickstart, you built and ran a real-time serverless application in localhost. Next, learn more about how to bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Bi-directional communicating in Serverless](#)

[Deploy Azure Functions with VS Code](#)

Quickstart: Create a serverless app with Azure Functions and Azure SignalR Service in Python

Article • 04/26/2024

Get started with Azure SignalR Service by using Azure Functions and Python to build a serverless application that broadcasts messages to clients. You'll run the function in the local environment, connecting to an Azure SignalR Service instance in the cloud.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure Account.

ⓘ Note

You can get the code in this article from [GitHub](#).

Prerequisites

This quickstart can be run on macOS, Windows, or Linux. You will need the following:

[+] Expand table

Prerequisite	Description
An Azure subscription	If you don't have an Azure subscription, create an Azure free account .
A code editor	You'll need a code editor such as Visual Studio Code .
Azure Functions Core Tools	Requires version 2.7.1505 or higher to run Python Azure Function apps locally.
Python 3.7+	Azure Functions requires Python 3.7+. See Supported Python versions .
Azurite	SignalR binding needs Azure Storage. You can use a local storage emulator when a function is running locally.
Azure CLI	Optionally, you can use the Azure CLI to create an Azure SignalR Service instance.

Create an Azure SignalR Service instance

In this section, you create a basic Azure SignalR instance to use for your app. The following steps use the Azure portal to create a new instance, but you can also use the Azure CLI. For more information, see the [az signalr create](#) command in the [Azure SignalR Service CLI Reference](#).

1. Sign in to the [Azure portal](#).
2. In the upper-left side of the page, select **+ Create a resource**.
3. On the **Create a resource** page, in the **Search services and marketplace** text box, enter **signalr** and then select **SignalR Service** from the list.
4. On the **SignalR Service** page, select **Create**.
5. On the **Basics** tab, you enter the essential information for your new SignalR Service instance. Enter the following values:

[] [Expand table](#)

Field	Suggested Value	Description
Subscription	Choose your subscription	Select the subscription you want to use to create a new SignalR Service instance.
Resource group	Create a resource group named <i>SignalRTTestResources</i>	<p>Select or create a resource group for your SignalR resource. It's useful to create a new resource group for this tutorial instead of using an existing resource group. To free resources after completing the tutorial, delete the resource group.</p> <p>Deleting a resource group also deletes all of the resources that belong to the group. This action can't be undone. Before you delete a resource group, make certain that it doesn't contain resources you want to keep.</p> <p>For more information, see Using resource groups to manage your Azure resources.</p>
Resource name	<i>testsignalr</i>	<p>Enter a unique resource name to use for the SignalR resource. If <i>testsignalr</i> is already taken in your region, add a digit or character until the name is unique.</p> <p>The name must be a string of 1 to 63 characters and contain only numbers, letters, and the hyphen (-) character. The name can't start or end with the hyphen character, and consecutive hyphen characters aren't valid.</p>

Field	Suggested Value	Description
Region	Choose your region	Select the appropriate region for your new SignalR Service instance. Azure SignalR Service isn't currently available in all regions. For more information, see Azure SignalR Service region availability
Pricing tier	Select Change and then choose Free (Dev/Test Only) . Choose Select to confirm your choice of pricing tier.	Azure SignalR Service has three pricing tiers: Free, Standard, and Premium. Tutorials use the Free tier, unless noted otherwise in the prerequisites. For more information about the functionality differences between tiers and pricing, see Azure SignalR Service pricing
Service mode	Choose the appropriate service mode	Use Default when you host the SignalR hub logic in your web apps and use SignalR service as a proxy. Use Serverless when you use Serverless technologies such as Azure Functions to host the SignalR hub logic. Classic mode is only for backward compatibility and isn't recommended to use. For more information, see Service mode in Azure SignalR Service .

You don't need to change the settings on the **Networking** and **Tags** tabs for the SignalR tutorials.

6. Select the **Review + create** button at the bottom of the **Basics** tab.
7. On the **Review + create** tab, review the values and then select **Create**. It takes a few moments for deployment to complete.
8. When the deployment is complete, select the **Go to resource** button.
9. On the SignalR resource page, select **Keys** from the menu on the left, under **Settings**.
10. Copy the **Connection string** for the primary key. You need this connection string to configure your app later in this tutorial.

Create the Azure Function project

Create a local Azure Function project.

1. From a command line, create a directory for your project.
2. Change to the project directory.
3. Use the Azure Functions `func init` command to initialize your function project.

```
Bash
```

```
# Initialize a function project
func init --worker-runtime python
```

Create the functions

After you initialize a project, you need to create functions. This project requires three functions:

- `index`: Hosts a web page for a client.
- `negotiate`: Allows a client to get an access token.
- `broadcast`: Uses a time trigger to periodically broadcast messages to all clients.

When you run the `func new` command from the root directory of the project, the Azure Functions Core Tools appends the function code in the `function_app.py` file. You'll edit the parameters and content as necessary by replacing the default code with the app code.

Create the index function

You can use this sample function as a template for your own functions.

Open the file `function_app.py`. This file will contain your functions. First, modify the file to include the necessary import statements, and define global variables that we will be using in the following functions.

```
Python
```

```
import azure.functions as func
import os
import requests
import json

app = func.FunctionApp()

etag = ''
start_count = 0
```

2. Add the function `index` by adding the following code

Python

```
@app.route(route="index", auth_level=func.AuthLevel.ANONYMOUS)
def index(req: func.HttpRequest) -> func.HttpResponse:
    f = open(os.path.dirname(os.path.realpath(__file__)) +
'/content/index.html')
    return func.HttpResponse(f.read(), mimetype='text/html')
```

This function hosts a web page for a client.

Create the negotiate function

Add the function `negotiate` by adding the following code

Python

```
@app.route(route="negotiate", auth_level=func.AuthLevel.ANONYMOUS, methods=
["POST"])
@app.generic_input_binding(arg_name="connectionInfo",
type="signalRConnectionInfo", hubName="serverless",
connectionStringSetting="AzureSignalRConnectionString")
def negotiate(req: func.HttpRequest, connectionInfo) -> func.HttpResponse:
    return func.HttpResponse(connectionInfo)
```

This function allows a client to get an access token.

Create a broadcast function.

Add the function `broadcast` by adding the following code

Python

```
@app.timer_trigger(schedule="*/1 * * * *", arg_name="myTimer",
run_on_startup=False,
use_monitor=False)
@app.generic_output_binding(arg_name="signalRMessages", type="signalR",
hubName="serverless",
connectionStringSetting="AzureSignalRConnectionString")
def broadcast(myTimer: func.TimerRequest, signalRMessages: func.Out[str]) ->
None:
    global etag
    global start_count
    headers = {'User-Agent': 'serverless', 'If-None-Match': etag}
    res = requests.get('https://api.github.com/repos/azure/azure-functions-
python-worker', headers=headers)
    if res.headers.get('ETag'):
```

```

        etag = res.headers.get('ETag')

    if res.status_code == 200:
        jres = res.json()
        start_count = jres['stargazers_count']

        signalRMessages.set(json.dumps({
            'target': 'newMessage',
            'arguments': [ 'Current star count of
https://api.github.com/repos/azure/azure-functions-python-worker is: ' +
str(start_count) ]
        }))

```

This function uses a time trigger to periodically broadcast messages to all clients.

Create the index.html file

The client interface for this app is a web page. The `index` function reads HTML content from the `content/index.html` file.

1. Create a folder called `content` in your project root folder.
2. Create the file `content/index.html`.
3. Copy the following content to the `content/index.html` file and save it:

HTML

```

<html>

    <body>
        <h1>Azure SignalR Serverless Sample</h1>
        <div id="messages"></div>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
signalr/3.1.7/signalr.min.js"></script>
        <script>
            let messages = document.querySelector('#messages');
            const apiBaseUrl = window.location.origin;
            const connection = new signalR.HubConnectionBuilder()
                .withUrl(apiBaseUrl + '/api')
                .configureLogging(signalR.LogLevel.Information)
                .build();
            connection.on('newMessage', (message) => {
                document.getElementById("messages").innerHTML = message;
            });

            connection.start()
                .catch(console.error);
        </script>
    </body>

```

```
</html>
```

Add the SignalR Service connection string to the function app settings

The last step is to set the SignalR Service connection string in Azure Function app settings.

1. In the Azure portal, go to the SignalR instance you deployed earlier.
2. Select **Keys** to view the connection strings for the SignalR Service instance.

The screenshot shows the 'serverlesschat - Keys' page in the Azure portal. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Keys (which is selected and highlighted in blue), Quickstart, and Help. The main area has a search bar at the top. Below it, there's a note about storing access keys securely. It shows the host name as 'serverlesschat.service.signalr.net'. Under the 'Primary' key, the KEY value is 'PRIMARY-KEY' and the CONNECTION STRING value is 'PRIMARY-CONNECTION-STRING'. The 'PRIMARY-CONNECTION-STRING' field is highlighted with a red rectangle.

3. Copy the primary connection string, and execute the command:

```
Bash  
  
func settings add AzureSignalRConnectionString "<signalr-connection-string>"
```

Run the Azure Function app locally

Start the Azurite storage emulator:

```
Bash
```

```
azurite
```

Run the Azure Function app in the local environment:

```
Bash
```

```
func start
```

ⓘ Note

If you see an errors showing read errors on the blob storage, ensure the 'AzureWebJobsStorage' setting in the *local.settings.json* file is set to `UseDevelopmentStorage=true`.

After the Azure Function is running locally, go to <http://localhost:7071/api/index>. The page displays the current star count for the GitHub Azure/azure-signalr repository. When you star or unstar the repository in GitHub, you'll see the refreshed count every few seconds.

Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know ↗](#).

Next steps

In this quickstart, you built and ran a real-time serverless application in local. Next, learn more about how to use bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Bi-directional communicating in Serverless](#)

Deploy Azure Functions with VS Code

Tutorial: Create a serverless notification app with Azure Functions and Azure Web PubSub service

Article • 01/12/2024

The Azure Web PubSub service helps you build real-time messaging web applications using WebSockets. Azure Functions is a serverless platform that lets you run your code without managing any infrastructure. In this tutorial, you learn how to use Azure Web PubSub service and Azure Functions to build a serverless application with real-time messaging under notification scenarios.

In this tutorial, you learn how to:

- ✓ Build a serverless notification app
- ✓ Work with Web PubSub function input and output bindings
- ✓ Run the sample functions locally
- ✓ Deploy the function to Azure Function App

Prerequisites

JavaScript Model v4

- A code editor, such as [Visual Studio Code](#)
- [Node.js](#), version 18.x or above.

Note

For more information about the supported versions of Node.js, see [Azure Functions runtime versions documentation](#).

- [Azure Functions Core Tools](#) (V4 or higher preferred) to run Azure Function apps locally and deploy to Azure.
- The [Azure CLI](#) to manage Azure resources.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Sign in to Azure

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Create an Azure Web PubSub service instance

Your application will connect to a Web PubSub service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *Web PubSub* in the search box and press enter. (You could also search the Azure Web PubSub from the `Web` category.)

2. Select **Web PubSub** from the search results, then select **Create**.

3. Enter the following settings.

 Expand table

Setting	Suggested value	Description
Resource name	Globally unique name	The globally unique Name that identifies your new Web PubSub service instance. Valid characters are a-z, A-Z, 0-9, and -.
Subscription	Your subscription	The Azure subscription under which this new Web PubSub service instance is created.

Setting	Suggested value	Description
Resource Group	myResourceGroup	Name for the new resource group in which to create your Web PubSub service instance.
Location	West US	Choose a region near you.
Pricing tier	Free	You can first try Azure Web PubSub service for free. Learn more details about Azure Web PubSub service pricing tiers
Unit count	-	Unit count specifies how many connections your Web PubSub service instance can accept. Each unit supports 1,000 concurrent connections at most. It is only configurable in the Standard tier.

Home > [Create a resource](#) >

+ Web PubSub Service

Web PubSub Service

[...](#)

[*](#) Basics Tags Review + create

Deploy fully managed WebPubSub Service at scale. [Learn more](#)

Project Details

Subscription [*](#) [\(i\)](#) Visual Studio Enterprise Subscription

Resource group [*](#) [\(i\)](#) [Create new](#)

Service Details

Resource Name Enter the name .webpubsub.azure.com

Region [*](#) [\(i\)](#) East US

Pricing tier ([View full pricing details](#)) [*](#) Standard

Unit count [*](#) [\(i\)](#) 1

4. Select **Create** to start deploying the Web PubSub service instance.

Create and run the functions locally

1. Make sure you have [Azure Functions Core Tools](#) installed. Now, create an empty directory for the project. Run command under this working directory. Use one of the given options below.

JavaScript Model v4

Bash

```
func init --worker-runtime javascript --model v4
```

2. Follow the steps to install `Microsoft.Azure.WebJobs.Extensions.WebPubSub`.

JavaScript Model v4

Confirm or update `host.json`'s extensionBundle to version 4.* or later to get Web PubSub support. For updating the `host.json`, open the file in editor, and then replace the existing version extensionBundle to version 4.* or later.

JSON

```
{
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.0.0)"
  }
}
```

3. Create an `index` function to read and host a static web page for clients.

Bash

```
func new -n index -t HttpTrigger
```

JavaScript Model v4

- Update `src/functions/index.js` and copy following codes.

JavaScript

```
const { app } = require('@azure/functions');
const { readFile } = require('fs/promises');

app.http('index', {
  methods: ['GET', 'POST'],
  authLevel: 'anonymous',
  handler: async (context) => {
    const content = await readFile('index.html', 'utf8',
      (err, data) => {
        if (err) {
          context.error(err)
          return
        }
        context.res = {
          status: 200,
          headers: {
            'Content-Type': 'text/html'
          },
          body: data
        }
      }
    )
  }
})
```

```

        }
    });

    return {
        status: 200,
        headers: {
            'Content-Type': 'text/html'
        },
        body: content,
    };
}
);

```

4. Create a `negotiate` function to help clients get service connection url with access token.

Bash

```
func new -n negotiate -t HttpTrigger
```

JavaScript Model v4

- Update `src/functions/negotiate.js` and copy following codes.

JavaScript

```

const { app, input } = require('@azure/functions');

const connection = input.generic({
    type: 'webPubSubConnection',
    name: 'connection',
    hub: 'notification'
});

app.http('negotiate', {
    methods: ['GET', 'POST'],
    authLevel: 'anonymous',
    extraInputs: [connection],
    handler: async (request, context) => {
        return { body:
            JSON.stringify(context.extraInputs.get('connection')) };
    },
});

```

5. Create a `notification` function to generate notifications with `TimerTrigger`.

Bash

```
func new -n notification -t TimerTrigger
```

JavaScript Model v4

- Update `src/functions/notification.js` and copy following codes.

JavaScript

```
const { app, output } = require('@azure/functions');

const wpsAction = output.generic({
    type: 'webPubSub',
    name: 'action',
    hub: 'notification'
});

app.timer('notification', {
    schedule: '*/* * * * *',
    extraOutputs: [wpsAction],
    handler: (myTimer, context) => {
        context.extraOutputs.set(wpsAction, {
            actionName: 'sendToAll',
            data: `[DateTime: ${new Date()}] Temperature: ${getValue(22, 1)}\xB0C, Humidity: ${getValue(40, 2)}%`,
            dataType: 'text',
        });
    },
});

function getValue(baseNum, floatNum) {
    return (baseNum + 2 * floatNum * (Math.random() - 0.5)).toFixed(3);
}
```

6. Add the client single page `index.html` in the project root folder and copy content.

HTML

```
<html>
  <body>
    <h1>Azure Web PubSub Notification</h1>
    <div id="messages"></div>
    <script>
      (async function () {
        let messages = document.querySelector('#messages');
        let res = await
```

```

fetch(`${window.location.origin}/api/negotiate`);
    let url = await res.json();
    let ws = new WebSocket(url.url);
    ws.onopen = () => console.log('connected');

    ws.onmessage = event => {
        let m = document.createElement('p');
        m.innerText = event.data;
        messages.appendChild(m);
    };
})();
</script>
</body>
</html>

```

JavaScript Model v4

7. Configure and run the Azure Function app

- In the browser, open the [Azure portal](#) and confirm the Web PubSub Service instance you deployed earlier was successfully created. Navigate to the instance.
- Select **Keys** and copy out the connection string.

The screenshot shows the 'Settings' blade for a Web PubSub Service instance in the Azure portal. On the left, there's a sidebar with 'Keys' selected, indicated by a red box. The main area shows two sections: 'Primary' and 'Secondary'. Each section has a 'Key' input field and a 'Connection string' input field, both of which are also highlighted with red boxes. Below each section is a 'Regenerate Primary Key' or 'Regenerate Secondary Key' button.

Run command in the function folder to set the service connection string. Replace <connection-string> with your value as needed.

Bash

```
func settings add WebPubSubConnectionString "<connection-string>"
```

① Note

TimerTrigger used in the sample has dependency on Azure Storage, but you can use local storage emulator when the Function is running locally. If you got some error like There was an error performing a read operation on the Blob Storage Secret Repository. Please ensure the 'AzureWebJobsStorage' connection string is valid., you'll need to download and enable [Storage Emulator](#).

Now you're able to run your local function by command.

Bash

```
func start --port 7071
```

And checking the running logs, you can visit your local host static page by visiting:
<http://localhost:7071/api/index>.

① Note

Some browers automatically redirect to https that leads to wrong url.
Suggest to use Edge and double check the url if rendering is not success.

Deploy Function App to Azure

Before you can deploy your function code to Azure, you need to create three resources:

- A resource group, which is a logical container for related resources.
- A storage account, which is used to maintain state and other information about your functions.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment and sharing of resources.

Use the following commands to create these items.

1. Sign in to Azure:

Azure CLI

```
az login
```

2. Create a resource group or you can skip by reusing the one of Azure Web PubSub service:

Azure CLI

```
az group create -n WebPubSubFunction -l <REGION>
```

3. Create a general-purpose storage account in your resource group and region:

Azure CLI

```
az storage account create -n <STORAGE_NAME> -l <REGION> -g  
WebPubSubFunction
```

4. Create the function app in Azure:

JavaScript Model v4

Azure CLI

```
az functionapp create --resource-group WebPubSubFunction --  
consumption-plan-location <REGION> --runtime node --runtime-version  
18 --functions-version 4 --name <FUNCTIONAPP_NAME> --storage-account  
<STORAGE_NAME>
```

 **Note**

Check [Azure Functions runtime versions documentation](#) to set --
runtime-version parameter to supported value.

5. Deploy the function project to Azure:

Once you create your function app in Azure, you're now ready to deploy your local functions project by using the `func azure functionapp publish` command.

Bash

```
func azure functionapp publish <FUNCTIONAPP_NAME> --publish-local-  
settings
```

(!) Note

Here we are deploying local settings `local.settings.json` together with command parameter `--publish-local-settings`. If you're using Microsoft Azure Storage Emulator, you can type `no` to skip overwriting this value on Azure following the prompt message: `App setting AzureWebJobsStorage is different between azure and local.settings.json, Would you like to overwrite value in azure? [yes/no/show]`. Besides, you can update Function App settings in **Azure Portal -> Settings -> Configuration**.

6. Now you can check your site from Azure Function App by navigating to URL:

`https://<FUNCTIONAPP_NAME>.azurewebsites.net/api/index`.

Clean up resources

If you're not going to continue to use this app, delete all resources created by this doc with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Use the search box to find the resource group by its name instead.
2. In the window that opens, select the resource group, and then select **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to run a serverless chat application. Now, you could start to build your own application.

[Tutorial: Create a simple chatroom with Azure Web PubSub](#)

[Azure Web PubSub bindings for Azure Functions](#)

[Explore more Azure Web PubSub samples](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

How to work with Event Grid triggers and bindings in Azure Functions

Article • 08/21/2024

Azure Functions provides built-in integration with Azure Event Grid by using [triggers and bindings](#). This article shows you how to configure and locally evaluate your Event Grid trigger and bindings. For more information about Event Grid trigger and output binding definitions and examples, see one of the following reference articles:

- [Azure Event Grid bindings Overview](#)
- [Azure Event Grid trigger for Azure Functions](#)
- [Azure Event Grid output binding for Azure Functions](#)

Create an event subscription

To start receiving Event Grid HTTP requests, you need a subscription to events raised by Event Grid. Event subscriptions specify the endpoint URL that invokes the function.

When you create an event subscription from your function's **Integration** tab in the [Azure portal](#), the URL is supplied for you. When you programmatically create an event subscription or when you create the event subscription from Event Grid, you'll need to provide the endpoint. The endpoint URL contains a system key, which you must obtain from Functions administrator REST APIs.

Get the webhook endpoint URL

The URL endpoint for your Event Grid triggered function depends on the version of the Functions runtime. The following example shows the version-specific URL pattern:

v2.x+	
	HTTP
	<code>https://{{functionappname}}.azurewebsites.net/runtime/webhooks/eventgrid? functionName={{functionname}}&code={{systemkey}}</code>

ⓘ Note

There is a version of the Blob storage trigger that also uses event subscriptions. The endpoint URL for this kind of Blob storage trigger has a path of `/runtime/webhooks/blobs`, whereas the path for an Event Grid trigger would be `/runtime/webhooks/EventGrid`. For a comparison of options for processing blobs, see [Trigger on a blob container](#).

Obtain the system key

The URL endpoint you construct includes a system key value. The system key is an authorization key, specific to the Event Grid webhook, that must be included in a request to the endpoint URL for an Event Grid trigger. The following section explains how to get the system key.

You can also get the master key for your function app from **Functions > App keys** in the portal.

⊗ Caution

The master key provides administrator access to your function app. Don't share this key with third parties or distribute it in native client applications.

For more information, see [Work with access keys in Azure Functions](#).

You can get the system key from your function app by using the following administrator APIs (HTTP GET):

v2.x+

```
http://{functionappname}.azurewebsites.net/admin/host/systemkeys/eventgr  
id_extension?code={masterkey}
```

This REST API is an administrator API, so it requires your function app [master key](#). Don't confuse the system key (for invoking an Event Grid trigger function) with the master key (for performing administrative tasks on the function app). When you subscribe to an Event Grid topic, be sure to use the system key.

Here's an example of the response that provides the system key:

```
{  
  "name": "eventgridextensionconfig_extension",  
  "value": "{the system key for the function}",  
  "links": [  
    {  
      "rel": "self",  
      "href": "{the URL for the function, without the system key}"  
    }  
  ]  
}
```

Create the subscription

You can create an event subscription either from the [Azure portal](#) or by using the Azure CLI.

Portal

For functions that you develop in the Azure portal with the Event Grid trigger, select **Integration** then choose the **Event Grid Trigger** and select **Create Event Grid subscription**.

The screenshot shows the Azure portal interface for a function named 'EventGridTrigger1'. On the left, there's a sidebar with 'Overview', 'Developer' (selected), 'Code + Test', 'Integration' (highlighted with a red box), 'Monitor', and 'Function Keys'. In the main area, under 'Integration', it says 'Edit the trigger and choose from a selection of inputs an Storage, Cosmos DB and others.' Below this is a 'Trigger' section with 'Event Grid Trigger (eventGridEvent)' highlighted with a red box. To the right, an 'Edit Trigger' modal is open. It has 'Save', 'Discard', and 'Delete' buttons at the top. Under 'Binding Type', 'Event Grid Trigger' is selected. In the 'Event Trigger parameter name*' field, 'eventGridEvent' is entered. At the bottom of the modal, a button labeled 'Create Event Grid subscription' is highlighted with a red box.

When you select this link, the portal opens the **Create Event Subscription** page with the current trigger endpoint already defined.

 **Create Event Subscription**
Event Grid

Basic [Filters](#) [Additional Features](#)

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

EVENT SUBSCRIPTION DETAILS

Name

Event Schema

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Types

Subscription

Resource Group

Resource

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Filter to Event Types

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type

Endpoint

Create

For more information about how to create subscriptions by using the Azure portal, see [Create custom event - Azure portal](#) in the Event Grid documentation.

For more information about how to create a subscription, see [the blob storage quickstart](#) or the other Event Grid quickstarts.

Local testing with viewer web app

To test an Event Grid trigger locally, you have to get Event Grid HTTP requests delivered from their origin in the cloud to your local machine. One way to do that is by capturing requests online and manually resending them on your local machine:

1. [Create a viewer web app](#) that captures event messages.
2. [Create an Event Grid subscription](#) that sends events to the viewer app.

3. [Generate a request](#) and copy the request body from the viewer app.
4. [Manually post the request](#) to the localhost URL of your Event Grid trigger function.

To send an HTTP post request, you need an HTTP test tool. Make sure to choose a tool that keeps your data secure. For more information, see [HTTP test tools](#).

When you're done testing, you can use the same subscription for production by updating the endpoint. Use the `az eventgrid event-subscription update` Azure CLI command.

Create a viewer web app

To simplify capturing event messages, you can deploy a [pre-built web app](#) that displays the event messages. The deployed solution includes an App Service plan, an App Service web app, and source code from GitHub.

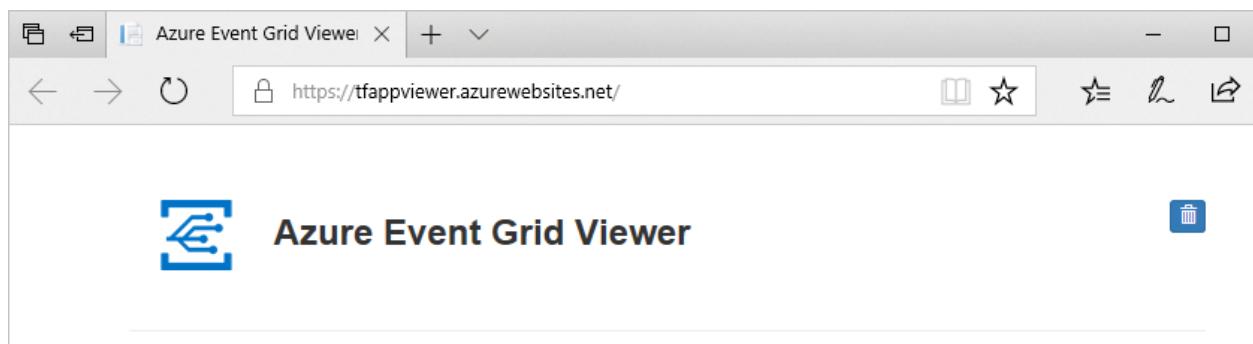
Select **Deploy to Azure** to deploy the solution to your subscription. In the Azure portal, provide values for the parameters.



The deployment may take a few minutes to complete. After the deployment has succeeded, view your web app to make sure it's running. In a web browser, navigate to:

`https://<your-site-name>.azurewebsites.net`

You see the site but no events have been posted to it yet.



Create an Event Grid subscription

Create an Event Grid subscription of the type you want to test, and give it the URL from your web app as the endpoint for event notification. The endpoint for your web app must include the suffix `/api/updates/`. So, the full URL is `https://<your-site-name>.azurewebsites.net/api/updates`

For information about how to create subscriptions by using the Azure portal, see [Create custom event - Azure portal](#) in the Event Grid documentation.

Generate a request

Trigger an event that will generate HTTP traffic to your web app endpoint. For example, if you created a blob storage subscription, upload or delete a blob. When a request shows up in your web app, copy the request body.

The subscription validation request will be received first; ignore any validation requests, and copy the event request.

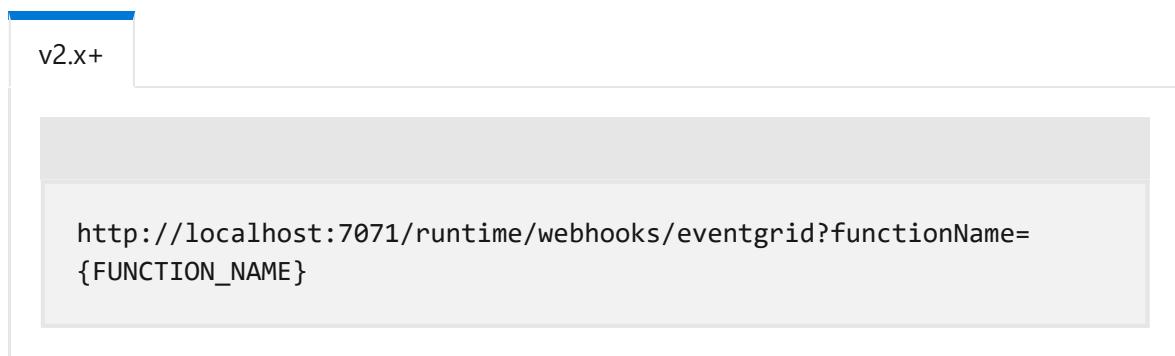
Manually post the request

Run your Event Grid function locally. The `Content-Type` and `aeg-event-type` headers are required to be manually set, while all other values can be left as default.

Use your HTTP test tool to create an HTTP POST request:

- Set a Content-Type: application/json header.
 - Set an aeg-event-type: Notification header.

- Paste the RequestBin data into the request body.
- Send an HTTP POST request to the endpoint that manually starts the Event Grid trigger.



The `functionName` parameter must be the name specified in the `FunctionName` attribute.

The Event Grid trigger function executes and shows logs similar to the following example:

```
C:\WINDOWS\system32\cmd.exe
[1/26/2018 8:16:46 PM] Function started (Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa)
[1/26/2018 8:16:46 PM] Executing 'EventGridTest' (Reason='EventGrid trigger fired at 2018-01-26T12:16:46.0126069-08:00', Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa)
[1/26/2018 8:16:46 PM] C# Event Grid function processed a request.
[1/26/2018 8:16:46 PM] Subject: /blobServices/default/containers/test0123/blobs/Default.rdp
[1/26/2018 8:16:46 PM] Time: 1/23/2018 5:02:19 PM
[1/26/2018 8:16:46 PM] Data: {
[1/26/2018 8:16:46 PM]     "api": "PutBlockList",
[1/26/2018 8:16:46 PM]     "clientRequestId": "2c169f2f-7b3b-4d99-839b-c92a2d25801b",
[1/26/2018 8:16:46 PM]     "requestId": "44d4f022-001e-003c-466b-940cba000000",
[1/26/2018 8:16:46 PM]     "eTag": "0x8D562831044DD0",
[1/26/2018 8:16:46 PM]     "contentType": "application/octet-stream",
[1/26/2018 8:16:46 PM]     "contentLength": 2248,
[1/26/2018 8:16:46 PM]     "blobType": "BlockBlob",
[1/26/2018 8:16:46 PM]     "url": "https://egblobstor0122.blob.core.windows.net/test0123/Default.rdp",
[1/26/2018 8:16:46 PM]     "sequencer": "000000000000272D000000000003D60F",
[1/26/2018 8:16:46 PM]     "storageDiagnostics": {
[1/26/2018 8:16:46 PM]         "batchId": "b4229b3a-4d50-4ff4-a9f2-039ccf26efe9"
[1/26/2018 8:16:46 PM]     }
[1/26/2018 8:16:46 PM] }
[1/26/2018 8:16:46 PM] Function completed (Success, Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa, Duration: 00:00:00.000)
```

Next steps

To learn more about Event Grid with Functions, see the following articles:

- [Azure Event Grid bindings for Azure Functions](#)
- [Tutorial: Automate resizing uploaded images using Event Grid](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Start/Stop VMs v2 overview

Article • 09/05/2024

The Start/Stop VMs v2 feature starts or stops Azure Virtual Machines instances across multiple subscriptions. It starts or stops virtual machines on user-defined schedules, provides insights through [Azure Application Insights](#), and send optional notifications by using [action groups](#). For most scenarios, Start/Stop VMs can manage virtual machines deployed and managed both by Azure Resource Manager and by Azure Service Manager (classic), which is [deprecated](#).

This new version of Start/Stop VMs v2 provides a decentralized low-cost automation option for customers who want to optimize their VM costs. It offers all of the same functionality as the original version that was available with Azure Automation, but it's designed to take advantage of newer technology in Azure. The Start/Stop VMs v2 relies on multiple Azure services and it will be charged based on the service that are deployed and consumed.

Important Start/Stop VMs v2 Updates

- No further development, enhancements, or updates will be available for Start/Stop v2 except when required to remain on supported versions of components and Azure services.
- The `TriggerAutoUpdate` and `UpdateStartStopV2` functions are now deprecated and will be removed in the future. To update Start/Stop v2, we recommend that you stop the site, install to the latest version from our [GitHub repository](#), and then start the site. To disable the automatic update functionality, set the Function App's [AzureClientOptions:EnableAutoUpdate application setting](#) to `false`. No built-in notification system is available for updates. After an update to Start/Stop v2 becomes available, we will update the [readme.md](#) in the GitHub repository. Third-party GitHub file watchers might be available to notify you of changes.
- As of August 19, 2024, Start/Stop v2 has been updated to the [.NET 8 isolated worker model](#).

Overview

Start/Stop VMs v2 is redesigned and it doesn't depend on Azure Automation or Azure Monitor Logs, as required by the previous version. This version relies on [Azure Functions](#)

to handle the VM start and stop execution.

A managed identity is created in Microsoft Entra ID for this Azure Functions application and allows Start/Stop VMs v2 to easily access other Microsoft Entra protected resources, such as the logic apps and Azure VMs. For more about managed identities in Microsoft Entra ID, see [Managed identities for Azure resources](#).

An HTTP trigger function endpoint is created to support the schedule and sequence scenarios included with the feature, as shown in the following table.

[+] [Expand table](#)

Name	Trigger	Description
Scheduled	HTTP	This function is for both scheduled and sequenced scenario (differentiated by the payload schema). It's the entry point function called from the Logic App and takes the payload to process the VM start or stop operation.
AutoStop	HTTP	This function supports the AutoStop scenario, which is the entry point function that is called from Logic App.
AutoStopVM	HTTP	This function is triggered automatically by the VM alert when the alert condition is true.
VirtualMachineRequestOrchestrator	Queue	This function gets the payload information from the Scheduled function and orchestrates the VM start and stop requests.
VirtualMachineRequestExecutor	Queue	This function performs the actual start and stop operation on the VM.
CreateAutoStopAlertExecutor	Queue	This function gets the payload information from the AutoStop function to create the alert on the VM.
HeartBeatAvailabilityTest	Timer	This function monitors the availability of the primary HTTP functions.
CostAnalyticsFunction	Timer	This function is used by Microsoft to estimate aggregate cost of Start/Stop V2 across customers. This function does not impact the functionality of Start/Stop V2.
SavingsAnalyticsFunction	Timer	This function is used by Microsoft to estimate aggregate savings of Start/Stop V2 across

Name	Trigger	Description
		customers. This function does not impact the functionality of Start/Stop V2.
VirtualMachineSavingsFunction	Queue	This function performs the actual savings calculation on a VM achieved by the Start/Stop V2 solution.
TriggerAutoUpdate	Timer	Deprecated. This function starts the auto update process based on the application setting "AzureClientOptions:EnableAutoUpdate=true".
UpdateStartStopV2	Queue	Deprecated. This function performs the actual auto update execution, which validates your current version with the available version and decides the final action.

For example, **Scheduled** HTTP trigger function is used to handle schedule and sequence scenarios. Similarly, **AutoStop** HTTP trigger function handles the auto stop scenario.

The queue-based trigger functions are required in support of this feature. All timer-based triggers are used to perform the availability test and to monitor the health of the system.

[Azure Logic Apps](#) is used to configure and manage the start and stop schedules for the VM take action by calling the function using a JSON payload. By default, during initial deployment it creates a total of five Logic Apps for the following scenarios:

- **Scheduled** - Start and stop actions are based on a schedule you specify against Azure Resource Manager and classic VMs. `ststv2_vms_Scheduled_start` and `ststv2_vms_Scheduled_stop` configure the scheduled start and stop.
- **Sequenced** - Start and stop actions are based on a schedule targeting VMs with pre-defined sequencing tags. Only two named tags are supported - `sequencestart` and `sequencestop`. `ststv2_vms_Sequenced_start` and `ststv2_vms_Sequenced_stop` configure the sequenced start and stop.

The proper way to use the sequence functionality is to create a tag named `sequencestart` on each VM you wish to be started in a sequence. The tag value needs to be an integer ranging from 1 to N for each VM in the respective scope. The tag is optional and if not present, the VM simply won't participate in the sequencing. The same criteria applies to stopping VMs with only the tag name being different and use `sequencestop` in this case. You have to configure both the tags in each VM to get start and stop action. If two or more VMs share the same tag value, those VMs would be started or stopped at the same time.

For example, the following table shows that both start and stop actions are processed in ascending order by the value of the tag.

VM Name	Tags	Action Order
VM1	sequencestart: 1 sequencestop: 2	Start : VM1, VM2
VM2	sequencestart: 2 sequencestop: 1	Stop : VM2, VM1

 **Note**

This scenario only supports Azure Resource Manager VMs.

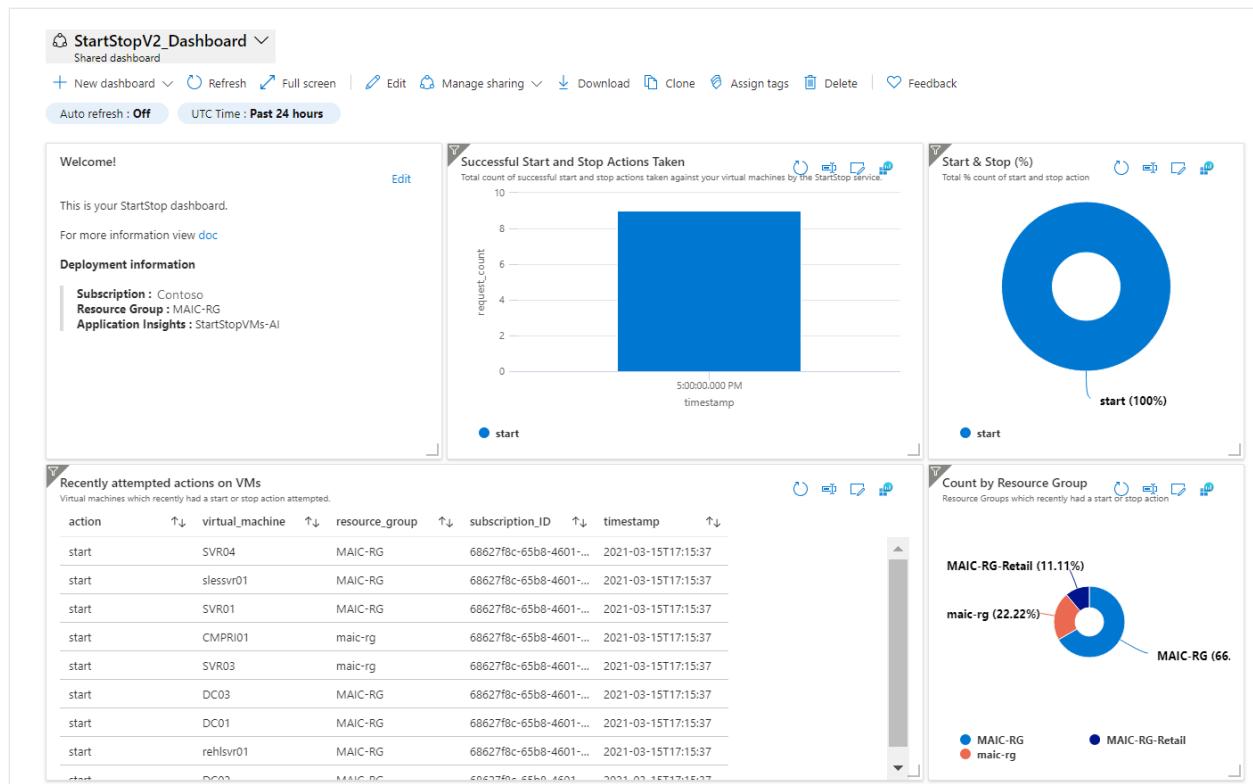
- **AutoStop** - This functionality is only used for performing a stop action against both Azure Resource Manager and classic VMs based on its CPU utilization. It can also be a scheduled-based *take action*, which creates alerts on VMs and based on the condition, the alert is triggered to perform the stop action.
`ststv2_vms_AutoStop` configures the auto stop functionality.

Each Start/Stop action supports assignment of one or more subscriptions, resource groups, or a list of VMs.

An Azure Storage account, which is required by Functions, is also used by Start/Stop VMs v2 for two purposes:

- Uses Azure Table Storage to store the execution operation metadata (that is, the start/stop VM action).
- Uses Azure Queue Storage to support the Azure Functions queue-based triggers.

All trace logging data from the function app execution is sent to your connected Application Insights instance. You can view the telemetry data stored in Application Insights from a set of pre-defined visualizations presented in a shared [Azure dashboard](#).



Email notifications are also sent as a result of the actions performed on the VMs.

New releases

When a new version of Start/Stop VMs v2 is released, your instance is auto-updated without having to manually redeploy.

Supported scoping options

Subscription

Scoping to a subscription can be used when you need to perform the start and stop action on all the VMs in an entire subscription, and you can select multiple subscriptions if necessary.

You can also specify a list of VMs to exclude and it will ignore them from the action. You can also use wildcard characters to specify all the names that simultaneously can be ignored.

Resource group

Scoping to a resource group can be used when you need to perform the start and stop action on all the VMs by specifying one or more resource group names, and across one or more subscriptions.

You can also specify a list of VMs to exclude and it will ignore them from the action. You can also use wildcard characters to specify all the names that simultaneously can be ignored.

VMList

Specifying a list of VMs can be used when you need to perform the start and stop action on a specific set of virtual machines, and across multiple subscriptions. This option doesn't support specifying a list of VMs to exclude.

Prerequisites

- You must have an Azure account with an active subscription. [Create an account for free ↗](#).
- To deploy the solution, your account must be granted the [Owner](#) permission in the subscription.
- Start/Stop VMs v2 is available in all Azure global and US Government cloud regions that are listed in [Products available by region ↗](#) page for Azure Functions.

Next steps

To deploy this feature, see [Deploy Start/Stop VMs](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Deploy Start/Stop VMs v2 to an Azure subscription

Article • 09/05/2024

Perform the steps in this article in sequence to install the Start/Stop VMs v2 feature. After completing the setup process, configure the schedules to customize it to your requirements.

Permissions and Policy considerations

Keep the following considerations in mind before and during deployment:

- The solution allows users with appropriate role-based access control (RBAC) permissions on the Start/Stop v2 deployment to add, remove, and manage schedules for virtual machines under the scope of the Start/Stop VMs v2 instance. This behavior is by design. In practice, this means a user who doesn't have explicit permissions on a virtual machine could still create start, stop, and autostop operations on that virtual machine when they have the permission to modify the Start/Stop v2 solution managing the virtual machine.
- Any users with access to the Start/Stop v2 solution could uncover cost, savings, operation history, and other data that is stored in the Application Insights instance used by the Start/Stop v2 application.
- When managing a Start/Stop v2 solution, you should consider the permissions of users to the Start/Stop v2 solution, particularly when they don't have permission to directly modify the target virtual machines.
- When you deploy the Start/Stop v2 solution to a new or existing resource group, a tag named **SolutionName** with a value of **StartStopV2** is added to resource group and to its resources that are deployed by Start/Stop v2. Any other tags on these resources are removed. If you have an Azure policy that denies management operations based on resource tags, you must allow management operations for resources that contain only this tag.

Deploy feature

The deployment is initiated from the [Start/Stop VMs v2 GitHub organization](#). While this feature is intended to manage all of your VMs in your subscription across all resource groups from a single deployment within the subscription, you can install

another instance of it based on the operations model or requirements of your organization. It also can be configured to centrally manage VMs across multiple subscriptions.

To simplify management and removal, we recommend you deploy Start/Stop VMs v2 to a dedicated resource group.

 **Note**

Currently this solution does not support specifying an existing Storage account or Application Insights resource.

 **Note**

The naming format for the function app and storage account has changed. To guarantee global uniqueness, a random and unique string is now appended to the names of these resource.

1. Open your browser and navigate to the Start/Stop VMs v2 [GitHub organization](#).
2. Select the deployment option based on the Azure cloud environment your Azure VMs are created in.
3. If prompted, sign in to the [Azure portal](#).
4. Choose the appropriate **Plan** from the drop-down box. When choosing a Zone Redundant plan (**Start/StopV2-AZ**), you must create your deployment in one of the following regions:
 - Australia East
 - Brazil South
 - Canada Central
 - Central US
 - East US
 - East US 2
 - France Central
 - Germany West Central
 - Japan East
 - North Europe
 - Southeast Asia
 - UK South
 - West Europe

- West US 2
- West US 3

5. Select **Create**, which opens the custom Azure Resource Manager deployment page in the Azure portal.

6. Enter the following values:

[Expand table](#)

Name	Value
Region	Select a region near you for new resources.
Resource Group Name	Specify the resource group name that will contain the individual resources for Start/Stop VMs.
Resource Group Region	Specify the region for the resource group. For example, Central US .
Azure Function App Name	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Application Insights Name	Specify the name of your Application Insights instance that will hold the analytics for Start/Stop VMs.
Application Insights Region	Specify the region for the Application Insights instance.
Storage Account Name	Specify the name of the Azure Storage account to store Start/Stop VMs execution telemetry.
Email Address	Specify one or more email addresses to receive status notifications, separated by a comma (,).

The screenshot shows the 'Custom deployment' page in the Azure portal. The 'Subscription' dropdown is set to 'Contoso'. The 'Region' parameter is set to 'West US'. The 'Resource Group Name' is 'ProdWUS'. The 'Resource Group Region' is 'westus'. The 'Azure Function App Name' is 'StartStopVMs'. The 'Application Insights Name' is 'ProdWUS-AI-Pri'. The 'Application Insights Region' is 'westus'. The 'Storage Account Name' is 'prodwusstartstopvm'. The 'Email Addresses' field contains 'jsmith@contoso.com'. At the bottom, there are buttons for 'Review + create' and 'Next : Review + create >'. The browser address bar shows the URL https://portal.azure.com/#create/Microsoft.Template.

7. Select **Review + create** on the bottom of the page.
8. Select **Create** to start the deployment.
9. Select the bell icon (notifications) from the top of the screen to see the deployment status. You shall see **Deployment in progress**. Wait until the deployment is completed.
10. Select **Go to resource group** from the notification pane. You shall see a screen similar to:

<input type="checkbox"/> Name ↑↓	Type ↑↓	Resource group ↑↓	Location ↑↓
<input type="checkbox"/>  StartStopV2_Dashboard (StartStopV2_Dashboard)	Shared dashboard	MAIC-RG	East US
<input type="checkbox"/>  startstopvms	Storage account	MAIC-RG	East US
<input type="checkbox"/>  StartStopVMs-AI	Application Insights	MAIC-RG	East US
<input type="checkbox"/>  StartStopVMs-FA	Function App	MAIC-RG	East US
<input type="checkbox"/>  ststv2_vms_AutoStop	Logic app	MAIC-RG	East US
<input type="checkbox"/>  ststv2_vms_Scheduled_start	Logic app	MAIC-RG	East US
<input type="checkbox"/>  ststv2_vms_Scheduled_stop	Logic app	MAIC-RG	East US
<input type="checkbox"/>  ststv2_vms_Sequenced_start	Logic app	MAIC-RG	East US
<input type="checkbox"/>  ststv2_vms_Sequenced_stop	Logic app	MAIC-RG	East US
<input type="checkbox"/>  Test-FunctionApp-Dev	Function App	MAIC-RG	East US
<input type="checkbox"/>  Test-FunctionApp-Dev	Application Insights	MAIC-RG	East US

Note

We are collecting operation and heartbeat telemetry to better assist you if you reach the support team for any troubleshooting. We are also collecting virtual machine event history to verify when the service acted on a virtual machine and how long a virtual machine was snoozed in order to determine the efficacy of the service.

Enable multiple subscriptions

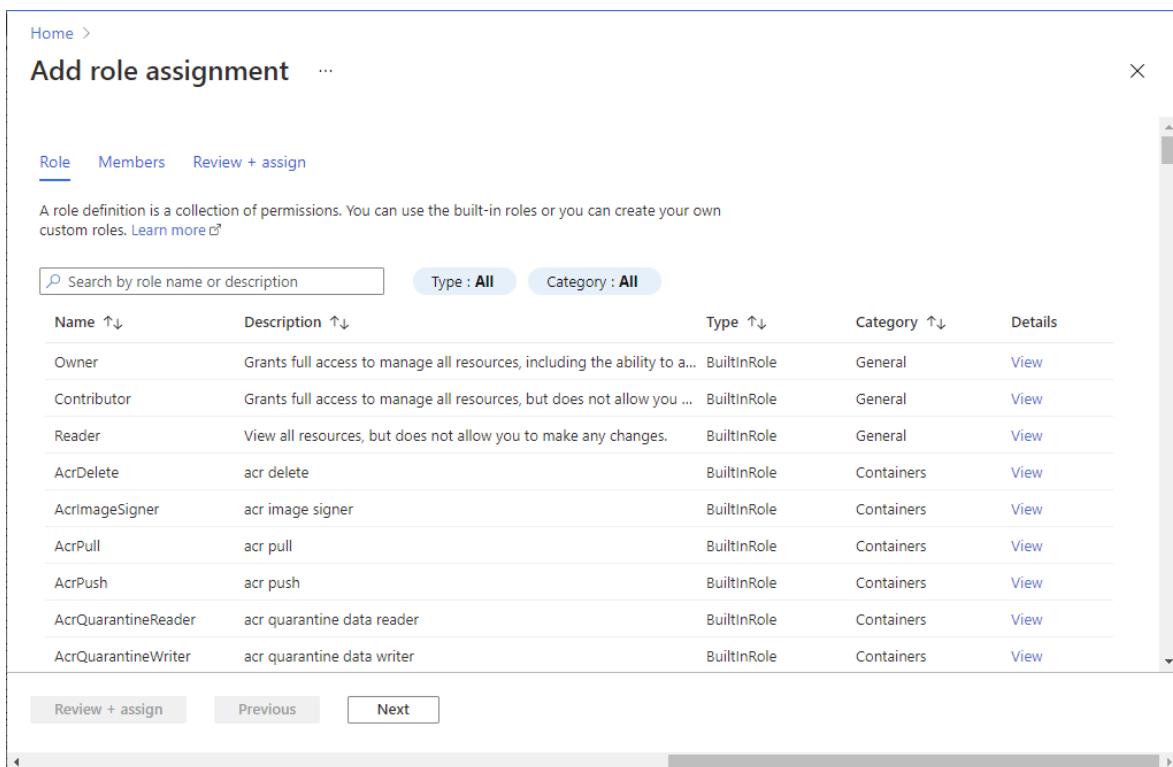
After the Start/Stop deployment completes, perform the following steps to enable Start/Stop VMs v2 to take action across multiple subscriptions.

1. Copy the value for the Azure Function App name that you specified during the deployment.
2. In the Azure portal, navigate to your secondary subscription.
3. Select **Access control (IAM)**.

4. Select Add > Add role assignment to open the Add role assignment page.
5. Assign the following role. For detailed steps, see [Assign Azure roles using the Azure portal](#).

 Expand table

Setting	Value
Role	Contributor
Assign access to	User, group, or service principal
Members	<Your Azure Function App name>



The screenshot shows the 'Add role assignment' dialog in the Azure portal. At the top, there's a breadcrumb navigation (Home >) and a title 'Add role assignment' with a three-dot ellipsis. Below the title are three tabs: 'Role' (which is selected and underlined), 'Members', and 'Review + assign'. A note below the tabs says: 'A role definition is a collection of permissions. You can use the built-in roles or you can create your own custom roles. [Learn more](#)'.

Below the note is a search bar labeled 'Search by role name or description' and filters for 'Type : All' and 'Category : All'. The main area is a table listing roles:

Name ↑↓	Description ↑↓	Type ↑↓	Category ↑↓	Details
Owner	Grants full access to manage all resources, including the ability to a...	BuiltInRole	General	View
Contributor	Grants full access to manage all resources, but does not allow you to ...	BuiltInRole	General	View
Reader	View all resources, but does not allow you to make any changes.	BuiltInRole	General	View
AcrDelete	acr delete	BuiltInRole	Containers	View
AcrImageSigner	acr image signer	BuiltInRole	Containers	View
AcrPull	acr pull	BuiltInRole	Containers	View
AcrPush	acr push	BuiltInRole	Containers	View
AcrQuarantineReader	acr quarantine data reader	BuiltInRole	Containers	View
AcrQuarantineWriter	acr quarantine data writer	BuiltInRole	Containers	View

At the bottom of the dialog are buttons for 'Review + assign', 'Previous', and 'Next'.

Configure schedules overview

To manage the automation method to control the start and stop of your VMs, you configure one or more of the included logic apps based on your requirements.

- Scheduled - Start and stop actions are based on a schedule you specify against Azure Resource Manager and classic VMs. `ststv2_vms_Scheduled_start` and `ststv2_vms_Scheduled_stop` configure the scheduled start and stop.
- Sequenced - Start and stop actions are based on a schedule targeting VMs with pre-defined sequencing tags. Only two named tags are supported - `sequencestart`

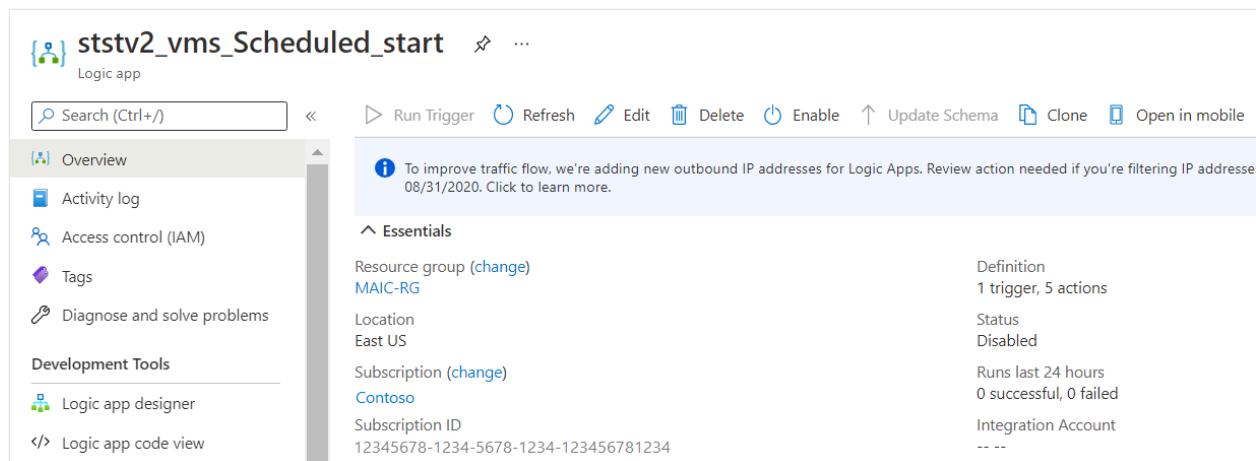
and `sequencestop`. `ststv2_vms_Sequenced_start` and `ststv2_vms_Sequenced_stop` configure the sequenced start and stop.

ⓘ Note

This scenario only supports Azure Resource Manager VMs.

- AutoStop - This functionality is only used for performing a stop action against both Azure Resource Manager and classic VMs based on its CPU utilization. It can also be a scheduled-based *take action*, which creates alerts on VMs and based on the condition, the alert is triggered to perform the stop action. `ststv2_vms_AutoStop` configures the auto-stop functionality.

If you need additional schedules, you can duplicate one of the Logic Apps provided using the **Clone** option in the Azure portal.



The screenshot shows the Azure Logic App 'ststv2_vms_Scheduled_start' in the Azure portal. The top navigation bar includes 'Run Trigger', 'Refresh', 'Edit', 'Delete', 'Enable', 'Update Schema', 'Clone', and 'Open in mobile'. A note at the top says: 'To improve traffic flow, we're adding new outbound IP addresses for Logic Apps. Review action needed if you're filtering IP addresses. 08/31/2020. Click to learn more.' The left sidebar has sections for 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Development Tools', 'Logic app designer', and 'Logic app code view'. The main content area displays 'Essentials' information: Resource group (MAIC-RG), Location (East US), Subscription (Contoso), and Subscription ID (12345678-1234-5678-1234-123456781234). It also shows 'Definition' (1 trigger, 5 actions), 'Status' (Disabled), and 'Runs last 24 hours' (0 successful, 0 failed). An 'Integration Account' section is shown with three dots.

Scheduled start and stop scenario

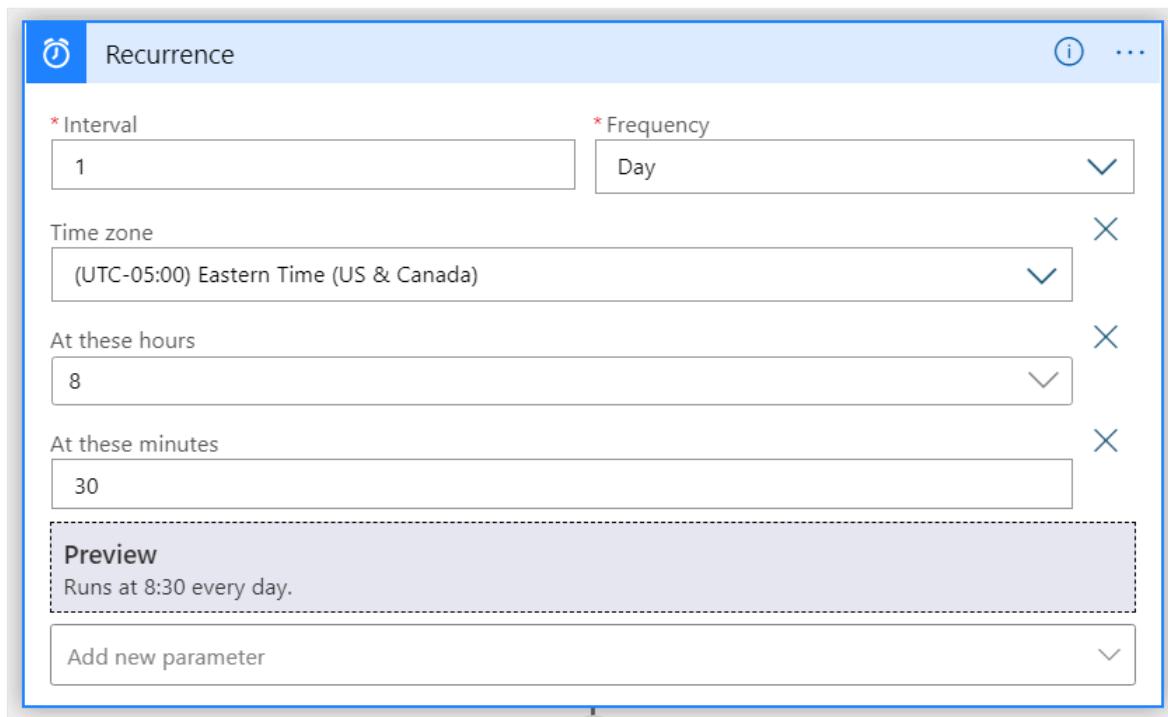
Perform the following steps to configure the scheduled start and stop action for Azure Resource Manager and classic VMs. For example, you can configure the `ststv2_vms_Scheduled_start` schedule to start them in the morning when you are in the office, and stop all VMs across a subscription when you leave work in the evening based on the `ststv2_vms_Scheduled_stop` schedule.

Configuring the logic app to just start the VMs is supported.

For each scenario, you can target the action against one or more subscriptions, single or multiple resource groups, and specify one or more VMs in an inclusion or exclusion list. You cannot specify them together in the same logic app.

1. Sign in to the [Azure portal](#) and then navigate to **Logic apps**.

- From the list of Logic apps, to configure scheduled start, select **ststv2_vms_Scheduled_start**. To configure scheduled stop, select **ststv2_vms_Scheduled_stop**.
- Select **Logic app designer** from the left-hand pane.
- After Logic App Designer appears, in the designer pane, select **Recurrence** to configure the logic app schedule. To learn about the specific recurrence options, see [Schedule recurring task](#).



! Note

If you do not provide a start date and time for the first recurrence, a recurrence will immediately run when you save the logic app, which might cause the VMs to start or stop before the scheduled run.

- In the designer pane, select **Function-Try** to configure the target settings. In the request body, if you want to manage VMs across all resource groups in the subscription, modify the request body as shown in the following example.

JSON

```
{
  "Action": "start",
  "EnableClassic": false,
  "RequestScopes": {
    "ExcludedVMLists": [],
    "Subscriptions": [
      ...
    ]
  }
}
```

```
        "/subscriptions/12345678-1234-5678-1234-123456781234/"  
    ]  
}  
}
```

Specify multiple subscriptions in the `subscriptions` array with each value separated by a comma as in the following example.

JSON

```
"Subscriptions": [  
    "/subscriptions/12345678-1234-5678-1234-123456781234/",  
    "/subscriptions/11111111-0000-1111-2222-444444444444/"  
]
```

In the request body, if you want to manage VMs for specific resource groups, modify the request body as shown in the following example. Each resource path specified must be separated by a comma. You can specify one resource group or more if required.

This example also demonstrates excluding a virtual machine. You can exclude the VM by specifying the VMs resource path or by wildcard.

JSON

```
{  
    "Action": "start",  
    "EnableClassic": false,  
    "RequestScopes": {  
        "Subscriptions": [  
            "/subscriptions/12345678-1234-5678-1234-123456781234/",  
            "/subscriptions/11111111-0000-1111-2222-444444444444/"  
        ],  
        "ResourceGroups": [  
            "/subscriptions/12345678-1234-5678-1234-  
123456781234/resourceGroups/rg1/",  
            "/subscriptions/11111111-0000-1111-2222-  
444444444444/resourceGroups/rg2/"  
        ],  
        "ExcludedVMLists": [  
            "/subscriptions/12345678-1234-5678-1234-  
123456781234/resourceGroups/vmrg1/providers/Microsoft.Compute/virtualMa  
chines/vm1"  
        ]  
    }  
}
```

Here the action will be performed on all the VMs except on the VM name starts with Az and Bz in both subscriptions.

JSON

```
{  
    "Action": "start",  
    "EnableClassic": false,  
    "RequestScopes": {  
        "ExcludedVMLists": ["Az*","Bz*"],  
        "Subscriptions": [  
            "/subscriptions/12345678-1234-5678-1234-123456781234/",  
            "/subscriptions/11111111-0000-1111-2222-444444444444/"  
        ]  
    }  
}
```

In the request body, if you want to manage a specific set of VMs within the subscription, modify the request body as shown in the following example. Each resource path specified must be separated by a comma. You can specify one VM if required.

JSON

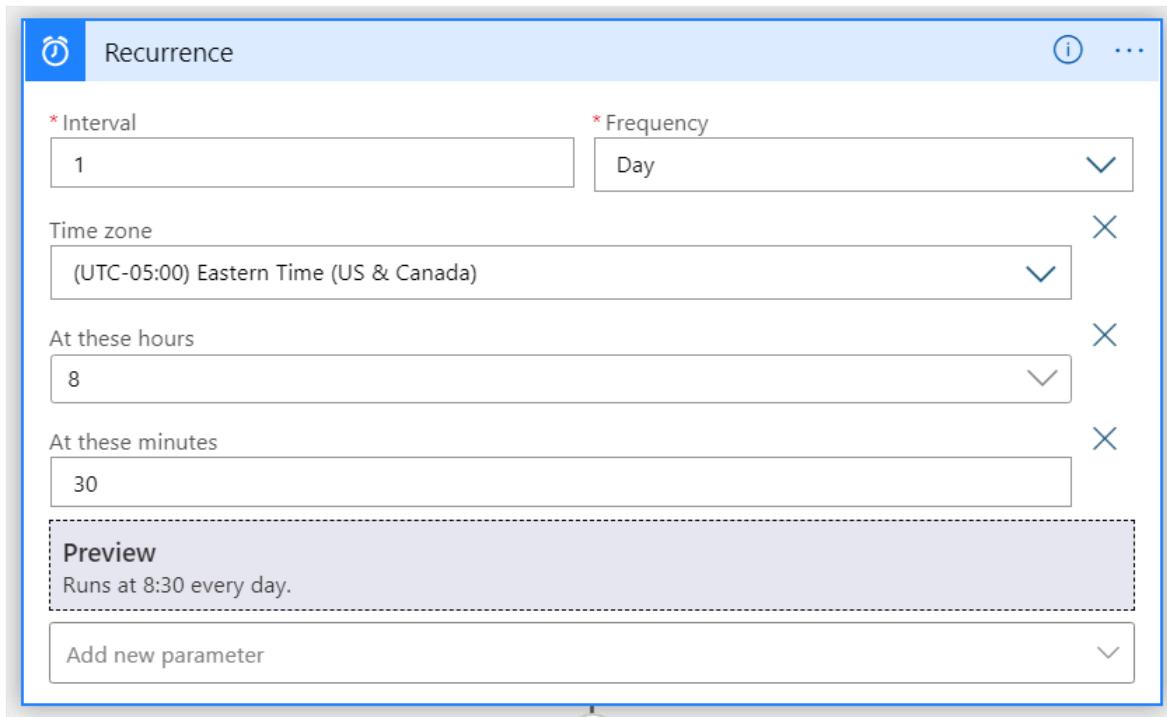
```
{  
    "Action": "start",  
    "EnableClassic": false,  
    "RequestScopes": {  
        "ExcludedVMLists": [],  
        "VMLists": [  
            "/subscriptions/12345678-1234-5678-1234-  
            123456781234/resourceGroups/rg1/providers/Microsoft.Compute/virtualMach  
            ines/vm1",  
            "/subscriptions/12345678-1234-5678-1234-  
            123456781234/resourceGroups/rg3/providers/Microsoft.Compute/virtualMach  
            ines/vm2",  
            "/subscriptions/11111111-0000-1111-2222-  
            444444444444/resourceGroups/rg2/providers/Microsoft.ClassicCompute/virt  
            ualMachines/vm30"  
        ]  
    }  
}
```

6. In the overview pane for the logic app, select **Enable**.

Sequenced start and stop scenario

In an environment that includes two or more components on multiple Azure Resource Manager VMs in a distributed application architecture, supporting the sequence in which components are started and stopped in order is important. Make sure you have applied the **sequencestart** and **sequencestop** tags to the target VMs as described on the [Overview page](#) before configuring this scenario.

1. From the list of Logic apps, to configure sequenced start, select **ststv2_vms_Sequenced_start**. To configure sequenced stop, select **ststv2_vms_Sequenced_stop**.
2. Select **Logic app designer** from the left-hand pane.
3. After Logic App Designer appears, in the designer pane, select **Recurrence** to configure the logic app schedule. To learn about the specific recurrence options, see [Schedule recurring task](#).



(!) Note

If you do not provide a start date and time for the first recurrence, a recurrence will immediately run when you save the logic app, which might cause the VMs to start or stop before the scheduled run.

4. In the designer pane, select **Function-Try** to configure the target settings and then select the **</> Code view** button in the top menu to edit the code for the **Function-Try** element. In the request body, if you want to manage VMs across all resource groups in the subscription, modify the request body as shown in the following example.

JSON

```
{  
    "Action": "start",  
    "EnableClassic": false,  
    "RequestScopes": {  
        "ExcludedVMLists": [],  
        "Subscriptions": [  
            "/subscriptions/12345678-1234-5678-1234-123456781234/"  
        ]  
    },  
    "Sequenced": true  
}
```

Specify multiple subscriptions in the `subscriptions` array with each value separated by a comma as in the following example.

JSON

```
"Subscriptions": [  
    "/subscriptions/12345678-1234-5678-1234-123456781234/",  
    "/subscriptions/11111111-0000-1111-2222-444444444444/"  
]
```

In the request body, if you want to manage VMs for specific resource groups, modify the request body as shown in the following example. Each resource path specified must be separated by a comma. You can specify one resource group if required.

This example also demonstrates excluding a virtual machine by its resource path compared to the example for scheduled start/stop, which used wildcards.

JSON

```
{  
    "Action": "start",  
    "EnableClassic": false,  
    "RequestScopes": {  
        "Subscriptions": [  
            "/subscriptions/12345678-1234-5678-1234-123456781234/",  
            "/subscriptions/11111111-0000-1111-2222-444444444444/"  
        ],  
        "ResourceGroups": [  
            "/subscriptions/12345678-1234-5678-1234-  
            123456781234/resourceGroups/rg1/",  
            "/subscriptions/11111111-0000-1111-2222-  
            444444444444/resourceGroups/rg2/"  
        ],  
        "ExcludedVMLists": [  
            "/subscriptions/12345678-1234-5678-1234-  
            123456781234/vmGroups/rg1/",  
            "/subscriptions/11111111-0000-1111-2222-  
            444444444444/vmGroups/rg2/"  
        ]  
    }  
}
```

```

        "/subscriptions/12345678-1234-5678-1234-
123456781234/resourceGroups/vmrg1/providers/Microsoft.Compute/virtualMa
chines/vm1"
    ],
},
"Sequenced": true
}

```

In the request body, if you want to manage a specific set of VMs within a subscription, modify the request body as shown in the following example. Each resource path specified must be separated by a comma. You can specify one VM if required.

JSON

```

{
  "Action": "start",
  "EnableClassic": false,
  "RequestScopes": {
    "ExcludedVMLists": [],
    "VMLists": [
      "/subscriptions/12345678-1234-5678-1234-
123456781234/resourceGroups/rg1/providers/Microsoft.Compute/virtualMach
ines/vm1",
      "/subscriptions/12345678-1234-5678-1234-
123456781234/resourceGroups/rg2/providers/Microsoft.ClassicCompute/virt
ualMachines/vm2",
      "/subscriptions/11111111-0000-1111-2222-
444444444444/resourceGroups/rg2/providers/Microsoft.ClassicCompute/virt
ualMachines/vm30"
    ]
  },
  "Sequenced": true
}

```

Auto stop scenario

Start/Stop VMs v2 can help manage the cost of running Azure Resource Manager and classic VMs in your subscription by evaluating machines that aren't used during non-peak periods, such as after hours, and automatically shutting them down if processor utilization is less than a specified percentage.

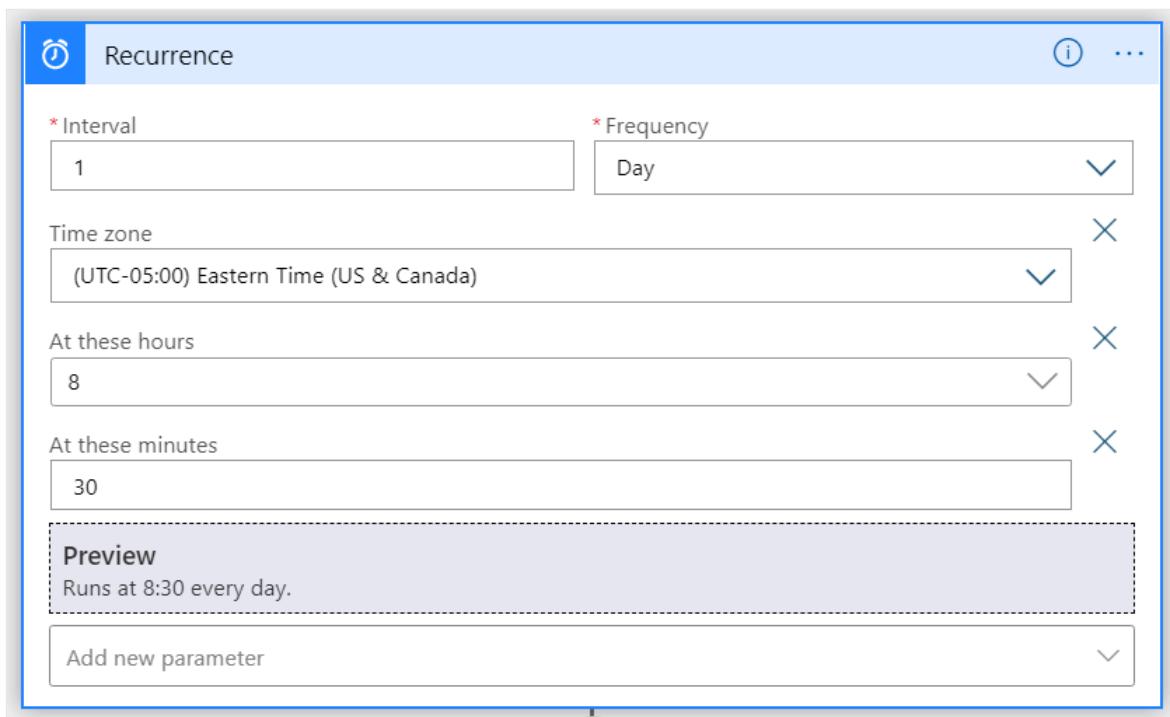
The following metric alert properties in the request body support customization:

- AutoStop_MetricName
- AutoStop_Condition
- AutoStop_Threshold

- AutoStop_Description
- AutoStop_Frequency
- AutoStop_Severity
- AutoStop_Threshold
- AutoStop_TimeAggregationOperator
- AutoStop_TimeWindow

To learn more about how Azure Monitor metric alerts work and how to configure them see [Metric alerts in Azure Monitor](#).

1. From the list of Logic apps, to configure auto stop, select **ststv2_vms_AutoStop**.
2. Select **Logic app designer** from the left-hand pane.
3. After Logic App Designer appears, in the designer pane, select **Recurrence** to configure the logic app schedule. To learn about the specific recurrence options, see [Schedule recurring task](#).



4. In the designer pane, select **Function-Try** to configure the target settings. In the request body, if you want to manage VMs across all resource groups in the subscription, modify the request body as shown in the following example.

JSON

```
{
  "Action": "stop",
  "EnableClassic": false,
  "AutoStop_MetricName": "Percentage CPU",
  "AutoStop_Condition": "LessThan",
  "AutoStop_Description": "Alert to stop the VM if the CPU % falls"
}
```

```

        "below the threshold",
        "AutoStop_Frequency": "00:05:00",
        "AutoStop_Severity": "2",
        "AutoStop_Threshold": "5",
        "AutoStop_TimeAggregationOperator": "Average",
        "AutoStop_TimeWindow": "06:00:00",
        "RequestScopes": {
            "Subscriptions": [
                "/subscriptions/12345678-1111-2222-3333-1234567891234/",
                "/subscriptions/12345678-2222-4444-5555-1234567891234/"
            ],
            "ExcludedVMLists": []
        }
    }
}

```

In the request body, if you want to manage VMs for specific resource groups, modify the request body as shown in the following example. Each resource path specified must be separated by a comma. You can specify one resource group if required.

JSON

```
{
    "Action": "stop",
    "AutoStop_Condition": "LessThan",
    "AutoStop_Description": "Alert to stop the VM if the CPU % falls below the threshold",
    "AutoStop_Frequency": "00:05:00",
    "AutoStop_MetricName": "Percentage CPU",
    "AutoStop_Severity": "2",
    "AutoStop_Threshold": "5",
    "AutoStop_TimeAggregationOperator": "Average",
    "AutoStop_TimeWindow": "06:00:00",
    "EnableClassic": false,
    "RequestScopes": {
        "ExcludedVMLists": [],
        "ResourceGroups": [
            "/subscriptions/12345678-1111-2222-3333-1234567891234/resourceGroups/vmrg1/",
            "/subscriptions/12345678-1111-2222-3333-1234567891234/resourceGroupsVmrg2/",
            "/subscriptions/12345678-2222-4444-5555-1234567891234/resourceGroups/VMHostingRG/"
        ]
    }
}
```

In the request body, if you want to manage a specific set of VMs within the subscription, modify the request body as shown in the following example. Each

resource path specified must be separated by a comma. You can specify one VM if required.

```
JSON

{
    "Action": "stop",
    "AutoStop_Condition": "LessThan",
    "AutoStop_Description": "Alert to stop the VM if the CPU % falls below the threshold",
    "AutoStop_Frequency": "00:05:00",
    "AutoStop_MetricName": "Percentage CPU",
    "AutoStop_Severity": "2",
    "AutoStop_Threshold": "5",
    "AutoStop_TimeAggregationOperator": "Average",
    "AutoStop_TimeWindow": "06:00:00",
    "EnableClassic": false,
    "RequestScopes": {
        "ExcludedVMLists": [],
        "VMLists": [
            "/subscriptions/12345678-1111-2222-3333-1234567891234/resourceGroups/rg3/providers/Microsoft.ClassicCompute/virtualMachines/Clasyvm11",
            "/subscriptions/12345678-1111-2222-3333-1234567891234/resourceGroups/vmrg1/providers/Microsoft.Compute/virtualMachines/vm1"
        ]
    }
}
```

VM Tags

You can also include or exclude specific VMs from start and stop actions by setting tags on the VMs themselves. To add a tag, navigate to the specific VM, select **Tags** from the left menu, and add a tag named `ssv2excludevm`. To exclude this VM from the start or stop action, set the value of this new tag to `true`. To include the VM in the action, set the value to `false`. This gives you a way to exclude specific VMs without having to update `ExcludedVMLists` in the payload configuration.

Next steps

To learn how to monitor status of your Azure VMs managed by the Start/Stop VMs v2 feature and perform other management tasks, see the [Manage Start/Stop VMs](#) article.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

How to manage Start/Stop VMs v2

Article • 06/15/2022

Azure dashboard

Start/Stop VMs v2 includes a [dashboard](#) to help you understand the management scope and recent operations against your VMs. It is a quick and easy way to verify the status of each operation that's performed on your Azure VMs. The visualization in each tile is based on a Log query and to see the query, select the **Open in logs blade** option in the right-hand corner of the tile. This opens the [Log Analytics](#) tool in the Azure portal, and from here you can evaluate the query and modify to support your needs, such as custom [log alerts](#), a custom [workbook](#), etc.

The log data each tile in the dashboard displays is refreshed every hour, with a manual refresh option on demand by clicking the **Refresh** icon on a given visualization, or by refreshing the full dashboard.

To learn about working with a log-based dashboard, see the following [tutorial](#).

Configure email notifications

To change email notifications after Start/Stop VMs v2 is deployed, you can modify the action group created during deployment.

1. In the Azure portal, navigate to **Monitor**, then **Alerts**. Select **Action groups**.
2. On the **Manage actions** page, select the action group called **StartStopV2_VM_Notification**.

All services > Monitor >

Manage actions

Rules management

Add action group Columns Refresh Delete

Action groups Action rules (preview)

Subscriptions : Contoso Resource groups : All resource groups

Showing 1 to 9 of 9 records.

Action group name ↑	Short name ↑	Resource group ↑↓	Subscription	Status
<input type="checkbox"/> StartStopV2_VM_Notification	StStAlertV2	maic-rg	12345678-1234-5678-1234-123456781234	Enabled

← Previous Page 1 of 1 Next →

The screenshot shows the 'Manage actions' page in the Azure Monitor. It displays a table of action groups. There is one entry in the table:

Action group name ↑	Short name ↑	Resource group ↑↓	Subscription	Status
<input type="checkbox"/> StartStopV2_VM_Notification	StStAlertV2	maic-rg	12345678-1234-5678-1234-123456781234	Enabled

3. On the **StartStopV2_VM_Notification** page, you can modify the Email/SMS/Push/Voice notification options. Add other actions or update your existing configuration to this action group and then click **OK** to save your changes.

Email/SMS message/Push/Voice

X

Add or edit an Email/SMS/Push/Voice action

Email

Email *



SMS (Carrier charges may apply)

Country code



Phone number

Azure app Push Notifications

Azure account email

Voice

Country code

1

Phone number

Enable the common alert schema. [Learn more](#)

Yes

No

OK

To learn more about action groups, see [action groups](#)

The following screenshot is an example email that is sent when the feature shuts down virtual machines.

The screenshot shows an email message in Microsoft Outlook. The subject of the email is "Start/Stop VMs during off-hours : Scheduled azure function has attempted an action". The message is from "Microsoft Azure" to "Jack Smith". A note at the top says, "If there are problems with how this message is displayed, click here to view it in a web browser." Below the message content, there is a "Microsoft Azure" logo and a section titled "Your Azure Monitor alert was triggered". It states, "We are notifying you because there are 1 counts of 'ScheduledStartStop_AzFunc'." A table titled "Essentials" provides details about the alert:

Name	ScheduledStartStop_AzFunc
Description	Start/Stop VMs during off-hours : Scheduled azure function has attempted an action
Severity	4
Resource	StartStopVMs-AI
Search interval start time	March 16, 2021 12:25:39 UTC
Search interval duration	5 min

Next steps

To handle problems during VM management, see [Troubleshoot Start/Stop VMs v2 issues](#).

How to remove Start/Stop VMs v2

Article • 06/09/2022

After you enable the Start/Stop VMs v2 feature to manage the running state of your Azure VMs, you may decide to stop using it. Removing this feature can be done by deleting the resource group dedicated to store the following resources in support of Start/Stop VMs v2:

- The Azure Functions applications
- Schedules in Azure Logic Apps
- The Application Insights instance
- Azure Storage account

ⓘ Note

If you run into problems during deployment, you encounter an issue when using Start/Stop VMs v2, or if you have a related question, you can submit an issue on [GitHub](#). Filing an Azure support incident from the [Azure support site](#) is not available for this version.

Delete the dedicated resource group

To delete the resource group, follow the steps outlined in the [Azure Resource Manager resource group and resource deletion](#) article.

ⓘ Note

You might need to manually remove the managed identity associated with the removed Start Stop V2 function app. You can determine whether you need to do this by going to your subscription and selecting **Access Control (IAM)**. From there you can filter by Type: **App Services or Function Apps**. If you find a managed identity that was left over from your removed Start Stop V2 installation, you must remove it. Leaving this managed identity could interfere with future installations.

Next steps

To re-deploy this feature, see [Deploy Start/Stop v2](#).

Adding pre-actions to schedules in Start/Stop VMs v2

Article • 09/30/2022

Pre-actions are a set of actions in Start/Stop VMs v2 that execute before scheduled start or stop actions. Some scenarios for using pre-actions before a start or stop action include:

- Create a backup of an Azure SQL Database.
- Send a message to Azure Application Insights.
- Call an external API.

Because Start/Stop VMs v2 uses [Azure Logic Apps](#) to manage its schedules, it's easy to add one or more pre-actions before the main action. To learn more about Logic Apps, see the [Logic Apps documentation](#).

This article describes how to use the Logic Apps Designer in the Azure portal to add an HTTP request pre-action to an existing scheduled start action in Start/Stop VMs v2. In your implementation, the pre-action can be any action supported by Logic Apps.

ⓘ Note

At this time, Start/Stop VMs v2 only supports pre-actions, which are run before the execution of the main action. Because Log Apps runs Start/Stop VMs v2 actions asynchronously, there's currently no way to trigger a post-action that occurs after the main action completes.

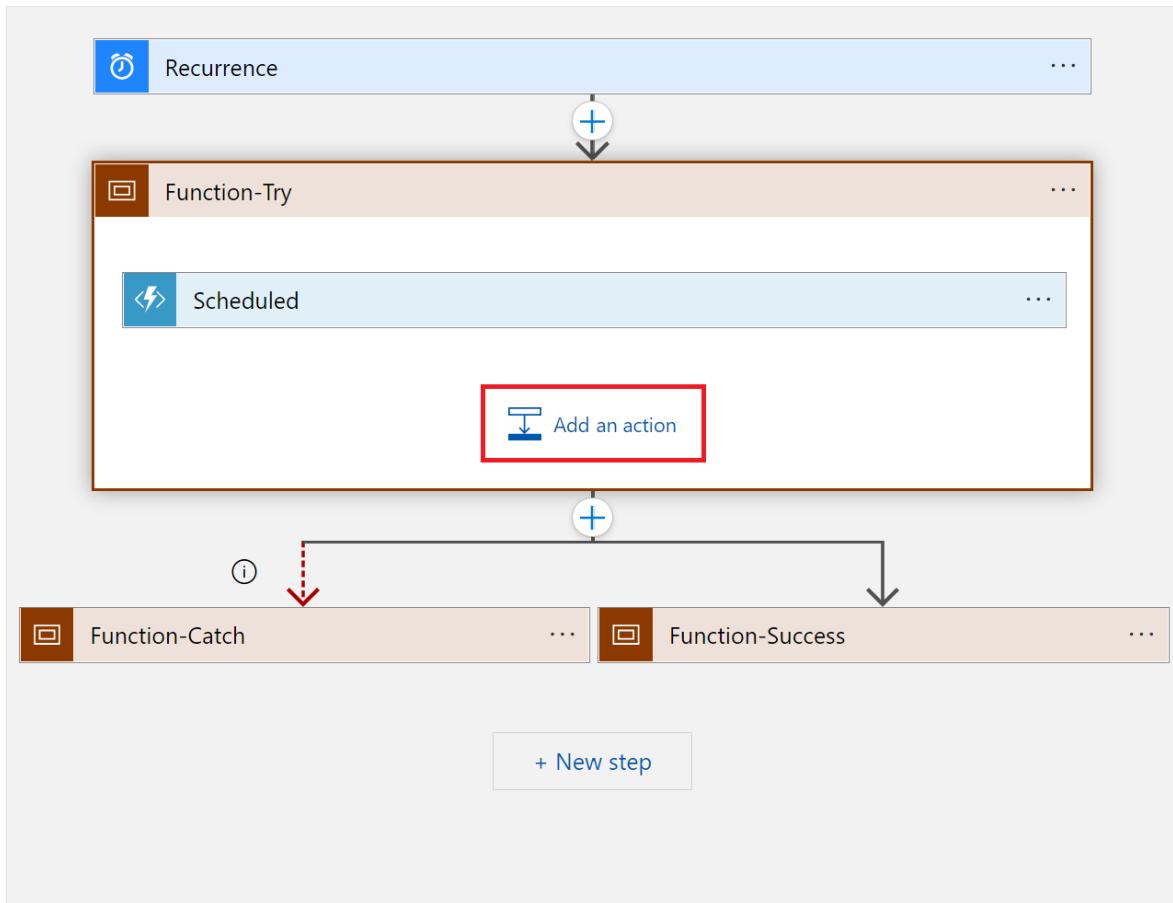
Prerequisites

You must first complete the steps in [Deploy Start/Stop VMs v2 to an Azure subscription](#), or else complete a default deployment from the [Start Stop V2 Deployments GitHub repository](#). Logic app and action names are based on the ones in a default deployment of Start/Stop VMs v2.

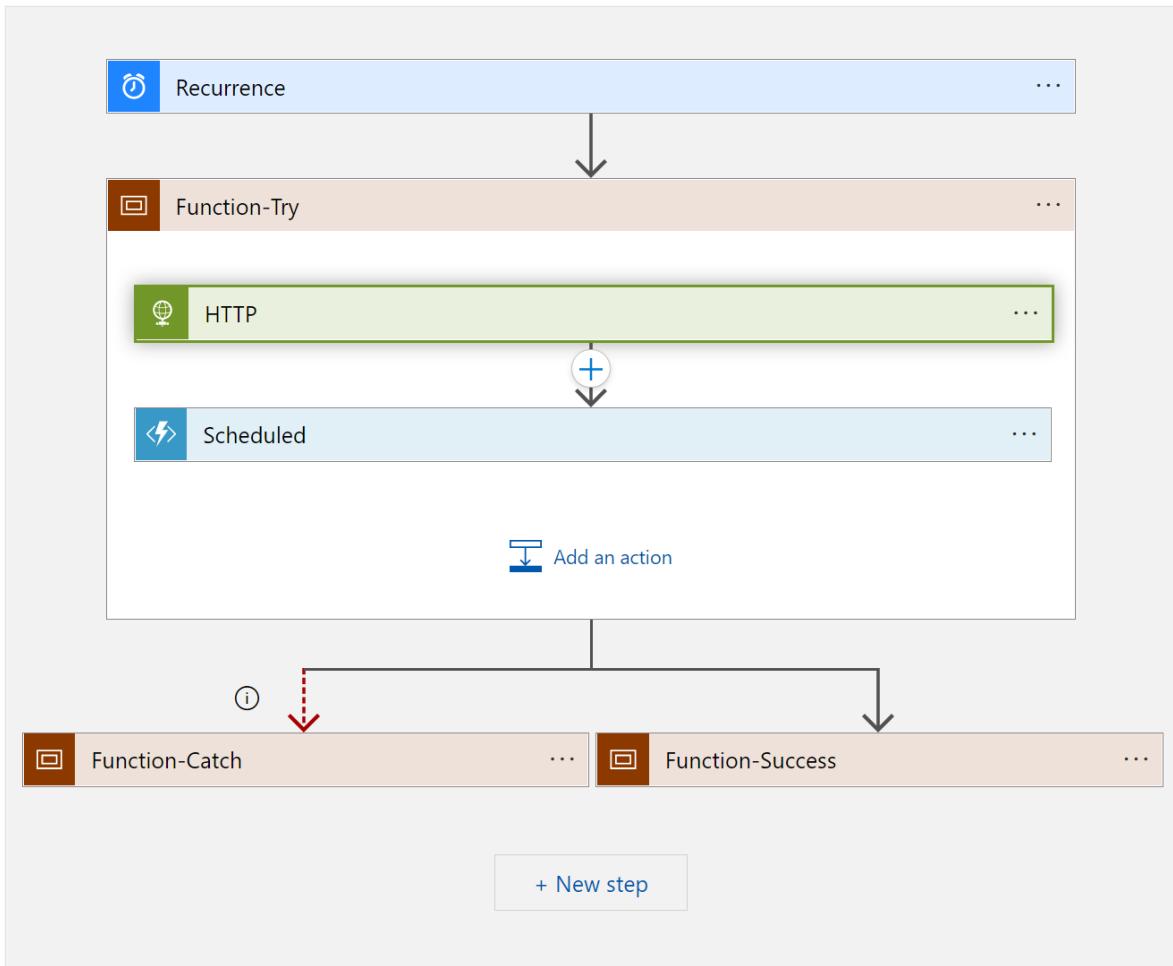
Create an HTTP request pre-action

The steps in this section require the `ststv2_vms_Scheduled_start` logic app that was created and deployed when you completed the [deployment article](#). However, the same basic process is used for all scheduled actions.

1. In the [Azure portal](#), search for and navigate to the resource group you created when you deployed Start/Stop VMs v2.
2. In the resource group, choose the logic app named `ststv2_vms_Scheduled_start`, which represents the default scheduled start action.
3. In **Overview** page of the logic app, select **Edit**.
4. In the Logic Apps Designer page, select **Function-Try** and then select **Add an action**.



5. Choose **HTTP**, select the **HTTP Method**, and add the **URL**. This HTTP request will be the pre-action for the scheduled start action, after you change the action order in **Function-Try**. You can also configure the HTTP action at a later time.
6. Drag the **Scheduled** action below the new **HTTP** action in the **Function-Try** step. The pre-action must come before the scheduled action in the step. Your app should now look like the following example:



At this point, you've defined a pre-action that's run before the start action scheduled by `ststv2_vms_Scheduled_start`.

Next steps

If you have issues working with Start/Stop VMs v2, see the [Troubleshoot Guide](#). For more assistance, you can also create an issue in the [Start Stop V2 Deployments GitHub repository](#).

Troubleshoot common issues with Start/Stop VMs

Article • 06/09/2022

This article provides information on troubleshooting and resolving issues that may occur while attempting to install and configure Start/Stop VMs. For general information, see [Start/Stop VMs overview](#).

General validation and troubleshooting

This section covers how to troubleshoot general issues with the schedules scenarios and help identify the root cause.

Azure dashboard

You can start by reviewing the Azure shared dashboard. The Azure shared dashboard deployed as part of Start/Stop VMs v2 is a quick and easy way to verify the status of each operation that's performed on your VMs. Refer to the **Recently attempted actions on VMs** tile to see all the recent operations executed on your VMs. There is some latency, around five minutes, for data to show up in the report as it pulls data from the Application Insights resource.

Logic Apps

Depending on which Logic Apps you have enabled to support your start/stop scenario, you can review its run history to help identify why the scheduled startup/shutdown scenario did not complete successfully for one or more target VMs. To learn how to review this in detail, see [Logic Apps run history](#).

Azure Storage

You can review the details for the operations performed on the VMs that are written to the table **requeststoretable** in the Azure storage account used for Start/Stop VMs v2. Perform the following steps to view those records.

1. Navigate to the storage account in the Azure portal and in the account select **Storage Explorer** from the left-hand pane.
2. Select **TABLES** and then select **requeststoretable**.

3. Each record in the table represents the start/stop action performed against an Azure VM based on the target scope defined in the logic app scenario. You can filter the results by any one of the record properties (for example, TIMESTAMP, ACTION, or TARGETTOLEVELRESOURCENAME).

Azure Functions

You can review the latest invocation details for any of the Azure Functions responsible for the VM start and stop execution. First let's review the execution flow.

The execution flow for both **Scheduled** and **Sequenced** scenario is controlled by the same function. The payload schema is what determines which scenario is performed. For the **Scheduled** scenario, the execution flow is - **Scheduled HTTP** > **VirtualMachineRequestOrchestrator** Queue > **VirtualMachineRequestExecutor** Queue.

From the logic app, the **Scheduled** HTTP function is invoked with Payload schema. Once the **Scheduled** HTTP function receives the request, it sends the information to the **Orchestrator** queue function, which in turn creates several queues for each VM to perform the action.

Perform the following steps to see the invocation details.

1. In the Azure portal, navigate to **Azure Functions**.
2. Select the Function app for Start/Stop VMs v2 from the list.
3. Select **Functions** from the left-hand pane.
4. In the list, you see several functions associated for each scenario. Select the **Scheduled** HTTP function.
5. Select **Monitor** from the left-hand pane.
6. Select the latest execution trace to see the invocation details and the message section for detailed logging.
7. Repeat the same steps for each function described as part of reviewing the execution flow earlier.

To learn more about monitoring Azure Functions, see [Analyze Azure Functions telemetry in Application Insights](#).

Next steps

Learn more about monitoring Azure Functions and logic apps:

- [Monitor Azure Functions](#).
- [How to configure monitoring for Azure Functions](#).

- [Monitor logic apps.](#)
- If you run into problems during deployment, you encounter an issue when using Start/Stop VMs v2, or if you have a related question, you can submit an issue on [GitHub](#). Filing an Azure support incident from the [Azure support site](#) is also available for this version.

Use Azure Functions to connect to an Azure SQL Database

Article • 01/30/2023

This article shows you how to use Azure Functions to create a scheduled job that connects to an Azure SQL Database or Azure SQL Managed Instance. The function code cleans up rows in a table in the database. The new C# function is created based on a pre-defined timer trigger template in Visual Studio 2019. To support this scenario, you must also set a database connection string as an app setting in the function app. For Azure SQL Managed Instance you need to [enable public endpoint](#) to be able to connect from Azure Functions. This scenario uses a bulk operation against the database.

If this is your first experience working with C# Functions, you should read the [Azure Functions C# developer reference](#).

Prerequisites

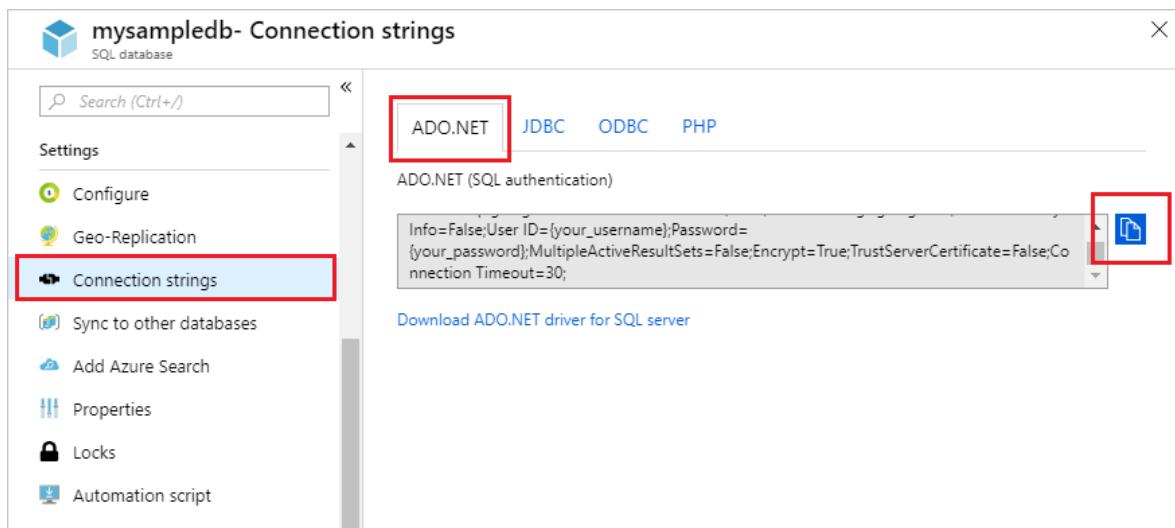
- Complete the steps in the article [Create your first function using Visual Studio](#) to create a local function app that targets version 2.x or a later version of the runtime. You must also have published your project to a function app in Azure.
- This article demonstrates a Transact-SQL command that executes a bulk cleanup operation in the **SalesOrderHeader** table in the AdventureWorksLT sample database. To create the AdventureWorksLT sample database, complete the steps in the article [Create a database in Azure SQL Database using the Azure portal](#).
- You must add a [server-level firewall rule](#) for the public IP address of the computer you use for this quickstart. This rule is required to be able access the SQL Database instance from your local computer.

Get connection information

You need to get the connection string for the database you created when you completed [Create a database in Azure SQL Database using the Azure portal](#).

1. Sign in to the [Azure portal](#).
2. Select **SQL Databases** from the left-hand menu, and select your database on the **SQL databases** page.

3. Select **Connection strings** under **Settings** and copy the complete ADO.NET connection string. For Azure SQL Managed Instance copy connection string for public endpoint.

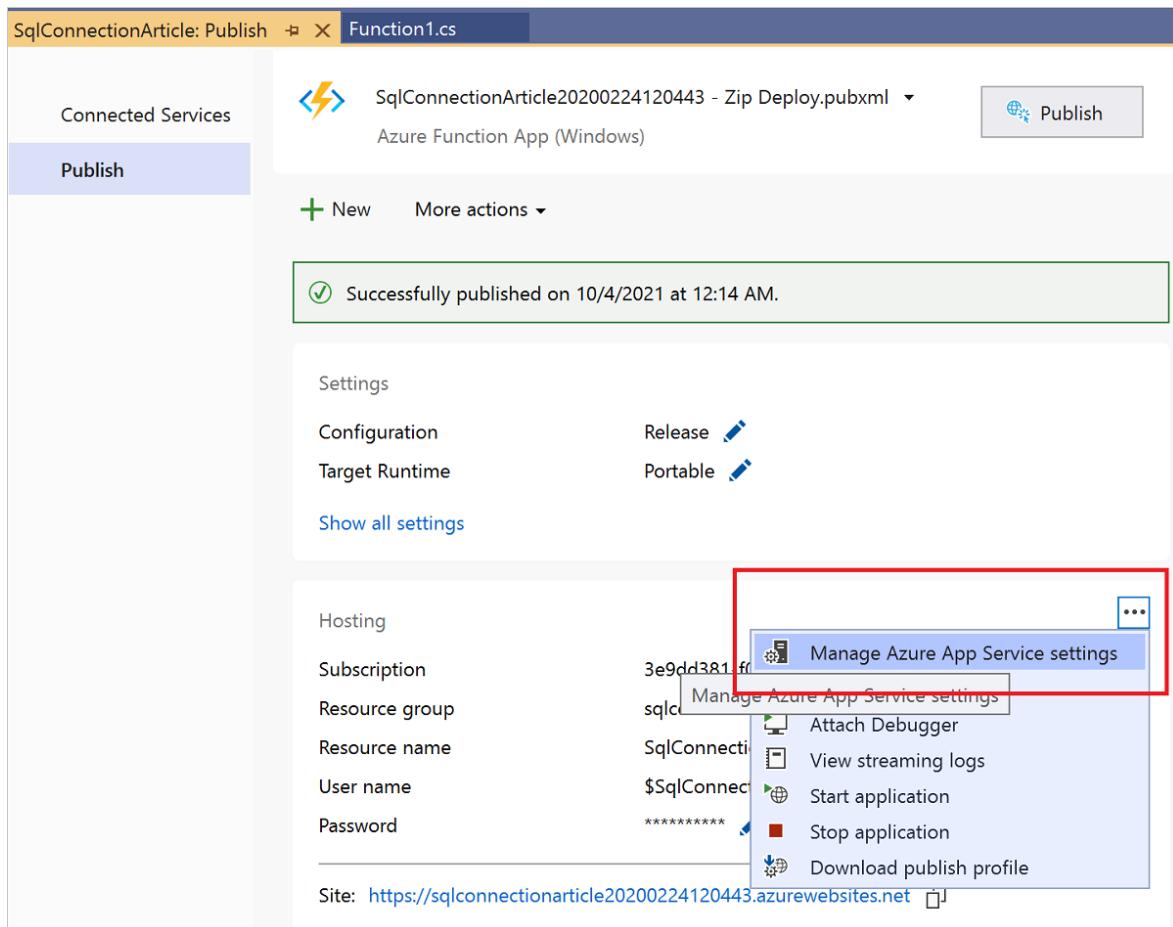


Set the connection string

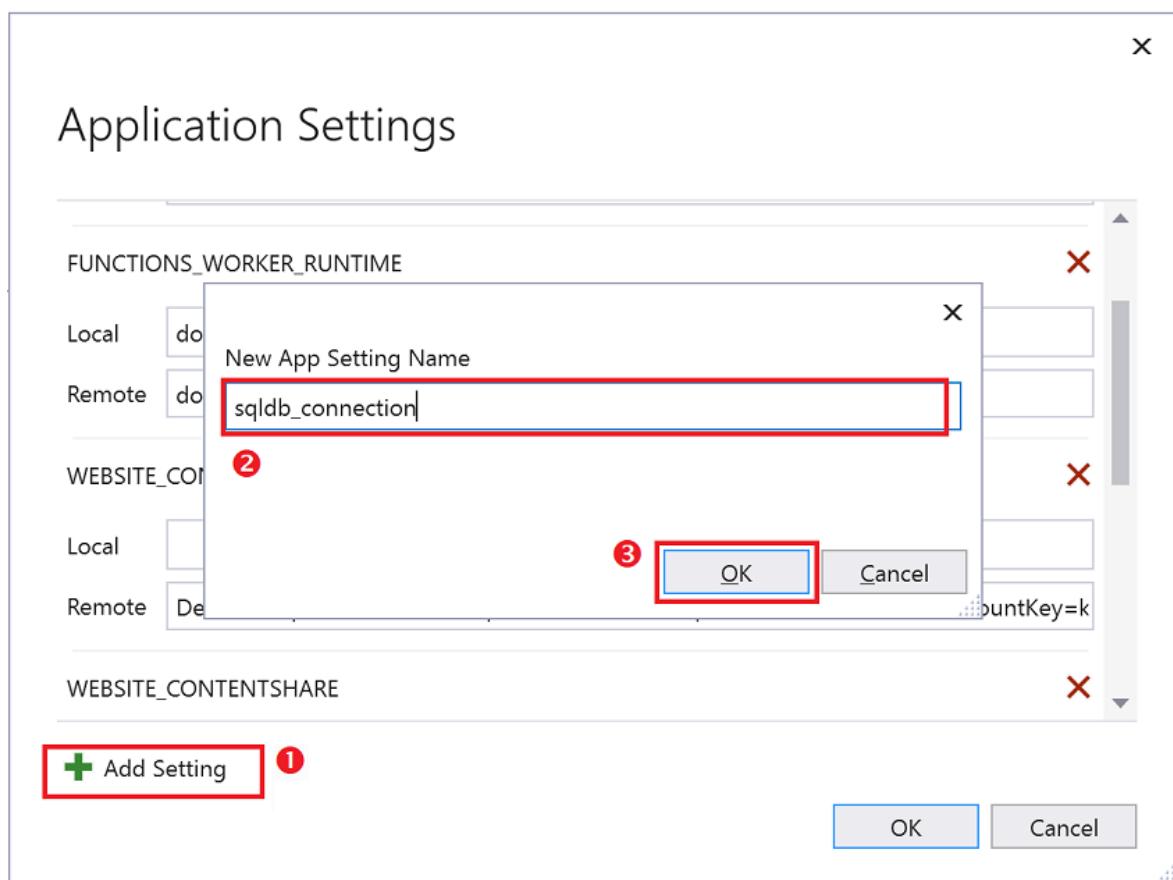
A function app hosts the execution of your functions in Azure. As a best security practice, store connection strings and other secrets in your function app settings. Using application settings prevents accidental disclosure of the connection string with your code. You can access app settings for your function app right from Visual Studio.

You must have previously published your app to Azure. If you haven't already done so, [Publish your function app to Azure](#).

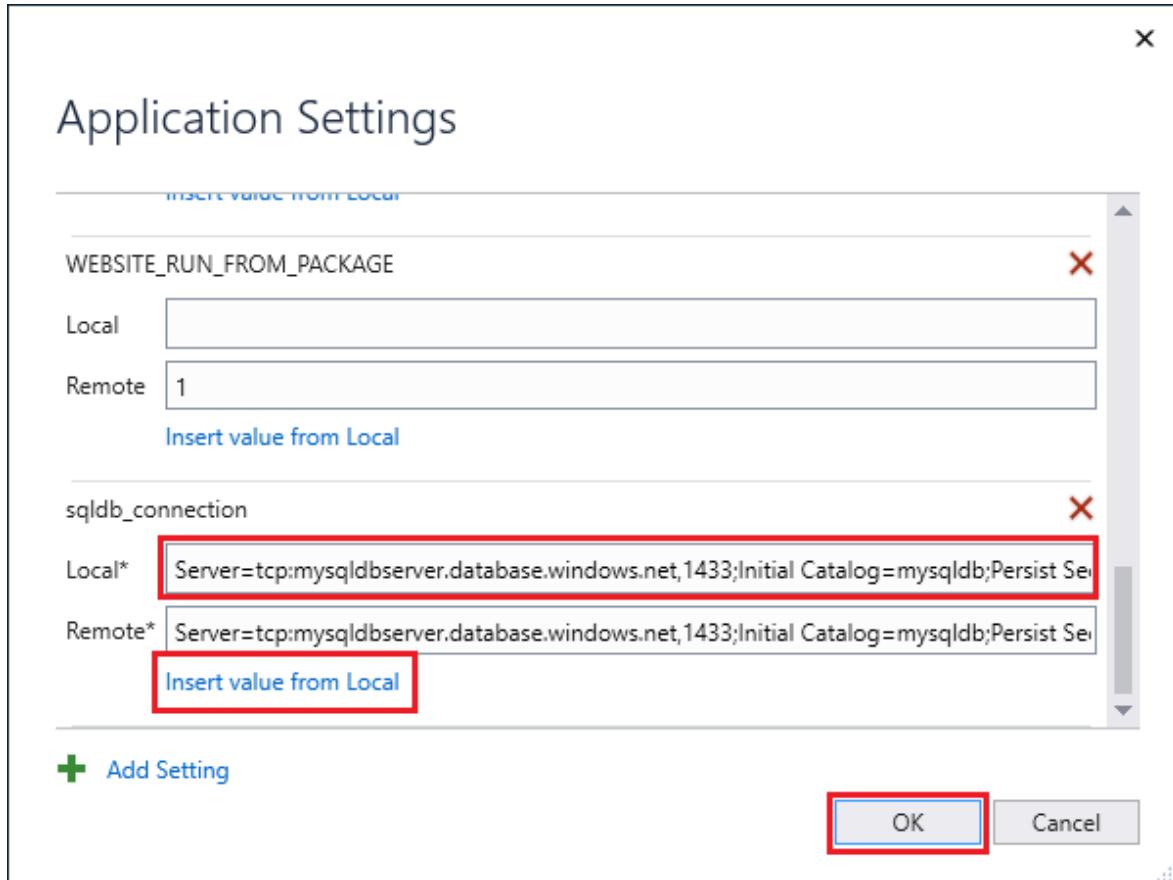
1. In Solution Explorer, right-click the function app project and choose **Publish**.
2. On the **Publish** page, select the ellipses (...) in the **Hosting** area, and choose **Manage Azure App Service settings**.



3. In Application Settings select Add setting, in New app setting name type `sqldb_connection`, and select OK.



4. In the new `sqlDb_connection` setting, paste the connection string you copied in the previous section into the **Local** field and replace `{your_username}` and `{your_password}` placeholders with real values. Select **Insert value from local** to copy the updated value into the **Remote** field, and then select **OK**.



The connection strings are stored encrypted in Azure (**Remote**). To prevent leaking secrets, the `local.settings.json` project file (**Local**) should be excluded from source control, such as by using a `.gitignore` file.

Add the SqlClient package to the project

You need to add the NuGet package that contains the `SqlClient` library. This data access library is needed to connect to SQL Database.

1. Open your local function app project in Visual Studio 2022.
2. In Solution Explorer, right-click the function app project and choose **Manage NuGet Packages**.
3. On the **Browse** tab, search for `Microsoft.Data.SqlClient` and, when found, select it.
4. In the `Microsoft.Data.SqlClient` page, select version `5.1.0` and then click **Install**.

5. When the install completes, review the changes and then click **OK** to close the **Preview** window.

6. If a **License Acceptance** window appears, click **I Accept**.

Now, you can add the C# function code that connects to your SQL Database.

Add a timer triggered function

1. In Solution Explorer, right-click the function app project and choose **Add > New Azure function**.

2. With the **Azure Functions** template selected, name the new item something like `DatabaseCleanup.cs` and select **Add**.

3. In the **New Azure function** dialog box, choose **Timer trigger** and then **Add**. This dialog creates a code file for the timer triggered function.

4. Open the new code file and add the following using statements at the top of the file:

```
C#  
  
using Microsoft.Data.SqlClient;  
using System.Threading.Tasks;
```

5. Replace the existing `Run` function with the following code:

```
C#  
  
[FunctionName("DatabaseCleanup")]
public static async Task Run([TimerTrigger("*/15 * * * *")]TimerInfo
myTimer, ILogger log)
{
    // Get the connection string from app settings and use it to create
    // a connection.
    var str = Environment.GetEnvironmentVariable("sqlDb_connection");
    using (SqlConnection conn = new SqlConnection(str))
    {
        conn.Open();
        var text = "UPDATE SalesLT.SalesOrderHeader " +
                  "SET [Status] = 5 WHERE ShipDate < GetDate();";

        using (SqlCommand cmd = new SqlCommand(text, conn))
        {
            // Execute the command and log the # rows affected.
            var rows = await cmd.ExecuteNonQueryAsync();
            log.LogInformation($"{rows} rows were updated");
        }
    }
}
```

```
        }
    }
}
```

This function runs every 15 seconds to update the `Status` column based on the ship date. To learn more about the Timer trigger, see [Timer trigger for Azure Functions](#).

6. Press **F5** to start the function app. The [Azure Functions Core Tools](#) execution window opens behind Visual Studio.
7. At 15 seconds after startup, the function runs. Watch the output and note the number of rows updated in the **SalesOrderHeader** table.

```
C:\AzureFunctionsTools\Releases\2.10.1\cli\func.exe
[10/29/2018 10:52:49 PM] SqlConnectionArticle.Function1.Run
[10/29/2018 10:52:49 PM]
[10/29/2018 10:52:49 PM] Host initialized (467ms)
[10/29/2018 10:52:49 PM] The next 5 occurrences of the 'SqlConnectionArticle.DatabaseCleanup.Run' schedule will be:
[10/29/2018 10:52:49 PM] 10/29/2018 3:53:00 PM
[10/29/2018 10:52:49 PM] 10/29/2018 3:53:15 PM
[10/29/2018 10:52:49 PM] 10/29/2018 3:53:30 PM
[10/29/2018 10:52:49 PM] 10/29/2018 3:53:45 PM
[10/29/2018 10:52:49 PM] 10/29/2018 3:54:00 PM
[10/29/2018 10:52:49 PM]
[10/29/2018 10:52:49 PM] Host started (821ms)
[10/29/2018 10:52:49 PM] Job host started
Hosting environment: Production
Content root path: C:\source\repos\SqlConnectionArticle\SqlConnectionArticle\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Listening on http://0.0.0.0:7071/
Hit CTRL-C to exit...

Http Functions:

    Function1: [GET,POST] http://localhost:7071/api/Function1

[10/29/2018 10:52:54 PM] Host lock lease acquired by instance ID '00000000000000000000000000000006C79E40E'.
[10/29/2018 10:53:00 PM] Executing 'DatabaseCleanup' (Reason='Timer fired at 2018-10-29T15:53:00.0271201-07:00', Id=2
67f6418-ddfb-4f5c-a065-5575618ca147)
[10/29/2018 10:53:09 PM] 32 rows were updated
```

On the first execution, you should update 32 rows of data. Following runs update no data rows, unless you make changes to the **SalesOrderHeader** table data so that more rows are selected by the `UPDATE` statement.

If you plan to [publish this function](#), remember to change the `TimerTrigger` attribute to a more reasonable [cron schedule](#) than every 15 seconds. You also need to make sure that your function app can access the Azure SQL Database or Azure SQL Managed Instance. For more information, see one of the following links based on your type of Azure SQL:

- [Azure SQL Database](#)
- [Azure SQL Managed Instance](#)

Next steps

Next, learn how to use Functions with Logic Apps to integrate with other services.

Create a function that integrates with Logic Apps

For more information about Functions, see the following articles:

- [Azure Functions developer reference](#)
Programmer reference for coding functions and defining triggers and bindings.
- [Testing Azure Functions](#)
Describes various tools and techniques for testing your functions.

Tutorial: Integrate Azure Functions with an Azure virtual network by using private endpoints

Article • 03/30/2023

This tutorial shows you how to use Azure Functions to connect to resources in an Azure virtual network by using private endpoints. You create a new function app using a new storage account that's locked behind a virtual network via the Azure portal. The virtual network uses a Service Bus queue trigger.

In this tutorial, you'll:

- ✓ Create a function app in the Elastic Premium plan with virtual network integration and private endpoints.
- ✓ Create Azure resources, such as the Service Bus
- ✓ Lock down your Service Bus behind a private endpoint.
- ✓ Deploy a function app that uses both the Service Bus and HTTP triggers.
- ✓ Test to see that your function app is secure inside the virtual network.
- ✓ Clean up resources.

Create a function app in a Premium plan

You create a C# function app in an [Elastic Premium plan](#), which supports networking capabilities such as virtual network integration on create along with serverless scale. This tutorial uses C# and Windows. Other languages and Linux are also supported.

1. On the Azure portal menu or the [Home](#) page, select [Create a resource](#).
2. On the [New](#) page, select [Compute > Function App](#).
3. On the [Basics](#) page, use the following table to configure the function app settings.

Setting	Suggested value	Description
Subscription	Your subscription	Subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group where you create your function app.
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z (case insensitive), 0-9, and -.

Setting	Suggested value	Description
Publish	Code	Choose to publish code files or a Docker container.
Runtime stack	.NET	This tutorial uses .NET.
Version	6 (LTS)	This tutorial uses .NET 6.0 running in the same process as the Functions host .
Region	Preferred region	Choose a region near you or near other services that your functions access.
Operating system	Windows	This tutorial uses Windows but also works for Linux.
Plan	Functions Premium	<p>Hosting plan that defines how resources are allocated to your function app. By default, when you select Premium, a new App Service plan is created. The default Sku and size is EP1, where <i>EP</i> stands for <i>elastic premium</i>. For more information, see the list of Premium SKUs.</p> <p>When you run JavaScript functions on a Premium plan, choose an instance that has fewer vCPUs. For more information, see Choose single-core Premium plans.</p>

4. Select **Next: Storage**. On the **Storage** page, enter the following settings.

Setting	Suggested value	Description
Storage account	Globally unique name	Create a storage account used by your function app. Storage account names must be between 3 and 24 characters long. They may contain numbers and lowercase letters only. You can also use an existing account that isn't restricted by firewall rules and meets the storage account requirements . When using Functions with a locked down storage account, a v2 storage account is needed. This is the default storage version created when creating a function app with networking capabilities through the create blade.

5. Select **Next: Networking**. On the **Networking** page, enter the following settings.

 **Note**

Some of these settings aren't visible until other options are selected.

Setting	Suggested value	Description
Enable public access	Off	Deny public network access will block all incoming traffic except that comes from private endpoints.
Enable network injection	On	The ability to configure your application with VNet integration at creation appears in the portal window after this option is switched to On .
Virtual Network	Create New	Select the Create New field. In the pop-out screen, provide a name for your virtual network and select Ok . Options to restrict inbound and outbound access to your function app on create are displayed. You must explicitly enable VNet integration in the Outbound access portion of the window to restrict outbound access.

Enter the following settings for the **Inbound access** section. This step creates a private endpoint on your function app.

💡 Tip

To continue interacting with your function app from portal, you'll need to add your local computer to the virtual network. If you don't wish to restrict inbound access, skip this step.

Setting	Suggested value	Description
Enable private endpoints	On	The ability to configure your application with VNet integration at creation appears in the portal after this option is enabled.
Private endpoint name	myInboundPrivateEndpointName	Name that identifies your new function app private endpoint.
Inbound subnet	Create New	This option creates a new subnet for your inbound private endpoint. Multiple private endpoints may be added to a singular subnet. Provide a Subnet Name . The Subnet Address Block may be left at the default value. Select Ok . To learn more about subnet sizing, see Subnets .

Setting	Suggested value	Description
DNS	Azure Private DNS Zone	This value indicates which DNS server your private endpoint uses. In most cases if you're working within Azure, Azure Private DNS Zone is the DNS zone you should use as using Manual for custom DNS zones have increased complexity.

Enter the following settings for the **Outbound access** section. This step integrates your function app with a virtual network on creation. It also exposes options to create private endpoints on your storage account and restrict your storage account from network access on create. When function app is vnet integrated, all outbound traffic by default goes [through the vnet..](#)

Setting	Suggested value	Description
Enable VNet Integration	On	This integrates your function app with a VNet on create and direct all outbound traffic through the VNet.
Outbound subnet	Create new	This creates a new subnet for your function app's VNet integration. A function app can only be VNet integrated with an empty subnet. Provide a Subnet Name . The Subnet Address Block may be left at the default value. If you wish to configure it, please learn more about Subnet sizing here. Select Ok . The option to create Storage private endpoints is displayed. To use your function app with virtual networks, you need to join it to a subnet.

Enter the following settings for the **Storage private endpoint** section. This step creates private endpoints for the blob, queue, file, and table endpoints on your storage account on create. This effectively integrates your storage account with the VNet.

Setting	Suggested value	Description
Add storage private endpoint	On	The ability to configure your application with VNet integration at creation is displayed in the portal after this option is enabled.
Private endpoint name	myInboundPrivateEndpointName	Name that identifies your storage account private endpoint.

Setting	Suggested value	Description
Private endpoint subnet	Create New	This creates a new subnet for your inbound private endpoint on the storage account. Multiple private endpoints may be added to a singular subnet. Provide a Subnet Name . The Subnet Address Block may be left at the default value. If you wish to configure it, please learn more about Subnet sizing here. Select Ok .
DNS	Azure Private DNS Zone	This value indicates which DNS server your private endpoint uses. In most cases if you're working within Azure, Azure Private DNS Zone is the DNS zone you should use as using Manual for custom DNS zones will have increased complexity.

6. Select **Next: Monitoring**. On the **Monitoring** page, enter the following settings.

Setting	Suggested value	Description
Application Insights	Default	Create an Application Insights resource of the same app name in the nearest supported region. Expand this setting if you need to change the New resource name or store your data in a different Location in an Azure geography .

7. Select **Review + create** to review the app configuration selections.

8. On the **Review + create** page, review your settings. Then select **Create** to create and deploy the function app.

9. In the upper-right corner of the portal, select the **Notifications** icon and watch for the **Deployment succeeded** message.

10. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

Congratulations! You've successfully created your premium function app.

Note

Some deployments may occasionally fail to create the private endpoints in the storage account with the error 'StorageAccountOperationInProgress'. This failure occurs even though the function app itself gets created successfully. When you

encounter such an error, delete the function app and retry the operation. You can instead create the private endpoints on the storage account manually.

Create a Service Bus

Next, you create a Service Bus instance that is used to test the functionality of your function app's network capabilities in this tutorial.

1. On the Azure portal menu or the **Home** page, select **Create a resource**.
2. On the **New** page, search for *Service Bus*. Then select **Create**.
3. On the **Basics** tab, use the following table to configure the Service Bus settings. All other settings can use the default values.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription in which your resources are created.
Resource group	myResourceGroup	The resource group you created with your function app.
Namespace name	myServiceBus	The name of the Service Bus instance for which the private endpoint is enabled.
Location	myFunctionRegion	The region where you created your function app.
Pricing tier	Premium	Choose this tier to use private endpoints with Azure Service Bus.

4. Select **Review + create**. After validation finishes, select **Create**.

Lock down your Service Bus

Create the private endpoint to lock down your Service Bus:

1. In your new Service Bus, in the menu on the left, select **Networking**.
2. On the **Private endpoint connections** tab, select **Private endpoint**.

The screenshot shows the Azure portal interface for a Service Bus Namespace named 'myServiceBus-tutorial'. In the left sidebar, under the 'Settings' section, the 'Networking' item is highlighted with a red box. At the top right, the 'Private endpoint connections' tab is also highlighted with a red box. The main content area displays a table with columns for 'Connection name' and 'Connection state', both currently showing 'No results.'

3. On the **Basics** tab, use the private endpoint settings shown in the following table.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription in which your resources are created.
Resource group	myResourceGroup	The resource group you created with your function app.
Name	sb-endpoint	The name of the private endpoint for the service bus.
Region	myFunctionRegion	The region where you created your storage account.

4. On the **Resource** tab, use the private endpoint settings shown in the following table.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Resource type	Microsoft.ServiceBus/namespaces	The resource type for the Service Bus.
Resource	myServiceBus	The Service Bus you created earlier in the tutorial.

Setting	Suggested value	Description
Target subresource	namespace	The private endpoint that is used for the namespace from the Service Bus.

5. On the **Virtual Network** tab, for the **Subnet** setting, choose **default**.
6. Select **Review + create**. After validation finishes, select **Create**.
7. After the private endpoint is created, return to the **Networking** section of your Service Bus namespace and check the **Public Access** tab.
8. Ensure **Selected networks** is selected.
9. Select **+ Add existing virtual network** to add the recently created virtual network.
10. On the **Add networks** tab, use the network settings from the following table:

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Virtual networks	myVirtualNet	The name of the virtual network to which your function app connects.
Subnets	functions	The name of the subnet to which your function app connects.

11. Select **Add your client IP address** to give your current client IP access to the namespace.

 **Note**

Allowing your client IP address is necessary to enable the Azure portal to publish messages to the queue later in this tutorial.

12. Select **Enable** to enable the service endpoint.
13. Select **Add** to add the selected virtual network and subnet to the firewall rules for the Service Bus.
14. Select **Save** to save the updated firewall rules.

Resources in the virtual network can now communicate with the Service Bus using the private endpoint.

Create a queue

Create the queue where your Azure Functions Service Bus trigger gets events:

1. In your Service Bus, in the menu on the left, select **Queues**.
2. Select **Queue**. For the purposes of this tutorial, provide the name *queue* as the name of the new queue.

The screenshot shows the Azure portal interface for creating a new queue in a Service Bus namespace. On the left, the navigation menu is visible with 'Queues' selected under 'Entities'. A red box highlights the 'Queues' link. On the right, a 'Create queue' dialog box is open. The 'Name' field is filled with 'queue', which is also highlighted by a red box. Other configuration options like 'Max queue size', 'Message time to live', and 'Lock duration' are shown, along with several checkboxes for additional settings. A large red box highlights the 'Create' button at the bottom right of the dialog.

3. Select **Create**.

Get a Service Bus connection string

1. In your Service Bus, in the menu on the left, select **Shared access policies**.
2. Select **RootManageSharedAccessKey**. Copy and save the **Primary Connection String**. You need this connection string when you configure the app settings.

Configure your function app settings

1. In your function app, in the menu on the left, select **Configuration**.
2. To use your function app with virtual networks and service bus, update the app settings shown in the following table. To add or edit a setting, select **+ New application setting** or the **Edit** icon in the rightmost column of the app settings table. When you finish, select **Save**.

Setting	Suggested value	Description
SERVICEBUS_CONNECTION	myServiceBusConnectionString	Create this app setting for the connection string of your Service Bus. This storage connection string is from the Get a Service Bus connection string section.
WEBSITE_CONTENTOVERVNET	1	Create this app setting. A value of 1 enables your function app to scale when your storage account is restricted to a virtual network.

3. Since you're using an Elastic Premium hosting plan, In the **Configuration** view, select the **Function runtime settings** tab. Set **Runtime Scale Monitoring** to **On**. Then select **Save**. Runtime-driven scaling allows you to connect non-HTTP trigger functions to services that run inside your virtual network.

The screenshot shows the Azure Function App Configuration page for a 'scaling-sample' function app. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, and Events (preview). The main area has tabs for Application settings, Function runtime settings (which is selected and highlighted in blue), and General settings. Under Function runtime settings, there's a 'Runtime version' dropdown set to '~4'. Below it is a section titled 'Runtime Scale Monitor...' with a radio button for 'On' (which is selected) and a note: 'Enable this setting to allow your app to be scaled based on runtime metrics.'

① Note

Runtime scaling isn't needed for function apps hosted in a Dedicated App Service plan.

Deploy a Service Bus trigger and HTTP trigger

① Note

Enabling private endpoints on a function app also makes the Source Control Manager (SCM) site publicly inaccessible. The following instructions give deployment directions using the Deployment Center within the function app. Alternatively, use **zip deploy** or **self-hosted agents** that are deployed into a subnet on the virtual network.

1. In GitHub, go to the following sample repository. It contains a function app and two functions, an HTTP trigger, and a Service Bus queue trigger.
<https://github.com/Azure-Samples/functions-vnet-tutorial>
2. At the top of the page, select **Fork** to create a fork of this repository in your own GitHub account or organization.
3. In your function app, in the menu on the left, select **Deployment Center**. Then select **Settings**.
4. On the **Settings** tab, use the deployment settings shown in the following table.

Setting	Suggested value	Description
---------	-----------------	-------------

Setting	Suggested value	Description
Source	GitHub	You should have created a GitHub repository for the sample code in step 2.
Organization	myOrganization	The organization your repo is checked into. It's usually your account.
Repository	functions-vnet-tutorial	The repository forked from https://github.com/Azure-Samples/functions-vnet-tutorial .
Branch	main	The main branch of the repository you created.
Runtime stack	.NET	The sample code is in C#.
Version	.NET Core 3.1	The runtime version.

5. Select Save.

vnet-app-tutorial | Deployment Center

Save Discard Browse Manage publish profile Redeploy/Sync Leave Feedback

Logs Settings * FTPS credentials

You're now in the production slot, which is not recommended for setting up CI/CD. Learn more

Deploy and build code from your preferred source and build provider. Learn more

Source* GitHub

Building with GitHub Actions. Change provider.

GitHub

If you can't find an organization or repository, you may need to enable additional permissions on GitHub. Learn more

Signed in as cachai2 Change Account

Organization* cachai2

Repository* functions-vnet-tutorial

Branch* master

Workflow Option* Add a workflow: Add a new workflow file 'master_vnet-app-tutorial.yml' in the selected repository and branch. Use available workflow: Use one of the workflow files available in the selected repository and branch.

Build

Runtime stack* .NET

Version* .NET Core 3.1

Deployment slots Deployment Center (Classic) Deployment Center

Settings Configuration Authentication / Authorization Authentication (preview) Application Insights Identity

6. Your initial deployment might take a few minutes. When your app is successfully deployed, on the Logs tab, you see a Success (Active) status message. If necessary, refresh the page.

Congratulations! You've successfully deployed your sample function app.

Test your locked-down function app

1. In your function app, in the menu on the left, select **Functions**.

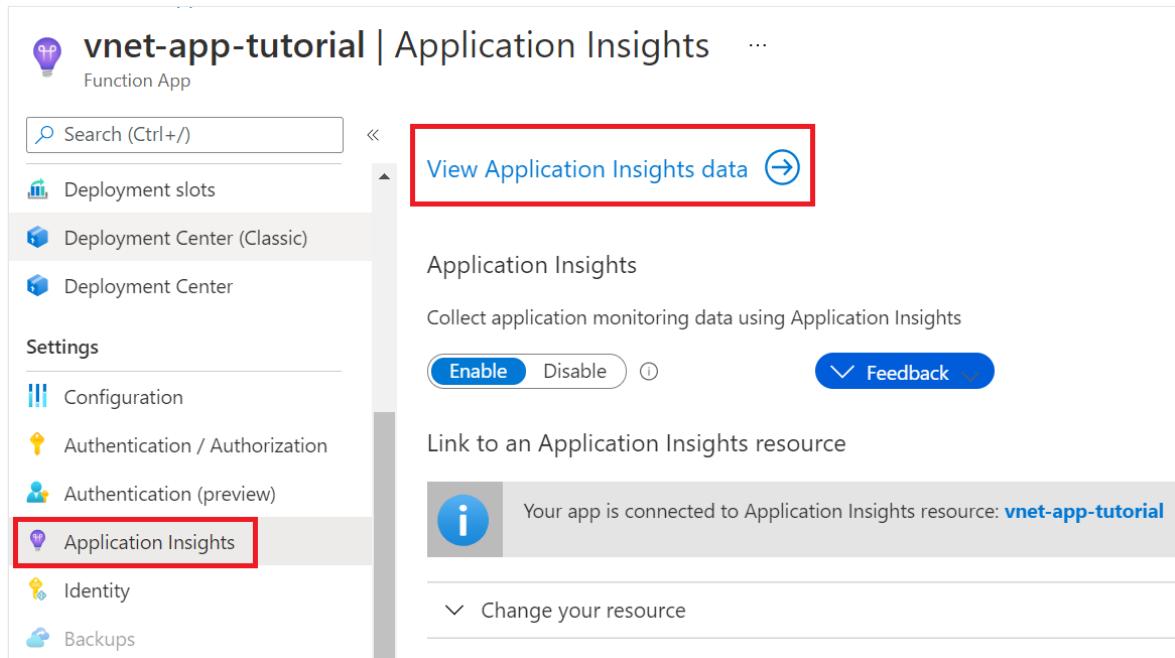
2. Select **ServiceBusQueueTrigger**.

3. In the menu on the left, select **Monitor**.

You see that you can't monitor your app. Your browser doesn't have access to the virtual network, so it can't directly access resources within the virtual network.

Here's an alternative way to monitor your function by using Application Insights:

1. In your function app, in the menu on the left, select **Application Insights**. Then select **View Application Insights data**.



2. In the menu on the left, select **Live metrics**.

3. Open a new tab. In your Service Bus, in the menu on the left, select **Queues**.

4. Select your queue.

5. In the menu on the left, select **Service Bus Explorer**. Under **Send**, for **Content Type**, choose **Text/Plain**. Then enter a message.

6. Select **Send** to send the message.

The screenshot shows the Service Bus Explorer interface for a queue named 'queue' under the namespace 'myServiceBus-tutorial'. The left sidebar contains navigation links for Overview, Access control (IAM), Diagnose and solve problems, Settings (with Shared access policies and Service Bus Explorer (preview) selected), Properties, Locks, Automation (Tasks (preview) and Export template), and Support + troubleshooting (New support request). The main area has tabs for Send, Receive, and Peek, with 'Send' selected. A form titled 'Send Message to Queue **queue**' is displayed, with 'Content Type *' set to 'Text/Plain' and the message body containing 'Hello World'. A red box highlights the message body input field. Below the message body is a checkbox for 'Expand Advanced Properties' and a section for 'Custom Properties' with two empty text fields for Name and Value. At the bottom is a large red-bordered 'Send' button.

- On the Live metrics tab, you should see that your Service Bus queue trigger has fired. If it hasn't, resend the message from Service Bus Explorer.

The screenshot shows the Application Insights Live metrics page for the 'vnet-app-tutorial' application. The left sidebar includes Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Investigate (Application map, Smart Detection, and Live metrics selected), Transaction search, and Availability. The main area displays 'Incoming Requests' and 'Outgoing Requests' metrics. To the right is a 'Sample telemetry' pane showing a list of trace logs. One log entry is highlighted with a red box: '45502 AM | Trace @c221a34ce7ffe37...1462144ac5e60b2 C# ServiceBus queue trigger function processed message: Hello World'. Other logs show trigger details and execution context.

Congratulations! You've successfully tested your function app setup with private endpoints.

Understand private DNS zones

You've used a private endpoint to connect to Azure resources. You're connecting to a private IP address instead of the public endpoint. Existing Azure services are configured to use an existing DNS to connect to the public endpoint. You must override the DNS configuration to connect to the private endpoint.

A private DNS zone is created for each Azure resource that was configured with a private endpoint. A DNS record is created for each private IP address associated with the

private endpoint.

The following DNS zones were created in this tutorial:

- privatelink.file.core.windows.net
- privatelink.blob.core.windows.net
- privatelink.servicebus.windows.net
- privatelink.azurewebsites.net

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select **myResourceGroup**.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

In this tutorial, you created a Premium function app, storage account, and Service Bus. You secured all of these resources behind private endpoints.

Use the following links to learn more Azure Functions networking options and private endpoints:

- [How to configure Azure Functions with a virtual network](#)
- [Networking options in Azure Functions](#)
- [Azure Functions Premium plan](#)
- [Service Bus private endpoints](#)
- [Azure Storage private endpoints](#)

Expose serverless APIs from HTTP endpoints using Azure API Management

Article • 02/02/2022

Azure Functions integrates with Azure API Management in the portal to let you expose your HTTP trigger function endpoints as REST APIs. These APIs are described using an OpenAPI definition. This JSON (or YAML) file contains information about what operations are available in an API. It includes details about how the request and response data for the API should be structured. By integrating your function app, you can have API Management generate these OpenAPI definitions.

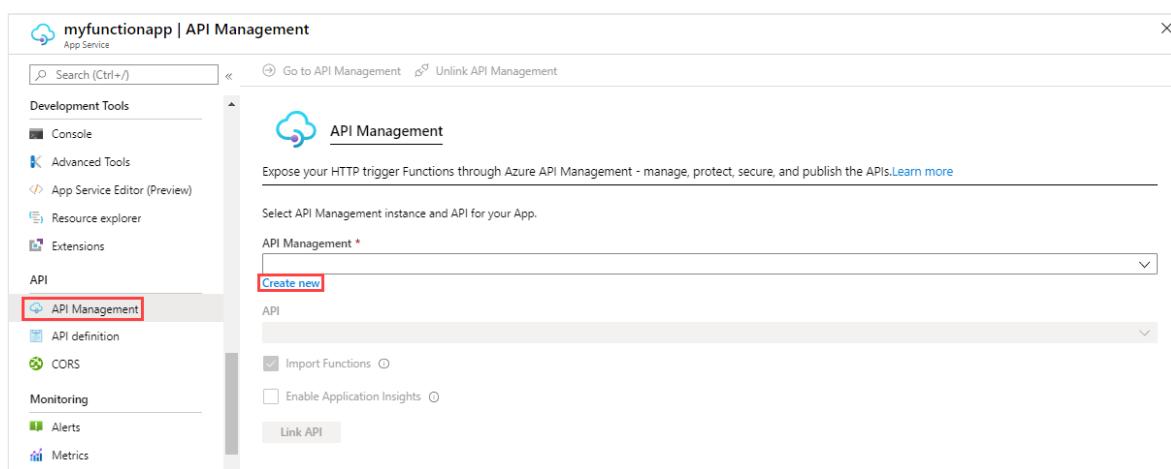
This article shows you how to integrate your function app with API Management. This integration works for function apps developed in any [supported language](#). You can also [import your function app from Azure API Management](#).

For C# class library functions, you can also [use Visual Studio](#) to create and publish serverless API that integrate with API Management.

Create the API Management instance

To create an API Management instance linked to your function app:

1. Select the function app, choose **API Management** from the left menu, and then select **Create new** under **API Management**.



2. Use the API Management settings as specified in the following table:

Setting	Suggested value	Description
Name	Globally unique name	A name is generated based on the name of your function app.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which this new resource is created.
Resource group	myResourceGroup	The same resource as your function app, which should get set for you.
Location	Location of the service	Consider choosing the same location as your function app.
Organization name	Contoso	The name of the organization used in the developer portal and for email notifications.
Administrator email	your email	Email that received system notifications from API Management.
Pricing tier	Consumption	Consumption tier isn't available in all regions. For complete pricing details, see the API Management pricing page

API Management service

Name *



.azure-api.net

Subscription *



Resource group *



[Create new](#)

Location *



Organization name * ⓘ



Administrator email * ⓘ



Pricing tier ([View full pricing details](#))



[Export](#)

3. Choose **Export** to create the API Management instance, which may take several minutes.
4. After Azure creates the instance, it enables the **Enable Application Insights** option on the page. Select it to send logs to the same place as the function application.

Import functions

After the API Management instance is created, you can import your HTTP triggered function endpoints. This example imports an endpoint named `TurbineRepair`.

1. In the API Management page, select **Link API**.
2. The **Import Azure Functions** opens with the `TurbineRepair` function highlighted. Choose **Select** to continue.

The screenshot shows the 'Import Azure Functions' dialog box. At the top, it says 'Import Azure Functions' and 'API Management service'. Below that is a message: 'Don't see an Azure Function? Azure API Management requires Azure Functions to use the HTTP trigger and Function or Anonymous authorization level setting.' A search bar with the placeholder 'Search to filter items...' is below the message. A table lists functions with columns: NAME, HTTP METHODS, and URL TEMPLATE. The function 'TurbineRepair' is listed with a checked checkbox in the NAME column, 'GET, POST' in the HTTP METHODS column, and 'TurbineRepair' in the URL TEMPLATE column. At the bottom of the dialog is a blue 'Select' button.

<input checked="" type="checkbox"/> NAME	HTTP METHODS	URL TEMPLATE
<input checked="" type="checkbox"/> TurbineRepair	GET, POST	TurbineRepair

3. In the **Create from Function App** page, accept the defaults, and then select **Create**.

Create from Function App

Basic | Full

* Function App	myfunctionapp
* Display name	myfunctionapp
* Name	myfunctionapp
API URL suffix	myfunctionapp
Base URL	https://myfunctionapp-apim.azure-api.net/myfunctionapp

Create

Cancel

Azure creates the API for the function.

Download the OpenAPI definition

After your functions have been imported, you can download the OpenAPI definition from the API Management instance.

1. Select Download OpenAPI definition at the top of the page.



Go to API Management



Download OpenAPI definition



Unlink API Management

2. Save the downloaded JSON file, and then open it. Review the definition.

Next steps

You can now refine the definition in API Management in the portal. You can also [learn more about API Management](#).

[Edit the OpenAPI definition in API Management](#)

Create serverless APIs in Visual Studio using Azure Functions and API Management integration

Article • 08/07/2024

REST APIs are often described using an OpenAPI definition (formerly known as Swagger) file. This file contains information about operations in an API and how the request and response data for the API should be structured.

In this tutorial, you learn how to:

- ✓ Create the code project in Visual Studio
- ✓ Install the OpenAPI extension
- ✓ Add an HTTP trigger endpoint, which includes OpenAPI definitions
- ✓ Test function APIs locally using built-in OpenAPI functionality
- ✓ Publish project to a function app in Azure
- ✓ Enable API Management integration
- ✓ Download the OpenAPI definition file

The serverless function you create provides an API that lets you determine whether an emergency repair on a wind turbine is cost-effective. Since you create both the function app and API Management instance in a consumption tier, your cost for completing this tutorial is minimal.

Prerequisites

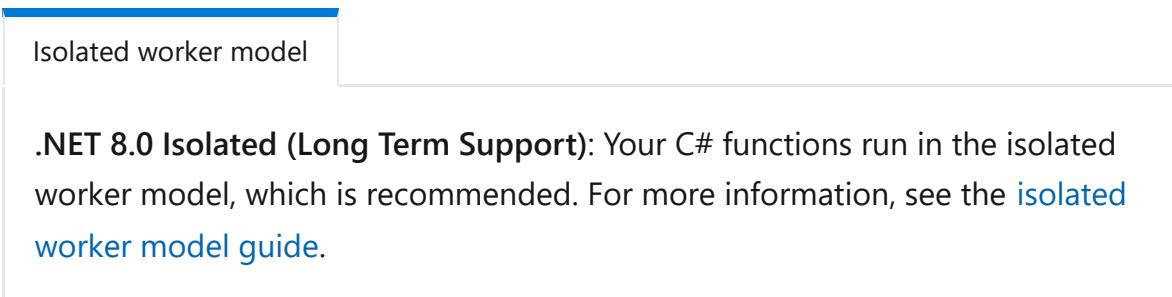
- [Visual Studio 2022](#). Make sure you select the **Azure development** workload during installation.
- An active [Azure subscription](#), [create a free account](#) before you begin.

Create the code project

The Azure Functions project template in Visual Studio creates a project that you can publish to a function app in Azure. You'll also create an HTTP triggered function from a template that supports OpenAPI definition file (formerly Swagger file) generation.

1. From the Visual Studio menu, select **File > New > Project**.

2. In **Create a new project**, enter *functions* in the search box, choose the **Azure Functions** template, and then select **Next**.
3. In **Configure your new project**, enter a **Project name** for your project like `TurbineRepair`, and then select **Create**.
4. For the **Create a new Azure Functions application** settings, select one of these options for **Functions worker**, where the option you choose depends on your chosen process model:



5. For the rest of the options, use the values in the following table:

[] Expand table

Setting	Value	Description
Function template	Empty	This creates a project without a trigger, which gives you more control over the name of the HTTP triggered function when you add it later.
Use Azurite for runtime storage account (AzureWebJobsStorage)	Selected	You can use the emulator for local development of HTTP trigger functions. Because a function app in Azure requires a storage account, one is assigned or created when you publish your project to Azure.
Authorization level	Function	When running in Azure, clients must provide a key when accessing the endpoint. For more information, see Authorization level .

6. Select **Create** to create the function project.

Next, you update the project by installing the OpenAPI extension for Azure Functions, which enables the discoverability of API endpoints in your app.

Install the OpenAPI extension

To install the OpenAPI extension:

1. From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.
2. In the console, run the following [Install-Package](#) command to install the OpenAPI extension:

The screenshot shows a command-line interface for the Azure Functions Package Manager. At the top, it says "Isolated worker model". Below that, under the heading "command", is the command: "NuGet\Install-Package Microsoft.Azure.Functions.Worker.Extensions.OpenApi -Version 1.5.1". A note below the command says: "You might need to update the [specific version](#), based on your version of .NET."

Now, you can add your HTTP endpoint function.

Add an HTTP endpoint function

In a C# class library, the bindings used by the function are defined by applying attributes in the code. To create a function with an HTTP trigger:

1. In **Solution Explorer**, right-click your project node and select **Add > New Azure Function**.
2. Enter **Turbine.cs** for the class, and then select **Add**.
3. Choose the **Http trigger** template, set **Authorization level** to **Function**, and then select **Add**. A Turbine.cs code file is added to your project that defines a new function endpoint with an HTTP trigger.

Now you can replace the HTTP trigger template code with code that implements the Turbine function endpoint, along with attributes that use OpenAPI to define endpoint.

Update the function code

The function uses an HTTP trigger that takes two parameters:

[\[+\] Expand table](#)

Parameter name	Description
<i>hours</i>	The estimated time to make a turbine repair, up to the nearest whole hour.
<i>capacity</i>	The capacity of the turbine, in kilowatts.

The function then calculates how much a repair costs, and how much revenue the turbine could make in a 24-hour period. Parameters are supplied either in the query string or in the payload of a POST request.

In the `Turbine.cs` project file, replace the contents of the class generated from the HTTP trigger template with the following code, which depends on your process model:

Isolated worker model

```
C#
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.WebJobs.Extensions.OpenApi.Core.Attributes;
using Microsoft.Azure.WebJobs.Extensions.OpenApi.Core.Enums;
using Microsoft.Extensions.Logging;
using Microsoft.OpenApi.Models;
using Newtonsoft.Json;
using System.Net;

namespace TurbineRepair
{
    public class Turbine
    {
        const double revenuePerkW = 0.12;
        const double technicianCost = 250;
        const double turbineCost = 100;

        private readonly ILogger<Turbine> _logger;

        public Turbine(ILogger<Turbine> logger)
        {
            _logger = logger;
        }

        [Function("TurbineRepair")]
        [OpenApiOperation(operationId: "Run")]
        [OpenApiSecurity("function_key", SecuritySchemeType.ApiKey, Name = "code", In = OpenApiSecurityLocationType.Query)]
        [OpenApiRequestBody("application/json", typeof(RequestBodyModel),
            Description = "JSON request body containing { hours, capacity}")]
        [OpenApiResponseWithBody(statusCode: HttpStatusCode.OK,
```

```

    contentType: "application/json", bodyType: typeof(string),
        Description = "The OK response message containing a JSON
result.")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "post", Route =
null)] HttpRequest req,
        ILogger log)
    {
        // Get request body data.
        string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
        dynamic? data = JsonConvert.DeserializeObject(requestBody);
        int? capacity = data?.capacity;
        int? hours = data?.hours;

        // Return bad request if capacity or hours are not passed in
        if (capacity == null || hours == null)
        {
            return new BadRequestObjectResult("Please pass capacity
and hours in the request body");
        }
        // Formulas to calculate revenue and cost
        double? revenueOpportunity = capacity * revenuePerkW * 24;
        double? costToFix = hours * technicianCost + turbineCost;
        string repairTurbine;

        if (revenueOpportunity > costToFix)
        {
            repairTurbine = "Yes";
        }
        else
        {
            repairTurbine = "No";
        };

        return new OkObjectResult(new
        {
            message = repairTurbine,
            revenueOpportunity = "$" + revenueOpportunity,
            costToFix = "$" + costToFix
        });
    }
    public class RequestBodyModel
    {
        public int Hours { get; set; }
        public int Capacity { get; set; }
    }
}
}

```

This function code returns a message of `Yes` or `No` to indicate whether an emergency repair is cost-effective. It also returns the revenue opportunity that the turbine

represents and the cost to fix the turbine.

Run and verify the API locally

When you run the function, the OpenAPI endpoints make it easy to try out the function locally using a generated page. You don't need to provide function access keys when running locally.

1. Press F5 to start the project. When Functions runtime starts locally, a set of OpenAPI and Swagger endpoints are shown in the output, along with the function endpoint.
2. In your browser, open the RenderSwaggerUI endpoint, which should look like `http://localhost:7071/api/swagger/ui`. A page is rendered, based on your OpenAPI definitions.
3. Select **POST > Try it out**, enter values for `hours` and `capacity` either as query parameters or in the JSON request body, and select **Execute**.

The screenshot shows the 'Try it out' dialog for a POST request to the '/TurbineRepair' endpoint. The dialog has a green header bar with 'POST' and the endpoint path. Below the header is a 'Parameters' section with a 'Cancel' button. The main body shows a table with 'Name' and 'Description' columns. A row for 'body object (body)' has an 'Edit Value | Model' link and a JSON input field containing '{ "hours": 6, "capacity": 2500 }'. A red box highlights this JSON input field. At the bottom, there's another 'Cancel' button, a 'Parameter content type' dropdown set to 'application/json', and a large blue 'Execute' button highlighted with a red box.

4. When you enter integer values like 6 for `hours` and 2500 for `capacity`, you get a JSON response that looks like the following example:

The screenshot shows a browser developer tools Network tab. The response status is 200 OK. The response body is highlighted with a red box and contains the following JSON object:

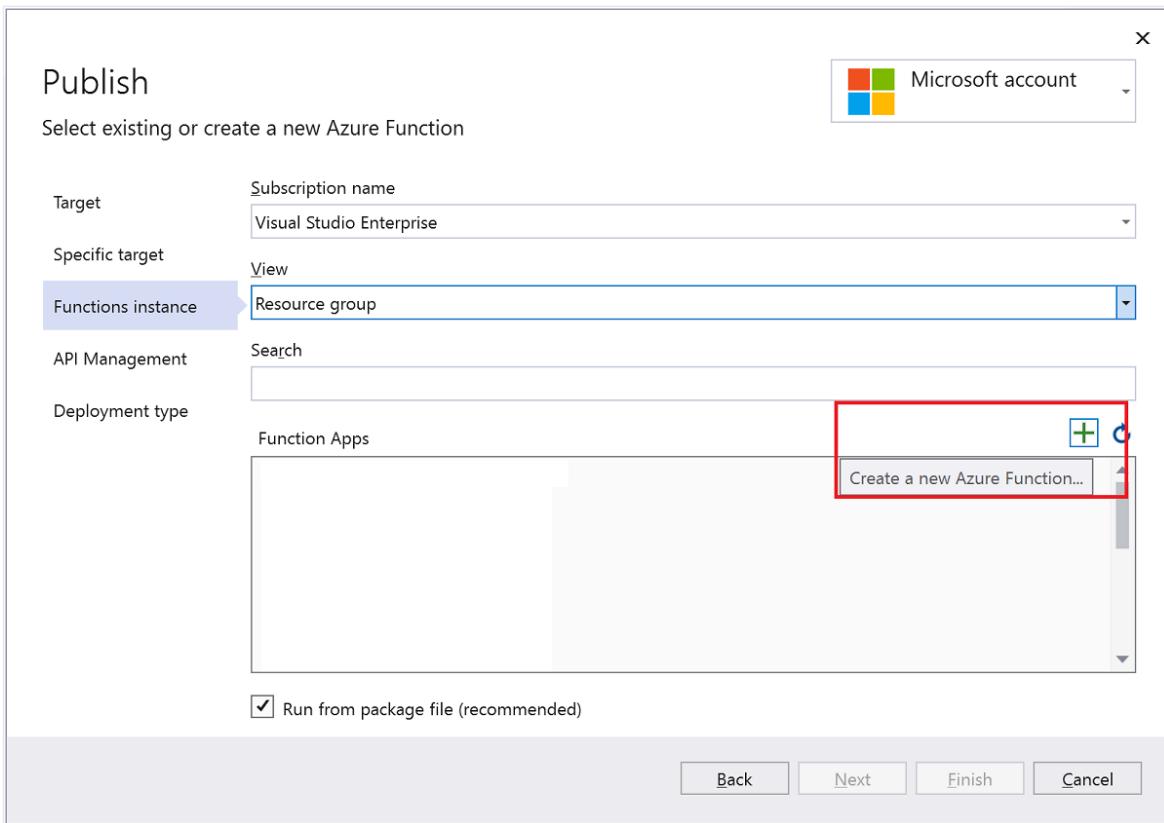
```
{  
  "message": "Yes",  
  "revenueOpportunity": "$7200",  
  "costToFix": "$1600"  
}
```

Now you have a function that determines the cost-effectiveness of emergency repairs. Next, you publish your project and API definitions to Azure.

Publish the project to Azure

Before you can publish your project, you must have a function app in your Azure subscription. Visual Studio publishing creates a function app the first time you publish your project. It can also create an API Management instance that integrates with your function app to expose the TurbineRepair API.

1. In **Solution Explorer**, right-click the project and select **Publish** and in **Target**, select **Azure** then **Next**.
2. For the **Specific target**, choose **Azure Function App (Windows)** to create a function app that runs on Windows, then select **Next**.
3. In **Function Instance**, choose **+ Create a new Azure Function....**



4. Create a new instance using the values specified in the following table:

[\[...\] Expand table](#)

Setting	Value	Description
Name	Globally unique name	Name that uniquely identifies your new function app. Accept this name or enter a new name. Valid characters are: a-z, 0-9, and -.
Subscription	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the drop-down list.
Resource group	Name of your resource group	The resource group in which to create your function app. Select an existing resource group from the drop-down list or choose New to create a new resource group.
Plan Type	Consumption	When you publish your project to a function app that runs in a Consumption plan , you pay only for executions of your functions app. Other hosting plans incur higher costs.
Location	Location of the service	Choose a Location in a region near you or other services your functions access.
Azure Storage	General-purpose storage account	An Azure Storage account is required by the Functions runtime. Select New to configure a general-purpose

Setting	Value	Description
		storage account. You can also choose an existing account that meets the storage account requirements .

Name: TurbineRepair20210506232328

Subscription name: Visual Studio Enterprise

Resource group: TurbineRepairGroup* [New...](#)

Plan Type: Consumption

Location: South Central US

Azure Storage: turbinerepair20210506232* (East US) [New...](#)

Buttons: Export..., Create, Cancel

5. Select **Create** to create a function app and its related resources in Azure. Status of resource creation is shown in the lower left of the window.
6. Back in **Functions instance**, make sure that **Run from package file** is checked. Your function app is deployed using [Zip Deploy](#) with [Run-From-Package](#) mode enabled. This deployment method is recommended for your functions project, since it results in better performance.
7. Select **Next**, and in **API Management** page, also choose **+ Create an API Management API**.
8. Create an **API in API Management** by using values in the following table:

[Expand table](#)

Setting	Value	Description
API name	TurbineRepair	Name for the API.
Subscription name	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the drop-down list.
Resource group	Name of your resource group	Select the same resource group as your function app from the drop-down list.
API Management service	New instance	Select New to create a new API Management instance in the same location in the serverless tier. Select OK to create the instance.

X

 API in API Management Microsoft account

Create new

API name
TurbineRepair

Subscription name
Visual Studio Enterprise

Resource group
TurbineRepairGroup (South Central US) [New...](#)

API Management service
*TurbineRepairApi (South Central US) [New...](#)

API URL suffix

[Export...](#) Create [Cancel](#)

9. Select **Create** to create the API Management instance with the TurbineRepair API from the function integration.

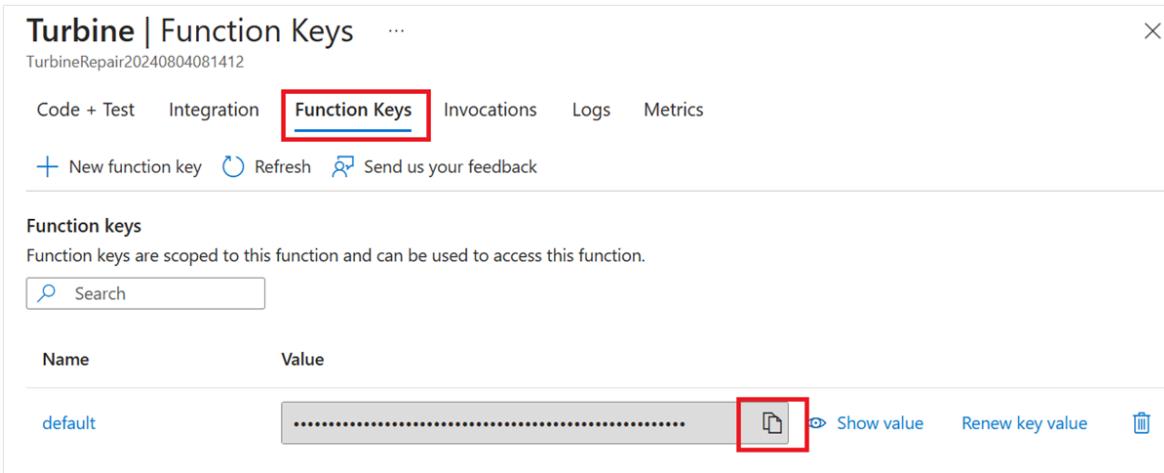
10. Select **Finish** and after the publish profile creation process completes, select **Close**.

11. Verify the Publish page now says **Ready to publish**, and then select **Publish** to deploy the package containing your project files to your new function app in Azure.

After the deployment completes, the root URL of the function app in Azure is shown in the **Publish** tab.

Get the function access key

1. In the **Publish** tab, select the ellipses (...) next to **Hosting** and select **Open in Azure portal**. The function app you created is opened in the Azure portal in your default browser.
2. Under **Functions** on the **Overview page**, select > **Turbine** then select **Function keys**.



The screenshot shows the 'Turbine | Function Keys' blade in the Azure portal. At the top, there's a header with the function name 'TurbineRepair20240804081412'. Below the header, there are several tabs: 'Code + Test', 'Integration', 'Function Keys' (which is highlighted with a red box), 'Invocations', 'Logs', and 'Metrics'. Under the 'Function Keys' tab, there's a 'New function key' button, a 'Refresh' button, and a 'Send us your feedback' link. The main area is titled 'Function keys' and contains a note: 'Function keys are scoped to this function and can be used to access this function.' Below this is a search bar labeled 'Search'. A table lists a single function key: 'Name' (default) and 'Value' (redacted). To the right of the value column are three icons: a copy-to-clipboard icon (highlighted with a red box), a 'Show value' link, a 'Renew key value' link, and a delete icon.

3. Under **Function keys**, select the *copy to clipboard* icon next to the **default** key. You can now set this key you copied in API Management so that it can access the function endpoint.

Configure API Management

1. In the function app page, expand **API** and select **API Management**.
2. If the function app isn't already connected to the new API Management instance, select it under **API Management**, select **API > OpenAPI Document on Azure Functions**, make sure **Import functions** is checked, and select **Link API**. Make sure that only **TurbineRepair** is selected for import and then **Select**.

3. Select **Go to API Management** at the top of the page, and in the API Management instance, expand **APIs**.

4. Under **APIs > All APIs**, select **OpenAPI Document on Azure Functions > POST Run**, then under **Inbound processing** select **Add policy > Set query parameters**.

5. Below **Inbound processing**, in **Set query parameters**, type `code` for **Name**, select **+Value**, paste in the copied function key, and select **Save**. API Management includes the function key when it passes calls through to the function endpoint.

The screenshot shows the 'Inbound processing' configuration for a POST operation. On the left, there's a sidebar with 'Operations' and 'Definitions' tabs. The main area shows the 'Inbound processing' section with a table for 'Set query parameters'. The table has columns: NAME, VALUE, ACTION, and DELETE. One row is present: NAME is 'code', VALUE is partially redacted, ACTION is 'override', and the DELETE button is visible. A red box highlights the 'Save' button at the bottom of the table.

NAME	VALUE	ACTION	DELETE
code	[REDACTED]	override	

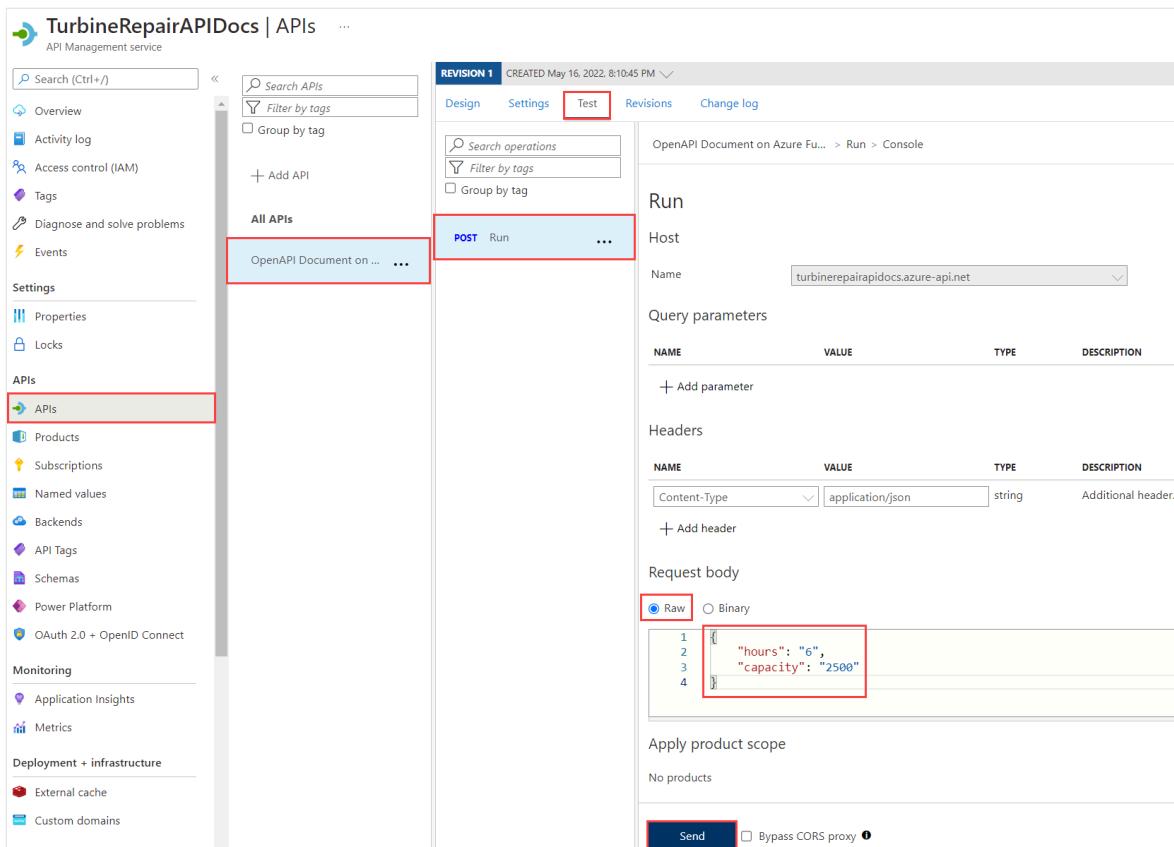
Now that the function key is set, you can call the `turbine` API endpoint to verify that it works when hosted in Azure.

Verify the API in Azure

1. In the API, select the **Test** tab and then **POST Run**, enter the following code in the **Request body > Raw**, and select **Send**:

The screenshot shows the 'Test' tab for a POST operation. The 'Request body > Raw' field contains the following JSON:

```
{  
  "hours": "6",  
  "capacity": "2500"  
}
```



As before, you can also provide the same values as query parameters.

2. Select **Send**, and then view the **HTTP response** to verify the same results are returned from the API.

Download the OpenAPI definition

If your API works as expected, you can download the OpenAPI definition for the new hosted APIs from API Management.

1. a. Under **APIs**, select **OpenAPI Document on Azure Functions**, select the ellipses (...), and select **Export**.

The screenshot shows the Azure API Management service interface. On the left, there's a sidebar with various navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Properties and Locks), APIs (Products, Subscriptions, Named values, Backends, API Tags), and a main APIs section. The 'APIs' link under 'APIs' is highlighted with a red box. In the main content area, there's a search bar for APIs, a 'Filter by tags' button, and a 'Group by tag' checkbox. Below that is a '+ Add API' button and a 'All APIs' section. An API entry is selected, with its details shown: 'OpenAPI Document on ...' and a three-dot ellipsis button. To the right of the API entry is a context menu with several options: Clone (with a copy icon), Add revision (with a pencil icon), Add version (with a list icon), Import (with an upward arrow icon), Export (highlighted with a red box and a downward arrow icon), Create Power Connector (with a diamond icon), and Delete (with a trash bin icon).

2. Choose the means of API export, including OpenAPI files in various formats. You can also [export APIs from Azure API Management to the Power Platform](#).

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select the group you created.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type the name of your group in the text box to confirm, and then select **Delete**.

Next steps

You've used Visual Studio 2022 to create a function that's self-documenting because of the [OpenAPI Extension](#) and integrated with API Management. You can now refine the definition in API Management in the portal. You can also [learn more about API Management](#).

[Edit the OpenAPI definition in API Management](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

How to use managed identities for App Service and Azure Functions

Article • 09/30/2024

ⓘ Note

Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention `<app-name>-<random-hash>. <region>.azurewebsites.net`. Existing app names will remain unchanged.

Example: `myapp-ds27dh7271ah175.westus-01.azurewebsites.net`

For further details, refer to [Unique Default Hostname for App Service Resource ↗](#).

This article shows you how to create a managed identity for App Service and Azure Functions applications and how to use it to access other resources.

ⓘ Important

Because [managed identities don't support cross-directory scenarios](#), they won't behave as expected if your app is migrated across subscriptions or tenants. To recreate the managed identities after such a move, see [Will managed identities be recreated automatically if I move a subscription to another directory?](#).

Downstream resources also need to have access policies updated to use the new identity.

ⓘ Note

Managed identities are not available for [apps deployed in Azure Arc](#).

A managed identity from Microsoft Entra ID allows your app to easily access other Microsoft Entra protected resources such as Azure Key Vault. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. For more about managed identities in Microsoft Entra ID, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to the app and is deleted if the app is deleted.
An app can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your app. An app can have multiple user-assigned identities, and one user-assigned identity can be assigned to multiple Azure resources, such as two App Service apps.

The managed identity configuration is specific to the slot. To configure a managed identity for a deployment slot in the portal, navigate to the slot first. To find the managed identity for your web app or deployment slot in your Microsoft Entra tenant from the Azure portal, search for it directly from the **Overview** page of your tenant. Usually, the slot name is similar to <app-name>/slots/<slot-name>.

This video shows you how to use managed identities for App Service.

[https://learn-video.azurefd.net/vod/player?id=4fdf7a78-b3ce-48df-b3ce-cd7796d0ad5a&locale=en-us&embedUrl=%2Fazure%2Fapp-service%2Foverview-managed-identity ↗](https://learn-video.azurefd.net/vod/player?id=4fdf7a78-b3ce-48df-b3ce-cd7796d0ad5a&locale=en-us&embedUrl=%2Fazure%2Fapp-service%2Foverview-managed-identity)

The steps in the video are also described in the following sections.

Add a system-assigned identity

Azure portal

1. Access your app's settings in the [Azure portal](#) ↗ under the **Settings** group in the left navigation pane.
2. Select **Identity**.
3. Within the **System assigned** tab, switch **Status** to **On**. Click **Save**.

The screenshot shows the Azure portal interface for managing a resource's identity. On the left, there's a navigation menu with options like Deployment, Settings, and Identity. The 'Identity' option is highlighted with a red box. The main content area shows the 'System assigned' tab selected under 'User assigned'. It includes a note about managed identities and a 'Save' button, which is also highlighted with a red box.

Add a user-assigned identity

Creating an app with a user-assigned identity requires that you create the identity and then add its resource identifier to your app config.

The screenshot shows the Azure portal interface for managing a resource's identity. The 'Identity' section is displayed, with the 'User assigned' tab selected. A modal window titled 'Add user assigned managed i...' is open, showing a list of identities. One identity, 'test', is selected and highlighted with a red box. An 'Add' button is also highlighted with a red box.

First, you'll need to create a user-assigned identity resource.

1. Create a user-assigned managed identity resource according to [these instructions](#).
2. In the left navigation for your app's page, scroll down to the **Settings** group.
3. Select **Identity**.
4. Select **User assigned > Add**.
5. Search for the identity you created earlier, select it, and select **Add**.

Once you select **Add**, the app restarts.

Configure target resource

You may need to configure the target resource to allow access from your app or function. For example, if you [request a token](#) to access Key Vault, you must also add an access policy that includes the managed identity of your app or function. Otherwise, your calls to Key Vault will be rejected, even if you use a valid token. The same is true for Azure SQL Database. To learn more about which resources support Microsoft Entra tokens, see [Azure services that support Microsoft Entra authentication](#).

Important

The back-end services for managed identities maintain a cache per resource URI for around 24 hours. If you update the access policy of a particular target resource and immediately retrieve a token for that resource, you may continue to get a cached token with outdated permissions until that token expires. There's currently no way to force a token refresh.

Connect to Azure services in app code

With its managed identity, an app can obtain tokens for Azure resources that are protected by Microsoft Entra ID, such as Azure SQL Database, Azure Key Vault, and Azure Storage. These tokens represent the application accessing the resource, and not any specific user of the application.

App Service and Azure Functions provide an internally accessible [REST endpoint](#) for token retrieval. The REST endpoint can be accessed from within the app with a standard HTTP GET, which can be implemented with a generic HTTP client in every language. For .NET, JavaScript, Java, and Python, the Azure Identity client library provides an abstraction over this REST endpoint and simplifies the development experience. Connecting to other Azure services is as simple as adding a credential object to the service-specific client.

HTTP GET

A raw HTTP GET request looks like the following example:

HTTP

```
GET /MSI/token?resource=https://vault.azure.net&api-version=2019-08-01
HTTP/1.1
Host: localhost:4141
X-IDENTITY-HEADER: 853b9a84-5bfa-4b22-a3f3-0b9a43d9ad8a
```

And a sample response might look like the following:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "access_token": "eyJ0eXAi...",
    "expires_on": "1586984735",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer",
    "client_id": "00001111-aaaa-2222-bbbb-3333cccc4444"
}
```

This response is the same as the [response for the Microsoft Entra service-to-service access token request](#). To access Key Vault, you will then add the value of `access_token` to a client connection with the vault.

For more information on the REST endpoint, see [REST endpoint reference](#).

Remove an identity

When you remove a system-assigned identity, it's deleted from Microsoft Entra ID. System-assigned identities are also automatically removed from Microsoft Entra ID when you delete the app resource itself.

Azure portal

1. In the left navigation of your app's page, scroll down to the **Settings** group.

2. Select **Identity**. Then follow the steps based on the identity type:

- **System-assigned identity:** Within the **System assigned** tab, switch **Status** to **Off**. Click **Save**.
- **User-assigned identity:** Select the **User assigned** tab, select the checkbox for the identity, and select **Remove**. Select **Yes** to confirm.

Note

There is also an application setting that can be set, WEBSITE_DISABLE_MSI, which just disables the local token service. However, it leaves the identity in place, and tooling will still show the managed identity as "on" or "enabled." As a result, use of this setting is not recommended.

REST endpoint reference

An app with a managed identity makes this endpoint available by defining two environment variables:

- **IDENTITY_ENDPOINT** - the URL to the local token service.
- **IDENTITY_HEADER** - a header used to help mitigate server-side request forgery (SSRF) attacks. The value is rotated by the platform.

The **IDENTITY_ENDPOINT** is a local URL from which your app can request tokens. To get a token for a resource, make an HTTP GET request to this endpoint, including the following parameters:

 Expand table

Parameter	In	Description
name		
resource	Query	The Microsoft Entra resource URI of the resource for which a token should be obtained. This could be one of the Azure services that support Microsoft Entra authentication or any other resource URI.
api-version	Query	The version of the token API to be used. Use <code>2019-08-01</code> .
X-IDENTITY-HEADER	Header	The value of the IDENTITY_HEADER environment variable. This header is used to help mitigate server-side request forgery (SSRF) attacks.
client_id	Query	(Optional) The client ID of the user-assigned identity to be used. Cannot be used on a request that includes <code>principal_id</code> , <code>mi_res_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.
principal_id	Query	(Optional) The principal ID of the user-assigned identity to be used. <code>object_id</code> is an alias that may be used instead. Cannot be used on a request that includes <code>client_id</code> , <code>mi_res_id</code> , or <code>object_id</code> . If all ID

Parameter	In	Description
name		parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.
mi_res_id	Query	(Optional) The Azure resource ID of the user-assigned identity to be used. Cannot be used on a request that includes <code>principal_id</code> , <code>client_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.

ⓘ Important

If you are attempting to obtain tokens for user-assigned identities, you must include one of the optional properties. Otherwise the token service will attempt to obtain a token for a system-assigned identity, which may or may not exist.

Next steps

- [Tutorial: Connect to SQL Database from App Service without secrets using a managed identity](#)
- [Access Azure Storage securely using a managed identity](#)
- [Call Microsoft Graph securely using a managed identity](#)
- [Connect securely to services with Key Vault secrets](#)

ⓘ Note: The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Customize an HTTP endpoint in Azure Functions

Article • 08/07/2024

In this article, you learn how to build highly scalable APIs with Azure Functions by customizing an HTTP trigger to handle specific actions in your API design. Azure Functions includes a collection of built-in HTTP triggers and bindings, which make it easy to author an endpoint in various languages, including Node.js, C#, and more. You also prepare to grow your API by integrating it with Azure Functions proxies and setting up mock APIs. Because these tasks are accomplished on top of the Functions serverless compute environment, you don't need to be concerned about scaling resources. Instead, you can just focus on your API logic.

Important

Azure Functions proxies is a legacy feature for [versions 1.x through 3.x](#) of the Azure Functions runtime. Support for proxies can be re-enabled in version 4.x for you to successfully upgrade your function apps to the latest runtime version. As soon as possible, you should switch to integrating your function apps with Azure API Management. API Management lets you take advantage of a more complete set of features for defining, securing, managing, and monetizing your Functions-based APIs. For more information, see [API Management integration](#).

To learn how to re-enable proxies support in Functions version 4.x, see [Re-enable proxies in Functions v4.x](#).

Prerequisites

This article uses as its starting point the resources created in [Create your first function in the Azure portal](#). If you haven't already done so, complete these steps now to create your function app.

After you create this function app, you can follow the procedures in this article.

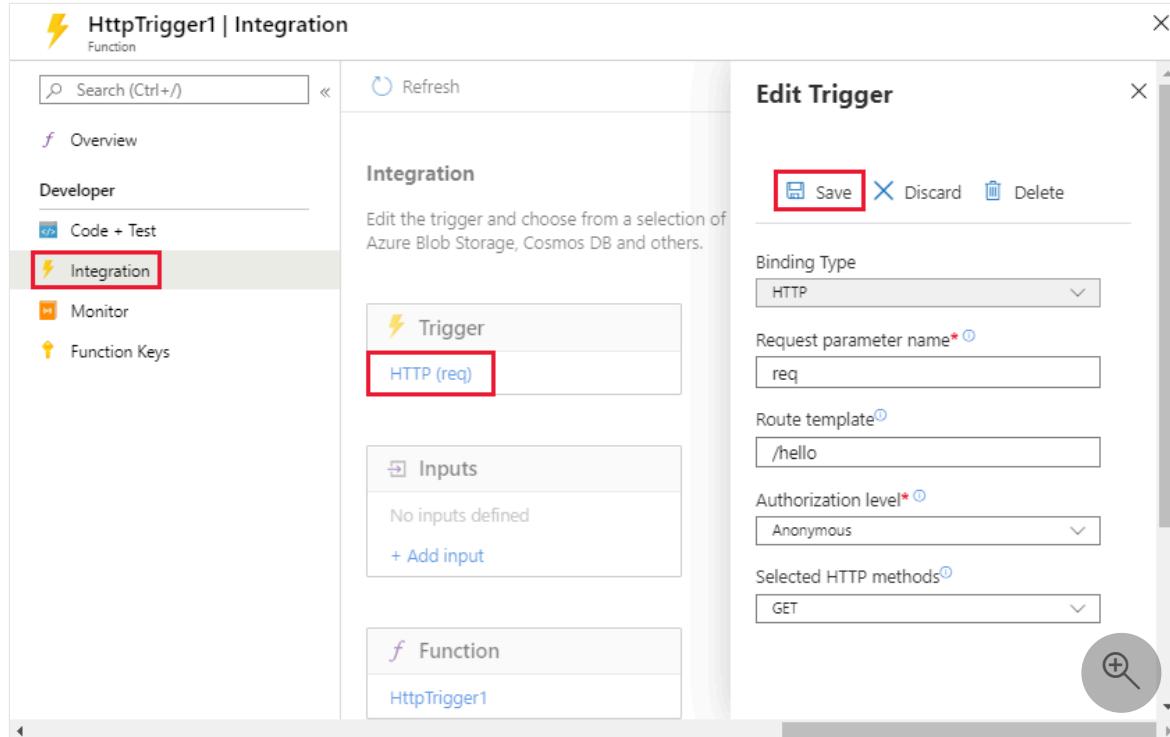
Sign in to Azure

Sign in to the [Azure portal](#)  with your Azure account.

Customize your HTTP function

By default, you configure your HTTP trigger function to accept any HTTP method. In this section, you modify the function to respond only to GET requests with `/api/hello`. You can use the default URL, `https://<yourapp>.azurewebsites.net/api/<funcname>?code=<functionkey>`:

1. Navigate to your function in the Azure portal. Select **Integration** in the left menu, and then select **HTTP (req)** under **Trigger**.



2. Use the HTTP trigger settings as specified in the following table.

[Expand table](#)

Field	Sample value	Description
Route template	hello	Determines what route is used to invoke this function
Authorization level	Anonymous	Optional: Makes your function accessible without an API key
Selected HTTP methods	GET	Allows only selected HTTP methods to be used to invoke this function

Because a global setting handles the `/api` base path prefix in the route template, you don't need to set it here.

3. Select Save.

For more information about customizing HTTP functions, see [Azure Functions HTTP triggers and bindings overview](#).

Test your API

Next, test your function to see how it works with the new API surface:

1. On the **Function** page, select **Code + Test** from the left menu.
2. Select **Get function URL** from the top menu and copy the URL. Confirm that your function now uses the `/api/hello` path.
3. Copy the URL to a new browser tab or your preferred REST client. Browsers use GET by default.
4. Add parameters to the query string in your URL. For example, `/api/hello/?name=John`.
5. Press Enter to confirm that your function is working. You should see the response, "Hello John."
6. You can also call the endpoint with another HTTP method to confirm that the function isn't executed. To do so, use one of these HTTP test tools:
 - [Visual Studio Code](#) with an extension from [Visual Studio Marketplace](#)
 - [PowerShell Invoke-RestMethod](#)
 - [Microsoft Edge - Network Console tool](#)
 - [Bruno](#)
 - [curl](#)

⊗ Caution

For scenarios where you have sensitive data, such as credentials, secrets, access tokens, API keys, and other similar information, make sure to use a tool that protects your data with the necessary security features, works offline or locally, doesn't sync your data to the cloud, and doesn't require that you sign in to an online account. This way, you reduce the risk around exposing sensitive data to the public.

Proxies overview

In the next section, you surface your API through a proxy. Azure Functions proxies allow you to forward requests to other resources. You define an HTTP endpoint as you would with an HTTP trigger. However, instead of writing code to execute when that endpoint is called, you provide a URL to a remote implementation. Doing so allows you to compose multiple API sources into a single API surface, which is easier for clients to consume, and is useful if you wish to build your API as microservices.

A proxy can point to any HTTP resource, such as:

- Azure Functions
- API apps in [Azure App Service](#)
- Docker containers in [App Service on Linux](#)
- Any other hosted API

To learn more about Azure Functions proxies, see [Work with legacy proxies].

 **Note**

Azure Functions proxies is available in Azure Functions versions 1.x to 3.x.

Create your first proxy

In this section, you create a new proxy, which serves as a frontend to your overall API.

Set up the frontend environment

Repeat the steps in [Create a function app](#) to create a new function app in which you create your proxy. This new app's URL serves as the frontend for our API, and the function app you previously edited serves as a backend:

1. Navigate to your new frontend function app in the portal.
2. Expand **Settings**, and then select **Environment variables**.
3. Select the **App settings** tab, where key/value pairs are stored.
4. Select **+ Add** to create a new setting. Enter **HELLO_HOST** for its **Name** and set its **Value** to the host of your backend function app, such as
`<YourBackendApp>.azurewebsites.net`.

This value is part of the URL that you copied earlier when you tested your HTTP function. You later reference this setting in the configuration.

 **Note**

It's recommended that you use app settings for the host configuration to prevent a hard-coded environment dependency for the proxy. Using app settings means that you can move the proxy configuration between environments, and the environment-specific app settings will be applied.

5. Select **Apply** to save the new setting. On the **App settings** tab, select **Apply**, and then select **Confirm** to restart the function app.

Create a proxy on the frontend

1. Navigate back to your front-end function app in the portal.
2. In the left-hand menu, expand **Functions**, select **Proxies**, and then select **Add**.
3. On the **New proxy** page, use the settings in the following table, and then select **Create**.

 [Expand table](#)

Field	Sample value	Description
Name	HelloProxy	A friendly name used only for management
Route template	/api/remotehello	Determines what route is used to invoke this proxy
Backend URL	https://%HELLO_HOST%/api/hello	Specifies the endpoint to which the request should be proxied

New proxy

Name
HelloProxy

Route template
/api/remotehello

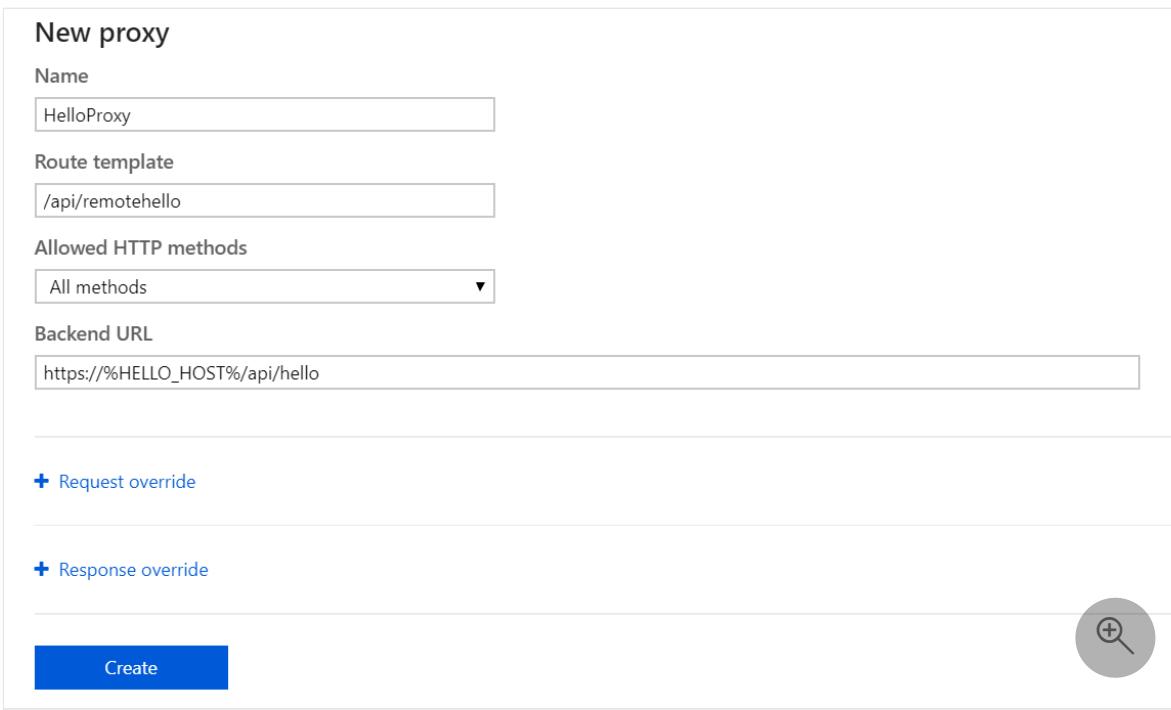
Allowed HTTP methods
All methods

Backend URL
`https://%HELLO_HOST%/api/hello`

+ Request override

+ Response override

Create 



Because Azure Functions proxies don't provide the `/api` base path prefix, you must include it in the route template. The `%HELLO_HOST%` syntax references the app setting you created earlier. The resolved URL points to your original function.

4. Try out your new proxy by copying the proxy URL and testing it in the browser or with your favorite HTTP client:

- For an anonymous function, use:

```
https://YOURPROXYAPP.azurewebsites.net/api/remotehello?name="Proxies".
```

- For a function with authorization, use:

```
https://YOURPROXYAPP.azurewebsites.net/api/remotehello?  
code=YOURCODE&name="Proxies".
```

Create a mock API

Next, you use a proxy to create a mock API for your solution. This proxy allows client development to progress, without needing to fully implement the backend. Later in development, you can create a new function app that supports this logic, and redirect your proxy to it:

1. To create this mock API, create a new proxy, this time using [App Service Editor](#). To get started, navigate to your function app in the Azure portal. Select **Platform features**, and then select **App Service Editor** under **Development Tools**.

The App Service Editor opens in a new tab.

2. Select `proxies.json` in the left pane. This file stores the configuration for all of your proxies. If you use one of the [Functions deployment methods](#), you maintain this file in source control. For more information about this file, see [Proxies advanced configuration](#).

Your `proxies.json` file should appear as follows:

JSON

```
{  
  "$schema": "http://json.schemastore.org/proxies",  
  "proxies": {  
    "HelloProxy": {  
      "matchCondition": {  
        "route": "/api/remotehello"  
      },  
      "backendUri": "https://%HELLO_HOST%/api/hello"  
    }  
  }  
}
```

3. Add your mock API. Replace your `proxies.json` file with the following code:

JSON

```
{  
  "$schema": "http://json.schemastore.org/proxies",  
  "proxies": {  
    "HelloProxy": {  
      "matchCondition": {  
        "route": "/api/remotehello"  
      },  
      "backendUri": "https://%HELLO_HOST%/api/hello"  
    },  
    "GetUserByName" : {  
      "matchCondition": {  
        "methods": [ "GET" ],  
        "route": "/api/users/{username}"  
      },  
      "responseOverrides": {  
        "response.statusCode": "200",  
        "response.headers.Content-Type" : "application/json",  
        "response.body": {  
          "name": "{username}",  
          "description": "Awesome developer and master of  
serverless APIs",  
          "skills": [  
            "Serverless",  
            "APIs",  
            "Azure",  
            "Cloud"  
          ]  
        }  
      }  
    }  
  }  
}
```

```
        ]
    }
}
}
```

This code adds a new proxy, `GetUserByName`, which omits the `backendUri` property. Instead of calling another resource, it modifies the default response from Azure Functions proxies by using a response override. You can also use request and response overrides with a backend URL. This technique is useful when you proxy to a legacy system, where you might need to modify headers, query parameters, and so on. For more information about request and response overrides, see [Modify requests and responses](#).

4. Test your mock API by calling the

`<YourProxyApp>.azurewebsites.net/api/users/{username}` endpoint with a browser or your favorite REST client. Replace `{username}` with a string value that represents a username.

Related content

In this article, you learned how to build and customize an API with Azure Functions. You also learned how to bring multiple APIs, including mock APIs, together as a unified API surface. You can use these techniques to build out APIs of any complexity, all while running on the serverless compute model provided by Azure Functions.

For more information about developing your API:

- [Azure Functions HTTP triggers and bindings overview](#)
- [Work with legacy proxies](#)
- [Expose serverless APIs from HTTP endpoints using Azure API Management](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Managing hybrid environments with PowerShell in Azure Functions and App Service Hybrid Connections

Article • 03/03/2023

The Azure App Service Hybrid Connections feature enables access to resources in other networks. You can learn more about this capability in the [Hybrid Connections](#) documentation. This article describes how to use this capability to run PowerShell functions that target an on-premises server. This server can then be used to manage all resources in the on-premises environment from an Azure PowerShell function.

Configure an on-premises server for PowerShell remoting

The following script enables PowerShell remoting, and it creates a new firewall rule and a WinRM https listener. For testing purposes, a self-signed certificate is used. In a production environment, we recommend that you use a signed certificate.

PowerShell

```
# For configuration of WinRM, see
# https://learn.microsoft.com/windows/win32/winrm/installation-and-
# configuration-for-windows-remote-management.

# Enable PowerShell remoting.
Enable-PSRemoting -Force

# Create firewall rule for WinRM. The default HTTPS port is 5986.
New-NetFirewallRule -Name "WinRM HTTPS"
    -DisplayName "WinRM HTTPS"
    -Enabled True
    -Profile "Any"
    -Action "Allow"
    -Direction "Inbound"
    -LocalPort 5986
    -Protocol "TCP"

# Create new self-signed-certificate to be used by WinRM.
$Thumbprint = (New-SelfSignedCertificate -DnsName $env:COMPUTERNAME -CertStoreLocation Cert:\LocalMachine\My).Thumbprint

# Create WinRM HTTPS listener.
$Cmd = "winrm create winrm/config/Listener?Address=*+Transport=HTTPS"
```

```
@{Hostname=""$env:COMPUTERNAME ""; CertificateThumbprint=""$Thumbprint""}  
cmd.exe /C $Cmd
```

Create a PowerShell function app in the portal

The App Service Hybrid Connections feature is available only in Basic, Standard, and Isolated pricing plans. When you create the function app with PowerShell, create or select one of these plans.

1. From the Azure portal menu or the **Home** page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. On the **Basics** page, use the function app settings as specified in the following table.

Setting	Suggested value	Description
Subscription	Your subscription	The subscription under which this new function app is created.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which to create your function app.
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z (case insensitive), 0-9, and -.
Publish	Code	Option to publish code files or a Docker container.
Runtime stack	Preferred language	Choose PowerShell Core.
Version	Version number	Choose the version of your installed runtime.
Region	Preferred region	Choose a region near you or near other services your functions access.

Home > New > Function App

Function App

[Basics](#) [Hosting](#) [Monitoring](#) [Tags](#) [Review + create](#)

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ APEX C+L - Aquent Vendor Subscriptions

Resource Group * ⓘ myResourceGroup-dma [Create new](#)

Instance Details

Function App name * myPowerShellFunctionApp-dma .azurewebsites.net

Publish * [Code](#) [Docker Container](#)

Runtime stack * Powershell Core

Version * 6

Region * Central US

[Review + create](#) [< Previous](#) [Next : Hosting >](#)

4. Select **Next : Hosting**. On the Hosting page, enter the following settings.

Setting	Suggested value	Description
Storage account	Globally unique name	Create a storage account used by your function app. Storage account names must be between 3 and 24 characters in length and can contain numbers and lowercase letters only. You can also use an existing account, which must meet the storage account requirements .
Operating system	Preferred operating system	An operating system is pre-selected for you based on your runtime stack selection, but you can change the setting if necessary.
Plan type	App service plan	Choose App service plan . When you run in an App Service plan, you must manage the scaling of your function app .

Home > New > Function App

Function App

Basics **Hosting** Monitoring Tags Review + create

Storage

When creating a function app, you must create or link to a general-purpose Azure Storage account that supports Blobs, Queue, and Table storage.

Storage account *

Operating system

Windows is the only supported Operating System for your selection of runtime stack.

Operating System * Windows Linux

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type *

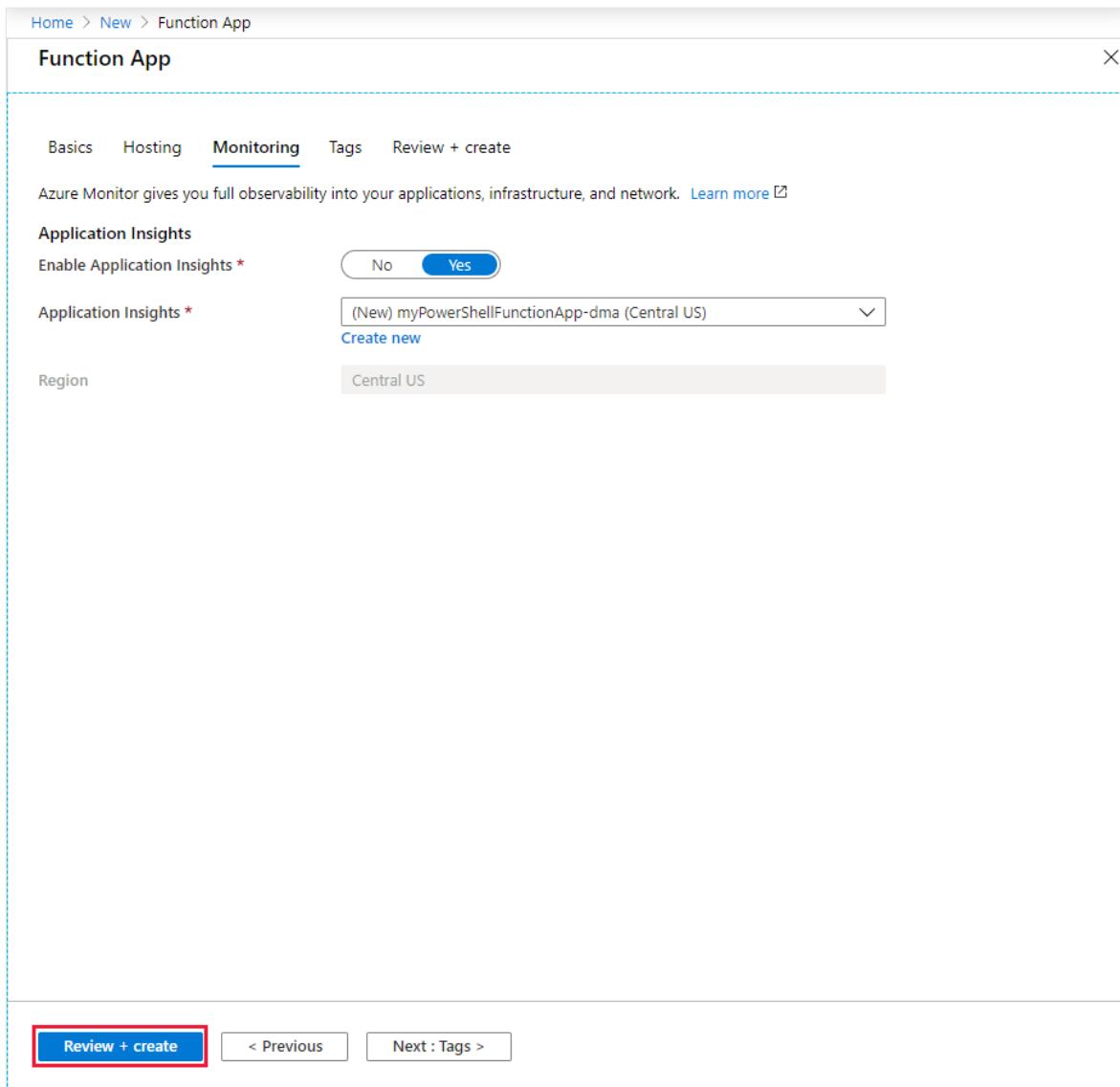
Windows Plan (Central US) *

Sku and size *
100 total ACU, 1.75 GB memory

[Review + create](#) [< Previous](#) [Next : Monitoring >](#)

5. Select **Next : Monitoring**. On the **Monitoring** page, enter the following settings.

Setting	Suggested value	Description
Application Insights	Default	Creates an Application Insights resource of the same <i>App name</i> in the nearest supported region. By expanding this setting or selecting Create new , you can change the Application Insights name or choose a different region in an Azure geography where you want to store your data.



6. Select **Review + create** to review the app configuration selections.
7. On the **Review + create** page, review your settings, and then select **Create** to provision and deploy the function app.
8. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.
9. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

Create a hybrid connection for the function app

Hybrid connections are configured from the networking section of the function app:

1. Under **Settings** in the function app you just created, select **Networking**.

2. Select Configure your hybrid connections endpoints.

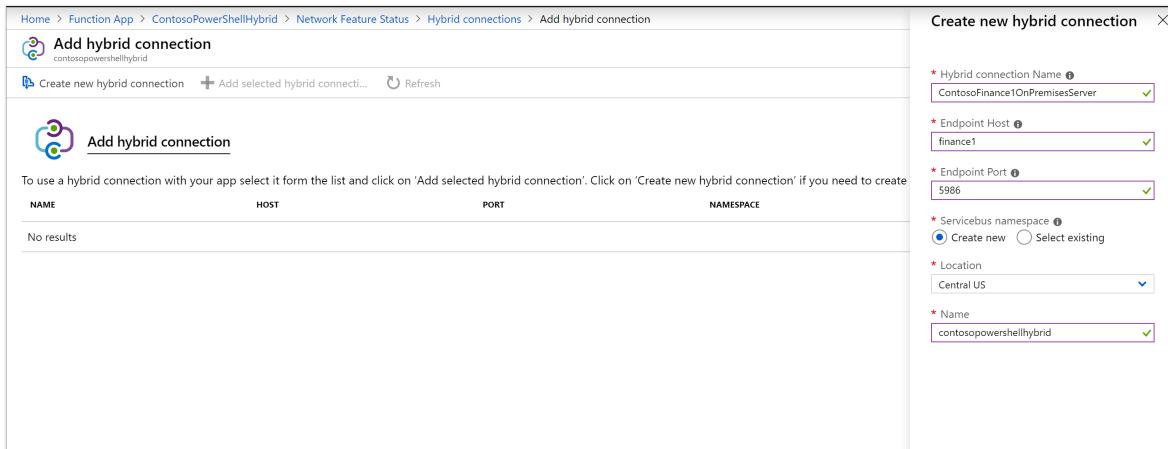
The screenshot shows the Azure portal interface for an App Service named 'myPowerShellFunctionApp-dma'. The left sidebar has a 'Networking' section highlighted with a red box. Within the main content area, under the 'Hybrid connections' heading, there is a link 'Configure your hybrid connection endpoints' which is also highlighted with a red box.

3. Select Add hybrid connection.

The screenshot shows the 'Hybrid connections' blade for a function app named 'ContosoPowerShellHybrid'. At the bottom left, there is a button labeled '+ Add hybrid connection' which is highlighted with a red box.

4. Enter information about the hybrid connection as shown after the following screenshot. For **Endpoint Host**, use the host name of the on-premises server for which you created the self-signed certificate. You'll have connection issues when

the certificate name and the host name of the on-premise server don't match. The port matches the default Windows remote management service port that was defined on the server earlier.



The screenshot shows two side-by-side windows. On the left is a list titled 'Add hybrid connection' under 'contosopowershellhybrid'. It has columns for NAME, HOST, PORT, and NAMESPACE, with a note below stating 'No results'. On the right is a 'Create new hybrid connection' dialog with the following fields filled in:

- * Hybrid connection Name: ContosoFinance1OnPremisesServer
- * Endpoint Host: finance1
- * Endpoint Port: 5986
- * Servicebus namespace: Create New
- * Location: Central US
- * Name: contosopowershellhybrid

Setting	Suggested value
Hybrid connection name	ContosoHybridOnPremisesServer
Endpoint Host	finance1
Endpoint Port	5986
Servicebus namespace	Create New
Location	Pick an available location
Name	contosopowershellhybrid

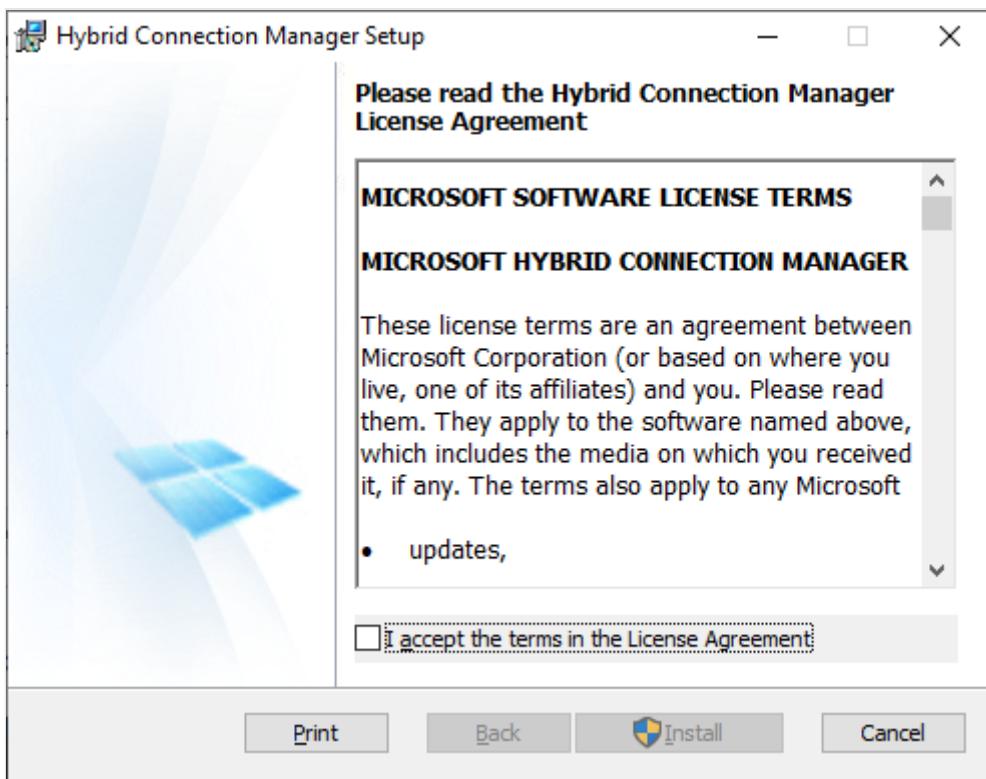
5. Select OK to create the hybrid connection.

Download and install the hybrid connection

1. Select Download connection manager to save the .msi file locally on your computer.

The screenshot shows the Azure portal interface for managing hybrid connections. At the top, the URL is Home > Function App > ContosoPowerShellHybrid > Network Feature Status > Hybrid connections. The page title is "Hybrid connections" with the subtext "contosopowershellhybrid". There is a "Refresh" button and a "Close" button (X). Below the title, there is a section titled "Hybrid connections" with a circular icon. It displays "Connections used: 1" and "Connections quota: 25". A note states: "App Service integration with hybrid connections enables your app to access a single TCP endpoint per hybrid connection. Here you can manage the new and classic hybrid connections used by your app." A link "Learn more" is provided. Below this, it says "App service plan (pricing tier): ServicePlanb818da2f-b9c1 (Standard)". The location is listed as "Central US". A red box highlights the "Download connection manager" button. A blue dashed box highlights the "Add hybrid connection" button. A table lists one connection: "ContosoFinance1OnPremisesServer" with status "Not connected", endpoint "finance1 : 5986", and namespace "contosopowershellhybrid". An ellipsis (...) button is also present.

2. Copy the *.msi* file from your local computer to the on-premises server.
3. Run the Hybrid Connection Manager installer to install the service on the on-premises server.



4. From the portal, open the hybrid connection and then copy the gateway connection string to the clipboard.

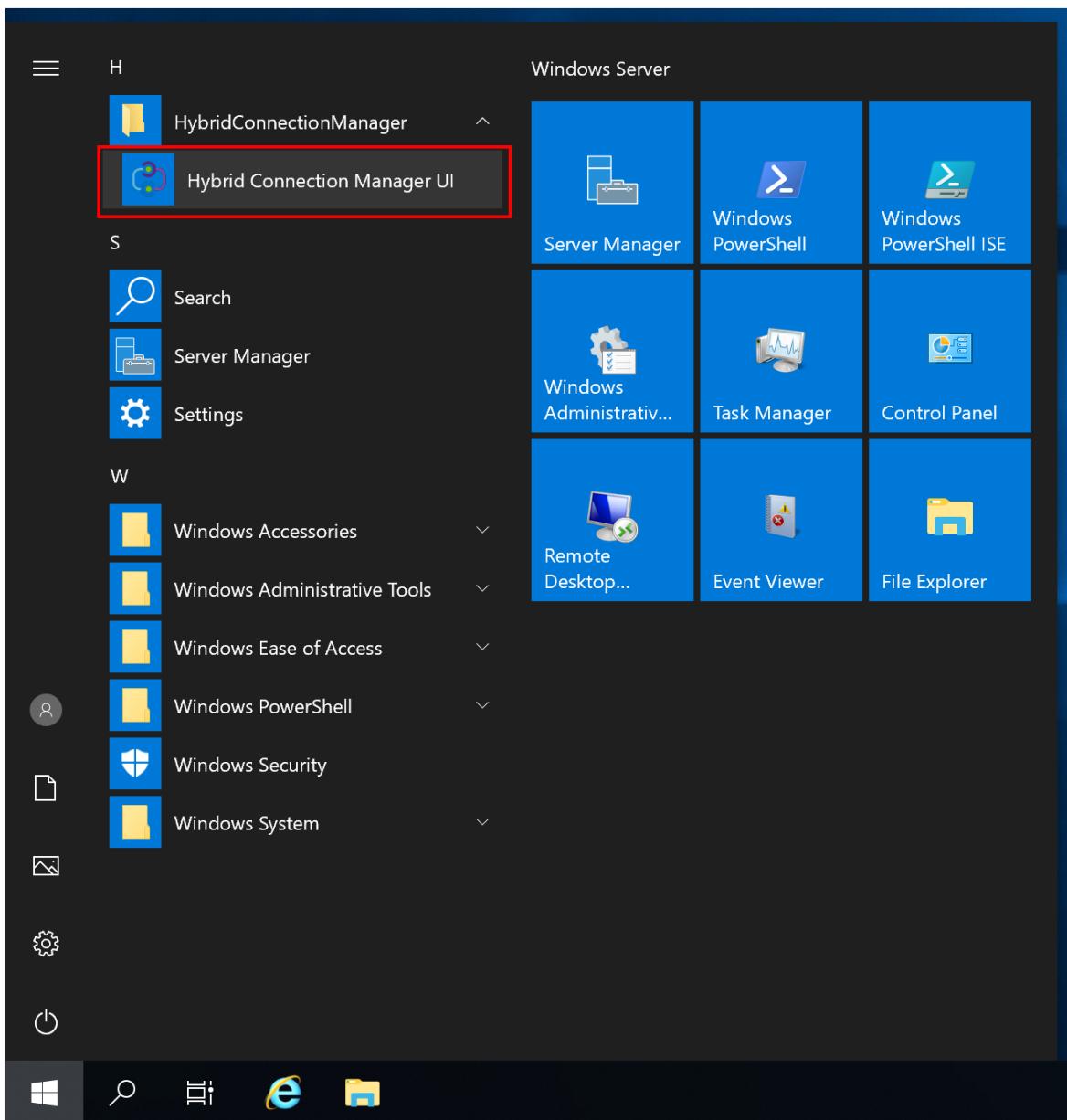
The screenshot shows the Azure portal interface for a hybrid connection named 'ContosoFinance1OnPremisesServer'. Key details include:

- HYBRID CONNECTION NAME:** ContosoFinance1OnPremisesServer
- ENDPOINT HOST:** finance1
- ENDPOINT PORT:** 5986
- SERVICE BUS NAMESPACE:** contosopowershellhybrid
- NAMESPACE LOCATION:** Central US
- HYBRID CONNECTION MANAGERS:** 0 connected

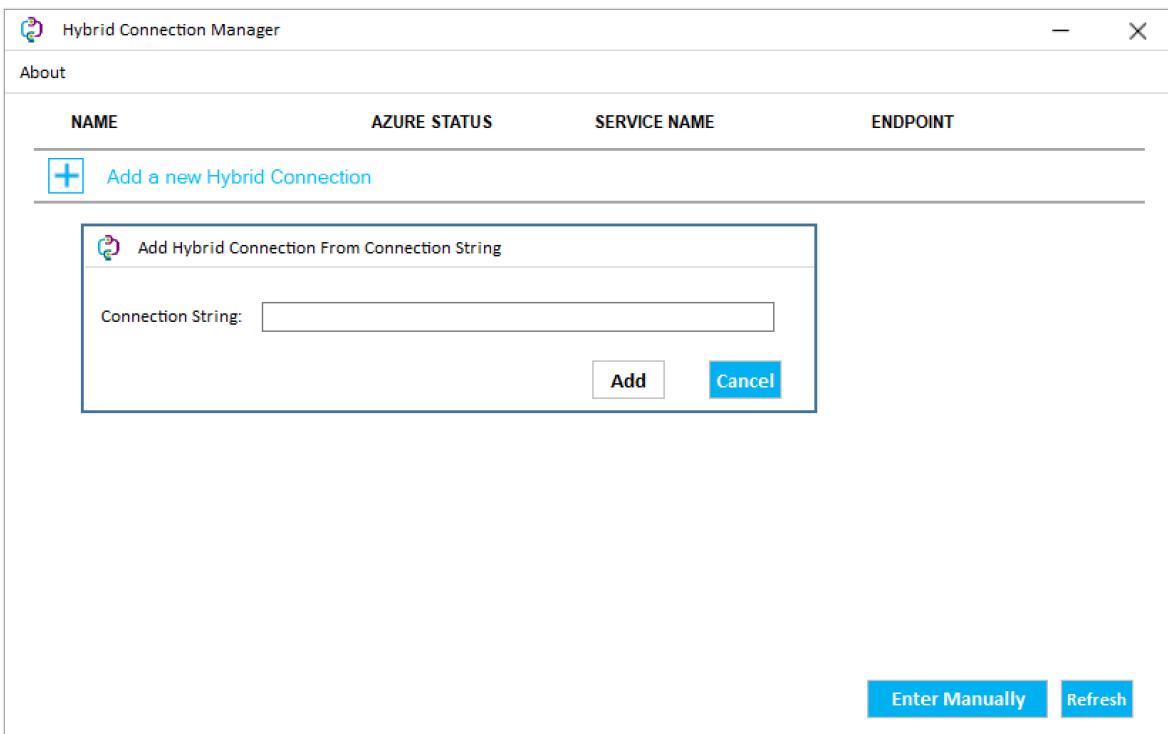
A context menu is open over the 'Copy to clipboard' button, showing the connection string:

```
GATEWAY=ContosoFinance1OnPremisesServer;Endpoint=sb://contoso...;
```

5. Open the Hybrid Connection Manager UI on the on-premises server.



6. Select Enter Manually and paste the connection string from the clipboard.



7. Restart the Hybrid Connection Manager from PowerShell if it doesn't show as connected.

```
PowerShell
Restart-Service HybridConnectionManager
```

Create an app setting for the password of an administrator account

1. Under **Settings** for your function app, select **Configuration**.
2. Select **+ New application setting**.

Home > myPowerShellFunctionApp-dma | Configuration

myPowerShellFunctionApp-dma | Configuration

App Service

Search (Ctrl+ /) Refresh Save Discard

Deployment Center

Settings

Configuration (highlighted)

Application settings Function runtime settings General settings

Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to display them in plain text in your browser by using the controls below. Application Settings are exposed as environment variables for access by your application at runtime. [Learn more](#)

+ New application setting Show values Advanced edit Filter

Name	Value	Source
APPINSIGHTS_INSTRUMENTATIONKEY	(Hidden value. Click to show value)	App Config
APPLICATIONINSIGHTS_CONNECTION_STRING	(Hidden value. Click to show value)	App Config
AzureWebJobsStorage	(Hidden value. Click to show value)	App Config
FUNCTIONS_EXTENSION_VERSION	(Hidden value. Click to show value)	App Config
FUNCTIONS_WORKER_RUNTIME	(Hidden value. Click to show value)	App Config

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel. Connection strings should only be used with a function app if you are using entity framework. For other scenarios use App Settings. [Learn more](#)

+ New connection string Show values Advanced edit Filter

Name	Value	Type	Deployment
(no connection strings to display)			

The screenshot shows the Azure portal's Configuration blade for a function app named 'myPowerShellFunctionApp-dma'. On the left, a sidebar lists various settings like Deployment Center, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, TLS/SSL settings, Networking, Scale up (App Service plan), Scale out (App Service plan), WebJobs, Push, MySQL In App, Properties, Locks, and Export template. Below these are sections for App Service plan, Development Tools, and Console. The 'Configuration' section is highlighted with a red box. At the top right, there are buttons for Refresh, Save, and Discard. The main area has tabs for Application settings, Function runtime settings, and General settings, with 'Application settings' selected. It contains a section for Application settings with a note about encryption and environment variables, followed by a table of current settings. A red box highlights the '+ New application setting' button. Below this is a section for Connection strings with a note about Entity Framework usage, followed by a table indicating no connection strings are currently displayed.

3. Name the setting **ContosoUserPassword**, and enter the password. Select **OK**.

4. Select **Save** to store the password in the function application.

Home > myPowerShellFunctionApp-dma | Configuration

myPowerShellFunctionApp-dma | Configuration

App Service

Search (Ctrl+ /) Refresh Save Discard

Events

Functions

- Functions
- App keys
- App files
- Proxies

Deployment

- Deployment slots
- Deployment Center

Settings

- Configuration
- Authentication / Authorization
- Application Insights
- Identity
- Backups
- Custom domains
- TLS/SSL settings
- Networking
- Scale up (App Service plan)
- Scale out (App Service plan)
- Webjobs
- Push
- MySQL In App
- Properties

Application settings *

Function runtime settings

General settings

Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to display them in plain text in your browser by using the controls below. Application Settings are exposed as environment variables for access by your application at runtime. [Learn more](#)

+ New application setting Show values Advanced edit Filter

Name	Value	Source	Deployment slot setting	Delete	Edit
APPINSIGHTS_INSTRUMENTATIONKEY	(Hidden value. Click to show value)	App Config			
APPLICATIONINSIGHTS_CONNECTION_STRING	(Hidden value. Click to show value)	App Config			
AzureWebJobsStorage	(Hidden value. Click to show value)	App Config			
ContosoAdminPassword	(Hidden value. Click to show value)	App Config			
FUNCTIONS_EXTENSION_VERSION	(Hidden value. Click to show value)	App Config			
FUNCTIONS_WORKER_RUNTIME	(Hidden value. Click to show value)	App Config			

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel. Connection strings should only be used with a function app if you are using entity framework. For other scenarios use App Settings. [Learn more](#)

+ New connection string Show values Advanced edit Filter

Name	Value	Type	Deployment...	Delete	Edit
(no connection strings to display)					

Create a function HTTP trigger

1. In your function app, select **Functions**, and then select **+ Add**.

Home > myPowerShellFunctionApp-dma | Functions

myPowerShellFunctionApp-dma | Functions

App Service

Search (Ctrl+ /) <> **Add** Develop Locally Refresh Enable Disable Delete

Filter by name...

Name ↑↓	Trigger ↑↓	Status ↑↓
No results.		

Functions

App keys

App files

Proxies

Deployment slots

Deployment Center

Configuration

Authentication / Authorization

Application Insights

Identity

Backups

Custom domains

TLS/SSL settings

Networking

The screenshot shows the Azure portal interface for an App Service named "myPowerShellFunctionApp-dma". The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events, Functions, App keys, App files, Proxies, Deployment slots, Deployment Center, Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, TLS/SSL settings, and Networking. The main content area displays a table with columns for Name, Trigger, and Status, showing "No results." A red box highlights the "Add" button at the top right of the main area, and another red box highlights the "Functions" link in the sidebar.

2. Select the **HTTP trigger** template.

New Function

Create a new function in this function app. Start by selecting a template below.

[Templates](#) [Details](#)

 Search by template name



HTTP trigger

A function that will be run whenever it receives an HTTP request, responding based on data in the body or query string



Timer trigger

A function that will be run on a specified schedule



Azure Queue Storage trigger

A function that will be run whenever a message is added to a specified Azure Storage queue



Azure Service Bus Queue trigger

A function that will be run whenever a message is added to a specified Service Bus queue



Azure Service Bus Topic trigger

A function that will be run whenever a message is added to the specified Service Bus topic



Azure Blob Storage trigger

A function that will be run whenever a blob is added to a specified container

3. Name the new function and select **Create Function**.

New Function

Create a new function in this function app. Start by selecting a template below.

Templates Details

New Function*

HttpTrigger1

Authorization level* ⓘ

Function

Create Function

Test the function

1. In the new function, select **Code + Test**. Replace the PowerShell code from the template with the following code:

PowerShell

```
# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Write to the Azure Functions log stream.
Write-Output "PowerShell HTTP trigger function processed a request."

# Note that ContosoUserPassword is a function app setting, so I can
# access it as $env:ContosoUserPassword.
$UserName = "ContosoUser"
$securedPassword = ConvertTo-SecureString $Env:ContosoUserPassword -
AsPlainText -Force
$Credential =
[System.management.automation.pscredential]::new($UserName,
$SecuredPassword)

# This is the name of the hybrid connection Endpoint.
$HybridEndpoint = "finance1"

$Script = {
    Param(
        [Parameter(Mandatory=$True)]
        [String] $Service
```

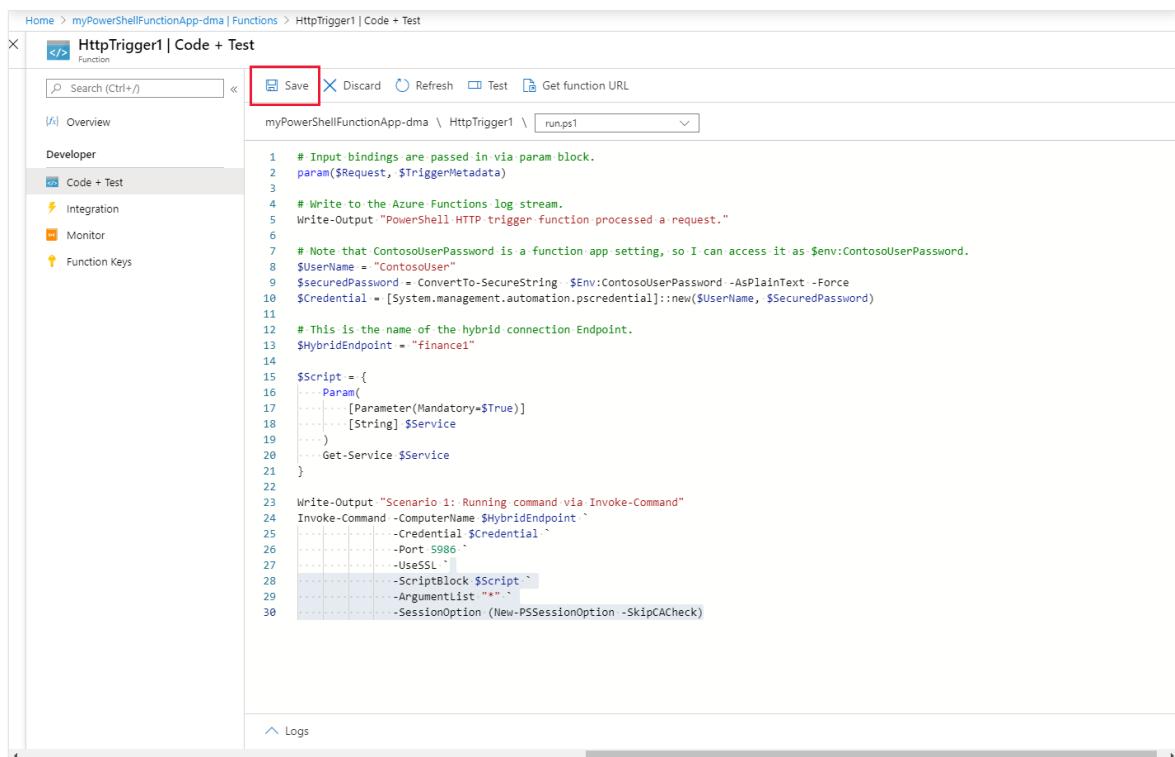
```

        )
    Get-Service $Service
}

Write-Output "Scenario 1: Running command via Invoke-Command"
Invoke-Command -ComputerName $HybridEndpoint `
    -Credential $Credential `
    -Port 5986 `
    -UseSSL `
    -ScriptBlock $Script `
    -ArgumentList "*"
    -SessionOption (New-PSSessionOption -SkipCACheck)

```

2. Select Save.



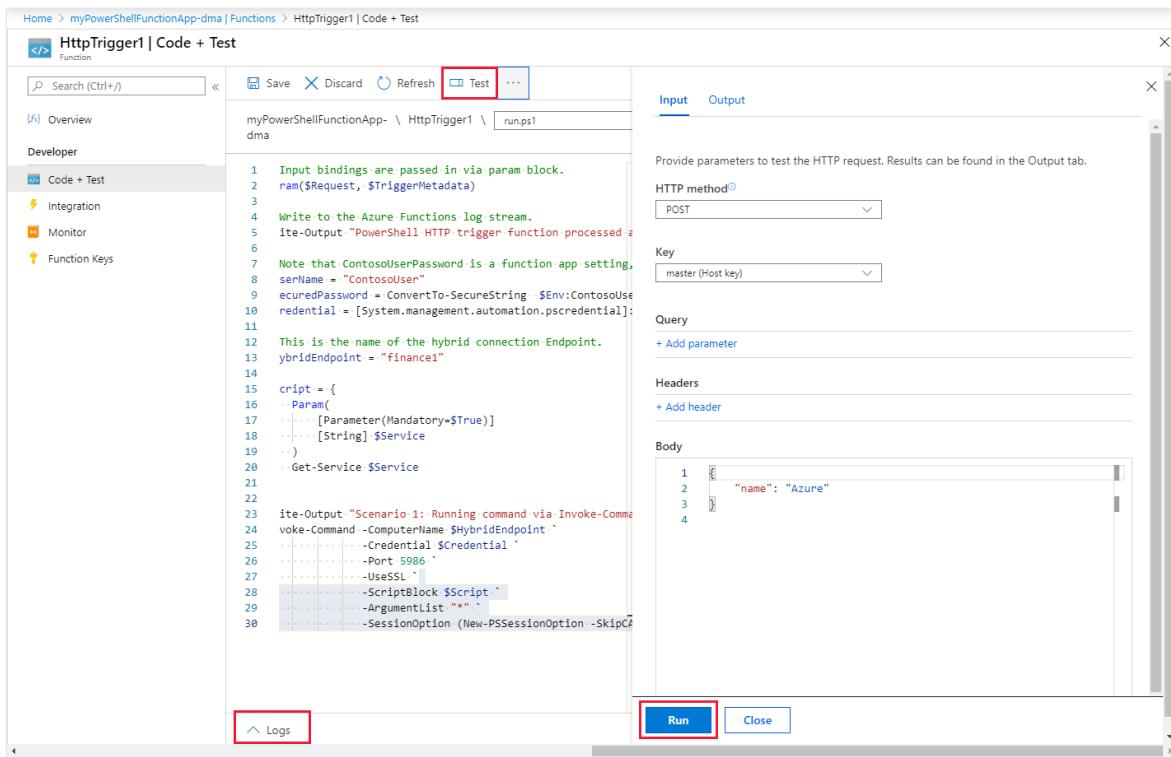
The screenshot shows the Azure Functions portal interface. The left sidebar has sections for Home, myPowerShellFunctionApp-dma, Functions, and HttpTrigger1. Under HttpTrigger1, there are tabs for Overview, Developer, and Code + Test. The Code + Test tab is selected. The main area displays the PowerShell script. A red box highlights the 'Save' button at the top of the code editor. The script itself is as follows:

```

1 # Input bindings are passed in via param block.
2 param($Request, $TriggerMetadata)
3
4 # Write to the Azure Functions log stream.
5 Write-Output "PowerShell HTTP trigger function processed a request."
6
7 # Note that ContosoUserPassword is a function app setting, so I can access it as $env:ContosoUserPassword.
8 $UserName = "ContosoUser"
9 $SecuredPassword = ConvertTo-SecureString $env:ContosoUserPassword -AsPlainText -Force
10 $Credential = [System.management.automation.pscredential]::new($UserName, $SecuredPassword)
11
12 # This is the name of the hybrid connection Endpoint.
13 $HybridEndpoint = "finance1"
14
15 $Script = {
16     Param(
17         [Parameter(Mandatory=$True)]
18         [String] $Service
19     )
20     Get-Service $Service
21 }
22
23 Write-Output "Scenario 1: Running command via Invoke-Command"
24 Invoke-Command -ComputerName $HybridEndpoint `
25     -Credential $Credential `
26     -Port 5986 `
27     -UseSSL `
28     -ScriptBlock $Script `
29     -ArgumentList "*"
30     -SessionOption (New-PSSessionOption -SkipCACheck)

```

3. Select Test, and then select Run to test the function. Review the logs to verify that the test was successful.



Managing other systems on-premises

You can use the connected on-premises server to connect to other servers and management systems in the local environment. This lets you manage your datacenter operations from Azure by using your PowerShell functions. The following script registers a PowerShell configuration session that runs under the provided credentials. These credentials must be for an administrator on the remote servers. You can then use this configuration to access other endpoints on the local server or datacenter.

```
PowerShell

# Input bindings are passed in via param block.
param($Request, $TriggerMetadata)

# Write to the Azure Functions log stream.
Write-Host "PowerShell HTTP trigger function processed a request."

# Note that ContosoUserPassword is a function app setting, so I can access it as $env:ContosoUserPassword.
$UserName = "ContosoUser"
$SecuredPassword = ConvertTo-SecureString $Env:ContosoUserPassword -AsPlainText -Force
$Credential = [System.management.automation.pscredential]::new($UserName, $SecuredPassword)

# This is the name of the hybrid connection Endpoint.
$HybridEndpoint = "finance1"

# The remote server that will be connected to run remote PowerShell commands on
```

```

$RemoteServer = "finance2".

Write-Output "Use hybrid connection server as a jump box to connect to a
remote machine"

# We are registering an endpoint that runs under credentials ($Credential)
# that has access to the remote server.
$SessionName = "HybridSession"
$ScriptCommand = {
    param (
        [Parameter(Mandatory=$True)]
        $SessionName)

    if (-not (Get-PSSessionConfiguration -Name $SessionName -ErrorAction
SilentlyContinue))
    {
        Register-PSSessionConfiguration -Name $SessionName -RunAsCredential
$Using:Credential
    }
}

Write-Output "Registering session on hybrid connection jumpbox"
Invoke-Command -ComputerName $HybridEndpoint `

    -Credential $Credential `

    -Port 5986 `

    -UseSSL `

    -ScriptBlock $ScriptCommand `

    -ArgumentList $SessionName `

    -SessionOption (New-PSSessionOption -SkipCACheck)

# Script to run on the jump box to run against the second machine.
$RemoteScriptCommand = {
    param (
        [Parameter(Mandatory=$True)]
        $ComputerName)
        # Write out the hostname of the hybrid connection server.
        hostname
        # Write out the hostname of the remote server.
        Invoke-Command -ComputerName $ComputerName -Credential
$Using:Credential -ScriptBlock {hostname} `

            -UseSSL -Port 5986 -SessionOption (New-
PSSessionOption -SkipCACheck)
}

Write-Output "Running command against remote machine via jumpbox by
connecting to the PowerShell configuration session"
Invoke-Command -ComputerName $HybridEndpoint `

    -Credential $Credential `

    -Port 5986 `

    -UseSSL `

    -ScriptBlock $RemoteScriptCommand `

    -ArgumentList $RemoteServer `

    -SessionOption (New-PSSessionOption -SkipCACheck) `

    -ConfigurationName $SessionName

```

Replace the following variables in this script with the applicable values from your environment:

- \$HybridEndpoint
- \$RemoteServer

In the two preceding scenarios, you can connect and manage your on-premises environments by using PowerShell in Azure Functions and Hybrid Connections. We encourage you to learn more about [Hybrid Connections](#) and [PowerShell in functions](#).

You can also use Azure [virtual networks](#) to connect to your on-premises environment through Azure Functions.

Next steps

[Learn more about working with PowerShell functions](#)

Troubleshoot error: "Azure Functions Runtime is unreachable"

Article • 02/16/2024

This article helps you troubleshoot the following error string that appears in the Azure portal:

"Error: Azure Functions Runtime is unreachable. Click here for details on storage configuration."

This issue occurs when the Functions runtime can't start. The most common reason for this is that the function app has lost access to its storage account. For more information, see [Storage account requirements](#).

The rest of this article helps you troubleshoot specific causes of this error, including how to identify and resolve each case.

Storage account was deleted

Every function app requires a storage account to operate. If that account is deleted, your functions won't work.

Start by looking up your storage account name in your application settings. Either `AzureWebJobsStorage` or `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` contains the name of your storage account as part of a connection string. For more information, see [App settings reference for Azure Functions](#).

Search for your storage account in the Azure portal to see whether it still exists. If it has been deleted, re-create the storage account and replace your storage connection strings. Your function code is lost, and you need to redeploy it.

Storage account application settings were deleted

In the preceding step, if you can't find a storage account connection string, it was likely deleted or overwritten. Deleting application settings most commonly happens when you're using deployment slots or Azure Resource Manager scripts to set application settings.

Required application settings

- Required:
 - [AzureWebJobsStorage](#)
- Required for Elastic Premium and Consumption plan functions:
 - [WEBSITE_CONTENTAZUREFILECONNECTIONSTRING](#)
 - [WEBSITE_CONTENTSHARE](#)

For more information, see [App settings reference for Azure Functions](#).

Guidance

- Don't check **slot setting** for any of these settings. If you swap deployment slots, the function app breaks.
- Don't modify these settings as part of automated deployments.
- These settings must be provided and valid at creation time. An automated deployment that doesn't contain these settings results in a function app that won't run, even if the settings are added later.

Storage account credentials are invalid

The previously discussed storage account connection strings must be updated if you regenerate storage keys. For more information about storage key management, see [Create an Azure Storage account](#).

Storage account is inaccessible

Your function app must be able to access the storage account. Common issues that block a function app's access to a storage account are:

- The function app is deployed to your App Service Environment (ASE) without the correct network rules to allow traffic to and from the storage account.
- The storage account firewall is enabled and not configured to allow traffic to and from functions. For more information, see [Configure Azure Storage firewalls and virtual networks](#).
- Verify that the `allowSharedKeyAccess` setting is set to `true`, which is its default value. For more information, see [Prevent Shared Key authorization for an Azure Storage account](#).

Daily execution quota is full

If you have a daily execution quota configured, your function app is temporarily disabled, which causes many of the portal controls to become unavailable.

To verify the quota in the [Azure portal](#), select **Platform Features > Function App Settings** in your function app. If you're over the **Daily Usage Quota** you've set, the following message is displayed:

"The Function App has reached daily usage quota and has been stopped until the next 24 hours time frame."

To resolve this issue, remove or increase the daily quota, and then restart your app. Otherwise, the execution of your app is blocked until the next day.

App is behind a firewall

Your function app might be unreachable for either of the following reasons:

- Your function app is hosted in an [internally load balanced App Service Environment](#) and it's configured to block inbound internet traffic.
- Your function app has [inbound IP restrictions](#) that are configured to block internet access.

The Azure portal makes calls directly to the running app to fetch the list of functions, and it makes HTTP calls to the Kudu endpoint. Platform-level settings under the **Platform Features** tab are still available.

To verify your ASE configuration:

1. Go to the network security group (NSG) of the subnet where the ASE resides.
2. Validate the inbound rules to allow traffic that's coming from the public IP of the computer where you're accessing the application.

You can also use the portal from a computer that's connected to the virtual network that's running your app or to a virtual machine that's running in your virtual network.

For more information about inbound rule configuration, see the "Network Security Groups" section of [Networking considerations for an App Service Environment](#).

Container errors on Linux

For function apps that run on Linux in a container, the `Azure Functions runtime is unreachable` error can occur as a result of problems with the container. Use the following procedure to review the container logs for errors:

1. Navigate to the Kudu endpoint for the function app, which is located at `https://<FUNCTION_APP>.scm.azurewebsites.net`, where `<FUNCTION_APP>` is the name of your app.
2. Download the Docker logs .zip file and review the contents on your local computer.
3. Check for any logged errors that indicate that the container is unable to start successfully.

Container image unavailable

Errors can occur when the container image being referenced is unavailable or fails to start correctly. Check for any logged errors that indicate that the container is unable to start successfully.

You need to correct any errors that prevent the container from starting for the function app run correctly.

When the container image can't be found, you see a `manifest unknown` error in the Docker logs. In this case, you can use the Azure CLI commands documented at [How to target Azure Functions runtime versions](#) to change the container image being referenced. If you've deployed a [custom container image](#), you need to fix the image and redeploy the updated version to the referenced registry.

App container has conflicting ports

Your function app might be in an unresponsive state due to conflicting port assignment upon startup. This can happen in the following cases:

- Your container has separate services running where one or more services are trying to bind to the same port as the function app.
- You've added an Azure Hybrid Connection that shares the same port value as the function app.

By default, the container in which your function app runs uses port `:80`. When other services in the same container are also trying to use port `:80`, the function app can fail to start. If your logs show port conflicts, change the default ports.

Host ID collision

Starting with version 3.x of the Functions runtime, host ID collision are detected and logged as a warning. In version 4.x, an error is logged and the host is stopped. If the runtime can't start for your function app, [review the logs](#). If there's a warning or an error about host ID collisions, follow the mitigation steps in [Host ID considerations](#).

Read-only app settings

Changing any *read-only* [App Service application settings](#) can put your function app into an unreachable state.

ASP.NET authentication overrides

Applies only to C# apps running [in-process with the Functions host](#).

Configuring ASP.NET authentication in a Functions startup class can override services that are required for the Azure portal to communicate with the host. This includes, but isn't limited to, any calls to `AddAuthentication()`. If the host's authentication services are overridden and the portal can't communicate with the host, it considers the app unreachable. This issue may result in errors such as: `No authentication handler is registered for the scheme 'ArmToken'..`

Next steps

Learn about monitoring your function apps:

[Monitor Azure Functions](#)

Diagnose and troubleshoot issues with the Azure Functions trigger for Azure Cosmos DB for NoSQL

Article • 08/14/2024

APPLIES TO:  NoSQL

This article covers common issues, workarounds, and diagnostic steps when you're using the [Azure Functions trigger for Azure Cosmos DB for NoSQL](#).

Dependencies

The Azure Functions' triggers and bindings for Azure Cosmos DB for NoSQL depend on the extension package [Microsoft.Azure.WebJobs.Extensions.CosmosDB](#) over the base Azure Functions runtime. Always keep these packages updated, because they include fixes and new features that can help you address any potential issues you might encounter.

Consume the Azure Cosmos DB SDK independently

The key functionality of the extension package is to provide support for the Azure Functions trigger and bindings for Azure Cosmos DB. The package also includes the [Azure Cosmos DB .NET SDK](#), which is helpful if you want to interact with Azure Cosmos DB programmatically without using the trigger and bindings.

If you want to use the Azure Cosmos DB SDK, make sure that you don't add to your project another NuGet package reference. Instead, let the SDK reference resolve through the Azure Functions extension package. Consume the Azure Cosmos DB SDK separately from the trigger and bindings.

Additionally, if you're manually creating your own instance of the [Azure Cosmos DB SDK client](#), you should follow the pattern of having only one instance of the client and [use a singleton pattern approach](#). This process avoids the potential socket issues in your operations.

Common scenarios and workarounds

Your Azure function fails with an error message that the collection "doesn't exist"

The Azure function fails with the following error message: "Either the source collection `collection-name` (in database `database-name`) or the lease collection `collection2-name` (in database `database2-name`) doesn't exist. Both collections must exist before the listener starts. To automatically create the lease collection, set `CreateLeaseCollectionIfNotExists` to `true`.

This error means that one or both of the Azure Cosmos DB containers that are required for the trigger to work either:

- Don't exist
- Aren't reachable to the Azure function

The error text itself tells you which Azure Cosmos DB database and container the trigger is looking for, based on your configuration.

To resolve this issue:

1. Verify the `Connection` attribute and that it references a setting that exists in your Azure function app.
 - The value on this attribute shouldn't be the connection string itself, but the name of the configuration setting.
2. Verify that the `databaseName` and `containerName` values exist in your Azure Cosmos DB account.
 - If you're using automatic value replacement (using `%settingName%` patterns), make sure that the name of the setting exists in your Azure function app.
3. If you don't specify a `LeaseContainerName/leaseContainerName` value, the default is `leases`. Verify that such a container exists.
 - Optionally, you can set the `CreateLeaseContainerIfNotExists` attribute in your trigger to `true` to automatically create it.
4. To ensure that your Azure Cosmos DB account isn't blocking the Azure function, verify your [Azure Cosmos DB account's firewall configuration](#).

Your Azure function fails to start, with error message "Shared throughput collection should have a partition

key "

Previous versions of the Azure Cosmos DB extension didn't support using a leases container that was created within a [shared throughput database](#).

To resolve this issue:

- Update the [Microsoft.Azure.WebJobs.Extensions.CosmosDB](#) extension to get the latest version.

Your Azure function fails to start, with error message

"PartitionKey must be supplied for this operation"

This error means that you're currently using a partitioned lease collection with an old [extension dependency](#).

To resolve this issue:

- Upgrade to the latest available version.

Your Azure function fails to start, with error message

"Forbidden (403); Substatus: 5300... The given request [POST ...] can't be authorized by Microsoft Entra token in data plane"

This error means that your function is attempting to [perform a nondata operation by using Microsoft Entra identities](#). You can't use `CreateLeaseContainerIfNotExists = true` when you're using Microsoft Entra identities.

Your Azure function fails to start, with error message "The lease collection, if partitioned, must have partition key equal to id"

This error means that your current leases container is partitioned, but the partition key path isn't `/id`.

To resolve this issue:

- Re-create the leases container with `/id` as the partition key.

When you try to run the trigger, you get the error message "Value can't be null. Parameter name: o" in your Azure function logs

This issue might arise if you're using the Azure portal and you select the **Run** button when you're inspecting an Azure function that uses the trigger. The trigger doesn't require you to select **Run** to start it. It automatically starts when you deploy the function.

To resolve this issue:

- To check the function's log stream on the Azure portal, go to your monitored container and insert some new items. The trigger runs automatically.

Your changes are taking too long to be received

This scenario can have multiple causes. Consider trying any or all of the following solutions:

- Are your Azure function and your Azure Cosmos DB account deployed in separate regions? For optimal network latency, your Azure function and your Azure Cosmos DB account should be colocated in the same Azure region.
- Are the changes that are happening in your Azure Cosmos DB container continuous or sporadic?
 - If they're sporadic, there could be some delay between the changes being stored and the Azure function that's picking them up. This behavior occurs when the trigger checks internally for changes in your Azure Cosmos DB container and finds no changes waiting to be read. The trigger then sleeps for a configurable amount of time (5 seconds, by default) before it checks for new changes. The trigger utilizes this behavior to avoid high request unit (RU) consumption. You can configure the sleep time through the `FeedPollDelay/feedPollDelay` setting in the [configuration](#) of your trigger. The value is expected to be in milliseconds.
- Your Azure Cosmos DB container might be [rate-limited](#).
- You can use the `PreferredLocations` attribute in your trigger to specify a comma-separated list of Azure regions to define a custom preferred connection order.
- The speed at which you're processing new changes dictates the speed at which your trigger receives those changes. Verify the function's [execution time](#), or [duration](#). If your function is slow, that increases the time it takes for the trigger to get new changes. If you see a recent increase in duration, a recent code change

might be affecting it. If the speed at which you're receiving operations on your Azure Cosmos DB container is faster than the speed of your trigger, it keeps lagging behind. You might want to investigate the function's code to determine the most time-consuming operation and how to optimize it.

- You can use [Debug logs](#) to check the Diagnostics and verify if there are networking delays.

Some changes are repeated in my trigger

The concept of a *change* is an operation on an item. The most common scenarios where events for the same item are received are:

- Your Function is failing during execution. If [retry policies](#) are enabled for your function or in cases where your Function execution is exceeding the allowed execution time, the same batch of changes can be delivered again to your Function. This failure is expected and by design, look at your Function logs for indication of failures and make sure you enabled [trigger logs](#) for further details.
- There's a load balancing of leases across instances. When instances increase or decrease, [load balancing](#) can cause the same batch of changes to be delivered to multiple Function instances. This load balancing is expected and by design, and should be transient. The [trigger logs](#) include the events when an instance acquires and releases leases.
- The item is being updated. The change feed can contain multiple operations for the same item. If the item is receiving updates, it can pick up multiple events (one for each update). One easy way to distinguish among different operations for the same item is to track the `_lsn` [property for each change](#). If the properties don't match, the changes are different.
- If you're identifying items only by `id`, remember that the unique identifier for an item is the `id` and its partition key. (Two items can have the same `id` but a different partition key).

Some changes are missing in your trigger

You might find that the function didn't pick up some of the changes that occurred in your Azure Cosmos DB for NoSQL container. Or some changes are missing at the destination when you're copying them. If so, try the solutions in this section.

- Make sure you have [logs](#) enabled. Verify no errors are happening during processing.
- When your Azure function receives the changes, it often processes them and could, optionally, send the result to another destination. When you're investigating missing changes, make sure that you measure which changes are being received at the ingestion point. Measure when the Azure function starts, not at the destination.
- If some changes are missing on the destination, this behavior could indicate that some error is happening during the Azure function execution after the changes were received.
 - In this scenario, the best course of action is to add `try/catch` blocks in your code and inside the loops that might be processing the changes. Adding it helps you detect any failure for a particular subset of items and handle them accordingly (send them to another storage for further analysis or retry). Alternatively, you can configure Azure Functions [retry policies](#).

ⓘ Note

The Azure Functions' trigger for Azure Cosmos DB for NOSQL, by default, won't retry a batch of changes if there was an unhandled exception during the code execution. This means that the reason that the changes didn't arrive at the destination might be because you've failed to process them.

- If the destination is another Azure Cosmos DB container and you're performing upsert operations to copy the items, verify the partition key definition on both containers. The partition key on the monitored and destination container must be the same. Upsert operations could be saving multiple source items as one at the destination because of this configuration difference.
- If you find that the trigger didn't receive some changes, the most common scenario is that another Azure function is running. The other function might be deployed in Azure or a function might be running locally on a developer's machine with exactly the same configuration. If so, this function might be stealing a subset of the changes that you would expect your Azure function to process.

Additionally, the scenario can be validated, if you know how many Azure function app instances you have running. If you inspect your leases container and count the number of lease items within it, the distinct values of the `owner` property in them should be equal to the number of instances of your function app. If there are more owners than

known Azure function app instances, it means that these extra owners are the ones "stealing" the changes.

One easy way to work around this situation is to apply a `LeaseCollectionPrefix/leaseCollectionPrefix` to your function with a new or different value or, alternatively, to test with a new leases container.

You need to restart and reprocess all the items in your container from the beginning

To reprocess all the items in a container from the beginning:

1. Stop your Azure function if it's currently running.
2. Delete the documents in the lease collection (or delete and re-create the lease collection so that it's empty).
3. Set the `StartFromBeginning` CosmosDBTrigger attribute in your function to `true`.
4. Restart the Azure function. The function now reads and processes all changes from the beginning.

Setting `StartFromBeginning` to `true` tells the Azure function to start reading changes from the beginning of the history of the collection instead of the current time.

This solution works only when there are no already-created leases (that is, documents in the leases collection).

Setting this property to `true` has no effect when there are leases already created. In this scenario, when a function is stopped and restarted, it begins reading from the last checkpoint, as defined in the leases collection.

Error: Binding can be done only with `IReadOnlyList<Document>` or `JArray`

This error happens if your Azure Functions project contains a manual NuGet package reference to the Azure Cosmos DB SDK with a version conflict. This error commonly occurs because the package has a version that's different from the one provided by the [Azure Cosmos DB extension for Azure Functions](#).

To work around this situation, remove the manual NuGet reference that was added, and let the Azure Cosmos DB SDK reference resolve through the Azure Cosmos DB extension for Azure Functions package.

Change the Azure function's polling interval for detecting changes

As explained earlier in the [Your changes are taking too long to be received](#) section, your Azure function sleeps for a configurable amount of time (5 seconds, by default) before it checks for new changes (to avoid high RU consumption). You can configure this sleep time through the `FeedPollDelay/feedPollDelay` setting in the [trigger configuration](#) (the value is expected to be in milliseconds).

Related content

- [Enable monitoring for your Azure function](#)
- [Troubleshoot the Azure Cosmos DB .NET SDK](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Troubleshoot Node.js apps in Azure Functions

Article • 09/26/2023

ⓘ Important

The content of this article changes based on your choice of the Node.js programming model in the selector at the top of the page. The v4 model is generally available and is designed to have a more flexible and intuitive experience for JavaScript and TypeScript developers. Learn more about the differences between v3 and v4 in the [migration guide](#).

This article provides a guide for troubleshooting common scenarios in Node.js function apps.

The **Diagnose and solve problems** tab in the [Azure portal](#) is a useful resource to monitor and diagnose possible issues related to your application. It also supplies potential solutions to your problems based on the diagnosis. For more information, see [Azure Function app diagnostics](#).

Another useful resource is the **Logs** tab in the [Azure portal](#) for your Application Insights instance so that you can run custom [KQL queries](#). The following example query shows how to view errors and warnings for your app in the past day:

Kusto

```
let myAppName = "<your app name>";
let startTime = ago(1d);
let endTime = now();
union traces,requests,exceptions
| where cloud_RoleName =~ myAppName
| where timestamp between (startTime .. endTime)
| where severityLevel > 2
```

If those resources didn't solve your problem, the following sections provide advice for specific application issues:

No functions found

If you see any of the following errors in your logs:

No HTTP triggers found.

No job functions found. Try making your job classes and methods public. If you're using binding extensions (e.g. Azure Storage, ServiceBus, Timers, etc.) make sure you've called the registration method for the extension(s) in your startup code (e.g. `builder.AddAzureStorage()`, `builder.AddServiceBus()`, `builder.AddTimers()`, etc.).

Try the following fixes:

- When running locally, make sure you're using Azure Functions Core Tools v4.0.5382 or higher.
- When running in Azure:
 - Make sure you're using [Azure Functions Runtime Version 4.25](#) or higher.
 - Make sure you're using Node.js v18 or higher.
 - Set the app setting `FUNCTIONS_NODE_BLOCK_ON_ENTRY_POINT_ERROR` to `true`. This setting is recommended for all model v4 apps and ensures that all entry point errors are visible in your application insights logs. For more information, see [App settings reference for Azure Functions](#).
 - Check your function app logs for entry point errors. The following example query shows how to view entry point errors for your app in the past day:

```
Kusto

let myAppName = "<your app name>";
let startTime = ago(1d);
let endTime = now();
union traces,requests,exceptions
| where cloud_RoleName =~ myAppName
| where timestamp between (startTime .. endTime)
| where severityLevel > 2
| where message has "entry point"
```

Undici request is not a constructor

If you get the following error in your function app logs:

```
System.Private.CoreLib: Exception while executing function: Functions.httpTrigger1.
System.Private.CoreLib: Result: Failure Exception: undici_1.Request is not a
constructor
```

Make sure you're using Node.js version 18.x or higher.

Failed to detect the Azure Functions runtime

If you get the following error in your function app logs:

WARNING: Failed to detect the Azure Functions runtime. Switching "@azure/functions" package to test mode - not all features are supported.

Check your `package.json` file for a reference to `applicationinsights` and make sure the version is `^2.7.1` or higher. After updating the version, run `npm install`

Get help from Microsoft

You can get more help from Microsoft in one of the following ways:

- Search the known issues in the [Azure Functions Node.js repository](#). If you don't see your issue mentioned, create a new issue and let us know what has happened.
- If you're not able to diagnose your problem using this guide, Microsoft support engineers are available to help diagnose issues with your application. Microsoft offers [various support plans](#). Create a support ticket in the **Support + troubleshooting** section of your function app page in the [Azure portal](#).

Next steps

- [Microsoft Q&A page for Azure Functions](#)
- [Azure Functions Node.js developer guide](#)

Troubleshoot Python errors in Azure Functions

Article • 02/25/2024

This article provides information to help you troubleshoot errors with your Python functions in Azure Functions. This article supports both the v1 and v2 programming models. Choose the model you want to use from the selector at the top of the article.

ⓘ Note

The Python v2 programming model is only supported in the 4.x functions runtime. For more information, see [Azure Functions runtime versions overview](#).

Here are the troubleshooting sections for common issues in Python functions:

Specifically with the v2 model, here are some known issues and their workarounds:

- [Could not load file or assembly](#)
- [Unable to resolve the Azure Storage connection named Storage](#)

General troubleshooting guides for Python Functions include:

- [ModuleNotFoundError and ImportError](#)
- [Cannot import 'cygrpc'](#)
- [Python exited with code 137](#)
- [Python exited with code 139](#)
- [Sync triggers failed](#)
- [Development issues in the Azure portal](#)

Troubleshoot: ModuleNotFoundError

This section helps you troubleshoot module-related errors in your Python function app. These errors typically result in the following Azure Functions error message:

Exception: ModuleNotFoundError: No module named 'module_name'.

This error occurs when a Python function app fails to load a Python module. The root cause for this error is one of the following issues:

- [The package can't be found](#)

- The package isn't resolved with proper Linux wheel
- The package is incompatible with the Python interpreter version
- The package conflicts with other packages
- The package supports only Windows and macOS platforms

View project files

To identify the actual cause of your issue, you need to get the Python project files that run on your function app. If you don't have the project files on your local computer, you can get them in one of the following ways:

- If the function app has a `WEBSITE_RUN_FROM_PACKAGE` app setting and its value is a URL, download the file by copying and pasting the URL into your browser.
- If the function app has `WEBSITE_RUN_FROM_PACKAGE` set to `1`, go to `https://<app-name>.scm.azurewebsites.net/api/vfs/data/SitePackages` and download the file from the latest `href` URL.
- If the function app doesn't have either of the preceding app settings, go to `https://<app-name>.scm.azurewebsites.net/api/settings` and find the URL under `SCM_RUN_FROM_PACKAGE`. Download the file by copying and pasting the URL into your browser.
- If suggestions resolve the issue, go to `https://<app-name>.scm.azurewebsites.net/DebugConsole` and view the content under `/home/site/wwwroot`.

The rest of this article helps you troubleshoot potential causes of this error by inspecting your function app's content, identifying the root cause, and resolving the specific issue.

Diagnose ModuleNotFoundError

This section details potential root causes of module-related errors. After you figure out which is the likely root cause, you can go to the related mitigation.

The package can't be found

Go to `.python_packages/lib/python3.6/site-packages/<package-name>` or `.python_packages/lib/site-packages/<package-name>`. If the file path doesn't exist, this missing path is likely the root cause.

Using third-party or outdated tools during deployment might cause this issue.

To mitigate this issue, see [Enable remote build](#) or [Build native dependencies](#).

The package isn't resolved with the proper Linux wheel

Go to `.python_packages/lib/python3.6/site-packages/<package-name>-<version>-dist-info` or `.python_packages/lib/site-packages/<package-name>-<version>-dist-info`. Use your favorite text editor to open the *wheel* file and check the **Tag:** section. The issue might be that the tag value doesn't contain `linux`.

Python functions run only on Linux in Azure. The Functions runtime v2.x runs on Debian Stretch, and the v3.x runtime runs on Debian Buster. The artifact is expected to contain the correct Linux binaries. When you use the `--build local` flag in Core Tools, third-party, or outdated tools, it might cause older binaries to be used.

To mitigate the issue, see [Enable remote build](#) or [Build native dependencies](#).

The package is incompatible with the Python interpreter version

Go to `.python_packages/lib/python3.6/site-packages/<package-name>-<version>-dist-info` or `.python_packages/lib/site-packages/<package-name>-<version>-dist-info`. In your text editor, open the *METADATA* file and check the **Classifiers:** section. If the section doesn't contain `Python :: 3`, `Python :: 3.6`, `Python :: 3.7`, `Python :: 3.8`, or `Python :: 3.9`, the package version is either too old or, more likely, it's already out of maintenance.

You can check the Python version of your function app from the [Azure portal](#). Navigate to your function app's **Overview** resource page to find the runtime version. The runtime version supports Python versions as described in the [Azure Functions runtime versions overview](#).

To mitigate the issue, see [Update your package to the latest version](#) or [Replace the package with equivalents](#).

The package conflicts with other packages

If you've verified that the package is resolved correctly with the proper Linux wheels, there might be a conflict with other packages. In certain packages, the PyPi documentation might clarify the incompatible modules. For example, in [azure 4.0.0](#), you find the following statement:

This package isn't compatible with `azure-storage`. If you installed `azure-storage`, or if you installed `azure 1.x/2.x` and didn't uninstall `azure-storage`, you must uninstall `azure-storage` first.

You can find the documentation for your package version in <https://pypi.org/project/<package-name>/<package-version>>.

To mitigate the issue, see [Update your package to the latest version](#) or [Replace the package with equivalents](#).

The package supports only Windows and macOS platforms

Open the `requirements.txt` with a text editor and check the package in <https://pypi.org/project/<package-name>>. Some packages run only on Windows and macOS platforms. For example, `pywin32` runs on Windows only.

The `Module Not Found` error might not occur when you're using Windows or macOS for local development. However, the package fails to import on Azure Functions, which uses Linux at runtime. This issue is likely to be caused by using `pip freeze` to export the virtual environment into `requirements.txt` from your Windows or macOS machine during project initialization.

To mitigate the issue, see [Replace the package with equivalents](#) or [Handcraft requirements.txt](#).

Mitigate ModuleNotFoundError

The following are potential mitigations for module-related issues. Use the [previously mentioned diagnoses](#) to determine which of these mitigations to try.

Enable remote build

Make sure that remote build is enabled. The way that you make sure depends on your deployment method.

Visual Studio Code

Make sure that the latest version of the [Azure Functions extension for Visual Studio Code](#) is installed. Verify that the `.vscode/settings.json` file exists and it contains the setting `"azureFunctions.scmDoBuildDuringDeployment": true`. If it doesn't, create the file with the `azureFunctions.scmDoBuildDuringDeployment` setting enabled, and then redeploy the project.

Build native dependencies

Make sure that the latest versions of both Docker and [Azure Functions Core Tools](#) are installed. Go to your local function project folder, and use `func azure functionapp publish <app-name> --build-native-deps` for deployment.

Update your package to the latest version

In the latest package version of <https://pypi.org/project/<package-name>>, check the **Classifiers:** section. The package should be `OS Independent`, or compatible with `POSIX` or `POSIX :: Linux` in **Operating System**. Also, the programming language should contain: `Python :: 3`, `Python :: 3.6`, `Python :: 3.7`, `Python :: 3.8`, or `Python :: 3.9`.

If these package items are correct, you can update the package to the latest version by changing the line `<package-name>~=<latest-version>` in *requirements.txt*.

Handcraft requirements.txt

Some developers use `pip freeze > requirements.txt` to generate the list of Python packages for their developing environments. Although this convenience should work in most cases, there can be issues in cross-platform deployment scenarios, such as developing functions locally on Windows or macOS, but publishing to a function app, which runs on Linux. In this scenario, `pip freeze` can introduce unexpected operating system-specific dependencies or dependencies for your local development environment. These dependencies can break the Python function app when it's running on Linux.

The best practice is to check the import statement from each `.py` file in your project source code and then check in only the modules in the *requirements.txt* file. This practice guarantees that the resolution of packages can be handled properly on different operating systems.

Replace the package with equivalents

First, take a look into the latest version of the package in <https://pypi.org/project/<package-name>>. This package usually has its own GitHub page. Go to the **Issues** section on GitHub and search to see whether your issue has been fixed. If it has been fixed, update the package to the latest version.

Sometimes, the package might have been integrated into [Python Standard Library](#) (such as `pathlib`). If so, because we provide a certain Python distribution in Azure Functions (Python 3.6, Python 3.7, Python 3.8, and Python 3.9), the package in your *requirements.txt* file should be removed.

However, if you're finding that the issue hasn't been fixed, and you're on a deadline, we encourage you to do some research to find a similar package for your project. Usually, the Python community provides you with a wide variety of similar libraries that you can use.

Disable dependency isolation flag

Set the application setting `PYTHON_ISOLATE_WORKER_DEPENDENCIES` to a value of `0`.

Troubleshoot: cannot import 'cygrpc'

This section helps you troubleshoot 'cygrpc'-related errors in your Python function app. These errors typically result in the following Azure Functions error message:

Cannot import name 'cygrpc' from 'grpc._cython'

This error occurs when a Python function app fails to start with a proper Python interpreter. The root cause for this error is one of the following issues:

- [The Python interpreter mismatches OS architecture](#)
- [The Python interpreter isn't supported by Azure Functions Python Worker](#)

Diagnose the 'cygrpc' reference error

There are several possible causes for errors that reference `cygrpc`, which are detailed in this section.

The Python interpreter mismatches OS architecture

This mismatch is most likely caused by a 32-bit Python interpreter being installed on your 64-bit operating system.

If you're running on an x64 operating system, ensure that your Python version 3.6, 3.7, 3.8, or 3.9 interpreter is also on a 64-bit version.

You can check your Python interpreter bitness by running the following commands:

On Windows in PowerShell, run `py -c 'import platform; print(platform.architecture()[0])'`.

On a Unix-like shell, run `python3 -c 'import platform; print(platform.architecture()[0])'`.

If there's a mismatch between Python interpreter bitness and the operating system architecture, download a proper Python interpreter from [Python Software Foundation](#).

The Python interpreter isn't supported by Azure Functions Python Worker

The Azure Functions Python Worker supports only [specific Python versions](#).

Check to see whether your Python interpreter matches your expected version by `py --version` in Windows or `python3 --version` in Unix-like systems. Ensure that the return result is one of the [supported Python versions](#).

If your Python interpreter version doesn't meet the requirements for Azure Functions, instead download a Python interpreter version that is supported by Functions from the [Python Software Foundation](#).

Troubleshoot: python exited with code 137

Code 137 errors are typically caused by out-of-memory issues in your Python function app. As a result, you get the following Azure Functions error message:

```
Microsoft.Azure.WebJobs.Script.Workers.WorkerProcessExitException : python exited with code 137
```

This error occurs when a Python function app is forced to terminate by the operating system with a `SIGKILL` signal. This signal usually indicates an out-of-memory error in your Python process. The Azure Functions platform has a [service limitation](#) that terminates any function apps that exceed this limit.

To analyze the memory bottleneck in your function app, see [Profile Python function app in local development environment](#).

Troubleshoot: python exited with code 139

This section helps you troubleshoot segmentation fault errors in your Python function app. These errors typically result in the following Azure Functions error message:

Microsoft.Azure.WebJobs.Script.Workers.WorkerProcessExitException : python exited with code 139

This error occurs when a Python function app is forced to terminate by the operating system with a `SIGSEGV` signal. This signal indicates violation of the memory segmentation, which can result from an unexpected reading from or writing into a restricted memory region. In the following sections, we provide a list of common root causes.

A regression from third-party packages

In your function app's `requirements.txt` file, an unpinned package gets upgraded to the latest version during each deployment to Azure. Package updates can potentially introduce regressions that affect your app. To recover from such issues, comment out the import statements, disable the package references, or pin the package to a previous version in `requirements.txt`.

Unpickling from a malformed .pkl file

If your function app is using the Python pickle library to load a Python object from a `.pkl` file, it's possible that the file contains a malformed bytes string or an invalid address reference. To recover from this issue, try commenting out the `pickle.load()` function.

Pyodbc connection collision

If your function app is using the popular ODBC database driver [pyodbc](#), it's possible that multiple connections are open within a single function app. To avoid this issue, use the singleton pattern, and ensure that only one pyodbc connection is used across the function app.

Sync triggers failed

The error `Sync triggers failed` can be caused by several issues. One potential cause is a conflict between customer-defined dependencies and Python built-in modules when your functions run in an App Service plan. For more information, see [Package management](#).

Troubleshoot: could not load file or assembly

You can see this error when you're running locally using the v2 programming model. This error is caused by a known issue to be resolved in an upcoming release.

This is an example message for this error:

```
DurableTask.Netherite.AzureFunctions: Could not load file or assembly  
'Microsoft.Azure.WebJobs.Extensions.DurableTask, Version=2.0.0.0, Culture=neutral,  
PublicKeyToken=014045d636e89289'.  
The system cannot find the file specified.
```

The error occurs because of an issue with how the extension bundle was cached. To troubleshoot the issue, run this command with `--verbose` to see more details:

```
Console  
func host start --verbose
```

It's likely you're seeing this caching issue when you see an extension loading log like `Loading startup extension <>` that isn't followed by `Loaded extension <>`.

To resolve this issue:

1. Find the `.azure-functions-core-tools` path by running:

```
Console  
func GetExtensionBundlePath
```

2. Delete the `.azure-functions-core-tools` directory.

```
Bash  
Bash  
rm -r <insert path>/ .azure-functions-core-tools
```

The cache directory is recreated when you run Core Tools again.

Troubleshoot: unable to resolve the Azure Storage connection

You might see this error in your local output as the following message:

```
Microsoft.Azure.WebJobs.Extensions.DurableTask: Unable to resolve the Azure  
Storage connection named 'Storage'.  
Value cannot be null. (Parameter 'provider')
```

This error is a result of how extensions are loaded from the bundle locally. To resolve this error, take one of the following actions:

- Use a storage emulator such as [Azurite](#). This option is a good one when you aren't planning to use a storage account in your function application.
- Create a storage account and add a connection string to the `AzureWebJobsStorage` environment variable in the `localsettings.json` file. Use this option when you're using a storage account trigger or binding with your application, or if you have an existing storage account. To get started, see [Create a storage account](#).

Development issues in the Azure portal

When using the [Azure portal](#), take into account these known issues and their workarounds:

- There are general limitations for writing your function code in the portal. For more information, see [Development limitations in the Azure portal](#).
- To delete a function from a function app in the portal, remove the function code from the file itself. The **Delete** button doesn't work to remove the function when using the Python v2 programming model.
- When creating a function in the portal, you might be admonished to use a different tool for development. There are several scenarios where you can't edit your code in the portal, including when a syntax error has been detected. In these scenarios, use [Visual Studio Code](#) or [Azure Functions Core Tools](#) to develop and publish your function code.

Next steps

If you're unable to resolve your issue, contact the Azure Functions team:

[Report an unresolved issue](#)

Improve throughput performance of Python apps in Azure Functions

Article • 02/14/2023

When developing for Azure Functions using Python, you need to understand how your functions perform and how that performance affects the way your function app gets scaled. The need is more important when designing highly performant apps. The main factors to consider when designing, writing, and configuring your functions apps are horizontal scaling and throughput performance configurations.

Horizontal scaling

By default, Azure Functions automatically monitors the load on your application and creates more host instances for Python as needed. Azure Functions uses built-in thresholds for different trigger types to decide when to add instances, such as the age of messages and queue size for QueueTrigger. These thresholds aren't user configurable. For more information, see [Event-driven scaling in Azure Functions](#).

Improving throughput performance

The default configurations are suitable for most of Azure Functions applications. However, you can improve the performance of your applications' throughput by employing configurations based on your workload profile. The first step is to understand the type of workload that you're running.

Workload type	Function app characteristics	Examples
I/O-bound	<ul style="list-style-type: none">App needs to handle many concurrent invocations.App processes a large number of I/O events, such as network calls and disk read/writes.	<ul style="list-style-type: none">Web APIs
CPU-bound	<ul style="list-style-type: none">App does long-running computations, such as image resizing.App does data transformation.	<ul style="list-style-type: none">Data processingMachine learning inference

As real world function workloads are usually a mix of I/O and CPU bound, you should profile the app under realistic production loads.

Performance-specific configurations

After you understand the workload profile of your function app, the following are configurations that you can use to improve the throughput performance of your functions.

- [Async](#)
- [Multiple language worker](#)
- [Max workers within a language worker process](#)
- [Event loop](#)
- [Vertical Scaling](#)

Async

Because [Python is a single-threaded runtime](#), a host instance for Python can process only one function invocation at a time by default. For applications that process a large number of I/O events and/or is I/O bound, you can improve performance significantly by running functions asynchronously.

To run a function asynchronously, use the `async def` statement, which runs the function with [asyncio](#) directly:

Python

```
async def main():
    await some_nonblocking_socket_io_op()
```

Here's an example of a function with HTTP trigger that uses [aiohttp](#) http client:

Python

```
import aiohttp

import azure.functions as func

async def main(req: func.HttpRequest) -> func.HttpResponse:
    async with aiohttp.ClientSession() as client:
        async with client.get("PUT_YOUR_URL_HERE") as response:
            return func.HttpResponse(await response.text())

    return func.HttpResponse(body='NotFound', status_code=404)
```

A function without the `async` keyword is run automatically in a `ThreadPoolExecutor` thread pool:

Python

```
# Runs in a ThreadPoolExecutor threadpool. Number of threads is defined by
# PYTHON_THREADPOOL_THREAD_COUNT.
# The example is intended to show how default synchronous functions are
# handled.

def main():
    some_blocking_socket_io()
```

In order to achieve the full benefit of running functions asynchronously, the I/O operation/library that is used in your code needs to have `async` implemented as well. Using synchronous I/O operations in functions that are defined as asynchronous **may hurt** the overall performance. If the libraries you're using don't have `async` version implemented, you may still benefit from running your code asynchronously by [managing event loop](#) in your app.

Here are a few examples of client libraries that have implemented `async` patterns:

- [aiohttp](#) - Http client/server for `asyncio`
- [Streams API](#) - High-level `async/await`-ready primitives to work with network connection
- [Janus Queue](#) - Thread-safe `asyncio`-aware queue for Python
- [pyzmq](#) - Python bindings for ZeroMQ

Understanding `async` in Python worker

When you define `async` in front of a function signature, Python marks the function as a coroutine. When you call the coroutine, it can be scheduled as a task into an event loop. When you call `await` in an `async` function, it registers a continuation into the event loop, which allows the event loop to process the next task during the wait time.

In our Python Worker, the worker shares the event loop with the customer's `async` function and it's capable for handling multiple requests concurrently. We strongly encourage our customers to make use of `asyncio` compatible libraries, such as [aiohttp](#) and [pyzmq](#). Following these recommendations increases your function's throughput compared to those libraries when implemented synchronously.

ⓘ Note

If your function is declared as `async` without any `await` inside its implementation, the performance of your function will be severely impacted since the event loop will be blocked which prohibits the Python worker from handling concurrent requests.

Use multiple language worker processes

By default, every Functions host instance has a single language worker process. You can increase the number of worker processes per host (up to 10) by using the `FUNCTIONS_WORKER_PROCESS_COUNT` application setting. Azure Functions then tries to evenly distribute simultaneous function invocations across these workers.

For CPU bound apps, you should set the number of language workers to be the same as or higher than the number of cores that are available per function app. To learn more, see [Available instance SKUs](#).

I/O-bound apps may also benefit from increasing the number of worker processes beyond the number of cores available. Keep in mind that setting the number of workers too high can affect overall performance due to the increased number of required context switches.

The `FUNCTIONS_WORKER_PROCESS_COUNT` applies to each host that Azure Functions creates when scaling out your application to meet demand.

Note

Multiple Python workers are not supported by the Python v2 programming model at this time. This means that enabling intelligent concurrency and setting `FUNCTIONS_WORKER_PROCESS_COUNT` greater than 1 is not supported for functions developed using the v2 model.

Set up max workers within a language worker process

As mentioned in the async [section](#), the Python language worker treats functions and [coroutines](#) differently. A coroutine is run within the same event loop that the language worker runs on. On the other hand, a function invocation is run within a [ThreadPoolExecutor](#), which is maintained by the language worker as a thread.

You can set the value of maximum workers allowed for running sync functions using the `PYTHON_THREADPOOL_THREAD_COUNT` application setting. This value sets the `max_worker` argument of the `ThreadPoolExecutor` object, which lets Python use a pool of at most `max_worker` threads to execute calls asynchronously. The `PYTHON_THREADPOOL_THREAD_COUNT` applies to each worker that Functions host creates, and Python decides when to create a new thread or reuse the existing idle thread. For older Python versions(that is, 3.8, 3.7, and 3.6), `max_worker` value is set to 1. For Python version 3.9 , `max_worker` is set to `None`.

For CPU-bound apps, you should keep the setting to a low number, starting from 1 and increasing as you experiment with your workload. This suggestion is to reduce the time spent on context switches and allowing CPU-bound tasks to finish.

For I/O-bound apps, you should see substantial gains by increasing the number of threads working on each invocation. The recommendation is to start with the Python default (the number of cores) + 4 and then tweak based on the throughput values you're seeing.

For mixed workloads apps, you should balance both `FUNCTIONS_WORKER_PROCESS_COUNT` and `PYTHON_THREADPOOL_THREAD_COUNT` configurations to maximize the throughput. To understand what your function apps spend the most time on, we recommend profiling them and setting the values according to their behaviors. To learn about these application settings, see [Use multiple worker processes](#).

Note

Although these recommendations apply to both HTTP and non-HTTP triggered functions, you might need to adjust other trigger specific configurations for non-HTTP triggered functions to get the expected performance from your function apps. For more information about this, please refer to this [Best practices for reliable Azure Functions](#).

Managing event loop

You should use asyncio compatible third-party libraries. If none of the third-party libraries meet your needs, you can also manage the event loops in Azure Functions. Managing event loops give you more flexibility in compute resource management, and it also makes it possible to wrap synchronous I/O libraries into coroutines.

There are many useful Python official documents discussing the [Coroutines and Tasks ↗](#) and [Event Loop ↗](#) by using the built-in `asyncio` library.

Take the following `requests` library as an example, this code snippet uses the `asyncio` library to wrap the `requests.get()` method into a coroutine, running multiple web requests to SAMPLE_URL concurrently.

Python

```
import asyncio
import json
import logging
```

```

import azure.functions as func
from time import time
from requests import get, Response

async def invoke_get_request(eventloop: asyncio.AbstractEventLoop) ->
    Response:
    # Wrap requests.get function into a coroutine
    single_result = await eventloop.run_in_executor(
        None, # using the default executor
        get, # each task call invoke_get_request
        'SAMPLE_URL' # the url to be passed into the requests.get function
    )
    return single_result

async def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    eventloop = asyncio.get_event_loop()

    # Create 10 tasks for requests.get synchronous call
    tasks = [
        asyncio.create_task(
            invoke_get_request(eventloop)
        ) for _ in range(10)
    ]

    done_tasks, _ = await asyncio.wait(tasks)
    status_codes = [d.result().status_code for d in done_tasks]

    return func.HttpResponse(body=json.dumps(status_codes),
                            mimetype='application/json')

```

Vertical scaling

You might be able to get more processing units, especially in CPU-bound operation, by upgrading to premium plan with higher specifications. With higher processing units, you can adjust the number of worker processes count according to the number of cores available and achieve higher degree of parallelism.

Next steps

For more information about Azure Functions Python development, see the following resources:

- [Azure Functions Python developer guide](#)
- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)

Profile Python apps memory usage in Azure Functions

Article • 04/27/2023

During development or after deploying your local Python function app project to Azure, it's a good practice to analyze for potential memory bottlenecks in your functions. Such bottlenecks can decrease the performance of your functions and lead to errors. The following instructions show you how to use the [memory-profiler](#) Python package, which provides line-by-line memory consumption analysis of your functions as they execute.

ⓘ Note

Memory profiling is intended only for memory footprint analysis in development environments. Please do not apply the memory profiler on production function apps.

Prerequisites

Before you start developing a Python function app, you must meet these requirements:

- [Python 3.7 or above](#). To check the full list of supported Python versions in Azure Functions, see the [Python developer guide](#).
- The [Azure Functions Core Tools](#), version 4.x or greater. Check your version with `func --version`. To learn about updating, see [Azure Functions Core Tools on GitHub](#).
- [Visual Studio Code](#) installed on one of the [supported platforms](#).
- An active Azure subscription.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Memory profiling process

1. In your requirements.txt, add `memory-profiler` to ensure the package is bundled with your deployment. If you're developing on your local machine, you may want

to activate a Python virtual environment and do a package resolution by `pip`

```
install -r requirements.txt.
```

2. In your function script (for example, `__init__.py` for the Python v1 programming model and `function_app.py` for the v2 model), add the following lines above the `main()` function. These lines ensure the root logger reports the child logger names, so that the memory profiling logs are distinguishable by the prefix `memory_profiler_logs`.

Python

```
import logging
import memory_profiler
root_logger = logging.getLogger()
root_logger.handlers[0].setFormatter(logging.Formatter("%(name)s: %(message)s"))
profiler_logstream = memory_profiler.LogFile('memory_profiler_logs',
True)
```

3. Apply the following decorator above any functions that need memory profiling. The decorator doesn't work directly on the trigger entrypoint `main()` method. You need to create subfunctions and decorate them. Also, due to a memory-profiler known issue, when applying to an async coroutine, the coroutine return value is always `None`.

Python

```
@memory_profiler.profile(stream=profiler_logstream)
```

4. Test the memory profiler on your local machine by using Azure Functions Core Tools command `func host start`. When you invoke the functions, they should generate a memory usage report. The report contains file name, line of code, memory usage, memory increment, and the line content in it.
5. To check the memory profiling logs on an existing function app instance in Azure, you can query the memory profiling logs for recent invocations with [Kusto](#) queries in Application Insights, Logs.

The screenshot shows the Azure Application Insights Logs blade. On the left, there's a sidebar with various monitoring and diagnostic settings, including 'Logs' which is highlighted with a red box. The main area has a search bar and a 'New Query' button. A query editor window is open with the following Kusto query:

```

traces
| where timestamp > ago(1d)
| where message startswith_cs "memory_profiler_logs:"
| parse message with "memory_profiler_logs: LineNumber" " TotalMem_MiB" " IncreMem_MiB" " Occurrences" " Contents"
| union (
    traces
    | where timestamp > ago(1d)
    | where message startswith_cs "memory_profiler_logs: Filename:" 
    | parse message with "memory_profiler_logs: Filename:" "FileName"
    | project timestamp, FileName, itemId
)
| project timestamp, LineNumber=iff(FileName != "", FileName, LineNumber), TotalMem_MiB, IncreMem_MiB, Occurrences, Contents, RequestId=itemId
| order by timestamp asc

```

Below the query editor, there are tabs for 'Results' and 'Chart'. The 'Results' tab is selected, showing a table with the following data:

timestamp [UTC]	LineNumber	TotalMem_MiB	IncreMem_MiB	Occurrences	Contents	RequestId
3/26/2021, 12:24:22.233 AM						c09da4e8-8dc9-11
3/26/2021, 12:24:22.233 AM	/home/site/wwwroot/HttpTriggerAs...					c09da4e8-8dc9-11
3/26/2021, 12:24:22.244 AM	Line #	Mem usage	Increment	Occurrences	Line Contents	c09da4e9-8dc9-11
3/26/2021, 12:24:22.244 AM	23	49.2 MiB	0.4 MiB	5	await response.text()	c09da4eb-8dc9-11
3/26/2021, 12:24:22.244 AM	19	48.8 MiB	48.8 MiB	1	@memory_profiler.profile(stream=profiler_logstream)	c09da4ea-8dc9-11

Kusto

```

traces
| where timestamp > ago(1d)
| where message startswith_cs "memory_profiler_logs:"
| parse message with "memory_profiler_logs: LineNumber" " TotalMem_MiB" " IncreMem_MiB" " Occurrences" " Contents"
| union (
    traces
    | where timestamp > ago(1d)
    | where message startswith_cs "memory_profiler_logs: Filename:" 
    | parse message with "memory_profiler_logs: Filename:" "FileName"
    | project timestamp, FileName, itemId
)
| project timestamp, LineNumber=iff(FileName != "", FileName, LineNumber), TotalMem_MiB, IncreMem_MiB, Occurrences, Contents,
RequestId=itemId
| order by timestamp asc

```

Example

Here's an example of performing memory profiling on an asynchronous and a synchronous HTTP trigger, named "HttpTriggerAsync" and "HttpTriggerSync" respectively. We'll build a Python function app that simply sends out GET requests to the Microsoft's home page.

Create a Python function app

A Python function app should follow Azure Functions specified [folder structure](#). To scaffold the project, we recommend using the Azure Functions Core Tools by running the following commands:



Bash

```
func init PythonMemoryProfilingDemo --python
cd PythonMemoryProfilingDemo
func new -l python -t HttpTrigger -n HttpTriggerAsync -a anonymous
func new -l python -t HttpTrigger -n HttpTriggerSync -a anonymous
```

Update file contents

The `requirements.txt` defines the packages that are used in our project. Besides the Azure Functions SDK and memory-profiler, we introduce `aiohttp` for asynchronous HTTP requests and `requests` for synchronous HTTP calls.

text

```
# requirements.txt

azure-functions
memory-profiler
aiohttp
requests
```

Create the asynchronous HTTP trigger.

v1

Replace the code in the asynchronous HTTP trigger `HttpTriggerAsync/_init_.py` with the following code, which configures the memory profiler, root logger format, and logger streaming binding.

Python

```
# HttpTriggerAsync/_init__.py

import azure.functions as func
import aiohttp
import logging
import memory_profiler

# Update root logger's format to include the logger name. Ensure logs
generated
# from memory profiler can be filtered by "memory_profiler_logs" prefix.
root_logger = logging.getLogger()
root_logger.handlers[0].setFormatter(logging.Formatter("%(name)s: %
(message)s"))
```

```
profiler_logstream = memory_profiler.LogFile('memory_profiler_logs',  
True)  
  
async def main(req: func.HttpRequest) -> func.HttpResponse:  
    await get_microsoft_page_async('https://microsoft.com')  
    return func.HttpResponse(  
        f"Microsoft page loaded.",  
        status_code=200  
    )  
  
@memory_profiler.profile(stream=profiler_logstream)  
async def get_microsoft_page_async(url: str):  
    async with aiohttp.ClientSession() as client:  
        async with client.get(url) as response:  
            await response.text()  
    # @memory_profiler.profile does not support return for coroutines.  
    # All returns become None in the parent functions.  
    # GitHub Issue:  
    # https://github.com/pythonprofilers/memory_profiler/issues/289
```

Create the synchronous HTTP trigger.

v1

Replace the code in the asynchronous HTTP trigger *HttpTriggerSync/_init__.py* with the following code.

Python

```
# HttpTriggerSync/_init__.py  
  
import azure.functions as func  
import requests  
import logging  
import memory_profiler  
  
# Update root logger's format to include the logger name. Ensure logs  
generated  
# from memory profiler can be filtered by "memory_profiler_logs" prefix.  
root_logger = logging.getLogger()  
root_logger.handlers[0].setFormatter(logging.Formatter("%(name)s: %  
(message)s"))  
profiler_logstream = memory_profiler.LogFile('memory_profiler_logs',  
True)  
  
def main(req: func.HttpRequest) -> func.HttpResponse:  
    content = profile_get_request('https://microsoft.com')  
    return func.HttpResponse(  
        f"Microsoft page response size: {len(content)}",  
        status_code=200  
    )
```

```
@memory_profiler.profile(stream=profiler_logstream)
def profile_get_request(url: str):
    response = requests.get(url)
    return response.content
```

Profile Python function app in local development environment

After you make the above changes, there are a few more steps to initialize a Python virtual environment for Azure Functions runtime.

1. Open a Windows PowerShell or any Linux shell as you prefer.
2. Create a Python virtual environment by `py -m venv .venv` in Windows, or `python3 -m venv .venv` in Linux.
3. Activate the Python virtual environment with `.venv\Scripts\Activate.ps1` in Windows PowerShell or `source .venv/bin/activate` in Linux shell.
4. Restore the Python dependencies with `pip install -r requirements.txt`
5. Start the Azure Functions runtime locally with Azure Functions Core Tools `func host start`
6. Send a GET request to `https://localhost:7071/api/HttpTriggerAsync` or `https://localhost:7071/api/HttpTriggerSync`.
7. It should show a memory profiling report similar to the following section in Azure Functions Core Tools.

text

```
Filename: <ProjectRoot>\HttpTriggerAsync\__init__.py
Line #      Mem usage     Increment  Occurrences   Line Contents
=====
19          45.1 MiB       45.1 MiB           1   @memory_profiler.profile
20                                async def
get_microsoft_page_async(url: str):
21          45.1 MiB       0.0 MiB           1       async with
aiohttp.ClientSession() as client:
22          46.6 MiB       1.5 MiB          10       async with
client.get(url) as response:
23          47.6 MiB       1.0 MiB           4       await
response.text()
```

Next steps

For more information about Azure Functions Python development, see the following resources:

- [Azure Functions Python developer guide](#)
- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)

Microsoft.Web sites/functions 2022-03-01

Article • 09/01/2023

Bicep resource definition

The sites/functions resource type can be deployed with operations that target:

- **Resource groups** - See [resource group deployment commands](#)

For a list of changed properties in each API version, see [change log](#).

Resource format

To create a Microsoft.Web/sites/functions resource, add the following Bicep to your template.

Bicep

```
resource symbolicname 'Microsoft.Web/sites/functions@2022-03-01' = {
  name: 'string'
  kind: 'string'
  parent: resourceSymbolicName
  properties: {
    config: any()
    config_href: 'string'
    files: {}
    function_app_id: 'string'
    href: 'string'
    invoke_url_template: 'string'
    isDisabled: bool
    language: 'string'
    script_href: 'string'
    script_root_path_href: 'string'
    secrets_file_href: 'string'
    test_data: 'string'
    test_data_href: 'string'
  }
}
```

Property values

sites/functions

Name	Description	Value
name	The resource name See how to set names and types for child resources in Bicep .	string (required)
kind	Kind of resource.	string
parent	In Bicep, you can specify the parent resource for a child resource. You only need to add this property when the child resource is declared outside of the parent resource. For more information, see Child resource outside parent resource .	Symbolic name for resource of type: sites
properties	FunctionEnvelope resource specific properties	FunctionEnvelopeProperties

FunctionEnvelopeProperties

Name	Description	Value
config	Config information.	For Bicep, you can use the any() function.
config_href	Config URI.	string
files	File list.	object
function_app_id	Function App ID.	string
href	Function URI.	string
invoke_url_template	The invocation URL	string
isDisabled	Gets or sets a value indicating whether the function is disabled	bool
language	The function language	string
script_href	Script URI.	string
script_root_path_href	Script root path URI.	string
secrets_file_href	Secrets file URI.	string

Name	Description	Value
test_data	Test data used when testing via the Azure Portal.	string
test_data_href	Test data URI.	string

Quickstart templates

The following quickstart templates deploy this resource type.

Template	Description
Front Door Premium with Azure Functions and Private Link ↗	This template creates a Front Door Premium and an Azure Functions app, and uses a private endpoint for Front Door to send traffic to the function app.
 Deploy to Azure ↗	
Front Door Standard/Premium with Azure Functions origin ↗	This template creates a Front Door Standard/Premium, an Azure Functions app, and configures the function app to validate that traffic has come through the Front Door origin.
 Deploy to Azure ↗	
Azure Function app and an HTTP-triggered function ↗	This example deploys an Azure Function app and an HTTP-triggered function inline in the template. It also deploys a Key Vault and populates a secret with the function app's host key.
 Deploy to Azure ↗	
Azure function with transform capabilites ↗	Creates a webhook based C# azure function with transform capabilites to use in logic apps integration scenarios
 Deploy to Azure ↗	

az functionapp

Reference

ⓘ Note

This command group has commands that are defined in both Azure CLI and at least one extension. Install each extension to benefit from its extended capabilities. [Learn more](#) about extensions.

Manage function apps. To install the Azure Functions Core tools see <https://github.com/Azure/azure-functions-core-tools>.

Commands

[] [Expand table](#)

Name	Description	Type	Status
az functionapp app	Commands to manage Azure Functions app.	Extension	Preview
az functionapp app up	Deploy to Azure Functions via GitHub actions.	Extension	Preview
az functionapp config	Configure a function app.	Core and Extension	GA
az functionapp config access-restriction	Methods that show, set, add, and remove access restrictions on a functionapp.	Core	GA
az functionapp config access-restriction add	Adds an Access Restriction to the function app.	Core	GA
az functionapp config access-restriction remove	Removes an Access Restriction from the functionapp.	Core	GA
az functionapp config access-restriction set	Sets if SCM site is using the same restrictions as the main site.	Core	GA
az functionapp config access-restriction show	Show Access Restriction settings for functionapp.	Core	GA
az functionapp config	Configure function app settings.	Core	GA

Name	Description	Type	Status
appsettings			
az functionapp config appsettings delete	Delete a function app's settings.	Core	GA
az functionapp config appsettings list	Show settings for a function app.	Core	GA
az functionapp config appsettings set	Update a function app's settings.	Core	GA
az functionapp config container	Manage an existing function app's container settings.	Core and Extension	GA
az functionapp config container delete	Delete an existing function app's container settings.	Core	GA
az functionapp config container set	Set an existing function app's container settings.	Core	GA
az functionapp config container set (appservice-kube extension)	Set an existing function app's container settings.	Extension	GA
az functionapp config container show	Get details of a function app's container settings.	Core	GA
az functionapp config hostname	Configure hostnames for a function app.	Core	GA
az functionapp config hostname add	Bind a hostname to a function app.	Core	GA
az functionapp config hostname delete	Unbind a hostname from a function app.	Core	GA
az functionapp config hostname get-external-ip	Get the external-facing IP address for a function app.	Core	GA
az functionapp config hostname list	List all hostname bindings for a function app.	Core	GA
az functionapp config set	Set an existing function app's configuration.	Core	GA
az functionapp config show	Get the details of an existing function app's configuration.	Core	GA
az functionapp config ssl	Configure SSL certificates.	Core	GA

Name	Description	Type	Status
az functionapp config ssl bind	Bind an SSL certificate to a function app.	Core	GA
az functionapp config ssl create	Create a Managed Certificate for a hostname in a function app.	Core	Preview
az functionapp config ssl delete	Delete an SSL certificate from a function app.	Core	GA
az functionapp config ssl import	Import an SSL certificate to a function app from Key Vault.	Core	GA
az functionapp config ssl list	List SSL certificates for a function app.	Core	GA
az functionapp config ssl show	Show the details of an SSL certificate for a function app.	Core	GA
az functionapp config ssl unbind	Unbind an SSL certificate from a function app.	Core	GA
az functionapp config ssl upload	Upload an SSL certificate to a function app.	Core	GA
az functionapp connection	Commands to manage functionapp connections.	Core and Extension	GA
az functionapp connection create	Create a connection between a functionapp and a target resource.	Core and Extension	GA
az functionapp connection create app-insights	Create a functionapp connection to app-insights.	Core	GA
az functionapp connection create appconfig	Create a functionapp connection to appconfig.	Core	GA
az functionapp connection create cognitiveservices	Create a functionapp connection to cognitiveservices.	Core	GA
az functionapp connection create confluent-cloud	Create a functionapp connection to confluent-cloud.	Core	GA
az functionapp connection create cosmos-cassandra	Create a functionapp connection to cosmos-cassandra.	Core	GA
az functionapp connection create cosmos-gremlin	Create a functionapp connection to cosmos-gremlin.	Core	GA
az functionapp connection create cosmos-mongo	Create a functionapp connection to cosmos-mongo.	Core	GA

Name	Description	Type	Status
az functionapp connection create cosmos-sql	Create a functionapp connection to cosmos-sql.	Core	GA
az functionapp connection create cosmos-table	Create a functionapp connection to cosmos-table.	Core	GA
az functionapp connection create eventhub	Create a functionapp connection to eventhub.	Core	GA
az functionapp connection create keyvault	Create a functionapp connection to keyvault.	Core	GA
az functionapp connection create mysql	Create a functionapp connection to mysql.	Core	Deprecated
az functionapp connection create mysql-flexible	Create a functionapp connection to mysql-flexible.	Core	GA
az functionapp connection create mysql-flexible (serviceconnector-passwordless extension)	Create a functionapp connection to mysql-flexible.	Extension	GA
az functionapp connection create postgres	Create a functionapp connection to postgres.	Core	Deprecated
az functionapp connection create postgres-flexible	Create a functionapp connection to postgres-flexible.	Core	GA
az functionapp connection create postgres-flexible (serviceconnector-passwordless extension)	Create a functionapp connection to postgres-flexible.	Extension	GA
az functionapp connection create redis	Create a functionapp connection to redis.	Core	GA
az functionapp connection create redis-enterprise	Create a functionapp connection to redis-enterprise.	Core	GA
az functionapp connection create servicebus	Create a functionapp connection to servicebus.	Core	GA
az functionapp connection create signalr	Create a functionapp connection to signalr.	Core	GA
az functionapp connection create sql	Create a functionapp connection to sql.	Core	GA

Name	Description	Type	Status
az functionapp connection create sql (serviceconnector-passwordless extension)	Create a functionapp connection to sql.	Extension	GA
az functionapp connection create storage-blob	Create a functionapp connection to storage-blob.	Core	GA
az functionapp connection create storage-file	Create a functionapp connection to storage-file.	Core	GA
az functionapp connection create storage-queue	Create a functionapp connection to storage-queue.	Core	GA
az functionapp connection create storage-table	Create a functionapp connection to storage-table.	Core	GA
az functionapp connection create webpubsub	Create a functionapp connection to webpubsub.	Core	GA
az functionapp connection delete	Delete a functionapp connection.	Core	GA
az functionapp connection list	List connections of a functionapp.	Core	GA
az functionapp connection list-configuration	List source configurations of a functionapp connection.	Core	GA
az functionapp connection list-support-types	List client types and auth types supported by functionapp connections.	Core	GA
az functionapp connection show	Get the details of a functionapp connection.	Core	GA
az functionapp connection update	Update a functionapp connection.	Core	GA
az functionapp connection update app-insights	Update a functionapp to app-insights connection.	Core	GA
az functionapp connection update appconfig	Update a functionapp to appconfig connection.	Core	GA
az functionapp connection update cognitiveservices	Update a functionapp to cognitiveservices connection.	Core	GA
az functionapp connection update confluent-cloud	Update a functionapp to confluent-cloud connection.	Core	GA

Name	Description	Type	Status
az functionapp connection update cosmos-cassandra	Update a functionapp to cosmos-cassandra connection.	Core	GA
az functionapp connection update cosmos-gremlin	Update a functionapp to cosmos-gremlin connection.	Core	GA
az functionapp connection update cosmos-mongo	Update a functionapp to cosmos-mongo connection.	Core	GA
az functionapp connection update cosmos-sql	Update a functionapp to cosmos-sql connection.	Core	GA
az functionapp connection update cosmos-table	Update a functionapp to cosmos-table connection.	Core	GA
az functionapp connection update eventhub	Update a functionapp to eventhub connection.	Core	GA
az functionapp connection update keyvault	Update a functionapp to keyvault connection.	Core	GA
az functionapp connection update mysql	Update a functionapp to mysql connection.	Core	Deprecated
az functionapp connection update mysql-flexible	Update a functionapp to mysql-flexible connection.	Core	GA
az functionapp connection update postgres	Update a functionapp to postgres connection.	Core	Deprecated
az functionapp connection update postgres-flexible	Update a functionapp to postgres-flexible connection.	Core	GA
az functionapp connection update redis	Update a functionapp to redis connection.	Core	GA
az functionapp connection update redis-enterprise	Update a functionapp to redis-enterprise connection.	Core	GA
az functionapp connection update servicebus	Update a functionapp to servicebus connection.	Core	GA
az functionapp connection update signalr	Update a functionapp to signalr connection.	Core	GA
az functionapp connection update sql	Update a functionapp to sql connection.	Core	GA
az functionapp connection update storage-blob	Update a functionapp to storage-blob connection.	Core	GA

Name	Description	Type	Status
az functionapp connection update storage-file	Update a functionapp to storage-file connection.	Core	GA
az functionapp connection update storage-queue	Update a functionapp to storage-queue connection.	Core	GA
az functionapp connection update storage-table	Update a functionapp to storage-table connection.	Core	GA
az functionapp connection update webpubsub	Update a functionapp to webpubsub connection.	Core	GA
az functionapp connection validate	Validate a functionapp connection.	Core	GA
az functionapp connection wait	Place the CLI in a waiting state until a condition of the connection is met.	Core	GA
az functionapp cors	Manage Cross-Origin Resource Sharing (CORS).	Core	GA
az functionapp cors add	Add allowed origins.	Core	GA
az functionapp cors credentials	Enable or disable access-control-allow-credentials.	Core	GA
az functionapp cors remove	Remove allowed origins.	Core	GA
az functionapp cors show	Show allowed origins.	Core	GA
az functionapp create	Create a function app.	Core	GA
az functionapp create (appservice-kube extension)	Create a function app.	Extension	GA
az functionapp delete	Delete a function app.	Core	GA
az functionapp deploy	Deploys a provided artifact to Azure functionapp.	Core	Preview
az functionapp deployment	Manage function app deployments.	Core and Extension	GA
az functionapp deployment config	Manage a function app's deployment configuration.	Core	GA
az functionapp deployment config set	Update an existing function app's deployment configuration.	Core	GA
az functionapp deployment	Get the details of a function app's	Core	GA

Name	Description	Type	Status
config show	deployment configuration.		
az functionapp deployment container	Manage container-based continuous deployment.	Core	GA
az functionapp deployment container config	Configure continuous deployment via containers.	Core	GA
az functionapp deployment container show-cd-url	Get the URL which can be used to configure webhooks for continuous deployment.	Core	GA
az functionapp deployment github-actions	Configure GitHub Actions for a functionapp.	Core	GA
az functionapp deployment github-actions add	Add a GitHub Actions workflow file to the specified repository. The workflow will build and deploy your app to the specified functionapp.	Core	GA
az functionapp deployment github-actions remove	Remove and disconnect the GitHub Actions workflow file from the specified repository.	Core	GA
az functionapp deployment list-publishing-credentials	Get the details for available function app publishing credentials.	Core	GA
az functionapp deployment list-publishing-profiles	Get the details for available function app deployment profiles.	Core	GA
az functionapp deployment slot	Manage function app deployment slots.	Core	GA
az functionapp deployment slot auto-swap	Configure deployment slot auto swap.	Core	GA
az functionapp deployment slot create	Create a deployment slot.	Core	GA
az functionapp deployment slot delete	Delete a deployment slot.	Core	GA
az functionapp deployment slot list	List all deployment slots.	Core	GA
az functionapp deployment slot swap	Swap deployment slots for a function app.	Core	GA
az functionapp deployment source	Manage function app deployment via source control.	Core and Extension	GA

Name	Description	Type	Status
az functionapp deployment source config	Manage deployment from git or Mercurial repositories.	Core	GA
az functionapp deployment source config-local-git	Get a URL for a git repository endpoint to clone and push to for function app deployment.	Core	GA
az functionapp deployment source config-zip	Perform deployment using the kudu zip push deployment for a function app.	Core	GA
az functionapp deployment source config-zip (appservice-kube extension)	Perform deployment using the kudu zip push deployment for a function app.	Extension	GA
az functionapp deployment source delete	Delete a source control deployment configuration.	Core	GA
az functionapp deployment source show	Get the details of a source control deployment configuration.	Core	GA
az functionapp deployment source sync	Synchronize from the repository. Only needed under manual integration mode.	Core	GA
az functionapp deployment source update-token	Update source control token cached in Azure app service.	Core	GA
az functionapp deployment user	Manage user credentials for deployment.	Core	GA
az functionapp deployment user set	Update deployment credentials.	Core	GA
az functionapp deployment user show	Gets publishing user.	Core	GA
az functionapp devops-pipeline	Azure Function specific integration with Azure DevOps. Please visit https://aka.ms/functions-azure-devops for more information.	Extension	GA
az functionapp devops-pipeline create	Create an Azure DevOps pipeline for a function app.	Extension	GA
az functionapp function	Manage function app functions.	Core	GA
az functionapp function delete	Delete a function.	Core	GA

Name	Description	Type	Status
az functionapp function keys	Manage function keys.	Core	GA
az functionapp function keys delete	Delete a function key.	Core	GA
az functionapp function keys list	List all function keys.	Core	GA
az functionapp function keys set	Create or update a function key.	Core	GA
az functionapp function list	List functions in a function app.	Core	GA
az functionapp function show	Get the details of a function.	Core	GA
az functionapp hybrid-connection	Methods that list, add and remove hybrid-connections from functionapp.	Core	GA
az functionapp hybrid-connection add	Add an existing hybrid-connection to a functionapp.	Core	GA
az functionapp hybrid-connection list	List the hybrid-connections on a functionapp.	Core	GA
az functionapp hybrid-connection remove	Remove a hybrid-connection from a functionapp.	Core	GA
az functionapp identity	Manage web app's managed identity.	Core	GA
az functionapp identity assign	Assign managed identity to the web app.	Core	GA
az functionapp identity remove	Disable web app's managed identity.	Core	GA
az functionapp identity show	Display web app's managed identity.	Core	GA
az functionapp keys	Manage function app keys.	Core	GA
az functionapp keys delete	Delete a function app key.	Core	GA
az functionapp keys list	List all function app keys.	Core	GA
az functionapp keys set	Create or update a function app key.	Core	GA
az functionapp list	List function apps.	Core	GA

Name	Description	Type	Status
az functionapp list-consumption-locations	List available locations for running function apps.	Core	GA
az functionapp list-flexconsumption-locations	List available locations for running function apps on the Flex Consumption plan.	Core	GA
az functionapp list-flexconsumption-runtimes	List available built-in stacks which can be used for function apps on the Flex Consumption plan.	Core	GA
az functionapp list-runtimes	List available built-in stacks which can be used for function apps.	Core	GA
az functionapp log	Manage function app logs.	Core	GA
az functionapp log deployment	Manage function app deployment logs.	Core	GA
az functionapp log deployment list	List deployment logs of the deployments associated with function app.	Core	GA
az functionapp log deployment show	Show deployment logs of the latest deployment, or a specific deployment if deployment-id is specified.	Core	GA
az functionapp plan	Manage App Service Plans for an Azure Function.	Core	GA
az functionapp plan create	Create an App Service Plan for an Azure Function.	Core	GA
az functionapp plan delete	Delete an App Service Plan.	Core	GA
az functionapp plan list	List App Service Plans.	Core	GA
az functionapp plan show	Get the App Service Plans for a resource group or a set of resource groups.	Core	GA
az functionapp plan update	Update an App Service plan for an Azure Function.	Core	GA
az functionapp restart	Restart a function app.	Core	GA
az functionapp restart (appservice-kube extension)	Restart a function app.	Extension	GA
az functionapp runtime	Manage a function app's runtime.	Core	GA

Name	Description	Type	Status
az functionapp runtime config	Manage a function app's runtime configuration.	Core	GA
az functionapp runtime config set	Update an existing function app's runtime configuration.	Core	GA
az functionapp runtime config show	Get the details of a function app's runtime configuration.	Core	GA
az functionapp scale	Manage a function app's scale.	Core	GA
az functionapp scale config	Manage a function app's scale configuration.	Core	GA
az functionapp scale config always-ready	Manage the always-ready settings in the scale configuration.	Core	GA
az functionapp scale config always-ready delete	Delete always-ready settings in the scale configuration.	Core	GA
az functionapp scale config always-ready set	Add or update existing always-ready settings in the scale configuration.	Core	GA
az functionapp scale config set	Update an existing function app's scale configuration.	Core	GA
az functionapp scale config show	Get the details of a function app's scale configuration.	Core	GA
az functionapp show	Get the details of a function app.	Core	GA
az functionapp show (appservice-kube extension)	Get the details of a function app.	Extension	GA
az functionapp start	Start a function app.	Core	GA
az functionapp stop	Stop a function app.	Core	GA
az functionapp update	Update a function app.	Core	GA
az functionapp vnet-integration	Methods that list, add, and remove virtual networks integrations from a functionapp.	Core	GA
az functionapp vnet-integration add	Add a regional virtual network integration to a functionapp.	Core	GA
az functionapp vnet-integration list	List the virtual network integrations on a functionapp.	Core	GA

Name	Description	Type	Status
az functionapp vnet-integration remove	Remove a regional virtual network integration from functionapp.	Core	GA

az functionapp create

[!\[\]\(edf67e01c1d073f3d30f2b69f8274f38_img.jpg\) Edit](#)

Create a function app.

The function app's name must be able to produce a unique FQDN as `AppName.azurewebsites.net`.

Azure CLI

```
az functionapp create --name
    --resource-group
    --storage-account
    [--always-ready-instances]
    [--app-insights]
    [--app-insights-key]
    [--assign-identity]
    [--consumption-plan-location]
    [--cpu]
    [--dal {false, true}]
    [--dapr-app-id]
    [--dapr-app-port]
    [--dapr-http-max-request-size]
    [--dapr-http-read-buffer-size]
    [--dapr-log-level {debug, error, info, warn}]
    [--deployment-container-image-name]
    [--deployment-local-git]
    [--deployment-source-branch]
    [--deployment-source-url]
    [--deployment-storage-auth-type
{StorageAccountConnectionString, SystemAssignedIdentity,
UserAssignedIdentity}]
    [--deployment-storage-auth-value]
    [--deployment-storage-container-name]
    [--deployment-storage-name]
    [--disable-app-insights {false, true}]
    [--docker-registry-server-password]
    [--docker-registry-server-user]
    [--enable-dapr {false, true}]
    [--environment]
    [--flexconsumption-location]
    [--functions-version {4}]
    [--https-only {false, true}]
    [--image]
```

```
[--instance-memory]
[--max-replicas]
[--maximum-instance-count]
[--memory]
[--min-replicas]
[--os-type {Linux, Windows}]
[--plan]
[--registry-password]
[--registry-server]
[--registry-username]
[--role]
[--runtime]
[--runtime-version]
[--scope]
[--subnet]
[--tags]
[--vnet]
[--workload-profile-name]
[--workspace]
```

Examples

Create a basic function app.

Azure CLI

```
az functionapp create -g MyResourceGroup -p MyPlan -n MyUniqueAppName -s
MyStorageAccount
```

Create a function app. (autogenerated)

Azure CLI

```
az functionapp create --consumption-plan-location westus --name
MyUniqueAppName --os-type Windows --resource-group MyResourceGroup --runtime
dotnet --storage-account MyStorageAccount
```

Create a function app using a private ACR image.

Azure CLI

```
az functionapp create -g MyResourceGroup -p MyPlan -n MyUniqueAppName --
runtime node --storage-account MyStorageAccount --deployment-container-
image-name myacr.azurecr.io/myimage:tag --docker-registry-server-password
passw0rd --docker-registry-server-user MyUser
```

Create a flex consumption function app. See <https://aka.ms/flex-http-concurrency> for more information on default http concurrency values.

Azure CLI

```
az functionapp create -g MyResourceGroup --name MyUniqueAppName -s  
MyStorageAccount --flexconsumption-location northeurope --runtime java --  
instance-memory 2048
```

Required Parameters

--name -n

Name of the new function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--storage-account -s

Provide a string value of a Storage Account in the provided Resource Group. Or Resource ID of a Storage Account in a different Resource Group.

Optional Parameters

--always-ready-instances

Preview

Space-separated configuration for the number of pre-allocated instances in the format `<name>=<value>`.

--app-insights

Name of the existing App Insights project to be added to the function app. Must be in the same resource group.

--app-insights-key

Instrumentation key of App Insights to be added.

--assign-identity

Accept system or user assigned identities separated by spaces. Use '[system]' to refer system assigned identity, or a resource id to refer user assigned identity. Check out help for more examples.

--consumption-plan-location -c

Geographic location where function app will be hosted. Use `az functionapp list-consumption-locations` to view available locations.

--cpu

Preview

The CPU in cores of the container app. e.g 0.75.

--dal --dapr-enable-api-logging

Enable/Disable API logging for the Dapr sidecar.

Accepted values: false, true

Default value: False

--dapr-app-id

The Dapr application identifier.

--dapr-app-port

The port Dapr uses to communicate to the application.

--dapr-http-max-request-size --dhmrs

Max size of request body http and grpc servers in MB to handle uploading of large files.

--dapr-http-read-buffer-size --dhrbs

Max size of http header read buffer in KB to handle when sending multi-KB headers.

--dapr-log-level

The log level for the Dapr sidecar.

Accepted values: debug, error, info, warn

--deployment-container-image-name

Deprecated

Option '--deployment-container-image-name' has been deprecated and will be removed in a future release. Use '--image' instead.

Container image, e.g. publisher/image-name:tag.

--deployment-local-git -l

Enable local git.

--deployment-source-branch -b

The branch to deploy.

--deployment-source-url -u

Git repository URL to link with manual integration.

--deployment-storage-auth-type --dsat

[Preview](#)

The deployment storage account authentication type.

Accepted values: StorageAccountConnectionString, SystemAssignedIdentity, UserAssignedIdentity

--deployment-storage-auth-value --dsav

[Preview](#)

The deployment storage account authentication value. For the user-assigned managed identity authentication type, this should be the user assigned identity resource id. For the storage account connection string authentication type, this should be the name of the app setting that will contain the storage account connection string. For the system assigned managed-identity authentication type, this parameter is not applicable and should be left empty.

--deployment-storage-container-name --dscn

[Preview](#)

The deployment storage account container name.

--deployment-storage-name --dsn

[Preview](#)

The deployment storage account name.

--disable-app-insights

Disable creating application insights resource during functionapp create. No logs will be available.

Accepted values: false, true

--docker-registry-server-password Deprecated

Option '--docker-registry-server-password' has been deprecated and will be removed in a future release. Use '--registry-password' instead.

The container registry server password. Required for private registries.

--docker-registry-server-user Deprecated

Option '--docker-registry-server-user' has been deprecated and will be removed in a future release. Use '--registry-username' instead.

The container registry server username.

--enable-dapr

Enable/Disable Dapr for a function app on an Azure Container App environment.

Accepted values: false, true

Default value: False

--environment Preview

Name of the container app environment.

--flexconsumption-location -f Preview

Geographic location where function app will be hosted. Use `az functionapp list-f
flexconsumption-locations` to view available locations.

--functions-version

The functions app version. NOTE: This will be required starting the next release cycle.

Accepted values: 4

--https-only

Redirect all traffic made to an app using HTTP to HTTPS.

Accepted values: false, true

Default value: False

--image -i

Container image, e.g. publisher/image-name:tag.

--instance-memory Preview

The instance memory size in MB. See <https://aka.ms/flex-instance-sizes> for more information on the supported values.

--max-replicas Preview

The maximum number of replicas when create function app on container app.

--maximum-instance-count Preview

The maximum number of instances.

--memory Preview

The memory size of the container app. e.g. 1.0Gi,.

--min-replicas Preview

The minimum number of replicas when create function app on container app.

--os-type

Set the OS type for the app to be created.

Accepted values: Linux, Windows

--plan -p

Name or resource id of the functionapp app service plan. Use 'appservice plan create' to get one. If using an App Service plan from a different resource group, the full resource id must be used and not the plan name.

--registry-password -w

The container registry server password. Required for private registries.

--registry-server Preview

The container registry server hostname, e.g. myregistry.azurecr.io.

--registry-username -d

The container registry server username.

--role

Role name or id the system assigned identity will have.

Default value: Contributor

--runtime

The functions runtime stack. Use "az functionapp list-runtimes" to check supported runtimes and versions.

--runtime-version

The version of the functions runtime stack. The functions runtime stack. Use "az functionapp list-runtimes" to check supported runtimes and versions.

--scope

Scope that the system assigned identity can access.

--subnet

Name or resource ID of the pre-existing subnet to have the webapp join. The --vnet is argument also needed if specifying subnet by name.

--tags

Space-separated tags: key[=value] [key[=value] ...]. Use "" to clear existing tags.

--vnet

Name or resource ID of the regional virtual network. If there are multiple vnets of the same name across different resource groups, use vnet resource id to specify which vnet to use. If vnet name is used, by default, the vnet in the same resource group as the webapp will be used. Must be used with --subnet argument.

--workload-profile-name

[Preview](#)

The workload profile name to run the container app on.

--workspace

Name of an existing log analytics workspace to be used for the application insights component.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp create (appservice-kube extension)

Create a function app.

The function app's name must be able to produce a unique FQDN as `AppName.azurewebsites.net`.

Azure CLI

```
az functionapp create --name
    --resource-group
    [--app-insights]
    [--app-insights-key]
    [--assign-identity]
    [--consumption-plan-location]
    [--custom-location]
    [--deployment-container-image-name]
    [--deployment-local-git]
    [--deployment-source-branch]
    [--deployment-source-url]
    [--disable-app-insights {false, true}]
    [--docker-registry-server-password]
    [--docker-registry-server-user]
    [--functions-version {4}]
    [--max-worker-count]
    [--min-worker-count]
    [--os-type {Linux, Windows}]
    [--plan]
    [--role]
    [--runtime]
    [--runtime-version]
    [--scope]
    [--storage-account]
    [--tags]
```

Examples

Create a basic function app.

Azure CLI

```
az functionapp create -g MyResourceGroup -p MyPlan -n MyUniqueAppName -s
MyStorageAccount
```

Create a function app. (autogenerated)

Azure CLI

```
az functionapp create --consumption-plan-location westus --name MyUniqueAppName --os-type Windows --resource-group MyResourceGroup --runtime dotnet --storage-account MyStorageAccount
```

Create a function app using a private ACR image.

Azure CLI

```
az functionapp create -g MyResourceGroup -p MyPlan -n MyUniqueAppName --runtime node --storage-account MyStorageAccount --deployment-container-image-name myacr.azurecr.io/myimage:tag --docker-registry-server-password passw0rd --docker-registry-server-user MyUser
```

Create a function app in an app service kubernetes environment

Azure CLI

```
az functionapp create -g MyResourceGroup -p MyPlan -n MyUniqueAppName -s MyStorageAccount --custom-location /subscriptions/sub_id/resourcegroups/group_name/providers/microsoft.extendedlocation/customlocations/custom_location_name
```

Create a function app in an app service kubernetes environment and in the same resource group as the custom location

Azure CLI

```
az functionapp create -g MyResourceGroup -p MyPlan -n MyUniqueAppName -s MyStorageAccount --custom-location custom_location_name
```

Required Parameters

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

Optional Parameters

--app-insights

Name of the existing App Insights project to be added to the Function app. Must be in the same resource group.

--app-insights-key

Instrumentation key of App Insights to be added.

--assign-identity

Accept system or user assigned identities separated by spaces. Use '[system]' to refer system assigned identity, or a resource id to refer user assigned identity. Check out help for more examples.

--consumption-plan-location -c

Geographic location where Function App will be hosted. Use `az functionapp list-consumption-locations` to view available locations.

--custom-location

Name or ID of the custom location. Use an ID for a custom location in a different resource group from the app.

--deployment-container-image-name -i

Linux only. Container image name from Docker Hub, e.g. publisher/image-name:tag.

--deployment-local-git -l

Enable local git.

--deployment-source-branch -b

The branch to deploy.

Default value: master

--deployment-source-url -u

Git repository URL to link with manual integration.

--disable-app-insights

Disable creating application insights resource during functionapp create. No logs will be available.

Accepted values: false, true

--docker-registry-server-password

The container registry server password. Required for private registries.

--docker-registry-server-user

The container registry server username.

--functions-version

The functions app version. Use "az functionapp list-runtimes" to check compatibility with runtimes and runtime versions.

Accepted values: 4

--max-worker-count Preview

Maximum number of workers to be allocated.

--min-worker-count Preview

Minimum number of workers to be allocated.

--os-type

Set the OS type for the app to be created.

Accepted values: Linux, Windows

--plan -p

Name or resource id of the function app service plan. Use 'appservice plan create' to get one.

--role

Role name or id the system assigned identity will have.

Default value: Contributor

--runtime

The functions runtime stack. Use "az functionapp list-runtimes" to check supported runtimes and versions.

--runtime-version

The version of the functions runtime stack. Use "az functionapp list-runtimes" to check supported runtimes and versions.

--scope

Scope that the system assigned identity can access.

--storage-account -s

Provide a string value of a Storage Account in the provided Resource Group. Or Resource ID of a Storage Account in a different Resource Group. Required for non-kubernetes function apps.

--tags

Space-separated tags: key[=value] [key[=value] ...]. Use "" to clear existing tags.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use `--debug` for full debug logs.

az functionapp delete

 Edit

Delete a function app.

Azure CLI

```
az functionapp delete [--ids]
                      [--keep-empty-plan]
                      [--name]
                      [--resource-group]
                      [--slot]
                      [--subscription]
```

Examples

Delete a function app. (autogenerated)

Azure CLI

```
az functionapp delete --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--keep-empty-plan

Keep empty app service plan.

--name -n

The name of the functionapp.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the productions slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az`

```
account set -s NAME_OR_ID.
```

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp deploy

 Edit

[Preview](#)

This command is in preview and under development. Reference and support levels:

https://aka.ms/CLI_refstatus

Deploys a provided artifact to Azure functionapp.

Azure CLI

```
az functionapp deploy [--async {false, true}]  
                      [--clean {false, true}]  
                      [--ids]  
                      [--ignore-stack {false, true}]  
                      [--name]  
                      [--resource-group]  
                      [--restart {false, true}]  
                      [--slot]  
                      [--src-path]  
                      [--src-url]  
                      [--subscription]  
                      [--target-path]  
                      [--timeout]  
                      [--type {ear, jar, lib, startup, static, war, zip}]
```

Examples

Deploy a war file asynchronously.

Azure CLI

```
az functionapp deploy --resource-group ResourceGroup --name AppName --src-path SourcePath --type war --async true
```

Deploy a static text file to wwwroot/staticfiles/test.txt

Azure CLI

```
az functionapp deploy --resource-group ResourceGroup --name AppName --src-path SourcePath --type static --target-path staticfiles/test.txt
```

Optional Parameters

--async

Asynchronous deployment.

Accepted values: false, true

--clean

If true, cleans the target directory prior to deploying the file(s). Default value is determined based on artifact type.

Accepted values: false, true

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--ignore-stack

If true, any stack-specific defaults are ignored.

Accepted values: false, true

--name -n

Name of the function app to deploy to.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--restart

If true, the web app will be restarted following the deployment, default value is true. Set this to false if you are deploying multiple artifacts and do not want to restart the site on the earlier deployments.

Accepted values: false, true

--slot -s

The name of the slot. Default to the productions slot if not specified.

--src-path

Path of the artifact to be deployed. Ex: "myapp.zip" or "/myworkspace/apps/myapp.war".

--src-url

URL of the artifact. The webapp will pull the artifact from this URL. Ex: "http://mysite.com/files/myapp.war?key=123".

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--target-path

Absolute path that the artifact should be deployed to. Defaults to "home/site/wwwroot/". Ex: "/home/site/deployments/tools/", "/home/site/scripts/startup-script.sh".

--timeout

Timeout for the deployment operation in milliseconds.

--type

Used to override the type of artifact being deployed.

Accepted values: ear, jar, lib, startup, static, war, zip

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az`

`account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp list

 Edit

List function apps.

```
Azure CLI
```

```
az functionapp list [--resource-group]
```

Examples

List all function apps in MyResourceGroup.

```
Azure CLI
```

```
az functionapp list --resource-group MyResourceGroup
```

List default host name and state for all function apps.

```
Azure CLI
```

```
az functionapp list --query "[].{hostName: defaultHostName, state: state}"
```

List all running function apps.

```
Azure CLI
```

```
az functionapp list --query "[?state=='Running']"
```

Optional Parameters

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp list-consumption-locations Edit

List available locations for running function apps.

Azure CLI

```
az functionapp list-consumption-locations
```

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp list-flexconsumption-locations

 Edit

List available locations for running function apps on the Flex Consumption plan.

Azure CLI

```
az functionapp list-flexconsumption-locations
```

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp list-flexconsumption-runtimes

 Edit

List available built-in stacks which can be used for function apps on the Flex Consumption plan.

Azure CLI

```
az functionapp list-flexconsumption-runtimes --location  
                                --runtime
```

Required Parameters

--location -l

Limit the output to just the runtimes available in the specified location.

--runtime

Limit the output to just the specified runtime.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp list-runtimes

 Edit

List available built-in stacks which can be used for function apps.

Azure CLI

```
az functionapp list-runtimes [--os {linux, windows}]
```

Optional Parameters

--os --os-type

Limit the output to just windows or linux runtimes.

Accepted values: linux, windows

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use `--debug` for full debug logs.

az functionapp restart

 Edit

Restart a function app.

Azure CLI

```
az functionapp restart [--ids]
                      [--name]
                      [--resource-group]
                      [--slot]
                      [--subscription]
```

Examples

Restart a function app. (autogenerated)

Azure CLI

```
az functionapp restart --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either `--ids` or other 'Resource Id' arguments.

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the production slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az`

```
account set -s NAME_OR_ID.
```

--verbose

Increase logging verbosity. Use `--debug` for full debug logs.

az functionapp restart (appservice-kube extension)

Restart a function app.

Azure CLI

```
az functionapp restart [--ids]
                      [--name]
                      [--resource-group]
                      [--slot]
                      [--subscription]
```

Examples

Restart a function app. (autogenerated)

Azure CLI

```
az functionapp restart --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either `--ids` or other 'Resource Id' arguments.

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the productions slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az`

```
account set -s NAME_OR_ID.
```

--verbose

Increase logging verbosity. Use `--debug` for full debug logs.

az functionapp show

 Edit

Get the details of a function app.

Azure CLI

```
az functionapp show [--ids]
                   [--name]
                   [--resource-group]
                   [--slot]
                   [--subscription]
```

Examples

Get the details of a function app. (autogenerated)

Azure CLI

```
az functionapp show --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either `--ids` or other 'Resource Id' arguments.

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the production slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp show (appservice-kube extension)

Get the details of a function app.

Azure CLI

```
az functionapp show [--ids]
                    [--name]
                    [--resource-group]
                    [--slot]
                    [--subscription]
```

Examples

Get the details of a function app. (autogenerated)

Azure CLI

```
az functionapp show --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the production slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp start

 Edit

Start a function app.

Azure CLI

```
az functionapp start [--ids]
                     [--name]
                     [--resource-group]
                     [--slot]
                     [--subscription]
```

Examples

Start a function app. (autogenerated)

Azure CLI

```
az functionapp start --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the productions slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp stop

 Edit

Stop a function app.

Azure CLI

```
az functionapp stop [--ids]
                   [--name]
                   [--resource-group]
                   [--slot]
                   [--subscription]
```

Examples

Stop a function app. (autogenerated)

Azure CLI

```
az functionapp stop --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

Name of the function app.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--slot -s

The name of the slot. Default to the production slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az functionapp update

 Edit

Update a function app.

Azure CLI

```
az functionapp update [--add]
                      [--force]
                      [--force-string]
                      [--ids]
                      [--name]
                      [--plan]
                      [--remove]
                      [--resource-group]
                      [--set]
                      [--slot]
                      [--subscription]
```

Examples

Update a function app. (autogenerated)

Azure CLI

```
az functionapp update --name MyFunctionApp --resource-group MyResourceGroup
```

Optional Parameters

--add

Add an object to a list of objects by specifying a path and key value pairs. Example: `-`

```
-add property.listProperty <key=value, string or JSON string>.
```

Default value: []

--force

Required if attempting to migrate functionapp from Premium to Consumption --plan.

Default value: False

--force-string

When using 'set' or 'add', preserve string literals instead of attempting to convert to JSON.

Default value: False

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

Name of the function app.

--plan

The name or resource id of the plan to update the functionapp with.

--remove

Remove a property or an element from a list. Example: `--remove property.list <indexToRemove>` OR `--remove propertyToRemove`.

Default value: []

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--set

Update an object by specifying a property path and value to set. Example: `--set property1.property2=<value>`.

Default value: []

--slot -s

The name of the slot. Default to the productions slot if not specified.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

▼ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

Accepted values: json, jsonc, none, table, tsv, yaml, yamlc

Default value: json

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

Azure Functions Core Tools reference

Article • 08/23/2024

This article provides reference documentation for the Azure Functions Core Tools, which lets you develop, manage, and deploy Azure Functions projects from your local computer. To learn more about using Core Tools, see [Work with Azure Functions Core Tools](#).

Core Tools commands are organized into the following contexts, each providing a unique set of actions.

[+] Expand table

Command context	Description
<code>func</code>	Commands used to create and run functions on your local computer.
<code>func azure</code>	Commands for working with Azure resources, including publishing.
<code>func</code> <code>azurecontainerapps</code>	Deploy containerized function app to Azure Container Apps.
<code>func durable</code>	Commands for working with Durable Functions .
<code>func extensions</code>	Commands for installing and managing extensions.
<code>func kubernetes</code>	Commands for working with Kubernetes and Azure Functions.
<code>func settings</code>	Commands for managing environment settings for the local Functions host.
<code>func templates</code>	Commands for listing available function templates.

Before using the commands in this article, you must [install the Core Tools](#).

func init

Creates a new Functions project in a specific language.

```
command
```

```
func init <PROJECT_FOLDER>
```

When you supply `<PROJECT_FOLDER>`, the project is created in a new folder with this name. Otherwise, the current folder is used.

`func init` supports the following options, which don't support version 1.x unless otherwise noted:

[+] Expand table

Option	Description
<code>--csx</code>	Creates .NET functions as C# script, which is the version 1.x behavior. Valid only with <code>--worker-runtime dotnet</code> .
<code>--docker</code>	Creates a Dockerfile for a container using a base image that is based on the chosen <code>--worker-runtime</code> . Use this option when you plan to deploy a containerized function app.
<code>--docker-only</code>	Adds a Dockerfile to an existing project. Prompts for the worker-runtime if not specified or set in local.settings.json. Use this option when you plan to deploy a containerized function app and the project already exists.
<code>--force</code>	Initialize the project even when there are existing files in the project. This setting overwrites existing files with the same name. Other files in the project folder aren't affected.
<code>--language</code>	Initializes a language-specific project. Currently supported when <code>--worker-runtime</code> set to <code>node</code> . Options are <code>typescript</code> and <code>javascript</code> . You can also use <code>--worker-runtime javascript</code> or <code>--worker-runtime typescript</code> .
<code>--managed-dependencies</code>	Installs managed dependencies. Currently, only the PowerShell worker runtime supports this functionality.
<code>--model</code>	Sets the desired programming model for a target language when more than one model is available. Supported options are <code>v1</code> and <code>v2</code> for Python and <code>v3</code> and <code>v4</code> for Node.js. For more information, see the Python developer guide and the Node.js developer guide , respectively.
<code>--source-control</code>	Controls whether a git repository is created. By default, a repository isn't created. When <code>true</code> , a repository is created.
<code>--worker-runtime</code>	Sets the language runtime for the project. Supported values are: <code>csharp</code> , <code>dotnet</code> , <code>dotnet-isolated</code> , <code>javascript</code> , <code>node</code> (JavaScript), <code>powershell</code> , <code>python</code> , and <code>typescript</code> . For Java, use <code>Maven</code> . To generate a language-agnostic project with just the project files, use <code>custom</code> . When not set, you're prompted to choose your runtime during initialization.
<code>--target-framework</code>	Sets the target framework for the function app project. Valid only with <code>--worker-runtime dotnet-isolated</code> . Supported values are: <code>net6.0</code> (default), <code>net7.0</code> , <code>net8.0</code> , and <code>net48</code> (.NET Framework 4.8).

 Note

When you use either `--docker` or `--docker-only` options, Core Tools automatically create the Dockerfile for C#, JavaScript, Python, and PowerShell functions. For Java functions, you must manually create the Dockerfile. For more information, see [Creating containerized function apps](#).

func logs

Gets logs for functions running in a Kubernetes cluster.

```
func logs --platform kubernetes --name <APP_NAME>
```

The `func logs` action supports the following options:

[\[+\] Expand table](#)

Option	Description
<code>--platform</code>	Hosting platform for the function app. Supported options: <code>kubernetes</code> .
<code>--name</code>	Function app name in Azure.

To learn more, see [Azure Functions on Kubernetes with KEDA](#).

func new

Creates a new function in the current project based on a template.

```
func new
```

When you run `func new` without the `--template` option, you're prompted to choose a template. In version 1.x, you're also required to choose the language.

The `func new` action supports the following options:

[\[+\] Expand table](#)

Option	Description
-- <code>authlevel</code>	Lets you set the authorization level for an HTTP trigger. Supported values are: <code>function</code> , <code>anonymous</code> , <code>admin</code> . Authorization isn't enforced when running locally. For more information, see Authorization level .
-- <code>csx</code>	(Version 2.x and later versions.) Generates the same C# script (.csx) templates used in version 1.x and in the portal.
-- <code>language</code> , - l	The template programming language, such as C#, F#, or JavaScript. This option is required in version 1.x. In version 2.x and later versions, you don't use this option because the language is defined by the worker runtime.
-- <code>name</code> , -n	The function name.
-- <code>template</code> , - t	Use the <code>func templates list</code> command to see the complete list of available templates for each supported language.

To learn more, see [Create a function](#).

func run

Version 1.x only.

Enables you to invoke a function directly, which is similar to running a function using the **Test** tab in the Azure portal. This action is only supported in version 1.x. For later versions, use `func start` and [call the function endpoint directly](#).

```
command
func run
```

The `func run` action supports the following options:

[\[+\] Expand table](#)

Option	Description
-- <code>content</code>	Inline content passed to the function.
-- <code>debug</code>	Attach a debugger to the host process before running the function.
-- <code>file</code>	The file name to use as content.
-- <code>no-interactive</code>	Doesn't prompt for input, which is useful for automation scenarios.

Option	Description
--timeout	Time to wait (in seconds) until the local Functions host is ready.

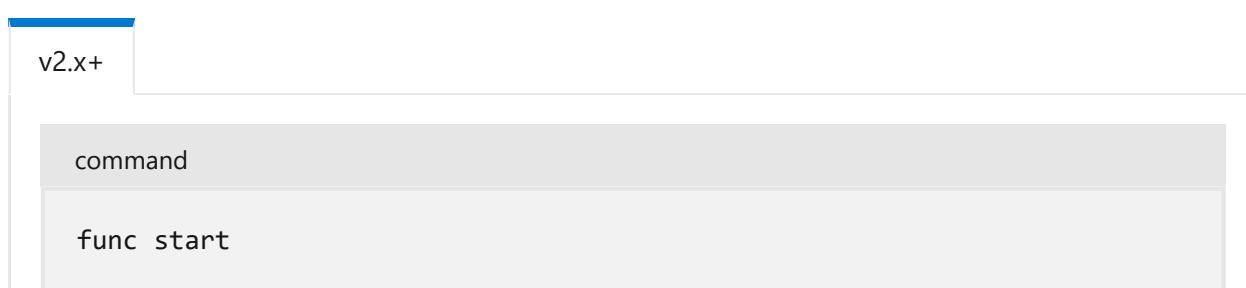
For example, to call an HTTP-triggered function and pass content body, run the following command:

```
func run MyHttpTrigger --content '{\"name\": \"Azure\"}'
```

func start

Starts the local runtime host and loads the function project in the current folder.

The specific command depends on the [runtime version](#).



`func start` supports the following options:

[Expand table](#)

Option	Description
--cert	The path to a .pfx file that contains a private key. Only supported with <code>--useHttps</code> .
--cors	A comma-separated list of CORS origins, with no spaces.
--cors-credentials	Allow cross-origin authenticated requests using cookies and the Authentication header.
--dotnet-isolated-debug	When set to <code>true</code> , pauses the .NET worker process until a debugger is attached from the .NET isolated project being debugged.
--enable-json-output	Emits console logs as JSON, when possible.
--enableAuth	Enable full authentication handling pipeline, with authorization requirements.

Option	Description
--functions	A space-separated list of functions to load.
--language-worker	Arguments to configure the language worker. For example, you can enable debugging for language worker by providing debug port and other required arguments .
--no-build	Don't build the current project before running. For .NET class projects only. The default is <code>false</code> .
--password	Either the password or a file that contains the password for a .pfx file. Only used with <code>--cert</code> .
--port	The local port to listen on. Default value: 7071.
--timeout	The timeout for the Functions host to start, in seconds. Default: 20 seconds.
--useHttps	Bind to <code>https://localhost:{port}</code> rather than to <code>http://localhost:{port}</code> . By default, this option creates a trusted certificate on your computer.

With the project running, you can [verify individual function endpoints](#).

func azure functionapp fetch-app-settings

Gets settings from a specific function app.

command

```
func azure functionapp fetch-app-settings <APP_NAME>
```

`func azure functionapp fetch-app-settings` supports these optional arguments:

[+] [Expand table](#)

Option	Description
--access-token	Lets you use a specific access token when performing authenticated <code>azure</code> actions.
--access-token-stdin	Reads a specific access token from a standard input. Use this when reading the token directly from a previous command such as az account get-access-token .

Option	Description
--management-url	Sets the management URL for your cloud. Use this when running in a sovereign cloud.
--slot	Optional name of a specific slot to which to publish.
--subscription	Sets the default subscription to use.

For more information, see [Download application settings](#).

Settings are downloaded into the local.settings.json file for the project. On-screen values are masked for security. You can protect settings in the local.settings.json file by [enabling local encryption](#).

func azure functionapp list-functions

Returns a list of the functions in the specified function app.

command

```
func azure functionapp list-functions <APP_NAME>
```

`func azure functionapp list-functions` supports these optional arguments:

[\[+\] Expand table](#)

Option	Description
--access-token	Lets you use a specific access token when performing authenticated <code>azure</code> actions.
--access-token-stdin	Reads a specific access token from a standard input. Use this when reading the token directly from a previous command such as az account get-access-token .
--management-url	Sets the management URL for your cloud. Use this when running in a sovereign cloud.
--show-keys	Shows HTTP function endpoint URLs that include their default access keys. These URLs can be used to access function endpoints with <code>function</code> level HTTP authentication .
--slot	Optional name of a specific slot to which to publish.
--subscription	Sets the default subscription to use.

func azure functionapp logstream

Connects the local command prompt to streaming logs for the function app in Azure.

command

```
func azure functionapp logstream <APP_NAME>
```

The default timeout for the connection is 2 hours. You can change the timeout by adding an app setting named [SCM_LOGSTREAM_TIMEOUT](#), with a timeout value in seconds. Not yet supported for Linux apps in the Consumption plan. For these apps, use the `--browser` option to view logs in the portal.

The `func azure functionapp logstream` command supports these optional arguments:

[\[+\] Expand table](#)

Option	Description
<code>--access-token</code>	Lets you use a specific access token when performing authenticated <code>azure</code> actions.
<code>--access-token-stdin</code>	Reads a specific access token from a standard input. Use this when reading the token directly from a previous command such as az account get-access-token .
<code>--browser</code>	Open Azure Application Insights Live Stream for the function app in the default browser.
<code>--management-url</code>	Sets the management URL for your cloud. Use this when running in a sovereign cloud.
<code>--slot</code>	Optional name of a specific slot to which to publish.
<code>--subscription</code>	Sets the default subscription to use.

For more information, see [Enable streaming execution logs in Azure Functions](#).

func azure functionapp publish

Deploys a Functions project to an existing function app resource in Azure.

command

```
func azure functionapp publish <APP_NAME>
```

For more information, see [Deploy project files](#).

The following publish options apply, based on version:

v2.x+	 Expand table
Option	Description
<code>--access-token</code>	Lets you use a specific access token when performing authenticated <code>azure</code> actions.
<code>--access-token-stdin</code>	Reads a specific access token from a standard input. Use this when reading the token directly from a previous command such as az account get-access-token .
<code>--additional-packages</code>	List of packages to install when building native dependencies. For example: <code>python3-dev libevent-dev</code> .
<code>--build, -b</code>	Performs build action when deploying to a Linux function app. Accepts: <code>remote</code> and <code>local</code> .
<code>--build-native-deps</code>	Skips generating the <code>.wheels</code> folder when publishing Python function apps.
<code>--csx</code>	Publish a C# script (.csx) project.
<code>--dotnet-cli-params</code>	When publishing compiled C# (.csproj) functions, the core tools calls <code>dotnet build --output bin/publish</code> . Any parameters passed to this are appended to the command line.
<code>--force</code>	Ignore prepublishing verification in certain scenarios.
<code>--list-ignored-files</code>	Displays a list of files that are ignored during publishing, which is based on the <code>.funcignore</code> file.
<code>--list-included-files</code>	Displays a list of files that are published, which is based on the <code>.funcignore</code> file.
<code>--management-url</code>	Sets the management URL for your cloud. Use this when running in a sovereign cloud.
<code>--no-build</code>	Project isn't built during publishing. For Python, <code>pip install</code> isn't performed.
<code>--nozip</code>	Turns the default <code>Run-From-Package</code> mode off.

Option	Description
<code>--overwrite-settings -y</code>	Suppress the prompt to overwrite app settings when <code>--publish-local-settings -i</code> is used.
<code>--publish-local-settings -i</code>	Publish settings in local.settings.json to Azure, prompting to overwrite if the setting already exists. If you're using a local storage emulator , first change the app setting to an actual storage connection .
<code>--publish-settings-only, -o</code>	Only publish settings and skip the content. Default is prompt.
<code>--show-keys</code>	Shows HTTP function endpoint URLs that include their default access keys. These URLs can be used to access function endpoints with <code>function</code> level HTTP authentication .
<code>--slot</code>	Optional name of a specific slot to which to publish.
<code>--subscription</code>	Sets the default subscription to use.

func azure storage fetch-connection-string

Gets the connection string for the specified Azure Storage account.

command

```
func azure storage fetch-connection-string <STORAGE_ACCOUNT_NAME>
```

For more information, see [Download a storage connection string](#).

func azurecontainerapps deploy

Deploys a containerized function app to an Azure Container Apps environment. Both the storage account used by the function app and the environment must already exist. For more information, see [Azure Container Apps hosting of Azure Functions](#).

command

```
func azurecontainerapps deploy --name <APP_NAME> --environment <ENVIRONMENT_NAME> --storage-account <STORAGE_CONNECTION> --resource-group <RESOURCE_GROUP> --image-name <IMAGE_NAME> --registry-server <REGISTRY_SERVER> --registry-username <USERNAME> --registry-password
```

<PASSWORD>

The following deployment options apply:

[+] Expand table

Option	Description
--access-token	Lets you use a specific access token when performing authenticated <code>azure</code> actions.
--access-token-stdin	Reads a specific access token from a standard input. Use this when reading the token directly from a previous command such as az account get-access-token .
--environment	The name of an existing Container Apps environment.
--image-build	When set to <code>true</code> , skips the local Docker build.
--image-name	The image name of an existing container in a container registry. The image name includes the tag name.
--location	Region for the deployment. Ideally, this is the same region as the environment and storage account resources.
--management-url	Sets the management URL for your cloud. Use this when running in sovereign cloud.
--name	The name used for the function app deployment in the Container Apps environment. This same name is also used when managing the function app in the portal. The name should be unique in the environment.
--registry	When set, a Docker build is run and the image is pushed to the registry set in <code>--registry</code> . You can't use <code>--registry</code> with <code>--image-name</code> . For Docker Hub, also use <code>--registry-username</code> .
--registry-password	The password or token used to retrieve the image from a private registry.
--registry-username	The username used to retrieve the image from a private registry.
--resource-group	The resource group in which to create the functions-related resources.
--storage-account	The connection string for the storage account to be used by the function app.
--subscription	Sets the default subscription to use.

Option	Description
--worker-runtime	Sets the runtime language of the function app. This parameter is only used with --image-name and --image-build, otherwise the language is determined during the local build. Supported values are: dotnet, dotnetIsolated, node, python, powershell, and custom (for customer handlers).

ⓘ Important

Storage connection strings and other service credentials are important secrets. Make sure to securely store any script files using `func azurecontainerapps deploy` and don't store them in any publicly accessible source control.

func deploy

The `func deploy` command is deprecated. Instead use [func kubernetes deploy](#).

func durable delete-task-hub

Deletes all storage artifacts in the Durable Functions task hub.

```
command  
func durable delete-task-hub
```

The `delete-task-hub` action supports the following options:

[\[+\] Expand table](#)

Option	Description
--connection-string-setting	Optional name of the setting containing the storage connection string to use.
--task-hub-name	Optional name of the Durable Task Hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable get-history

Returns the history of the specified orchestration instance.

command

```
func durable get-history --id <INSTANCE_ID>
```

The `get-history` action supports the following options:

[+] [Expand table](#)

Option	Description
<code>--id</code>	Specifies the ID of an orchestration instance (required).
<code>--connection-string-setting</code>	Optional name of the setting containing the storage connection string to use.
<code>--task-hub-name</code>	Optional name of the Durable Task Hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable get-instances

Returns the status of all orchestration instances. Supports paging using the `top` parameter.

command

```
func durable get-instances
```

The `get-instances` action supports the following options:

[+] [Expand table](#)

Option	Description
<code>--continuation-token</code>	Optional token that indicates a specific page/section of the requests to return.
<code>--connection-string-setting</code>	Optional name of the app setting that contains the storage connection string to use.
<code>--created-after</code>	Optionally, get the instances created after this date/time (UTC). All ISO 8601 formatted datetimes are accepted.
<code>--created-before</code>	Optionally, get the instances created before a specific date/time (UTC). All ISO 8601 formatted datetimes are accepted.

Option	Description
--runtime-status	Optionally, get the instances whose status match a specific status, including <code>running</code> , <code>completed</code> , and <code>failed</code> . You can provide one or more space-separated statuses.
--top	Optionally limit the number of records returned in a given request.
--task-hub-name	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable get-runtime-status

Returns the status of the specified orchestration instance.

```
command
func durable get-runtime-status --id <INSTANCE_ID>
```

The `get-runtime-status` action supports the following options:

[\[+\] Expand table](#)

Option	Description
--connection-string-setting	Optional name of the setting containing the storage connection string to use.
--id	Specifies the ID of an orchestration instance (required).
--show-input	When set, the response contains the input of the function.
--show-output	When set, the response contains the execution history.
--task-hub-name	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable purge-history

Purge orchestration instance state, history, and blob storage for orchestrations older than the specified threshold.

```
command
```

```
func durable purge-history
```

The `purge-history` action supports the following options:

[+] [Expand table](#)

Option	Description
<code>--connection-string-setting</code>	Optional name of the setting containing the storage connection string to use.
<code>--created-after</code>	Optionally delete the history of instances created after this date/time (UTC). All ISO 8601 formatted datetime values are accepted.
<code>--created-before</code>	Optionally delete the history of instances created before this date/time (UTC). All ISO 8601 formatted datetime values are accepted.
<code>--runtime-status</code>	Optionally delete the history of instances whose status match a specific status, including <code>completed</code> , <code>terminated</code> , <code>canceled</code> , and <code>failed</code> . You can provide one or more space-separated statuses. If you don't include <code>--runtime-status</code> , instance history is deleted regardless of status.
<code>--task-hub-name</code>	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable raise-event

Raises an event to the specified orchestration instance.

```
command
```

```
func durable raise-event --event-name <EVENT_NAME> --event-data <DATA>
```

The `raise-event` action supports the following options:

[+] [Expand table](#)

Option	Description
<code>--connection-string-setting</code>	Optional name of the setting containing the storage connection string to use.

Option	Description
--event-data	Data to pass to the event, either inline or from a JSON file (required). For files, prefix the path to the file with an ampersand (@), such as <code>@path/to/file.json</code> .
--event-name	Name of the event to raise (required).
--id	Specifies the ID of an orchestration instance (required).
--task-hub-name	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable rewind

Rewinds the specified orchestration instance.

```
command
func durable rewind --id <INSTANCE_ID> --reason <REASON>
```

The `rewind` action supports the following options:

[\[+\] Expand table](#)

Option	Description
--connection-string-setting	Optional name of the setting containing the storage connection string to use.
--id	Specifies the ID of an orchestration instance (required).
--reason	Reason for rewinding the orchestration (required).
--task-hub-name	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable start-new

Starts a new instance of the specified orchestrator function.

```
command
```

```
func durable start-new --id <INSTANCE_ID> --function-name <FUNCTION_NAME> --  
input <INPUT>
```

The `start-new` action supports the following options:

[+] Expand table

Option	Description
<code>--connection-string-setting</code>	Optional name of the setting containing the storage connection string to use.
<code>--function-name</code>	Name of the orchestrator function to start (required).
<code>--id</code>	Specifies the ID of an orchestration instance (required).
<code>--input</code>	Input to the orchestrator function, either inline or from a JSON file (required). For files, prefix the path to the file with an ampersand (@), such as <code>@path/to/file.json</code> .
<code>--task-hub-name</code>	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func durable terminate

Stops the specified orchestration instance.

command

```
func durable terminate --id <INSTANCE_ID> --reason <REASON>
```

The `terminate` action supports the following options:

[+] Expand table

Option	Description
<code>--connection-string-setting</code>	Optional name of the setting containing the storage connection string to use.
<code>--id</code>	Specifies the ID of an orchestration instance (required).
<code>--reason</code>	Reason for stopping the orchestration (required).

Option	Description
--task-hub-name	Optional name of the Durable Functions task hub to use.

To learn more, see the [Durable Functions documentation](#).

func extensions install

Manually installs Functions extensions in a non-.NET project or in a C# script project.

command

```
func extensions install --package Microsoft.Azure.WebJobs.Extensions.  
<EXTENSION> --version <VERSION>
```

The `install` action supports the following options:

[] [Expand table](#)

Option	Description
-- <code>configPath</code>	Path of the directory containing extensions.csproj file.
--csx	Supports C# scripting (.csx) projects.
--force	Update the versions of existing extensions.
--output	Output path for the extensions.
--package	Identifier for a specific extension package. When not specified, all referenced extensions are installed, as with <code>func extensions sync</code> .
--source	NuGet feed source when not using NuGet.org.
--version	Extension package version.

The following example installs version 5.0.1 of the Event Hubs extension in the local project:

command

```
func extensions install --package  
Microsoft.Azure.WebJobs.Extensions.EventHubs --version 5.0.1
```

The following considerations apply when using `func extensions install`:

- For compiled C# projects (both in-process and isolated worker process), instead use standard NuGet package installation methods, such as `dotnet add package`.
- To manually install extensions using Core Tools, you must have the [.NET 6.0 SDK](#) installed.
- When possible, you should instead use [extension bundles](#). The following are some reasons why you might need to install extensions manually:
 - You need to access a specific version of an extension not available in a bundle.
 - You need to access a custom extension not available in a bundle.
 - You need to access a specific combination of extensions not available in a single bundle.
- Before you can manually install extensions, you must first remove the `extensionBundle` object from the `host.json` file that defines the bundle. No action is taken when an extension bundle is already set in your [host.json file](#).
- The first time you explicitly install an extension, a .NET project file named `extensions.csproj` is added to the root of your app project. This file defines the set of NuGet packages required by your functions. While you can work with the [NuGet package references](#) in this file, Core Tools lets you install extensions without having to manually edit this C# project file.

func extensions sync

Installs all extensions added to the function app.

The `sync` action supports the following options:

[\[+\] Expand table](#)

Option	Description
<code>--configPath</code>	Path of the directory containing <code>extensions.csproj</code> file.
<code>--csx</code>	Supports C# scripting (<code>.csx</code>) projects.
<code>--output</code>	Output path for the extensions.

Regenerates a missing `extensions.csproj` file. No action is taken when an extension bundle is defined in your `host.json` file.

func kubernetes deploy

Deploys a Functions project as a custom docker container to a Kubernetes cluster.

```
command
```

```
func kubernetes deploy
```

This command builds your project as a custom container and publishes it to a Kubernetes cluster. Custom containers must have a Dockerfile. To create an app with a Dockerfile, use the `--dockerfile` option with the [func init](#) command.

The following Kubernetes deployment options are available:

[+] [Expand table](#)

Option	Description
<code>--dry-run</code>	Optionally displays the deployment template, without execution.
<code>--config-map-name</code>	Optional name of an existing config map with function app settings to use in the deployment. Requires <code>--use-config-map</code> . The default behavior is to create settings based on the <code>Values</code> object in the local.settings.json file .
<code>--cooldown-period</code>	The cool-down period (in seconds) after all triggers are no longer active before the deployment scales back down to zero, with a default of 300 s.
<code>--ignore-errors</code>	Continues the deployment after a resource returns an error. The default behavior is to stop on error.
<code>--image-name</code>	The name of the image to use for the pod deployment and from which to read functions.
<code>--keda-version</code>	Sets the version of KEDA to install. Valid options are: <code>v1</code> and <code>v2</code> (default).
<code>--keys-secret-name</code>	The name of a Kubernetes Secrets collection to use for storing access keys .
<code>--max-replicas</code>	Sets the maximum replica count for to which the Horizontal Pod Autoscaler (HPA) scales.
<code>--min-replicas</code>	Sets the minimum replica count below which HPA won't scale.
<code>--mount-funckeys-as-containervolume</code>	Mounts the access keys as a container volume.
<code>--name</code>	The name used for the deployment and other artifacts in Kubernetes.
<code>--namespace</code>	Sets the Kubernetes namespace to which to deploy, which defaults to the default namespace.

Option	Description
--no-docker	Functions are read from the current directory instead of from an image. Requires mounting the image filesystem.
--registry	When set, a Docker build is run and the image is pushed to a registry of that name. You can't use --registry with --image-name. For Docker, use your username.
--polling-interval	The polling interval (in seconds) for checking non-HTTP triggers, with a default of 30s.
--pull-secret	The secret used to access private registry credentials.
--secret-name	The name of an existing Kubernetes Secrets collection that contains function app settings to use in the deployment. The default behavior is to create settings based on the <code>Values</code> object in the local.settings.json file .
--show-service-fqdn	Displays the URLs of HTTP triggers with the Kubernetes FQDN instead of the default behavior of using an IP address.
--service-type	Sets the type of Kubernetes Service. Supported values are: <code>ClusterIP</code> , <code>NodePort</code> , and <code>LoadBalancer</code> (default).
--use-config-map	Use a <code>ConfigMap</code> object (v1) instead of a <code>Secret</code> object (v1) to configure function app settings . The map name is set using <code>--config-map-name</code> .

Core Tools uses the local Docker CLI to build and publish the image. Make sure your Docker is already installed locally. Run the `docker login` command to connect to your account.

To learn more, see [Deploying a function app to Kubernetes](#).

func kubernetes install

Installs KEDA in a Kubernetes cluster.

```
command
func kubernetes install
```

Installs KEDA to the cluster defined in the `kubectl config` file.

The `install` action supports the following options:

[\[\] Expand table](#)

Option	Description
--dry-run	Displays the deployment template, without execution.
--keda-version	Sets the version of KEDA to install. Valid options are: v1 and v2 (default).
--namespace	Supports installation to a specific Kubernetes namespace. When not set, the default namespace is used.

To learn more, see [Managing KEDA and functions in Kubernetes](#).

func kubernetes remove

Removes KEDA from the Kubernetes cluster defined in the kubectl config file.

command
func kubernetes remove

Removes KEDA from the cluster defined in the kubectl config file.

The `remove` action supports the following options:

[\[\] Expand table](#)

Option	Description
--namespace	Supports uninstall from a specific Kubernetes namespace. When not set, the default namespace is used.

To learn more, see [Uninstalling KEDA from Kubernetes](#).

func settings add

Adds a new setting to the `values` collection in the [local.settings.json](#) file.

command
func settings add <SETTING_NAME> <VALUE>

Replace `<SETTING_NAME>` with the name of the app setting and `<VALUE>` with the value of the setting.

The `add` action supports the following option:

[+] Expand table

Option	Description
<code>--connectionString</code>	Adds the name-value pair to the <code>ConnectionStrings</code> collection instead of the <code>Values</code> collection. Only use the <code>ConnectionStrings</code> collection when required by certain frameworks. To learn more, see local.settings.json file .

func settings decrypt

Decrypts previously encrypted values in the `Values` collection in the [local.settings.json file](#).

```
command
func settings decrypt
```

Connection string values in the `ConnectionStrings` collection are also decrypted. In `local.settings.json`, `IsEncrypted` is also set to `false`. Encrypt local settings to reduce the risk of leaking valuable information from `local.settings.json`. In Azure, application settings are always stored encrypted.

func settings delete

Removes an existing setting from the `Values` collection in the [local.settings.json file](#).

```
command
func settings delete <SETTING_NAME>
```

Replace `<SETTING_NAME>` with the name of the app setting and `<VALUE>` with the value of the setting.

The `delete` action supports the following option:

[+] Expand table

Option	Description
-- <code>connectionString</code>	Removes the name-value pair from the <code>ConnectionStrings</code> collection instead of from the <code>Values</code> collection.

func settings encrypt

Encrypts the values of individual items in the `Values` collection in the [local.settings.json file](#).

command

```
func settings encrypt
```

Connection string values in the `ConnectionStrings` collection are also encrypted. In `local.settings.json`, `IsEncrypted` is also set to `true`, which specifies that the local runtime decrypts settings before using them. Encrypt local settings to reduce the risk of leaking valuable information from `local.settings.json`. In Azure, application settings are always stored encrypted.

func settings list

Outputs a list of settings in the `Values` collection in the [local.settings.json file](#).

command

```
func settings list
```

Connection strings from the `ConnectionStrings` collection are also output. By default, values are masked for security. You can use the `--showValue` option to display the actual value.

The `list` action supports the following option:

[\[+\] Expand table](#)

Option	Description
<code>--showValue</code>	Shows the actual unmasked values in the output.

func templates list

Lists the available function (trigger) templates.

The `list` action supports the following option:

[Expand table

Option	Description
<code>--language</code>	Language for which to filter returned templates. Default is to return all languages.

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Az.Functions

Reference

Microsoft Azure PowerShell - Azure Functions service cmdlets for Azure Resource Manager in Windows PowerShell and PowerShell Core.\n\nFor information on Azure Functions, please visit the following: <https://learn.microsoft.com/azure/azure-functions/>

Functions

[] [Expand table](#)

Get-AzFunctionApp	Gets function apps in a subscription.
Get-AzFunctionAppAvailableLocation	Gets the location where a function app for the given os and plan type is available.
Get-AzFunctionAppPlan	Get function apps plans in a subscription.
Get-AzFunctionAppSetting	Gets app settings for a function app.
New-AzFunctionApp	Creates a function app.
New-AzFunctionAppPlan	Creates a function app service plan.
Remove-AzFunctionApp	Deletes a function app.
Remove-AzFunctionAppPlan	Deletes a function app plan.
Remove-AzFunctionAppSetting	Removes app settings from a function app.
Restart-AzFunctionApp	Restarts a function app.
Start-AzFunctionApp	Starts a function app.
Stop-AzFunctionApp	Stops a function app.
Update-AzFunctionApp	Updates a function app.
Update-AzFunctionAppPlan	Updates a function app service plan.
Update-AzFunctionAppSetting	Adds or updates app settings in a function app.

Runtime

Reference

Packages

[] [Expand table](#)

com.microsoft.azure.functions
com.microsoft.azure.functions.annotation

functions Package

Reference

Modules

[] Expand table

blob
cosmosdb
durable_functions
eventgrid
eventhub
http
kafka
meta
queue
servicebus
timer

Classes

[] Expand table

Context	Function invocation context.
Document	An Azure Document. Document objects are <code>UserDict</code> subclasses and behave like dicts.
DocumentList	A <code>UserList</code> subclass containing a list of Document objects
EntityContext	A durable function entity context.
EventGridEvent	An EventGrid event message.
EventGridOutputEvent	An EventGrid event message.

EventHubEvent	A concrete implementation of Event Hub message type.
HttpRequest	An HTTP request object.
HttpResponse	An HTTP response object.
InputStream	File-like object representing an input blob.
KafkaConverter	
KafkaEvent	A concrete implementation of Kafka event message type.
KafkaTriggerConverter	
OrchestrationContext	A durable function orchestration context.
Out	An interface to set function output parameters.
QueueMessage	A Queue message object.
ServiceBusMessage	
TimerRequest	Timer request object.
WsgiMiddleware	

Functions

get_binding_registry

```
get_binding_registry()
```

App settings reference for Azure Functions

Article • 07/11/2024

Application settings in a function app contain configuration options that affect all functions for that function app. These settings are accessed as environment variables. This article lists the app settings that are available in function apps.

There are several ways that you can add, update, and delete function app settings:

- [In the Azure portal](#).
- [By using the Azure CLI](#).
- [By using Azure PowerShell](#).

Changes to function app settings require your function app to be restarted.

In this article, example connection string values are truncated for readability.

App setting considerations

When using app settings, you should be aware of the following considerations:

- Changes to function app settings require your function app to be restarted.
- In setting names, double-underscore (`_`) and semicolon (`:`) are considered reserved values. Double-underscores are interpreted as hierarchical delimiters on both Windows and Linux, and colons are interpreted in the same way only on Linux. For example, the setting `AzureFunctionsWebHost__hostid=somehost_123456` would be interpreted as the following JSON object:

JSON

```
"AzureFunctionsWebHost": {  
    "hostid": "somehost_123456"  
}
```

In this article, only double-underscores are used, since they're supported on both operating systems. Most of the settings that support managed identity connections use double-underscores.

- When Functions runs locally, app settings are specified in the `values` collection in the `local.settings.json`.
- There are other function app configuration options in the `host.json` file and in the `local.settings.json` file.
- You can use application settings to override host.json setting values without having to change the host.json file itself. This is helpful for scenarios where you need to configure or modify specific host.json settings for a specific environment. This also lets you change host.json settings without having to republish your project. To learn more, see the [host.json reference article](#).
- This article documents the settings that are most relevant to your function apps. Because Azure Functions runs on App Service, other application settings are also supported. For more information, see [Environment variables and app settings in Azure App Service](#).
- Some scenarios also require you to work with settings documented in [App Service site settings](#).
- Changing any *read-only* [App Service application settings](#) can put your function app into an unresponsive state.
- Take care when updating application settings by using REST APIs, including ARM templates. Because these APIs replace the existing application settings, you must include all existing settings when adding or modifying settings using REST APIs or ARM templates. When possible use Azure CLI or Azure PowerShell to programmatically work with application settings. For more information, see [Work with application settings](#).

APPINSIGHTS_INSTRUMENTATIONKEY

The instrumentation key for Application Insights. Don't use both `APPINSIGHTS_INSTRUMENTATIONKEY` and `APPLICATIONINSIGHTS_CONNECTION_STRING`. When possible, use `APPLICATIONINSIGHTS_CONNECTION_STRING`. When Application Insights runs in a sovereign cloud, you must use `APPLICATIONINSIGHTS_CONNECTION_STRING`. For more information, see [How to configure monitoring for Azure Functions](#).

[+] [Expand table](#)

Key	Sample value
APPINSIGHTS_INSTRUMENTATIONKEY	55555555-af77-484b-9032-64f83bb83bb

Don't use both `APPINSIGHTS_INSTRUMENTATIONKEY` and `APPLICATIONINSIGHTS_CONNECTION_STRING`. Use of `APPLICATIONINSIGHTS_CONNECTION_STRING` is recommended.

ⓘ Note

On March 31, 2025, support for instrumentation key ingestion will end. Instrumentation key ingestion will continue to work, but we'll no longer provide updates or support for the feature. [Transition to connection strings](#) to take advantage of [new capabilities](#).

APPLICATIONINSIGHTS_AUTHENTICATION_STRING

The connection string for Application Insights by using Microsoft Entra authentication. Use this setting when you must connect to your Application Insights workspace by using Microsoft Entra authentication. The string contains the client ID of either a system-assigned or a user-assigned managed identity that is authorized to publish telemetry to your Application Insights workspace. For more information, see [Microsoft Entra authentication for Application Insights](#).

[+] [Expand table](#)

Key	Sample value
APPLICATIONINSIGHTS_AUTHENTICATION_STRING	ClientId=<YOUR_CLIENT_ID>;Authorization=AAD

ⓘ Note

When using `APPLICATIONINSIGHTS_AUTHENTICATION_STRING` to connect to Application Insights using Microsoft Entra authentication, you should also [Disable local authentication for Application Insights](#). This configuration requires Microsoft Entra authentication in order for telemetry to be ingested into your workspace.

APPLICATIONINSIGHTS_CONNECTION_STRING

The connection string for Application Insights. Don't use both `APPINSIGHTS_INSTRUMENTATIONKEY` and `APPLICATIONINSIGHTS_CONNECTION_STRING`. While the use of `APPLICATIONINSIGHTS_CONNECTION_STRING` is recommended in all cases, it's required in the following cases:

- When your function app requires the added customizations supported by using the connection string.
- When your Application Insights instance runs in a sovereign cloud, which requires a custom endpoint.

For more information, see [Connection strings](#).

[+] [Expand table](#)

Key	Sample value
APPLICATIONINSIGHTS_CONNECTION_STRING	InstrumentationKey=...

To connect to Application Insights with Microsoft Entra authentication, you should instead use [APPLICATIONINSIGHTS_AUTHENTICATION_STRING](#).

AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL

ⓘ Important

Azure Functions proxies is a legacy feature for [versions 1.x through 3.x](#) of the Azure Functions runtime. For more information about legacy support in version 4.x, see [Functions proxies](#).

By default, Functions proxies use a shortcut to send API calls from proxies directly to functions in the same function app. This shortcut is used instead of creating a new HTTP request. This setting allows you to disable that shortcut behavior.

[Expand table](#)

Key	Value	Description
AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL	true	Calls with a backend URL pointing to a function in the local function app won't be sent directly to the function. Instead, the requests are directed back to the HTTP frontend for the function app.
AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL	false	Calls with a backend URL pointing to a function in the local function app are forwarded directly to the function. <code>false</code> is the default value.

AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES

ⓘ Important

Azure Functions proxies is a legacy feature for [versions 1.x through 3.x](#) of the Azure Functions runtime. For more information about legacy support in version 4.x, see [Functions proxies](#).

This setting controls whether the characters `%2F` are decoded as slashes in route parameters when they're inserted into the backend URL.

[Expand table](#)

Key	Value	Description
AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES	true	Route parameters with encoded slashes are decoded.
AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES	false	All route parameters are passed along unchanged, which is the default behavior.

For example, consider the `proxies.json` file for a function app at the `myfunction.com` domain.

JSON

```
{
  "$schema": "http://json.schemastore.org/proxies",
  "proxies": {
    "root": {
      "matchCondition": {
        "route": "/{*all}"
      }
    },
  }
}
```

```

        "backendUri": "example.com/{all}"
    }
}

```

When `AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES` is set to `true`, the URL `example.com/api%2ftest` resolves to `example.com/api/test`. By default, the URL remains unchanged as `example.com/test%2fapi`. For more information, see [Functions proxies](#).

AZURE_FUNCTIONS_ENVIRONMENT

Configures the runtime [hosting environment](#) of the function app when running in Azure. This value is read during initialization, and only these values are honored by the runtime:

[Expand table](#)

Value	Description
Production	Represents a production environment, with reduced logging and full performance optimizations. This is the default when <code>AZURE_FUNCTIONS_ENVIRONMENT</code> either isn't set or is set to an unsupported value.
Staging	Represents a staging environment, such as when running in a staging slot .
Development	A development environment supports more verbose logging and other reduced performance optimizations. The Azure Functions Core Tools sets <code>AZURE_FUNCTIONS_ENVIRONMENT</code> to <code>Development</code> when running on your local computer. This setting can't be overridden in the local.settings.json file.

Use this setting instead of `ASPNETCORE_ENVIRONMENT` when you need to change the runtime environment in Azure to something other than `Production`. For more information, see [Environment-based Startup class and methods](#).

This setting isn't available in version 1.x of the Functions runtime.

AzureFunctionsJobHost_*

In version 2.x and later versions of the Functions runtime, application settings can override `host.json` settings in the current environment. These overrides are expressed as application settings named `AzureFunctionsJobHost_path_to_setting`. For more information, see [Override host.json values](#).

AzureFunctionsWebHost_hostid

Sets the host ID for a given function app, which should be a unique ID. This setting overrides the automatically generated host ID value for your app. Use this setting only when you need to prevent host ID collisions between function apps that share the same storage account.

A host ID must meet the following requirements:

- Be between 1 and 32 characters
- contain only lowercase letters, numbers, and dashes
- Not start or end with a dash
- Not contain consecutive dashes

An easy way to generate an ID is to take a GUID, remove the dashes, and make it lower case, such as by converting the GUID `1835D7B5-5C98-4790-815D-072CC94C6F71` to the value `1835d7b55c984790815d072cc94c6f71`.

[Expand table](#)

Key	Sample value
AzureFunctionsWebHost__hostid	myuniquefunctionappname123456789

For more information, see [Host ID considerations](#).

AzureWebJobsDashboard

This setting is deprecated and is only supported when running on version 1.x of the Azure Functions runtime.

Optional storage account connection string for storing logs and displaying them in the **Monitor** tab in the portal. The storage account must be a general-purpose one that supports blobs, queues, and tables. To learn more, see [Storage account requirements](#).

[Expand table](#)

Key	Sample value
AzureWebJobsDashboard	DefaultEndpointsProtocol=https;AccountName=...

AzureWebJobsDisableHomepage

A value of `true` disables the default landing page that is shown for the root URL of a function app. The default value is `false`.

[Expand table](#)

Key	Sample value
AzureWebJobsDisableHomepage	true

When this app setting is omitted or set to `false`, a page similar to the following example is displayed in response to the URL `<functionappname>.azurewebsites.net`.



AzureWebJobsDotNetReleaseCompilation

`true` means use Release mode when compiling .NET code; `false` means use Debug mode. Default is `true`.

[Expand table](#)

Key	Sample value
AzureWebJobsDotNetReleaseCompilation	<code>true</code>

AzureWebJobsFeatureFlags

A comma-delimited list of beta features to enable. Beta features enabled by these flags aren't production ready, but can be enabled for experimental use before they go live.

[Expand table](#)

Key	Sample value
AzureWebJobsFeatureFlags	<code>feature1,feature2,EnableProxies</code>

Add `EnableProxies` to this list to re-enable proxies on version 4.x of the Functions runtime while you plan your migration to Azure API Management. For more information, see [Re-enable proxies in Functions v4.x](#).

AzureWebJobsKubernetesSecretName

Indicates the Kubernetes Secrets resource used for storing keys. Supported only when running in Kubernetes. This setting requires you to set `AzureWebJobsSecretStorageType` to `kubernetes`. When `AzureWebJobsKubernetesSecretName` isn't set, the repository is considered read only. In this case, the values must be generated before deployment. The [Azure Functions Core Tools](#) generates the values automatically when deploying to Kubernetes.

[Expand table](#)

Key	Sample value
AzureWebJobsKubernetesSecretName	<code><SECRETS_RESOURCE></code>

To learn more, see [Secret repositories](#).

AzureWebJobsSecretStorageKeyVaultClientId

The client ID of the user-assigned managed identity or the app registration used to access the vault where keys are stored. This setting requires you to set `AzureWebJobsSecretStorageType` to `keyvault`. Supported in version 4.x and later versions of the Functions runtime.

[Expand table](#)

Key	Sample value
AzureWebJobsSecretStorageKeyVaultClientId	<code><CLIENT_ID></code>

To learn more, see [Secret repositories](#).

AzureWebJobsSecretStorageKeyVaultClientSecret

The secret for client ID of the user-assigned managed identity or the app registration used to access the vault where keys are stored. This setting requires you to set `AzureWebJobsSecretStorageType` to `keyvault`. Supported in version 4.x and later versions of the Functions runtime.

[+] Expand table

Key	Sample value
AzureWebJobsSecretStorageKeyVaultClientSecret	<CLIENT_SECRET>

To learn more, see [Secret repositories](#).

AzureWebJobsSecretStorageKeyVaultName

The name of a key vault instance used to store keys. This setting is only supported for version 3.x of the Functions runtime. For version 4.x, instead use `AzureWebJobsSecretStorageKeyVaultUri`. This setting requires you to set

`AzureWebJobsSecretStorageType` to `keyvault`.

The vault must have an access policy corresponding to the system-assigned managed identity of the hosting resource. The access policy should grant the identity the following secret permissions: `Get`, `Set`, `List`, and `Delete`.

When your functions run locally, the developer identity is used, and settings must be in the `local.settings.json` file.

[+] Expand table

Key	Sample value
AzureWebJobsSecretStorageKeyVaultName	<VAULT_NAME>

To learn more, see [Secret repositories](#).

AzureWebJobsSecretStorageKeyVaultTenantId

The tenant ID of the app registration used to access the vault where keys are stored. This setting requires you to set `AzureWebJobsSecretStorageType` to `keyvault`. Supported in version 4.x and later versions of the Functions runtime. To learn more, see [Secret repositories](#).

[+] Expand table

Key	Sample value
AzureWebJobsSecretStorageKeyVaultTenantId	<TENANT_ID>

AzureWebJobsSecretStorageKeyVaultUri

The URI of a key vault instance used to store keys. Supported in version 4.x and later versions of the Functions runtime.

This is the recommended setting for using a key vault instance for key storage. This setting requires you to set

`AzureWebJobsSecretStorageType` to `keyvault`.

The `AzureWebJobsSecretStorageKeyVaultUri` value should be the full value of **Vault URI** displayed in the **Key Vault** overview tab, including `https://`.

The vault must have an access policy corresponding to the system-assigned managed identity of the hosting resource. The access policy should grant the identity the following secret permissions: `Get`, `Set`, `List`, and `Delete`.

When your functions run locally, the developer identity is used, and settings must be in the `local.settings.json` file.

[Expand table](#)

Key	Sample value
AzureWebJobsSecretStorageKeyVaultUri	<a href="https://<VAULT_NAME>.vault.azure.net">https://<VAULT_NAME>.vault.azure.net

To learn more, see [Use Key Vault references for Azure Functions](#).

AzureWebJobsSecretStorageSas

A Blob Storage SAS URL for a second storage account used for key storage. By default, Functions uses the account set in `AzureWebJobsStorage`. When using this secret storage option, make sure that `AzureWebJobsSecretStorageType` isn't explicitly set or is set to `blob`. To learn more, see [Secret repositories](#).

[Expand table](#)

Key	Sample value
AzureWebJobsSecretStorageSas	<BLOB_SAS_URL>

AzureWebJobsSecretStorageType

Specifies the repository or provider to use for key storage. Keys are always encrypted before being stored using a secret unique to your function app.

[Expand table](#)

Key	Value	Description
AzureWebJobsSecretStorageType	<code>blob</code>	Keys are stored in a Blob storage container in the account provided by the <code>AzureWebJobsStorage</code> setting. Blob storage is the default behavior when <code>AzureWebJobsSecretStorageType</code> isn't set. To specify a different storage account, use the <code>AzureWebJobsSecretStorageSas</code> setting to indicate the SAS URL of a second storage account.
AzureWebJobsSecretStorageType	<code>files</code>	Keys are persisted on the file system. This is the default behavior for Functions v1.x.
AzureWebJobsSecretStorageType	<code>keyvault</code>	Keys are stored in a key vault instance set by <code>AzureWebJobsSecretStorageKeyVaultName</code> .
AzureWebJobsSecretStorageType	<code>kubernetes</code>	Supported only when running the Functions runtime in Kubernetes. When <code>AzureWebJobsKubernetesSecretName</code> isn't set, the repository is considered read only. In this case, the values must be generated before deployment. The Azure Functions Core Tools generates the values automatically when deploying to Kubernetes.

To learn more, see [Secret repositories](#).

AzureWebJobsStorage

Specifies the connection string for an Azure Storage account that the Functions runtime uses for normal operations. Some uses of this storage account by Functions include key management, timer trigger management, and Event Hubs checkpoints. The storage account must be a general-purpose one that supports blobs, queues, and tables. For more information, see [Storage account requirements](#).

[Expand table](#)

Key	Sample value
AzureWebJobsStorage	<code>DefaultEndpointsProtocol=https;AccountName=...</code>

Instead of a connection string, you can use an identity-based connection for this storage account. For more information, see [Connecting to host storage with an identity](#).

AzureWebJobsStorage__accountName

When using an identity-based storage connection, sets the account name of the storage account instead of using the connection string in `AzureWebJobsStorage`. This syntax is unique to `AzureWebJobsStorage` and can't be used for other identity-based connections.

[Expand table](#)

Key	Sample value
AzureWebJobsStorage__accountName	<STORAGE_ACCOUNT_NAME>

For sovereign clouds or when using a custom DNS, you must instead use the service-specific `AzureWebJobsStorage_*ServiceUri` settings.

AzureWebJobsStorage__blobServiceUri

When using an identity-based storage connection, sets the data plane URI of the blob service of the storage account.

[Expand table](#)

Key	Sample value
AzureWebJobsStorage__blobServiceUri	<code>https://<STORAGE_ACCOUNT_NAME>.blob.core.windows.net</code>

Use this setting instead of `AzureWebJobsStorage__accountName` in sovereign clouds or when using a custom DNS. For more information, see [Connecting to host storage with an identity](#).

AzureWebJobsStorage__queueServiceUri

When using an identity-based storage connection, sets the data plane URI of the queue service of the storage account.

[Expand table](#)

Key	Sample value
AzureWebJobsStorage__queueServiceUri	<code>https://<STORAGE_ACCOUNT_NAME>.queue.core.windows.net</code>

Use this setting instead of `AzureWebJobsStorage__accountName` in sovereign clouds or when using a custom DNS. For more information, see [Connecting to host storage with an identity](#).

AzureWebJobsStorage__tableServiceUri

When using an identity-based storage connection, sets data plane URI of a table service of the storage account.

[Expand table](#)

Key	Sample value
AzureWebJobsStorage__tableServiceUri	<code>https://<STORAGE_ACCOUNT_NAME>.table.core.windows.net</code>

Use this setting instead of `AzureWebJobsStorage__accountName` in sovereign clouds or when using a custom DNS. For more information, see [Connecting to host storage with an identity](#).

AzureWebJobs_TypeScriptPath

Path to the compiler used for TypeScript. Allows you to override the default if you need to.

[Expand table](#)

Key	Sample value
AzureWebJobs_TypeScriptPath	<code>%HOME%\typescript</code>

DOCKER_REGISTRY_SERVER_PASSWORD

Indicates the password used to access a private container registry. This setting is only required when deploying your containerized function app from a private container registry. For more information, see [Environment variables and app settings in Azure App Service](#).

DOCKER_REGISTRY_SERVER_URL

Indicates the URL of a private container registry. This setting is only required when deploying your containerized function app from a private container registry. For more information, see [Environment variables and app settings in Azure App Service](#).

DOCKER_REGISTRY_SERVER_USERNAME

Indicates the account used to access a private container registry. This setting is only required when deploying your containerized function app from a private container registry. For more information, see [Environment variables and app settings in Azure App Service](#).

DOCKER_SHM_SIZE

Sets the shared memory size (in bytes) when the Python worker is using shared memory. To learn more, see [Shared memory](#).

[Expand table](#)

Key	Sample value
DOCKER_SHM_SIZE	<code>268435456</code>

The value above sets a shared memory size of ~256 MB.

Requires that `FUNCTIONS_WORKER_SHARED_MEMORY_DATA_TRANSFER_ENABLED` be set to `1`.

ENABLE_ORYX_BUILD

Indicates whether the [Oryx build system](#) is used during deployment. `ENABLE_ORYX_BUILD` must be set to `true` when doing remote build deployments to Linux. For more information, see [Remote build](#).

[Expand table](#)

Key	Sample value
ENABLE_ORYX_BUILD	true

FUNCTION_APP_EDIT_MODE

Indicates whether you're able to edit your function app in the Azure portal. Valid values are `readwrite` and `readonly`.

[Expand table](#)

Key	Sample value
FUNCTION_APP_EDIT_MODE	readonly

The value is set by the runtime based on the language stack and deployment status of your function app. For more information, see [Development limitations in the Azure portal](#).

FUNCTIONS_EXTENSION_VERSION

The version of the Functions runtime that hosts your function app. A tilde (~) with major version means use the latest version of that major version (for example, `~4`). When new minor versions of the same major version are available, they're automatically installed in the function app.

[Expand table](#)

Key	Sample value
FUNCTIONS_EXTENSION_VERSION	<code>~4</code>

The following major runtime version values are supported:

[Expand table](#)

Value	Runtime target	Comment
<code>~4</code>	4.x	Recommended
<code>~1</code>	1.x	Support ends September 14, 2026

A value of `~4` means that your app runs on version 4.x of the runtime. A value of `~1` pins your app to version 1.x of the runtime. Runtime versions 2.x and 3.x are no longer supported. For more information, see [Azure Functions runtime versions overview](#). If requested by support to pin your app to a specific minor version, use the full version number (for example, `4.0.12345`). For more information, see [How to target Azure Functions runtime versions](#).

FUNCTIONS_INPROC_NET8_ENABLED

Indicates whether an app can use .NET 8 on the in-process model. To use .NET 8 on the in-process model, this value must be set to `1`. See [Updating to target .NET 8](#) for complete instructions, including other required configuration values.

[Expand table](#)

Key	Sample value
FUNCTIONS_INPROC_NET8_ENABLED	<code>1</code>

Set to `0` to disable support for .NET 8 on the in-process model.

FUNCTIONS_NODE_BLOCK_ON_ENTRY_POINT_ERROR

This app setting is a temporary way for Node.js apps to enable a breaking change that makes entry point errors easier to troubleshoot on Node.js v18 or lower. It's highly recommended to use `true`, especially for programming model v4 apps, which always use entry point files. The behavior without the breaking change (`false`) ignores entry point errors and doesn't log them in Application Insights.

Starting with Node.js v20, the app setting has no effect and the breaking change behavior is always enabled.

For Node.js v18 or lower, the app setting can be used and the default behavior depends on if the error happens before or after a model v4 function has been registered:

- If the error is thrown before (for example if you're using model v3 or your entry point file doesn't exist), the default behavior matches `false`.
- If the error is thrown after (for example if you try to register duplicate model v4 functions), the default behavior matches `true`.

[Expand table](#)

Key	Value	Description
FUNCTIONS_NODE_BLOCK_ON_ENTRY_POINT_ERROR	<code>true</code>	Block on entry point errors and log them in Application Insights.
FUNCTIONS_NODE_BLOCK_ON_ENTRY_POINT_ERROR	<code>false</code>	Ignore entry point errors and don't log them in Application Insights.

FUNCTIONS_REQUEST_BODY_SIZE_LIMIT

Overrides the default limit on the body size of requests sent to HTTP endpoints. The value is given in bytes, with a default maximum request size of 104857600 bytes.

[Expand table](#)

Key	Sample value
FUNCTIONS_REQUEST_BODY_SIZE_LIMIT	<code>250000000</code>

FUNCTIONS_V2_COMPATIBILITY_MODE

Important

This setting is no longer supported. It was originally provided to enable a short-term workaround for apps that targeted the v2.x runtime to be able to instead run on the v3.x runtime while it was still supported. Except for legacy apps that run on version 1.x, all function apps must run on version 4.x of the Functions runtime:

`FUNCTIONS_EXTENSION_VERSION=~4`. For more information, see [Azure Functions runtime versions overview](#).

FUNCTIONS_WORKER_PROCESS_COUNT

Specifies the maximum number of language worker processes, with a default value of `1`. The maximum value allowed is `10`. Function invocations are evenly distributed among language worker processes. Language worker processes are spawned every 10 seconds until the count set by `FUNCTIONS_WORKER_PROCESS_COUNT` is reached. Using multiple language worker processes isn't the same as [scaling](#). Consider using this setting when your workload has a mix of CPU-bound and I/O-bound invocations. This setting applies to all language runtimes, except for .NET running in process (`FUNCTIONS_WORKER_RUNTIME=dotnet`).

[Expand table](#)

Key	Sample value
FUNCTIONS_WORKER_PROCESS_COUNT	2

FUNCTIONS_WORKER_RUNTIME

The language or language stack of the worker runtime to load in the function app. This corresponds to the language being used in your application (for example, `python`). Starting with version 2.x of the Azure Functions runtime, a given function app can only support a single language.

[Expand table](#)

Key	Sample value
FUNCTIONS_WORKER_RUNTIME	node

Valid values:

[Expand table](#)

Value	Language/language stack
<code>dotnet</code>	C# (class library) C# (script)
<code>dotnet-isolated</code>	C# (isolated worker process)
<code>java</code>	Java
<code>node</code>	JavaScript TypeScript
<code>powershell</code>	PowerShell
<code>python</code>	Python
<code>custom</code>	Other

FUNCTIONS_WORKER_SHARED_MEMORY_DATA_TRANSFER_EN

This setting enables the Python worker to use shared memory to improve throughput. Enable shared memory when your Python function app is hitting memory bottlenecks.

[Expand table](#)

Key	Sample value
FUNCTIONS_WORKER_SHARED_MEMORY_DATA_TRANSFER_ENABLED	1

With this setting enabled, you can use the `DOCKER_SHM_SIZE` setting to set the shared memory size. To learn more, see [Shared memory](#).

JAVA_OPTS

Used to customize the Java virtual machine (JVM) used to run your Java functions when running on a [Premium plan](#) or [Dedicated plan](#). When running on a Consumption plan, instead use `languageWorkers_java_arguments`. For more information, see [Customize JVM](#).

languageWorkers_java_arguments

Used to customize the Java virtual machine (JVM) used to run your Java functions when running on a [Consumption plan](#). This setting does increase the cold start times for Java functions running in a Consumption plan. For a Premium or Dedicated plan, instead use `JAVA_OPTS`. For more information, see [Customize JVM](#).

MDMaxBackgroundUpgradePeriod

Controls the managed dependencies background update period for PowerShell function apps, with a default value of `7.00:00:00` (weekly).

Each PowerShell worker process initiates checking for module upgrades on the PowerShell Gallery on process start and every `MDMaxBackgroundUpgradePeriod` after that. When a new module version is available in the PowerShell Gallery, it's installed to the file system and made available to PowerShell workers. Decreasing this value lets your function app get newer module versions sooner, but it also increases the app resource usage (network I/O, CPU, storage). Increasing this value decreases the app's resource usage, but it can also delay delivering new module versions to your app.

[Expand table](#)

Key	Sample value
<code>MDMaxBackgroundUpgradePeriod</code>	<code>7.00:00:00</code>

To learn more, see [Dependency management](#).

MDNewSnapshotCheckPeriod

Specifies how often each PowerShell worker checks whether managed dependency upgrades have been installed. The default frequency is `01:00:00` (hourly).

After new module versions are installed to the file system, every PowerShell worker process must be restarted. Restarting PowerShell workers affects your app availability as it can interrupt current function execution. Until all PowerShell worker processes are restarted, function invocations can use either the old or the new module versions. Restarting all PowerShell workers completes within `MDNewSnapshotCheckPeriod`.

Within every `MDNewSnapshotCheckPeriod`, the PowerShell worker checks whether or not managed dependency upgrades have been installed. When upgrades have been installed, a restart is initiated. Increasing this value decreases the frequency of interruptions because of restarts. However, the increase might also increase the time during which function invocations could use either the old or the new module versions, nondeterministically.

[Expand table](#)

Key	Sample value
<code>MDNewSnapshotCheckPeriod</code>	<code>01:00:00</code>

To learn more, see [Dependency management](#).

MDMinBackgroundUpgradePeriod

The period of time after a previous managed dependency upgrade check before another upgrade check is started, with a default of `1.00:00:00` (daily).

To avoid excessive module upgrades on frequent Worker restarts, checking for module upgrades isn't performed when any worker has already initiated that check in the last `MDMinBackgroundUpgradePeriod`.

[Expand table](#)

Key	Sample value
MDMinBackgroundUpgradePeriod	1.00:00:00

To learn more, see [Dependency management](#).

PIP_INDEX_URL

This setting lets you override the base URL of the Python Package Index, which by default is `https://pypi.org/simple`. Use this setting when you need to run a remote build using custom dependencies. These custom dependencies can be in a package index repository compliant with PEP 503 (the simple repository API) or in a local directory that follows the same format.

[Expand table](#)

Key	Sample value
PIP_INDEX_URL	<code>http://my.custom.package.repo/simple</code>

To learn more, see [pip documentation for --index-url](#) and using [Custom dependencies](#) in the Python developer reference.

PIP_EXTRA_INDEX_URL

The value for this setting indicates an extra index URL for custom packages for Python apps, to use in addition to the `--index-url`. Use this setting when you need to run a remote build using custom dependencies that are found in an extra package index. Should follow the same rules as `--index-url`.

[Expand table](#)

Key	Sample value
PIP_EXTRA_INDEX_URL	<code>http://my.custom.package.repo/simple</code>

To learn more, see [pip documentation for --extra-index-url](#) and [Custom dependencies](#) in the Python developer reference.

PROJECT

A [continuous deployment](#) setting that tells the Kudu deployment service the folder in a connected repository to location the deployable project.

[Expand table](#)

Key	Sample value
PROJECT	<code>WebProject/WebProject.csproj</code>

PYTHON_ISOLATE_WORKER_DEPENDENCIES

The configuration is specific to Python function apps. It defines the prioritization of module loading order. By default, this value is set to `0`.

[Expand table](#)

Key	Value	Description
PYTHON_ISOLATE_WORKER_DEPENDENCIES	0	Prioritize loading the Python libraries from internal Python worker's dependencies, which is the default behavior. Third-party libraries defined in requirements.txt might be shadowed.
PYTHON_ISOLATE_WORKER_DEPENDENCIES	1	Prioritize loading the Python libraries from application's package defined in requirements.txt. This prevents your libraries from colliding with internal Python worker's libraries.

PYTHON_ENABLE_DEBUG_LOGGING

Enables debug-level logging in a Python function app. A value of 1 enables debug-level logging. Without this setting or with a value of 0, only information and higher-level logs are sent from the Python worker to the Functions host. Use this setting when debugging or tracing your Python function executions.

When debugging Python functions, make sure to also set a debug or trace [logging level](#) in the host.json file, as needed. To learn more, see [How to configure monitoring for Azure Functions](#).

PYTHON_ENABLE_WORKER_EXTENSIONS

The configuration is specific to Python function apps. Setting this to 1 allows the worker to load in [Python worker extensions](#) defined in requirements.txt. It enables your function app to access new features provided by third-party packages. It can also change the behavior of function load and invocation in your app. Ensure the extension you choose is trustworthy as you bear the risk of using it. Azure Functions gives no express warranties to any extensions. For how to use an extension, visit the extension's manual page or readme doc. By default, this value sets to 0.

[Expand table](#)

Key	Value	Description
PYTHON_ENABLE_WORKER_EXTENSIONS	0	Disable any Python worker extension.
PYTHON_ENABLE_WORKER_EXTENSIONS	1	Allow Python worker to load extensions from requirements.txt.

PYTHON_THREADPOOL_THREAD_COUNT

Specifies the maximum number of threads that a Python language worker would use to execute function invocations, with a default value of 1 for Python version 3.8 and below. For Python version 3.9 and above, the value is set to None. This setting doesn't guarantee the number of threads that would be set during executions. The setting allows Python to expand the number of threads to the specified value. The setting only applies to Python functions apps. Additionally, the setting applies to synchronous functions invocation and not for coroutines.

[Expand table](#)

Key	Sample value	Max value
PYTHON_THREADPOOL_THREAD_COUNT	2	32

SCALE_CONTROLLER_LOGGING_ENABLED

This setting is currently in preview.

This setting controls logging from the Azure Functions scale controller. For more information, see [Scale controller logs](#).

[Expand table](#)

Key	Sample value
SCALE_CONTROLLER_LOGGING_ENABLED	AppInsights:Verbose

The value for this key is supplied in the format <DESTINATION>:<VERBOSITY>, which is defined as follows:

[Expand table](#)

Property	Description
<DESTINATION>	<p>The destination to which logs are sent. Valid values are <code>AppInsights</code> and <code>Blob</code>.</p> <p>When you use <code>AppInsights</code>, ensure that the Application Insights is enabled in your function app.</p> <p>When you set the destination to <code>Blob</code>, logs are created in a blob container named <code>azure-functions-scale-controller</code> in the default storage account set in the <code>AzureWebJobsStorage</code> application setting.</p>
<VERBOSITY>	<p>Specifies the level of logging. Supported values are <code>None</code>, <code>Warning</code>, and <code>Verbose</code>.</p> <p>When set to <code>Verbose</code>, the scale controller logs a reason for every change in the worker count, and information about the triggers that factor into those decisions. Verbose logs include trigger warnings and the hashes used by the triggers before and after the scale controller runs.</p>

Tip

Keep in mind that while you leave scale controller logging enabled, it impacts the [potential costs of monitoring your function app](#). Consider enabling logging until you have collected enough data to understand how the scale controller is behaving, and then disabling it.

SCM_DO_BUILD_DURING_DEPLOYMENT

Controls remote build behavior during deployment. When `SCM_DO_BUILD_DURING_DEPLOYMENT` is set to `true`, the project is built remotely during deployment.

[Expand table](#)

Key	Sample value
SCM_DO_BUILD_DURING_DEPLOYMENT	true

SCM_LOGSTREAM_TIMEOUT

Controls the timeout, in seconds, when connected to streaming logs. The default value is 7200 (2 hours).

[Expand table](#)

Key	Sample value
SCM_LOGSTREAM_TIMEOUT	1800

The above sample value of `1800` sets a timeout of 30 minutes. For more information, see [Enable streaming execution logs in Azure Functions](#).

WEBSITE_CONTENTAZUREFILECONNECTIONSTRING

Connection string for storage account where the function app code and configuration are stored in event-driven scaling plans. For more information, see [Storage account connection setting](#).

[Expand table](#)

Key	Sample value
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING	DefaultEndpointsProtocol=https;AccountName=...

This setting is required for Consumption and Elastic Premium plan apps running on both Windows and Linux. It's not required for Dedicated plan apps, which aren't dynamically scaled by Functions.

Changing or removing this setting can cause your function app to not start. To learn more, see [this troubleshooting article](#).

Azure Files doesn't support using managed identity when accessing the file share. For more information, see [Azure Files supported authentication scenarios](#).

WEBSITE_CONTENTOVERVNET

ⓘ Important

WEBSITE_CONTENTOVERVNET is a legacy app setting that has been replaced by the [vnetContentShareEnabled](#) site property.

A value of `1` enables your function app to scale when you have your storage account restricted to a virtual network. You should enable this setting when restricting your storage account to a virtual network. Only required when using `WEBSITE_CONTENTSHARE` and `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`. To learn more, see [Restrict your storage account to a virtual network](#).

[Expand table](#)

Key	Sample value
WEBSITE_CONTENTOVERVNET	1

This app setting is required on the [Elastic Premium](#) and [Dedicated \(App Service\) plans](#) (Standard and higher). Not supported when running on a [Consumption plan](#).

ⓘ Note

You must take special care when routing to the content share in a storage account shared by multiple function apps in the same plan. For more information, see [Consistent routing through virtual networks](#) in the Storage considerations article.

WEBSITE_CONTENTSHARE

The name of the file share that Functions uses to store function app code and configuration files. This content is required by event-driven scaling plans. Used with `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`. Default is a unique string generated by the runtime, which begins with the function app name. For more information, see [Storage account connection setting](#).

[Expand table](#)

Key	Sample value
WEBSITE_CONTENTSHARE	functionapp091999e2

This setting is required for Consumption and Premium plan apps on both Windows and Linux. It's not required for Dedicated plan apps, which aren't dynamically scaled by Functions.

The share is created when your function app is created. Changing or removing this setting can cause your function app to not start. To learn more, see [this troubleshooting article](#).

The following considerations apply when using an Azure Resource Manager (ARM) template or Bicep file to create a function app during deployment:

- When you don't set a `WEBSITE_CONTENTSHARE` value for the main function app or any apps in slots, unique share values are generated for you. Not setting `WEBSITE_CONTENTSHARE` is *the recommended approach* for an ARM template deployment.
- There are scenarios where you must set the `WEBSITE_CONTENTSHARE` value to a predefined value, such as when you [use a secured storage account in a virtual network](#). In this case, you must set a unique share name for the main function app and the app for each deployment slot. In the case of a storage account secured by a virtual network, you must also create the share itself as part of your automated deployment. For more information, see [Secured deployments](#).
- Don't make `WEBSITE_CONTENTSHARE` a slot setting.
- When you specify `WEBSITE_CONTENTSHARE`, the value must follow [this guidance for share names](#).

WEBSITE_DNS_SERVER

Sets the DNS server used by an app when resolving IP addresses. This setting is often required when using certain networking functionality, such as [Azure DNS private zones](#) and [private endpoints](#).

[+] [Expand table](#)

Key	Sample value
WEBSITE_DNS_SERVER	168.63.129.16

WEBSITE_ENABLE_BROTLI_ENCODING

Controls whether Brotli encoding is used for compression instead of the default gzip compression. When `WEBSITE_ENABLE_BROTLI_ENCODING` is set to `1`, Brotli encoding is used; otherwise gzip encoding is used.

WEBSITE_FUNCTIONS_ARMCACHE_ENABLED

Disables caching when deploying function apps using Azure Resource Manager (ARM) templates.

[+] [Expand table](#)

Key	Sample value
WEBSITE_FUNCTIONS_ARMCACHE_ENABLED	0

WEBSITE_MAX_DYNAMIC_APPLICATION_SCALE_OUT

The maximum number of instances that the app can scale out to. Default is no limit.

Important

This setting is in preview. An [app property for function max scale out](#) has been added and is the recommended way to limit scale out.

[+] [Expand table](#)

Key	Sample value
WEBSITE_MAX_DYNAMIC_APPLICATION_SCALE_OUT	5

WEBSITE_NODE_DEFAULT_VERSION

Windows only. Sets the version of Node.js to use when running your function app on Windows. You should use a tilde (~) to have the runtime use the latest available version of the targeted major version. For example, when set to ~18, the latest version of Node.js 18 is used. When a major version is targeted with a tilde, you don't have to manually update the minor version.

[Expand table](#)

Key	Sample value
WEBSITE_NODE_DEFAULT_VERSION	~18

WEBSITE_OVERRIDE_STICKY_DIAGNOSTICS_SETTINGS

When performing a [slot swap](#) on a function app running on a Premium plan, the swap can fail when the dedicated storage account used by the app is network restricted. This failure is caused by a legacy [application logging feature](#), which is shared by both Functions and App Service. This setting overrides that legacy logging feature and allows the swap to occur.

[Expand table](#)

Key	Sample value
WEBSITE_OVERRIDE_STICKY_DIAGNOSTICS_SETTINGS	0

Add `WEBSITE_OVERRIDE_STICKY_DIAGNOSTICS_SETTINGS` with a value of 0 to all slots to make sure that legacy diagnostic settings don't block your swaps. You can also add this setting and value to just the production slot as a [deployment slot \(sticky\) setting](#).

WEBSITE_OVERRIDE_STICKY_EXTENSION VERSIONS

By default, the version settings for function apps are specific to each slot. This setting is used when upgrading functions by using [deployment slots](#). This prevents unanticipated behavior due to changing versions after a swap. Set to 0 in production and in the slot to make sure that all version settings are also swapped. For more information, see [Upgrade using slots](#).

[Expand table](#)

Key	Sample value
WEBSITE_OVERRIDE_STICKY_EXTENSION VERSIONS	0

WEBSITE_RUN_FROM_PACKAGE

Enables your function app to run from a package file, which can be locally mounted or deployed to an external URL.

[Expand table](#)

Key	Sample value
WEBSITE_RUN_FROM_PACKAGE	1

Valid values are either a URL that resolves to the location of an external deployment package file, or 1. When set to 1, the package must be in the d:\home\data\SitePackages folder. When you use zip deployment with WEBSITE_RUN_FROM_PACKAGE enabled, the package is automatically uploaded to this location. In preview, this setting was named WEBSITE_RUN_FROM_ZIP. For more information, see [Run your functions from a package file](#).

When you deploy from an external package URL, you must also manually sync triggers. For more information, see [Trigger syncing](#).

WEBSITE_SKIP_CONTENTSHARE_VALIDATION

The WEBSITE_CONTENTAZUREFILECONNECTIONSTRING and WEBSITE_CONTENTSHARE settings have extra validation checks to ensure that the app can be properly started. Creation of application settings fail when the function app can't properly call out to the downstream Storage Account or Key Vault due to networking constraints or other limiting factors. When WEBSITE_SKIP_CONTENTSHARE_VALIDATION is set to 1, the validation check is skipped; otherwise the value defaults to 0 and the validation takes place.

[Expand table](#)

Key	Sample value
WEBSITE_SKIP_CONTENTSHARE_VALIDATION	1

If validation is skipped and either the connection string or content share isn't valid, the app won't be able to start properly. In this case, functions return HTTP 500 errors. For more information, see [Troubleshoot error: "Azure Functions Runtime is unreachable"](#)

WEBSITE_SLOT_NAME

Read-only. Name of the current deployment slot. The name of the production slot is Production.

[Expand table](#)

Key	Sample value
WEBSITE_SLOT_NAME	Production

WEBSITE_TIME_ZONE

Allows you to set the timezone for your function app.

[Expand table](#)

Key	OS	Sample value
WEBSITE_TIME_ZONE	Windows	Eastern Standard Time
WEBSITE_TIME_ZONE	Linux	America/New_York

The default time zone used with the CRON expressions is Coordinated Universal Time (UTC). To have your CRON expression based on another time zone, create an app setting for your function app named WEBSITE_TIME_ZONE.

The value of this setting depends on the operating system and plan on which your function app runs.

[Expand table](#)

Operating system	Plan	Value
Windows	All	Set the value to the name of the desired time zone as given by the second line from each pair given by the Windows command <code>tzutil.exe /L</code>
Linux	Premium	Set the value to the name of the desired time zone as shown in the tz database .
	Dedicated	

 Note

`WEBSITE_TIME_ZONE` and `TZ` are not currently supported when running on Linux in a Consumption plan. In this case, setting `WEBSITE_TIME_ZONE` or `TZ` can create SSL-related issues and cause metrics to stop working for your app.

For example, Eastern Time in the US (represented by `Eastern Standard Time` (Windows) or `America/New_York` (Linux)) currently uses UTC-05:00 during standard time and UTC-04:00 during daylight time. To have a timer trigger fire at 10:00 AM Eastern Time every day, create an app setting for your function app named `WEBSITE_TIME_ZONE`, set the value to `Eastern Standard Time` (Windows) or `America/New_York` (Linux), and then use the following NCrontab expression:

```
"0 0 10 * * *"
```

When you use `WEBSITE_TIME_ZONE` the time is adjusted for time changes in the specific timezone, including daylight saving time and changes in standard time.

WEBSITE_USE_PLACEHOLDER

Indicates whether to use a specific [cold start](#) optimization when running on the [Consumption plan](#). Set to `0` to disable the cold-start optimization on the Consumption plan.

[Expand table](#)

Key	Sample value
WEBSITE_USE_PLACEHOLDER	1

WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED

Indicates whether to use a specific [cold start](#) optimization when running .NET isolated worker process functions on the [Consumption plan](#). Set to `0` to disable the cold-start optimization on the Consumption plan.

[Expand table](#)

Key	Sample value
WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED	1

WEBSITE_VNET_ROUTE_ALL

 Important

WEBSITE_VNET_ROUTE_ALL is a legacy app setting that has been replaced by the [vnetRouteAllEnabled](#) site setting.

Indicates whether all outbound traffic from the app is routed through the virtual network. A setting value of `1` indicates that all application traffic is routed through the virtual network. You'll need this setting when configuring [Regional virtual network integration](#) in the Elastic Premium and Dedicated hosting plans. It's also used when a [virtual network NAT gateway](#) is used to define a static outbound IP address.

[\[+\] Expand table](#)

Key	Sample value
WEBSITE_VNET_ROUTE_ALL	<code>1</code>

WEBSITES_ENABLE_APP_SERVICE_STORAGE

Indicates whether the `/home` directory is shared across scaled instances, with a default value of `true`. You should set this to `false` when deploying your function app in a container.

App Service site settings

Some configurations must be maintained at the App Service level as site settings, such as language versions. These settings are managed in the portal, by using REST APIs, or by using Azure CLI or Azure PowerShell. The following are site settings that could be required, depending on your runtime language, OS, and versions:

alwaysOn

On a function app running in a [Dedicated \(App Service\) plan](#), the Functions runtime goes idle after a few minutes of inactivity, at which point only requests to an HTTP trigger *wakes-up* your function app. To make sure that your non-HTTP triggered functions run correctly, including Timer trigger functions, enable Always On for the function app by setting the `alwaysOn` site setting to a value of `true`.

linuxFxVersion

For function apps running on Linux, `linuxFxVersion` indicates the language and version for the language-specific worker process. This information is used, along with `FUNCTIONS_EXTENSION_VERSION`, to determine which specific Linux container image is installed to run your function app. This setting can be set to a predefined value or a custom image URI.

This value is set for you when you create your Linux function app. You might need to set it for ARM template and Bicep deployments and in certain upgrade scenarios.

Valid linuxFxVersion values

You can use the following Azure CLI command to see a table of current `linuxFxVersion` values, by supported Functions runtime version:

Azure CLI

```
az functionapp list-runtimes --os linux --query "[].{stack:join(' ', [runtime, version]),  
LinuxFxVersion:linux_fx_version, SupportedFunctionsVersions:to_string(supported_functions_versions[])}" --  
output table
```

The previous command requires you to upgrade to version 2.40 of the Azure CLI.

Custom images

When you create and maintain your own custom linux container for your function app, the `linuxFxVersion` value is instead in the format `DOCKER|<IMAGE_URI>`, as in the following example:

```
linuxFxVersion = "DOCKER|contoso.com/azurefunctionsimage:v1.0.0"
```

This indicates the registry source of the deployed container. For more information, see [Working with containers and Azure Functions](#).

 **Important**

When creating your own containers, you are required to keep the base image of your container updated to the latest supported base image. Supported base images for Azure Functions are language-specific and are found in the [Azure Functions base image repos](#).

The Functions team is committed to publishing monthly updates for these base images. Regular updates include the latest minor version updates and security fixes for both the Functions runtime and languages. You should regularly update your container from the latest base image and redeploy the updated version of your container.

netFrameworkVersion

Sets the specific version of .NET for C# functions. For more information, see [Update your function app in Azure](#).

powerShellVersion

Sets the specific version of PowerShell on which your functions run. For more information, see [Changing the PowerShell version](#).

When running locally, you instead use the `FUNCTIONS_WORKER_RUNTIME_VERSION` setting in the local.settings.json file.

vnetContentShareEnabled

Apps running in a Premium plan use a file share to store content. The name of this content share is stored in the `WEBSITE_CONTENTSHARE` app setting and its connection string is stored in `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`. To route traffic between your function app and content share through a virtual network, you must also set `vnetContentShareEnabled` to `true`. Enabling this site property is a requirement when [restricting your storage account to a virtual network](#) in the Elastic Premium and Dedicated hosting plans.

 **Note**

You must take special care when routing to the content share in a storage account shared by multiple function apps in the same plan. For more information, see [Consistent routing through virtual networks](#) in the Storage considerations article.

This site property replaces the legacy `WEBSITE_CONTENTOVERVNET` setting.

vnetImagePullEnabled

Functions [supports function apps running in Linux containers](#). To connect and pull from a container registry inside a virtual network, you must set `vnetImagePullEnabled` to `true`. This site property is supported in the Elastic Premium and Dedicated

hosting plans. The Flex Consumption plan doesn't rely on site properties or app settings to configure Networking. For more information, see [Flex Consumption plan deprecations](#).

vnetRouteAllEnabled

Indicates whether all outbound traffic from the app is routed through the virtual network. A setting value of `true` indicates that all application traffic is routed through the virtual network. Use this setting when configuring [Regional virtual network integration](#) in the Elastic Premium and Dedicated plans. It's also used when a [virtual network NAT gateway](#) is used to define a static outbound IP address. For more information, see [Configure application routing](#).

This site setting replaces the legacy [WEBSITE_VNET_ROUTE_ALL](#) setting.

Flex Consumption plan deprecations

In the [Flex Consumption plan](#), these site properties and application settings are deprecated and shouldn't be used when creating function app resources:

[+] [Expand table](#)

Setting/property	Reason
<code>ENABLE_ORYX_BUILD</code>	Replaced by the <code>remoteBuild</code> parameter when deploying in Flex Consumption
<code>FUNCTIONS_EXTENSION_VERSION</code>	App Setting is set by the backend. A value of <code>~1</code> can be ignored.
<code>FUNCTIONS_WORKER_RUNTIME</code>	Replaced by <code>name</code> in <code>properties.functionAppConfig.runtime</code>
<code>FUNCTIONS_WORKER_RUNTIME_VERSION</code>	Replaced by <code>version</code> in <code>properties.functionAppConfig.runtime</code>
<code>FUNCTIONS_MAX_HTTP_CONCURRENCY</code>	Replaced by scale and concurrency's trigger section
<code>FUNCTIONS_WORKER_PROCESS_COUNT</code>	Setting not valid
<code>FUNCTIONS_WORKER_DYNAMIC_CONCURRENCY_ENABLED</code>	Setting not valid
<code>SCM_DO_BUILD_DURING_DEPLOYMENT</code>	Replaced by the <code>remoteBuild</code> parameter when deploying in Flex Consumption
<code>WEBSITE_CONTENTAZUREFILECONNECTIONSTRING</code>	Replaced by <code>functionAppConfig</code> 's deployment section
<code>WEBSITE_CONTENTOVERVNET</code>	Not used for networking in Flex Consumption
<code>WEBSITE_CONTENTSHARE</code>	Replaced by <code>functionAppConfig</code> 's deployment section
<code>WEBSITE_DNS_SERVER</code>	DNS is inherited from the integrated VNet in Flex
<code>WEBSITE_NODE_DEFAULT_VERSION</code>	Replaced by <code>version</code> in <code>properties.functionAppConfig.runtime</code>
<code>WEBSITE_RUN_FROM_PACKAGE</code>	Not used for deployments in Flex Consumption
<code>WEBSITE_SKIP_CONTENTSHARE_VALIDATION</code>	Content share is not used in Flex Consumption
<code>WEBSITE_VNET_ROUTE_ALL</code>	Not used for networking in Flex Consumption
<code>properties.alwaysOn</code>	Not valid
<code>properties.containerSize</code>	Renamed as <code>instanceMemoryMB</code>
<code>properties.ftpssState</code>	FTPS not supported
<code>properties.isReserved</code>	Not valid
<code>properties.IsXenon</code>	Not valid
<code>properties.javaVersion</code>	Replaced by <code>version</code> in <code>properties.functionAppConfig.runtime</code>

Setting/property	Reason
<code>properties.LinuxFxVersion</code>	Replaced by <code>properties.functionAppConfig.runtime</code>
<code>properties.netFrameworkVersion</code>	Replaced by <code>version</code> in <code>properties.functionAppConfig.runtime</code>
<code>properties.powerShellVersion</code>	Replaced by <code>version</code> in <code>properties.functionAppConfig.runtime</code>
<code>properties.siteConfig.functionAppScaleLimit</code>	Renamed as <code>maximumInstanceCount</code>
<code>properties.siteConfig.preWarmedInstanceCount</code>	Renamed as <code>alwaysReadyInstances</code>
<code>properties.use32BitWorkerProcess</code>	32-bit not supported
<code>properties.vnetBackupRestoreEnabled</code>	Not used for networking in Flex Consumption
<code>properties.vnetContentShareEnabled</code>	Not used for networking in Flex Consumption
<code>properties.vnetImagePullEnabled</code>	Not used for networking in Flex Consumption
<code>properties.vnetRouteAllEnabled</code>	Not used for networking in Flex Consumption
<code>properties.windowsFxVersion</code>	Not valid

Next steps

[Learn how to update app settings](#)

[See configuration settings in the host.json file](#)

[See other app settings for App Service apps](#)

Feedback

Was this page helpful?

 Yes
 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Overview of Azure functions for Azure Cache for Redis

Article • 07/11/2024

This article describes how to use Azure Cache for Redis with Azure Functions to create optimized serverless and event-driven architectures.

Azure Functions provide an event-driven programming model where triggers and bindings are key features. With Azure Functions, you can easily build event-driven serverless applications. Azure Cache for Redis provides a set of building blocks and best practices for building distributed applications, including microservices, state management, pub/sub messaging, and more.

Azure Cache for Redis can be used as a trigger for Azure Functions, allowing you to initiate a serverless workflow. This functionality can be highly useful in data architectures like a write-behind cache, or any event-based architectures.

You can integrate Azure Cache for Redis and Azure Functions to build functions that react to events from Azure Cache for Redis or external systems.

[] Expand table

Action	Direction	Support level
Trigger on Redis pub sub messages	Trigger	Preview
Trigger on Redis lists	Trigger	Preview
Trigger on Redis streams	Trigger	Preview
Read a cached value	Input	Preview
Write a values to cache	Output	Preview

Scope of availability for functions triggers and bindings

[] Expand table

Tier	Basic	Standard, Premium	Enterprise, Enterprise Flash
Pub/Sub	Yes	Yes	Yes

Tier	Basic	Standard, Premium	Enterprise, Enterprise Flash
Lists	Yes	Yes	Yes
Streams	Yes	Yes	Yes
Bindings	Yes	Yes	Yes

ⓘ Important

Redis triggers are currently only supported for functions running in either a [Elastic Premium plan](#) or a dedicated [App Service plan](#).

Install extension

Isolated worker model

Functions run in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing [this NuGet package](#).

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Redis --prerelease
```

Install bundle

Redis connection string

Azure Cache for Redis triggers and bindings have a required property for the cache connection string. The connection string can be found on the [Access keys](#) menu in the Azure Cache for Redis portal. The Redis trigger or binding looks for an environmental variable holding the connection string with the name passed to the `Connection` parameter.

In local development, the `Connection` can be defined using the [local.settings.json](#) file. When deployed to Azure, [application settings](#) can be used.

When connecting to a cache instance with an Azure function, you can use three types of connections in your deployments: Connection string, System-assigned managed identity, and User-assigned managed identity

For local development, you can also use service principal secrets.

Use the `appsettings` to configure each of the following types of client authentication, assuming the `Connection` was set to `Redis` in the function.

Connection string

JSON

```
"Redis": "<cacheName>.redis.cache.windows.net:6380,password=..."
```

System-assigned managed identity

JSON

```
"Redis:redisHostName": "<cacheName>.redis.cache.windows.net",
"Redis:principalId": "<principalId>"
```

User-assigned managed identity

JSON

```
"Redis:redisHostName": "<cacheName>.redis.cache.windows.net",
"Redis:principalId": "<principalId>",
"Redis:clientId": "<clientId>"
```

Service Principal Secret

Connections using Service Principal Secrets are only available during local development.

JSON

```
"Redis:redisHostName": "<cacheName>.redis.cache.windows.net",
"Redis:principalId": "<principalId>",
"Redis:clientId": "<clientId>",
"Redis:tenantId": "<tenantId>",
"Redis:clientSecret": "<clientSecret>"
```

Related content

- [Introduction to Azure Functions](#)
 - [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
 - [Tutorial: Create a write-behind cache by using Azure Functions and Azure Cache for Redis](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RedisPubSubTrigger for Azure Functions

Article • 07/12/2024

Redis features [publish/subscribe functionality](#) that enables messages to be sent to Redis and broadcast to subscribers.

For more information about Azure Cache for Redis triggers and bindings, [Redis Extension for Azure Functions](#).

Scope of availability for functions triggers

[] Expand table

Tier	Basic	Standard, Premium	Enterprise, Enterprise Flash
Pub/Sub Trigger	Yes	Yes	Yes

⚠ Warning

This trigger isn't supported on a [consumption plan](#) because Redis PubSub requires clients to always be actively listening to receive all messages. For consumption plans, your function might miss certain messages published to the channel.

Examples

[] Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

ⓘ Important

For .NET functions, using the *isolated worker* model is recommended over the *in-process* model. For a comparison of the *in-process* and *isolated worker* models, see differences between the *isolated worker* model and the *in-process* model for .NET on Azure Functions.

Isolated worker model

This sample listens to the channel `pubsubTest`.

C#

```
using Microsoft.Extensions.Logging;

namespace Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisPubSubTrigger
{
    internal class SimplePubSubTrigger
    {
        private readonly ILogger<SimplePubSubTrigger> logger;

        public SimplePubSubTrigger(ILogger<SimplePubSubTrigger> logger)
        {
            this.logger = logger;
        }

        [Function(nameof(SimplePubSubTrigger))]
        public void Run(
            [RedisPubSubTrigger(Common.connectionStringSetting,
"pubsubTest")] string message)
        {
            logger.LogInformation(message);
        }
    }
}
```

This sample listens to any keyspace notifications for the key `keyspaceTest`.

C#

```
using Microsoft.Extensions.Logging;

namespace Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisPubSubTrigger
{
    internal class KeyspaceTrigger
    {
```

```

    private readonly ILogger<KeyspaceTrigger> logger;

    public KeyspaceTrigger(ILogger<KeyspaceTrigger> logger)
    {
        this.logger = logger;
    }

    [Function(nameof(KeyspaceTrigger))]
    public void Run(
        [RedisPubSubTrigger(Common.ConnectionStringSetting,
    "__keyspace@0__:keyspaceTest")] string message)
    {
        logger.LogInformation(message);
    }
}

```

This sample listens to any `keyevent` notifications for the delete command [DEL ↴](#).

C#

```

using Microsoft.Extensions.Logging;

namespace
Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisPubSubTri
gger
{
    internal class KeyeventTrigger
    {
        private readonly ILogger<KeyeventTrigger> logger;

        public KeyeventTrigger(ILogger<KeyeventTrigger> logger)
        {
            this.logger = logger;
        }

        [Function(nameof(KeyeventTrigger))]
        public void Run(
            [RedisPubSubTrigger(Common.ConnectionStringSetting,
    "__keyevent@0__:del")] string message)
        {
            logger.LogInformation($"Key '{message}' deleted.");
        }
    }
}

```

Attributes

Parameter	Description	Required	Default
Connection	The name of the application setting that contains the cache connection string, such as: <code><cacheName>.redis.cache.windows.net:6380,password...</code>	Yes	
Channel	The pub sub channel that the trigger should listen to. Supports glob-style channel patterns. This field can be resolved using INameResolver .	Yes	

ⓘ Important

The `connection` parameter does not hold the Redis cache connection string itself. Instead, it points to the name of the environment variable that holds the connection string. This makes the application more secure. For more information, see [Redis connection string](#).

Usage

Redis features [publish/subscribe functionality](#) that enables messages to be sent to Redis and broadcast to subscribers. The `RedisPubSubTrigger` enables Azure Functions to be triggered on pub/sub activity. The `RedisPubSubTrigger` subscribes to a specific channel pattern using [PSUBSCRIBE](#), and surfaces messages received on those channels to the function.

Prerequisites and limitations

- The `RedisPubSubTrigger` isn't capable of listening to [keyspace notifications](#) on clustered caches.
- Basic tier functions don't support triggering on `keyspace` or `keyevent` notifications through the `RedisPubSubTrigger`.
- The `RedisPubSubTrigger` isn't supported on a [consumption plan](#) because Redis PubSub requires clients to always be actively listening to receive all messages. For consumption plans, your function might miss certain messages published to the channel.
- Functions with the `RedisPubSubTrigger` shouldn't be scaled out to multiple instances. Each instance listens and processes each pub sub message, resulting in duplicate processing.

⚠ Warning

This trigger isn't supported on a [consumption plan](#) because Redis PubSub requires clients to always be actively listening to receive all messages. For consumption plans, your function might miss certain messages published to the channel.

Triggering on keyspace notifications

Redis offers a built-in concept called [keyspace notifications](#). When enabled, this feature publishes notifications of a wide range of cache actions to a dedicated pub/sub channel. Supported actions include actions that affect specific keys, called *keyspace notifications*, and specific commands, called *keyevent notifications*. A huge range of Redis actions are supported, such as `SET`, `DEL`, and `EXPIRE`. The full list can be found in the [keyspace notification documentation](#).

The `keyspace` and `keyevent` notifications are published with the following syntax:

```
PUBLISH __keyspace@0__:<affectedKey> <command>
PUBLISH __keyevent@0__:<affectedCommand> <key>
```

Because these events are published on pub/sub channels, the `RedisPubSubTrigger` is able to pick them up. See the [RedisPubSubTrigger](#) section for more examples.

ⓘ Important

In Azure Cache for Redis, `keyspace` events must be enabled before notifications are published. For more information, see [Advanced Settings](#).

[] Expand table

Type	Description
<code>string</code>	The channel message serialized as JSON (UTF-8 encoded for byte types) in the format that follows.
<code>Custom</code>	The trigger uses Json.NET serialization to map the message from the channel into the given custom type.

JSON string format

JSON

```
{  
    "SubscriptionChannel": "__keyspace@0__:*",  
    "Channel": "__keyspace@0__:mykey",  
    "Message": "set"  
}
```

Related content

- [Introduction to Azure Functions](#)
- [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
- [Tutorial: Create a write-behind cache by using Azure Functions and Azure Cache for Redis](#)
- [Redis connection string](#)
- [Redis pub sub messages ↗](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RedisListTrigger for Azure Functions

Article • 07/12/2024

The `RedisListTrigger` pops new elements from a list and surfaces those entries to the function.

For more information about Azure Cache for Redis triggers and bindings, [Redis Extension for Azure Functions](#).

Scope of availability for functions triggers

[+] Expand table

Tier	Basic	Standard, Premium	Enterprise, Enterprise Flash
Lists	Yes	Yes	Yes

ⓘ Important

Redis triggers aren't currently supported for functions running in the [Consumption plan](#).

Example

ⓘ Important

For .NET functions, using the *isolated worker* model is recommended over the *in-process* model. For a comparison of the *in-process* and *isolated worker* models, see differences between the *isolated worker* model and the *in-process* model for .NET on Azure Functions.

The following sample polls the key `listTest:`

Isolated worker model

C#

```
using Microsoft.Extensions.Logging;
```

```

namespace Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisListTrigger
{
    public class SimpleListTrigger
    {
        private readonly ILogger<SimpleListTrigger> logger;

        public SimpleListTrigger(ILogger<SimpleListTrigger> logger)
        {
            this.logger = logger;
        }

        [Function(nameof(SimpleListTrigger))]
        public void Run(
            [RedisListTrigger(Common.connectionStringSetting,
"listTest")] string entry)
        {
            logger.LogInformation(entry);
        }
    }
}

```

Attributes

[\[+\] Expand table](#)

Parameter	Description	Required	Default
Connection	The name of the application setting that contains the cache connection string, such as: <code><cacheName>.redis.cache.windows.net:6380,password...</code>	Yes	
Key	Key to read from. This field can be resolved using INameResolver .	Yes	
PollingIntervalInMs	How often to poll Redis in milliseconds.	Optional	1000
MessagesPerWorker	How many messages each functions instance should process. Used to determine how many instances the function should scale to.	Optional	100
Count	Number of entries to pop from Redis at one time. Entries are processed in parallel. Only supported on Redis 6.2+ using the COUNT argument in LPOP and RPOP .	Optional	10

Parameter	Description	Required	Default
ListPopFromBeginning	Determines whether to pop entries from the beginning using LPOP , or to pop entries from the end using RPOP .	Optional	true

See the Example section for complete examples.

Usage

The `RedisListTrigger` pops new elements from a list and surfaces those entries to the function. The trigger polls Redis at a configurable fixed interval, and uses [LPOP](#) and [RPOP](#) to pop entries from the lists.

[\[+\] Expand table](#)

Type	Description
<code>byte[]</code>	The message from the channel.
<code>string</code>	The message from the channel.
<code>Custom</code>	The trigger uses Json.NET serialization to map the message from the channel from a <code>string</code> into a custom type.

Related content

- [Introduction to Azure Functions](#)
- [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
- [Tutorial: Create a write-behind cache by using Azure Functions and Azure Cache for Redis](#)
- [Redis connection string](#)
- [Redis lists](#)

Feedback

Was this page helpful?

[Yes](#)

[No](#)

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

RedisStreamTrigger for Azure Functions

Article • 07/12/2024

The `RedisStreamTrigger` reads new entries from a stream and surfaces those elements to the function.

[+] Expand table

Tier	Basic	Standard, Premium	Enterprise, Enterprise Flash
Streams	Yes	Yes	Yes

ⓘ Important

Redis triggers aren't currently supported for functions running in the [Consumption plan](#).

Example

ⓘ Important

For .NET functions, using the *isolated worker* model is recommended over the *in-process* model. For a comparison of the *in-process* and *isolated worker* models, see differences between the *isolated worker* model and the *in-process* model for .NET on Azure Functions.

[+] Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

Isolated worker model

C#

```
using Microsoft.Extensions.Logging;

namespace Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisStreamTrigger
{
    internal class SimpleStreamTrigger
    {
        private readonly ILogger<SimpleStreamTrigger> logger;

        public SimpleStreamTrigger(ILogger<SimpleStreamTrigger> logger)
        {
            this.logger = logger;
        }

        [Function(nameof(SimpleStreamTrigger))]
        public void Run(
            [RedisStreamTrigger(Common.connectionStringSetting,
"streamKey")] string entry)
        {
            logger.LogInformation(entry);
        }
    }
}
```

Attributes

[] Expand table

Parameters	Description	Required	Default
Connection	The name of the application setting that contains the cache connection string, such as: <code><cacheName>.redis.cache.windows.net:6380,password...</code>	Yes	
Key	Key to read from.	Yes	
PollingIntervalInMs	How often to poll the Redis server in milliseconds.	Optional	1000
MessagesPerWorker	The number of messages each functions worker should process. Used to determine how many workers the function should scale to.	Optional	100
Count	Number of elements to pull from Redis at one time.	Optional	10

Parameters	Description	Required	Default
<code>DeleteAfterProcess</code>	Indicates if the function deletes the stream entries after processing.	Optional	<code>false</code>

See the Example section for complete examples.

Usage

The `RedisStreamTrigger` Azure Function reads new entries from a stream and surfaces those entries to the function.

The trigger polls Redis at a configurable fixed interval, and uses [XREADGROUP ↗](#) to read elements from the stream.

The consumer group for all instances of a function is the name of the function, that is, `SimpleStreamTrigger` for the [StreamTrigger sample ↗](#).

Each functions instance uses the [WEBSITE_INSTANCE_ID](#) or generates a random GUID to use as its consumer name within the group to ensure that scaled out instances of the function don't read the same messages from the stream.

[\[+\] Expand table](#)

Type	Description
<code>byte[]</code>	The message from the channel.
<code>string</code>	The message from the channel.
<code>Custom</code>	The trigger uses Json.NET serialization to map the message from the channel from a <code>string</code> into a custom type.

Related content

- [Introduction to Azure Functions](#)
- [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
- [Using Azure Functions and Azure Cache for Redis to create a write-behind cache](#)
- [Redis connection string](#)
- [Redis streams ↗](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Cache for Redis input binding for Azure Functions

Article • 07/12/2024

When a function runs, the Azure Cache for Redis input binding retrieves data from a cache and passes it to your function as an input parameter.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

ⓘ Important

For .NET functions, using the *isolated worker* model is recommended over the *in-process* model. For a comparison of the *in-process* and *isolated worker* models, see differences between the *isolated worker* model and the *in-process* model for .NET on Azure Functions.

The following code uses the key from the pub/sub trigger to obtain and log the value from an input binding using a `GET` command:

Isolated process

C#

```
using Microsoft.Extensions.Logging;  
namespace
```

```

Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisInputBind
ing
{
    public class SetGetter
    {
        private readonly ILogger<SetGetter> logger;

        public SetGetter(ILogger<SetGetter> logger)
        {
            this.logger = logger;
        }

        [Function(nameof(SetGetter))]
        public void Run(
            [RedisPubSubTrigger(Common.connectionStringSetting,
"__keyevent@0__:set")] string key,
            [RedisInput(Common.connectionStringSetting, "GET
{Message}")] string value)
        {
            logger.LogInformation($"Key '{key}' was set to value
'{value}'");
        }
    }
}

```

More samples for the Azure Cache for Redis input binding are available in the [GitHub repository](#).

Attributes

Note

Not all commands are supported for this binding. At the moment, only read commands that return a single output are supported. The full list can be found [here](#).

 Expand table

Attribute	Description
<code>property</code>	
<code>Connection</code>	The name of the application setting that contains the cache connection string, such as: <code><cacheName>.redis.cache.windows.net:6380,password...</code>
<code>Command</code>	The redis-cli command to be executed on the cache with all arguments separated by spaces, such as: <code>GET key</code> , <code>HGET key field</code> .

See the [Example section](#) for complete examples.

Usage

The input binding expects to receive a string from the cache.

When you use a custom type as the binding parameter, the extension tries to deserialize a JSON-formatted string into the custom type of this parameter.

Related content

- [Introduction to Azure Functions](#)
- [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
- [Tutorial: Create a write-behind cache by using Azure Functions and Azure Cache for Redis](#)
- [Redis connection string](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Cache for Redis output binding for Azure Functions

Article • 07/12/2024

The Azure Cache for Redis output bindings lets you change the keys in a cache based on a set of available trigger on the cache.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

The following example shows a pub/sub trigger on the set event with an output binding to the same Redis instance. The set event triggers the cache and the output binding returns a delete command for the key that triggered the function.

ⓘ Important

For .NET functions, using the *isolated worker* model is recommended over the *in-process* model. For a comparison of the *in-process* and *isolated worker* models, see differences between the *isolated worker* model and the *in-process* model for .NET on Azure Functions.

Isolated process

c#

```
using Microsoft.Extensions.Logging;
```

```

namespace
Microsoft.Azure.Functions.Worker.Extensions.Redis.Samples.RedisOutputBinding
{
    internal class SetDeleter
    {
        [Function(nameof(SetDeleter))]
        [RedisOutput(Common.connectionString, "DEL")]
        public static string Run(
            [RedisPubSubTrigger(Common.connectionString,
"__keyevent@0__:set")] string key,
            ILogger logger)
        {
            logger.LogInformation($"Deleting recently SET key '{key}'");
            return key;
        }
    }
}

```

In-process

C#

```

using Microsoft.Extensions.Logging;

namespace
Microsoft.Azure.WebJobs.Extensions.Redis.Samples.RedisOutputBinding
{
    internal class SetDeleter
    {
        [FunctionName(nameof(SetDeleter))]
        public static void Run(
            [RedisPubSubTrigger(Common.connectionStringSetting,
"__keyevent@0__:set")] string key,
            [Redis(Common.connectionStringSetting, "DEL")] out string[]
arguments,
            ILogger logger)
        {
            logger.LogInformation($"Deleting recently SET key '{key}'");
            arguments = new string[] { key };
        }
    }
}

```

Attributes

Note

All commands are supported for this binding.

The way in which you define an output binding parameter depends on whether your C# functions runs [in-process](#) or in an [isolated worker process](#).

The output binding is defined this way:

 Expand table

Definition	Example	Description
On an <code>out</code> parameter	<code>[Redis(<Connection>, <Command>)] out string <Return_Variable></code>	The string variable returned by the method is a key value that the binding uses to execute the command against the specific cache.

In this case, the type returned by the method is a key value that the binding uses to execute the command against the specific cache.

When your function has multiple output bindings, you can instead apply the binding attribute to the property of a type that is a key value, which the binding uses to execute the command against the specific cache. For more information, see [Multiple output bindings](#).

Regardless of the C# process mode, the same properties are supported by the output binding attribute:

 Expand table

Attribute	Description
<code>property</code>	
<code>Connection</code>	The name of the application setting that contains the cache connection string, such as: <code><cacheName>.redis.cache.windows.net:6380,password...</code>
<code>Command</code>	The redis-cli command to be executed on the cache, such as: <code>DEL</code> .

See the [Example section](#) for complete examples.

Usage

The output returns a string, which is the key of the cache entry on which apply the specific command.

There are three types of connections that are allowed from an Azure Functions instance to a Redis Cache in your deployments. For local development, you can also use service principal secrets. Use the `appsettings` to configure each of the following types of client authentication, assuming the `Connection` was set to `Redis` in the function.

Related content

- [Introduction to Azure Functions](#)
- [Tutorial: Get started with Azure Functions triggers in Azure Cache for Redis](#)
- [Tutorial: Create a write-behind cache by using Azure Functions and Azure Cache for Redis](#)
- [Redis connection string](#)
- [Multiple output bindings](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Cosmos DB bindings for Azure Functions 1.x

Article • 09/27/2023

ⓘ Important

Support will end for version 1.x of the Azure Functions runtime on September 14, 2026 [↗](#). We highly recommend that you [migrate your apps to version 4.x](#) for full support.

This article explains how to work with [Azure Cosmos DB](#) bindings in Azure Functions. Azure Functions supports trigger, input, and output bindings for Azure Cosmos DB.

ⓘ Note

This article is for Azure Functions 1.x. For information about how to use these bindings in Functions 2.x and higher, see [Azure Cosmos DB bindings for Azure Functions 2.x](#).

This binding was originally named DocumentDB. In Azure Functions version 1.x, only the trigger was renamed Azure Cosmos DB; the input binding, output binding, and NuGet package retain the DocumentDB name.

ⓘ Note

Azure Cosmos DB bindings are only supported for use with the SQL API. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API, including [Azure Cosmos DB for MongoDB](#), [Azure Cosmos DB for Apache Cassandra](#), [Azure Cosmos DB for Apache Gremlin](#), and [Azure Cosmos DB for Table](#).

Packages - Functions 1.x

The Azure Cosmos DB bindings for Functions version 1.x are provided in the [Microsoft.Azure.WebJobs.Extensions.DocumentDB](#) [↗](#) NuGet package, version 1.x. Source code for the bindings is in the [azure-webjobs-sdk-extensions](#) [↗](#) GitHub repository.

The following table tells how to add support for this binding in each development environment.

Development environment	To add support in Functions 1.x
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#	Automatic
Portal development	Automatic

Trigger

The Azure Cosmos DB Trigger uses the [Azure Cosmos DB Change Feed](#) to listen for inserts and updates across partitions. The change feed publishes inserts and updates, not deletions.

Trigger - example

C#

The following example shows an [in-process C# function](#) that is invoked when there are inserts or updates in the specified database and collection.

```
C#  
  
using Microsoft.Azure.Documents;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Host;  
using System.Collections.Generic;  
  
namespace CosmosDBSamplesV1  
{  
    public static class CosmosTrigger  
    {  
        [FunctionName("CosmosTrigger")]  
        public static void Run([CosmosDBTrigger(  
            databaseName: "ToDoItems",  
            collectionName: "Items",  
            ConnectionStringSetting = "CosmosDBConnection",  
            LeaseCollectionName = "leases",  
            CreateLeaseCollectionIfNotExists =  
true)]IReadOnlyList<Document> documents,  
            TraceWriter log)  
        {  
            if (documents != null && documents.Count > 0)
```

```
        {
            log.Info($"Documents modified: {documents.Count}");
            log.Info($"First document Id: {documents[0].Id}");
        }
    }
}
```

Trigger - attributes

C#

For [in-process C# class libraries](#), use the `CosmosDBTrigger` attribute.

The attribute's constructor takes the database name and collection name. For information about those settings and other properties that you can configure, see [Trigger - configuration](#). Here's a `CosmosDBTrigger` attribute example in a method signature:

C#

```
[FunctionName("DocumentUpdates")]
public static void Run(
    [CosmosDBTrigger("database", "collection",
ConnectionStringSetting = "myCosmosDB")]
    IReadOnlyList<Document> documents,
    TraceWriter log)
{
    ...
}
```

For a complete example, see [Trigger - C# example](#).

Trigger - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `CosmosDBTrigger` attribute.

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to <code>cosmosDBTrigger</code> .
<code>direction</code>	n/a	Must be set to <code>in</code> . This parameter is set automatically when you

function.json property	Attribute property	Description
		create the trigger in the Azure portal.
name	n/a	The variable name used in function code that represents the list of documents with changes.
connectionStringSetting	ConnectionStringSetting	The name of an app setting that contains the connection string used to connect to the Azure Cosmos DB account being monitored.
databaseName	DatabaseName	The name of the Azure Cosmos DB database with the collection being monitored.
collectionName	CollectionName	The name of the collection being monitored.
leaseConnectionStringSetting	LeaseConnectionStringSetting	(Optional) The name of an app setting that contains the connection string to the service which holds the lease collection. When not set, the <code>connectionStringSetting</code> value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases collection must have write permissions.
leaseDatabaseName	LeaseDatabaseName	(Optional) The name of the database that holds the collection used to store leases. When not set, the value of the <code>databaseName</code> setting is used. This parameter is automatically set when the binding is created in the portal.
leaseCollectionName	LeaseCollectionName	(Optional) The name of the collection used to store leases. When not set, the value <code>leases</code> is used.
createLeaseCollectionIfNotExists	CreateLeaseCollectionIfNotExists	(Optional) When set to <code>true</code> , the leases collection is automatically

function.json property	Attribute property	Description
		created when it doesn't already exist. The default value is <code>false</code> .
leasesCollectionThroughput	LeasesCollectionThroughput	(Optional) Defines the amount of Request Units to assign when the leases collection is created. This setting is only used When <code>createLeaseCollectionIfNotExists</code> is set to <code>true</code> . This parameter is automatically set when the binding is created using the portal.
leaseCollectionPrefix	LeaseCollectionPrefix	(Optional) When set, it adds a prefix to the leases created in the Lease collection for this Function, effectively allowing two separate Azure Functions to share the same Lease collection by using different prefixes.
feedPollDelay	FeedPollDelay	(Optional) When set, it defines, in milliseconds, the delay in between polling a partition for new changes on the feed, after all current changes are drained. Default is 5000 (5 seconds).
leaseAcquireInterval	LeaseAcquireInterval	(Optional) When set, it defines, in milliseconds, the interval to kick off a task to compute if partitions are distributed evenly among known host instances. Default is 13000 (13 seconds).
leaseExpirationInterval	LeaseExpirationInterval	(Optional) When set, it defines, in milliseconds, the interval for which the lease is taken on a lease representing a partition. If the lease is not renewed within this interval, it will cause it to expire and ownership of the partition will move to another instance. Default is 60000 (60 seconds).
leaseRenewInterval	LeaseRenewInterval	(Optional) When set, it defines, in milliseconds, the renew interval for all leases for partitions

function.json property	Attribute property	Description
		currently held by an instance. Default is 17000 (17 seconds).
checkpointFrequency	CheckpointFrequency	(Optional) When set, it defines, in milliseconds, the interval between lease checkpoints. Default is always after each Function call.
maxItemsPerInvocation	MaxItemsPerInvocation	(Optional) When set, it customizes the maximum amount of items received per Function call.
startFromBeginning	StartFromBeginning	(Optional) When set, it tells the Trigger to start reading changes from the beginning of the history of the collection instead of the current time. This only works the first time the Trigger starts, as in subsequent runs, the checkpoints are already stored. Setting this to <code>true</code> when there are leases already created has no effect.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

Trigger - usage

The trigger requires a second collection that it uses to store *leases* over the partitions. Both the collection being monitored and the collection that contains the leases must be available for the trigger to work.

ⓘ Important

If multiple functions are configured to use an Azure Cosmos DB trigger for the same collection, each of the functions should use a dedicated lease collection or specify a different `LeaseCollectionPrefix` for each function. Otherwise, only one of the functions will be triggered. For information about the prefix, see the [Configuration section](#).

The trigger doesn't indicate whether a document was updated or inserted, it just provides the document itself. If you need to handle updates and inserts differently, you could do that by implementing timestamp fields for insertion or update.

Input

The Azure Cosmos DB input binding uses the SQL API to retrieve one or more Azure Cosmos DB documents and passes them to the input parameter of the function. The document ID or query parameters can be determined based on the trigger that invokes the function.

Input - example

C#

This section contains the following examples:

- [Queue trigger, look up ID from JSON](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [HTTP trigger, look up ID from route data, using SqlQuery](#)
- [HTTP trigger, get multiple docs, using SqlQuery](#)
- [HTTP trigger, get multiple docs, using DocumentClient](#)

The examples refer to a simple `ToDoItem` type:

C#

```
namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Queue trigger, look up ID from JSON

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by a queue message that contains a JSON object. The queue trigger parses the JSON into an object named `ToDoItemLookup`, which contains the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

C#

```
namespace CosmosDBSamplesV1
{
    public class ToDoItemLookup
    {
        public string ToDoItemId { get; set; }
    }
}
```

C#

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromJSON
    {
        [FunctionName("DocByIdFromJSON")]
        public static void Run(
            [QueueTrigger("todoqueueforlookup")] ToDoItemLookup
            ToDoItemLookup,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{ToDoItemId}")][ToDoItem toDoItem,
                TraceWriter log)
        {
            log.Info($"C# Queue trigger function processed Id={toDoItemLookup?.ToDoItemId}");

            if (toDoItem == null)
            {
                log.Info($"ToDo item not found");
            }
            else
            {
                log.Info($"Found ToDo item, Description={toDoItem.Description}");
            }
        }
    }
}
```

HTTP trigger, look up ID from query string

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to

look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

```
C#  
  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.Azure.WebJobs.Host;  
using System.Net;  
using System.Net.Http;  
  
namespace CosmosDBSamplesV1  
{  
    public static class DocByIdFromQueryString  
    {  
        [FunctionName("DocByIdFromQueryString")]  
        public static HttpResponseMessage Run(  
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",  
Route = null)]HttpRequestMessage req,  
            [DocumentDB(  
                databaseName: "ToDoItems",  
                collectionName: "Items",  
                ConnectionStringSetting = "CosmosDBConnection",  
                Id = "{Query.id}")] ToDoItem ToDoItem,  
            TraceWriter log)  
        {  
            log.Info("C# HTTP trigger function processed a request.");  
            if (ToDoItem == null)  
            {  
                log.Info($"ToDo item not found");  
            }  
            else  
            {  
                log.Info($"Found ToDo item, Description={  
                    ToDoItem.Description}");  
            }  
            return req.CreateResponse(HttpStatusCode.OK);  
        }  
    }  
}
```

HTTP trigger, look up ID from route data

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

```
C#
```

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Net;
using System.Net.Http;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromRouteData
    {
        [FunctionName("DocByIdFromRouteData")]
        public static HttpResponseMessage Run(
            [HttpTrigger(
                AuthorizationLevel.Anonymous, "get", "post",
                Route = "todoitems/{id}")]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{id}")] ToDoItem ToDoItem,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");

            if (ToDoItem == null)
            {
                log.Info($"ToDo item not found");
            }
            else
            {
                log.Info($"Found ToDo item, Description={ToDoItem.Description}");
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}

```

[Skip input examples](#)

HTTP trigger, look up ID from route data, using SqlQuery

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

C#

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromRouteDataUsingSqlQuery
    {
        [FunctionName("DocByIdFromRouteDataUsingSqlQuery")]
        public static HttpResponseMessage Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
                Route = "todoitems2/{id}")]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                SqlQuery = "select * from ToDoItems r where r.id =
{id}")]
            IEnumerable<ToDoItem> ToDoItems,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");
            foreach (ToDoItem toItem in ToDoItems)
            {
                log.Info(toItem.Description);
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}

```

[Skip input examples](#)

HTTP trigger, get multiple docs, using SqlQuery

The following example shows a [C# function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

C#

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;

```

```

namespace CosmosDBSamplesV1
{
    public static class DocsBySqlQuery
    {
        [FunctionName("DocsBySqlQuery")]
        public static HttpResponseMessage Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
Route = null)]
            HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                SqlQuery = "SELECT top 2 * FROM c order by c._ts desc")]
                IEnumerable<ToDoItem> ToDoItems,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");
            foreach (ToDoItem toItem in ToDoItems)
            {
                log.Info(toItem.Description);
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}

```

[Skip input examples](#)

HTTP trigger, get multiple docs, using DocumentClient (C#)

The following example shows a [C# function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

C#

```

using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

```

```

namespace CosmosDBSamplesV1
{
    public static class DocsByUsingDocumentClient
    {
        [FunctionName("DocsByUsingDocumentClient")]
        public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
Route = null)]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")]
        DocumentClient client,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");

            Uri collectionUri =
UriFactory.CreateDocumentCollectionUri("ToDoItems", "Items");
            string searchterm = req.GetQueryNameValuePairs()
                .FirstOrDefault(q => string.Compare(q.Key, "searchterm",
true) == 0)
                .Value;

            if (searchterm == null)
            {
                return req.CreateResponse(HttpStatusCode.NotFound);
            }

            log.Info($"Searching for word: {searchterm} using Uri:
{collectionUri.ToString()}");
            IDocumentQuery<ToDoItem> query =
client.CreateDocumentQuery<ToDoItem>(collectionUri)
                .Where(p => p.Description.Contains(searchterm))
                .AsDocumentQuery();

            while (query.HasMoreResults)
            {
                foreach (ToDoItem result in await
query.ExecuteNextAsync())
                {
                    log.Info(result.Description);
                }
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}

```

Input - attributes

In [in-process C# class libraries](#), use the `DocumentDB` attribute.

The attribute's constructor takes the database name and collection name. For information about those settings and other properties that you can configure, see [the following configuration section](#).

Input - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `DocumentDB` attribute.

function.json property	Attribute property	Description
<code>type</code>	<code>n/a</code>	Must be set to <code>documentdb</code> .
<code>direction</code>	<code>n/a</code>	Must be set to <code>in</code> .
<code>name</code>	<code>n/a</code>	Name of the binding parameter that represents the document in the function.
<code>databaseName</code>	<code>DatabaseName</code>	The database containing the document.
<code>collectionName</code>	<code>CollectionName</code>	The name of the collection that contains the document.
<code>id</code>	<code>Id</code>	The ID of the document to retrieve. This property supports binding expressions . Don't set both the <code>id</code> and <code>sqlQuery</code> properties. If you don't set either one, the entire collection is retrieved.
<code>sqlQuery</code>	<code>SqlQuery</code>	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: <code>SELECT * FROM c WHERE c.departmentId = {departmentId}</code> . Don't set both the <code>id</code> and <code>sqlQuery</code> properties. If you don't set either one, the entire collection is retrieved.
<code>connection</code>	<code>ConnectionStringSetting</code>	The name of the app setting containing your Azure Cosmos DB connection string.
<code>partitionKey</code>	<code>PartitionKey</code>	Specifies the partition key value for the lookup. May include binding parameters.

When you're developing locally, add your application settings in the [local.settings.json](#) file in the `values` collection.

Input - usage

C#

When the function exits successfully, any changes made to the input document via named input parameters are automatically persisted.

Output

The Azure Cosmos DB output binding lets you write a new document to an Azure Cosmos DB database using the SQL API.

Output - example

C#

This section contains the following examples:

- Queue trigger, write one doc
- Queue trigger, write docs using `IAsyncCollector`

The examples refer to a simple `ToDoItem` type:

C#

```
namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Queue trigger, write one doc

The following example shows a [C# function](#) that adds a document to a database, using data provided in message from Queue storage.

C#

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System;

namespace CosmosDBSamplesV1
{
    public static class WriteOneDoc
    {
        [FunctionName("WriteOneDoc")]
        public static void Run(
            [QueueTrigger("todoqueueforwrite")] string queueMessage,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")]out
dynamic document,
            TraceWriter log)
        {
            document = new { Description = queueMessage, id =
Guid.NewGuid() };

            log.Info($"C# Queue trigger function inserted one row");
            log.Info($"Description={queueMessage}");
        }
    }
}
```

Queue trigger, write docs using IAsyncCollector

The following example shows a [C# function](#) that adds a collection of documents to a database, using data provided in a queue message JSON.

C#

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Threading.Tasks;

namespace CosmosDBSamplesV1
{
    public static class WriteDocsIAsyncCollector
    {
        [FunctionName("WriteDocsIAsyncCollector")]
        public static async Task Run(
            [QueueTrigger("todoqueueforwritemulti")] ToDoItem[]
todoItemsIn,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
```

```
        ConnectionStringSetting = "CosmosDBConnection")]
        IAsyncCollector<ToDoItem> ToDoItemsOut,
        TraceWriter log)
    {
        log.Info($"C# Queue trigger function processed
{ToDoItemsIn?.Length} items");

        foreach (ToDoItem toItem in ToDoItemsIn)
        {
            log.Info($"Description={toItem.Description}");
            await ToDoItemsOut.AddAsync(toItem);
        }
    }
}
```

Output - attributes

C#

In [in-process C# class libraries](#), use the [DocumentDB](#) attribute.

The attribute's constructor takes the database name and collection name. For information about those settings and other properties that you can configure, see [Output - configuration](#). Here's a [DocumentDB](#) attribute example in a method signature:

C#

```
[FunctionName("QueueToDocDB")]
public static void Run(
    [QueueTrigger("myqueue-items", Connection =
"AzureWebJobsStorage")] string myQueueItem,
    [DocumentDB("ToDoList", "Items", Id = "id",
ConnectionStringSetting = "myCosmosDB")] out dynamic document)
{
    ...
}
```

For a complete example, see [Output](#).

Output - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `DocumentDB` attribute.

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to <code>documentdb</code> .
<code>direction</code>	n/a	Must be set to <code>out</code> .
<code>name</code>	n/a	Name of the binding parameter that represents the document in the function.
<code>databaseName</code>	<code>DatabaseName</code>	The database containing the collection where the document is created.
<code>collectionName</code>	<code>CollectionName</code>	The name of the collection where the document is created.
<code>createIfNotExists</code>	<code>CreateIfNotExists</code>	A boolean value to indicate whether the collection is created when it doesn't exist. The default is <i>false</i> because new collections are created with reserved throughput, which has cost implications. For more information, see the pricing page .
<code>partitionKey</code>	<code>PartitionKey</code>	When <code>CreateIfNotExists</code> is true, defines the partition key path for the created collection.
<code>collectionThroughput</code>	<code>CollectionThroughput</code>	When <code>CreateIfNotExists</code> is true, defines the throughput of the created collection.
<code>connection</code>	<code>ConnectionStringSetting</code>	The name of the app setting containing your Azure Cosmos DB connection string.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

Output - usage

By default, when you write to the output parameter in your function, a document is created in your database. This document has an automatically generated GUID as the document ID. You can specify the document ID of the output document by specifying the `id` property in the JSON object passed to the output parameter.

ⓘ Note

When you specify the ID of an existing document, it gets overwritten by the new output document.

Exceptions and return codes

Binding	Reference
Azure Cosmos DB	HTTP status codes for Azure Cosmos DB

Next steps

- Learn more about serverless database computing with [Azure Cosmos DB](#)
- Learn more about [Azure Functions triggers and bindings](#)

Azure Cosmos DB trigger and bindings for Azure Functions 2.x and higher overview

Article • 09/27/2023

This set of articles explains how to work with [Azure Cosmos DB](#) bindings in Azure Functions 2.x and higher. Azure Functions supports trigger, input, and output bindings for Azure Cosmos DB.

Action	Type
Run a function when an Azure Cosmos DB document is created or modified	Trigger
Read an Azure Cosmos DB document	Input binding
Save changes to an Azure Cosmos DB document	Output binding

ⓘ Note

This reference is for **Azure Functions version 2.x and higher**. For information about how to use these bindings in Functions 1.x, see [Azure Cosmos DB bindings for Azure Functions 1.x](#).

This binding was originally named DocumentDB. In Azure Functions version 2.x and higher, the trigger, bindings, and package are all named Azure Cosmos DB.

Supported APIs

Azure Cosmos DB bindings are only supported for use with Azure Cosmos DB for NoSQL. Support for Azure Cosmos DB for Table is provided by using the [Table storage bindings](#), starting with extension 5.x. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API, including [Azure Cosmos DB for MongoDB](#), [Azure Cosmos DB for Cassandra](#), and [Azure Cosmos DB for Apache Gremlin](#).

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

The process for installing the extension varies depending on the extension version:

Extension 4.x+

This version of the Azure Cosmos DB bindings extension introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

Add the extension to your project by installing the [NuGet package](#), version 4.x.

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see binding type details for the mode and version.

Extension 4.x+

The isolated worker process supports parameter types according to the tables below. Support for binding to types from [Microsoft.Azure.Cosmos](#) is in preview.

Cosmos DB trigger

When you want the function to process a single document, the Cosmos DB trigger can bind to the following types:

Type	Description
JSON serializable types	Functions tries to deserialize the JSON data of the document from the Cosmos DB change feed into a plain-old CLR object (POCO) type.

When you want the function to process a batch of documents, the Cosmos DB trigger can bind to the following types:

Type	Description
<code>IEnumerable<T></code> where <code>T</code> is a JSON serializable type	An enumeration of entities included in the batch. Each entry represents one document from the Cosmos DB change feed.

Cosmos DB input binding

When you want the function to process a single document, the Cosmos DB input binding can bind to the following types:

Type	Description
JSON serializable types	Functions attempts to deserialize the JSON data of the document into a plain-old CLR object (POCO) type.

When you want the function to process multiple documents from a query, the Cosmos DB input binding can bind to the following types:

Type	Description
<code>IEnumerable<T></code> where <code>T</code> is a JSON serializable type	An enumeration of entities returned by the query. Each entry represents one document.
<code>CosmosClient</code> ¹	A client connected to the Cosmos DB account.
<code>Database</code> ¹	A client connected to the Cosmos DB database.
<code>Container</code> ¹	A client connected to the Cosmos DB container.

¹ To use these types, you need to reference

[Microsoft.Azure.Functions.Worker.Extensions.CosmosDB 4.4.0 or later](#) and the [common dependencies for SDK type bindings](#).

Cosmos DB output binding

When you want the function to write to a single document, the Cosmos DB output binding can bind to the following types:

Type	Description
JSON serializable types	An object representing the JSON content of a document. Functions attempts to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write to multiple documents, the Cosmos DB output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is JSON serializable type	An array containing multiple documents. Each entry represents one document.

For other output scenarios, create and use types from [Microsoft.Azure.Cosmos](#) directly.

Exceptions and return codes

Binding	Reference
Azure Cosmos DB	HTTP status codes for Azure Cosmos DB

host.json settings

This section describes the configuration settings available for this binding in versions 2.x and higher. Settings in the host.json file apply to all functions in a function app instance. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

Extension 4.x+

```

JSON

{
  "version": "2.0",
  "extensions": {
    "cosmosDB": {
      "connectionMode": "Gateway",
      "userAgentSuffix": "MyDesiredUserAgentStamp"
    }
}

```

```
    }  
}
```

Property	Default	Description
connectionMode	Gateway	The connection mode used by the function when connecting to the Azure Cosmos DB service. Options are <code>Direct</code> and <code>Gateway</code> .
userAgentSuffix	n/a	Adds the specified string value to all requests made by the trigger or binding to the service. This makes it easier for you to track the activity in Azure Monitor, based on a specific function app and filtering by <code>User Agent</code> .

Next steps

- Run a function when an Azure Cosmos DB document is created or modified (Trigger)
- Read an Azure Cosmos DB document (Input binding)
- Save changes to an Azure Cosmos DB document (Output binding)

Azure Cosmos DB trigger for Azure Functions 2.x and higher

Article • 01/19/2024

The Azure Cosmos DB Trigger uses the [Azure Cosmos DB change feed](#) to listen for inserts and updates across partitions. The change feed publishes new and updated items, not including updates from deletions.

For information on setup and configuration details, see the [overview](#).

Cosmos DB scaling decisions for the Consumption and Premium plans are done via target-based scaling. For more information, see [Target-based scaling](#).

Example

The usage of the trigger depends on the extension package version and the C# modality used in your function app, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

The following examples depend on the extension version for the given C# mode.

Extension 4.x+

This example refers to a simple `ToDoItem` type:

```
C#
public class ToDoItem
{
    public string? Id { get; set; }
    public string? Description { get; set; }
}
```

The following function is invoked when there are inserts or updates in the specified database and collection.

```
C#
[Function("CosmosTrigger")]
public void Run([CosmosDBTrigger(
    databaseName: "ToDoItems",
    containerName:"TriggerItems",
    Connection = "CosmosDBConnection",
    LeaseContainerName = "leases",
    CreateLeaseContainerIfNotExists = true)] IReadOnlyList<ToDoItem> todoItems,
    FunctionContext context)
{
    if (todoItems is not null && todoItems.Any())
    {
        foreach (var doc in todoItems)
        {
            _logger.LogInformation("ToDoItem: {desc}", doc.Description);
        }
    }
}
```

```
    }  
}
```

Attributes

Both [in-process](#) and [isolated process](#) C# libraries use the [CosmosDBTriggerAttribute](#) to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Extension 4.x+

[Expand table](#)

Attribute property	Description
Connection	The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account being monitored. For more information, see Connections .
DatabaseName	The name of the Azure Cosmos DB database with the container being monitored.
ContainerName	The name of the container being monitored.
LeaseConnection	(Optional) The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account that holds the lease container. When not set, the <code>Connection</code> value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases container must have write permissions.
LeaseDatabaseName	(Optional) The name of the database that holds the container used to store leases. When not set, the value of the <code>databaseName</code> setting is used.
LeaseContainerName	(Optional) The name of the container used to store leases. When not set, the value <code>leases</code> is used.
CreateLeaseContainerIfNotExists	(Optional) When set to <code>true</code> , the leases container is automatically created when it doesn't already exist. The default value is <code>false</code> . When using Microsoft Entra identities if you set the value to <code>true</code> , creating containers is not an allowed operation and your Function won't be able to start.
LeasesContainerThroughput	(Optional) Defines the number of Request Units to assign when the leases container is created. This setting is only used when <code>CreateLeaseContainerIfNotExists</code> is set to <code>true</code> . This parameter is automatically set when the binding is created using the portal.
LeaseContainerPrefix	(Optional) When set, the value is added as a prefix to the leases created in the Lease container for this function. Using a prefix allows two separate Azure Functions to share the same Lease container by using different prefixes.
FeedPollDelay	(Optional) The time (in milliseconds) for the delay between polling a partition for new changes on the feed, after all current changes are drained. Default is 5,000 milliseconds, or 5 seconds.
LeaseAcquireInterval	(Optional) When set, it defines, in milliseconds, the interval to kick off a task to compute if partitions are distributed evenly among known host instances. Default is 13000 (13 seconds).
LeaseExpirationInterval	(Optional) When set, it defines, in milliseconds, the interval for which the lease is taken on a lease representing a partition. If the lease is not renewed within this interval, it will cause it to expire and ownership of the partition will move to another instance. Default is 60000 (60 seconds).

Attribute property	Description
<code>LeaseRenewInterval</code>	(Optional) When set, it defines, in milliseconds, the renew interval for all leases for partitions currently held by an instance. Default is 17000 (17 seconds).
<code>MaxItemsPerInvocation</code>	(Optional) When set, this property sets the maximum number of items received per Function call. If operations in the monitored container are performed through stored procedures, <code>transaction scope</code> is preserved when reading items from the change feed. As a result, the number of items received could be higher than the specified value so that the items changed by the same transaction are returned as part of one atomic batch.
<code>StartFromBeginning</code>	(Optional) This option tells the Trigger to read changes from the beginning of the container's change history instead of starting at the current time. Reading from the beginning only works the first time the trigger starts, as in subsequent runs, the checkpoints are already stored. Setting this option to <code>true</code> when there are leases already created has no effect.
<code>StartFromTime</code>	(Optional) Gets or sets the date and time from which to initialize the change feed read operation. The recommended format is ISO 8601 with the UTC designator, such as <code>2021-02-16T14:19:29Z</code> . This is only used to set the initial trigger state. After the trigger has a lease state, changing this value has no effect.
<code>PreferredLocations</code>	(Optional) Defines preferred locations (regions) for geo-replicated database accounts in the Azure Cosmos DB service. Values should be comma-separated. For example, "East US, South Central US, North Europe".

See the [Example section](#) for complete examples.

Usage

The trigger requires a second collection that it uses to store *leases* over the partitions. Both the collection being monitored and the collection that contains the leases must be available for the trigger to work.

i Important

If multiple functions are configured to use an Azure Cosmos DB trigger for the same collection, each of the functions should use a dedicated lease collection or specify a different `LeaseCollectionPrefix` for each function. Otherwise, only one of the functions is triggered. For information about the prefix, see the [Attributes section](#).

The trigger doesn't indicate whether a document was updated or inserted, it just provides the document itself. If you need to handle updates and inserts differently, you could do that by implementing timestamp fields for insertion or update.

The parameter type supported by the Azure Cosmos DB trigger depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension 4.x+

When you want the function to process a single document, the Cosmos DB trigger can bind to the following types:

[] [Expand table](#)

Type	Description
JSON serializable types	Functions tries to deserialize the JSON data of the document from the Cosmos DB change feed into a plain-old CLR object (POCO) type.

When you want the function to process a batch of documents, the Cosmos DB trigger can bind to the following types:

[\[+\] Expand table](#)

Type	Description
<code>IEnumerable<T></code> where <code>T</code> is a JSON serializable type	An enumeration of entities included in the batch. Each entry represents one document from the Cosmos DB change feed.

Connections

The `connectionStringSetting/connection` and `leaseConnectionStringSetting/leaseConnection` properties are references to environment configuration which specifies how the app should connect to Azure Cosmos DB. They may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#). This option is only available for the `connection` and `leaseConnection` versions from [version 4.x or higher of the extension](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

The connection string for your database account should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

Identity-based connections

If you are using [version 4.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To do this, you would define settings under a common prefix which maps to the `connection` property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

[\[+\] Expand table](#)

Property	Environment variable template	Description	Example value
Account Endpoint	<code><CONNECTION_NAME_PREFIX>__accountEndpoint</code>	The Azure Cosmos DB account endpoint URI.	<code>https://<database_account_name>.documents.azure.com:443/</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

Cosmos DB does not use Azure RBAC for data operations. Instead, it uses a [Cosmos DB built-in RBAC system](#) which is built on similar concepts. You will need to create a role assignment that provides access to your database account at runtime. Azure RBAC Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Azure Cosmos DB extension in normal operation. Your application may require additional permissions based on the code you write.

 Expand table

Binding type	Example built-in roles ¹
Trigger ²	Cosmos DB Built-in Data Contributor
Input binding	Cosmos DB Built-in Data Reader
Output binding	Cosmos DB Built-in Data Contributor

¹ These roles cannot be used in an Azure RBAC role assignment. See the [Cosmos DB built-in RBAC system](#) documentation for details on how to assign these roles.

² When using identity, Cosmos DB treats container creation as a management operation. It is not available as a data-plane operation for the trigger. You will need to ensure that you create the containers needed by the trigger (including the lease container) before setting up your function.

Next steps

- [Read an Azure Cosmos DB document \(Input binding\)](#)
- [Save changes to an Azure Cosmos DB document \(Output binding\)](#)

Azure Cosmos DB input binding for Azure Functions 2.x and higher

Article • 07/31/2023

The Azure Cosmos DB input binding uses the SQL API to retrieve one or more Azure Cosmos DB documents and passes them to the input parameter of the function. The document ID or query parameters can be determined based on the trigger that invokes the function.

For information on setup and configuration details, see the [overview](#).

ⓘ Note

When the collection is partitioned, lookup operations must also specify the partition key value.

Example

Unless otherwise noted, examples in this article target version 3.x of the [Azure Cosmos DB extension](#). For use with extension version 4.x, you need to replace the string `collection` in property and attribute names with `container`.

A C# function can be created by using one of the following C# modes:

- [In-process class library](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.
- [Isolated worker process class library](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.

In-process

This section contains the following examples for using [in-process C# class library functions](#) with extension version 3.x:

- [Queue trigger, look up ID from JSON](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [HTTP trigger, look up ID from route data, using SqlQuery](#)
- [HTTP trigger, get multiple docs, using SqlQuery](#)
- [HTTP trigger, get multiple docs, using DocumentClient](#)
- [HTTP trigger, get multiple docs, using CosmosClient \(v4 extension\)](#)

The examples refer to a simple `ToDoItem` type:

```
C#  
  
namespace CosmosDBSamplesV2  
{  
    public class ToDoItem  
    {  
        [JsonProperty("id")]  
        public string Id { get; set; }  
    }  
}
```

```

    [JsonProperty("partitionKey")]
    public string PartitionKey { get; set; }

    public string Description { get; set; }
}

```

Queue trigger, look up ID from JSON

The following example shows a **C# function** that retrieves a single document. The function is triggered by a queue message that contains a JSON object. The queue trigger parses the JSON into an object of type `ToDoItemLookup`, which contains the ID and partition key value to look up. That ID and partition key value are used to retrieve a `ToDoItem` document from the specified database and collection.

```

C#

namespace CosmosDBSamplesV2
{
    public class ToDoItemLookup
    {
        public string ToDoItemId { get; set; }

        public string ToDoItemPartitionKeyValue { get; set; }
    }
}

```

```

C#

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class DocByIdFromJSON
    {
        [FunctionName("DocByIdFromJSON")]
        public static void Run(
            [QueueTrigger("todoqueueforlookup")] ToDoItemLookup ToDoItemLookup,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{ToDoItemId}",
                PartitionKey = "{ToDoItemPartitionKeyValue}")] ToDoItem ToDoItem,
            ILogger log)
        {
            log.LogInformation($"C# Queue trigger function processed Id={ToDoItemLookup?.ToDoItemId} Key={ToDoItemLookup?.ToDoItemPartitionKeyValue}");

            if (ToDoItem == null)
            {
                log.LogInformation($"ToDo item not found");
            }
            else
            {
                log.LogInformation($"Found ToDo item, Description={ToDoItem.Description}");
            }
        }
    }
}

```

HTTP trigger, look up ID from query string

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID and partition key value to look up. That ID and partition key value are used to retrieve a `ToDoItem` document from the specified database and collection.

ⓘ Note

The HTTP query string parameter is case-sensitive.

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class DocByIdFromQueryString
    {
        [FunctionName("DocByIdFromQueryString")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
            HttpRequest req,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{Query.id}",
                PartitionKey = "{Query.partitionKey}")] ToDoItem ToDoItem,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");

            if (ToDoItem == null)
            {
                log.LogInformation($"ToDo item not found");
            }
            else
            {
                log.LogInformation($"Found ToDo item, Description={ToDoItem.Description}");
            }
            return new OkResult();
        }
    }
}
```

HTTP trigger, look up ID from route data

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID and partition key value to look up. That ID and partition key value are used to retrieve a `ToDoItem` document from the specified database and collection.

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
```

```

using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class DocByIdFromRouteData
    {
        [FunctionName("DocByIdFromRouteData")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
                Route = "todoitems/{partitionKey}/{id}")]HttpRequest req,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{id}",
                PartitionKey = "{partitionKey}")] ToDoItem ToDoItem,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");

            if (ToDoItem == null)
            {
                log.LogInformation($"ToDo item not found");
            }
            else
            {
                log.LogInformation($"Found ToDo item, Description={ToDoItem.Description}");
            }
            return new OkResult();
        }
    }
}

```

HTTP trigger, look up ID from route data, using SqlQuery

The following example shows a [C# function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

The example shows how to use a binding expression in the `SqlQuery` parameter. You can pass route data to the `SqlQuery` parameter as shown, but currently [you can't pass query string values](#).

Note

If you need to query by just the ID, it is recommended to use a look up, like the [previous examples](#), as it will consume less `request units`. Point read operations (GET) are [more efficient](#) than queries by ID.

C#

```

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class DocByIdFromRouteDataUsingSqlQuery

```

```

    {
        [FunctionName("DocByIdFromRouteDataUsingSqlQuery")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
                Route = "todoitems2/{id}")]HttpRequest req,
            [CosmosDB("ToDoItems", "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                SqlQuery = "select * from ToDoItems r where r.id = {id}")]
                IEnumerable<ToDoItem> ToDoItems,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");

            foreach (ToDoItem ToDoItem in ToDoItems)
            {
                log.LogInformation(ToDoItem.Description);
            }
            return new OkResult();
        }
    }
}

```

HTTP trigger, get multiple docs, using SqlCommand

The following example shows a [C# function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

```

C#

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class DocsBySqlQuery
    {
        [FunctionName("DocsBySqlQuery")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
                HttpRequest req,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                SqlQuery = "SELECT top 2 * FROM c order by c._ts desc")]
                IEnumerable<ToDoItem> ToDoItems,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");
            foreach (ToDoItem ToDoItem in ToDoItems)
            {
                log.LogInformation(ToDoItem.Description);
            }
            return new OkResult();
        }
    }
}

```

HTTP trigger, get multiple docs, using DocumentClient

The following example shows a [C# function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

! Note

You can also use the `IDocumentClient` interface to make testing easier.

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace CosmosDBSamplesV2
{
    public static class DocsByUsingDocumentClient
    {
        [FunctionName("DocsByUsingDocumentClient")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
                Route = null)]HttpRequest req,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")] DocumentClient client,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");

            var searchterm = req.Query["searchterm"];
            if (string.IsNullOrWhiteSpace(searchterm))
            {
                return (ActionResult)new NotFoundResult();
            }

            Uri collectionUri = UriFactory.CreateDocumentCollectionUri("ToDoItems", "Items");

            log.LogInformation($"Searching for: {searchterm}");

            IDocumentQuery<ToDoItem> query = client.CreateDocumentQuery<ToDoItem>(collectionUri)
                .Where(p => p.Description.Contains(searchterm))
                .AsDocumentQuery();

            while (query.HasMoreResults)
            {
                foreach (ToDoItem result in await query.ExecuteNextAsync())
                {
                    log.LogInformation(result.Description);
                }
            }
            return new OkResult();
        }
    }
}
```

```
    }  
}
```

HTTP trigger, get multiple docs, using CosmosClient

The following example shows a [C# function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `CosmosClient` instance provided by the Azure Cosmos DB binding, available in [extension version 4.x](#), to read a list of documents. The `CosmosClient` instance could also be used for write operations.

C#

```
using System.Linq;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Http;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.Cosmos;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Host;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.Extensions.Logging;  
  
namespace CosmosDBSamplesV2  
{  
    public static class DocsByUsingCosmosClient  
    {  
        [FunctionName("DocsByUsingCosmosClient")]  
        public static async Task<IActionResult> Run(  
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",  
                Route = null)]HttpRequest req,  
            [CosmosDB(  
                databaseName: "ToDoItems",  
                containerName: "Items",  
                Connection = "CosmosDBConnection")] CosmosClient client,  
            ILogger log)  
        {  
            log.LogInformation("C# HTTP trigger function processed a request.");  
  
            var searchterm = req.Query["searchterm"].ToString();  
            if (string.IsNullOrWhiteSpace(searchterm))  
            {  
                return (ActionResult)new NotFoundResult();  
            }  
  
            Container container = client.GetDatabase("ToDoItems").GetContainer("Items");  
  
            log.LogInformation($"Searching for: {searchterm}");  
  
            QueryDefinition queryDefinition = new QueryDefinition(  
                "SELECT * FROM items i WHERE CONTAINS(i.Description, @searchterm)"  
                .WithParameter("@searchterm", searchterm));  
            using (FeedIterator<ToDoItem> resultSet = container.GetItemQueryIterator<ToDoItem>(queryDefinition))  
            {  
                while (resultSet.HasMoreResults)  
                {  
                    FeedResponse<ToDoItem> response = await resultSet.ReadNextAsync();  
                    ToDoItem item = response.First();  
                    log.LogInformation(item.Description);  
                }  
            }  
  
            return new OkResult();  
        }  
}
```

```
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Extension 4.x+

Attribute property	Description
Connection	The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account being queried. For more information, see Connections .
DatabaseName	The name of the Azure Cosmos DB database with the container being monitored.
ContainerName	The name of the container being monitored.
PartitionKey	Specifies the partition key value for the lookup. May include binding parameters. It is required for lookups in partitioned containers.
Id	The ID of the document to retrieve. This property supports binding expressions . Don't set both the <code>Id</code> and <code>SqlQuery</code> properties. If you don't set either one, the entire container is retrieved.
SqlQuery	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: <code>SELECT * FROM c WHERE c.departmentId = {departmentId}</code> . Don't set both the <code>Id</code> and <code>SqlQuery</code> properties. If you don't set either one, the entire container is retrieved.
PreferredLocations	(Optional) Defines preferred locations (regions) for geo-replicated database accounts in the Azure Cosmos DB service. Values should be comma-separated. For example, <code>East US, South Central US, North Europe</code> .

See the [Example section](#) for complete examples.

Usage

Extension 4.x+

When the function exits successfully, any changes made to the input document are automatically persisted.

The parameter type supported by the Cosmos DB input binding depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension 4.x+

See [Binding types](#) for a list of supported types.

Connections

The `connectionStringSetting/connection` and `leaseConnectionStringSetting/leaseConnection` properties are references to environment configuration which specifies how the app should connect to Azure Cosmos DB. They may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#). This option is only available for the `connection` and `leaseConnection` versions from [version 4.x or higher of the extension](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

The connection string for your database account should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

Identity-based connections

If you are using [version 4.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Azure Active Directory identity](#). To do this, you would define settings under a common prefix which maps to the `connection` property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Account Endpoint	<code><CONNECTION_NAME_PREFIX>__accountEndpoint</code>	The Azure Cosmos DB account endpoint URI.	<code>https://<database_account_name>.documents.azure.com:443/</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It

would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

Cosmos DB does not use Azure RBAC for data operations. Instead, it uses a [Cosmos DB built-in RBAC system](#) which is built on similar concepts. You will need to create a role assignment that provides access to your database account at runtime. Azure RBAC Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Azure Cosmos DB extension in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles ¹
Trigger ²	Cosmos DB Built-in Data Contributor
Input binding	Cosmos DB Built-in Data Reader
Output binding	Cosmos DB Built-in Data Contributor

¹ These roles cannot be used in an Azure RBAC role assignment. See the [Cosmos DB built-in RBAC system](#) documentation for details on how to assign these roles.

² When using identity, Cosmos DB treats container creation as a management operation. It is not available as a data-plane operation for the trigger. You will need to ensure that you create the containers needed by the trigger (including the lease container) before setting up your function.

Next steps

- Run a function when an Azure Cosmos DB document is created or modified (Trigger)
- Save changes to an Azure Cosmos DB document (Output binding)

Azure Cosmos DB output binding for Azure Functions 2.x and higher

Article • 10/05/2023

The Azure Cosmos DB output binding lets you write a new document to an Azure Cosmos DB database using the SQL API.

For information on setup and configuration details, see the [overview](#).

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Example

Unless otherwise noted, examples in this article target version 3.x of the [Azure Cosmos DB extension](#). For use with extension version 4.x, you need to replace the string `collection` in property and attribute names with `container`.

Isolated worker model

The following code defines a `MyDocument` type:

```
C#  
  
public class MyDocument  
{  
    public string Id { get; set; }  
  
    public string Text { get; set; }  
  
    public int Number { get; set; }  
  
    public bool Boolean { get; set; }  
}
```

In the following example, the return type is an `IReadOnlyList<T>`, which is a modified list of documents from trigger binding parameter:

```
C#  
  
using System.Collections.Generic;  
using System.Linq;  
using Microsoft.Azure.Functions.Worker;  
using Microsoft.Extensions.Logging;  
  
namespace SampleApp  
{  
    public class CosmosDBFunction  
    {  
        private readonly ILogger<CosmosDBFunction> _logger;
```

```

public CosmosDBFunction(ILogger<CosmosDBFunction> logger)
{
    _logger = logger;
}

//<docsnippet_exponential_backoff_retry_example>
[Function(nameof(CosmosDBFunction))]
[ExponentialBackoffRetry(5, "00:00:04", "00:15:00")]
[CosmosDBOutput("%CosmosDb%", "%CosmosContainerOut%", Connection = "CosmosDBConnection",
CreateIfNotExists = true)]
public object Run(
    [CosmosDBTrigger(
        "%CosmosDb%",
        "%CosmosContainerIn%",
        Connection = "CosmosDBConnection",
        LeaseContainerName = "leases",
        CreateLeaseContainerIfNotExists = true)] IReadOnlyList<MyDocument> input,
    FunctionContext context)
{
    if (input != null && input.Any())
    {
        foreach (var doc in input)
        {
            _logger.LogInformation("Doc Id: {id}", doc.Id);
        }

        // Cosmos Output
        return input.Select(p => new { id = p.Id });
    }

    return null;
}
//</docsnippet_exponential_backoff_retry_example>
}

```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Extension 4.x+

Attribute property	Description
Connection	The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account being monitored. For more information, see Connections .
DatabaseName	The name of the Azure Cosmos DB database with the container being monitored.
ContainerName	The name of the container being monitored.
CreateIfNotExists	A boolean value to indicate whether the container is created when it doesn't exist. The default is <i>false</i> because new containers are created with reserved throughput, which has cost implications. For more information, see the pricing page .
PartitionKey	When <code>CreateIfNotExists</code> is true, it defines the partition key path for the created container. May include binding parameters.
ContainerThroughput	When <code>CreateIfNotExists</code> is true, it defines the <code>throughput</code> of the created container.

Attribute property	Description
PreferredLocations	(Optional) Defines preferred locations (regions) for geo-replicated database accounts in the Azure Cosmos DB service. Values should be comma-separated. For example, <code>East US, South Central US, North Europe</code> .

See the [Example section](#) for complete examples.

Usage

By default, when you write to the output parameter in your function, a document is created in your database. You should specify the document ID of the output document by specifying the `id` property in the JSON object passed to the output parameter.

ⓘ Note

When you specify the ID of an existing document, it gets overwritten by the new output document.

The parameter type supported by the Cosmos DB output binding depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension 4.x+

When you want the function to write to a single document, the Cosmos DB output binding can bind to the following types:

Type	Description
JSON serializable types	An object representing the JSON content of a document. Functions attempts to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write to multiple documents, the Cosmos DB output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is JSON serializable type	An array containing multiple documents. Each entry represents one document.

For other output scenarios, create and use types from [Microsoft.Azure.Cosmos](#) directly.

Connections

The `connectionStringSetting/connection` and `leaseConnectionStringSetting/leaseConnection` properties are references to environment configuration which specifies how the app should connect to Azure Cosmos DB. They may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#). This option is only available for the `connection` and `leaseConnection` versions from [version 4.x or higher of the extension](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

The connection string for your database account should be stored in an application setting with a name matching the value specified by the connection property of the binding configuration.

Identity-based connections

If you are using [version 4.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Azure Active Directory identity](#). To do this, you would define settings under a common prefix which maps to the connection property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Account Endpoint	<CONNECTION_NAME_PREFIX>__accountEndpoint	The Azure Cosmos DB account endpoint URI.	https://<database_account_name>.documents.azure.com:443/

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the [principle of least privilege](#), granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

Cosmos DB does not use Azure RBAC for data operations. Instead, it uses a [Cosmos DB built-in RBAC system](#) which is built on similar concepts. You will need to create a role assignment that provides access to your database account at runtime. Azure RBAC Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Azure Cosmos DB extension in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles ¹
Trigger ²	Cosmos DB Built-in Data Contributor
Input binding	Cosmos DB Built-in Data Reader
Output binding	Cosmos DB Built-in Data Contributor

¹ These roles cannot be used in an Azure RBAC role assignment. See the [Cosmos DB built-in RBAC system documentation](#) for details on how to assign these roles.

² When using identity, Cosmos DB treats container creation as a management operation. It is not available as a data-plane operation for the trigger. You will need to ensure that you create the containers needed by the trigger (including the lease container) before setting up your function.

Exceptions and return codes

Binding	Reference
Azure Cosmos DB	HTTP status codes for Azure Cosmos DB

Next steps

- Run a function when an Azure Cosmos DB document is created or modified (Trigger)
- Read an Azure Cosmos DB document (Input binding)

Migrate function apps from Azure Cosmos DB extension version 3.x to version 4.x

Article • 07/10/2024

This article highlights considerations for upgrading your existing Azure Functions applications that use the Azure Cosmos DB extension version 3.x to use the newer [extension version 4.x](#). Migrating from version 3.x to version 4.x of the Azure Cosmos DB extension has breaking changes for your application.

Important

On August 31, 2024 the Azure Cosmos DB extension version 3.x will be retired. The extension and all applications using the extension will continue to function, but Azure Cosmos DB will cease to provide further maintenance and support for this extension. We recommend migrating to the latest version 4.x of the extension.

This article walks you through the process of migrating your function app to run on version 4.x of the Azure Cosmos DB extension. Because project upgrade instructions are language dependent, make sure to choose your development language from the selector at the [top of the article](#).

Changes to item ID generation

Item ID is no longer auto populated in the Extension. Therefore, the Item ID must specifically include a generated ID for cases where you were using the Output Binding to create items. To maintain the same behavior as the previous version, you can assign a random GUID as ID property.

Update the extension version

.NET Functions use bindings that are installed in the project as NuGet packages. Depending on your Functions process model, the NuGet package to update varies.

 Expand table

Functions process model	Azure Cosmos DB extension	Recommended version
In-process model	Microsoft.Azure.WebJobs.Extensions.CosmosDB 🔗	>= 4.3.0
Isolated worker model	Microsoft.Azure.Functions.Worker.Extensions.CosmosDB 🔗	>= 4.4.1

Update your `.csproj` project file to use the latest extension version for your process model. The following `.csproj` file uses version 4 of the Azure Cosmos DB extension.

Isolated worker model

```
XML

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
    <OutputType>Exe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Azure.Functions.Worker"
      Version="1.21.0" />
    <PackageReference
      Include="Microsoft.Azure.Functions.Worker.Extensions.CosmosDB"
      Version="4.6.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk"
      Version="1.16.4" />
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
</Project>
```

Rename the binding attributes

Both [in-process](#) and [isolated process](#) C# libraries use the [CosmosDBTriggerAttribute](#) [🔗](#) to define the function.

The following table only includes attributes that were renamed or were removed from the version 3 extension. For a full list of attributes available in the version 4 extension, visit the [attribute reference](#).

[] Expand table

Version 3 attribute property	Version 4 attribute property	Version 4 attribute description
ConnectionStringSetting	Connection	The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account being monitored. For more information, see Connections .
CollectionName	ContainerName	The name of the container being monitored.
LeaseConnectionStringSetting	LeaseConnection	(Optional) The name of an app setting or setting collection that specifies how to connect to the Azure Cosmos DB account that holds the lease container.
		When not set, the <code>Connection</code> value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases container must have write permissions.
LeaseCollectionName	LeaseContainerName	(Optional) The name of the container used to store leases. When not set, the value <code>leases</code> is used.
CreateLeaseCollectionIfNotExists	CreateLeaseContainerIfNotExists	(Optional) When set to <code>true</code> , the leases container is automatically created when it doesn't already exist. The default value is <code>false</code> . When using Microsoft Entra identities if you set the value to <code>true</code> , creating containers isn't an allowed operation and your Function won't be able to start.
LeasesCollectionThroughput	LeasesContainerThroughput	(Optional) Defines the number of Request Units to assign when the

Version 3 attribute property	Version 4 attribute property	Version 4 attribute description
		leases container is created. This setting is only used when <code>CreateLeaseContainerIfNotExists</code> is set to <code>true</code> . This parameter is automatically set when the binding is created using the portal.
<code>LeaseCollectionPrefix</code>	<code>LeaseContainerPrefix</code>	(Optional) When set, the value is added as a prefix to the leases created in the Lease container for this function. Using a prefix allows two separate Azure Functions to share the same Lease container by using different prefixes.
<code>UseMultipleWriteLocations</code>	<i>Removed</i>	This attribute is no longer needed as it's automatically detected.
<code>UseDefaultJsonSerialization</code>	<i>Removed</i>	This attribute is no longer needed as you can fully customize the serialization using built in support in the Azure Cosmos DB version 3 .NET SDK .
<code>CheckpointInterval</code>	<i>Removed</i>	This attribute has been removed in the version 4 extension.
<code>CheckpointDocumentCount</code>	<i>Removed</i>	This attribute has been removed in the version 4 extension.

Modify your function code

The Azure Functions extension version 4 is built on top of the Azure Cosmos DB .NET SDK version 3, which removed support for the [Document class](#). Instead of receiving a list of `Document` objects with each function invocation, which you must then deserialize into your own object type, you can now directly receive a list of objects of your own type.

This example refers to a simple `ToDoItem` type.

```
cs
namespace CosmosDBSamples
{
```

```
// Customize the model with your own desired properties
public class ToDoItem
{
    public string id { get; set; }
    public string Description { get; set; }
}
}
```

Changes to the attribute names must be made directly in the code when defining your Function.

cs

```
using System.Collections.Generic;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamples
{
    public static class CosmosTrigger
    {
        [FunctionName("CosmosTrigger")]
        public static void Run([CosmosDBTrigger(
            databaseName: "databaseName",
            containerName: "containerName",
            Connection = "CosmosDBConnectionString",
            LeaseContainerName = "leases",
            CreateLeaseContainerIfNotExists = true)] IReadOnlyList<ToDoItem>
input, ILogger log)
        {
            if (input != null && input.Count > 0)
            {
                log.LogInformation("Documents modified " + input.Count);
                log.LogInformation("First document Id " + input[0].id);
            }
        }
    }
}
```

ⓘ Note

If your scenario relied on the dynamic nature of the `Document` type to identify different schemas and types of events, you can use a base abstract type with the common properties across your types or dynamic types like `IObject` (when using `Microsoft.Azure.WebJobs.Extensions.CosmosDB`) and `JsonNode` (when using `Microsoft.Azure.Functions.Worker.Extensions.CosmosDB`) that allow to access properties like `Document` did.

Additionally, if you are using the Output Binding, please review the [change in item ID generation](#) to verify if you need additional code changes.

Next steps

- Run a function when an Azure Cosmos DB document is created or modified (Trigger)
 - Read an Azure Cosmos DB document (Input binding)
 - Save changes to an Azure Cosmos DB document (Output binding)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Azure Data Explorer bindings for Azure Functions overview (preview)

Article • 03/31/2024

This set of articles explains how to work with [Azure Data Explorer](#) bindings in Azure Functions. Azure Functions supports input bindings and output bindings for Azure Data Explorer clusters.

[+] [Expand table](#)

Action	Type
Read data from a database	Input binding
Ingest data to a database	Output binding

Install the extension

The extension NuGet package you install depends on the C# mode you're using in your function app.

Isolated worker model

Functions run in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing [this NuGet package](#).

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Kusto --prerelease
```

Kusto connection string

Azure Data Explorer bindings for Azure Functions have a required property for the connection string on all bindings. The connection string is documented at [Kusto connection strings](#).

Considerations

- Azure Data Explorer binding supports version 4.x and later of the Functions runtime.
- Source code for the Azure Data Explorer bindings is in [this GitHub repository](#).
- This binding requires connectivity to Azure Data Explorer. For input bindings, users require **Viewer** permissions. For output bindings, users require **Ingestor** permissions. For more information about permissions, see [Role-based access control](#).

Next steps

- [Read data from a database \(input binding\)](#)
 - [Save data to a database \(output binding\)](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Azure Data Explorer input bindings for Azure Functions (preview)

Article • 06/30/2023

The Azure Data Explorer input binding retrieves data from a database.

Examples

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime.
- **C# script:** Used primarily when you create C# functions in the Azure portal.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

More samples for the Azure Data Explorer input binding (out of process) are available in the [GitHub repository](#).

This section contains the following examples:

- [HTTP trigger, get row by ID from query string](#)
- [HTTP trigger, get multiple rows from route data](#)

The examples refer to a `Product` class and the Products table, both of which are defined in the previous sections.

HTTP trigger, get row by ID from query string

The following example shows a [C# function](#) that retrieves a single record. The function is triggered by an HTTP request that uses a query string to specify the ID. That ID is used to retrieve a `Product` record with the specified query.

ⓘ Note

The HTTP query string parameter is case sensitive.

cs

```
using System.Text.Json.Nodes;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Kusto;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples.Common;

namespace Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.InputBindingSamples
{
    public static class GetProductsQuery
    {
        [Function("GetProductsQuery")]
        public static JsonArray Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "getproductsquery")] HttpRequestData req,
            [KustoInput(Database: "productsdb",
            KqlCommand = "declare query_parameters (productId:long);Products | where ProductID == productId",
            KqlParameters = "@productId={Query.productId}", Connection = "KustoConnectionString")] JsonArray products)
        {
            return products;
        }
    }
}
```

HTTP trigger, get multiple rows from route parameter

The following example shows a [C# function](#) that retrieves records returned by the query (based on the name of the product, in this case). The function is triggered by an HTTP request that uses route data to specify the value of a query parameter. That parameter is used to filter the `Product` records in the specified query.

```
cs

using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Kusto;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples.Common;

namespace Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.InputBindingSamples
{
    public static class GetProductsFunction
    {
        [Function("GetProductsFunction")]
        public static IEnumerable<Product> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "getproductsfn/{name}")] HttpRequestData req,
            [KustoInput(Database: "productsdb",
            KqlCommand = "declare query_parameters (name:string);GetProductsByName(name)",
            KqlParameters = "@name={name}", Connection = "KustoConnectionString")] IEnumerable<Product> products)
        {
            return products;
        }
    }
}
```

Attributes

The [C# library](#) uses the [KustoAttribute](#) attribute to declare the Azure Data Explorer bindings on the function, which has the following properties.

[Expand table](#)

Attribute property	Description
Database	Required. The database against which the query must be executed.
Connection	Required. The name of the variable that holds the connection string, resolved through environment variables or through function app settings. Defaults to look up on the variable <code>KustoConnectionString</code> . At runtime, this variable is looked up against the environment. Documentation on the connection string is at Kusto connection strings . For example: "KustoConnectionString": "Data Source=https://your_cluster.kusto.windows.net;Database=your_Database;Fed=True;AppClientId=your_AppId;AppKey=your_AppKey;Authority Id=your_TenantId".
KqlCommand	Required. The <code>KqlQuery</code> parameter that must be executed. Can be a KQL query or a KQL function call.
KqlParameters	Optional. Parameters that act as predicate variables for <code>KqlCommand</code> . For example, "@name={name},@Id={id}", where <code>{name}</code> and <code>{id}</code> are substituted at runtime with actual values acting as predicates. The parameter name and the parameter value can't contain a comma (,) or an equal sign (=).
ManagedServiceIdentity	Optional. You can use a managed identity to connect to Azure Data Explorer. To use a system managed identity, use "system." Any other identity names are interpreted as a user managed identity.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

Usage

The attribute's constructor takes the database and the attributes `KqlCommand` and `KqlParameters` and the connection setting name. The KQL command can be a KQL statement or a KQL function. The connection string setting name corresponds to the application setting (in `local.settings.json` for local development) that contains the [Kusto connection strings](#). For example: "KustoConnectionString": "Data Source=https://your_cluster.kusto.windows.net;Database=your_Database;Fed=True;AppClientId=your_AppId;AppKey=your_AppKey;Authority Id=your_TenantId". Queries executed by the input binding are parameterized. The values provided in the KQL parameters are used at runtime.

Next steps

[Save data to a table \(output binding\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) 

Azure Data Explorer output bindings for Azure Functions (preview)

Article • 06/30/2023

When a function runs, the Azure Data Explorer output binding ingests data to Azure Data Explorer.

For information on setup and configuration details, see the [overview](#).

Examples

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime.
- **C# script:** Used primarily when you create C# functions in the Azure portal.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

More samples for the Azure Data Explorer output binding are available in the [GitHub repository](#).

This section contains the following examples:

- [HTTP trigger, write one record](#)
- [HTTP trigger, write records with mapping](#)

The examples refer to `Product` class and a corresponding database table:

```
cs

public class Product
{
    [JsonProperty(nameof(ProductID))]
    public long ProductID { get; set; }

    [JsonProperty(nameof(Name))]
    public string Name { get; set; }

    [JsonProperty(nameof(Cost))]
    public double Cost { get; set; }
}
```

Kusto

```
.create-merge table Products (ProductID:long, Name:string, Cost:double)
```

HTTP trigger, write one record

The following example shows a [C# function](#) that adds a record to a database. The function uses data provided in an HTTP POST request as a JSON body.

```
cs

using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Kusto;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples.Common;

namespace Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples
```

```

{
    public static class AddProduct
    {
        [Function("AddProduct")]
        [KustoOutput(Database: "productsdb", Connection = "KustoConnectionString", TableName = "Products")]
        public static async Task<Product> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = "addproductuni")]
            HttpRequestData req)
        {
            Product? prod = await req.ReadFromJsonAsync<Product>();
            return prod ?? new Product { };
        }
    }
}

```

HTTP trigger, write records with mapping

The following example shows a [C# function](#) that adds a collection of records to a database. The function uses mapping that transforms a `Product` to `Item`.

To transform data from `Product` to `Item`, the function uses a mapping reference:

Kusto

```

.create-merge table Item (ItemID:long, ItemName:string, ItemCost:float)

-- Create a mapping that transforms an Item to a Product

.create-or-alter table Product ingestion json mapping "item_to_product_json"
'[{"column":"ProductID","path":"$.ItemID"}, {"column":"Name","path":"$.ItemName"}, {"column":"Cost","path":"$.ItemCost"}]'

```

cs

```

namespace Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples.Common
{
    public class Item
    {
        public long ItemID { get; set; }

        public string? ItemName { get; set; }

        public double ItemCost { get; set; }
    }
}

```

cs

```

using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Kusto;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples.Common;

namespace Microsoft.Azure.WebJobs.Extensions.Kusto.SamplesOutOfProc.OutputBindingSamples
{
    public static class AddProductsWithMapping
    {
        [Function("AddProductsWithMapping")]
        [KustoOutput(Database: "productsdb", Connection = "KustoConnectionString", TableName = "Products", MappingRef = "item_to_product_json")]
        public static async Task<Item> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = "addproductswithmapping")]
            HttpRequestData req)
        {
            Item? item = await req.ReadFromJsonAsync<Item>();
            return item ?? new Item { };
        }
    }
}

```

Attributes

The [C# library](#) uses the [KustoAttribute](#) attribute to declare the Azure Data Explorer bindings on the function, which has the following properties.

[Expand table](#)

Attribute property	Description
Database	Required. The database against which the query must be executed.
Connection	Required. The name of the variable that holds the connection string, which is resolved through environment variables or through function app settings. Defaults to look up on the variable <code>KustoConnectionString</code> . At runtime, this variable is looked up against the environment. Documentation on the connection string is at Kusto connection strings . For example: "KustoConnectionString": "Data Source=https://your_cluster.kusto.windows.net;Database=your_Database;Fed=True;AppClientId=your_AppId;AppKey=your_AppKey;AuthorityId=your_TenantId".
TableName	Required. The table to ingest the data into.
MappingRef	Optional. Attribute to pass a mapping ref that's already defined in the cluster.
ManagedServiceIdentity	Optional. A managed identity can be used to connect to Azure Data Explorer. To use a system managed identity, use "system." Any other identity names are interpreted as a user managed identity.
DataFormat	Optional. The default data format is <code>multijson/json</code> . It can be set to <code>text</code> formats supported in the <code>datasource</code> format enumeration . Samples are validated and provided for CSV and JSON formats.

When you're developing locally, add your application settings in the [local.settings.json](#) file in the `values` collection.

Usage

The attribute's constructor takes the database and the attributes `TableName`, `MappingRef`, and `DataFormat` and the connection setting name. The KQL command can be a KQL statement or a KQL function. The connection string setting name corresponds to the application setting (in `local.settings.json` for local development) that contains the [Kusto connection strings](#). For example: "KustoConnectionString": "Data Source=https://your_cluster.kusto.windows.net;Database=your_Database;Fed=True;AppClientId=your_AppId;AppKey=your_AppKey;AuthorityId=your_TenantId". Queries executed by the input binding are parameterized. The values provided in the KQL parameters are used at runtime.

Next steps

[Read data from a table \(input binding\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Azure OpenAI extension for Azure Functions

Article • 07/12/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI extension for Azure Functions implements a set of triggers and bindings that enable you to easily integrate features and behaviors of [Azure OpenAI Service](#) into your function code executions.

Azure Functions is an event-driven compute service that provides a set of [triggers and bindings](#) to easily connect with other Azure services.

With the integration between Azure OpenAI and Functions, you can build functions that can:

[+] [Expand table](#)

Action	Trigger/binding type
Use a standard text prompt for content completion	Azure OpenAI text completion input binding
Respond to an assistant request to call a function	Azure OpenAI assistant trigger
Create an assistant	Azure OpenAI assistant create output binding
Message an assistant	Azure OpenAI assistant post input binding
Get assistant history	Azure OpenAI assistant query input binding
Read text embeddings	Azure OpenAI embeddings input binding
Write to a vector database	Azure OpenAI embeddings store output binding
Read from a vector database	Azure OpenAI semantic search input binding

Install extension

The extension NuGet package you install depends on the C# mode [in-process](#) or [isolated worker process](#) you're using in your function app:

Isolated process

Add the Azure OpenAI extension to your project by installing the [Microsoft.Azure.Functions.Worker.Extensions.OpenAI](#) NuGet package, which you can do using the .NET CLI:

dotnet

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.OpenAI -  
-prerelease
```

When using a vector database for storing content, you should also install at least one of these NuGet packages:

- Azure AI Search:
[Microsoft.Azure.Functions.Worker.Extensions.OpenAI.AzureAISeach](#)
- Azure Cosmos DB for MongoDB:
[Microsoft.Azure.Functions.Worker.Extensions.OpenAI.CosmosDBSearch](#)
- Azure Data Explorer:
[Microsoft.Azure.Functions.Worker.Extensions.OpenAI.Kusto](#)

Application settings

To use the Azure OpenAI binding extension, you need to add one or more of these settings, which are used to connect to your OpenAI resource. During local development, you also need to add these settings to your `local.settings.json` file.

[Expand table](#)

Setting name	Description
<code>AZURE_OPENAI_ENDPOINT</code>	Required. Sets the endpoint of the OpenAI resource used by your bindings.
<code>AZURE_OPENAI_KEY</code>	Sets the key used to access an Azure OpenAI resource.
<code>OPENAI_API_KEY</code>	Sets the key used to access a non-Azure OpenAI resource.
<code>AZURE_CLIENT_ID</code>	Sets a user-assigned managed identity used to access the Azure OpenAI

Setting name	Description
	resource.

For more information, see [Work with application settings](#).

Related content

- [Azure OpenAI extension samples](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Azure OpenAI assistant trigger for Azure Functions

Article • 05/24/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI assistant trigger lets you run your code based on custom chat bot or skill request made to an assistant.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about Azure OpenAI assistants, see [Azure OpenAI Assistants API](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example demonstrates how to create an assistant that adds a new todo task to a database. The trigger has a static description of `Create a new todo task` used by the model. The function itself takes a string, which represents a new task to add. When executed, the function adds the task as a new todo item in a custom item store and returns a response from the store.

C#

```
[Function(nameof(AddTodo))]
public Task AddTodo([AssistantSkillTrigger("Create a new todo task")] string
taskDescription)
{
    if (string.IsNullOrEmpty(taskDescription))
    {
        throw new ArgumentException("Task description cannot be empty");
    }

    this.logger.LogInformation("Adding todo: {task}", taskDescription);
```

```
        string todoId = Guid.NewGuid().ToString()[..6];
        return this.todoManager.AddTodoAsync(new TodoItem(todoId,
taskDescription));
}
```

Attributes

Apply the `AssistantSkillTrigger` attribute to define an assistant trigger, which supports these parameters:

[+] Expand table

Parameter	Description
<code>FunctionDescription</code>	Gets the description of the assistant function, which is provided to the model.
<code>FunctionName</code>	<i>Optional.</i> Gets or sets the name of the function called by the assistant.
<code>ParameterDescriptionJson</code>	<i>Optional.</i> Gets or sets a JSON description of the function parameter, which is provided to the model. For more information, see Usage .
<code>Model</code>	<i>Optional.</i> Gets or sets the OpenAI chat model deployment to use, with a default value of <code>gpt-3.5-turbo</code> .

See the [Example section](#) for complete examples.

Usage

When `parameterDescriptionJson` JSON value isn't provided, it's autogenerated. For more information on the syntax of this object, see the [OpenAI API documentation](#).

Related content

- [Assistant samples](#)
- [Azure OpenAI extension](#)
- [Azure OpenAI assistant query input binding](#)
- [Azure OpenAI assistant create output binding](#)
- [Azure OpenAI assistant post input binding](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure OpenAI assistant query input binding for Azure Functions

Article • 05/21/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI assistant query input binding allows you to integrate Assistants API queries into your code executions.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about Azure OpenAI assistants, see [Azure OpenAI Assistants API](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example demonstrates the creation process, where the HTTP GET function that queries the conversation history of the assistant chat bot. The response to the prompt is returned in the HTTP response.

C#

```
[AssistantQueryInput("{assistantId}", TimestampUtc = "  
{Query.timestampUTC}")]  
AssistantState state  
{  
    return new OkObjectResult(state);  
}  
}
```

Attributes

Apply the `AssistantQuery` attribute to define an assistant query input binding, which supports these parameters:

Parameter	Description
Id	Gets the ID of the assistant to query.
TimeStampUtc	<i>Optional.</i> Gets or sets the timestamp of the earliest message in the chat history to fetch. The timestamp should be in ISO 8601 format - for example, 2023-08-01T00:00:00Z.

Usage

See the [Example section](#) for complete examples.

Related content

- [Assistant samples ↗](#)
- [Azure OpenAI extension](#)
- [Azure OpenAI assistant trigger](#)
- [Azure OpenAI assistant create output binding](#)
- [Azure OpenAI assistant post input binding](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure OpenAI assistant create output binding for Azure Functions

Article • 05/21/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI assistant create output binding allows you to create a new assistant chat bot from your function code execution.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about Azure OpenAI assistants, see [Azure OpenAI Assistants API](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example demonstrates the creation process, where the HTTP PUT function that creates a new assistant chat bot with the specified ID. The response to the prompt is returned in the HTTP response.

C#

```
public static async Task<CreateChatBotOutput> CreateAssistant(
    [HttpTrigger(AuthorizationLevel.Anonymous, "put", Route =
"assistants/{assistantId}")] HttpRequestData req,
    string assistantId)
{
    string instructions =
    """
        Don't make assumptions about what values to plug into functions.
        Ask for clarification if a user request is ambiguous.
    """;

    using StreamReader reader = new(req.Body);

    string request = await reader.ReadToEndAsync();
```

```
        return new CreateChatBotOutput
    {
        HttpResponse = new ObjectResult(new { assistantId }) { StatusCode =
202 },
        ChatBotCreateRequest = new AssistantCreateRequest(assistantId,
instructions)
    {
        ChatStorageConnectionSetting = "AzureWebJobsStorage",
        CollectionName = "SampleChatState",
    },
},
};
```

Attributes

Apply the `CreateAssistant` attribute to define an assistant create output binding, which supports these parameters:

[+] Expand table

Parameter	Description
<code>Id</code>	The identifier of the assistant to create.
<code>Instructions</code>	<i>Optional.</i> The instructions that are provided to assistant to follow.

Usage

See the [Example section](#) for complete examples.

Related content

- [Assistant samples ↗](#)
- [Azure OpenAI extension](#)
- [Azure OpenAI assistant trigger](#)
- [Azure OpenAI assistant query input binding](#)
- [Azure OpenAI assistant post input binding](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Azure OpenAI assistant post input binding for Azure Functions

Article • 05/21/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI assistant post input binding lets you send prompts to assistant chat bots.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about Azure OpenAI assistants, see [Azure OpenAI Assistants API](./ai-services/openai/

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example demonstrates the creation process, where the HTTP POST function that sends user prompts to the assistant chat bot. The response to the prompt is returned in the HTTP response.

C#

```
[Function(nameof(PostUserQuery))]
public static async Task<IActionResult> PostUserQuery(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route =
"assistants/{assistantId}")] HttpRequestData req,
    string assistantId,
    [AssistantPostInput("{assistantId}", "{Query.message}", Model =
"%CHAT_MODEL_DEPLOYMENT_NAME%")] AssistantState state)
{
    return new
OkObjectResult(state.RecentMessages.LastOrDefault()?.Content ?? "No response
returned.");
}

/// <summary>
```

```
    /// HTTP GET function that queries the conversation history of the
    // assistant chat bot.
    /// </summary>
    [Function(nameof(GetChatState))]
    public static async Task<IActionResult> GetChatState(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
"assistants/{assistantId}")] HttpRequestData req,
        string assistantId,
        [AssistantQueryInput("{assistantId}", TimestampUtc =
{Query.timestampUTC})] AssistantState state)
    {
        return new OkObjectResult(state);
    }
}
```

Attributes

Apply the `PostUserQuery` attribute to define an assistant post input binding, which supports these parameters:

[Expand table

Parameter	Description
<code>Id</code>	The ID of the assistant to update.
<code>Model</code>	The name of the OpenAI chat model to use. For Azure OpenAI, this value is the name of the model deployment.

Usage

See the [Example section](#) for complete examples.

Related content

- [Assistant samples ↗](#)
- [Azure OpenAI extension](#)
- [Azure OpenAI assistant trigger](#)
- [Azure OpenAI assistant query input binding](#)
- [Azure OpenAI assistant create output binding](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure OpenAI embeddings input binding for Azure Functions

Article • 05/21/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI embeddings input binding allows you to generate embeddings for inputs. The binding can generate embeddings from files or raw text inputs.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about embeddings in Azure OpenAI Service, see [Understand embeddings in Azure OpenAI Service](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example shows how to generate embeddings for a raw text string.

C#

```
[Function(nameof(GenerateEmbeddings_Http_RequestAsync))]
public async Task GenerateEmbeddings_Http_RequestAsync(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = "embeddings")]
    HttpRequestData req,
    [EmbeddingsInput("{RawText}", InputType.RawText, Model =
    "%EMBEDDING_MODEL_DEPLOYMENT_NAME%")] EmbeddingsContext embeddings)
{
    using StreamReader reader = new(req.Body);
    string request = await reader.ReadToEndAsync();

    EmbeddingsRequest? requestBody =
    JsonSerializer.Deserialize<EmbeddingsRequest>(request);

    this.logger.LogInformation(
        "Received {count} embedding(s) for input text containing {length}
        characters.",
        embeddings.Count,
```

```

    requestBody?.RawText?.Length);

    // TODO: Store the embeddings into a database or other storage.
}

```

This example shows how to retrieve embeddings stored at a specified file that is accessible to the function.

```

C# 

[Function(nameof(GetEmbeddings_Http_FilePath))]
public async Task GetEmbeddings_Http_FilePath(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = "embeddings-
from-file")] HttpRequestData req,
    [EmbeddingsInput("{FilePath}", InputType.FilePath, MaxChunkLength = 512,
Model = "%EMBEDDING_MODEL_DEPLOYMENT_NAME%")] EmbeddingsContext embeddings)
{
    using StreamReader reader = new(req.Body);
    string request = await reader.ReadToEndAsync();

    EmbeddingsRequest? requestBody =
JsonSerializer.Deserialize<EmbeddingsRequest>(request);
    this.logger.LogInformation(
        "Received {count} embedding(s) for input file '{path}'.",
        embeddings.Count,
        requestBody?.FilePath);

    // TODO: Store the embeddings into a database or other storage.
}

```

Attributes

Apply the `EmbeddingsInput` attribute to define an embeddings input binding, which supports these parameters:

[\[\] Expand table](#)

Parameter	Description
Input	The input string for which to generate embeddings.
Model	<i>Optional.</i> The ID of the model to use, which defaults to <code>text-embedding-ada-002</code> . You shouldn't change the model for an existing database. For more information, see Usage .
MaxChunkLength	<i>Optional.</i> The maximum number of characters used for chunking the input. For more information, see Usage .

Parameter	Description
MaxOverlap	<i>Optional.</i> Gets or sets the maximum number of characters to overlap between chunks.
InputType	<i>Optional.</i> Gets the type of the input.

See the [Example section](#) for complete examples.

Usage

Changing the default embeddings `model` changes the way that embeddings are stored in the vector database. Changing the default model can cause the lookups to start misbehaving when they don't match the rest of the data that was previously ingested into the vector database. The default model for embeddings is `text-embedding-ada-002`.

When calculating the maximum character length for input chunks, consider that the maximum input tokens allowed for second-generation input embedding models like `text-embedding-ada-002` is `8191`. A single token is approximately four characters in length (in English), which translates to roughly 32,000 (English) characters of input that can fit into a single chunk.

Related content

- [Embeddings samples](#)
- [Azure OpenAI extensions for Azure Functions](#)

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Azure OpenAI embeddings store output binding for Azure Functions

Article • 05/21/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI embeddings store output binding allows you to write files to a semantic document store that can be referenced later in a semantic search.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about semantic ranking in Azure AI Search, see [Semantic ranking in Azure AI Search](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example writes an HTTP input stream to a semantic document store at the provided URL.

C#

```
[Function("IngestFile")]
public static async Task<EmbeddingsStoreOutputResponse> IngestFile(
    [HttpTrigger(AuthorizationLevel.Function, "post")] HttpRequestData req)
{
    using StreamReader reader = new(req.Body);
    string request = await reader.ReadToEndAsync();

    EmbeddingsRequest? requestBody =
        JsonSerializer.Deserialize<EmbeddingsRequest>(request);

    if (requestBody == null || requestBody.Url == null)
    {
        throw new ArgumentException("Invalid request body. Make sure that
you pass in {\"Url\": value } as the request body.");
    }
}
```

```

Uri uri = new(requestBody.Url);
string filename = Path.GetFileName(uri.AbsolutePath);

IActionResult result = new OkObjectResult(new { status =
HttpStatusCode.OK });

return new EmbeddingsStoreOutputResponse
{
    HttpResponse = result,
    SearchableDocument = new SearchableDocument(filename)
};
}

public class EmbeddingsStoreOutputResponse
{
    [EmbeddingsStoreOutput("{Url}", InputType.Url, "AIEndpoint",
"openai-index", Model = "%EMBEDDING_MODEL_DEPLOYMENT_NAME%")]
    public required SearchableDocument SearchableDocument { get; init; }

    public IActionResult? HttpResponse { get; set; }
}

```

Attributes

Apply the `EmbeddingsStoreOutput` attribute to define an embeddings store output binding, which supports these parameters:

[\[\] Expand table](#)

Parameter	Description
Input	The input string for which to generate embeddings.
Model	<i>Optional.</i> The ID of the model to use, which defaults to <code>text-embedding-ada-002</code> . You shouldn't change the model for an existing database. For more information, see Usage .
MaxChunkLength	<i>Optional.</i> The maximum number of characters used for chunking the input. For more information, see Usage .
MaxOverlap	<i>Optional.</i> Gets or sets the maximum number of characters to overlap between chunks.
InputType	<i>Optional.</i> Gets the type of the input.
ConnectionName	The name of an app setting or environment variable that contains the connection string value. This property supports binding expressions.

Parameter	Description
Collection	The name of the collection or table or index to search. This property supports binding expressions.

Usage

See the [Example section](#) for complete examples.

Related content

- [Semantic search samples ↗](#)
- [Azure OpenAI extensions for Azure Functions](#)
- [Azure OpenAI semantic search input binding for Azure Functions](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure OpenAI Semantic Search Input Binding for Azure Functions

Article • 05/23/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI semantic search input binding allows you to use semantic search on your embeddings.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about semantic ranking in Azure AI Search, see [Semantic ranking in Azure AI Search](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example shows how to perform a semantic search on a file.

C#

```
[Function("PromptFile")]
public static IActionResult PromptFile(
    [HttpTrigger(AuthorizationLevel.Function, "post")]
    SemanticSearchRequest unused,
    [SemanticSearchInput("AIEndpoint", "openai-index", Query = "{Prompt}",
        ChatModel = "%CHAT_MODEL_DEPLOYMENT_NAME%", EmbeddingsModel =
        "%EMBEDDING_MODEL_DEPLOYMENT_NAME%")] SemanticSearchContext result)
{
    return new ContentResult { Content = result.Response, ContentType =
        "text/plain" };
}
```

Attributes

Apply the `SemanticSearchInput` attribute to define a semantic search input binding, which supports these parameters:

[+] Expand table

Parameter	Description
<code>ConnectionName</code>	The name of an app setting or environment variable that contains the connection string value. This property supports binding expressions.
<code>Collection</code>	The name of the collection or table or index to search. This property supports binding expressions.
<code>Query</code>	The semantic query text to use for searching. This property supports binding expressions.
<code>EmbeddingsModel</code>	The ID of the model to use for embeddings. The default value is <code>text-embedding-3-small</code> . This property supports binding expressions.
<code>ChatModel</code>	Gets or sets the name of the Large Language Model to invoke for chat responses. The default value is <code>gpt-3.5-turbo</code> . This property supports binding expressions.
<code>SystemPrompt</code>	<i>Optional.</i> Gets or sets the system prompt to use for prompting the large language model. The system prompt is appended with knowledge that is fetched as a result of the <code>Query</code> . The combined prompt is sent to the OpenAI Chat API. This property supports binding expressions.
<code>MaxKnowledgeCount</code>	<i>Optional.</i> Gets or sets the number of knowledge items to inject into the <code>SystemPrompt</code> .

Usage

See the [Example section](#) for complete examples.

Related content

- [Semantic search samples ↗](#)
- [Azure OpenAI extensions for Azure Functions](#)
- [Azure OpenAI embeddings store output binding for Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Azure OpenAI text completion input binding for Azure Functions

Article • 07/08/2024

ⓘ Important

The Azure OpenAI extension for Azure Functions is currently in preview.

The Azure OpenAI text completion input binding allows you to bring the results text completion APIs into your code executions. You can define the binding to use both predefined prompts with parameters or pass through an entire prompt.

For information on setup and configuration details of the Azure OpenAI extension, see [Azure OpenAI extensions for Azure Functions](#). To learn more about Azure OpenAI completions, see [Learn how to generate or manipulate text](#).

ⓘ Note

While both C# process models are supported, only [isolated worker model](#) examples are provided.

Example

This example demonstrates the *templating* pattern, where the HTTP trigger function takes a `name` parameter and embeds it into a text prompt, which is then sent to the Azure OpenAI completions API by the extension. The response to the prompt is returned in the HTTP response.

C#

```
[Function(nameof(WhoIs))]
public static IActionResult WhoIs(
    [HttpTrigger(AuthorizationLevel.Function, Route = "whois/{name}")]
    HttpRequestData req,
    [TextCompletionInput("Who is {name}?", Model =
"%CHAT_MODEL_DEPLOYMENT_NAME%")] TextCompletionResponse response)
{
    return new OkObjectResult(response.Content);
}
```

```
/// <summary>
```

This example takes a prompt as input, sends it directly to the completions API, and returns the response as the output.

C#

```
[HttpTrigger(AuthorizationLevel.Function, "post")] HttpRequestData
req,
    [TextCompletionInput("{Prompt}", Model =
"%CHAT_MODEL_DEPLOYMENT_NAME%")] TextCompletionResponse response,
    ILogger log)
{
    string text = response.Content;
    return new OkObjectResult(text);
}
```

Attributes

The specific attribute you apply to define a text completion input binding depends on your C# process mode.

Isolated process

In the [isolated worker model](#), apply `TextCompletionInput` to define a text completion input binding.

The attribute supports these parameters:

 [Expand table](#)

Parameter	Description
<code>Prompt</code>	Gets or sets the prompt to generate completions for, encoded as a string.
<code>Model</code>	Gets or sets the ID of the model to use as a string, with a default value of <code>gpt-3.5-turbo</code> .
<code>Temperature</code>	<i>Optional.</i> Gets or sets the sampling temperature to use, as a string between <code>0</code> and <code>2</code> . Higher values (<code>0.8</code>) make the output more random, while lower values like (<code>0.2</code>) make output more focused and deterministic. You should use either <code>Temperature</code> or <code>TopP</code> , but not both.

Parameter	Description
TopP	<i>Optional.</i> Gets or sets an alternative to sampling with temperature, called nucleus sampling, as a string. In this sampling method, the model considers the results of the tokens with <code>top_p</code> probability mass. So <code>0.1</code> means only the tokens comprising the top 10% probability mass are considered. You should use either <code>Temperature</code> or <code>TopP</code> , but not both.
MaxTokens	<i>Optional.</i> Gets or sets the maximum number of tokens to generate in the completion, as a string with a default of <code>100</code> . The token count of your prompt plus <code>max_tokens</code> can't exceed the model's context length. Most models have a context length of 2,048 tokens (except for the newest models, which support 4096).

Usage

See the [Example section](#) for complete examples.

Related content

- [Text completion samples ↗](#)
- [Azure OpenAI extensions for Azure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure SQL bindings for Azure Functions overview

Article • 12/06/2023

This set of articles explains how to work with [Azure SQL](#) bindings in Azure Functions. Azure Functions supports input bindings, output bindings, and a function trigger for the Azure SQL and SQL Server products.

[] [Expand table](#)

Action	Type
Trigger a function when a change is detected on a SQL table	SQL trigger
Read data from a database	Input binding
Save data to a database	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing this [NuGet package](#).

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Sql
```

To use a preview version of the Microsoft.Azure.Functions.Worker.Extensions.Sql package, add the `--prerelease` flag to the command. You can view preview functionality on the [Azure Functions SQL Extensions release page](#).

Bash

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Sql --
```

① Note

Breaking changes between preview releases of the Azure SQL bindings for Azure Functions requires that all Functions targeting the same database use the same version of the SQL extension package.

SQL connection string

Azure SQL bindings for Azure Functions have a required property for the connection string on all bindings and triggers. These pass the connection string to the `Microsoft.Data.SqlClient` library and supports the connection string as defined in the [SqlClient ConnectionString documentation](#). Notable keywords include:

- `Authentication` allows a function to connect to Azure SQL with Microsoft Entra ID, including [Active Directory Managed Identity](#)
- `Command Timeout` allows a function to wait for specified amount of time in seconds before terminating a query (default 30 seconds)
- `ConnectRetryCount` allows a function to automatically make additional reconnection attempts, especially applicable to Azure SQL Database serverless tier (default 1)
- `Pooling` allows a function to reuse connections to the database, which can improve performance (default `true`). Additional settings for connection pooling include `Connection Lifetime`, `Max Pool Size`, and `Min Pool Size`. Learn more about connection pooling in the [ADO.NET documentation](#)

Considerations

- Azure SQL binding supports version 4.x and later of the Functions runtime.
- Source code for the Azure SQL bindings can be found in [this GitHub repository](#).
- This binding requires connectivity to an Azure SQL or SQL Server database.
- Output bindings against tables with columns of data types `NTEXT`, `TEXT`, or `IMAGE` aren't supported and data upserts will fail. These types [will be removed](#) in a future version of SQL Server and aren't compatible with the `OPENJSON` function used by this Azure Functions binding.

Samples

In addition to the samples for C#, Java, JavaScript, PowerShell, and Python available in the [Azure SQL bindings GitHub repository](#), more are available in Azure Samples:

- [C# ToDo API sample with Azure SQL bindings](#)
- [Use SQL bindings in Azure Stream Analytics](#)
- [Send data from Azure SQL with Python](#)

Next steps

- [Read data from a database \(Input binding\)](#)
- [Save data to a database \(Output binding\)](#)
- [Run a function when data is changed in a SQL table \(Trigger\)](#)
- [Learn how to connect Azure Function to Azure SQL with managed identity](#)

Azure SQL trigger for Functions

Article • 06/26/2024

ⓘ Note

In consumption plan functions, automatic scaling is not supported for SQL trigger. If the automatic scaling process stops the function, all processing of events will stop and it will need to be manually restarted.

Use premium or dedicated plans for [scaling benefits](#) with SQL trigger.

The Azure SQL trigger uses [SQL change tracking](#) functionality to monitor a SQL table for changes and trigger a function when a row is created, updated, or deleted. For configuration details for change tracking for use with the Azure SQL trigger, see [Set up change tracking](#). For information on setup details of the Azure SQL extension for Azure Functions, see the [SQL binding overview](#).

The Azure SQL trigger scaling decisions for the Consumption and Premium plans are done via target-based scaling. For more information, see [Target-based scaling](#).

Functionality Overview

The Azure SQL trigger binding uses a polling loop to check for changes, triggering the user function when changes are detected. At a high level, the loop looks like this:

```
while (true) {
    1. Get list of changes on table - up to a maximum number controlled by
       the Sql_Trigger_MaxBatchSize setting
    2. Trigger function with list of changes
    3. Wait for delay controlled by Sql_Trigger_PollingIntervalMs setting
}
```

Changes are processed in the order that their changes were made, with the oldest changes being processed first. A couple notes about change processing:

1. If changes to multiple rows are made at once the exact order that they are sent to the function is based on the order returned by the CHANGETABLE function
2. Changes are "batched" together for a row. If multiple changes are made to a row between each iteration of the loop then only a single change entry exists for that

row which will show the difference between the last processed state and the current state

3. If changes are made to a set of rows, and then another set of changes are made to half of those same rows, then the half of the rows that weren't changed a second time are processed first. This processing logic is due to the above note with the changes being batched - the trigger will only see the "last" change made and use that for the order it processes them in

For more information on change tracking and how it's used by applications such as Azure SQL triggers, see [work with change tracking](#).

Example usage

Isolated worker model

More samples for the Azure SQL trigger are available in the [GitHub repository](#).

The example refers to a `ToDoItem` class and a corresponding database table:

C#

```
namespace AzureSQL.ToDo
{
    public class ToDoItem
    {
        public Guid Id { get; set; }
        public int? order { get; set; }
        public string title { get; set; }
        public string url { get; set; }
        public bool? completed { get; set; }
    }
}
```

SQL

```
CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

[Change tracking](#) is enabled on the database and on the table:

SQL

```
ALTER DATABASE [SampleDatabase]
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON);

ALTER TABLE [dbo].[ToDo]
ENABLE CHANGE_TRACKING;
```

The SQL trigger binds to a `IReadOnlyList<SqlChange<T>>`, a list of `SqlChange` objects each with two properties:

- **Item:** the item that was changed. The type of the item should follow the table schema as seen in the `ToDoItem` class.
- **Operation:** a value from `SqlChangeOperation` enum. The possible values are `Insert`, `Update`, and `Delete`.

The following example shows a [C# function](#) that is invoked when there are changes to the `ToDo` table:

CS

```
using System;
using System.Collections.Generic;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Sql;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

namespace AzureSQL.ToDo
{
    public static class ToDoTrigger
    {
        [Function("ToDoTrigger")]
        public static void Run(
            [SqlTrigger("[dbo].[ToDo]", "SqlConnectionString")]
            IReadOnlyList<SqlChange<ToDoItem>> changes,
            FunctionContext context)
        {
            var logger = context.GetLogger("ToDoTrigger");
            foreach (SqlChange<ToDoItem> change in changes)
            {
                ToDoItem toItem = change.Item;
                logger.LogInformation($"Change operation:
{change.Operation}");
                logger.LogInformation($"Id: {toItem.Id}, Title:
{toItem.title}, Url: {toItem.url}, Completed:
{toItem.completed}");
            }
        }
    }
}
```

```
    }  
}
```

Attributes

The [C# library](#) uses the [SqlTrigger](#) attribute to declare the SQL trigger on the function, which has the following properties:

[+] [Expand table](#)

Attribute property	Description
TableName	Required. The name of the table monitored by the trigger.
ConnectionStringSetting	Required. The name of an app setting that contains the connection string for the database containing the table monitored for changes. The connection string setting name corresponds to the application setting (in <code>local.settings.json</code> for local development) that contains the connection string to the Azure SQL or SQL Server instance.
LeasesTableName	Optional. Name of the table used to store leases. If not specified, the leases table name will be <code>Leases_{FunctionId}_{TableId}</code> . More information on how this is generated can be found here .

Optional Configuration

The following optional settings can be configured for the SQL trigger for local development or for cloud deployments.

host.json

This section describes the configuration settings available for this binding in versions 2.x and higher. Settings in the host.json file apply to all functions in a function app instance. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

[+] [Expand table](#)

Setting	Default	Description
MaxBatchSize	100	The maximum number of changes processed with each iteration of the trigger loop before being sent to the triggered function.
PollingIntervalMs	1000	The delay in milliseconds between processing each batch of changes. (1000 ms is 1 second)
MaxChangesPerWorker	1000	The upper limit on the number of pending changes in the user table that are allowed per application-worker. If the count of changes exceeds this limit, it might result in a scale-out. The setting only applies for Azure Function Apps with runtime driven scaling enabled .

Example host.json file

Here is an example host.json file with the optional settings:

JSON

```
{
  "version": "2.0",
  "extensions": {
    "Sql": {
      "MaxBatchSize": 300,
      "PollingIntervalMs": 1000,
      "MaxChangesPerWorker": 100
    }
  },
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      }
    },
    "logLevel": {
      "default": "Trace"
    }
  }
}
```

local.setting.json

The local.settings.json file stores app settings and settings used by local development tools. Settings in the local.settings.json file are used only when you're running your

project locally. When you publish your project to Azure, be sure to also add any required settings to the app settings for the function app.

ⓘ Important

Because the local.settings.json may contain secrets, such as connection strings, you should never store it in a remote repository. Tools that support Functions provide ways to synchronize settings in the local.settings.json file with the [app settings](#) in the function app to which your project is deployed.

[+] Expand table

Setting	Default	Description
Sql_Trigger_BatchSize	100	The maximum number of changes processed with each iteration of the trigger loop before being sent to the triggered function.
Sql_Trigger_PollingIntervalMs	1000	The delay in milliseconds between processing each batch of changes. (1000 ms is 1 second)
Sql_Trigger_MaxChangesPerWorker	1000	The upper limit on the number of pending changes in the user table that are allowed per application-worker. If the count of changes exceeds this limit, it might result in a scale-out. The setting only applies for Azure Function Apps with runtime driven scaling enabled .

Example local.settings.json file

Here is an example local.settings.json file with the optional settings:

JSON
<pre>{ "IsEncrypted": false, "Values": { "AzureWebJobsStorage": "UseDevelopmentStorage=true", "FUNCTIONS_WORKER_RUNTIME": "dotnet", "SqlConnectionString": "", "Sql_Trigger_MaxBatchSize": 300, "Sql_Trigger_PollingIntervalMs": 1000, "Sql_Trigger_MaxChangesPerWorker": 100 } }</pre>

Set up change tracking (required)

Setting up change tracking for use with the Azure SQL trigger requires two steps. These steps can be completed from any SQL tool that supports running queries, including [Visual Studio Code](#), [Azure Data Studio](#) or [SQL Server Management Studio](#).

1. Enable change tracking on the SQL database, substituting `your database name` with the name of the database where the table to be monitored is located:

SQL

```
ALTER DATABASE [your database name]
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON);
```

The `CHANGE_RETENTION` option specifies the time period for which change tracking information (change history) is kept. The retention of change history by the SQL database might affect trigger functionality. For example, if the Azure Function is turned off for several days and then resumed, the database will contain the changes that occurred in past two days in the above setup example.

The `AUTO_CLEANUP` option is used to enable or disable the clean-up task that removes old change tracking information. If a temporary problem that prevents the trigger from running, turning off auto cleanup can be useful to pause the removal of information older than the retention period until the problem is resolved.

More information on change tracking options is available in the [SQL documentation](#).

2. Enable change tracking on the table, substituting `your table name` with the name of the table to be monitored (changing the schema if appropriate):

SQL

```
ALTER TABLE [dbo].[your table name]
ENABLE CHANGE_TRACKING;
```

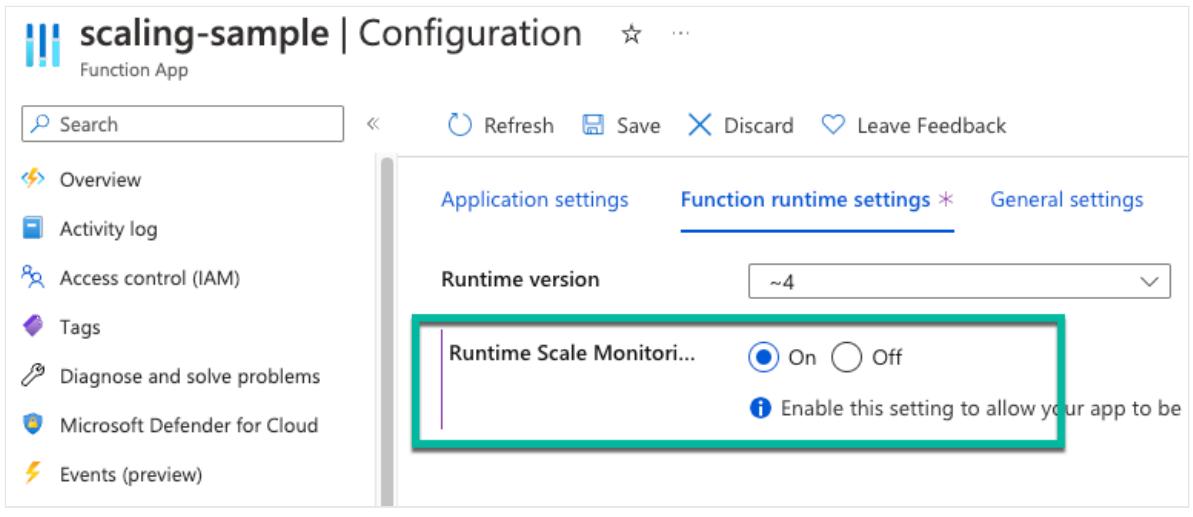
The trigger needs to have read access on the table being monitored for changes and to the change tracking system tables. Each function trigger has an associated change tracking table and leases table in a schema `az_func`. These tables are created by the trigger if they don't yet exist. More information on these data structures is available in the Azure SQL binding library [documentation](#).

Enable runtime-driven scaling

Optionally, your functions can scale automatically based on the number of changes that are pending to be processed in the user table. To allow your functions to scale properly on the Premium plan when using SQL triggers, you need to enable runtime scale monitoring.

Azure portal

In the Azure portal, in your function app, choose **Configuration** and on the **Function runtime settings** tab turn **Runtime scale monitoring** to **On**.



Retry support

Further information on the SQL trigger [retry support](#) and [leases tables](#) is available in the GitHub repository.

Startup retries

If an exception occurs during startup then the host runtime automatically attempts to restart the trigger listener with an exponential backoff strategy. These retries continue until either the listener is successfully started or the startup is canceled.

Broken connection retries

If the function successfully starts but then an error causes the connection to break (such as the server going offline) then the function continues to try and reopen the connection until the function is either stopped or the connection succeeds. If the

connection is successfully re-established then it picks up processing changes where it left off.

Note that these retries are outside the built-in idle connection retry logic that `SqlClient` has which can be configured with the `ConnectRetryCount` and `ConnectRetryInterval` [connection string options](#). The built-in idle connection retries are attempted first and if those fail to reconnect then the trigger binding attempts to re-establish the connection itself.

Function exception retries

If an exception occurs in the user function when processing changes then the batch of rows currently being processed are retried again in 60 seconds. Other changes are processed as normal during this time, but the rows in the batch that caused the exception are ignored until the timeout period has elapsed.

If the function execution fails five times in a row for a given row then that row is completely ignored for all future changes. Because the rows in a batch are not deterministic, rows in a failed batch might end up in different batches in subsequent invocations. This means that not all rows in the failed batch will necessarily be ignored. If other rows in the batch were the ones causing the exception, the "good" rows might end up in a different batch that doesn't fail in future invocations.

Next steps

- [Read data from a database \(Input binding\)](#)
 - [Save data to a database \(Output binding\)](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Azure SQL input binding for Azure Functions

Article • 06/26/2024

When a function runs, the Azure SQL input binding retrieves data from a database and passes it to the input parameter of the function.

For information on setup and configuration details, see the [overview](#).

Examples

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

More samples for the Azure SQL input binding are available in the [GitHub repository](#).

This section contains the following examples:

- [HTTP trigger, get row by ID from query string](#)
- [HTTP trigger, get multiple rows from route data](#)
- [HTTP trigger, delete rows](#)

The examples refer to a `ToDoItem` class and a corresponding database table:

C#

```
namespace AzureSQL.ToDo
{
    public class ToDoItem
    {
        public Guid Id { get; set; }
        public int? order { get; set; }
        public string title { get; set; }
        public string url { get; set; }
        public bool? completed { get; set; }
    }
}
```

SQL

```
CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

HTTP trigger, get row by ID from query string

The following example shows a [C# function](#) that retrieves a single record. The function is [triggered by an HTTP request](#) that uses a query string to specify the ID. That ID is used to retrieve a `ToDoItem` record with the specified query.

ⓘ Note

The HTTP query string parameter is case-sensitive.

cs

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Sql;
using Microsoft.Azure.Functions.Worker.Http;

namespace AzureSQLSamples
{
    public static class GetToDoItem
    {
        [FunctionName("GetToDoItem")]
    }
}
```

```

        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
"gettodoitem")]
            HttpRequest req,
            [SqlInput(commandText: "select [Id], [order], [title],
[url], [completed] from dbo.ToDo where Id = @Id",
            commandType: System.Data.CommandType.Text,
            parameters: "@Id={Query.id}",
            connectionStringSetting: "SqlConnectionString")]
            IEnumerable<ToDoItem> ToDoItem)
        {
            return new OkObjectResult(ToDoItem.FirstOrDefault());
        }
    }
}

```

HTTP trigger, get multiple rows from route parameter

The following example shows a [C# function](#) that retrieves documents returned by the query. The function is [triggered by an HTTP request](#) that uses route data to specify the value of a query parameter. That parameter is used to filter the `ToDoItem` records in the specified query.

cs

```

using System.Collections.Generic;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Extensions.Sql;
using Microsoft.Azure.Functions.Worker.Http;

namespace AzureSQLSamples
{
    public static class GetToDoItems
    {
        [FunctionName("GetToDoItems")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
"gettodoitems/{priority}")]
            HttpRequest req,
            [SqlInput(commandText: "select [Id], [order], [title],
[url], [completed] from dbo.ToDo where [Priority] > @Priority",
            commandType: System.Data.CommandType.Text,
            parameters: "@Priority={priority}",
            connectionStringSetting: "SqlConnectionString")]
            IEnumerable<ToDoItem> ToDoItems)
        {
            return new OkObjectResult(ToDoItems);
        }
    }
}

```

```
}
```

HTTP trigger, delete rows

The following example shows a [C# function](#) that executes a stored procedure with input from the HTTP request query parameter.

The stored procedure `dbo.DeleteToDo` must be created on the SQL database. In this example, the stored procedure deletes a single record or all records depending on the value of the parameter.

SQL

```
CREATE PROCEDURE [dbo].[DeleteToDo]
@Id NVARCHAR(100)
AS
    DECLARE @UID UNIQUEIDENTIFIER = TRY_CAST(@ID AS UNIQUEIDENTIFIER)
    IF @UID IS NOT NULL AND @Id != ''
        BEGIN
            DELETE FROM dbo.ToDo WHERE Id = @UID
        END
    ELSE
        BEGIN
            DELETE FROM dbo.ToDo WHERE @ID = ''
        END

    SELECT [Id], [order], [title], [url], [completed] FROM dbo.ToDo
GO
```

CS

```
namespace AzureSQL ToDo
{
    public static class DeleteToDo
    {
        // delete all items or a specific item from querystring
        // returns remaining items
        // uses input binding with a stored procedure DeleteToDo to
        delete items and return remaining items
        [FunctionName("DeleteToDo")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "delete", Route =
"DeleteFunction")] HttpRequest req,
            ILogger log,
            [SqlInput(commandText: "DeleteToDo", commandType:
System.Data.CommandType.StoredProcedure,
            parameters: "@Id={Query.id}", connectionStringSetting:
"SqlConnectionString")]
```

```

        IEnumerable<ToDoItem> ToDoItems)
    {
        return new OkObjectResult(ToDoItems);
    }
}

```

Attributes

The [C# library](#) uses the [SqlAttribute](#) attribute to declare the SQL bindings on the function, which has the following properties:

[\[+\] Expand table](#)

Attribute property	Description
CommandText	Required. The Transact-SQL query command or name of the stored procedure executed by the binding.
ConnectionStringSetting	Required. The name of an app setting that contains the connection string for the database against which the query or stored procedure is being executed. This value isn't the actual connection string and must instead resolve to an environment variable name.
 CommandType	Required. A CommandType value, which is Text for a query and StoredProcedure for a stored procedure.
 Parameters	Optional. Zero or more parameter values passed to the command during execution as a single string. Must follow the format <code>@param1=param1,@param2=param2</code> . Neither the parameter name nor the parameter value can contain a comma (,) or an equals sign (=).

When you're developing locally, add your application settings in the [local.settings.json](#) file in the `values` collection.

Usage

The attribute's constructor takes the SQL command text, the command type, parameters, and the connection string setting name. The command can be a Transact-SQL (T-SQL) query with the command type `System.Data.CommandType.Text` or stored procedure name with the command type `System.Data.CommandType.StoredProcedure`. The connection string setting name corresponds to the application setting (in `local.settings.json` for local development) that contains the [connection string](#) to the Azure SQL or SQL Server instance.

Queries executed by the input binding are [parameterized](#) in Microsoft.Data.SqlClient to reduce the risk of [SQL injection](#) from the parameter values passed into the binding.

If an exception occurs when a SQL input binding is executed then the function code won't execute. This may result in an error code being returned, such as an HTTP trigger returning a 500 error code.

Next steps

- [Save data to a database \(Output binding\)](#)
 - [Run a function when data is changed in a SQL table \(Trigger\)](#)
 - [Run a function from a HTTP request \(trigger\)](#)
 - [Review ToDo API sample with Azure SQL bindings](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Azure SQL output binding for Azure Functions

Article • 06/26/2024

The Azure SQL output binding lets you write to a database.

For information on setup and configuration details, see the [overview](#).

Examples

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

More samples for the Azure SQL output binding are available in the [GitHub repository](#).

This section contains the following examples:

- [HTTP trigger, write one record](#)
- [HTTP trigger, write to two tables](#)

The examples refer to a `ToDoItem` class and a corresponding database table:

C#

```
namespace AzureSQL.ToDo
{
```

```
public class ToDoItem
{
    public Guid Id { get; set; }
    public int? order { get; set; }
    public string title { get; set; }
    public string url { get; set; }
    public bool? completed { get; set; }
}
```

SQL

```
CREATE TABLE dbo.ToDo (
    [Id] UNIQUEIDENTIFIER PRIMARY KEY,
    [order] INT NULL,
    [title] NVARCHAR(200) NOT NULL,
    [url] NVARCHAR(200) NOT NULL,
    [completed] BIT NOT NULL
);
```

To return [multiple output bindings](#) in our samples, we'll create a custom return type:

cs

```
public static class OutputType
{
    [SqlOutput("dbo.ToDo", connectionStringSetting:
    "SqlConnectionString")]
    public ToDoItem ToDoItem { get; set; }
    public HttpResponseMessage HttpResponse { get; set; }
}
```

HTTP trigger, write one record

The following example shows a [C# function](#) that adds a record to a database, using data provided in an HTTP POST request as a JSON body. The return object is the `OutputType` class we created to handle both an HTTP response and the SQL output binding.

cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Microsoft.Azure.Functions.Worker.Extensions.Sql;
using Microsoft.Azure.Functions.Worker;
```

```

using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

namespace AzureSQL.ToDo
{
    public static class PostToDo
    {
        // create a new ToDoItem from body object
        // uses output binding to insert new item into ToDo table
        [FunctionName("PostToDo")]
        public static async Task<OutputType> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route =
"PostFunction")] HttpRequestData req,
            FunctionContext executionContext)
        {
            var logger = executionContext.GetLogger("PostToDo");
            logger.LogInformation("C# HTTP trigger function processed a
request.");

            string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
            ToDoItem toItem = JsonConvert.DeserializeObject<ToDoItem>
(requestBody);

            // generate a new id for the todo item
            toItem.Id = Guid.NewGuid();

            // set Url from env variable ToDoUri
            toItem.url =
Environment.GetEnvironmentVariable("ToDoUri")+"?
id="+toItem.Id.ToString();

            // if completed is not provided, default to false
            if (toItem.completed == null)
            {
                toItem.completed = false;
            }

            return new OutputType()
            {
                ToDoItem = toItem,
                HttpResponseMessage =
req.CreateResponse(System.Net.HttpStatusCode.Created)
            }
        }

        public static class OutputType
        {
            [SqlOutput("dbo.ToDo", connectionStringSetting:
"SqlConnectionString")]
            public ToDoItem ToDoItem { get; set; }

            public HttpResponseMessage HttpResponse { get; set; }
        }
    }
}

```

```
    }  
}
```

HTTP trigger, write to two tables

The following example shows a [C# function](#) that adds records to a database in two different tables (`dbo.ToDo` and `dbo.RequestLog`), using data provided in an HTTP POST request as a JSON body and multiple output bindings.

SQL

```
CREATE TABLE dbo.RequestLog (  
    Id int identity(1,1) primary key,  
    RequestTimeStamp datetime2 not null,  
    ItemCount int not null  
)
```

To use an additional output binding, we add a class for `RequestLog` and modify our `OutputType` class:

cs

```
using System;  
using System.Collections.Generic;  
using System.IO;  
using System.Threading.Tasks;  
using Microsoft.Azure.Functions.Worker.Extensions.Sql;  
using Microsoft.Azure.Functions.Worker;  
using Microsoft.Azure.Functions.Worker.Http;  
using Microsoft.Extensions.Logging;  
using Newtonsoft.Json;  
  
namespace AzureSQL.ToDo  
{  
    public static class PostToDo  
    {  
        // create a new ToDoItem from body object  
        // uses output binding to insert new item into ToDo table  
        [FunctionName("PostToDo")]  
        public static async Task<OutputType> Run(  
            [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route =  
"PostFunction")] HttpRequestData req,  
            FunctionContext executionContext)  
        {  
            string requestBody = await new  
StreamReader(req.Body).ReadToEndAsync();  
            ToDoItem ToDoItem = JsonConvert.DeserializeObject<ToDoItem>(requestBody);
```

```

        // generate a new id for the todo item
        ToDoItem.Id = Guid.NewGuid();

        // set Url from env variable ToDoUri
        ToDoItem.url =
Environment.GetEnvironmentVariable("ToDoUri")+"?
id="+ToDoItem.Id.ToString();

        // if completed is not provided, default to false
        if (ToDoItem.completed == null)
        {
            ToDoItem.completed = false;
        }

        requestLog = new RequestLog();
        requestLog.RequestTimeStamp = DateTime.Now;
        requestLog.ItemCount = 1;

        return new OutputType()
        {
            ToDoItem = ToDoItem,
            RequestLog = requestLog,
            HttpResponse =
req.CreateResponse(System.Net.HttpStatusCode.Created)
        }
    }
}

public class RequestLog {
    public DateTime RequestTimeStamp { get; set; }
    public int ItemCount { get; set; }
}

public static class OutputType
{
    [SqlOutput("dbo.ToDo", connectionStringSetting:
"SqlConnectionConnectionString")]
    public ToDoItem ToDoItem { get; set; }

    [SqlOutput("dbo.RequestLog", connectionStringSetting:
"SqlConnectionConnectionString")]
    public RequestLog RequestLog { get; set; }

    public HttpResponseMessage HttpResponse { get; set; }
}

}

```

Attributes

The C# library uses the [SqlAttribute](#) attribute to declare the SQL bindings on the function, which has the following properties:

[+] Expand table

Attribute property	Description
CommandText	Required. The name of the table being written to by the binding.
ConnectionStringSetting	Required. The name of an app setting that contains the connection string for the database to which data is being written. This isn't the actual connection string and must instead resolve to an environment variable.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

Usage

The `CommandText` property is the name of the table where the data is to be stored. The connection string setting name corresponds to the application setting that contains the [connection string](#) to the Azure SQL or SQL Server instance.

The output bindings use the T-SQL [MERGE](#) statement which requires [SELECT](#) permissions on the target database.

If an exception occurs when a SQL output binding is executed then the function code stop executing. This may result in an error code being returned, such as an HTTP trigger returning a 500 error code. If the `IAsyncCollector` is used in a .NET function then the function code can handle exceptions throw by the call to `FlushAsync()`.

Next steps

- [Read data from a database \(Input binding\)](#)
- [Run a function when data is changed in a SQL table \(Trigger\)](#)
- [Review ToDo API sample with Azure SQL bindings](#)

Feedback

Was this page helpful?

Yes

No

Provide product feedback ↗

Azure Blob storage bindings for Azure Functions overview

Article • 09/27/2023

Azure Functions integrates with [Azure Storage](#) via [triggers and bindings](#). Integrating with Blob storage allows you to build functions that react to changes in blob data as well as read and write values.

Action	Type
Run a function as blob storage data changes	Trigger
Read blob storage data in a function	Input binding
Allow a function to write blob storage data	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

The functionality of the extension varies depending on the extension version:

Extension 5.x and higher

This version introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

This version allows you to bind to types from [Azure.Storage.Blobs](#). Learn more about how these new types are different from `WindowsAzure.Storage` and `Microsoft.Azure.Storage` and how to migrate to them from the [Azure.Storage.Blobs Migration Guide](#).

Add the extension to your project by installing the [Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs NuGet package](#), version 5.x.

Using the .NET CLI:

.NET CLI

```
dotnet add package  
Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs --version  
5.0.0
```

① Note

Azure Blobs, Azure Queues, and Azure Tables now use separate extensions and are referenced individually. For example, to use the triggers and bindings for all three services in your .NET isolated-process app, you should add the following packages to your project:

- [Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs](#)
- [Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues](#)
- [Microsoft.Azure.Functions.Worker.Extensions.Tables](#)

Previously, the extensions shipped together as [Microsoft.Azure.Functions.Worker.Extensions.Storage, version 4.x](#). This same package also has a [5.x version](#), which references the split packages for blobs and queues only. When upgrading your package references from older versions, you may therefore need to additionally reference the new [Microsoft.Azure.Functions.Worker.Extensions.Tables](#) NuGet package. Also, when referencing these newer split packages, make sure you are not referencing an older version of the combined storage package, as this will result in conflicts from two definitions of the same bindings.

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see binding type details for the mode and version.

Extension 5.x and higher

The isolated worker process supports parameter types according to the tables below.

Blob trigger

The blob trigger can bind to the following types:

Type	Description
<code>string</code>	The blob content as a string. Use when the blob content is simple text.
<code>byte[]</code>	The bytes of the blob content.
JSON serializable types	When a blob contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
<code>Stream</code> ¹	An input stream of the blob content.
<code>BlobClient</code> ¹ , <code>BlockBlobClient</code> ¹ , <code>PageBlobClient</code> ¹ , <code>AppendBlobClient</code> ¹ , <code>BlobBaseClient</code> ¹	A client connected to the blob. This set of types offers the most control for processing the blob and can be used to write back to the blob if the connection has sufficient permission.

¹ To use these types, you need to reference

[Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs 6.0.0 or later](#) and the common dependencies for SDK type bindings.

Blob input binding

When you want the function to process a single blob, the blob input binding can bind to the following types:

Type	Description
<code>string</code>	The blob content as a string. Use when the blob content is simple text.

Type	Description
<code>byte[]</code>	The bytes of the blob content.
JSON serializable types	When a blob contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
<code>Stream</code> ¹	An input stream of the blob content.
<code>BlobClient</code> ¹ , <code>BlockBlobClient</code> ¹ , <code>PageBlobClient</code> ¹ , <code>AppendBlobClient</code> ¹ , <code>BlobBaseClient</code> ¹	A client connected to the blob. This set of types offers the most control for processing the blob and can be used to write back to it if the connection has sufficient permission.

When you want the function to process multiple blobs from a container, the blob input binding can bind to the following types:

Type	Description
<code>T[]</code> or <code>List<T></code> where <code>T</code> is one of the single blob input binding types	An array or list of multiple blobs. Each entry represents one blob from the container. You can also bind to any interfaces implemented by these types, such as <code>IEnumerable<T></code> .
<code>BlobContainerClient</code> ¹	A client connected to the container. This type offers the most control for processing the container and can be used to write to it if the connection has sufficient permission.

¹ To use these types, you need to reference

[Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs 6.0.0 or later](#) and the [common dependencies for SDK type bindings](#).

Blob output binding

When you want the function to write to a single blob, the blob output binding can bind to the following types:

Type	Description
<code>string</code>	The blob content as a string. Use when the blob content is simple text.
<code>byte[]</code>	The bytes of the blob content.
JSON serializable types	An object representing the content of a JSON blob. Functions attempts to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write to multiple blobs, the blob output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is one of the single blob output binding types	An array containing content for multiple blobs. Each entry represents the content of one blob.

For other output scenarios, create and use types from [Azure.Storage.Blobs](#) directly.

host.json settings

This section describes the function app configuration settings available for functions that use this binding. These settings only apply when using extension version 5.0.0 and higher. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

ⓘ Note

This section doesn't apply to extension versions before 5.0.0. For those earlier versions, there aren't any function app-wide configuration settings for blobs.

JSON

```
{  
  "version": "2.0",  
  "extensions": {  
    "blobs": {  
      "maxDegreeOfParallelism": 4,  
      "poisonBlobThreshold": 1  
    }  
  }  
}
```

Property	Default	Description
maxDegreeOfParallelism	8 * (the number of available cores)	The integer number of concurrent invocations allowed for all blob-triggered functions in a given function app. The minimum allowed value is 1.
poisonBlobThreshold	5	The integer number of times to try processing a message before moving it to the poison queue. The minimum allowed value is 1.

Next steps

- Run a function when blob storage data changes
- Read blob storage data when a function runs
- Write blob storage data from a function

Azure Blob storage trigger for Azure Functions

Article • 05/14/2024

The Blob storage trigger starts a function when a new or updated blob is detected. The blob contents are provided as [input to the function](#).

Tip

There are several ways to execute your function code based on changes to blobs in a storage container. If you choose to use the Blob storage trigger, note that there are two implementations offered: a polling-based one (referenced in this article) and an event-based one. It is recommended that you use the [event-based implementation](#) as it has lower latency than the other. Also, the Flex Consumption plan supports only the event-based Blob storage trigger.

For details about differences between the two implementations of the Blob storage trigger, as well as other triggering options, see [Working with blobs](#).

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

The following example is a [C# function](#) that runs in an isolated worker process and uses a blob trigger with both blob input and blob output blob bindings. The function is triggered by the creation of a blob in the *test-samples-trigger* container. It reads a text file from the *test-samples-input* container and creates a new text file in an output container based on the name of the triggered file.

C#

```
public static class BlobFunction
{
```

```

[Function(nameof(BlobFunction))]
[BlobOutput("test-samples-output/{name}-output.txt")]
public static string Run(
    [BlobTrigger("test-samples-trigger/{name}")] string myTriggerItem,
    [BlobInput("test-samples-input/sample1.txt")] string myBlob,
    FunctionContext context)
{
    var logger = context.GetLogger("BlobFunction");
    logger.LogInformation("Triggered Item = {myTriggerItem}", myTriggerItem);
    logger.LogInformation("Input Item = {myBlob}", myBlob);

    // Blob Output
    return "blob-output content";
}
}

```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the `BlobAttribute` attribute to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

The attribute's constructor takes the following parameters:

[Expand table](#)

Parameter	Description
<code>BlobPath</code>	The path to the blob.
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to Azure Blobs. See Connections .
<code>Access</code>	Indicates whether you will be reading or writing.
<code>Source</code>	Sets the source of the triggering event. Use <code>BlobTriggerSource.EventGrid</code> for an Event Grid-based blob trigger , which provides much lower latency. The default is <code>BlobTriggerSource.LogsAndContainerScan</code> , which uses the standard polling mechanism to detect changes in the container.

Isolated worker model

Here's an `BlobTrigger` attribute in a method signature:

C#

```

[Function(nameof(BlobFunction))]
[BlobOutput("test-samples-output/{name}-output.txt")]
public static string Run(
    [BlobTrigger("test-samples-trigger/{name}")] string myTriggerItem,
    [BlobInput("test-samples-input/sample1.txt")] string myBlob,
    FunctionContext context)

```

When you're developing locally, add your application settings in the `local.settings.json` file in the `Values` collection.

See the [Example section](#) for complete examples.

Metadata

The blob trigger provides several metadata properties. These properties can be used as part of binding expressions in other bindings or as parameters in your code. These values have the same semantics as the [CloudBlob](#) type.

[Expand table](#)

Property	Type	Description
<code>BlobTrigger</code>	<code>string</code>	The path to the triggering blob.
<code>Uri</code>	<code>System.Uri</code>	The blob's URI for the primary location.
<code>Properties</code>	<code>BlobProperties</code>	The blob's system properties.
<code>Metadata</code>	<code>IDictionary<string, string></code>	The user-defined metadata for the blob.

The following example logs the path to the triggering blob, including the container:

C#

```
public static void Run(string myBlob, string blobTrigger, ILogger log)
{
    log.LogInformation($"Full blob path: {blobTrigger}");
}
```

Usage

The binding types supported by Blob trigger depend on the extension package version and the C# modality used in your function app.

Isolated worker model

The blob trigger can bind to the following types:

[Expand table](#)

Type	Description
<code>string</code>	The blob content as a string. Use when the blob content is simple text.
<code>byte[]</code>	The bytes of the blob content.
JSON serializable types	When a blob contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
<code>Stream</code> ¹	An input stream of the blob content.
<code>BlobClient</code> ¹ , <code>BlockBlobClient</code> ¹ , <code>PageBlobClient</code> ¹ , <code>AppendBlobClient</code> ¹ , <code>BlobBaseClient</code> ¹	A client connected to the blob. This set of types offers the most control for processing the blob and can be used to write back to the blob if the connection has sufficient permission.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs 6.0.0 or later](#) and the common dependencies for SDK type bindings.

Binding to `string`, or `Byte[]` is only recommended when the blob size is small. This is recommended because the entire blob contents are loaded into memory. For most blobs, use a `Stream` or `BlobClient` type. For more information, see [Concurrency and memory usage](#).

If you get an error message when trying to bind to one of the Storage SDK types, make sure that you have a reference to [the correct Storage SDK version](#).

You can also use the [StorageAccountAttribute](#) to specify the storage account to use. You can do this when you need to use a different storage account than other functions in the library. The constructor takes the name of an app setting that contains a storage connection string. The attribute can be applied at the parameter, method, or class level. The following example shows class level and method level:

C#

```
[StorageAccount("ClassLevelStorageAppSetting")]
public static class AzureFunctions
{
    [FunctionName("BlobTrigger")]
    [StorageAccount("FunctionLevelStorageAppSetting")]
    public static void Run( //...
    {
        ....
    }
}
```

The storage account to use is determined in the following order:

- The `BlobTrigger` attribute's `Connection` property.
- The `StorageAccount` attribute applied to the same parameter as the `BlobTrigger` attribute.
- The `StorageAccount` attribute applied to the function.
- The `StorageAccount` attribute applied to the class.
- The default storage account for the function app, which is defined in the `AzureWebJobsStorage` application setting.

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to Azure Blobs. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Manage storage account access keys](#). The connection string must be for a general-purpose storage account, not a [Blob storage account](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage". If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [version 5.x or higher of the extension \(bundle 3.x or higher for non-.NET language stacks\)](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

[] [Expand table](#)

Property	Environment variable template	Description	Example value
Blob Service URI	<code><CONNECTION_NAME_PREFIX>__serviceUri</code> ¹	The data plane URI of the blob service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>

¹ `<CONNECTION_NAME_PREFIX>__blobServiceUri` can be used as an alias. If the connection configuration will be used by a blob trigger, `blobServiceUri` must also be accompanied by `queueServiceUri`. See below.

The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables. The URI can only designate the blob service. As an alternative, you can provide a URI specifically for each service, allowing a single connection to be used. If both versions are provided, the multi-service form is used. To configure the connection for multiple services, instead of

`<CONNECTION_NAME_PREFIX>__serviceUri`, set:

[] [Expand table](#)

Property	Environment variable template	Description	Example value
Blob Service URI	<code><CONNECTION_NAME_PREFIX>__blobServiceUri</code>	The data plane URI of the blob service to which you're connecting, using the	<code>https://<storage_account_name>.blob.core.windows.net</code>

Property	Environment variable template	Description	Example value
		HTTPS scheme.	
Queue Service URI (required for blob triggers ²)	<CONNECTION_NAME_PREFIX>__queueServiceUri	The data plane URI of a queue service, using the HTTPS scheme. This value is only needed for blob triggers.	https://<storage_account_name>.queue.core.windows.net

² The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue. In the `serviceUri` form, the `AzureWebJobsStorage` connection is used. However, when specifying `blobServiceUri`, a queue service URI must also be provided with `queueServiceUri`. It's recommended that you use the service from the same storage account as the blob service. You also need to make sure the trigger can read and write messages in the configured queue service by assigning a role like [Storage Queue Data Contributor](#).

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the [principle of least privilege](#), granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You need to create a role assignment that provides access to your blob container at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Blob Storage extension in normal operation. Your application may require further permissions based on the code you write.

Binding type	Example built-in roles
Trigger	Storage Blob Data Owner and Storage Queue Data Contributor ¹ Extra permissions must also be granted to the AzureWebJobsStorage connection. ²
Input binding	Storage Blob Data Reader
Output binding	Storage Blob Data Owner

¹ The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue on the storage account specified by the connection.

² The AzureWebJobsStorage connection is used internally for blobs and queues that enable the trigger. If it's configured to use an identity-based connection, it needs extra permissions beyond the default requirement. The required permissions are covered by the [Storage Blob Data Owner](#), [Storage Queue Data Contributor](#), and [Storage Account Contributor](#) roles. To learn more, see [Connecting to host storage with an identity](#).

Blob name patterns

You can specify a blob name pattern in the `path` property in `function.json` or in the `BlobTrigger` attribute constructor. The name pattern can be a [filter or binding expression](#). The following sections provide examples.

 Tip

A container name can't contain a resolver in the name pattern.

Get file name and extension

The following example shows how to bind to the blob file name and extension separately:

JSON

```
"path": "input/{blobname}.{blobextension}",
```

If the blob is named *original-Blob1.txt*, the values of the `blobname` and `blobextension` variables in function code are *original-Blob1* and *txt*.

Filter on blob name

The following example triggers only on blobs in the `input` container that start with the string "original-":

JSON

```
"path": "input/original-{name}",
```

If the blob name is *original-Blob1.txt*, the value of the `name` variable in function code is `Blob1.txt`.

Filter on file type

The following example triggers only on `.png` files:

```
JSON
```

```
"path": "samples/{name}.png",
```

Filter on curly braces in file names

To look for curly braces in file names, escape the braces by using two braces. The following example filters for blobs that have curly braces in the name:

```
JSON
```

```
"path": "images/{{20140101}}-{name}",
```

If the blob is named `{20140101}-soundfile.mp3`, the `name` variable value in the function code is `soundfile.mp3`.

Polling and latency

Polling works as a hybrid between inspecting logs and running periodic container scans. Blobs are scanned in groups of 10,000 at a time with a continuation token used between intervals. If your function app is on the Consumption plan, there can be up to a 10-minute delay in processing new blobs if a function app has gone idle.

⚠ Warning

Storage logs are created on a "best effort" basis. There's no guarantee that all events are captured.

Under some conditions, logs may be missed.

If you require faster or more reliable blob processing, you should consider switching your hosting to use an App Service plan with Always On enabled, which may result in increased costs. You might also consider using a trigger other than the classic polling blob trigger. For more information and a comparison of the various triggering options for blob storage containers, see [Trigger on a blob container](#).

Blob receipts

The Azure Functions runtime ensures that no blob trigger function gets called more than once for the same new or updated blob. To determine if a given blob version has been processed, it maintains *blob receipts*.

Azure Functions stores blob receipts in a container named `azure-webjobs-hosts` in the Azure storage account for your function app (defined by the app setting `AzureWebJobsStorage`). A blob receipt has the following information:

- The triggered function (`<FUNCTION_APP_NAME>.Functions.<FUNCTION_NAME>`, for example:
`MyFunctionApp.Functions.CopyBlob`)
- The container name
- The blob type (`BlockBlob` or `PageBlob`)
- The blob name
- The ETag (a blob version identifier, for example: `0x8D1DC6E70A277EF`)

To force reprocessing of a blob, delete the blob receipt for that blob from the `azure-webjobs-hosts` container manually. While reprocessing might not occur immediately, it's guaranteed to occur at a later point in time. To reprocess immediately, the `scaninfo` blob in `azure-webjobs-hosts/blobscaninfo` can be updated. Any blobs with a last modified timestamp after the `LatestScan` property will be scanned again.

Poison blobs

When a blob trigger function fails for a given blob, Azure Functions retries that function a total of five times by default.

If all 5 tries fail, Azure Functions adds a message to a Storage queue named `webjobs-blobtrigger-poison`. The maximum number of retries is configurable. The same `MaxDequeueCount` setting is used for poison blob handling and poison queue message handling. The queue message for poison blobs is a JSON object that contains the following properties:

- `FunctionId` (in the format `<FUNCTION_APP_NAME>.Functions.<FUNCTION_NAME>`)
- `BlobType` (`BlockBlob` or `PageBlob`)
- `ContainerName`
- `BlobName`
- `ETag` (a blob version identifier, for example: `0x8D1DC6E70A277EF`)

Memory usage and concurrency

When you bind to an [output type](#) that doesn't support steaming, such as `string`, or `Byte[]`, the runtime must load the entire blob into memory more than one time during processing. This can result in higher-than expected memory usage when processing blobs. When possible, use a stream-supporting type. Type support depends on the C# mode and extension version. For more information, see [Binding types](#).

Memory usage can be further impacted when multiple function instances are concurrently processing blob data. If you are having memory issues using a Blob trigger, consider reducing the number of concurrent executions permitted. Of course, reducing the concurrency can have the side effect of increasing the backlog of blobs waiting to be processed. The memory limits of your function app depends on the plan. For more information, see [Service limits](#).

The way that you can control the number of concurrent executions depends on the version of the Storage extension you are using.

Extension 5.x and higher

When using version 5.0.0 of the Storage extension or a later version, you control trigger concurrency by using the `maxDegreeOfParallelism` setting in the [blobs configuration in host.json](#).

Limits apply separately to each function that uses a blob trigger.

host.json properties

The `host.json` file contains settings that control blob trigger behavior. See the [host.json settings](#) section for details regarding available settings.

Next steps

- [Read blob storage data when a function runs](#)
- [Write blob storage data from a function](#)

Azure Blob storage input binding for Azure Functions

Article • 05/12/2024

The input binding allows you to read blob storage data as input to an Azure Function.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated process

The following example is a [C# function](#) that runs in an isolated worker process and uses a blob trigger with both blob input and blob output blob bindings. The function is triggered by the creation of a blob in the *test-samples-trigger* container. It reads a text file from the *test-samples-input* container and creates a new text file in an output container based on the name of the triggered file.

C#

```
public static class BlobFunction
{
    [Function(nameof(BlobFunction))]
    [BlobOutput("test-samples-output/{name}-output.txt")]
    public static string Run(
        [BlobTrigger("test-samples-trigger/{name}")] string myTriggerItem,
        [BlobInput("test-samples-input/sample1.txt")] string myBlob,
        FunctionContext context)
    {
        var logger = context.GetLogger("BlobFunction");
        logger.LogInformation("Triggered Item = {myTriggerItem}", myTriggerItem);
        logger.LogInformation("Input Item = {myBlob}", myBlob);

        // Blob Output
        return "blob-output content";
    }
}
```

```
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated process

Isolated worker process defines an input binding by using a `BlobInputAttribute` attribute, which takes the following parameters:

[Expand table](#)

Parameter	Description
<code>BlobPath</code>	The path to the blob.
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to Azure Blobs. See Connections .

When you're developing locally, add your application settings in the `local.settings.json` file in the `Values` collection.

See the [Example section](#) for complete examples.

Usage

The binding types supported by Blob input depend on the extension package version and the C# modality used in your function app.

Isolated process

When you want the function to process a single blob, the blob input binding can bind to the following types:

[Expand table](#)

Type	Description
<code>string</code>	The blob content as a string. Use when the blob content is simple text.
<code>byte[]</code>	The bytes of the blob content.
JSON serializable types	When a blob contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
<code>Stream</code> ¹	An input stream of the blob content.
<code>BlobClient</code> ¹ , <code>BlockBlobClient</code> ¹ , <code>PageBlobClient</code> ¹ ,	A client connected to the blob. This set of types offers the most control for processing the blob and can be used to write back to it if the connection has sufficient permission.

Type	Description
<code>AppendBlobClient¹,</code> <code>BlobBaseClient¹</code>	

When you want the function to process multiple blobs from a container, the blob input binding can bind to the following types:

[Expand table](#)

Type	Description
<code>T[]</code> or <code>List<T></code> where <code>T</code> is one of the single blob input binding types	An array or list of multiple blobs. Each entry represents one blob from the container. You can also bind to any interfaces implemented by these types, such as <code>IEnumerable<T></code> .
<code>BlobContainerClient¹</code>	A client connected to the container. This type offers the most control for processing the container and can be used to write to it if the connection has sufficient permission.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs 6.0.0 or later](#) and the [common dependencies for SDK type bindings](#).

Binding to `string`, or `Byte[]` is only recommended when the blob size is small. This is recommended because the entire blob contents are loaded into memory. For most blobs, use a `Stream` or `BlobClient` type. For more information, see [Concurrency and memory usage](#).

If you get an error message when trying to bind to one of the Storage SDK types, make sure that you have a reference to [the correct Storage SDK version](#).

You can also use the [StorageAccountAttribute](#) to specify the storage account to use. You can do this when you need to use a different storage account than other functions in the library. The constructor takes the name of an app setting that contains a storage connection string. The attribute can be applied at the parameter, method, or class level. The following example shows class level and method level:

```
C#
[StorageAccount("ClassLevelStorageAppSetting")]
public static class AzureFunctions
{
    [FunctionName("BlobTrigger")]
    [StorageAccount("FunctionLevelStorageAppSetting")]
    public static void Run( //...
    {
        ....
    }
}
```

The storage account to use is determined in the following order:

- The `BlobTrigger` attribute's `Connection` property.
- The `StorageAccount` attribute applied to the same parameter as the `BlobTrigger` attribute.
- The `StorageAccount` attribute applied to the function.
- The `StorageAccount` attribute applied to the class.
- The default storage account for the function app, which is defined in the `AzureWebJobsStorage` application setting.

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to Azure Blobs. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Manage storage account access keys](#). The connection string must be for a general-purpose storage account, not a [Blob storage account](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage". If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [version 5.x or higher of the extension](#) (bundle 3.x or higher for non-.NET language stacks), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

[+] Expand table

Property	Environment variable template	Description	Example value
Blob Service URI	<code><CONNECTION_NAME_PREFIX>__serviceUri</code> ¹	The data plane URI of the blob service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>

¹ `<CONNECTION_NAME_PREFIX>__blobServiceUri` can be used as an alias. If the connection configuration will be used by a blob trigger, `blobServiceUri` must also be accompanied by `queueServiceUri`. See below.

The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables. The URI can only designate the blob service. As an alternative, you can provide a URI specifically for each service, allowing a single connection to be used. If both versions are provided, the multi-service form is used. To configure the connection for multiple services, instead of

`<CONNECTION_NAME_PREFIX>_serviceUri`, set:

[+] Expand table

Property	Environment variable template	Description	Example value
Blob Service URI	<code><CONNECTION_NAME_PREFIX>_blobServiceUri</code>	The data plane URI of the blob service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>
Queue Service URI <small>(required for blob triggers²)</small>	<code><CONNECTION_NAME_PREFIX>_queueServiceUri</code>	The data plane URI of a queue service, using the HTTPS scheme. This value is only needed for blob triggers.	<code>https://<storage_account_name>.queue.core.windows.net</code>

² The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue. In the `serviceUri` form, the `AzureWebJobsStorage` connection is used. However, when specifying `blobServiceUri`, a queue service URI must also be provided with `queueServiceUri`. It's recommended that you use the service from the same storage account as the blob service. You also need to make sure the trigger can read and write messages in the configured queue service by assigning a role like [Storage Queue Data Contributor](#).

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You need to create a role assignment that provides access to your blob container at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Blob Storage extension in normal operation. Your application may require further permissions based on the code you write.

 Expand table

Binding type	Example built-in roles
Trigger	Storage Blob Data Owner and Storage Queue Data Contributor ¹ Extra permissions must also be granted to the AzureWebJobsStorage connection. ²
Input binding	Storage Blob Data Reader
Output binding	Storage Blob Data Owner

¹ The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue on the storage account specified by the connection.

² The AzureWebJobsStorage connection is used internally for blobs and queues that enable the trigger. If it's configured to use an identity-based connection, it needs extra permissions beyond the default requirement. The required permissions are covered by the [Storage Blob Data Owner](#), [Storage Queue Data Contributor](#), and [Storage Account Contributor](#) roles. To learn more, see [Connecting to host storage with an identity](#).

Next steps

- [Run a function when blob storage data changes](#)
- [Write blob storage data from a function](#)

Azure Blob storage output binding for Azure Functions

Article • 09/27/2023

The output binding allows you to modify and delete blob storage data in an Azure Function.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

The following example is a [C# function](#) that runs in an isolated worker process and uses a blob trigger with both blob input and blob output blob bindings. The function is triggered by the creation of a blob in the *test-samples-trigger* container. It reads a text file from the *test-samples-input* container and creates a new text file in an output container based on the name of the triggered file.

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace SampleApp
{
    public static class BlobFunction
    {
        [Function(nameof(BlobFunction))]
        [BlobOutput("test-samples-output/{name}-output.txt")]
        public static string Run(
            [BlobTrigger("test-samples-trigger/{name}")] string myTriggerItem,
            [BlobInput("test-samples-input/sample1.txt")] string myBlob,
            FunctionContext context)
        {
            var logger = context.GetLogger("BlobFunction");
            logger.LogInformation("Triggered Item = {myTriggerItem}", myTriggerItem);
            logger.LogInformation("Input Item = {myBlob}", myBlob);

            // Blob Output
            return "blob-output content";
        }
    }
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated worker model

The `BlobOutputAttribute` constructor takes the following parameters:

Parameter	Description
<code>BlobPath</code>	The path to the blob.
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to Azure Blobs. See Connections .

When you're developing locally, add your application settings in the `local.settings.json` file in the `values` collection.

See the [Example section](#) for complete examples.

Usage

The binding types supported by blob output depend on the extension package version and the C# modality used in your function app.

Isolated worker model

When you want the function to write to a single blob, the blob output binding can bind to the following types:

Type	Description
<code>string</code>	The blob content as a string. Use when the blob content is simple text.
<code>byte[]</code>	The bytes of the blob content.
<code>JSON serializable types</code>	An object representing the content of a JSON blob. Functions attempts to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write to multiple blobs, the blob output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is one of the single blob output binding types	An array containing content for multiple blobs. Each entry represents the content of one blob.

For other output scenarios, create and use types from [Azure.Storage.Blobs](#) directly.

Binding to `string`, or `Byte[]` is only recommended when the blob size is small. This is recommended because the entire blob contents are loaded into memory. For most blobs, use a `Stream` or `BlobClient` type. For more information, see [Concurrency and memory usage](#).

If you get an error message when trying to bind to one of the Storage SDK types, make sure that you have a reference to [the correct Storage SDK version](#).

You can also use the [StorageAccountAttribute](#) to specify the storage account to use. You can do this when you need to use a different storage account than other functions in the library. The constructor takes the name of an app setting that contains a storage connection string. The attribute can be applied at the parameter, method, or class level. The following example shows class level and method level:

C#

```
[StorageAccount("ClassLevelStorageAppSetting")]
public static class AzureFunctions
{
    [FunctionName("BlobTrigger")]
    [StorageAccount("FunctionLevelStorageAppSetting")]
    public static void Run( //...
    {
        ....
    }
}
```

The storage account to use is determined in the following order:

- The `BlobTrigger` attribute's `connection` property.
- The `StorageAccount` attribute applied to the same parameter as the `BlobTrigger` attribute.
- The `StorageAccount` attribute applied to the function.
- The `StorageAccount` attribute applied to the class.
- The default storage account for the function app, which is defined in the `AzureWebJobsStorage` application setting.

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to Azure Blobs. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Manage storage account access keys](#). The connection string must be for a general-purpose storage account, not a [Blob storage account](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage". If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [version 5.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Blob Service URI	<code><CONNECTION_NAME_PREFIX>__serviceUri</code> ¹	The data plane URI of the blob service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>

¹ `<CONNECTION_NAME_PREFIX>__blobServiceUri` can be used as an alias. If the connection configuration will be used by a blob trigger, `blobServiceUri` must also be accompanied by `queueServiceUri`. See below.

The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables. The URI can only designate the blob service. As an alternative, you can provide a URI specifically for each service, allowing a single connection to be used. If both versions are provided, the multi-service form is used. To configure the connection for multiple services, instead of

`<CONNECTION_NAME_PREFIX>__serviceUri`, set:

Property	Environment variable template	Description	Example value
Blob Service URI	<code><CONNECTION_NAME_PREFIX>__blobServiceUri</code>	The data plane URI of the blob service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>
Queue Service URI (required for blob triggers ²)	<code><CONNECTION_NAME_PREFIX>__queueServiceUri</code>	The data plane URI of a queue service, using the HTTPS scheme. This value is only needed for blob triggers.	<code>https://<storage_account_name>.queue.core.windows.net</code>

² The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue. In the `serviceUri` form, the `AzureWebJobsStorage` connection is used. However, when specifying `blobServiceUri`, a queue service URI must also be provided with `queueServiceUri`. It's recommended that you use the service from the same storage account as the blob service. You also need to make sure the trigger can read and write messages in the configured queue service by assigning a role like [Storage Queue Data Contributor](#).

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the [principle of least privilege](#), granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You need to create a role assignment that provides access to your blob container at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Blob Storage extension in normal operation. Your application may require further permissions based on the code you write.

Binding type	Example built-in roles
Trigger	Storage Blob Data Owner and Storage Queue Data Contributor ¹ Extra permissions must also be granted to the <code>AzureWebJobsStorage</code> connection. ²
Input binding	Storage Blob Data Reader
Output binding	Storage Blob Data Owner

¹ The blob trigger handles failure across multiple retries by writing [poison blobs](#) to a queue on the storage account specified by the connection.

² The `AzureWebJobsStorage` connection is used internally for blobs and queues that enable the trigger. If it's configured to use an identity-based connection, it needs extra permissions beyond the default requirement. The required permissions are covered by the [Storage Blob Data Owner](#), [Storage Queue Data Contributor](#), and [Storage Account Contributor](#) roles. To learn more, see [Connecting to host storage with an identity](#).

Exceptions and return codes

Binding	Reference
Blob	Blob Error Codes
Blob, Table, Queue	Storage Error Codes
Blob, Table, Queue	Troubleshooting

Next steps

- Run a function when blob storage data changes
- Read blob storage data when a function runs

Dapr Extension for Azure Functions

Article • 05/10/2024

The Dapr Extension for Azure Functions is a set of tools and services that allow developers to easily integrate Azure Functions with the [Distributed Application Runtime \(Dapr\)](#) platform.

Azure Functions is an event-driven compute service that provides a set of [triggers and bindings](#) to easily connect with other Azure services. Dapr provides a set of building blocks and best practices for building distributed applications, including microservices, state management, pub/sub messaging, and more.

With the integration between Dapr and Functions, you can build functions that react to events from Dapr or external systems.

[+] [Expand table](#)

Action	Direction	Type
Trigger on a Dapr input binding	N/A	daprBindingTrigger
Trigger on a Dapr service invocation	N/A	daprServiceInvocationTrigger
Trigger on a Dapr topic subscription	N/A	daprTopicTrigger
Pull in Dapr state for an execution	In	daprState
Pull in Dapr secrets for an execution	In	daprSecret
Save a value to a Dapr state	Out	daprState
Invoke another Dapr app	Out	daprInvoke
Publish a message to a Dapr topic	Out	daprPublish
Send a value to a Dapr output binding	Out	daprBinding

Install extension

The extension NuGet package you install depends on the C# mode [in-process](#) or [isolated worker process](#) you're using in your function app:

In-process

This extension is available by installing the [NuGet package](#), version 1.0.0.

Using the .NET CLI:

.NET CLI

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Dapr
```

Dapr enablement

You can configure Dapr using various [arguments and annotations][dapr-args] based on the runtime context. You can configure Dapr for Azure Functions through two channels:

- Infrastructure as Code (IaC) templates, as in Bicep or Azure Resource Manager (ARM) templates
- The Azure portal

When using an IaC template, specify the following arguments in the `properties` section of the container app resource definition.

Bicep

Bicep

```
DaprConfig: {  
    enabled: true  
    appId: '${envResourceNamePrefix}-funcapp'  
    appPort: 3001  
    httpReadBufferSize: ''  
    httpMaxRequestSize: ''  
    logLevel: ''  
    enableApiLogging: true  
}
```

The above Dapr configuration values are considered application-scope changes. When you run a container app in multiple-revision mode, changes to these settings won't create a new revision. Instead, all existing revisions are restarted to ensure they're configured with the most up-to-date values.

When configuring Dapr using the Azure portal, navigate to your function app and select **Dapr** from the left-side menu:

Dapr settings

Dapr * Enabled Disabled

App Id *

App port
Please note while using Dapr Extension for Azure Functions, app port is mandatory when using Dapr triggers and should be empty when using Dapr bindings. [Learn more](#)

HTTP read buffer size KB

HTTP max request size MB

Log level

API logging

Components

In order for your app to use a specific Dapr component, it must be added to your Container Apps Environment with the proper scopes setup to give your app access to it. [Click here to manage your Dapr components.](#)

Name ↑	Type ↑
--------	--------

Dapr ports and listeners

When you're triggering a function from Dapr, the extension exposes port `3001` automatically to listen to incoming requests from the Dapr sidecar.

ⓘ Important

Port `3001` is only exposed and listened to if a Dapr trigger is defined in the function app. When using Dapr, the sidecar waits to receive a response from the defined port before completing instantiation. *Do not* define the `dapr.io/port` annotation or `--app-port` unless you have a trigger. Doing so may lock your application from the Dapr sidecar.

If you're only using input and output bindings, port `3001` doesn't need to be exposed or defined.

By default, when Azure Functions tries to communicate with Dapr, it calls Dapr over the port resolved from the environment variable `DAPR_HTTP_PORT`. If that variable is null, it defaults to port `3500`.

You can override the Dapr address used by input and output bindings by setting the `DaprAddress` property in the `function.json` for the binding (or the attribute). By default, it uses `http://localhost:{DAPR_HTTP_PORT}`.

The function app still exposes another port and endpoint for things like HTTP triggers, which locally defaults to `7071`, but in a container, defaults to `80`.

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

In-process class library

An in-process class library is a compiled C# function runs in the same process as the Functions runtime.

The Dapr Extension supports parameter types according to the table below.

[+] Expand table

Binding	Parameter types
Dapr trigger	<code>daprBindingTrigger</code> ↗ <code>daprServiceInvocationTrigger</code> ↗ <code>daprTopicTrigger</code> ↗
Dapr input	<code>daprState</code> ↗ <code>daprSecret</code> ↗
Dapr output	<code>daprState</code> ↗ <code>daprInvoke</code> ↗ <code>daprPublish</code> ↗ <code>daprBinding</code> ↗

For examples using these types, see [the GitHub repository for the extension](#) ↗.

Try out the Dapr Extension for Azure Functions

Learn how to use the Dapr Extension for Azure Functions via the provided samples.

[+] Expand table

Samples	Description
Quickstart ↗	Get started using the Dapr Pub/sub binding and <code>HttpTrigger</code> .
Dapr Kafka ↗	Learn how to use the Azure Functions Dapr Extension with the Kafka bindings Dapr component.
.NET In-process ↗	Learn how to use Azure Functions in-process model to integrate with multiple Dapr components in .NET, like Service Invocation, Pub/sub, Bindings, and State Management.
.NET Isolated ↗	Integrate with Dapr components in .NET using the Azure Functions out-of-proc (OOP) execution model.

Troubleshooting

This section describes how to troubleshoot issues that can occur when using the Dapr extension for Azure Functions.

Ensure Dapr is enabled in your environment

If you're using Dapr bindings and triggers in Azure Functions, and Dapr isn't enabled in your environment, you might receive the error message: `Dapr sidecar isn't present.`

`Please see (https://aka.ms/azure-functions-dapr-sidecar-missing) for more information.` To enable Dapr in your environment:

- If your Azure Function is deployed in Azure Container Apps, refer to [Dapr enablement instructions for the Dapr extension for Azure Functions](#).
- If your Azure Function is deployed in Kubernetes, verify that your [deployment's YAML configuration ↗](#) has the following annotations:

```
YAML

annotations:
  ...
  dapr.io/enabled: "true"
  dapr.io/app-id: "functionapp"
  # You should only set app-port if you are using a Dapr trigger in
  # your code.
  dapr.io/app-port: "<DAPR_APP_PORT>"
  ...
```

- If you're running your Azure Function locally, run the following command to ensure you're [running the function app with Dapr ↗](#):

Bash

```
dapr run --app-id functionapp --app-port <DAPR_APP_PORT> --components-path <COMPONENTS_PATH> -- func host start
```

Verify app-port value in Dapr configuration

The Dapr extension for Azure Functions starts an HTTP server on port `3001` by default. You can configure this port using the [DAPR_APP_PORT environment variable](#).

If you provide an incorrect app port value when running an Azure Functions app, you might receive the error message: `The Dapr sidecar is configured to listen on port {portInt}, but the app server is running on port {appPort}. This may cause unexpected behavior. For more information, visit [this link](https://aka.ms/azfunc-dapr-app-config-error)`. To resolve this error message:

1. In your container app's Dapr settings:

- If you're using a Dapr trigger in your code, verify that the app port is set to `3001` or to the value of the `DAPR_APP_PORT` environment variable.
- If you're *not* using a Dapr trigger in your code, verify that the app port is *not* set. It should be empty.

2. Verify that you provide the correct app port value in the Dapr configuration.

- If you're using Azure Container Apps, specify the app port in Bicep:

Bash

```
DaprConfig: {  
    ...  
    appPort: <DAPR_APP_PORT>  
    ...  
}
```

- If you're using a Kubernetes environment, set the `dapr.io/app-port` annotation:

```
annotations:  
    ...
```

```
dapr.io/app-port: "<DAPR_APP_PORT>"  
...
```

- If you're developing locally, verify you set `--app-port` when running the function app with Dapr:

```
dapr run --app-id functionapp --app-port <DAPR_APP_PORT> --  
components-path <COMPONENTS_PATH> -- func host start
```

Next steps

[Learn more about Dapr.](#) ↗

Dapr Input Bindings trigger for Azure Functions

Article • 05/21/2024

Azure Functions can be triggered on a Dapr input binding using the following Dapr events.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] [Expand table](#)

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

C#

```
[FunctionName("ConsumeMessageFromKafka")]
public static void Run(
    // Note: the value of BindingName must match the binding name in
    components/kafka-bindings.yaml
    [DaprBindingTrigger(BindingName = "%KafkaBindingName%")] JObject
    triggerData,
    ILogger log)
{
    log.LogInformation("Hello from Kafka!");
    log.LogInformation($"Trigger data: {triggerData}");
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprBindingTrigger` to trigger a Dapr input binding, which supports the following properties.

 [Expand table](#)

Parameter	Description
<code>BindingName</code>	The name of the Dapr trigger. If not specified, the name of the function is used as the trigger name.

See the [Example section](#) for complete examples.

Usage

To use the Dapr Input Binding trigger, start by setting up a Dapr input binding component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr input binding component specs ↗](#)
- [How to: Trigger your application with input bindings ↗](#)

Next steps

[Learn more about Dapr service invocation. ↗](#)

Dapr Service Invocation trigger for Azure Functions

Article • 05/10/2024

Azure Functions can be triggered on a Dapr service invocation using the following Dapr events.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

C#

```
[FunctionName("CreateNewOrder")]
public static void Run(
    [DaprServiceInvocationTrigger] JObject payload,
    [DaprState("%StateStoreName%", Key = "order")] out JToken order,
    ILogger log)
{
    log.LogInformation("C# function processed a CreateNewOrder request
from the Dapr Runtime.");

    // payload must be of the format { "data": { "value": "some value" }
}
```

```
    order = payload["data"];
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprServiceInvocationTrigger` to trigger a Dapr service invocation binding, which supports the following properties.

[\[+\] Expand table](#)

Parameter	Description
<code>MethodName</code>	<i>Optional.</i> The name of the method the Dapr caller should use. If not specified, the name of the function is used as the method name.

See the [Example section](#) for complete examples.

Usage

To use a Dapr Service Invocation trigger, learn more about which components to use with the Service Invocation trigger and how to set them up in the official Dapr documentation.

- [Dapr component specs ↗](#)
- [Dapr service invocation ↗](#)

Next steps

[Learn more about Dapr service invocation. ↗](#)

Dapr Topic trigger for Azure Functions

Article • 05/21/2024

Azure Functions can be triggered on a Dapr topic subscription using the following Dapr events.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

C#

```
[FunctionName("TransferEventBetweenTopics")]
public static void Run(
    [DaprTopicTrigger("%PubSubName%", Topic = "A")] CloudEvent subEvent,
    [DaprPublish(PubSubName = "%PubSubName%", Topic = "B")] out
DaprPubSubEvent pubEvent,
    ILogger log)
{
    log.LogInformation("C# function processed a
TransferEventBetweenTopics request from the Dapr Runtime.");

    pubEvent = new DaprPubSubEvent("Transfer from Topic A: " +
subEvent.Data);
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprTopicTrigger` to trigger a Dapr pub/sub binding, which supports the following properties.

 [Expand table](#)

Parameter	Description
<code>PubSubName</code>	The name of the Dapr pub/sub.
<code>Topic</code>	The name of the Dapr topic.

See the [Example section](#) for complete examples.

Usage

To use a Dapr Topic trigger, start by setting up a Dapr pub/sub component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr pub/sub component specs ↗](#)
- [How to: Publish a message and subscribe to a topic ↗](#)

Next steps

[Learn more about Dapr publish and subscribe. ↗](#)

Dapr State input binding for Azure Functions

Article • 05/21/2024

The Dapr state input binding allows you to read Dapr state during a function execution.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

C#

```
[FunctionName("StateInputBinding")]
public static IActionResult Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", Route =
"state/{key}")] HttpRequest req,
    [DaprState("statestore", Key = "{key}")] string state,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    return new OkObjectResult(state);
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprState` to read Dapr state into your function, which supports these parameters:

 [Expand table](#)

Parameter	Description
<code>StateStore</code>	The name of the state store to retrieve state.
<code>Key</code>	The name of the key to retrieve from the specified state store.

See the [Example section](#) for complete examples.

Usage

To use the Dapr state input binding, start by setting up a Dapr state store component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr state store component specs ↗](#)
- [How to: Save state ↗](#)

Next steps

[Learn more about Dapr state management. ↗](#)

Dapr Secret input binding for Azure Functions

Article • 05/21/2024

The Dapr secret input binding allows you to read secrets data as input during function execution.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] Expand table

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

C#

```
[FunctionName("RetrieveSecret")]
public static void Run(
    [DaprServiceInvocationTrigger] object args,
    [DaprSecret("kubernetes", "my-secret", Metadata =
"metadata.namespace=default")] IDictionary<string, string> secret,
    ILogger log)
{
    log.LogInformation("C# function processed a RetrieveSecret request
from the Dapr Runtime.");
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprSecret` to define a Dapr secret input binding, which supports these parameters:

[Expand table](#)

Parameter	Description
<code>SecretStoreName</code>	The name of the secret store to get the secret.
<code>Key</code>	The key identifying the name of the secret to get.
<code>Metadata</code>	<i>Optional.</i> An array of metadata properties in the form <code>"key1=value1&key2=value2"</code> .

See the [Example section](#) for complete examples.

Usage

To use the Dapr secret input binding, start by setting up a Dapr secret store component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr secret store component specs ↗](#)
- [How to: Retrieve a secret ↗](#)

Next steps

[Learn more about Dapr secrets. ↗](#)

Dapr State output binding for Azure Functions

Article • 05/10/2024

The Dapr state output binding allows you to save a value to a Dapr state during a function execution.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] [Expand table](#)

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

The following example demonstrates using the Dapr state output binding to persist a new state into the state store.

C#

```
[FunctionName("StateOutputBinding")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"state/{key}")] HttpRequest req,
    [DaprState("statestore", Key = "{key}")] IAsyncCollector<string>
state,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");
```

```

        string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
await state.AddAsync(requestBody);

return new OkResult();
}

```

Attributes

In-process

In the [in-process model](#), use the `DaprState` to define a Dapr state output binding, which supports these parameters:

[Expand table](#)

Parameter	Description	Can be sent via Attribute	Can be sent via RequestBody
<code>StateStore</code>	The name of the state store to save state.	✓	✗
<code>Key</code>	The name of the key to save state within the state store.	✓	✓
<code>Value</code>	<i>Required.</i> The value being stored.	✗	✓

If properties are defined in both Attributes and `RequestBody`, priority is given to data provided in `RequestBody`.

See the [Example section](#) for complete examples.

Usage

To use the Dapr state output binding, start by setting up a Dapr state store component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr state store component specs ↗](#)
- [How to: Save state ↗](#)

Next steps

[Learn more about Dapr state management.](#) ↗

Dapr Invoke output binding for Azure Functions

Article • 05/10/2024

The Dapr invoke output binding allows you to invoke another Dapr application during a function execution.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] [Expand table](#)

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

The following example demonstrates using a Dapr invoke output binding to perform a Dapr service invocation operation hosted in another Dapr-ized application. In this example, the function acts like a proxy.

C#

```
[FunctionName("InvokeOutputBinding")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", Route =
"invoke/{appId}/{methodName}")] HttpRequest req,
    [DaprInvoke(AppId = "{appId}", MethodName = "{methodName}", HttpVerb
= "post")] IAsyncCollector<InvokeMethodParameters> output,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");
}
```

```

    string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();

    var outputContent = new InvokeMethodParameters
    {
        Body = requestBody
    };

    await output.AddAsync(outputContent);

    return new OkResult();
}

```

Attributes

In-process

In the [in-process model](#), use the `DaprInvoke` attribute to define a Dapr invoke output binding, which supports these parameters:

[Expand table](#)

Parameter	Description	Can be sent via Attribute	Can be sent via RequestBody
<code>AppId</code>	The Dapr app ID to invoke.	✓	✓
<code>MethodName</code>	The method name of the app to invoke.	✓	✓
<code>HttpVerb</code>	<i>Optional.</i> HTTP verb to use of the app to invoke. Default is <code>POST</code> .	✓	✓
<code>Body</code>	<i>Required.</i> The body of the request.	✗	✓

If properties are defined in both Attributes and `RequestBody`, priority is given to data provided in `RequestBody`.

See the [Example section](#) for complete examples.

Usage

To use the Dapr service invocation output binding, learn more about [how to use Dapr service invocation](#) in the official Dapr documentation ↗.

Next steps

[Learn more about Dapr service invocation.](#) ↗

Dapr Publish output binding for Azure Functions

Article • 05/10/2024

The Dapr publish output binding allows you to publish a message to a Dapr topic during a function execution.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] [Expand table](#)

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

The following example demonstrates using a Dapr publish output binding to perform a Dapr publish operation to a pub/sub component and topic.

C#

```
[FunctionName("PublishOutputBinding")]
public static void Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"topic/{topicName}")] HttpRequest req,
    [DaprPublish(PubSubName = "%PubSubName%", Topic = "{topicName}")]
    out DaprPubSubEvent pubSubEvent,
    ILogger log)
{
    string requestBody = new StreamReader(req.Body).ReadToEnd();
```

```
    pubSubEvent = new DaprPubSubEvent(requestBody);  
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprPublish` to define a Dapr publish output binding, which supports these parameters:

[Expand table](#)

function.json property	Description	Can be sent via Attribute	Can be sent via RequestBody
PubSubName	The name of the Dapr pub/sub to send the message.	✓	✓
Topic	The name of the Dapr topic to send the message.	✓	✓
Payload	<i>Required.</i> The message being published.	✗	✓

If properties are defined in both Attributes and `RequestBody`, priority is given to data provided in `RequestBody`.

See the [Example section](#) for complete examples.

Usage

To use the Dapr publish output binding, start by setting up a Dapr pub/sub component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr pub/sub component specs ↗](#)
- [How to: Publish a message and subscribe to a topic ↗](#)

Next steps

[Learn more about Dapr publish and subscribe.](#) ↗

Dapr Binding output binding for Azure Functions

Article • 05/10/2024

The Dapr output binding allows you to send a value to a Dapr output binding during a function execution.

For information on setup and configuration details of the Dapr extension, see the [Dapr extension overview](#).

Example

A C# function can be created using one of the following C# modes:

[+] [Expand table](#)

Execution model	Description
Isolated worker model	Your function code runs in a separate .NET worker process. Use with supported versions of .NET and .NET Framework . To learn more, see Develop .NET isolated worker process functions .
In-process model	Your function code runs in the same process as the Functions host process. Supports only Long Term Support (LTS) versions of .NET . To learn more, see Develop .NET class library functions .

In-process

The following example demonstrates using a Dapr service invocation trigger and a Dapr output binding to read and process a binding request.

C#

```
[FunctionName("SendMessageToKafka")]
public static async Task Run(
    [DaprServiceInvocationTrigger] JObject payload,
    [DaprBinding(BindingName = "%KafkaBindingName%", Operation =
"create")] IAsyncCollector<object> messages,
    ILogger log)
{
    log.LogInformation("C# function processed a SendMessageToKafka
request.");
```

```
    await messages.AddAsync(payload);
}
```

Attributes

In-process

In the [in-process model](#), use the `DaprBinding` to define a Dapr binding output binding, which supports these parameters:

[Expand table](#)

Parameter	Description	Can be sent via Attribute	Can be sent via RequestBody
BindingName	The name of the Dapr binding.	✓	✓
Operation	The configured binding operation.	✓	✓
Metadata	The metadata namespace.	✗	✓
Data	<i>Required.</i> The data for the binding operation.	✗	✓

If properties are defined in both Attributes and `RequestBody`, priority is given to data provided in `RequestBody`.

See the [Example section](#) for complete examples.

Usage

To use the Dapr output binding, start by setting up a Dapr output binding component. You can learn more about which component to use and how to set it up in the official Dapr documentation.

- [Dapr output binding component specs ↗](#)
- [How to: Use output bindings to interface with external resources ↗](#)

Next steps

[Learn more about Dapr service invocation.](#) ↗

Azure Event Grid bindings for Azure Functions

Article • 07/11/2023

This reference shows how to connect to Azure Event Grid using Azure Functions triggers and bindings.

Event Grid is an Azure service that sends HTTP requests to notify you about events that happen in publishers. A *publisher* is the service or resource that originates the event. For example, an Azure blob storage account is a publisher, and a [blob upload or deletion is an event](#). Some [Azure services have built-in support for publishing events to Event Grid](#).

Event *handlers* receive and process events. Azure Functions is one of several [Azure services that have built-in support for handling Event Grid events](#). Functions provides an Event Grid trigger, which invokes a function when an event is received from Event Grid. A similar output binding can be used to send events from your function to an [Event Grid custom topic](#).

You can also use an HTTP trigger to handle Event Grid Events. To learn more, see [Receive events to an HTTP endpoint](#). We recommend using the Event Grid trigger over HTTP trigger.

Action	Type
Run a function when an Event Grid event is dispatched	Trigger
Sends an Event Grid event	Output binding
Control the returned HTTP status code	HTTP endpoint

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

In-process

Functions execute in the same process as the Functions host. To learn more, see [Develop C# class library functions using Azure Functions](#).

In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. To update existing binding extensions for C# script apps running in the portal without having to republish your function app, see [Update your extensions](#).

The functionality of the extension varies depending on the extension version:

Extension v3.x

This section describes using a [class library](#). For [C# scripting](#), you would need to instead [install the extension bundle](#), version 3.x.

This version of the extension supports updated Event Grid binding parameter types of [Azure.Messaging.CloudEvent](#) and [Azure.Messaging.EventGrid.EventGridEvent](#).

Add this version of the extension to your project by installing the [NuGet package](#), version 3.x.

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

In-process

An in-process class library is a compiled C# function runs in the same process as the Functions runtime.

Choose a version to see binding type details for the mode and version.

Extension v3.x

The Event Grid extension supports parameter types according to the table below.

Binding	Parameter types
Event Grid trigger	CloudEvent EventGridEvent BinaryData Newtonsoft.Json.Linq JObject string

Binding	Parameter types
Event Grid output (single event)	<code>CloudEvent</code> <code>EventGridEvent</code> <code>BinaryData</code> <code>Newtonsoft.Json.Linq JObject</code> ↗ <code>string</code>
Event Grid output (multiple events)	<code>ICollector<T></code> or <code>IAsyncCollector<T></code> where <code>T</code> is one of the single event types

Next steps

- If you have questions, submit an issue to the team [here](#) ↗
- [Event Grid trigger](#)
- [Event Grid output binding](#)
- [Run a function when an Event Grid event is dispatched](#)
- [Dispatch an Event Grid event](#)

Azure Event Grid trigger for Azure Functions

Article • 09/11/2023

Use the function trigger to respond to an event sent by an [Event Grid source](#). You must have an event subscription to the source to receive events. To learn how to create an event subscription, see [Create a subscription](#). For information on binding setup and configuration, see the [overview](#).

ⓘ Note

Event Grid triggers aren't natively supported in an internal load balancer App Service Environment (ASE). The trigger uses an HTTP request that can't reach the function app without a gateway into the virtual network.

Example

For an HTTP trigger example, see [Receive events to an HTTP endpoint](#).

The type of the input parameter used with an Event Grid trigger depends on these three factors:

- Functions runtime version
- Binding extension version
- Modality of the C# function.

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

When running your C# function in an isolated worker process, you need to define a custom type for event properties. The following example defines a `MyEventType` class.

C#

```
public class MyEventType
{
    public string Id { get; set; }

    public string Topic { get; set; }

    public string Subject { get; set; }

    public string EventType { get; set; }

    public DateTime EventTime { get; set; }

    public IDictionary<string, object> Data { get; set; }
}
```

The following example shows how the custom type is used in both the trigger and an Event Grid output binding:

C#

```
public static class EventGridFunction
{
    [Function(nameof(EventGridFunction))]
    [EventGridOutput(TopicEndpointUri = "MyEventGridTopicUriSetting",
    TopicKeySetting = "MyEventGridTopicKeySetting")]
    public static MyEventType Run([EventGridTrigger] MyEventType input,
    FunctionContext context)
    {
        var logger = context.GetLogger(nameof(EventGridFunction));

        logger.LogInformation(input.Data.ToString());

        var outputEvent = new MyEventType()
        {
            Id = "unique-id",
            Subject = "abc-subject",
            Data = new Dictionary<string, object>
            {
                { "myKey", "myValue" }
            }
        };

        return outputEvent;
    }
}
```

```
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the [EventGridTrigger](#) attribute. C# script instead uses a function.json configuration file as described in the [C# scripting guide](#).

Isolated worker model

Here's an `EventGridTrigger` attribute in a method signature:

C#

```
[Function(nameof(EventGridFunction))]
[EventGridOutput(TopicEndpointUri = "MyEventGridTopicUriSetting",
TopicKeySetting = "MyEventGridTopicKeySetting")]
public static MyEventType Run([EventGridTrigger] MyEventType input,
FunctionContext context)
{
```

See the [Example section](#) for complete examples.

Usage

The Event Grid trigger uses a webhook HTTP request, which can be configured using the same [host.json settings as the HTTP Trigger](#).

The parameter type supported by the Event Grid trigger depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension v3.x

When you want the function to process a single event, the Event Grid trigger can bind to the following types:

Type	Description
JSON serializable types	Functions tries to deserialize the JSON data of the event into a plain-old CLR object (POCO) type.

Type	Description
<code>string</code>	The event as a string.
<code>BinaryData</code> ¹	The bytes of the event message.
<code>CloudEvent</code> ¹	The event object. Use when Event Grid is configured to deliver using the CloudEvents schema.
<code>EventGridEvent</code> ¹	The event object. Use when Event Grid is configured to deliver using the Event Grid schema.

When you want the function to process a batch of events, the Event Grid trigger can bind to the following types:

Type	Description
<code>CloudEvent[]</code> ¹ ,	An array of events from the batch. Each entry represents one event.
<code>EventGridEvent[]</code> ¹ ,	
<code>string[]</code> ,	
<code>BinaryData[]</code> ¹	

¹ To use these types, you need to reference

[Microsoft.Azure.Functions.Worker.Extensions.EventGrid 3.3.0 or later](#) and the common dependencies for SDK type bindings.

Event schema

Data for an Event Grid event is received as a JSON object in the body of an HTTP request. The JSON looks similar to the following example:

JSON

```
[{
  "topic": "/subscriptions/{subscriptionid}/resourceGroups/eg0122/providers/Microsoft.Storage/storageAccounts/egblobstore",
  "subject": "/blobServices/default/containers/{containername}/blobs/blobname.jpg",
  "eventType": "Microsoft.Storage.BlobCreated",
  "eventTime": "2018-01-23T17:02:19.6069787Z",
  "id": "{guid}",
  "data": {
    "api": "PutBlockList",
    "clientRequestId": "{guid}",
    "requestId": "{guid}",
    "eTag": "0x8D562831044DD00",
    "blobSize": 123456789
  }
}]
```

```
"contentType": "application/octet-stream",
"contentLength": 2248,
"blobType": "BlockBlob",
"url":
"https://egblobstore.blob.core.windows.net/{containername}/blobname.jpg",
"sequencer": "00000000000272D000000000003D60F",
"storageDiagnostics": {
    "batchId": "{guid}"
},
},
"dataVersion": "",
"metadataVersion": "1"
}]
```

The example shown is an array of one element. Event Grid always sends an array and may send more than one event in the array. The runtime invokes your function once for each array element.

The top-level properties in the event JSON data are the same among all event types, while the contents of the `data` property are specific to each event type. The example shown is for a blob storage event.

For explanations of the common and event-specific properties, see [Event properties](#) in the Event Grid documentation.

Next steps

- If you have questions, submit an issue to the team [here](#) ↗
- Dispatch an Event Grid event

Azure Event Grid output binding for Azure Functions

Article • 09/24/2023

Use the Event Grid output binding to write events to a custom topic. You must have a valid [access key for the custom topic](#). The Event Grid output binding doesn't support shared access signature (SAS) tokens.

For information on setup and configuration details, see [How to work with Event Grid triggers and bindings in Azure Functions](#).

ⓘ Important

The Event Grid output binding is only available for Functions 2.x and higher.

Example

The type of the output parameter used with an Event Grid output binding depends on the Functions runtime version, the binding extension version, and the modality of the C# function. The C# function can be created using one of the following C# modes:

- [In-process class library](#): compiled C# function that runs in the same process as the Functions runtime.
- [Isolated worker process class library](#): compiled C# function that runs in a worker process isolated from the runtime.

Isolated worker model

The following example shows how the custom type is used in both the trigger and an Event Grid output binding:

```
C#  
  
using System;  
using System.Collections.Generic;  
using Microsoft.Azure.Functions.Worker;  
using Microsoft.Extensions.Logging;  
  
namespace SampleApp  
{  
    public static class EventGridFunction  
    {  
        [Function(nameof(EventGridFunction))]
```

```

[EventGridOutput(TopicEndpointUri = "MyEventGridTopicUriSetting",
TopicKeySetting = "MyEventGridTopicKeySetting")]
    public static MyEventType Run([EventGridTrigger] MyEventType
input, FunctionContext context)
{
    var logger = context.GetLogger(nameof(EventGridFunction));

    logger.LogInformation(input.Data.ToString());

    var outputEvent = new MyEventType()
    {
        Id = "unique-id",
        Subject = "abc-subject",
        Data = new Dictionary<string, object>
        {
            { "myKey", "myValue" }
        }
    };

    return outputEvent;
}
}

public class MyEventType
{
    public string Id { get; set; }

    public string Topic { get; set; }

    public string Subject { get; set; }

    public string EventType { get; set; }

    public DateTime EventTime { get; set; }

    public IDictionary<string, object> Data { get; set; }
}
}

```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to configure the binding. C# script instead uses a function.json configuration file as described in the [C# scripting guide](#).

The attribute's constructor takes the name of an application setting that contains the name of the custom topic, and the name of an application setting that contains the topic key.

The following table explains the parameters for the `EventGridOutputAttribute`.

Parameter	Description
<code>TopicEndpointUri</code>	The name of an app setting that contains the URI for the custom topic, such as <code>MyTopicEndpointUri</code> .
<code>TopicKeySetting</code>	The name of an app setting that contains an access key for the custom topic.
<code>connection*</code>	The value of the common prefix for the setting that contains the topic endpoint URI. For more information about the naming format of this application setting, see Identity-based authentication .

*Support for identity-based connections requires version 3.3.x or higher of the extension.

When you're developing locally, add your application settings in the `local.settings.json` file in the `Values` collection.

Important

Make sure that you set the value of `TopicEndpointUri` to the name of an app setting that contains the URI of the custom topic. Don't specify the URI of the custom topic directly in this property. The same applies when using `Connection`.

See the [Example section](#) for complete examples.

Usage

The parameter type supported by the Event Grid output binding depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension v3.x

When you want the function to write a single event, the Event Grid output binding can bind to the following types:

Type	Description
<code>string</code>	The event as a string.
<code>byte[]</code>	The bytes of the event message.

Type	Description
JSON serializable types	An object representing a JSON event. Functions tries to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write multiple events, the Event Grid output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is one of the single event types	An array containing multiple events. Each entry represents one event.

For other output scenarios, create and use types from [Azure.Messaging.EventGrid](#) directly.

Connections

There are two ways of authenticating to an Event Grid topic when using the Event Grid output binding:

Authentication method	Description
Using a topic key	Set the <code>TopicEndpointUri</code> and <code>TopicKeySetting</code> properties, as described in Use a topic key .
Using an identity	Set the <code>Connection</code> property to the name of a shared prefix for multiple application settings, together defining identity-based authentication . This method is supported when using version 3.3.x or higher of the extension.

Use a topic key

Use the following steps to configure a topic key:

1. Follow the steps in [Get access keys](#) to obtain the topic key for your Event Grid topic.
2. In your application settings, create a setting that defines the topic key value. Use the name of this setting for the `TopicKeySetting` property of the binding.
3. In your application settings, create a setting that defines the topic endpoint. Use the name of this setting for the `TopicEndpointUri` property of the binding.

Identity-based authentication

When using version 3.3.x or higher of the extension, you can connect to an Event Grid topic using an [Azure Active Directory identity](#) to avoid having to obtain and work with topic keys.

To do this, create an application setting that returns the topic endpoint URI, where the name of the setting combines a unique *common prefix*, such as `myawesometopic`, with the value `_topicEndpointUri`. You then use the common prefix `myawesometopic` when you define the `connection` property in the binding.

In this mode, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Topic Endpoint URI	<code><CONNECTION_NAME_PREFIX>_topicEndpointUri</code>	The topic endpoint.	<code>https://<topic-name>.centralus-1.eventgrid.azure.net/api/events</code>

More properties may be set to customize the connection. See [Common properties for identity-based connections](#).

Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for managed identity-based connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `<CONNECTION_NAME_PREFIX>:topicEndpointUri`.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You must create a role assignment that provides access to your Event Grid topic at runtime. Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Event Hubs extension in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles
Output binding	EventGrid Contributor , EventGrid Data Sender

Next steps

- [Dispatch an Event Grid event](#)

Azure Event Hubs trigger and bindings for Azure Functions

Article • 03/08/2022

This article explains how to work with [Azure Event Hubs](#) bindings for Azure Functions. Azure Functions supports trigger and output bindings for Event Hubs.

Action	Type
Respond to events sent to an event hub event stream.	Trigger
Write events to an event stream	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

In-process

Functions execute in the same process as the Functions host. To learn more, see [Develop C# class library functions using Azure Functions](#).

The functionality of the extension varies depending on the extension version:

Extension v5.x+

This version introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

This version uses the newer Event Hubs binding type [Azure.Messaging.EventHubs.EventData](#).

This extension version is available by installing the [NuGet package](#), version 5.x.

host.json settings

The `host.json` file contains settings that control behavior for the Event Hubs trigger. The configuration is different depending on the extension version.

Extension v5.x+

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "eventHubs": {  
            "maxEventBatchSize" : 100,  
            "minEventBatchSize" : 25,  
            "maxWaitTime" : "00:05:00",  
            "batchCheckpointFrequency" : 1,  
            "prefetchCount" : 300,  
            "transportType" : "amqpWebSockets",  
            "webProxy" : "https://proxyserver:8080",  
            "customEndpointAddress" : "amqps://company.gateway.local",  
            "targetUnprocessedEventThreshold" : 75,  
            "initialOffsetOptions" : {  
                "type" : "fromStart",  
                "enqueuedTimeUtc" : ""  
            },  
            "clientRetryOptions":{  
                "mode" : "exponential",  
                "tryTimeout" : "00:01:00",  
                "delay" : "00:00:00.80",  
                "maximumDelay" : "00:01:00",  
                "maximumRetries" : 3  
            }  
        }  
    }  
}
```

Property	Default	Description
maxEventBatchSize	10	The maximum number of events included in a batch for a single invocation. Must be at least 1.

Property	Default	Description
minEventBatchSize ¹	1	<p>The minimum number of events desired in a batch. The minimum applies only when the function is receiving multiple events and must be less than <code>maxEventBatchSize</code>. The minimum size isn't strictly guaranteed. A partial batch is dispatched when a full batch can't be prepared before the <code>maxWaitTime</code> has elapsed. Partial batches are also likely for the first invocation of the function after scaling takes place.</p>
maxWaitTime ¹	00:01:00	<p>The maximum interval that the trigger should wait to fill a batch before invoking the function. The wait time is only considered when <code>minEventBatchSize</code> is larger than 1 and is otherwise ignored. If less than <code>minEventBatchSize</code> events were available before the wait time elapses, the function is invoked with a partial batch. The longest allowed wait time is 10 minutes.</p> <p>NOTE: This interval is not a strict guarantee for the exact timing on which the function is invoked. There is a small margin of error due to timer precision. When scaling takes place, the first invocation with a partial batch may occur more quickly or may take up to twice the configured wait time.</p>
batchCheckpointFrequency	1	<p>The number of batches to process before creating a checkpoint for the event hub.</p>
prefetchCount	300	<p>The number of events that is eagerly requested from Event Hubs and held in a local cache to allow reads to avoid waiting on a network operation</p>

Property	Default	Description
transportType	amqpTcp	The protocol and transport that is used for communicating with Event Hubs. Available options: <code>amqpTcp</code> , <code>amqpWebSockets</code>
webProxy	null	The proxy to use for communicating with Event Hubs over web sockets. A proxy cannot be used with the <code>amqpTcp</code> transport.
customEndpointAddress	null	The address to use when establishing a connection to Event Hubs, allowing network requests to be routed through an application gateway or other path needed for the host environment. The fully qualified namespace for the event hub is still needed when a custom endpoint address is used and must be specified explicitly or via the connection string.
targetUnprocessedEventThreshold ¹	null	The desired number of unprocessed events per function instance. The threshold is used in target-based scaling to override the default scaling threshold inferred from the <code>maxEventBatchSize</code> option. When set, the total unprocessed event count is divided by this value to determine the number of function instances needed. The instance count will be rounded up to a number that creates a balanced partition distribution.
initialOffsetOptions/type	fromStart	The location in the event stream to start processing when a checkpoint does not exist in storage. Applies to all partitions. For more information, see the OffsetType documentation . Available options: <code>fromStart</code> , <code>fromEnd</code> , <code>fromEnqueuedTime</code>

Property	Default	Description
initialOffsetOptions/enqueuedTimeUtc	null	Specifies the enqueued time of the event in the stream from which to start processing. When <code>initialOffsetOptions/type</code> is configured as <code>fromEnqueuedTime</code> , this setting is mandatory. Supports time in any format supported by <code>DateTime.Parse()</code> , such as <code>2020-10-26T20:31Z</code> . For clarity, you should also specify a timezone. When timezone isn't specified, Functions assumes the local timezone of the machine running the function app, which is UTC when running on Azure.
clientRetryOptions mode	exponential	The approach to use for calculating retry delays. Exponential mode retries attempts with a delay based on a back-off strategy where each attempt will increase the duration that it waits before retrying. The fixed mode retries attempts at fixed intervals with each delay having a consistent duration. Available options: <code>exponential</code> , <code>fixed</code>
clientRetryOptions tryTimeout	00:01:00	The maximum duration to wait for an Event Hubs operation to complete, per attempt.
clientRetryOptions delay	00:00:00.80	The delay or back-off factor to apply between retry attempts.
clientRetryOptions maximumDelay	00:00:01	The maximum delay to allow between retry attempts.
clientRetryOptions maximumRetries	3	The maximum number of retry attempts before considering the associated operation to have failed.

¹ Using `minEventBatchSize` and `maxWaitTime` requires [v5.3.0](#) of the `Microsoft.Azure.WebJobs.Extensions.EventHubs` package, or a later version.

The `clientRetryOptions` are used to retry operations between the Functions host and Event Hubs (such as fetching events and sending events). Refer to guidance on

[Azure Functions error handling and retries](#) for information on applying retry policies to individual functions.

For a reference of host.json in Azure Functions 2.x and beyond, see [host.json reference for Azure Functions](#).

Next steps

- [Respond to events sent to an event hub event stream \(Trigger\)](#)
- [Write events to an event stream \(Output binding\)](#)

Azure Event Hubs trigger for Azure Functions

Article • 09/11/2023

This article explains how to work with [Azure Event Hubs](#) trigger for Azure Functions. Azure Functions supports trigger and [output bindings](#) for Event Hubs.

For information on setup and configuration details, see the [overview](#).

Use the function trigger to respond to an event sent to an event hub event stream. You must have read access to the underlying event hub to set up the trigger. When the function is triggered, the message passed to the function is typed as a string.

Event Hubs scaling decisions for the Consumption and Premium plans are done via Target Based Scaling. For more information, see [Target Based Scaling](#).

For information about how Azure Functions responds to events sent to an event hub event stream using triggers, see [Integrate Event Hubs with serverless functions on Azure](#).

Example

Isolated worker model

The following example shows a [C# function](#) that is triggered based on an event hub, where the input message string is written to the logs:

```
C#  
  
{  
    private readonly ILogger<EventHubsFunction> _logger;  
  
    public EventHubsFunction(ILogger<EventHubsFunction> logger)  
    {  
        _logger = logger;  
    }  
  
    [Function(nameof(EventHubFunction))]  
    [FixedDelayRetry(5, "00:00:10")]  
    [EventHubOutput("dest", Connection = "EventHubConnection")]  
    public string EventHubFunction(  
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to configure the trigger. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated worker model

Use the `EventHubTriggerAttribute` to define a trigger on an event hub, which supports the following properties.

Parameters	Description
<code>EventHubName</code>	The name of the event hub. When the event hub name is also present in the connection string, that value overrides this property at runtime. Can be referenced in app settings , like <code>%eventHubName%</code>
<code>ConsumerGroup</code>	An optional property that sets the consumer group used to subscribe to events in the hub. When omitted, the <code>\$Default</code> consumer group is used.
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to Event Hubs. To learn more, see Connections .

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

Usage

To learn more about how Event Hubs trigger and IoT Hub trigger scales, see [Consuming Events with Azure Functions](#).

The parameter type supported by the Event Hubs output binding depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension v5.x+

When you want the function to process a single event, the Event Hubs trigger can bind to the following types:

Type	Description
<code>string</code>	The event as a string. Use when the event is simple text.
<code>byte[]</code>	The bytes of the event.
JSON serializable types	When an event contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
<code>Azure.Messaging.EventHubs.EventData</code> ¹	The event object. If you are migrating from any older versions of the Event Hubs SDKs, note that this version drops support for the legacy <code>Body</code> type in favor of EventBody .

When you want the function to process a batch of events, the Event Hubs trigger can bind to the following types:

Type	Description
<code>string[]</code>	An array of events from the batch, as strings. Each entry represents one event.

Type	Description
<code>EventData[]</code> ¹	An array of events from the batch, as instances of <code>Azure.Messaging.EventHubs.EventData</code> . Each entry represents one event.
<code>T[]</code> where <code>T</code> is a JSON serializable type ¹	An array of events from the batch, as instances of a custom POCO type. Each entry represents one event.

¹ To use these types, you need to reference

[Microsoft.Azure.Functions.Worker.Extensions.EventHubs 5.5.0 or later](#) and the [common dependencies for SDK type bindings](#).

Event metadata

The Event Hubs trigger provides several [metadata properties](#). Metadata properties can be used as part of binding expressions in other bindings or as parameters in your code. The properties come from the [EventData](#) class.

Property	Type	Description
<code>PartitionContext</code>	PartitionContext	The <code>PartitionContext</code> instance.
<code>EnqueuedTimeUtc</code>	DateTime	The enqueued time in UTC.
<code>Offset</code>	<code>string</code>	The offset of the data relative to the event hub partition stream. The offset is a marker or identifier for an event within the Event Hubs stream. The identifier is unique within a partition of the Event Hubs stream.
<code>PartitionKey</code>	<code>string</code>	The partition to which event data should be sent.
<code>Properties</code>	<code>IDictionary<String, Object></code>	The user properties of the event data.
<code>SequenceNumber</code>	<code>Int64</code>	The logical sequence number of the event.
<code>SystemProperties</code>	<code>IDictionary<String, Object></code>	The system properties, including the event data.

See [code examples](#) that use these properties earlier in this article.

Connections

The `connection` property is a reference to environment configuration which specifies how the app should connect to Event Hubs. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

Obtain this connection string by clicking the **Connection Information** button for the [namespace](#), not the event hub itself. The connection string must be for an Event Hubs namespace, not the event hub itself.

When used for triggers, the connection string must have at least "read" permissions to activate the function. When used for output bindings, the connection string must have "send" permissions to send messages to the event stream.

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

Identity-based connections

If you are using [version 5.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Azure Active Directory identity](#). To do this, you would define settings under a common prefix which maps to the `connection` property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Fully Qualified Namespace	<code><CONNECTION_NAME_PREFIX>_fullyQualifiedNamespace</code>	The fully qualified Event Hubs namespace.	<code>myeventhubs.servicebus.windows.net</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

⚠ Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `<CONNECTION_NAME_PREFIX>:fullyQualifiedNamespace`.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You will need to create a role assignment that provides access to your event hub at runtime. The scope of the role assignment can be for an Event Hubs namespace, or the event hub itself. Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Event Hubs extension in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles
Trigger	Azure Event Hubs Data Receiver , Azure Event Hubs Data Owner
Output binding	Azure Event Hubs Data Sender

host.json settings

The [host.json](#) file contains settings that control Event Hubs trigger behavior. See the [host.json settings](#) section for details regarding available settings.

Next steps

- [Write events to an event stream \(Output binding\)](#)

Azure Event Hubs output binding for Azure Functions

Article • 04/26/2024

This article explains how to work with [Azure Event Hubs](#) bindings for Azure Functions. Azure Functions supports trigger and output bindings for Event Hubs.

For information on setup and configuration details, see the [overview](#).

Use the Event Hubs output binding to write events to an event stream. You must have send permission to an event hub to write events to it.

Make sure the required package references are in place before you try to implement an output binding.

Example

Isolated worker model

The following example shows a [C# function](#) that writes a message string to an event hub, using the method return value as the output:

```
C#  
  
[Function(nameof(EventHubFunction))]  
[FixedDelayRetry(5, "00:00:10")]  
[EventHubOutput("dest", Connection = "EventHubConnection")]  
public string EventHubFunction(  
    [EventHubTrigger("src", Connection = "EventHubConnection")] string[] input,  
    FunctionContext context)  
{  
    _logger.LogInformation("First Event Hubs triggered message: {msg}", input[0]);  
  
    var message = $"Output message created at {DateTime.Now}";  
    return message;  
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to configure the binding. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated worker model

Use the `[EventHubOutputAttribute]` to define an output binding to an event hub, which supports the following properties.

[Expand table](#)

Parameters	Description
EventHubName	The name of the event hub. When the event hub name is also present in the connection string, that value overrides this property at runtime.
Connection	The name of an app setting or setting collection that specifies how to connect to Event Hubs. To learn more, see Connections .

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

Usage

The parameter type supported by the Event Hubs output binding depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension v5.x+

When you want the function to write a single event, the Event Hubs output binding can bind to the following types:

[Expand table](#)

Type	Description
<code>string</code>	The event as a string. Use when the event is simple text.
<code>byte[]</code>	The bytes of the event.
JSON serializable types	An object representing the event. Functions tries to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write multiple events, the Event Hubs output binding can bind to the following types:

[Expand table](#)

Type	Description
<code>T[]</code> where <code>T</code> is one of the single event types	An array containing multiple events. Each entry represents one event.

For other output scenarios, create and use types from [Microsoft.Azure.EventHubs](#) directly.

Connections

The `connection` property is a reference to environment configuration which specifies how the app should connect to Event Hubs. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

Obtain this connection string by clicking the **Connection Information** button for the [namespace](#), not the event hub itself. The connection string must be for an Event Hubs namespace, not the event hub itself.

When used for triggers, the connection string must have at least "read" permissions to activate the function. When used for output bindings, the connection string must have "send" permissions to send messages to the event stream.

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

Identity-based connections

If you are using [version 5.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To do this, you would define settings under a common prefix which maps to the `connection` property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

 Expand table

Property	Environment variable template	Description	Example value
Fully Qualified Namespace	<code><CONNECTION_NAME_PREFIX>_fullyQualifiedNamespace</code>	The fully qualified Event Hubs namespace.	<code>myeventhubns.servicebus.windows.net</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, <CONNECTION_NAME_PREFIX>:fullyQualifiedNamespace.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the [principle of least privilege](#), granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You will need to create a role assignment that provides access to your event hub at runtime. The scope of the role assignment can be for an Event Hubs namespace, or the event hub itself. Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Event Hubs extension in normal operation. Your application may require additional permissions based on the code you write.

 Expand table

Binding type	Example built-in roles
Trigger	Azure Event Hubs Data Receiver , Azure Event Hubs Data Owner
Output binding	Azure Event Hubs Data Sender

Exceptions and return codes

 Expand table

Binding	Reference
Event Hubs	Operations Guide

Next steps

- Respond to events sent to an event hub event stream (Trigger)

Azure Functions HTTP triggers and bindings overview

Article • 03/31/2024

Azure Functions may be invoked via HTTP requests to build serverless APIs and respond to [webhooks](#).

[+] Expand table

Action	Type
Run a function from an HTTP request	Trigger
Return an HTTP response from a function	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

The functionality of the extension varies depending on the extension version:

Functions v2.x+

Add the extension to your project by installing the [NuGet package](#), version 3.x.

ⓘ Note

An additional extension package is needed for [ASP.NET Core integration in .NET Isolated](#)

host.json settings

This section describes the configuration settings available for this binding in versions 2.x and higher. Settings in the host.json file apply to all functions in a function app instance. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

ⓘ Note

For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

JSON

```
{  
  "extensions": {  
    "http": {  
      "routePrefix": "api",  
      "maxOutstandingRequests": 200,  
      "maxConcurrentRequests": 100,  
      "dynamicThrottlesEnabled": true,  
      "hsts": {  
        "isEnabled": true,  
        "maxAge": "10"  
      },  
      "customHeaders": {  
        "X-Content-Type-Options": "nosniff"  
      }  
    }  
  }  
}
```

[+] Expand table

Property	Default	Description
customHeaders	none	Allows you to set custom headers in the HTTP response. The previous example adds the <code>X-Content-Type-Options</code> header to the response to avoid content type sniffing. This custom header applies to all HTTP triggered functions in the function app.
dynamicThrottlesEnabled	true*	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like <code>connections/threads/processes/memory/cpu/etc</code> and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a <code>429 "Too Busy"</code> response until the counter(s) return to normal levels.

Property	Default	Description
		*The default in a Consumption plan is <code>true</code> . The default in a Dedicated plan is <code>false</code> .
hsts	not enabled	When <code>isEnabled</code> is set to <code>true</code> , the HTTP Strict Transport Security (HSTS) behavior of .NET Core is enforced, as defined in the HstsOptions class . The above example also sets the <code>maxAge</code> property to 10 days. Supported properties of <code>hsts</code> are:
[+] Expand table		
Property	Description	
excludedHosts	A string array of host names for which the HSTS header isn't added.	
includeSubDomains	Boolean value that indicates whether the <code>includeSubDomain</code> parameter of the Strict-Transport-Security header is enabled.	
maxAge	String that defines the <code>max-age</code> parameter of the Strict-Transport-Security header.	
preload	Boolean that indicates whether the <code>preload</code> parameter of the Strict-Transport-Security header is enabled.	
maxConcurrentRequests	100*	The maximum number of HTTP functions that are executed in parallel. This value allows you to control concurrency, which can help manage resource utilization. For example, you might have an HTTP function that uses a large number of system resources (memory/cpu/sockets) such that it causes issues when concurrency is too high. Or you might have a function that makes outbound requests to a third-party service, and those calls need to be rate limited. In these cases, applying a throttle here can help. *The default for a Consumption plan is 100. The default for a Dedicated plan is unbounded (-1).
maxOutstandingRequests	200*	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows

Property	Default	Description
		<p>callers to employ time-based retry strategies, and also helps you to control maximum request latencies. This only controls queuing that occurs within the script host execution path. Other queues such as the ASP.NET request queue will still be in effect and unaffected by this setting.</p> <p>*The default for a Consumption plan is 200. The default for a Dedicated plan is unbounded (-1).</p>
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.

Next steps

- [Run a function from an HTTP request](#)
- [Return an HTTP response from a function](#)

Azure Functions HTTP trigger

Article • 07/18/2024

The HTTP trigger lets you invoke a function with an HTTP request. You can use an HTTP trigger to build serverless APIs and respond to webhooks.

The default return value for an HTTP-triggered function is:

- `HTTP 204 No Content` with an empty body in Functions 2.x and higher
- `HTTP 200 OK` with an empty body in Functions 1.x

To modify the HTTP response, configure an [output binding](#).

For more information about HTTP bindings, see the [overview](#) and [output binding reference](#).

Tip

If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`. For more information, see [How to manage connections in Azure Functions](#).

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full

support.

The code in this article defaults to .NET Core syntax, used in Functions version 2.x and higher. For information on the 1.x syntax, see the [1.x functions templates](#).

Isolated worker model

The following example shows an HTTP trigger that returns a "hello, world" response as an [IActionResult](#), using [ASP.NET Core integration in .NET Isolated](#):

C#

```
[Function("HttpFunction")]
public IActionResult Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get")] HttpRequest req)
{
    return new OkObjectResult($"Welcome to Azure Functions,
{req.Query["name"]}!");
}
```

The following example shows an HTTP trigger that returns a "hello world" response as an [HttpResponseData](#) object:

C#

```
[Function(nameof(HttpFunction))]
public static HttpResponseData
Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route =
null)] HttpRequestData req,
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger(nameof(HttpFunction));
    logger.LogInformation("message logged");

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-8");
    response.WriteString("Welcome to .NET isolated worker !!");

    return response;
}
```

Attributes

Both the [isolated worker model](#) and the [in-process model](#) use the [HttpTriggerAttribute](#) to define the trigger binding. C# script instead uses a function.json configuration file as

described in the [C# scripting guide](#).

Isolated worker model

In [isolated worker model](#) function apps, the `HttpTriggerAttribute` supports the following parameters:

[Expand table](#)

Parameters	Description
<code>AuthLevel</code>	Determines what keys, if any, need to be present on the request in order to invoke the function. For supported values, see Authorization level .
<code>Methods</code>	An array of the HTTP methods to which the function responds. If not specified, the function responds to all HTTP methods. See customize the HTTP endpoint .
<code>Route</code>	Defines the route template, controlling to which request URLs your function responds. The default value if none is provided is <code><functionname></code> . For more information, see customize the HTTP endpoint .

Usage

This section details how to configure your HTTP trigger function binding.

Payload

Isolated worker model

The trigger input type is declared as one of the following types:

[Expand table](#)

Type	Description
<code>HttpRequest</code>	<i>Use of this type requires that the app is configured with ASP.NET Core integration in .NET Isolated.</i> This gives you full access to the request object and overall <code>HttpContext</code> .
<code>HttpRequestData</code>	A projection of the request object.
A custom type	When the body of the request is JSON, the runtime tries to parse it to set the object properties.

When the trigger parameter is of type `HttpRequestData` or `HttpRequest`, custom types can also be bound to other parameters using `Microsoft.Azure.Functions.Worker.Http.FromBodyAttribute`. Use of this attribute requires [Microsoft.Azure.Functions.Worker.Extensions.Http version 3.1.0 or later](#). This is a different type than the similar attribute in `Microsoft.AspNetCore.Mvc`. When using ASP.NET Core integration, you need a fully qualified reference or `using` statement. This example shows how to use the attribute to get just the body contents while still having access to the full `HttpRequest`, using ASP.NET Core integration:

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Functions.Worker;
using FromBodyAttribute =
    Microsoft.Azure.Functions.Worker.Http.FromBodyAttribute;

namespace AspNetIntegration
{
    public class BodyBindingHttpTrigger
    {
        [Function(nameof(BodyBindingHttpTrigger))]
        public IActionResult
        Run([HttpTrigger(AuthorizationLevel.Anonymous, "post")]
            [FromBody] Person person)
        {
            return new OkObjectResult(person);
        }

        public record Person(string Name, int Age);
    }
}
```

Customize the HTTP endpoint

By default when you create a function for an HTTP trigger, the function is addressable with a route of the form:

HTTP

`https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>`

You can customize this route using the optional `route` property on the HTTP trigger's input binding. You can use any [Web API Route Constraint](#) with your parameters.

Isolated worker model

The following function code accepts two parameters `category` and `id` in the route and writes a response using both parameters.

C#

```
[Function("HttpTrigger1")]
public static HttpResponseMessage
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post",
Route = "products/{category:alpha}/{id:int?}")] HttpRequestData req,
string category, int? id,
FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpTrigger1");
    logger.LogInformation("C# HTTP trigger function processed a
request.");

    var message = String.Format($"Category: {category}, ID: {id}");
    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-8");
    response.WriteString(message);

    return response;
}
```

Using this configuration, the function is now addressable with the following route instead of the original route.

```
https://<APP_NAME>.azurewebsites.net/api/products/electronics/357
```

This configuration allows the function code to support two parameters in the address, `category` and `ID`. For more information on how route parameters are tokenized in a URL, see [Routing in ASP.NET Core](#).

By default, all function routes are prefixed with `api`. You can also customize or remove the prefix using the `extensions.http.routePrefix` property in your `host.json` file. The following example removes the `api` route prefix by using an empty string for the prefix in the `host.json` file.

JSON

```
{
  "extensions": {
```

```
        "http": {
            "routePrefix": ""
        }
    }
}
```

Using route parameters

Route parameters that defined a function's `route` pattern are available to each binding. For example, if you have a route defined as `"route": "products/{id}"` then a table storage binding can use the value of the `{id}` parameter in the binding configuration.

When you use route parameters, an `invoke_URL_template` is automatically created for your function. Your clients can use the URL template to understand the parameters they need to pass in the URL when calling your function using its URL. Navigate to one of your HTTP-triggered functions in the [Azure portal](#) and select **Get function URL**.

You can programmatically access the `invoke_URL_template` by using the Azure Resource Manager APIs for [List Functions](#) or [Get Function](#).

Working with client identities

If your function app is using [App Service Authentication / Authorization](#), you can view information about authenticated clients from your code. This information is available as [request headers injected by the platform](#).

You can also read this information from binding data.

ⓘ Note

Access to authenticated client information is currently only available for .NET languages. It also isn't supported in version 1.x of the Functions runtime.

Information regarding authenticated clients is available as a [ClaimsPrincipal](#), which is available as part of the request context as shown in the following example:

Isolated worker model

The authenticated user is available via [HTTP Headers](#).

Authorization level

The authorization level is a string value that indicates the kind of [authorization key](#) that's required to access the function endpoint. For an HTTP triggered function, the authorization level can be one of the following values:

[+] [Expand table](#)

Level value	Description
anonymous	No access key is required.
function	A function-specific key is required to access the endpoint.
admin	The master key is required to access the endpoint.

When a level isn't explicitly set, authorization defaults to the `function` level.

Function access keys

Functions lets you use access keys to make it harder to access your function endpoints. Unless the authorization level on an HTTP triggered function is set to `anonymous`, requests must include an access key in the request. For more information, see [Work with access keys in Azure Functions](#).

Access key authorization

Most HTTP trigger templates require an access key in the request. So your HTTP request normally looks like the following URL:

```
HTTP  
https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>?code=<API_KEY>
```

The key can be included in a query string variable named `code`, as mentioned earlier. It can also be included in an `x-functions-key` HTTP header. The value of the key can be any function key defined for the function, or any host key.

You can allow anonymous requests, which don't require keys. You can also require that the master key is used. You change the default authorization level by using the `authLevel` property in the binding JSON.

ⓘ Note

When running functions locally, authorization is disabled regardless of the specified authorization level setting. After publishing to Azure, the `authLevel` setting in your trigger is enforced. Keys are still required when running [locally in a container](#).

Webhooks

ⓘ Note

Webhook mode is only available for version 1.x of the Functions runtime. This change was made to improve the performance of HTTP triggers in version 2.x and higher.

In version 1.x, webhook templates provide another validation for webhook payloads. In version 2.x and higher, the base HTTP trigger still works and is the recommended approach for webhooks.

WebHook type

The `webHookType` binding property indicates the type of webhook supported by the function, which also dictates the supported payload. The webhook type can be one of the following values:

[+] [Expand table](#)

Type value	Description
<code>genericJson</code>	A general-purpose webhook endpoint without logic for a specific provider. This setting restricts requests to only those using HTTP POST and with the <code>application/json</code> content type.
<code>github</code>	The function responds to GitHub webhooks . Don't use the <code>authLevel</code> property with GitHub webhooks.
<code>slack</code>	The function responds to Slack webhooks . Don't use the <code>authLevel</code> property with Slack webhooks.

When setting the `webHookType` property, don't also set the `methods` property on the binding.

GitHub webhooks

To respond to GitHub webhooks, first create your function with an HTTP Trigger, and set the `webHookType` property to `github`. Then copy its URL and API key into the [Add webhook](#) page of your GitHub repository.

The screenshot shows the 'Add webhook' configuration page. It includes a note about sending POST requests to the specified URL with event details. The 'Payload URL' field contains 'https://example.com/postreceive'. The 'Content type' dropdown is set to 'application/json'. The 'Secret' field is empty. All three input fields are highlighted with a red border.

Webhooks / Add webhook

We'll send a `POST` request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, `x-www-form-urlencoded`, etc). More information can be found in [our developer documentation](#).

Payload URL *

https://example.com/postreceive

Content type

application/json

Secret

Slack webhooks

The Slack webhook generates a token for you instead of letting you specify it, so you must configure a function-specific key with the token from Slack. See [Authorization keys](#).

Webhooks and keys

Webhook authorization is handled by the webhook receiver component, part of the HTTP trigger, and the mechanism varies based on the webhook type. Each mechanism does rely on a key. By default, the function key named "default" is used. To use a different key, configure the webhook provider to send the key name with the request in one of the following ways:

- **Query string:** The provider passes the key name in the `clientid` query string parameter, such as `https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>?clientid=<KEY_NAME>`.
- **Request header:** The provider passes the key name in the `x-functions-clientid` header.

Content types

Passing binary and form data to a non-C# function requires that you use the appropriate content-type header. Supported content types include `octet-stream` for binary data and [multipart types](#).

Known issues

In non-C# functions, requests sent with the content-type `image/jpeg` results in a `string` value passed to the function. In cases like these, you can manually convert the `string` value to a byte array to access the raw binary data.

Limits

The HTTP request length is limited to 100 MB (104,857,600 bytes), and the URL length is limited to 4 KB (4,096 bytes). These limits are specified by the `httpRuntime` element of the runtime's [Web.config file](#).

If a function that uses the HTTP trigger doesn't complete within 230 seconds, the [Azure Load Balancer](#) will time out and return an HTTP 502 error. The function will continue running but will be unable to return an HTTP response. For long-running functions, we recommend that you follow async patterns and return a location where you can ping the status of the request. For information about how long a function can run, see [Scale and hosting - Consumption plan](#).

Next steps

- [Return an HTTP response from a function](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Azure Functions HTTP output bindings

Article • 05/22/2024

Use the HTTP output binding to respond to the HTTP request sender (HTTP trigger). This binding requires an HTTP trigger and allows you to customize the response associated with the trigger's request.

The default return value for an HTTP-triggered function is:

- `HTTP 204 No Content` with an empty body in Functions 2.x and higher
- `HTTP 200 OK` with an empty body in Functions 1.x

Attribute

Isolated worker model

A return value attribute isn't required when using `HttpResponseData`. However, when using a [ASP.NET Core integration](#) and [multi-binding output objects](#), the `[HttpResultAttribute]` attribute should be applied to the object property. The attribute takes no parameters. To learn more, see [Usage](#).

Usage

To send an HTTP response, use the language-standard response patterns.

In .NET, the response type depends on the C# mode:

Isolated worker model

The HTTP triggered function returns an object of one of the following types:

- [IActionResult](#)¹ (or `Task<IActionResult>`)
- [HttpResponse](#)¹ (or `Task<HttpResponse>`)
- [HttpResponseData](#) (or `Task<HttpResponseData>`)
- JSON serializable types representing the response body for a `200 OK` response.

¹ This type is only available when using [ASP.NET Core integration](#).

When one of these types is used as part of [multi-binding output objects](#), the `[HttpResponse]` attribute should be applied to the object property. The attribute takes no parameters.

For example responses, see the [trigger examples](#).

Next steps

- [Run a function from an HTTP request](#)

Azure IoT Hub bindings for Azure Functions

Article • 03/08/2022

This set of articles explains how to work with Azure Functions bindings for IoT Hub. The IoT Hub support is based on the [Azure Event Hubs Binding](#).

ⓘ Important

While the following code samples use the Event Hub API, the given syntax is applicable for IoT Hub functions.

Action	Type
Respond to events sent to an IoT hub event stream.	Trigger

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

In-process

Functions execute in the same process as the Functions host. To learn more, see [Develop C# class library functions using Azure Functions](#).

The functionality of the extension varies depending on the extension version:

Extension v5.x+

This version introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

This version uses the newer Event Hubs binding type [Azure.Messaging.EventHubs.EventData](#).

This extension version is available by installing the [NuGet package](#), version 5.x.

host.json settings

The [host.json](#) file contains settings that control behavior for the Event Hubs trigger. The configuration is different depending on the extension version.

Extension v5.x+

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "eventHubs": {  
            "maxEventBatchSize" : 100,  
            "minEventBatchSize" : 25,  
            "maxWaitTime" : "00:05:00",  
            "batchCheckpointFrequency" : 1,  
            "prefetchCount" : 300,  
            "transportType" : "amqpWebSockets",  
            "webProxy" : "https://proxyserver:8080",  
            "customEndpointAddress" : "amqps://company.gateway.local",  
            "targetUnprocessedEventThreshold" : 75,  
            "initialOffsetOptions" : {  
                "type" : "fromStart",  
                "enqueuedTimeUtc" : ""  
            },  
            "clientRetryOptions":{  
                "mode" : "exponential",  
                "tryTimeout" : "00:01:00",  
                "delay" : "00:00:00.80",  
                "maximumDelay" : "00:01:00",  
                "maximumRetries" : 3  
            }  
        }  
    }  
}
```

Property	Default	Description
maxEventBatchSize	10	The maximum number of events included in a batch for a single invocation. Must be at least 1.

Property	Default	Description
minEventBatchSize ¹	1	<p>The minimum number of events desired in a batch. The minimum applies only when the function is receiving multiple events and must be less than <code>maxEventBatchSize</code>. The minimum size isn't strictly guaranteed. A partial batch is dispatched when a full batch can't be prepared before the <code>maxWaitTime</code> has elapsed. Partial batches are also likely for the first invocation of the function after scaling takes place.</p>
maxWaitTime ¹	00:01:00	<p>The maximum interval that the trigger should wait to fill a batch before invoking the function. The wait time is only considered when <code>minEventBatchSize</code> is larger than 1 and is otherwise ignored. If less than <code>minEventBatchSize</code> events were available before the wait time elapses, the function is invoked with a partial batch. The longest allowed wait time is 10 minutes.</p> <p>NOTE: This interval is not a strict guarantee for the exact timing on which the function is invoked. There is a small margin of error due to timer precision. When scaling takes place, the first invocation with a partial batch may occur more quickly or may take up to twice the configured wait time.</p>
batchCheckpointFrequency	1	The number of batches to process before creating a checkpoint for the event hub.
prefetchCount	300	The number of events that is eagerly requested from Event Hubs and held in a local cache to allow reads to avoid waiting on a network operation

Property	Default	Description
transportType	amqpTcp	The protocol and transport that is used for communicating with Event Hubs. Available options: <code>amqpTcp</code> , <code>amqpWebSockets</code>
webProxy	null	The proxy to use for communicating with Event Hubs over web sockets. A proxy cannot be used with the <code>amqpTcp</code> transport.
customEndpointAddress	null	The address to use when establishing a connection to Event Hubs, allowing network requests to be routed through an application gateway or other path needed for the host environment. The fully qualified namespace for the event hub is still needed when a custom endpoint address is used and must be specified explicitly or via the connection string.
targetUnprocessedEventThreshold ¹	null	The desired number of unprocessed events per function instance. The threshold is used in target-based scaling to override the default scaling threshold inferred from the <code>maxEventBatchSize</code> option. When set, the total unprocessed event count is divided by this value to determine the number of function instances needed. The instance count will be rounded up to a number that creates a balanced partition distribution.
initialOffsetOptions/type	fromStart	The location in the event stream to start processing when a checkpoint does not exist in storage. Applies to all partitions. For more information, see the OffsetType documentation . Available options: <code>fromStart</code> , <code>fromEnd</code> , <code>fromEnqueuedTime</code>

Property	Default	Description
initialOffsetOptions/enqueuedTimeUtc	null	Specifies the enqueued time of the event in the stream from which to start processing. When <code>initialOffsetOptions/type</code> is configured as <code>fromEnqueuedTime</code> , this setting is mandatory. Supports time in any format supported by <code>DateTime.Parse()</code> , such as <code>2020-10-26T20:31Z</code> . For clarity, you should also specify a timezone. When timezone isn't specified, Functions assumes the local timezone of the machine running the function app, which is UTC when running on Azure.
clientRetryOptions mode	exponential	The approach to use for calculating retry delays. Exponential mode retries attempts with a delay based on a back-off strategy where each attempt will increase the duration that it waits before retrying. The fixed mode retries attempts at fixed intervals with each delay having a consistent duration. Available options: <code>exponential</code> , <code>fixed</code>
clientRetryOptions tryTimeout	00:01:00	The maximum duration to wait for an Event Hubs operation to complete, per attempt.
clientRetryOptions delay	00:00:00.80	The delay or back-off factor to apply between retry attempts.
clientRetryOptions maximumDelay	00:00:01	The maximum delay to allow between retry attempts.
clientRetryOptions maximumRetries	3	The maximum number of retry attempts before considering the associated operation to have failed.

¹ Using `minEventBatchSize` and `maxWaitTime` requires [v5.3.0](#) of the `Microsoft.Azure.WebJobs.Extensions.EventHubs` package, or a later version.

The `clientRetryOptions` are used to retry operations between the Functions host and Event Hubs (such as fetching events and sending events). Refer to guidance on

[Azure Functions error handling and retries](#) for information on applying retry policies to individual functions.

For a reference of host.json in Azure Functions 2.x and beyond, see [host.json reference for Azure Functions](#).

Next steps

- [Respond to events sent to an event hub event stream \(Trigger\)](#)
- [Write events to an event stream \(Output binding\)](#)

Azure IoT Hub trigger for Azure Functions

Article • 09/11/2023

This article explains how to work with Azure Functions bindings for IoT Hub. The IoT Hub support is based on the [Azure Event Hubs Binding](#).

For information on setup and configuration details, see the [overview](#).

Important

While the following code samples use the Event Hub API, the given syntax is applicable for IoT Hub functions.

Use the function trigger to respond to an event sent to an event hub event stream. You must have read access to the underlying event hub to set up the trigger. When the function is triggered, the message passed to the function is typed as a string.

Event Hubs scaling decisions for the Consumption and Premium plans are done via Target Based Scaling. For more information, see [Target Based Scaling](#).

For information about how Azure Functions responds to events sent to an event hub event stream using triggers, see [Integrate Event Hubs with serverless functions on Azure](#).

Example

Isolated worker model

The following example shows a [C# function](#) that is triggered based on an event hub, where the input message string is written to the logs:

C#

```
{  
    private readonly ILogger<EventHubsFunction> _logger;  
  
    public EventHubsFunction(ILogger<EventHubsFunction> logger)  
    {  
        _logger = logger;  
    }  
  
    [Function(nameof(EventHubFunction))]
```

```
[FixedDelayRetry(5, "00:00:10")]
[EventHubOutput("dest", Connection = "EventHubConnection")]
public string EventHubFunction()
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to configure the trigger. C# script instead uses a function.json configuration file as described in the [C# scripting guide](#).

Isolated worker model

Use the `EventHubTriggerAttribute` to define a trigger on an event hub, which supports the following properties.

Parameters	Description
<code>EventHubName</code>	The name of the event hub. When the event hub name is also present in the connection string, that value overrides this property at runtime. Can be referenced in app settings , like <code>%eventHubName%</code>
<code>ConsumerGroup</code>	An optional property that sets the consumer group used to subscribe to events in the hub. When omitted, the <code>\$Default</code> consumer group is used.
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to Event Hubs. To learn more, see Connections .

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

Usage

To learn more about how Event Hubs trigger and IoT Hub trigger scales, see [Consuming Events with Azure Functions](#).

The parameter type supported by the Event Hubs output binding depends on the Functions runtime version, the extension package version, and the C# modality used.

Extension v5.x+

When you want the function to process a single event, the Event Hubs trigger can bind to the following types:

Type	Description
<code>string</code>	The event as a string. Use when the event is simple text.
<code>byte[]</code>	The bytes of the event.
JSON serializable types	When an event contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
<code>Azure.Messaging.EventHubs.EventData</code> ¹	<p>The event object.</p> <p>If you are migrating from any older versions of the Event Hubs SDKs, note that this version drops support for the legacy <code>Body</code> type in favor of EventBody.</p>

When you want the function to process a batch of events, the Event Hubs trigger can bind to the following types:

Type	Description
<code>string[]</code>	An array of events from the batch, as strings. Each entry represents one event.
<code>EventData[]</code> ¹	An array of events from the batch, as instances of Azure.Messaging.EventHubs.EventData . Each entry represents one event.
<code>T[]</code> where <code>T</code> is a JSON serializable type ¹	An array of events from the batch, as instances of a custom POCO type. Each entry represents one event.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.EventHubs 5.5.0 or later](#) and the common dependencies for SDK type bindings.

Event metadata

The Event Hubs trigger provides several [metadata properties](#). Metadata properties can be used as part of binding expressions in other bindings or as parameters in your code. The properties come from the [EventData](#) class.

Property	Type	Description
<code>PartitionContext</code>	PartitionContext	The <code>PartitionContext</code> instance.

Property	Type	Description
EnqueuedTimeUtc	DateTime	The enqueued time in UTC.
Offset	string	The offset of the data relative to the event hub partition stream. The offset is a marker or identifier for an event within the Event Hubs stream. The identifier is unique within a partition of the Event Hubs stream.
PartitionKey	string	The partition to which event data should be sent.
Properties	IDictionary<String, Object>	The user properties of the event data.
SequenceNumber	Int64	The logical sequence number of the event.
SystemProperties	IDictionary<String, Object>	The system properties, including the event data.

See [code examples](#) that use these properties earlier in this article.

Connections

The `connection` property is a reference to environment configuration that contains name of an application setting containing a connection string. You can get this connection string by selecting the **Connection Information** button for the [namespace](#). The connection string must be for an Event Hubs namespace, not the event hub itself.

The connection string must have at least "read" permissions to activate the function.

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

ⓘ Note

Identity-based connections aren't supported by the IoT Hub trigger. If you need to use managed identities end-to-end, you can instead use IoT Hub Routing to send data to an event hub you control. In that way, outbound routing can be authenticated with managed identity the event can be read **from that event hub using managed identity**.

host.json properties

The [host.json](#) file contains settings that control Event Hub trigger behavior. See the [host.json settings](#) section for details regarding available settings.

Next steps

- [Write events to an event stream \(Output binding\)](#)

Apache Kafka bindings for Azure Functions overview

Article • 03/31/2024

The Kafka extension for Azure Functions lets you write values out to [Apache Kafka](#) topics by using an output binding. You can also use a trigger to invoke your functions in response to messages in Kafka topics.

ⓘ Important

Kafka bindings are only available for Functions on the [Elastic Premium Plan](#) and [Dedicated \(App Service\).plan](#). They are only supported on version 3.x and later version of the Functions runtime.

[+] [Expand table](#)

Action	Type
Run a function based on a new Kafka event.	Trigger
Write to the Kafka event stream.	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing this [NuGet package](#).

Enable runtime scaling

To allow your functions to scale properly on the Premium plan when using Kafka triggers and bindings, you need to enable runtime scale monitoring.

Azure portal

In the Azure portal, in your function app, choose **Configuration** and on the **Function runtime settings** tab turn **Runtime scale monitoring** to **On**.

The screenshot shows the Azure portal's Configuration page for a Function App named "scaling-sample". The "Function runtime settings" tab is active. In this tab, there is a section titled "Runtime Scale Monitor..." with a radio button set to "On". A callout bubble provides instructions: "Enable this setting to allow your app to be".

host.json settings

This section describes the configuration settings available for this binding in versions 3.x and higher. Settings in the host.json file apply to all functions in a function app instance. For more information about function app configuration settings in versions 3.x and later versions, see the [host.json reference for Azure Functions](#).

JSON

```
{  
    "version": "2.0",  
    "extensions": {  
        "kafka": {  
            "maxBatchSize": 64,  
            "SubscriberIntervalInSeconds": 1,  
            "ExecutorChannelCapacity": 1,  
            "ChannelFullRetryIntervalInMs": 50  
        }  
    }  
}
```

[] Expand table

Property	Default	Type	Description
ChannelFullRetryIntervalInMs	50	Trigger	Defines the subscriber retry interval, in milliseconds, used when attempting to add items to an at-capacity channel.
ExecutorChannelCapacity	1	Both	Defines the channel message capacity. Once capacity is reached, the Kafka subscriber pauses until the function catches up.
MaxBatchSize	64	Trigger	Maximum batch size when calling a Kafka triggered function.
SubscriberIntervalInSeconds	1	Trigger	Defines the minimum frequency incoming messages are executed, per function in seconds. Only when the message volume is less than <code>MaxBatchSize</code> / <code>SubscriberIntervalInSeconds</code>

The following properties, which are inherited from the [Apache Kafka C/C++ client library](#), are also supported in the `kafka` section of host.json, for either triggers or both output bindings and triggers:

[] Expand table

Property	Applies to	librdkafka equivalent
AutoCommitIntervalMs	Trigger	<code>auto.commit.interval.ms</code>
AutoOffsetReset	Trigger	<code>auto.offset.reset</code>
FetchMaxBytes	Trigger	<code>fetch.max.bytes</code>
LibkafkaDebug	Both	<code>debug</code>
MaxPartitionFetchBytes	Trigger	<code>max.partition.fetch.bytes</code>
MaxPollIntervalMs	Trigger	<code>max.poll.interval.ms</code>
MetadataMaxAgeMs	Both	<code>metadata.max.age.ms</code>
QueuedMinMessages	Trigger	<code>queued.min.messages</code>
QueuedMaxMessagesKbytes	Trigger	<code>queued.max.messages.kbytes</code>
ReconnectBackoffMs	Trigger	<code>reconnect.backoff.max.ms</code>
ReconnectBackoffMaxMs	Trigger	<code>reconnect.backoff.max.ms</code>
SessionTimeoutMs	Trigger	<code>session.timeout.ms</code>

Property	Applies to	librdkafka equivalent
SocketKeepaliveEnable	Both	<code>socket.keepalive.enable</code>
StatisticsIntervalMs	Trigger	<code>statistics.interval.ms</code>

Next steps

- Run a function from an Apache Kafka event stream

Apache Kafka trigger for Azure Functions

Article • 02/17/2023

You can use the Apache Kafka trigger in Azure Functions to run your function code in response to messages in Kafka topics. You can also use a [Kafka output binding](#) to write from your function to a topic. For information on setup and configuration details, see [Apache Kafka bindings for Azure Functions overview](#).

ⓘ Important

Kafka bindings are only available for Functions on the [Elastic Premium Plan](#) and [Dedicated \(App Service\) plan](#). They are only supported on version 3.x and later version of the Functions runtime.

Example

The usage of the trigger depends on the C# modality used in your function app, which can be one of the following modes:

In-process

An [in-process class library](#) is a compiled C# function runs in the same process as the Functions runtime.

The attributes you use depend on the specific event provider.

Confluent (in-process)

The following example shows a C# function that reads and logs the Kafka message as a Kafka event:

C#

```
[FunctionName("KafkaTrigger")]
public static void Run(
    [KafkaTrigger("BrokerList",
                  "topic",
                  Username = "ConfluentCloudUserName",
                  Password = "ConfluentCloudPassword",
```

```

        Protocol = BrokerProtocol.SaslSsl,
        AuthenticationMode = BrokerAuthenticationMode.Plain,
        ConsumerGroup = "$Default")] KafkaEventData<string>
kevent, ILogger log)
{
    log.LogInformation($"C# Kafka trigger function processed a message:
{kevent.Value}");
}

```

To receive events in a batch, use an input string or `KafkaEventData` as an array, as shown in the following example:

C#

```

[FunctionName("KafkaTriggerMany")]
public static void Run(
    [KafkaTrigger("BrokerList",
        "topic",
        Username = "ConfluentCloudUserName",
        Password = "ConfluentCloudPassword",
        Protocol = BrokerProtocol.SaslSsl,
        AuthenticationMode = BrokerAuthenticationMode.Plain,
        ConsumerGroup = "$Default")] KafkaEventData<string>[]
events, ILogger log)
{
    foreach (KafkaEventData<string> kevent in events)
    {
        log.LogInformation($"C# Kafka trigger function processed a
message: {kevent.Value}");
    }
}

```

The following function logs the message and headers for the Kafka Event:

C#

```

[FunctionName("KafkaTriggerSingleWithHeaders")]
public static void Run(
    [KafkaTrigger("BrokerList",
        "topic",
        Username = "ConfluentCloudUserName",
        Password = "ConfluentCloudPassword",
        Protocol = BrokerProtocol.SaslSsl,
        AuthenticationMode = BrokerAuthenticationMode.Plain,
        ConsumerGroup = "$Default")] KafkaEventData<string>
kevent, ILogger log)
{
    log.LogInformation($"C# Kafka trigger function processed a message:
{kevent.Value}");
    log.LogInformation("Headers: ");
    var headers = kevent.Headers;
}

```

```

        foreach (var header in headers)
    {
        log.LogInformation($"Key = {header.Key} Value =
{System.Text.Encoding.UTF8.GetString(header.Value)}");
    }
}

```

You can define a generic [Avro schema](#) for the event passed to the trigger. The following string value defines the generic Avro schema:

C#

```

const string PageViewsSchema = @"{
""type"": ""record"",
""name"": ""pageviews"",
""namespace"": ""ksql"",
""fields"": [
{
    ""name"": ""viewtime"",
    ""type"": ""long""
},
{
    ""name"": ""userid"",
    ""type"": ""string""
},
{
    ""name"": ""pageid"",
    ""type"": ""string""
}
]
}";

```

In the following function, an instance of `GenericRecord` is available in the `KafkaEvent.Value` property:

C#

```

[FunctionName(nameof(PageViews))]
public static void PageViews(
    [KafkaTrigger("LocalBroker", "pageviews", AvroSchema =
PageViewsSchema, ConsumerGroup = "azfunc")] KafkaEventData<string,
GenericRecord>[] kafkaEvents,
    long[] offsetArray,
    int[] partitionArray,
    string[] topicArray,
    DateTime[] timestampArray,
    ILogger logger)
{
    for (int i = 0; i < kafkaEvents.Length; i++)
    {
        var kafkaEvent = kafkaEvents[i];
    }
}

```

```

        if (kafkaEvent.Value is GenericRecord genericRecord)
    {
        logger.LogInformation(${timestampArray[i]} ${topicArray[i]} / ${partitionArray[i]} / ${offsetArray[i]}:
{GenericToJson(genericRecord)});
    }
}

```

You can define a specific [Avro schema](#) for the event passed to the trigger. The following defines the `UserRecord` class:

C#

```

public const string SchemaText = @"
{
    ""type"": ""record"",
    ""name"": ""UserRecord"",
    ""namespace"": ""KafkaFunctionSample"",
    ""fields"": [
        {
            ""name"": ""registertime"",
            ""type"": ""long""
        },
        {
            ""name"": ""userid"",
            ""type"": ""string""
        },
        {
            ""name"": ""regionid"",
            ""type"": ""string""
        },
        {
            ""name"": ""gender"",
            ""type"": ""string""
        }
    ]
};"

```

In the following function, an instance of `UserRecord` is available in the `KafkaEvent.Value` property:

C#

```

[FunctionName(nameof(User))]
public static void User(
    [KafkaTrigger("LocalBroker", "users", ConsumerGroup = "azfunc")]
    KafkaEventData<string, UserRecord>[] kafkaEvents,
    ILogger logger)
{
    foreach (var kafkaEvent in kafkaEvents)

```

```

    {
        logger.LogInformation($""
{JsonConvert.SerializeObject(kafkaEvent.Value)}");
    }
}

```

For a complete set of working .NET examples, see the [Kafka extension repository](#).

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the `KafkaTriggerAttribute` to define the function trigger.

The following table explains the properties you can set using this trigger attribute:

Parameter	Description
<code>BrokerList</code>	(Required) The list of Kafka brokers monitored by the trigger. See Connections for more information.
<code>Topic</code>	(Required) The topic monitored by the trigger.
<code>ConsumerGroup</code>	(Optional) Kafka consumer group used by the trigger.
<code>AvroSchema</code>	(Optional) Schema of a generic record when using the Avro protocol.
<code>AuthenticationMode</code>	(Optional) The authentication mode when using Simple Authentication and Security Layer (SASL) authentication. The supported values are <code>Gssapi</code> , <code>Plain</code> (default), <code>ScramSha256</code> , <code>ScramSha512</code> .
<code>Username</code>	(Optional) The username for SASL authentication. Not supported when <code>AuthenticationMode</code> is <code>Gssapi</code> . See Connections for more information.
<code>Password</code>	(Optional) The password for SASL authentication. Not supported when <code>AuthenticationMode</code> is <code>Gssapi</code> . See Connections for more information.
<code>Protocol</code>	(Optional) The security protocol used when communicating with brokers. The supported values are <code>plaintext</code> (default), <code>ssl</code> , <code>sasl_plaintext</code> , <code>sasl_ssl</code> .
<code>SslCaLocation</code>	(Optional) Path to CA certificate file for verifying the broker's certificate.
<code>SslCertificateLocation</code>	(Optional) Path to the client's certificate.
<code>SslKeyLocation</code>	(Optional) Path to client's private key (PEM) used for authentication.
<code>SslKeyPassword</code>	(Optional) Password for client's certificate.

Usage

In-process

Kafka events are passed to the function as `KafkaEventData<string>` objects or arrays. Strings and string arrays that are JSON payloads are also supported.

In a Premium plan, you must enable runtime scale monitoring for the Kafka output to be able to scale out to multiple instances. To learn more, see [Enable runtime scaling](#).

For a complete set of supported host.json settings for the Kafka trigger, see [host.json settings](#).

Connections

All connection information required by your triggers and bindings should be maintained in application settings and not in the binding definitions in your code. This is true for credentials, which should never be stored in your code.

ⓘ Important

Credential settings must reference an [application setting](#). Don't hard-code credentials in your code or configuration files. When running locally, use the [local.settings.json file](#) for your credentials, and don't publish the local.settings.json file.

Confluent

When connecting to a managed Kafka cluster provided by [Confluent in Azure](#), make sure that the following authentication credentials for your Confluent Cloud environment are set in your trigger or binding:

Setting	Recommended value	Description
BrokerList	<code>BootstrapServer</code>	App setting named <code>BootstrapServer</code> contains the value of bootstrap server found in Confluent Cloud settings page. The value resembles <code>xyz-xyzxyz.westeurope.azure.confluent.cloud:9092</code> .

Setting	Recommended value	Description
Username	<code>ConfluentCloudUsername</code>	App setting named <code>ConfluentCloudUsername</code> contains the API access key from the Confluent Cloud web site.
Password	<code>ConfluentCloudPassword</code>	App setting named <code>ConfluentCloudPassword</code> contains the API secret obtained from the Confluent Cloud web site.

The string values you use for these settings must be present as [application settings in Azure](#) or in the `Values` collection in the [local.settings.json file](#) during local development.

You should also set the `Protocol`, `AuthenticationMode`, and `SslCaLocation` in your binding definitions.

Next steps

- [Write to an Apache Kafka stream from a function](#)

Apache Kafka output binding for Azure Functions

Article • 06/14/2024

The output binding allows an Azure Functions app to write messages to a Kafka topic.

ⓘ Important

Kafka bindings are only available for Functions on the [Elastic Premium Plan](#) and [Dedicated \(App Service\).plan](#). They are only supported on version 3.x and later version of the Functions runtime.

Example

The usage of the binding depends on the C# modality used in your function app, which can be one of the following:

Isolated worker model

An [isolated worker process class library](#) compiled C# function runs in a process isolated from the runtime.

The attributes you use depend on the specific event provider.

Confluent

The following example has a custom return type that is `MultipleOutputType`, which consists of an HTTP response and a Kafka output.

C#

```
[Function("KafkaOutput")]

public static MultipleOutputType Output(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = null)]
    HttpRequestData req,
    FunctionContext executionContext)
{
    var log = executionContext.GetLogger("HttpFunction");
    log.LogInformation("C# HTTP trigger function processed a request.");
```

```

        string message = req.FunctionContext
            .BindingContext
            .BindingData["message"]
            .ToString();

    var response = req.CreateResponse(HttpStatusCode.OK);
    return new MultipleOutputType()
    {
        Kevent = message,
        HttpResponseMessage = response
    };
}

```

In the class `MultipleOutputType`, `Kevent` is the output binding variable for the Kafka binding.

C#

```

public class MultipleOutputType
{
    [KafkaOutput("BrokerList",
        "topic",
        Username = "ConfluentCloudUserName",
        Password = "ConfluentCloudPassword",
        Protocol = BrokerProtocol.SaslSsl,
        AuthenticationMode = BrokerAuthenticationMode.Plain
    )]
    public string Kevent { get; set; }

    public HttpResponseMessage HttpResponseMessage { get; set; }
}

```

To send a batch of events, pass a string array to the output type, as shown in the following example:

C#

```

[Function("KafkaOutputMany")]

public static MultipleOutputTypeForBatch Output(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = null)]
    HttpRequestData req,
    FunctionContext executionContext)
{
    var log = executionContext.GetLogger("HttpFunction");
    log.LogInformation("C# HTTP trigger function processed a request.");
    var response = req.CreateResponse(HttpStatusCode.OK);

    string[] messages = new string[2];
    messages[0] = "one";
    messages[1] = "two";
}

```

```

        return new MultipleOutputTypeForBatch()
    {
        Kevents = messages,
        HttpResponseMessage = response
    };
}

```

The string array is defined as `Kevents` property on the class, on which the output binding is defined:

```
C#
public class MultipleOutputTypeForBatch
{
    [KafkaOutput("BrokerList",
                  "topic",
                  Username = "ConfluentCloudUserName",
                  Password = "ConfluentCloudPassword",
                  Protocol = BrokerProtocol.SaslSsl,
                  AuthenticationMode = BrokerAuthenticationMode.Plain
    )]
    public string[] Kevents { get; set; }

    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

The following function adds headers to the Kafka output data:

```
C#
[Function("KafkaOutputWithHeaders")]

public static MultipleOutputType Output(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = null)]
HttpRequestData req,
    FunctionContext executionContext)
{
    var log = executionContext.GetLogger("HttpFunction");
    log.LogInformation("C# HTTP trigger function processed a request.");

    string message = req.FunctionContext
        .BindingContext
        .BindingData["message"]
        .ToString();

    string kevent = "{\"Offset\":364,\"Partition\":0,\"Topic\":\"kafkaeventhubtest1\",\"Timestamp\":\"2022-04-09T03:20:06.591Z\",\"Value\": \"\" + message + \"\",\"Headers\": [{\"Key\": \"test\", \"Value\": \"dotnet-isolated\" }] }";
    var response = req.CreateResponse(HttpStatusCode.OK);
    return new MultipleOutputType()
```

```

    {
        Kevent = kevent,
        HttpResponseMessage = response
    };
}

```

For a complete set of working .NET examples, see the [Kafka extension repository](#).

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the `Kafka` attribute to define the function trigger.

The following table explains the properties you can set using this attribute:

[\[+\] Expand table](#)

Parameter	Description
BrokerList	(Required) The list of Kafka brokers to which the output is sent. See Connections for more information.
Topic	(Required) The topic to which the output is sent.
AvroSchema	(Optional) Schema of a generic record when using the Avro protocol.
MaxMessageBytes	(Optional) The maximum size of the output message being sent (in MB), with a default value of <code>1</code> .
BatchSize	(Optional) Maximum number of messages batched in a single message set, with a default value of <code>10000</code> .
EnableIdempotence	(Optional) When set to <code>true</code> , guarantees that messages are successfully produced exactly once and in the original produce order, with a default value of <code>false</code> .
MessageTimeoutMs	(Optional) The local message timeout, in milliseconds. This value is only enforced locally and limits the time a produced message waits for successful delivery, with a default <code>300000</code> . A time of <code>0</code> is infinite. This value is the maximum time used to deliver a message (including retries). Delivery error occurs when either the retry count or the message timeout are exceeded.
RequestTimeoutMs	(Optional) The acknowledgment timeout of the output request, in milliseconds, with a default of <code>5000</code> .
MaxRetries	(Optional) The number of times to retry sending a failing Message, with a default of <code>2</code> . Retrying may cause reordering, unless <code>EnableIdempotence</code> is

Parameter	Description
	set to <code>true</code> .
AuthenticationMode	(Optional) The authentication mode when using Simple Authentication and Security Layer (SASL) authentication. The supported values are <code>Gssapi</code> , <code>Plain</code> (default), <code>ScramSha256</code> , <code>ScramSha512</code> .
Username	(Optional) The username for SASL authentication. Not supported when <code>AuthenticationMode</code> is <code>Gssapi</code> . See Connections for more information.
Password	(Optional) The password for SASL authentication. Not supported when <code>AuthenticationMode</code> is <code>Gssapi</code> . See Connections for more information.
Protocol	(Optional) The security protocol used when communicating with brokers. The supported values are <code>plaintext</code> (default), <code>ssl</code> , <code>sasl_plaintext</code> , <code>sasl_ssl</code> .
SslCaLocation	(Optional) Path to CA certificate file for verifying the broker's certificate.
SslCertificateLocation	(Optional) Path to the client's certificate.
SslKeyLocation	(Optional) Path to client's private key (PEM) used for authentication.
SslKeyPassword	(Optional) Password for client's certificate.

Usage

Both keys and values types are supported with built-in [Avro](#) and [Protobuf](#) serialization.

Please make sure to have access to the Kafka topic to which you are trying to write. You configure the binding with access and connection credentials to the Kafka topic.

In a Premium plan, you must enable runtime scale monitoring for the Kafka output to be able to scale out to multiple instances. To learn more, see [Enable runtime scaling](#).

For a complete set of supported host.json settings for the Kafka trigger, see [host.json settings](#).

Connections

All connection information required by your triggers and bindings should be maintained in application settings and not in the binding definitions in your code. This is true for credentials, which should never be stored in your code.

ⓘ Important

Credential settings must reference an [application setting](#). Don't hard-code credentials in your code or configuration files. When running locally, use the [local.settings.json file](#) for your credentials, and don't publish the local.settings.json file.

Confluent

When connecting to a managed Kafka cluster provided by [Confluent in Azure](#), make sure that the following authentication credentials for your Confluent Cloud environment are set in your trigger or binding:

[\[+\] Expand table](#)

Setting	Recommended value	Description
BrokerList	BootstrapServer	App setting named <code>BootstrapServer</code> contains the value of bootstrap server found in Confluent Cloud settings page. The value resembles <code>xyz-xyzxyz.westeurope.azure.confluent.cloud:9092</code> .
Username	ConfluentCloudUsername	App setting named <code>ConfluentCloudUsername</code> contains the API access key from the Confluent Cloud web site.
Password	ConfluentCloudPassword	App setting named <code>ConfluentCloudPassword</code> contains the API secret obtained from the Confluent Cloud web site.

The string values you use for these settings must be present as [application settings in Azure](#) or in the `values` collection in the [local.settings.json file](#) during local development.

You should also set the `Protocol`, `AuthenticationMode`, and `SslCaLocation` in your binding definitions.

Next steps

- Run a function from an Apache Kafka event stream

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Mobile Apps bindings for Azure Functions

Article • 06/14/2022

ⓘ Note

Azure Mobile Apps bindings are only available to Azure Functions 1.x. They are not supported in Azure Functions 2.x and higher.

This article explains how to work with [Azure Mobile Apps](#) bindings in Azure Functions. Azure Functions supports input and output bindings for Mobile Apps.

The Mobile Apps bindings let you read and update data tables in mobile apps.

Packages - Functions 1.x

Mobile Apps bindings are provided in the

[Microsoft.Azure.WebJobs.Extensions.MobileApps](#) NuGet package, version 1.x. Source code for the package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

The following table tells how to add support for this binding in each development environment.

Development environment	To add support in Functions 1.x
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#	Automatic
Portal development	Automatic

Input

The Mobile Apps input binding loads a record from a mobile table endpoint and passes it into your function. In C# and F# functions, any changes made to the record are automatically sent back to the table when the function exits successfully.

Input - example

See the language-specific example:

C# script

The following example shows a Mobile Apps input binding in a *function.json* file and a [C# script function](#) that uses the binding. The function is triggered by a queue message that has a record identifier. The function reads the specified record and modifies its `Text` property.

Here's the binding data in the *function.json* file:

JSON

```
{  
  "bindings": [  
    {  
      "name": "myQueueItem",  
      "queueName": "myqueue-items",  
      "connection": "",  
      "type": "queueTrigger",  
      "direction": "in"  
    },  
    {  
      "name": "record",  
      "type": "mobileTable",  
      "tableName": "MyTable",  
      "id": "{queueTrigger}",  
      "connection": "My_MobileApp_URL",  
      "apiKey": "My_MobileApp_Key",  
      "direction": "in"  
    }  
  ]  
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

C#

```
#r "Newtonsoft.Json"  
using Newtonsoft.Json.Linq;  
  
public static void Run(string myQueueItem, JObject record)  
{  
  if (record != null)  
  {  
    record["Text"] = "This has changed.";  
  }  
}
```

Input - attributes

In [C# class libraries](#), use the `MobileTable` attribute.

For information about attribute properties that you can configure, see [the following configuration section](#).

Input - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `MobileTable` attribute.

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to "mobileTable"
<code>direction</code>	n/a	Must be set to "in"
<code>name</code>	n/a	Name of input parameter in function signature.
<code>tableName</code>	<code>TableName</code>	Name of the mobile app's data table
<code>id</code>	<code>Id</code>	The identifier of the record to retrieve. Can be static or based on the trigger that invokes the function. For example, if you use a queue trigger for your function, then " <code>id": "{queueTrigger}"</code> uses the string value of the queue message as the record ID to retrieve.
<code>connection</code>	<code>Connection</code>	The name of an app setting that has the mobile app's URL. The function uses this URL to construct the required REST operations against your mobile app. Create an app setting in your function app that contains the mobile app's URL, then specify the name of the app setting in the <code>connection</code> property in your input binding. The URL looks like <code>http://<appname>.azurewebsites.net</code> .
<code>apiKey</code>	<code>ApiKey</code>	The name of an app setting that has your mobile app's API key. Provide the API key if you implement an API key in your Node.js mobile app, or implement an API key in your .NET mobile app. To provide the key, create an app setting in your function app that contains the API key, then add the <code>apiKey</code> property in your input binding with the name of the app setting.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

ⓘ Important

Don't share the API key with your mobile app clients. It should only be distributed securely to service-side clients, like Azure Functions. Azure Functions stores your connection information and API keys as app settings so that they are not checked into your source control repository. This safeguards your sensitive information.

Input - usage

In C# functions, when the record with the specified ID is found, it is passed into the named `JObject` parameter. When the record is not found, the parameter value is `null`.

In JavaScript functions, the record is passed into the `context.bindings.<name>` object. When the record is not found, the parameter value is `null`.

In C# and F# functions, any changes you make to the input record (input parameter) are automatically sent back to the table when the function exits successfully. You can't modify a record in JavaScript functions.

Output

Use the Mobile Apps output binding to write a new record to a Mobile Apps table.

Output - example

C#

The following example shows a [C# function](#) that is triggered by a queue message and creates a record in a mobile app table.

C#

```
[FunctionName("MobileAppsOutput")]
[return: MobileTable(ApiKeySetting = "MyMobileAppKey", TableName =
"MyTable", MobileAppUriSetting = "MyMobileAppUri")]
public static object Run(
    [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")]
    string myQueueItem,
    TraceWriter log)
{
    return new { Text = $"I'm running in a C# function! {myQueueItem}" }
```

```
};  
}
```

Output - attributes

In [C# class libraries](#), use the `MobileTable` attribute.

For information about attribute properties that you can configure, see [Output - configuration](#). Here's a `MobileTable` attribute example in a method signature:

C#

```
C#  
  
[FunctionName("MobileAppsOutput")]  
[return: MobileTable(ApiKeySetting = "MyMobileAppKey", TableName =  
"MyTable", MobileAppUriSetting = "MyMobileAppUri")]  
public static object Run(  
    [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")]  
    string myQueueItem,  
    TraceWriter log)  
{  
    ...  
}
```

Output - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `MobileTable` attribute.

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to "mobileTable"
<code>direction</code>	n/a	Must be set to "out"
<code>name</code>	n/a	Name of output parameter in function signature.
<code>tableName</code>	<code>TableName</code>	Name of the mobile app's data table

function.json Attribute property	Description
connection	MobileAppUriSetting The name of an app setting that has the mobile app's URL. The function uses this URL to construct the required REST operations against your mobile app. Create an app setting in your function app that contains the mobile app's URL, then specify the name of the app setting in the <code>connection</code> property in your input binding. The URL looks like <code>http://<appname>.azurewebsites.net</code> .
apiKey	ApiKeySetting The name of an app setting that has your mobile app's API key. Provide the API key if you implement an API key in your Node.js mobile app backend, or implement an API key in your .NET mobile app backend. To provide the key, create an app setting in your function app that contains the API key, then add the <code>apiKey</code> property in your input binding with the name of the app setting.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

ⓘ Important

Don't share the API key with your mobile app clients. It should only be distributed securely to service-side clients, like Azure Functions. Azure Functions stores your connection information and API keys as app settings so that they are not checked into your source control repository. This safeguards your sensitive information.

Output - usage

In C# script functions, use a named output parameter of type `out object` to access the output record. In C# class libraries, the `MobileTable` attribute can be used with any of the following types:

- `ICollector<T>` or `IAsyncCollector<T>`, where `T` is either `JObject` or any type with a `public string Id` property.
- `out JObject`
- `out T` or `out T[]`, where `T` is any Type with a `public string Id` property.

In Node.js functions, use `context.bindings.<name>` to access the output record.

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Notification Hubs output bindings for Azure Functions

Article • 07/02/2024

This article explains how to send push notifications by using [Azure Notification Hubs](#) bindings in Azure Functions. Azure Functions supports output bindings for Notification Hubs.

You must configure Notification Hubs for the Platform Notifications Service (PNS) you want to use. For more information about how to get push notifications in your client app from Notification Hubs, see [Quickstart: Set up push notifications in a notification hub](#).

ⓘ Important

Google has [deprecated Google Cloud Messaging \(GCM\) in favor of Firebase Cloud Messaging \(FCM\)](#). However, output bindings for Notification Hubs doesn't support FCM. To send notifications using FCM, use the [Firebase API](#) directly in your function or use [template notifications](#).

Packages: Functions 1.x

ⓘ Important

[Support will end for version 1.x of the Azure Functions runtime on September 14, 2026](#). We highly recommend that you [migrate your apps to version 4.x](#) for full support.

The Notification Hubs bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.NotificationHubs](#) NuGet package, version 1.x. Source code for the package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

The following table lists how to add support for output binding in each development environment.

[+] Expand table

Development environment	To add support in Functions 1.x
Local development: C# class library	Install the package
Local development: C# script, JavaScript, F#	Automatic
Portal development	Automatic

Packages: Functions 2.x and higher

Output binding isn't available in Functions 2.x and higher.

Example: template

The notifications you send can be native notifications or [template notifications](#). A native notification targets a specific client platform, as configured in the `platform` property of the output binding. A template notification can be used to target multiple platforms.

Template examples for each language:

- [C# script: out parameter](#)
- [C# script: asynchronous](#)
- [C# script: JSON](#)
- [C# script: library types](#)
- [F#](#)
- [JavaScript](#)

C# script template example: out parameter

This example sends a notification for a [template registration](#) that contains a `message` placeholder in the template:

```
cs

using System;
using System.Threading.Tasks;
using System.Collections.Generic;

public static void Run(string myQueueItem, out IDictionary<string, string>
notification, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    notification = GetTemplateProperties(myQueueItem);
}
```

```
private static IDictionary<string, string> GetTemplateProperties(string message)
{
    Dictionary<string, string> templateProperties = new Dictionary<string, string>();
    templateProperties["message"] = message;
    return templateProperties;
}
```

C# script template example: asynchronous

If you're using asynchronous code, out parameters aren't allowed. In this case, use `IAsyncCollector` to return your template notification. The following code is an asynchronous example of the previous example:

cs

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;

public static async Task Run(string myQueueItem,
IAsyncCollector<IDictionary<string, string>> notification, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    log.Info($"Sending Template Notification to Notification Hub");
    await notification.AddAsync(GetTemplateProperties(myQueueItem));
}

private static IDictionary<string, string> GetTemplateProperties(string message)
{
    Dictionary<string, string> templateProperties = new Dictionary<string, string>();
    templateProperties["user"] = "A new user wants to be added : " +
message;
    return templateProperties;
}
```

C# script template example: JSON

This example sends a notification for a `template registration` that contains a `message` placeholder in the template using a valid JSON string:

cs

```

using System;

public static void Run(string myQueueItem, out string notification,
TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    notification = "{\"message\":\"Hello from C#. Processed a queue
item!\\"}";
}

```

C# script template example: library types

This example shows how to use types defined in the [Microsoft Azure Notification Hubs Library](#):

```

cs

#r "Microsoft.Azure.NotificationHubs"

using System;
using System.Threading.Tasks;
using Microsoft.Azure.NotificationHubs;

public static void Run(string myQueueItem, out Notification notification,
TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    notification = GetTemplateNotification(myQueueItem);
}

private static TemplateNotification GetTemplateNotification(string message)
{
    Dictionary<string, string> templateProperties = new Dictionary<string,
string>();
    templateProperties["message"] = message;
    return new TemplateNotification(templateProperties);
}

```

F# template example

This example sends a notification for a [template registration](#) that contains `location` and `message`:

```

F#

let Run(myTimer: TimerInfo, notification: byref<IDictionary<string,
string>>) =

```

```
notification = dict [("location", "Redmond"); ("message", "Hello from F#!")]
```

JavaScript template example

This example sends a notification for a [template registration](#) that contains `location` and `message`:

JavaScript

```
module.exports = async function (context, myTimer) {
    var timeStamp = new Date().toISOString();

    if (myTimer.IsPastDue)
    {
        context.log('Node.js is running late!');
    }
    context.log('Node.js timer trigger function ran!', timeStamp);
    context.bindings.notification = {
        location: "Redmond",
        message: "Hello from Node!"
    };
};
```

Example: APNS native

This C# script example shows how to send a native Apple Push Notification Service (APNS) notification:

cs

```
#r "Microsoft.Azure.NotificationHubs"
#r "Newtonsoft.Json"

using System;
using Microsoft.Azure.NotificationHubs;
using Newtonsoft.Json;

public static async Task Run(string myQueueItem,
IAsyncCollector<Notification> notification, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    // In this example, the queue item is a new user to be processed in the
    // form of a JSON string with
    // a "name" value.
    //
    // The JSON format for a native Apple Push Notification Service (APNS)
```

```

notification is:
// { "aps": { "alert": "notification message" }}

log.LogInformation($"Sending APNS notification of a new user");
dynamic user = JsonConvert.DeserializeObject(myQueueItem);
string apnsNotificationPayload = "{\"aps\": {\"alert\": \"A new user
wants to be added (" +
                                user.name + ")\" }}";
log.LogInformation($"{apnsNotificationPayload}");
await notification.AddAsync(new
AppleNotification(apnsNotificationPayload));
}

```

Example: WNS native

This C# script example shows how to use types defined in the [Microsoft Azure Notification Hubs Library](#) to send a native Windows Push Notification Service (WNS) toast notification:

```

cs

#r "Microsoft.Azure.NotificationHubs"
#r "Newtonsoft.Json"

using System;
using Microsoft.Azure.NotificationHubs;
using Newtonsoft.Json;

public static async Task Run(string myQueueItem,
IAsyncCollector<Notification> notification, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    // In this example, the queue item is a new user to be processed in the
    // form of a JSON string with
    // a "name" value.
    //
    // The XML format for a native WNS toast notification is ...
    // <?xml version="1.0" encoding="utf-8"?>
    // <toast>
    //     <visual>
    //         <binding template="ToastText01">
    //             <text id="1">notification message</text>
    //         </binding>
    //     </visual>
    // </toast>

    log.Info($"Sending WNS toast notification of a new user");
    dynamic user = JsonConvert.DeserializeObject(myQueueItem);
    string wnsNotificationPayload = "<?xml version=\"1.0\" encoding=\"utf-
8\"?>" +

```

```

        "<toast><visual><binding
template=\"ToastText01\">" +
                    "<text id=\"1\">" +
                    "A new user wants to be added (" +
                    "+ user.name + ")" +
                    "</text>" +
                "</binding></visual></toast>";

        log.Info(${wnsNotificationPayload});
        await notification.AddAsync(new
WindowsNotification(wnsNotificationPayload));
    }
}

```

Attributes

In [C# class libraries](#), use the `NotificationHub` attribute.

The attribute's constructor parameters and properties are described in the [Configuration](#) section.

Configuration

The following table lists the binding configuration properties that you set in the `function.json` file and the `NotificationHub` attribute:

[] [Expand table](#)

function.json property	Attribute property	Description
<code>type</code>	n/a	Set to <code>notificationHub</code> .
<code>direction</code>	n/a	Set to <code>out</code> .
<code>name</code>	n/a	Variable name used in function code for the notification hub message.
<code>tagExpression</code>	<code>TagExpression</code>	Tag expressions allow you to specify that notifications be delivered to a set of devices that are registered to receive notifications matching the tag expression. For more information, see Routing and tag expressions .
<code>hubName</code>	<code>HubName</code>	The name of the notification hub resource in the Azure portal.

function.json	Attribute property	Description
property		
connection	ConnectionStringSetting	The name of an app setting that contains a Notification Hubs connection string. Set the connection string to the <i>DefaultFullSharedAccessSignature</i> value for your notification hub. For more information, see Connection string setup .
platform	Platform	<p>The platform property indicates the client platform your notification targets. By default, if the platform property is omitted from the output binding, template notifications can be used to target any platform configured on the Azure Notification Hub. For more information about using templates to send cross-platform notifications with an Azure Notification Hub, see Notification Hubs templates.</p> <p>When platform is set, it must be one of the following values:</p> <ul style="list-style-type: none"> • <code>apns</code>: Apple Push Notification Service. For more information on configuring the notification hub for APNS and receiving the notification in a client app, see Send push notifications to iOS with Azure Notification Hubs. • <code>adm</code>: Amazon Device Messaging. For more information on configuring the notification hub for Azure Deployment Manager (ADM) and receiving the notification in a Kindle app, see Send push notifications to Android devices using Firebase SDK. • <code>wns</code>: Windows Push Notification Services targeting Windows platforms. WNS also supports Windows Phone 8.1 and later. For more information, see Send notifications to Universal Windows Platform apps using Azure Notification Hubs. • <code>mpns</code>: Microsoft Push Notification Service. This platform supports Windows Phone 8 and earlier Windows Phone platforms. For more information, see Send notifications to Universal Windows Platform apps using Azure Notification Hubs.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

function.json file example

Here's an example of a Notification Hubs binding in a *function.json* file:

JSON

```
{  
  "bindings": [  
    {  
      "type": "notificationHub",  
      "direction": "out",  
      "name": "notification",  
      "tagExpression": "",  
      "hubName": "my-notification-hub",  
      "connection": "MyHubConnectionString",  
      "platform": "apns"  
    }  
,  
    {"disabled": false  
  }
```

Connection string setup

To use a notification hub output binding, you must configure the connection string for the hub. You can select an existing notification hub or create a new one from the **Integrate** tab in the Azure portal. You can also configure the connection string manually.

To configure the connection string to an existing notification hub:

1. Navigate to your notification hub in the [Azure portal](#), choose **Access policies**, and select the copy button next to the **DefaultFullSharedAccessSignature** policy.

The connection string for the *DefaultFullSharedAccessSignature* policy is copied to your notification hub. This connection string lets your function send notification

messages to the hub.

Policy Name	Permission	Connection String
DefaultListenSharedAccessSignature	Listen	Endpoint=sb://myNotificationHub-1.servicebus.windows.net/...
DefaultFullSharedAccessSignature	Manage,Listen,Send	Endpoint=sb://myNotificationHub-1.servicebus.windows.net/...

2. Navigate to your function app in the Azure portal, expand **Settings**, and then select **Environment variables**.
3. From the **App setting** tab, select **+ Add** to add a key such as **MyHubConnectionString**. The **Name** of this app setting is the output binding connection setting in *function.json* or the .NET attribute. For more information, see [Configuration](#).
4. For the value, paste the copied *DefaultFullSharedAccessSignature* connection string from your notification hub, and then select **Apply**.

When you're developing locally, add your application settings in the [local.settings.json](#) file in the `Values` collection.

Exceptions and return codes

[+] [Expand table](#)

Binding	Reference
Notification Hub	Operations Guide

Related content

- [Azure Functions triggers and bindings concepts](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Azure Queue storage trigger and bindings for Azure Functions overview

Article • 07/11/2023

Azure Functions can run as new Azure Queue storage messages are created and can write queue messages within a function.

Action	Type
Run a function as queue storage data changes	Trigger
Write queue storage messages	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

In-process

Functions execute in the same process as the Functions host. To learn more, see [Develop C# class library functions using Azure Functions](#).

In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. To update existing binding extensions for C# script apps running in the portal without having to republish your function app, see [Update your extensions](#).

The functionality of the extension varies depending on the extension version:

Extension 5.x+

This section describes using a [class library](#). For [C# scripting](#), you would need to instead [install the extension bundle](#), version 4.x.

This version introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

This version allows you to bind to types from [Azure.Storage.Queues](#).

This extension is available by installing the [Microsoft.Azure.WebJobs.Extensions.Storage.Queues NuGet package](#), version 5.x.

Using the .NET CLI:

.NET CLI

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage.Queues --version 5.0.0
```

!Note

Azure Blobs, Azure Queues, and Azure Tables now use separate extensions and are referenced individually. For example, to use the triggers and bindings for all three services in your .NET in-process app, you should add the following packages to your project:

- [Microsoft.Azure.WebJobs.Extensions.Storage.Blobs](#)
- [Microsoft.Azure.WebJobs.Extensions.Storage.Queues](#)
- [Microsoft.Azure.WebJobs.Extensions.Tables](#)

Previously, the extensions shipped together as

[Microsoft.Azure.WebJobs.Extensions.Storage, version 4.x](#). This same package also has a 5.x version, which references the split packages for blobs and queues only. When upgrading your package references from older versions, you may therefore need to additionally reference the new [Microsoft.Azure.WebJobs.Extensions.Tables](#) NuGet package. Also, when referencing these newer split packages, make sure you are not referencing an older version of the combined storage package, as this will result in conflicts from two definitions of the same bindings.

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

In-process

An in-process class library is a compiled C# function runs in the same process as the

Functions runtime.

Choose a version to see binding type details for the mode and version.

Extension 5.x+

The Azure Queues extension supports parameter types according to the table below.

Binding scenario	Parameter types
Queue trigger	<code>QueueMessage</code> JSON serializable types ¹ <code>string</code> <code>byte[]</code> <code>BinaryData</code>
Queue output (single message)	<code>QueueMessage</code> JSON serializable types ¹ <code>string</code> <code>byte[]</code> <code>BinaryData</code>
Queue output (multiple messages)	<code>QueueClient</code> <code>ICollector<T></code> or <code>IAsyncCollector<T></code> where <code>T</code> is one of the single message types

¹ Messages containing JSON data can be deserialized into known plain-old CLR object (POCO) types.

host.json settings

This section describes the configuration settings available for this binding in versions 2.x and higher. Settings in the host.json file apply to all functions in a function app instance. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

Note

For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

JSON

```
{
  "version": "2.0",
  "extensions": {
    "queues": {
      "maxPollingInterval": "00:00:02",
      "visibilityTimeout" : "00:00:30",
      "batchSize": 16,
      "maxDequeueCount": 5,
      "newBatchThreshold": 8,
      "messageEncoding": "base64"
    }
  }
}
```

Property	Default	Description
maxPollingInterval	00:01:00	The maximum interval between queue polls. The minimum interval is 00:00:00.100 (100 ms). Intervals increment up to <code>maxPollingInterval</code> . The default value of <code>maxPollingInterval</code> is 00:01:00 (1 min). <code>maxPollingInterval</code> must not be less than 00:00:00.100 (100 ms). In Functions 2.x and later, the data type is a <code>TimeSpan</code> . In Functions 1.x, it is in milliseconds.
visibilityTimeout	00:00:00	The time interval between retries when processing of a message fails.
batchSize	16	<p>The number of queue messages that the Functions runtime retrieves simultaneously and processes in parallel. When the number being processed gets down to the <code>newBatchThreshold</code>, the runtime gets another batch and starts processing those messages. So the maximum number of concurrent messages being processed per function is <code>batchSize</code> plus <code>newBatchThreshold</code>. This limit applies separately to each queue-triggered function.</p> <p>If you want to avoid parallel execution for messages received on one queue, you can set <code>batchSize</code> to 1. However, this setting eliminates concurrency as long as your function app runs only on a single virtual machine (VM). If the function app scales out to multiple VMs, each VM could run one instance of each queue-triggered function.</p> <p>The maximum <code>batchSize</code> is 32.</p>
maxDequeueCount	5	The number of times to try processing a message before

Property	Default	Description
		moving it to the poison queue.
newBatchThreshold	N*batchSize/2	Whenever the number of messages being processed concurrently gets down to this number, the runtime retrieves another batch. N represents the number of vCPUs available when running on App Service or Premium Plans. Its value is 1 for the Consumption Plan.
messageEncoding	base64	This setting is only available in extension bundle version 5.0.0 and higher . It represents the encoding format for messages. Valid values are base64 and none.

Next steps

- [Run a function as queue storage data changes \(Trigger\)](#)
- [Write queue storage messages \(Output binding\)](#)

Azure Queue storage trigger for Azure Functions

Article • 01/30/2024

The queue storage trigger runs a function as messages are added to Azure Queue storage.

Azure Queue storage scaling decisions for the Consumption and Premium plans are done via target-based scaling. For more information, see [Target-based scaling](#).

Example

Use the queue trigger to start a function when a new item is received on a queue. The queue message is provided as input to the function.

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

The following example shows a [C# function](#) that polls the `input-queue` queue and writes several messages to an output queue each time a queue item is processed.

C#

```
[Function(nameof(QueueFunction))]
[QueueOutput("output-queue")]
public string[] Run([QueueTrigger("input-queue")] Album myQueueItem, FunctionContext context)
{
    // Use a string array to return more than one message.
    string[] messages = {
        $"Album name = {myQueueItem.Name}",
        $"Album songs = {myQueueItem.Songs.ToString()}";

    _logger.LogInformation("{msg1},{msg2}", messages[0], messages[1]);

    // Queue Output messages
    return messages;
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the [QueueTriggerAttribute](#) to define the function. C# script instead uses a function.json configuration file as described in the [C# scripting guide](#).

Isolated worker model

In [C# class libraries](#), the attribute's constructor takes the name of the queue to monitor, as shown in the following example:

C#

```
[Function(nameof(QueueFunction))]  
[QueueOutput("output-queue")]  
public string[] Run([QueueTrigger("input-queue")] Album myQueueItem, FunctionContext context)
```

This example also demonstrates setting the [connection string setting](#) in the attribute itself.

See the [Example section](#) for complete examples.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

Usage

ⓘ Note

Functions expect a *base64* encoded string. Any adjustments to the encoding type (in order to prepare data as a *base64* encoded string) need to be implemented in the calling service.

The usage of the Queue trigger depends on the extension package version, and the C# modality used in your function app, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see usage details for the mode and version.

Extension 5.x+

The queue trigger can bind to the following types:

[Expand table](#)

Type	Description
<code>string</code>	The message content as a string. Use when the message is simple text..
<code>byte[]</code>	The bytes of the message.
JSON serializable types	When a queue message contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
QueueMessage ¹	The message.
BinaryData ¹	The bytes of the message.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues 5.2.0 or later](#) and the [common dependencies for SDK type bindings](#).

Metadata

The queue trigger provides several [metadata properties](#). These properties can be used as part of binding expressions in other bindings or as parameters in your code, for language workers that provide this access to message metadata.

The message metadata properties are members of the [CloudQueueMessage](#) class.

[Expand table](#)

Property	Type	Description
QueueTrigger	string	Queue payload (if a valid string). If the queue message payload is a string, <code>QueueTrigger</code> has the same value as the variable named by the <code>name</code> property in <code>function.json</code> .
DequeueCount	long	The number of times this message has been dequeued.
ExpirationTime	DateTimeOffset	The time that the message expires.
Id	string	Queue message ID.
InsertionTime	DateTimeOffset	The time that the message was added to the queue.
NextVisibleTime	DateTimeOffset	The time that the message will next be visible.
PopReceipt	string	The message's pop receipt.

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to Azure Queues. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Manage storage account access keys](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [version 5.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

[+] [Expand table](#)

Property	Environment variable template	Description	Example value
Queue Service URI	<code><CONNECTION_NAME_PREFIX>__queueServiceUri</code> ¹	The data plane URI of the queue service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.queue.core.windows.net</code>

¹ `<CONNECTION_NAME_PREFIX>__serviceUri` can be used as an alias. If both forms are provided, the `queueServiceUri` form is used. The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables.

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

i Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the [principle of least privilege](#), granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You will need to create a role assignment that provides access to your queue at runtime. Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Queue Storage extension in normal operation. Your application may require additional permissions based on the code you write.

[Expand table](#)

Binding type	Example built-in roles
Trigger	Storage Queue Data Reader , Storage Queue Data Message Processor
Output binding	Storage Queue Data Contributor , Storage Queue Data Message Sender

Poison messages

When a queue trigger function fails, Azure Functions retries the function up to five times for a given queue message, including the first try. If all five attempts fail, the functions runtime adds a message to a queue named `<originalqueueusername>-poison`. You can write a function to process messages from the poison queue by logging them or sending a notification that manual attention is needed.

To handle poison messages manually, check the [dequeueCount](#) of the queue message.

Peek lock

The peek-lock pattern happens automatically for queue triggers. As messages are dequeued, they are marked as invisible and associated with a 10-minute timeout managed by the Storage service. This timeout can't be changed.

When the function starts, it starts processing a message under the following conditions.

- If the function is successful, then the function execution completes and the message is deleted.
- If the function fails, then the message visibility is reset. After being reset, the message is reprocessed the next time the function requests a new message.
- If the function never completes due to a crash, the message visibility expires and the message reappears in the queue.

All of the visibility mechanics are handled by the Storage service, not the Functions runtime.

Polling algorithm

The queue trigger implements a random exponential back-off algorithm to reduce the effect of idle-queue polling on storage transaction costs.

The algorithm uses the following logic:

- When a message is found, the runtime waits 100 milliseconds and then checks for another message
- When no message is found, it waits about 200 milliseconds before trying again.
- After subsequent failed attempts to get a queue message, the wait time continues to increase until it reaches the maximum wait time, which defaults to one minute.
- The maximum wait time is configurable via the `maxPollingInterval` property in the [host.json file](#).

For local development the maximum polling interval defaults to two seconds.

Note

In regards to billing when hosting function apps in the Consumption plan, you are not charged for time spent polling by the runtime.

Concurrency

When there are multiple queue messages waiting, the queue trigger retrieves a batch of messages and invokes function instances concurrently to process them. By default, the batch size is 16. When the number being processed gets down to 8, the runtime gets another batch and starts processing those messages. So the maximum number of concurrent messages being processed per function on one virtual machine (VM) is 24. This limit applies separately to each queue-triggered function on each VM. If your function app scales out to multiple VMs, each VM will wait for triggers and attempt to run functions. For example, if a function app scales out to 3 VMs, the default maximum number of concurrent instances of one queue-triggered function is 72.

The batch size and the threshold for getting a new batch are configurable in the [host.json file](#). If you want to minimize parallel execution for queue-triggered functions in a function app, you can set the batch size to 1. This setting eliminates concurrency only so long as your function app runs on a single virtual machine (VM).

The queue trigger automatically prevents a function from processing a queue message multiple times simultaneously.

host.json properties

The host.json file contains settings that control queue trigger behavior. See the [host.json settings](#) section for details regarding available settings.

Next steps

- [Write queue storage messages \(Output binding\)](#)

Azure Queue storage output bindings for Azure Functions

Article • 08/15/2024

Azure Functions can create new Azure Queue storage messages by setting up an output binding.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

C#

```
[Function(nameof(QueueFunction))]
[QueueOutput("output-queue")]
public string[] Run([QueueTrigger("input-queue")] Album myQueueItem, FunctionContext context)
{
    // Use a string array to return more than one message.
    string[] messages = {
        $"Album name = {myQueueItem.Name}",
        $"Album songs = {myQueueItem.Songs.ToString()}};

    _logger.LogInformation("{msg1},{msg2}", messages[0], messages[1]);

    // Queue Output messages
    return messages;
}
```

For an end-to-end example of how to configure an output binding to Queue storage, see one of these articles:

- [Connect functions to Azure Storage using Visual Studio](#)
- [Connect functions to Azure Storage using Visual Studio Code](#)
- [Connect functions to Azure Storage using command line tools](#)

Attributes

The attribute that defines an output binding in C# libraries depends on the mode in which the C# class library runs.

Isolated worker model

When running in an isolated worker process, you use the [QueueOutputAttribute](#), which takes the name of the queue, as shown in the following example:

C#

```
[Function(nameof(QueueFunction))]  
[QueueOutput("output-queue")]  
public string[] Run([QueueTrigger("input-queue")] Album myQueueItem, FunctionContext context)
```

Only returned variables are supported when running in an isolated worker process. Output parameters can't be used.

See the [Example section](#) for complete examples.

Usage

The usage of the Queue output binding depends on the extension package version and the C# modality used in your function app, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see usage details for the mode and version.

Extension 5.x+

When you want the function to write a single message, the queue output binding can bind to the following types:

[Expand table](#)

Type	Description
<code>string</code>	The message content as a string. Use when the message is simple text.
<code>byte[]</code>	The bytes of the message.
<code>JSON serializable types</code>	An object representing the content of a JSON message. Functions tries to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write multiple messages, the queue output binding can bind to the following types:

[Expand table](#)

Type	Description
<code>T[]</code> where <code>T</code> is one of the single message types	An array containing content for multiple messages. Each entry represents one message.

For other output scenarios, create and use types from [Azure.Storage.Queues](#) directly.

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to Azure Queues. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Manage storage account access keys](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [version 5.x or higher of the extension](#) ([bundle 3.x or higher](#) for non-.NET language stacks), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

[\[+\] Expand table](#)

Property	Environment variable template	Description	Example value
Queue Service URI	<code><CONNECTION_NAME_PREFIX>__queueServiceUri</code> ¹	The data plane URI of the queue service to which you're	<code>https://<storage_account_name>.queue.core.windows.net</code>

Property	Environment variable template	Description	Example value
		connecting, using the HTTPS scheme.	

¹ `<CONNECTION_NAME_PREFIX>_serviceUri` can be used as an alias. If both forms are provided, the `queueServiceUri` form is used. The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables.

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not supported**. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

i Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the [principle of least privilege](#), granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You will need to create a role assignment that provides access to your queue at runtime. Management roles like [Owner](#) are not sufficient. The following table shows built-in roles that are recommended when using the Queue Storage extension in normal operation. Your application may require additional permissions based on the code you write.

[] [Expand table](#)

Binding type	Example built-in roles
Trigger	Storage Queue Data Reader , Storage Queue Data Message Processor
Output binding	Storage Queue Data Contributor , Storage Queue Data Message Sender

Exceptions and return codes

[] [Expand table](#)

Binding	Reference
Queue	Queue Error Codes
Blob, Table, Queue	Storage Error Codes
Blob, Table, Queue	Troubleshooting

Next steps

- Run a function as queue storage data changes (Trigger)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RabbitMQ bindings for Azure Functions overview

Article • 03/31/2024

ⓘ Note

The RabbitMQ bindings are only fully supported on [Premium](#) and [Dedicated App Service](#) plans. Consumption plans aren't supported.

RabbitMQ bindings are only supported for Azure Functions version 3.x and later versions.

Azure Functions integrates with [RabbitMQ](#) via [triggers and bindings](#). The Azure Functions RabbitMQ extension allows you to send and receive messages using the RabbitMQ API with Functions.

[+] Expand table

Action	Type
Run a function when a RabbitMQ message comes through the queue	Trigger
Send RabbitMQ messages	Output binding

Prerequisites

Before working with the RabbitMQ extension, you must [set up your RabbitMQ endpoint](#). To learn more about RabbitMQ, see the [getting started page](#).

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing this [NuGet package](#).

Next steps

- Run a function when a RabbitMQ message is created (Trigger)
- Send RabbitMQ messages from Azure Functions (Output binding)

RabbitMQ trigger for Azure Functions overview

Article • 09/27/2023

ⓘ Note

The RabbitMQ bindings are only fully supported on **Premium and Dedicated** plans. Consumption is not supported.

Use the RabbitMQ trigger to respond to messages from a RabbitMQ queue.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

Isolated worker model

```
C#  
  
[Function(nameof(RabbitMQFunction))]  
[RabbitMQOutput(QueueName = "destinationQueue", ConnectionStringSetting  
= "RabbitMQConnection")]  
public static string Run([RabbitMQTrigger("queue",  
ConnectionStringSetting = "RabbitMQConnection")] string item,  
FunctionContext context)  
{  
    var logger = context.GetLogger(nameof(RabbitMQFunction));  
  
    logger.LogInformation(item);  
  
    var message = $"Output message created at {DateTime.Now}";
```

```
    return message;  
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the attribute to define the function. C# script instead uses a [function.json configuration file](#).

The attribute's constructor takes the following parameters:

Parameter	Description
QueueName	Name of the queue from which to receive messages.
HostName	Hostname of the queue, such as 10.26.45.210. Ignored when using <code>ConnectionStringSetting</code> .
UserNameSetting	Name of the app setting that contains the username to access the queue, such as <code>UserNameSetting: "%< UserNameFromSettings >%"</code> . Ignored when using <code>ConnectionStringSetting</code> .
PasswordSetting	Name of the app setting that contains the password to access the queue, such as <code>PasswordSetting: "%< PasswordFromSettings >%"</code> . Ignored when using <code>ConnectionStringSetting</code> .
ConnectionStringSetting	The name of the app setting that contains the RabbitMQ message queue connection string. The trigger won't work when you specify the connection string directly instead through an app setting. For example, when you have set <code>ConnectionStringSetting: "rabbitMQConnection"</code> , then in both the <code>local.settings.json</code> and in your function app you need a setting like <code>"RabbitMQConnection" : "< ActualConnectionString >"</code> .
Port	Gets or sets the port used. Defaults to 0, which points to the RabbitMQ client's default port setting of 5672.

Isolated worker model

In [C# class libraries](#), use the [RabbitMQTrigger](#) attribute.

Here's a `RabbitMQTrigger` attribute in a method signature for an isolated worker process library:

C#

```
[Function(nameof(RabbitMQFunction))]  
[RabbitMQOutput(QueueName = "destinationQueue", ConnectionStringSetting  
= "RabbitMQConnection")]  
public static string Run([RabbitMQTrigger("queue",  
ConnectionStringSetting = "RabbitMQConnection")] string item,  
FunctionContext context)  
{
```

See the [Example section](#) for complete examples.

Usage

The parameter type supported by the RabbitMQ trigger depends on the C# modality used.

Isolated worker model

The RabbitMQ bindings currently support only string and serializable object types when running in an isolated process.

For a complete example, see C# [example](#).

Dead letter queues

Dead letter queues and exchanges can't be controlled or configured from the RabbitMQ trigger. To use dead letter queues, pre-configure the queue used by the trigger in RabbitMQ. Refer to the [RabbitMQ documentation](#) ↗.

host.json settings

This section describes the configuration settings available for this binding in versions 2.x and higher. Settings in the host.json file apply to all functions in a function app instance. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

JSON

```
{  
    "version": "2.0",  
    "extensions": {
```

```

    "rabbitMQ": {
        "prefetchCount": 100,
        "queueName": "queue",
        "connectionString": "amqp://user:password@url:port",
        "port": 10
    }
}

```

Property	Default	Description
prefetchCount	30	Gets or sets the number of messages that the message receiver can simultaneously request and is cached.
queueName	n/a	Name of the queue to receive messages from.
connectionString	n/a	The RabbitMQ message queue connection string. The connection string is directly specified here and not through an app setting.
port	0	(ignored if using connectionString) Gets or sets the Port used. Defaults to 0, which points to rabbitmq client's default port setting: 5672.

Local testing

① Note

The connectionString takes precedence over "hostName", "userName", and "password". If these are all set, the connectionString will override the other two.

If you're testing locally without a connection string, you should set the "hostName" setting and "userName" and "password" if applicable in the "rabbitMQ" section of `host.json`:

JSON

```
{
    "version": "2.0",
    "extensions": {
        "rabbitMQ": {
            ...
            "hostName": "localhost",
            "username": "userNameSetting",
            "password": "passwordSetting"
        }
    }
}
```

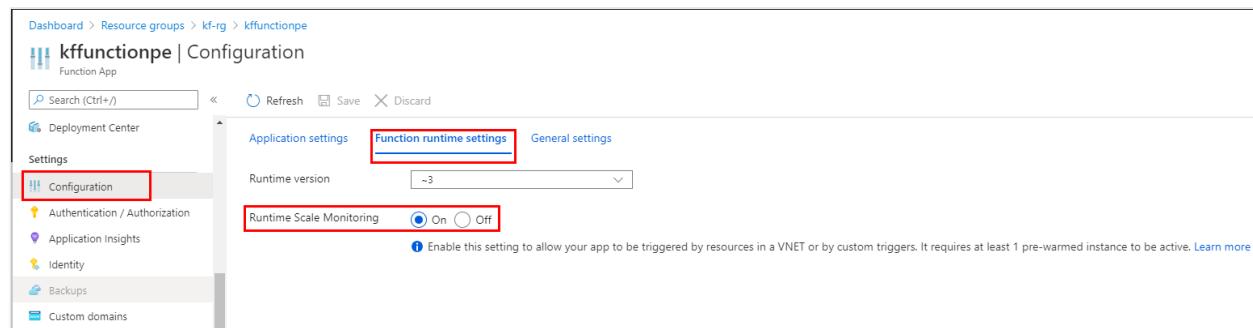
```
}
```

Property	Default	Description
hostName	n/a	(ignored if using connectionString) Hostname of the queue (Ex: 10.26.45.210)
userName	n/a	(ignored if using connectionString) Name to access the queue
password	n/a	(ignored if using connectionString) Password to access the queue

Enable Runtime Scaling

In order for the RabbitMQ trigger to scale out to multiple instances, the **Runtime Scale Monitoring** setting must be enabled.

In the portal, this setting can be found under **Configuration > Function runtime settings** for your function app.



In the CLI, you can enable **Runtime Scale Monitoring** by using the following command:

Azure CLI

```
az resource update -g <resource_group> -n <function_app_name>/config/web --  
set properties.functionsRuntimeScaleMonitoringEnabled=1 --resource-type  
Microsoft.Web/sites
```

Monitoring RabbitMQ endpoint

To monitor your queues and exchanges for a certain RabbitMQ endpoint:

- Enable the [RabbitMQ management plugin ↗](#)

- Browse to `http://{node-hostname}:15672` and log in with your user name and password.

Next steps

- [Send RabbitMQ messages from Azure Functions \(Output binding\)](#)

RabbitMQ output binding for Azure Functions overview

Article • 09/27/2023

ⓘ Note

The RabbitMQ bindings are only fully supported on **Premium and Dedicated** plans. Consumption is not supported.

Use the RabbitMQ output binding to send messages to a RabbitMQ queue.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

Isolated worker model

C#

```
[Function(nameof(RabbitMQFunction))]
[RabbitMQOutput(QueueName = "destinationQueue", ConnectionStringSetting
= "RabbitMQConnection")]
public static string Run([RabbitMQTrigger("queue",
ConnectionStringSetting = "RabbitMQConnection")] string item,
FunctionContext context)
{
    var logger = context.GetLogger(nameof(RabbitMQFunction));

    logger.LogInformation(item);

    var message = $"Output message created at {DateTime.Now}";
```

```
    return message;  
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the attribute to define the function. C# script instead uses a [function.json configuration file](#).

The attribute's constructor takes the following parameters:

Parameter	Description
QueueName	Name of the queue from which to receive messages.
HostName	Hostname of the queue, such as 10.26.45.210. Ignored when using <code>ConnectionStringSetting</code> .
UserNameSetting	Name of the app setting that contains the username to access the queue, such as <code>UserNameSetting: "%< UserNameFromSettings >%"</code> . Ignored when using <code>ConnectionStringSetting</code> .
PasswordSetting	Name of the app setting that contains the password to access the queue, such as <code>PasswordSetting: "%< PasswordFromSettings >%"</code> . Ignored when using <code>ConnectionStringSetting</code> .
ConnectionStringSetting	The name of the app setting that contains the RabbitMQ message queue connection string. The trigger won't work when you specify the connection string directly instead through an app setting. For example, when you have set <code>ConnectionStringSetting: "rabbitMQConnection"</code> , then in both the <code>local.settings.json</code> and in your function app you need a setting like <code>"RabbitMQConnection" : "< ActualConnectionString >"</code> .
Port	Gets or sets the port used. Defaults to 0, which points to the RabbitMQ client's default port setting of 5672.

Isolated worker model

In [C# class libraries](#), use the [RabbitMQTrigger](#) attribute.

Here's a `RabbitMQTrigger` attribute in a method signature for an isolated worker process library:

C#

```
[Function(nameof(RabbitMQFunction))]
[RabbitMQOutput(QueueName = "destinationQueue", ConnectionStringSetting
= "RabbitMQConnection")]
public static string Run([RabbitMQTrigger("queue",
ConnectionStringSetting = "RabbitMQConnection")] string item,
    FunctionContext context)
{
```

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file.

function.json property	Description
<code>type</code>	Must be set to <code>RabbitMQ</code> .
<code>direction</code>	Must be set to <code>out</code> .
<code>name</code>	The name of the variable that represents the queue in function code.
<code>queueName</code>	Name of the queue to send messages to.
<code>hostName</code>	Hostname of the queue, such as 10.26.45.210. Ignored when using <code>connectStringSetting</code> .
<code>userName</code>	Name of the app setting that contains the username to access the queue, such as <code>UserNameSetting: "< UserNameFromSettings >"</code> . Ignored when using <code>connectStringSetting</code> .
<code>password</code>	Name of the app setting that contains the password to access the queue, such as <code>UserNameSetting: "< UserNameFromSettings >"</code> . Ignored when using <code>connectStringSetting</code> .
<code>connectionStringSetting</code>	The name of the app setting that contains the RabbitMQ message queue connection string. The trigger won't work when you specify the connection string directly instead of through an app setting in <code>local.settings.json</code> . For example, when you have set <code>connectionStringSetting: "rabbitMQConnection"</code> then in both the <code>local.settings.json</code> and in your function app you need a setting like <code>"rabbitMQConnection" : "< ActualConnectionString >"</code> .
<code>port</code>	Gets or sets the Port used. Defaults to 0, which points to the RabbitMQ client's default port setting of <code>5672</code> .

When you're developing locally, add your application settings in the `local.settings.json` file in the `values` collection.

See the [Example section](#) for complete examples.

Usage

The parameter type supported by the RabbitMQ trigger depends on the Functions runtime version, the extension package version, and the C# modality used.

Isolated worker model

The RabbitMQ bindings currently support only string and serializable object types when running in an isolated worker process.

For a complete example, see C# [example](#).

Next steps

- Run a function when a RabbitMQ message is created (Trigger)

Azure Functions SendGrid bindings

Article • 03/31/2024

This article explains how to send email by using [SendGrid](#) bindings in Azure Functions. Azure Functions supports an output binding for SendGrid.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Azure Functions developer reference](#)
- [Create your first function](#)
- C# developer references:
 - [In-process class library](#)
 - [Isolated worker process class library](#)
 - [C# script](#)
- [Azure Functions triggers and bindings concepts](#)
- [Code and test Azure Functions locally](#)

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

The functionality of the extension varies depending on the extension version:

Functions v2.x+

Add the extension to your project by installing the [NuGet package](#), version 3.x.

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model**: Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- **In-process model**: Compiled C# function that runs in the same process as the Functions runtime.
- **C# script**: Used primarily when you create C# functions in the Azure portal.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

ⓘ Isolated worker model

We don't currently have an example for using the SendGrid binding in a function app running in an isolated worker process.

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the output binding. C# script instead uses a function.json configuration file.

ⓘ Isolated worker model

In [isolated worker process](#) function apps, the `SendGridOutputAttribute` supports the following parameters:

[\[+\] Expand table](#)

Attribute/annotation property	Description
<code>ApiKey</code>	The name of an app setting that contains your API key. If not set, the default app setting name is <code>AzureWebJobsSendGridApiKey</code> .
<code>To</code>	(Optional) The recipient's email address.
<code>From</code>	(Optional) The sender's email address.

Attribute/annotation	Description
property	
Subject	(Optional) The subject of the email.
Text	(Optional) The email content.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

host.json settings

This section describes the configuration settings available for this binding in versions 2.x and higher. Settings in the host.json file apply to all functions in a function app instance. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about function app configuration settings in versions 2.x and later versions, see [host.json reference for Azure Functions](#).

! Note

For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

JSON
<pre>{ "version": "2.0", "extensions": { "sendGrid": { "from": "Azure Functions <samples@functions.com>" } } }</pre>

[] Expand table

Property	Default	Description
from	n/a	The sender's email address across all functions.

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Service Bus bindings for Azure Functions

Article • 11/01/2023

Azure Functions integrates with [Azure Service Bus](#) via [triggers and bindings](#). Integrating with Service Bus allows you to build functions that react to and send queue or topic messages.

Action	Type
Run a function when a Service Bus queue or topic message is created	Trigger
Send Azure Service Bus messages	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project installing this [NuGet package](#).

The functionality of the extension varies depending on the extension version:

Extension 5.x+

This version introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

This version allows you to bind to types from [Azure.Messaging.ServiceBus](#).

Add the extension to your project by installing the [NuGet package](#), version 5.x.

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see binding type details for the mode and version.

Extension 5.x+

The isolated worker process supports parameter types according to the tables below.

Service Bus trigger

When you want the function to process a single message, the Service Bus trigger can bind to the following types:

Type	Description
<code>string</code>	The message as a string. Use when the message is simple text.
<code>byte[]</code>	The bytes of the message.
JSON serializable types	When an event contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
ServiceBusReceivedMessage ¹	The message object. When binding to <code>ServiceBusReceivedMessage</code> , you can optionally also include a parameter of type ServiceBusMessageActions ¹ to perform message settlement actions.

When you want the function to process a batch of messages, the Service Bus trigger can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is one of the single message types	An array of events from the batch. Each entry represents one event. When binding to <code>ServiceBusReceivedMessage[]</code> , you can optionally

Type	Description
	also include a parameter of type ServiceBusMessageActions ¹ to perform message settlement actions.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.ServiceBus 5.14.1 or later](#)¹ and the common dependencies for SDK type bindings.

Service Bus output binding

When you want the function to write a single message, the Service Bus output binding can bind to the following types:

Type	Description
<code>string</code>	The message as a string. Use when the message is simple text.
<code>byte[]</code>	The bytes of the message.
JSON serializable types	An object representing the message. Functions attempts to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write multiple messages, the Service Bus output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is one of the single message types	An array containing multiple message. Each entry represents one message.

For other output scenarios, create and use types from [Azure.Messaging.ServiceBus](#) directly.

host.json settings

This section describes the configuration settings available for this binding, which depends on the runtime and extension version.

Extension 5.x+

JSON

```
{
  "version": "2.0",
```

```

    "extensions": {
        "serviceBus": {
            "clientRetryOptions": {
                "mode": "exponential",
                "tryTimeout": "00:01:00",
                "delay": "00:00:00.80",
                "maxDelay": "00:01:00",
                "maxRetries": 3
            },
            "prefetchCount": 0,
            "transportType": "amqpWebSockets",
            "webProxy": "https://proxyserver:8080",
            "autoCompleteMessages": true,
            "maxAutoLockRenewalDuration": "00:05:00",
            "maxConcurrentCalls": 16,
            "maxConcurrentSessions": 8,
            "maxMessageBatchSize": 1000,
            "minMessageBatchSize": 1,
            "maxBatchWaitTime": "00:00:30",
            "sessionIdleTimeout": "00:01:00",
            "enableCrossEntityTransactions": false
        }
    }
}

```

The `clientRetryOptions` settings only apply to interactions with the Service Bus service. They don't affect retries of function executions. For more information, see [Retries](#).

Property	Default	Description
mode	Exponential	The approach to use for calculating retry delays. The default exponential mode retries attempts with a delay based on a back-off strategy where each attempt increases the wait duration before retrying. The <code>Fixed</code> mode retries attempts at fixed intervals with each delay having a consistent duration.
tryTimeout	00:01:00	The maximum duration to wait for an operation per attempt.
delay	00:00:00.80	The delay or back-off factor to apply between retry attempts.
maxDelay	00:01:00	The maximum delay to allow between retry attempts
maxRetries	3	The maximum number of retry attempts before considering the associated

Property	Default	Description
		operation to have failed.
prefetchCount	0	Gets or sets the number of messages that the message receiver can simultaneously request.
transportType	amqpTcp	The protocol and transport that is used for communicating with Service Bus. Available options: <code>amqpTcp</code> , <code>amqpWebSockets</code>
webProxy	n/a	The proxy to use for communicating with Service Bus over web sockets. A proxy cannot be used with the <code>amqpTcp</code> transport.
autoCompleteMessages	true	Determines whether or not to automatically complete messages after successful execution of the function and should be used in place of the <code>autoComplete</code> configuration setting.
maxAutoLockRenewalDuration	00:05:00	The maximum duration within which the message lock will be renewed automatically. This setting only applies for functions that receive a single message at a time.
maxConcurrentCalls	16	The maximum number of concurrent calls to the callback that should be initiated per scaled instance. By default, the Functions runtime processes multiple messages concurrently. This setting is used only when the <code>isSessionsEnabled</code> property or attribute on the trigger is set to <code>false</code> . This setting only applies for functions that receive a single message at a time.
maxConcurrentSessions	8	The maximum number of sessions that can be handled concurrently per scaled instance. This setting is used only when the <code>isSessionsEnabled</code> property or attribute on the trigger is set to <code>true</code> . This setting only applies for functions that receive a single message at a time.
maxMessageBatchSize	1000	The maximum number of messages that will be passed to each function call. This setting only applies for functions that receive a batch of messages.

Property	Default	Description
<code>minMessageBatchSize</code> ¹	1	<p>The minimum number of messages desired in a batch. The minimum applies only when the function is receiving multiple messages and must be less than <code>maxMessageBatchSize</code>.</p> <p>The minimum size isn't strictly guaranteed. A partial batch is dispatched when a full batch can't be prepared before the <code>maxBatchWaitTime</code> has elapsed.</p>
<code>maxBatchWaitTime</code> ¹	00:00:30	<p>The maximum interval that the trigger should wait to fill a batch before invoking the function. The wait time is only considered when <code>minMessageBatchSize</code> is larger than 1 and is ignored otherwise. If less than <code>minMessageBatchSize</code> messages were available before the wait time elapses, the function is invoked with a partial batch. The longest allowed wait time is 50% of the entity message lock duration, meaning the maximum allowed is 2 minutes and 30 seconds. Otherwise, you may get lock exceptions.</p> <p>NOTE: This interval is not a strict guarantee for the exact timing on which the function is invoked. There is a small margin of error due to timer precision.</p>
<code>sessionIdleTimeout</code>	n/a	The maximum amount of time to wait for a message to be received for the currently active session. After this time has elapsed, the session will be closed and the function will attempt to process another session.
<code>enableCrossEntityTransactions</code>	false	Whether or not to enable transactions that span multiple entities on a Service Bus namespace.

¹ Using `minMessageBatchSize` and `maxBatchWaitTime` requires v5.10.0² of the `Microsoft.Azure.WebJobs.Extensions.ServiceBus` package, or a later version.

Next steps

- Run a function when a Service Bus queue or topic message is created (Trigger)

- Send Azure Service Bus messages from Azure Functions (Output binding)

Azure Service Bus trigger for Azure Functions

Article • 10/16/2023

Use the Service Bus trigger to respond to messages from a Service Bus queue or topic. Starting with extension version 3.1.0, you can trigger on a session-enabled queue or topic.

For information on setup and configuration details, see the [overview](#).

Service Bus scaling decisions for the Consumption and Premium plans are made based on target-based scaling. For more information, see [Target-based scaling](#).

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

This code defines and initializes the `ILogger`:

```
C#  
  
private readonly ILogger<ServiceBusReceivedMessageFunctions> _logger;  
  
public ServiceBusReceivedMessageFunctions(ILogger<ServiceBusReceivedMessageFunctions> logger)  
{  
    _logger = logger;  
}
```

This example shows a [C# function](#) that receives a single Service Bus queue message and writes it to the logs:

```
C#  
  
[Function(nameof(ServiceBusReceivedMessageFunction))]  
[ServiceBusOutput("outputQueue", Connection = "ServiceBusConnection")]  
public string ServiceBusReceivedMessageFunction(  
    [ServiceBusTrigger("queue", Connection = "ServiceBusConnection")] ServiceBusReceivedMessage  
    message)  
{  
    _logger.LogInformation("Message ID: {id}", message.MessageId);  
    _logger.LogInformation("Message Body: {body}", message.Body);  
    _logger.LogInformation("Message Content-Type: {contentType}", message.ContentType);  
  
    var outputMessage = $"Output message created at {DateTime.Now}";  
    return outputMessage;  
}
```

This example shows a [C# function](#) that receives multiple Service Bus queue messages in a single batch and writes each to the logs:

```
C#  
  
[Function(nameof(ServiceBusReceivedMessageBatchFunction))]  
public void ServiceBusReceivedMessageBatchFunction(  
    [ServiceBusTrigger("queue", Connection = "ServiceBusConnection", IsBatched = true)]  
    ServiceBusReceivedMessage[] messages)  
{  
    foreach (ServiceBusReceivedMessage message in messages)  
    {  
        _logger.LogInformation("Message ID: {id}", message.MessageId);  
        _logger.LogInformation("Message Body: {body}", message.Body);  
        _logger.LogInformation("Message Content-Type: {contentType}", message.ContentType);  
    }  
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the [ServiceBusTriggerAttribute](#) attribute to define the function trigger. C# script instead uses a function.json configuration file as described in the [C# scripting guide](#).

Isolated worker model

The following table explains the properties you can set using this trigger attribute:

Property	Description
QueueName	Name of the queue to monitor. Set only if monitoring a queue, not for a topic.
TopicName	Name of the topic to monitor. Set only if monitoring a topic, not for a queue.
SubscriptionName	Name of the subscription to monitor. Set only if monitoring a topic, not for a queue.
Connection	The name of an app setting or setting collection that specifies how to connect to Service Bus. See Connections .
IsBatched	Messages are delivered in batches. Requires an array or collection type.
IsSessionsEnabled	<code>true</code> if connecting to a session-aware queue or subscription. <code>false</code> otherwise, which is the default value.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

See the [Example section](#) for complete examples.

Usage

The following parameter types are supported by all C# modalities and extension versions:

Type	Description
System.String	Use when the message is simple text.

Type	Description
byte[]	Use for binary data messages.
Object	When a message contains JSON, Functions tries to deserialize the JSON data into known plain-old CLR object type.

Messaging-specific parameter types contain additional message metadata. The specific types supported by the Service Bus trigger depend on the Functions runtime version, the extension package version, and the C# modality used.

Extension 5.x and higher

When you want the function to process a single message, the Service Bus trigger can bind to the following types:

Type	Description
string	The message as a string. Use when the message is simple text.
byte[]	The bytes of the message.
JSON serializable types	When an event contains JSON data, Functions tries to deserialize the JSON data into a plain-old CLR object (POCO) type.
ServiceBusReceivedMessage ¹	The message object.

When you want the function to process a batch of messages, the Service Bus trigger can bind to the following types:

Type	Description
T[] where T is one of the single message types	An array of events from the batch. Each entry represents one event.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.ServiceBus 5.12.0 or later](#) and the [common dependencies for SDK type bindings](#).

ⓘ Note

The isolated process model does not yet support message settlement scenarios for Service Bus triggers.

When the `Connection` property isn't defined, Functions looks for an app setting named `AzureWebJobsServiceBus`, which is the default name for the Service Bus connection string. You can also set the `Connection` property to specify the name of an application setting that contains the Service Bus connection string to use.

For a complete example, see [the examples section](#).

Connections

The `connection` property is a reference to environment configuration which specifies how the app should connect to Service Bus. It may specify:

- The name of an application setting containing a [connection string](#)

- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Get the management credentials](#). The connection string must be for a Service Bus namespace, not limited to a specific queue or topic.

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name. For example, if you set `connection` to "MyServiceBus", the Functions runtime looks for an app setting that is named "AzureWebJobsMyServiceBus". If you leave `connection` empty, the Functions runtime uses the default Service Bus connection string in the app setting that is named "AzureWebJobsServiceBus".

Identity-based connections

If you are using [version 5.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To do this, you would define settings under a common prefix which maps to the `connection` property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Fully Qualified Namespace	<code><CONNECTION_NAME_PREFIX>_fullyQualifiedNamespace</code>	The fully qualified Service Bus namespace.	<code><service_bus_namespace>.servicebus.windows.net</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

ⓘ Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `<CONNECTION_NAME_PREFIX>:fullyQualifiedNamespace`.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You'll need to create a role assignment that provides access to your topics and queues at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Service Bus extension in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles
Trigger ¹	Azure Service Bus Data Receiver , Azure Service Bus Data Owner
Output binding	Azure Service Bus Data Sender

¹ For triggering from Service Bus topics, the role assignment needs to have effective scope over the Service Bus subscription resource. If only the topic is included, an error will occur. Some clients, such as the Azure portal, don't expose the Service Bus subscription resource as a scope for role assignment. In such cases, the Azure CLI may be used instead. To learn more, see [Azure built-in roles for Azure Service Bus](#).

Poison messages

Poison message handling can't be controlled or configured in Azure Functions. Service Bus handles poison messages itself.

PeekLock behavior

The Functions runtime receives a message in [PeekLock mode](#). It calls `Complete` on the message if the function finishes successfully, or calls `Abandon` if the function fails. If the function runs longer than the `PeekLock` timeout, the lock is automatically renewed as long as the function is running.

The `maxAutoRenewDuration` is configurable in `host.json`, which maps to `ServiceBusProcessor.MaxAutoLockRenewalDuration`. The default value of this setting is 5 minutes.

Message metadata

Messaging-specific types let you easily retrieve [metadata as properties of the object](#). These properties depend on the Functions runtime version, the extension package version, and the C# modality used.

Extension 5.x and higher

These properties are members of the `ServiceBusReceivedMessage` class.

Property	Type	Description
ApplicationProperties	ApplicationProperties	Properties set by the sender.
ContentType	string	A content type identifier utilized by the sender and receiver for application-specific logic.
CorrelationId	string	The correlation ID.
DeliveryCount	Int32	The number of deliveries.
EnqueuedTime	DateTime	The enqueued time in UTC.
ScheduledEnqueueTimeUtc	DateTime	The scheduled enqueued time in UTC.
ExpiresAt	DateTime	The expiration time in UTC.
MessageId	string	A user-defined value that Service Bus can use to identify duplicate messages, if enabled.
ReplyTo	string	The reply to queue address.
Subject	string	The application-specific label which can be used in place of the Label metadata property.
To	string	The send to address.

Next steps

- [Send Azure Service Bus messages from Azure Functions \(Output binding\)](#)

Azure Service Bus output binding for Azure Functions

Article • 01/15/2024

Use Azure Service Bus output binding to send queue or topic messages.

For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

This code defines and initializes the `ILogger`:

```
C#  
  
private readonly ILogger<ServiceBusReceivedMessageFunctions> _logger;  
  
public ServiceBusReceivedMessageFunctions(ILogger<ServiceBusReceivedMessageFunctions> logger)  
{  
    _logger = logger;  
}
```

This example shows a [C# function](#) that receives a message and writes it to a second queue:

```
C#  
  
[Function(nameof(ServiceBusReceivedMessageFunction))]  
[ServiceBusOutput("outputQueue", Connection = "ServiceBusConnection")]  
public string ServiceBusReceivedMessageFunction(  
    [ServiceBusTrigger("queue", Connection = "ServiceBusConnection")] ServiceBusReceivedMessage  
    message)  
{  
    _logger.LogInformation("Message ID: {id}", message.MessageId);  
    _logger.LogInformation("Message Body: {body}", message.Body);  
    _logger.LogInformation("Message Content-Type: {contentType}", message.ContentType);  
  
    var outputMessage = $"Output message created at {DateTime.Now}";  
    return outputMessage;  
}
```

This example uses an HTTP trigger with an `OutputType` object to both send an HTTP response and write the output message.

C#

```
[Function("HttpSendMsg")]
public async Task<OutputType> Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req, FunctionContext context)
{
    _logger.LogInformation($"C# HTTP trigger function processed a request for
{context.InvocationId}.");

    HttpResponseMessage response = req.CreateResponse(HttpStatusCode.OK);
    await response.WriteStringAsync("HTTP response: Message sent");

    return new OutputType()
    {
        OutputEvent = "MyMessage",
        HttpResponseMessage = response
    };
}
```

This code defines the multiple output type `OutputType`, which includes the Service Bus output binding definition on `OutputEvent`:

C#

```
public class OutputType
{
    [ServiceBusOutput("TopicOrQueueName", Connection = "ServiceBusConnection")]
    public string OutputEvent { get; set; }

    public HttpResponseMessage HttpResponseMessage { get; set; }
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the output binding. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated worker model

In [C# class libraries](#), use the [ServiceBusOutputAttribute](#) to define the queue or topic written to by the output.

The following table explains the properties you can set using the attribute:

[Expand table](#)

Property	Description
EntityType	Sets the entity type as either <code>Queue</code> for sending messages to a queue or <code>Topic</code> when sending messages to a topic.
QueueOrTopicName	Name of the topic or queue to send messages to. Use <code>EntityType</code> to set the destination type.
Connection	The name of an app setting or setting collection that specifies how to connect to Service Bus. See Connections .

See the [Example section](#) for complete examples.

Usage

The following output parameter types are supported by all C# modalities and extension versions:

[Expand table](#)

Type	Description
<code>System.String</code>	Use when the message to write is simple text. When the parameter value is null when the function exits, Functions doesn't create a message.
<code>byte[]</code>	Use for writing binary data messages. When the parameter value is null when the function exits, Functions doesn't create a message.
<code>Object</code>	When a message contains JSON, Functions serializes the object into a JSON message payload. When the parameter value is null when the function exits, Functions creates a message with a null object.

Messaging-specific parameter types contain additional message metadata. The specific types supported by the output binding depend on the Functions runtime version, the extension package version, and the C# modality used.

Extension 5.x and higher

When you want the function to write a single message, the Service Bus output binding can bind to the following types:

[Expand table](#)

Type	Description
<code>string</code>	The message as a string. Use when the message is simple text.
<code>byte[]</code>	The bytes of the message.
JSON serializable types	An object representing the message. Functions attempts to serialize a plain-old CLR object (POCO) type into JSON data.

When you want the function to write multiple messages, the Service Bus output binding can bind to the following types:

[Expand table](#)

Type	Description
<code>T[]</code> where <code>T</code> is one of the single message types	An array containing multiple message. Each entry represents one message.

For other output scenarios, create and use types from [Azure.Messaging.ServiceBus](#) directly.

In Azure Functions 1.x, the runtime creates the queue if it doesn't exist and you have set `accessRights` to `manage`. In Azure Functions version 2.x and higher, the queue or topic must already exist; if you specify a queue or topic that doesn't exist, the function fails.

For a complete example, see [the examples section](#).

Connections

The `connection` property is a reference to environment configuration which specifies how the app should connect to Service Bus. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string, follow the steps shown at [Get the management credentials](#). The connection string must be for a Service Bus namespace, not limited to a specific queue or topic.

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name. For example, if you set `connection` to "MyServiceBus", the Functions runtime looks for an app setting that is named "AzureWebJobsMyServiceBus". If you leave `connection` empty, the Functions runtime uses the default Service Bus connection string in the app setting that is named "AzureWebJobsServiceBus".

Identity-based connections

If you are using [version 5.x or higher of the extension](#), instead of using a connection string with a secret, you can have the app use an [Microsoft Entra identity](#). To do this, you would define settings under a common prefix which maps to the `connection` property in the trigger and binding configuration.

In this mode, the extension requires the following properties:

[Expand table](#)

Property	Environment variable template	Description	Example value
Fully Qualified Namespace	<code><CONNECTION_NAME_PREFIX>_fullyQualifiedNamespace</code>	The fully qualified Service Bus namespace.	<code><service_bus_namespace>.servicebus.windows.net</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `<CONNECTION_NAME_PREFIX>:fullyQualifiedNamespace`.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

i Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You'll need to create a role assignment that provides access to your topics and queues at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Service Bus extension in normal operation. Your application may require additional permissions based on the code you write.

[] [Expand table](#)

Binding type	Example built-in roles
Trigger ¹	Azure Service Bus Data Receiver , Azure Service Bus Data Owner
Output binding	Azure Service Bus Data Sender

¹ For triggering from Service Bus topics, the role assignment needs to have effective scope over the Service Bus subscription resource. If only the topic is included, an error will occur. Some clients, such as the Azure portal, don't expose the Service Bus subscription resource as a scope for role assignment. In such cases, the Azure CLI may be used instead. To learn more, see [Azure built-in roles for Azure Service Bus](#).

Exceptions and return codes

[] [Expand table](#)

Binding	Reference
Service Bus	Service Bus Error Codes
Service Bus	Service Bus Limits

Next steps

- Run a function when a Service Bus queue or topic message is created (Trigger)

Migrate function apps from Azure Service Bus extension version 4.x to version 5.x

Article • 01/14/2024

This article highlights considerations for upgrading your existing Azure Functions applications that use the Azure Service Bus extension version 4.x to use the newer [extension version 5.x](#). Migrating from version 4.x to version 5.x of the Azure Service Bus extension has breaking changes for your application.

ⓘ Important

On March 31, 2025 the Azure Service Bus extension version 4.x will be retired. The extension and all applications using the extension will continue to function, but Azure Service Bus will cease to provide further maintenance and support for this extension. We recommend migrating to the latest version 5.x of the extension.

This article walks you through the process of migrating your function app to run on version 5.x of the Azure Service Bus extension. Because project upgrade instructions are language dependent, make sure to choose your development language from the selector at the [top of the article](#).

Update the extension version

.NET Functions uses extensions that are installed in the project as NuGet packages. Depending on your Functions process model, the NuGet package to update varies.

[+] Expand table

Functions process model	Azure Service Bus extension	Recommended version
In-process model	Microsoft.Azure.WebJobs.Extensions.ServiceBus	>= 5.13.4
Isolated worker model	Microsoft.Azure.Functions.Worker.Extensions.ServiceBus	>= 5.14.1

Update your `.csproj` project file to use the latest extension version for your process model. The following `.csproj` file uses version 5 of the Azure Service Bus extension.

Isolated worker model

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
    <OutputType>Exe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Azure.Functions.Worker" Version="1.21.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Extensions.ServiceBus" Version="5.16.0" />
    <PackageReference Include="Microsoft.Azure.Functions.Worker.Sdk" Version="1.16.4" />
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
</Project>
```

Modify your function code

The Azure Functions Azure Service Bus extension version 5 is built on top of the `Azure.Messaging.ServiceBus` SDK version 3, which removed support for the `Message` class. Instead, use the `ServiceBusReceivedMessage` type to receive message metadata from Service Bus Queues and Subscriptions.

Next steps

- Run a function when a Service Bus queue or topic message is created (Trigger)
- Send Azure Service Bus messages from Azure Functions (Output binding)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

SignalR Service bindings for Azure Functions

Article • 04/02/2024

This set of articles explains how to authenticate and send real-time messages to clients connected to [Azure SignalR Service](#) by using SignalR Service bindings in Azure Functions. Azure Functions runtime version 2.x and higher supports input and output bindings for SignalR Service.

[+] [Expand table](#)

Action	Type
Handle messages from SignalR Service	Trigger binding
Return the service endpoint URL and access token	Input binding
Send SignalR Service messages and manage groups	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing this [NuGet package](#).

Connection string settings

Add the `AzureSignalRConnectionString` key to the `host.json` file that points to the application setting with your connection string. For local development, this value may exist in the `local.settings.json` file.

For details on how to configure and use SignalR Service and Azure Functions together, refer to [Azure Functions development and configuration with Azure SignalR Service](#).

Next steps

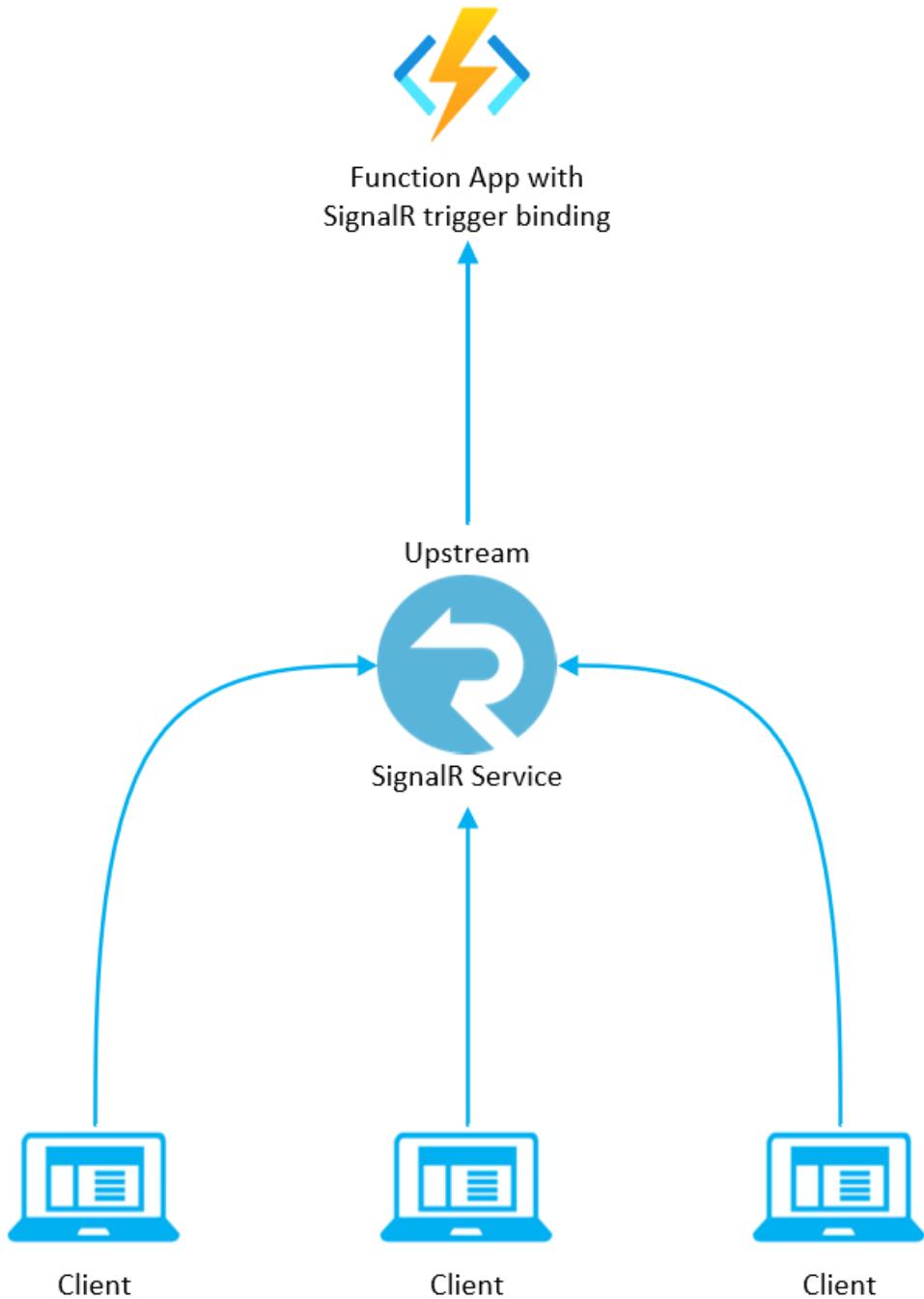
- Handle messages from SignalR Service (Trigger binding)
- Return the service endpoint URL and access token (Input binding)
- Send SignalR Service messages (Output binding)

SignalR Service trigger binding for Azure Functions

Article • 04/02/2024

Use the *SignalR* trigger binding to respond to messages sent from Azure SignalR Service. When function is triggered, messages passed to the function is parsed as a json object.

In SignalR Service serverless mode, SignalR Service uses the [Upstream](#) feature to send messages from client to Function App. And Function App uses SignalR Service trigger binding to handle these messages. The general architecture is shown below:



For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C#

functions running on LTS and non-LTS versions .NET and the .NET Framework.

- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime.
- **C# script:** Used primarily when you create C# functions in the Azure portal.

ⓘ Important

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

The following sample shows a C# function that receives a message event from clients and logs the message content.

C#

```
[Function(nameof(OnClientMessage))]
public static void OnClientMessage(
    [SignalRTrigger("Hub", "messages", "sendMessage", "content",
ConnectionStringSetting = "SignalRConnection")]
    SignalRIExecutionContext invocationContext, string content,
FunctionContext functionContext)
{
    var logger = functionContext.GetLogger(nameof(OnClientMessage));
    logger.LogInformation("Connection {connectionId} sent a message.
Message content: {content}", invocationContext.ConnectionId, content);
}
```

ⓘ Important

Class based model of SignalR Service bindings in C# isolated worker doesn't optimize how you write SignalR triggers due to the limitation of C# worker model. For more information about class based model, see [Class based model](#).

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the `SignalRTrigger` attribute to define the function. C# script instead uses a [function.json configuration file](#).

The following table explains the properties of the `SignalRTrigger` attribute.

[Expand table](#)

Attribute property	Description
<code>HubName</code>	This value must be set to the name of the SignalR hub for the function to be triggered.
<code>Category</code>	This value must be set as the category of messages for the function to be triggered. The category can be one of the following values: <ul style="list-style-type: none"> • connections: Including <i>connected</i> and <i>disconnected</i> events • messages: Including all other events except those in <i>connections</i> category
<code>Event</code>	This value must be set as the event of messages for the function to be triggered. For <i>messages</i> category, event is the <i>target</i> in invocation message that clients send. For <i>connections</i> category, only <i>connected</i> and <i>disconnected</i> is used.
<code>ParameterNames</code>	(Optional) A list of names that binds to the parameters.
<code>ConnectionStringSetting</code>	The name of the app setting that contains the SignalR Service connection string, which defaults to <code>AzureSignalRConnectionString</code> .

See the [Example section](#) for complete examples.

Usage

Payloads

The trigger input type is declared as either `InvocationContext` or a custom type. If you choose `InvocationContext`, you get full access to the request content. For a custom type, the runtime tries to parse the JSON request body to set the object properties.

InvocationContext

`InvocationContext` contains all the content in the message sent from a SignalR service, which includes the following properties:

Property	Description
Arguments	Available for <i>messages</i> category. Contains <i>arguments</i> in invocation message ↗
Error	Available for <i>disconnected</i> event. It can be Empty if the connection closed with no error, or it contains the error messages.
Hub	The hub name that the message belongs to.
Category	The category of the message.
Event	The event of the message.
ConnectionId	The connection ID of the client that sends the message.
UserId	The user identity of the client that sends the message.
Headers	The headers of the request.
Query	The query of the request when clients connect to the service.
Claims	The claims of the client.

Using ParameterNames

The property `ParameterNames` in `SignalRTrigger` lets you bind arguments of invocation messages to the parameters of functions. You can use the name you defined as part of [binding expressions](#) in other binding or as parameters in your code. That gives you a more convenient way to access arguments of `InvocationContext`.

Say you have a JavaScript SignalR client trying to invoke method `broadcast` in Azure Function with two arguments `message1`, `message2`.

JavaScript

```
await connection.invoke("broadcast", message1, message2);
```

After you set `parameterNames`, the names you defined correspond to the arguments sent on the client side.

CS

```
[SignalRTrigger(parameterNames: new string[] {"arg1, arg2"})]
```

Then, the `arg1` contains the content of `message1`, and `arg2` contains the content of `message2`.

ParameterNames considerations

For the parameter binding, the order matters. If you're using `ParameterNames`, the order in `ParameterNames` matches the order of the arguments you invoke in the client. If you're using attribute `[SignalRParameter]` in C#, the order of arguments in Azure Function methods matches the order of arguments in clients.

`ParameterNames` and attribute `[SignalRParameter]` **cannot** be used at the same time, or you'll get an exception.

SignalR Service integration

SignalR Service needs a URL to access Function App when you're using SignalR Service trigger binding. The URL should be configured in **Upstream Settings** on the SignalR Service side.

The screenshot shows the Azure SignalR Service settings page. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Keys, Quickstart, Scale, Settings (which is selected), CORS, Identity, Private endpoint connections, Network access control, and Properties. The main area has tabs for Save and Discard. A note says "Choose Default mode when all the hubs have hub servers, choose Serverless mode if not. It usually takes up to 30 seconds to take effect." Below that is a "Service Mode" section with buttons for Default, Serverless (which is selected), and Classic. Under "Upstream Settings", it says: "Upstreams allow external services to be notified when certain events for certain hubs happen. When the specified events happen, we'll send a POST request to the first matching URL pattern with the parameters evaluated. There are three supported parameters: {hub}, {event}, and {category}. The matching rule supports three formats. Take the Event Rules as an example: • Use asterisk(*) to match any event. • Use comma(,) to join multiple events, for example, connect,disconnect matches events connect and disconnect. • Use the full event name to match the event, for example, connect matches connect event." There are buttons for Upstream URL Pattern, Hub Rules, Event Rules, and Category Rules. A text input field says "Add an upstream URL pattern".

When using SignalR Service trigger, the URL can be simple and formatted as follows:

The screenshot shows the Azure Function trigger configuration. It has a tab for HTTP. The URL template is shown as: <Function_App_URL>/runtime/webhooks/signalr?code=<API_KEY>

The `Function_App_URL` can be found on Function App's Overview page and the `API_KEY` is generated by Azure Function. You can get the `API_KEY` from `signalr_extension` in the

App keys blade of Function App.

The screenshot shows the 'App keys' blade for an Azure Function App. On the left, there's a sidebar with 'Functions' selected, followed by 'App keys' (which is highlighted in yellow), 'App files', and 'Proxies'. Below that is the 'Deployment' section with 'Deployment slots' and 'Deployment Center'. The main area is titled 'System keys' and contains a note: 'System keys are automatically managed by the Function runtime. System Kkeys provide granular access to functions runtime features.' There are two buttons at the top: 'Show values' and 'Filter'. A table below lists one key: 'Name' (signalr_extension) and 'Value' (default). To the right of the value is a note: 'Hidden value. Click to show value'. At the bottom right are 'Renew key value' and a trash bin icon.

If you want to use more than one Function App together with one SignalR Service, upstream can also support complex routing rules. Find more details at [Upstream settings](#).

Step-by-step sample

You can follow the sample in GitHub to deploy a chat room on Function App with SignalR Service trigger binding and upstream feature: [Bidirectional chat room sample ↗](#)

Next steps

- [Azure Functions development and configuration with Azure SignalR Service](#)
- [SignalR Service Trigger binding sample ↗](#)
- [SignalR Service Trigger binding sample in isolated worker process ↗](#)

SignalR Service input binding for Azure Functions

Article • 03/21/2024

Before a client can connect to Azure SignalR Service, it must retrieve the service endpoint URL and a valid access token. The *SignalRConnectionInfo* input binding produces the SignalR Service endpoint URL and a valid token that are used to connect to the service. The token is time-limited and can be used to authenticate a specific user to a connection. Therefore, you shouldn't cache the token or share it between clients. Usually you use *SignalRConnectionInfo* with HTTP trigger for clients to retrieve the connection information.

For more information on how to use this binding to create a "negotiate" function that is compatible with a SignalR client SDK, see [Azure Functions development and configuration with Azure SignalR Service](#). For information on setup and configuration details, see the [overview](#).

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

Isolated worker model

The following example shows a [C# function](#) that acquires SignalR connection information using the input binding and returns it over HTTP.

```
C#  
  
[Function(nameof(Negotiate))]  
public static string  
Negotiate([HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequestData  
req,  
          [SignalRConnectionInfoInput(HubName = "serverless")] string  
connectionInfo)  
{
```

```
// The serialization of the connection info object is done by the
framework. It should be camel case. The SignalR client respects the
camel case response only.
return connectionInfo;
}
```

Usage

Authenticated tokens

When an authenticated client triggers the function, you can add a user ID claim to the generated token. You can easily add authentication to a function app using [App Service Authentication](#).

App Service authentication sets HTTP headers named `x-ms-client-principal-id` and `x-ms-client-principal-name` that contain the authenticated user's client principal ID and name, respectively.

You can set the `UserId` property of the binding to the value from either header using a [binding expression](#): `{headers.x-ms-client-principal-id}` or `{headers.x-ms-client-principal-name}`.

Isolated worker model

```
cs

[Function("Negotiate")]
public static string
Negotiate([HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequestData
req,
          [SignalRConnectionInfoInput(HubName = "hubName1", UserId = "
{headers.x-ms-client-principal-id}")] string connectionInfo)
{
    // The serialization of the connection info object is done by the
    framework. It should be camel case. The SignalR client respects the
    camel case response only.
    return connectionInfo;
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to define the function. C# script instead uses a `function.json` configuration file.

Isolated worker model

The following table explains the properties of the `SignalRConnectionInfoInput` attribute:

[+] [Expand table](#)

Attribute property	Description
<code>HubName</code>	Required. The hub name.
<code>ConnectionStringSetting</code>	The name of the app setting that contains the SignalR Service connection string, which defaults to <code>AzureSignalRConnectionString</code> .
<code>UserId</code>	Optional. The user identifier of a SignalR connection. You can use a binding expression to bind the value to an HTTP request header or query.
<code>IdToken</code>	Optional. A JWT token whose claims will be added to the user claims. It should be used together with <code>ClaimTypeList</code> . You can use a binding expression to bind the value to an HTTP request header or query.
<code>ClaimTypeList</code>	Optional. A list of claim types, which filter the claims in <code>IdToken</code> .

Binding expressions for HTTP trigger

It's a common scenario that the values of some attributes of SignalR input binding come from HTTP requests. Therefore, we show how to bind values from HTTP requests to SignalR input binding attributes via [binding expression](#).

[+] [Expand table](#)

HTTP metadata type	Binding expression format	Description	Example
HTTP request query	<code>{query.QUERY_PARAMETER_NAME}</code>	Binds the value of corresponding query parameter to an attribute	<code>{query.userName}</code>

HTTP metadata type	Binding expression format	Description	Example
HTTP request header	{headers.HEADER_NAME}	Binds the value of a header to an attribute	{headers.token}

Next steps

- Handle messages from SignalR Service (Trigger binding)
- Send SignalR Service messages (Output binding)

SignalR Service output binding for Azure Functions

Article • 03/21/2024

Use the *SignalR* output binding to send one or more messages using Azure SignalR Service. You can broadcast a message to:

- All connected clients
- Connected clients in a specified group
- Connected clients authenticated to a specific user

The output binding also allows you to manage groups, such as adding a client or user to a group, removing a client or user from a group.

For information on setup and configuration details, see the [overview](#).

Example

Broadcast to all clients

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

Isolated worker model

The following example shows a function that sends a message using the output binding to all connected clients. The *newMessage* is the name of the method to be invoked on each client.

C#

```
[Function(nameof(BroadcastToAll))]
[SignalROutput(HubName = "chat", ConnectionStringSetting =
"SignalRConnection")]
public static SignalRMessageAction
```

```
BroadcastToAll([HttpTrigger(AuthorizationLevel.Anonymous, "post")]
HttpRequestData req)
{
    using var bodyReader = new StreamReader(req.Body);
    return new SignalRMessageAction("newMessage")
    {
        // broadcast to all the connected clients without specifying any
        connection, user or group.
        Arguments = new[] { bodyReader.ReadToEnd() },
    };
}
```

Send to a user

You can send a message only to connections that have been authenticated to a user by setting the *user ID* in the SignalR message.

Isolated worker model

```
C#
[Function(nameof(SendToUser))]
[SignalROutput(HubName = "chat", ConnectionStringSetting =
"SignalRConnection")]
public static SignalRMessageAction
SendToUser([HttpTrigger(AuthorizationLevel.Anonymous, "post")]
HttpRequestData req)
{
    using var bodyReader = new StreamReader(req.Body);
    return new SignalRMessageAction("newMessage")
    {
        Arguments = new[] { bodyReader.ReadToEnd() },
        UserId = "userToSend",
    };
}
```

Send to a group

You can send a message only to connections that have been added to a group by setting the *group name* in the SignalR message.

Isolated worker model

```
C#
```

```
[Function(nameof(SendToGroup))]
[SignalROutput(HubName = "chat", ConnectionStringSetting =
"SignalRConnection")]
public static SignalRMessageAction
SendToGroup([HttpTrigger(AuthorizationLevel.Anonymous, "post")]
HttpRequestData req)
{
    using var bodyReader = new StreamReader(req.Body);
    return new SignalRMessageAction("newMessage")
    {
        Arguments = new[] { bodyReader.ReadToEnd() },
        GroupName = "groupToSend"
    };
}
```

Group management

SignalR Service allows users or connections to be added to groups. Messages can then be sent to a group. You can use the `SignalR` output binding to manage groups.

Isolated worker model

Specify `SignalRGroup ActionType` to add or remove a member. The following example removes a user from a group.

C#

```
[Function(nameof(RemoveFromGroup))]
[SignalROutput(HubName = "chat", ConnectionStringSetting =
"SignalRConnection")]
public static SignalRGroupAction
RemoveFromGroup([HttpTrigger(AuthorizationLevel.Anonymous, "post")]
HttpRequestData req)
{
    return new SignalRGroupAction(SignalRGroup ActionType.Remove)
    {
        GroupName = "group1",
        UserId = "user1"
    };
}
```

 Note

In order to get the `ClaimsPrincipal` correctly bound, you must have configured the authentication settings in Azure Functions.

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attribute to define the function. C# script instead uses a [function.json configuration file](#).

Isolated worker model

The following table explains the properties of the `SignalROutput` attribute.

 [Expand table](#)

Attribute property	Description
<code>HubName</code>	This value must be set to the name of the SignalR hub for which the connection information is generated.
<code>ConnectionStringSetting</code>	The name of the app setting that contains the SignalR Service connection string, which defaults to <code>AzureSignalRConnectionString</code> .

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

Next steps

- [Handle messages from SignalR Service \(Trigger binding\)](#)
- [Return the service endpoint URL and access token \(Input binding\)](#)

Azure Tables bindings for Azure Functions

Article • 03/31/2024

Azure Functions integrates with [Azure Tables](#) via [triggers and bindings](#). Integrating with Azure Tables allows you to build functions that read and write data using [Azure Cosmos DB for Table](#) and [Azure Table Storage](#).

[+] [Expand table](#)

Action	Type
Read table data in a function	Input binding
Allow a function to write table data	Output binding

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

The process for installing the extension varies depending on the extension version:

Azure Tables extension

This version introduces the ability to [connect using an identity instead of a secret](#). For a tutorial on configuring your function apps with managed identities, see the [creating a function app with identity-based connections tutorial](#).

This version allows you to bind to types from [Azure.Data.Tables](#). It also introduces the ability to use Azure Cosmos DB for Table.

This extension is available by installing the [Microsoft.Azure.Functions.Worker.Extensions.Tables NuGet package](#) into a project using version 5.x or higher of the extensions for [blobs](#) and [queues](#).

Using the .NET CLI:

```
.NET CLI

# Install the Azure Tables extension
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Tables --version 1.0.0

# Update the combined Azure Storage extension (to a version which no longer includes Azure Tables)
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Storage --version 5.0.0
```

 **Note**

Azure Blobs, Azure Queues, and Azure Tables now use separate extensions and are referenced individually. For example, to use the triggers and bindings for all three services in your .NET isolated-process app, you should add the following packages to your project:

- [Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs](#)
- [Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues](#)
- [Microsoft.Azure.Functions.Worker.Extensions.Tables](#)

Previously, the extensions shipped together as

[Microsoft.Azure.Functions.Worker.Extensions.Storage, version 4.x](#). This same package also has a [5.x version](#), which references the split packages for blobs and queues only. When upgrading your package references from older versions, you may therefore need to additionally reference the new [Microsoft.Azure.Functions.Worker.Extensions.Tables](#) NuGet package. Also, when referencing these newer split packages, make sure you are not referencing an older version of the combined storage package, as this will result in conflicts from two definitions of the same bindings.

If you're writing your application using F#, you must also configure this extension as part of the app's [startup configuration](#). In the call to `ConfigureFunctionsWorkerDefaults()` or `ConfigureFunctionsWebApplication()`, add a delegate that takes an `IFunctionsWorkerApplication` parameter. Then within the body of that delegate, call `ConfigureTablesExtension()` on the object:

```
F#
```

```
let hostBuilder = new HostBuilder()
hostBuilder.ConfigureFunctionsWorkerDefaults(fun (context:
HostBuilderContext) (appBuilder: IFunctionsWorkerApplicationBuilder) ->
    appBuilder.ConfigureTablesExtension() |> ignore
) |> ignore
```

Binding types

The binding types supported for .NET depend on both the extension version and C# execution mode, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see binding type details for the mode and version.

Azure Tables extension

The isolated worker process supports parameter types according to the tables below. Support for binding to types from [Azure.Data.Tables](#) is in preview.

Azure Tables input binding

When working with a single table entity, the Azure Tables input binding can bind to the following types:

 Expand table

Type	Description
A JSON serializable type that implements ITableEntity	Functions attempts to deserialize the entity into a plain-old CLR object (POCO) type. The type must implement ITableEntity or have a string <code>RowKey</code> property and a string <code>PartitionKey</code> property.
TableEntity ¹	The entity as a dictionary-like type.

When working with multiple entities from a query, the Azure Tables input binding can bind to the following types:

[] [Expand table](#)

Type	Description
<code>IEnumerable<T> where T implements ITableEntity</code>	An enumeration of entities returned by the query. Each entry represents one entity. The type <code>T</code> must implement <code>ITableEntity</code> or have a string <code>RowKey</code> property and a string <code>PartitionKey</code> property.
<code>TableClient</code> ¹	A client connected to the table. This offers the most control for processing the table and can be used to write to it if the connection has sufficient permission.

¹ To use these types, you need to reference

[Microsoft.Azure.Functions.Worker.Extensions.Tables 1.2.0 or later](#) and the common dependencies for SDK type bindings.

Azure Tables output binding

When you want the function to write to a single entity, the Azure Tables output binding can bind to the following types:

[] [Expand table](#)

Type	Description
A JSON serializable type that implements <code>[ITableEntity]</code>	Functions attempts to serialize a plain-old CLR object (POCO) type as the entity. The type must implement <code>[ITableEntity]</code> or have a string <code>RowKey</code> property and a string <code>PartitionKey</code> property.

When you want the function to write to multiple entities, the Azure Tables output binding can bind to the following types:

[] [Expand table](#)

Type	Description
<code>T[]</code> where <code>T</code> is one of the single entity types	An array containing multiple entities. Each entry represents one entity.

For other output scenarios, create and use types from [Azure.Data.Tables](#) directly.

Next steps

- Read table data when a function runs
- Write table data from a function

Azure Tables input bindings for Azure Functions

Article • 09/21/2023

Use the Azure Tables input binding to read a table in [Azure Cosmos DB for Table](#) or [Azure Table Storage](#).

For information on setup and configuration details, see the [overview](#).

Example

The usage of the binding depends on the extension package version and the C# modality used in your function app, which can be one of the following:

Isolated worker model

An [isolated worker process class library](#) compiled C# function runs in a process isolated from the runtime.

Choose a version to see examples for the mode and version.

Azure Tables extension

The following `MyTableData` class represents a row of data in the table:

```
C#  
  
public class MyTableData : Azure.Data.Tables.ITableEntity  
{  
    public string Text { get; set; }  
  
    public string PartitionKey { get; set; }  
    public string RowKey { get; set; }  
    public DateTimeOffset? Timestamp { get; set; }  
    public ETag ETag { get; set; }  
}
```

The following function, which is started by a Queue Storage trigger, reads a row key from the queue, which is used to get the row from the input table. The expression `{queueTrigger}` binds the row key to the message metadata, which is the message string.

```
C#  
  
[Function("TableFunction")]  
[TableOutput("OutputTable", Connection = "AzureWebJobsStorage")]  
public static MyTableData Run(  
    [QueueTrigger("table-items")] string input,  
    [TableInput("MyTable", "<PartitionKey>", "{queueTrigger}")] MyTableData tableInput,  
    FunctionContext context)  
{  
    var logger = context.GetLogger("TableFunction");  
  
    logger.LogInformation($"PK={tableInput.PartitionKey}, RK={tableInput.RowKey}, Text={tableInput.Text}");  
  
    return new MyTableData()  
    {  
        PartitionKey = "queue",  
    };  
}
```

```

        RowKey = Guid.NewGuid().ToString(),
        Text = $"Output record with rowkey {input} created at {DateTime.Now}"
    };
}

```

The following Queue-triggered function returns the first 5 entities as an `IEnumerable<T>`, with the partition key value set as the queue message.

C#

```

[Function("TestFunction")]
public static void Run([QueueTrigger("myqueue", Connection = "AzureWebJobsStorage")] string
partition,
    [TableInput("inTable", "{queueTrigger}", Take = 5, Filter = "Text eq 'test'", Connection =
"AzureWebJobsStorage")] IEnumerable<MyTableData> tableInputs,
    FunctionContext context)
{
    var logger = context.GetLogger("TestFunction");
    logger.LogInformation(partition);
    foreach (MyTableData tableInput in tableInputs)
    {
        logger.LogInformation($"PK={tableInput.PartitionKey}, RK={tableInput.RowKey}, Text=
{tableInput.Text}");
    }
}

```

The `Filter` and `Take` properties are used to limit the number of entities returned.

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated worker model

In [C# class libraries](#), the `TableInputAttribute` supports the following properties:

Attribute property	Description
<code>TableName</code>	The name of the table.
<code>PartitionKey</code>	Optional. The partition key of the table entity to read.
<code>RowKey</code>	Optional. The row key of the table entity to read.
<code>Take</code>	Optional. The maximum number of entities to read into an <code>IEnumerable<T></code> . Can't be used with <code>RowKey</code> .
<code>Filter</code>	Optional. An OData filter expression for entities to read into an <code>IEnumerable<T></code> . Can't be used with <code>RowKey</code> .
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to the table service. See Connections .

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to your table service. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#)

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string for tables in Azure Table storage, follow the steps shown at [Manage storage account access keys](#). To obtain a connection string for tables in Azure Cosmos DB for Table, follow the steps shown at the [Azure Cosmos DB for Table FAQ](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage". If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [the Tables API extension](#), instead of using a connection string with a secret, you can have the app use an [Azure Active Directory identity](#). This only applies when accessing tables in Azure Storage. To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Table Service URI	<code><CONNECTION_NAME_PREFIX>__tableServiceUri¹</code>	The data plane URI of the Azure Storage table service to which you're connecting, using the	<code>https://<storage_account_name>.table.core.windows.net</code>

Property	Environment variable template	Description	Example value
		HTTPS scheme.	

¹ `<CONNECTION_NAME_PREFIX>_serviceUri` can be used as an alias. If both forms are provided, the `tableServiceUri` form is used. The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables.

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables in Azure Storage. The URI can only designate the table service. As an alternative, you can provide a URI specifically for each service under the same prefix, allowing a single connection to be used.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

ⓘ Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You'll need to create a role assignment that provides access to your Azure Storage table service at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Azure Tables extension against Azure Storage in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles (Azure Storage ¹)
Input binding	Storage Table Data Reader
Output binding	Storage Table Data Contributor

¹ If your app is instead connecting to tables in Azure Cosmos DB for Table, using an identity isn't supported and the connection must use a connection string.

Usage

The usage of the binding depends on the extension package version, and the C# modality used in your function app, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see usage details for the mode and version.

Azure Tables extension

When working with a single table entity, the Azure Tables input binding can bind to the following types:

Type	Description
A JSON serializable type that implements ITableEntity	Functions attempts to deserialize the entity into a plain-old CLR object (POCO) type. The type must implement ITableEntity or have a string <code>RowKey</code> property and a string <code>PartitionKey</code> property.
TableEntity ¹	The entity as a dictionary-like type.

When working with multiple entities from a query, the Azure Tables input binding can bind to the following types:

Type	Description
<code>IEnumerable<T></code> where <code>T</code> implements ITableEntity	An enumeration of entities returned by the query. Each entry represents one entity. The type <code>T</code> must implement ITableEntity or have a string <code>RowKey</code> property and a string <code>PartitionKey</code> property.
TableClient ¹	A client connected to the table. This offers the most control for processing the table and can be used to write to it if the connection has sufficient permission.

¹ To use these types, you need to reference [Microsoft.Azure.Functions.Worker.Extensions.Tables 1.2.0 or later](#) and the [common dependencies for SDK type bindings](#).

For specific usage details, see [Example](#).

Next steps

- [Write table data from a function](#)

Azure Tables output bindings for Azure Functions

Article • 09/21/2023

Use an Azure Tables output binding to write entities to a table in [Azure Cosmos DB for Table](#) or [Azure Table Storage](#).

For information on setup and configuration details, see the [overview](#)

ⓘ Note

This output binding only supports creating new entities in a table. If you need to update an existing entity from your function code, instead use an Azure Tables SDK directly.

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework. Extensions for isolated worker process functions use `Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

The following `MyTableData` class represents a row of data in the table:

```
C#  
  
public class MyTableData : Azure.Data.Tables.ITableEntity  
{  
    public string Text { get; set; }  
  
    public string PartitionKey { get; set; }  
    public string RowKey { get; set; }  
    public DateTimeOffset? Timestamp { get; set; }  
    public ETag ETag { get; set; }  
}
```

The following function, which is started by a Queue Storage trigger, writes a new `MyDataTable` entity to a table named `OutputTable`.

```
C#  
  
[Function("TableFunction")]  
[TableOutput("OutputTable", Connection = "AzureWebJobsStorage")]  
public static MyTableData Run(  
    [QueueTrigger("table-items")] string input,  
    [TableInput("MyTable", "<PartitionKey>", "{queueTrigger}")] MyTableData tableInput,
```

```

    FunctionContext context)
{
    var logger = context.GetLogger("TableFunction");

    logger.LogInformation($"PK={tableInput.PartitionKey}, RK={tableInput.RowKey}, Text=
{tableInput.Text}");

    return new MyTableData()
    {
        PartitionKey = "queue",
        RowKey = Guid.NewGuid().ToString(),
        Text = $"Output record with rowkey {input} created at {DateTime.Now}"
    };
}

```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the function. C# script instead uses a `function.json` configuration file as described in the [C# scripting guide](#).

Isolated worker model

In [C# class libraries](#), the `TableInputAttribute` supports the following properties:

Attribute property	Description
<code>TableName</code>	The name of the table to which to write.
<code>PartitionKey</code>	The partition key of the table entity to write.
<code>RowKey</code>	The row key of the table entity to write.
<code>Connection</code>	The name of an app setting or setting collection that specifies how to connect to the table service. See Connections .

Connections

The `connection` property is a reference to environment configuration that specifies how the app should connect to your table service. It may specify:

- The name of an application setting containing a [connection string](#)
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#)

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used.

Connection string

To obtain a connection string for tables in Azure Table storage, follow the steps shown at [Manage storage account access keys](#). To obtain a connection string for tables in Azure Cosmos DB for Table, follow the steps shown at the [Azure Cosmos DB for Table FAQ](#).

This connection string should be stored in an application setting with a name matching the value specified by the `connection` property of the binding configuration.

If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set `connection` to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage". If you leave `connection` empty, the Functions runtime uses the default Storage connection string in the app setting that is named `AzureWebJobsStorage`.

Identity-based connections

If you're using [the Tables API extension](#), instead of using a connection string with a secret, you can have the app use an [Azure Active Directory identity](#). This only applies when accessing tables in Azure Storage. To use an identity, you define settings under a common prefix that maps to the `connection` property in the trigger and binding configuration.

If you're setting `connection` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#). For all other connections, the extension requires the following properties:

Property	Environment variable template	Description	Example value
Table Service URI	<code><CONNECTION_NAME_PREFIX>__tableServiceUri</code> ¹	The data plane URI of the Azure Storage table service to which you're connecting, using the HTTPS scheme.	<code>https://<storage_account_name>.table.core.windows.net</code>

¹ `<CONNECTION_NAME_PREFIX>__serviceUri` can be used as an alias. If both forms are provided, the `tableServiceUri` form is used. The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables.

Other properties may be set to customize the connection. See [Common properties for identity-based connections](#).

The `serviceUri` form can't be used when the overall connection configuration is to be used across blobs, queues, and/or tables in Azure Storage. The URI can only designate the table service. As an alternative, you can provide a URI specifically for each service under the same prefix, allowing a single connection to be used.

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. For most Azure services, this means you need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

Important

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You'll need to create a role assignment that provides access to your Azure Storage table service at runtime. Management roles like [Owner](#) aren't sufficient. The following table shows built-in roles that are recommended when using the Azure Tables extension against Azure Storage in normal operation. Your application may require additional permissions based on the code you write.

Binding type	Example built-in roles (Azure Storage ¹)
Input binding	Storage Table Data Reader
Output binding	Storage Table Data Contributor

¹ If your app is instead connecting to tables in Azure Cosmos DB for Table, using an identity isn't supported and the connection must use a connection string.

Usage

The usage of the binding depends on the extension package version, and the C# modality used in your function app, which can be one of the following:

Isolated worker model

An isolated worker process class library compiled C# function runs in a process isolated from the runtime.

Choose a version to see usage details for the mode and version.

Azure Tables extension

When you want the function to write to a single entity, the Azure Tables output binding can bind to the following types:

Type	Description
A JSON serializable type that implements <code>[ITableEntity]</code>	Functions attempts to serialize a plain-old CLR object (POCO) type as the entity. The type must implement <code>[ITableEntity]</code> or have a string <code>RowKey</code> property and a string <code>PartitionKey</code> property.

When you want the function to write to multiple entities, the Azure Tables output binding can bind to the following types:

Type	Description
<code>T[]</code> where <code>T</code> is one of the single entity types	An array containing multiple entities. Each entry represents one entity.

For other output scenarios, create and use types from [Azure.Data.Tables](#) directly.

For specific usage details, see [Example](#).

Exceptions and return codes

Binding	Reference
Table	Table Error Codes
Blob, Table, Queue	Storage Error Codes
Blob, Table, Queue	Troubleshooting

Next steps

[Learn more about Azure functions triggers and bindings](#)

Timer trigger for Azure Functions

Article • 02/19/2024

This article explains how to work with timer triggers in Azure Functions. A timer trigger lets you run a function on a schedule.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Azure Functions developer reference](#)
- [Create your first function](#)
- C# developer references:
 - [In-process class library](#)
 - [Isolated worker process class library](#)
 - [C# script](#)
- [Azure Functions triggers and bindings concepts](#)
- [Code and test Azure Functions locally](#)

For information on how to manually run a timer-triggered function, see [Manually run a non HTTP-triggered function](#).

Support for this binding is automatically provided in all development environments. You don't have to manually install the package or register the extension.

Source code for the timer extension package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

Example

This example shows a C# function that executes each time the minutes have a value divisible by five. For example, when the function starts at 18:55:00, the next execution is at 19:00:00. A `TimerInfo` object is passed to the function.

A C# function can be created by using one of the following C# modes:

- **Isolated worker model:** Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.

Extensions for isolated worker process functions use

`Microsoft.Azure.Functions.Worker.Extensions.*` namespaces.

- **In-process model:** Compiled C# function that runs in the same process as the Functions runtime. In a variation of this model, Functions can be run using [C# scripting](#), which is supported primarily for C# portal editing. Extensions for in-process functions use `Microsoft.Azure.WebJobs.Extensions.*` namespaces.

Isolated worker model

C#

```
//<docsnippet_fixed_delay_retry_example>
[Function(nameof(TimerFunction))]
[FixedDelayRetry(5, "00:00:10")]
public static void Run([TimerTrigger("0 */5 * * *")] TimerInfo
timerInfo,
    FunctionContext context)
{
    var logger = context.GetLogger(nameof(TimerFunction));
```

Attributes

In-process C# library uses [TimerTriggerAttribute](#) from `Microsoft.Azure.WebJobs.Extensions` whereas isolated worker process C# library uses [TimerTriggerAttribute](#) from `Microsoft.Azure.Functions.Worker.Extensions.Timer` to define the function. C# script instead uses a [function.json](#) configuration file.

Isolated worker model

[Expand table](#)

Attribute	Description
<code>property</code>	
<code>Schedule</code>	A CRON expression or a TimeSpan value. A <code>TimeSpan</code> can be used only for a function app that runs on an App Service Plan. You can put the schedule expression in an app setting and set this property to the app setting name wrapped in % signs, as <code>%scheduleAppSetting%</code> .
<code>RunOnStartup</code>	If <code>true</code> , the function is invoked when the runtime starts. For example, the runtime starts when the function app wakes up after going idle due to inactivity. when the function app restarts due to function changes, and

Attribute	Description
property	when the function app scales out. <i>Use with caution.</i> RunOnStartup should rarely if ever be set to <code>true</code> , especially in production.
UseMonitor	Set to <code>true</code> or <code>false</code> to indicate whether the schedule should be monitored. Schedule monitoring persists schedule occurrences to aid in ensuring the schedule is maintained correctly even when function app instances restart. If not set explicitly, the default is <code>true</code> for schedules that have a recurrence interval greater than or equal to 1 minute. For schedules that trigger more than once per minute, the default is <code>false</code> .

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `values` collection.

✖ Caution

Don't set **runOnStartup** to `true` in production. Using this setting makes code execute at highly unpredictable times. In certain production settings, these extra executions can result in significantly higher costs for apps hosted in a Consumption plan. For example, with **runOnStartup** enabled the trigger is invoked whenever your function app is scaled. Make sure you fully understand the production behavior of your functions before enabling **runOnStartup** in production.

See the [Example section](#) for complete examples.

Usage

When a timer trigger function is invoked, a timer object is passed into the function. The following JSON is an example representation of the timer object.

JSON

```
{
  "Schedule": {
    "AdjustForDST": true
  },
  "ScheduleStatus": {
    "Last": "2016-10-04T10:15:00+00:00",
    "LastUpdated": "2016-10-04T10:16:00+00:00",
    "Next": "2016-10-04T10:20:00+00:00"
  }
}
```

```
        "IsPastDue":false  
    }
```

The `isPastDue` property is `true` when the current function invocation is later than scheduled. For example, a function app restart might cause an invocation to be missed.

NCRONTAB expressions

Azure Functions uses the [NCronTab](#) library to interpret NCRONTAB expressions. An NCRONTAB expression is similar to a CRON expression except that it includes an additional sixth field at the beginning to use for time precision in seconds:

```
{second} {minute} {hour} {day} {month} {day-of-week}
```

Each field can have one of the following types of values:

[+] Expand table

Type	Example	When triggered
A specific value	<code>0 5 * * *</code>	Once every hour of the day at minute 5 of each hour
All values (*)	<code>0 * 5 * * *</code>	At every minute in the hour, during hour 5
A range (- operator)	<code>5-7 * * * *</code> <code>*</code>	Three times a minute - at seconds 5 through 7 during every minute of every hour of each day
A set of values (, operator)	<code>5,8,10 * * *</code> <code>* *</code>	Three times a minute - at seconds 5, 8, and 10 during every minute of every hour of each day
An interval value (/ operator)	<code>0 */5 * * *</code> <code>*</code>	12 times an hour - at second 0 of every 5th minute of every hour of each day

To specify months or days you can use numeric values, names, or abbreviations of names:

- For days, the numeric values are 0 to 6 where 0 starts with Sunday.
- Names are in English. For example: `Monday`, `January`.
- Names are case-insensitive.
- Names can be abbreviated. Three letters is the recommended abbreviation length. For example: `Mon`, `Jan`.

NCRONTAB examples

Here are some examples of NCRONTAB expressions you can use for the timer trigger in Azure Functions.

[+] Expand table

Example	When triggered
<code>0 */5 * * *</code>	once every five minutes
<code>0 0 * * * *</code>	once at the top of every hour
<code>0 0 */2 * * *</code>	once every two hours
<code>0 0 9-17 * * *</code>	once every hour from 9 AM to 5 PM
<code>0 30 9 * * *</code>	at 9:30 AM every day
<code>0 30 9 * * 1-5</code>	at 9:30 AM every weekday
<code>0 30 9 * Jan Mon</code>	at 9:30 AM every Monday in January

ⓘ Note

NCRONTAB expression supports both **five field** and **six field** format. The sixth field position is a value for seconds which is placed at the beginning of the expression.

NCRONTAB time zones

The numbers in a CRON expression refer to a time and date, not a time span. For example, a 5 in the `hour` field refers to 5:00 AM, not every 5 hours.

The default time zone used with the CRON expressions is Coordinated Universal Time (UTC). To have your CRON expression based on another time zone, create an app setting for your function app named `WEBSITE_TIME_ZONE`.

The value of this setting depends on the operating system and plan on which your function app runs.

[+] Expand table

Operating system	Plan	Value
Windows	All	Set the value to the name of the desired time zone as given by the second line from each pair given by the Windows command

Operating system	Plan	Value
		<code>tzutil.exe /L</code>
Linux	Premium Dedicated	Set the value to the name of the desired time zone as shown in the tz database .

ⓘ Note

`WEBSITE_TIME_ZONE` and `TZ` are not currently supported when running on Linux in a Consumption plan. In this case, setting `WEBSITE_TIME_ZONE` or `TZ` can create SSL-related issues and cause metrics to stop working for your app.

For example, Eastern Time in the US (represented by `Eastern Standard Time` (Windows) or `America/New_York` (Linux)) currently uses UTC-05:00 during standard time and UTC-04:00 during daylight time. To have a timer trigger fire at 10:00 AM Eastern Time every day, create an app setting for your function app named `WEBSITE_TIME_ZONE`, set the value to `Eastern Standard Time` (Windows) or `America/New_York` (Linux), and then use the following NCrontab expression:

```
"0 0 10 * * *"
```

When you use `WEBSITE_TIME_ZONE` the time is adjusted for time changes in the specific timezone, including daylight saving time and changes in standard time.

TimeSpan

A `TimeSpan` can be used only for a function app that runs on an App Service Plan.

Unlike a CRON expression, a `TimeSpan` value specifies the time interval between each function invocation. When a function completes after running longer than the specified interval, the timer immediately invokes the function again.

Expressed as a string, the `TimeSpan` format is `hh:mm:ss` when `hh` is less than 24. When the first two digits are 24 or greater, the format is `dd:hh:mm`. Here are some examples:

[+] [Expand table](#)

Example	When triggered
"01:00:00"	every hour
"00:01:00"	every minute
"25:00:00:00"	every 25 days
"1.00:00:00"	every day

Scale-out

If a function app scales out to multiple instances, only a single instance of a timer-triggered function is run across all instances. It will not trigger again if there is an outstanding invocation is still running.

Function apps sharing Storage

If you are sharing storage accounts across function apps that are not deployed to app service, you might need to explicitly assign host ID to each app.

[\[+\] Expand table](#)

Functions version	Setting
2.x (and higher)	<code>AzureFunctionsWebHost__hostid</code> environment variable
1.x	<code>id</code> in <code>host.json</code>

You can omit the identifying value or manually set each function app's identifying configuration to a different value.

The timer trigger uses a storage lock to ensure that there is only one timer instance when a function app scales out to multiple instances. If two function apps share the same identifying configuration and each uses a timer trigger, only one timer runs.

Retry behavior

Unlike the queue trigger, the timer trigger doesn't retry after a function fails. When a function fails, it isn't called again until the next time on the schedule.

Manually invoke a timer trigger

The timer trigger for Azure Functions provides an HTTP webhook that can be invoked to manually trigger the function. This can be extremely useful in the following scenarios.

- Integration testing
- Slot swaps as part of a smoke test or warmup activity
- Initial deployment of a function to immediately populate a cache or lookup table in a database

Please refer to [manually run a non HTTP-triggered function](#) for details on how to manually invoke a timer triggered function.

Troubleshooting

For information about what to do when the timer trigger doesn't work as expected, see [Investigating and reporting issues with timer triggered functions not firing](#).

Next steps

[Go to a quickstart that uses a timer trigger](#)

[Learn more about Azure functions triggers and bindings](#)

Twilio binding for Azure Functions

Article • 03/31/2024

This article explains how to send text messages by using [Twilio](#) bindings in Azure Functions. Azure Functions supports output bindings for Twilio.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Azure Functions developer reference](#)
- [Create your first function](#)
- C# developer references:
 - [In-process class library](#)
 - [Isolated worker process class library](#)
 - [C# script](#)
- [Azure Functions triggers and bindings concepts](#)
- [Code and test Azure Functions locally](#)

Install extension

The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

The functionality of the extension varies depending on the extension version:

Functions v2.x+

There is currently no support for Twilio for an isolated worker process app.

Example

Unless otherwise noted, these examples are specific to version 2.x and later version of the Functions runtime.

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

 **Important**

[Support will end for the in-process model on November 10, 2026](#). We highly recommend that you [migrate your apps to the isolated worker model](#) for full support.

Isolated worker model

The Twilio binding isn't currently supported for a function app running in an isolated worker process.

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use attributes to define the output binding. C# script instead uses a [function.json configuration file](#).

Isolated worker model

The Twilio binding isn't currently supported for a function app running in an isolated worker process.

When you're developing locally, add your application settings in the [local.settings.json file](#) in the `Values` collection.

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Functions warmup trigger

Article • 11/07/2023

This article explains how to work with the warmup trigger in Azure Functions. A warmup trigger is invoked when an instance is added to scale a running function app. The warmup trigger lets you define a function that runs when a new instance of your function app is started. You can use a warmup trigger to preload custom dependencies so your functions are ready to start processing requests immediately. Some actions for a warmup trigger might include opening connections, loading dependencies, or running any other custom logic before your app begins receiving traffic.

The following considerations apply when using a warmup trigger:

- The warmup trigger isn't available to apps running on the [Consumption plan](#).
- The warmup trigger isn't supported on version 1.x of the Functions runtime.
- Support for the warmup trigger is provided by default in all development environments. You don't have to manually install the package or register the extension.
- There can be only one warmup trigger function per function app, and it can't be invoked after the instance is already running.
- The warmup trigger is only called during scale-out operations, not during restarts or other nonscaling startups. Make sure your logic can load all required dependencies without relying on the warmup trigger. Lazy loading is a good pattern to achieve this goal.
- Dependencies created by warmup trigger should be shared with other functions in your app. To learn more, see [Static clients](#).
- If the [built-in authentication](#) (also known as Easy Auth) is used, [HTTPS Only](#) should be enabled for the warmup trigger to get invoked.

Example

A C# function can be created by using one of the following C# modes:

- [Isolated worker model](#): Compiled C# function that runs in a worker process that's isolated from the runtime. Isolated worker process is required to support C# functions running on LTS and non-LTS versions .NET and the .NET Framework.
- [In-process model](#): Compiled C# function that runs in the same process as the Functions runtime.
- [C# script](#): Used primarily when you create C# functions in the Azure portal.

Isolated worker model

The following example shows a [C# function](#) that runs on each new instance when added to your app.

C#

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace SampleApp
{
    public static class Warmup
    {
        [Function(nameof(Warmup))]
        public static void Run([WarmupTrigger] object warmupContext,
        FunctionContext context)
        {
            var logger = context.GetLogger(nameof(Warmup));
            logger.LogInformation("Function App instance is now warm!");
        }
    }
}
```

Attributes

Both [in-process](#) and [isolated worker process](#) C# libraries use the `WarmupTrigger` attribute to define the function. C# script instead uses a [function.json configuration file](#).

Isolated worker model

Use the `WarmupTrigger` attribute to define the function. This attribute has no parameters.

See the [Example section](#) for complete examples.

Usage

The following considerations apply to using a warmup function in C#:

Isolated worker model

- Your function must be named `warmup` (case-insensitive) using the `Function` attribute.
- A return value attribute isn't required.
- Use the `Microsoft.Azure.Functions.Worker.Extensions.Warmup` package
- You can pass an object instance to the function.

Next steps

- [Learn more about Azure functions triggers and bindings](#)
- [Learn more about Premium plan](#)

Web PubSub bindings for Azure Functions

Article • 09/03/2024

This set of articles explains how to authenticate, send real-time messages to clients connected to [Azure Web PubSub](#) by using Azure Web PubSub bindings in Azure Functions.

[] [Expand table](#)

Action	Type
Handle client events from Web PubSub	Trigger binding
Handle client events from Web PubSub with HTTP trigger, or return client access URL and token	Input binding
Invoke service APIs	Output binding

[Samples](#)

Install extension

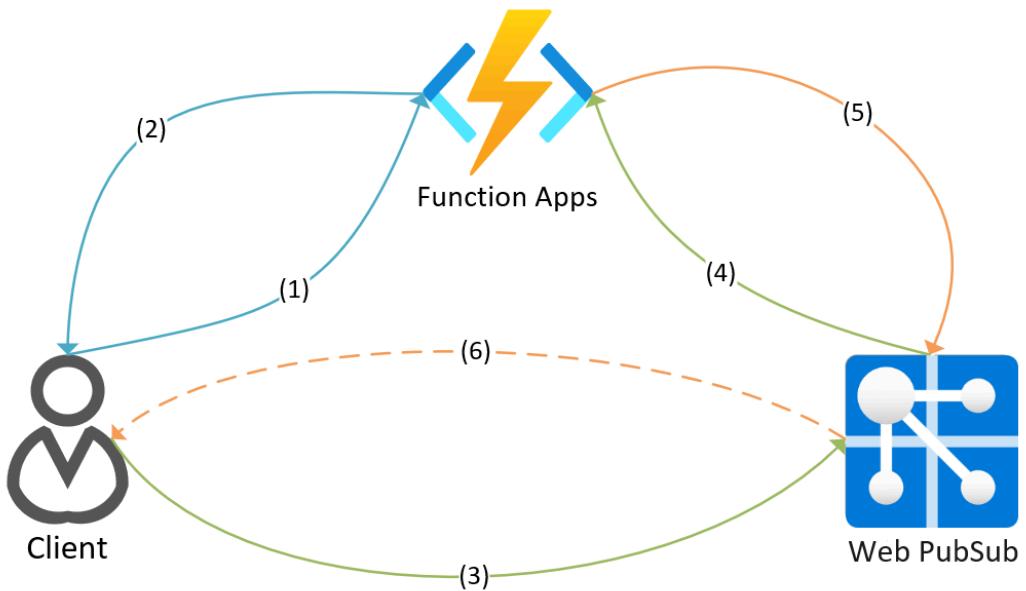
The extension NuGet package you install depends on the C# mode you're using in your function app:

Isolated worker model

Functions execute in an isolated C# worker process. To learn more, see [Guide for running C# Azure Functions in an isolated worker process](#).

Add the extension to your project by installing this [NuGet package](#).

Key concepts



(1)-(2) `WebPubSubConnection` input binding with `HttpTrigger` to generate client connection.

(3)-(4) `WebPubSubTrigger` trigger binding or `WebPubSubContext` input binding with `HttpTrigger` to handle service request.

(5)-(6) `WebPubSub` output binding to request service do something.

Connection string settings

Add the `WebPubSubConnectionString` key to the `host.json` file that points to the application setting with your connection string. For local development, this value may exist in the `local.settings.json` file.

For details on how to configure and use Web PubSub and Azure Functions together, refer to [Tutorial: Create a serverless notification app with Azure Functions and Azure Web PubSub service](#).

Next steps

- Handle client events from Web PubSub (Trigger binding)
- Handle client events from Web PubSub with HTTP trigger, or return client access URL and token (Input binding)
- Invoke service APIs (Output binding)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Web PubSub trigger binding for Azure Functions

Article • 09/03/2024

Use Azure Web PubSub trigger to handle client events from Azure Web PubSub service.

The trigger endpoint pattern would be as follows, which should be set in Web PubSub service side (Portal: settings -> event handler -> URL Template). In the endpoint pattern, the query part `code=<API_KEY>` is REQUIRED when you're using Azure Function App for security reasons. The key can be found in [Azure portal](#). Find your function app resource and navigate to **Functions** -> **App keys** -> **System keys** -> **webpubsub_extension** after you deploy the function app to Azure. Though, this key isn't needed when you're working with local functions.

The screenshot shows the Azure portal interface for a Function App. At the top, there is a placeholder URL: <Function_App_Url>/runtime/webhooks/webpubsub?code=<API_KEY>. Below this, the portal navigation bar includes 'Function App' and 'Directory: Microsoft'. The left sidebar contains links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events (preview), Functions (selected), App keys (highlighted with a red box), App files, Proxies, Deployment (Deployment slots, Deployment Center), Settings (Configuration, Authentication), and Help.

The main content area displays the 'Host keys (all functions)' section, which allows users to use host keys for their clients to access all HTTP functions in the app. It includes a 'New host key' button, a 'Show values' link, and a 'Filter host keys' input field. A table lists two host keys: '_master' and 'default', both with hidden values.

Below this is the 'System keys' section, which manages system keys automatically by the Function runtime. It provides granular access to functions runtime features. A 'Show values' link and a 'Filter system keys' input field are present. A table lists one system key: 'webpubsub_extension', also with a hidden value.

Example

The following sample shows how to handle user events from clients.

Isolated worker model

C#

```
[Function("Broadcast")]
public static void Run(
[WebPubSubTrigger("<hub>", WebPubSubEventType.User, "message")]
UserEventRequest request, ILogger log)
{
    log.LogInformation($"Request from:
{request.ConnectionContext.UserId}");
    log.LogInformation($"Request message data: {request.Data}");
    log.LogInformation($"Request message dataType: {request.DataType}");
}
```

`WebPubSubTrigger` binding also supports return value in synchronize scenarios, for example, system `Connect` and user event, when server can check and deny the client request, or send messages to the caller directly. `Connect` event respects `ConnectEventResponse` and `EventErrorResponse`, and user event respects `UserEventResponse` and `EventErrorResponse`, rest types not matching current scenario is ignored.

Isolated worker model

C#

```
[Function("Broadcast")]
public static UserEventResponse Run(
[WebPubSubTrigger("<hub>", WebPubSubEventType.User, "message")]
UserEventRequest request)
{
    return new UserEventResponse("[SYSTEM ACK] Received.");
}
```

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file.

 Expand table

function.json property	Attribute property	Description
type	n/a	Required - must be set to <code>webPubSubTrigger</code> .
direction	n/a	Required - must be set to <code>in</code> .
name	n/a	Required - the variable name used in function code for the parameter that receives the event data.
hub	Hub	Required - the value must be set to the name of the Web PubSub hub for the function to be triggered. We support set the value in attribute as higher priority, or it can be set in app settings as a global value.
eventType	WebPubSubEventType	Required - the value must be set as the event type of messages for the function to be triggered. The value should be either <code>user</code> or <code>system</code> .
eventName	EventName	<p>Required - the value must be set as the event of messages for the function to be triggered.</p> <p>For <code>system</code> event type, the event name should be in <code>connect</code>, <code>connected</code>, <code>disconnected</code>.</p> <p>For user-defined subprotocols, the event name is <code>message</code>.</p> <p>For system supported subprotocol <code>json.webpubsub.azure.v1.</code>, the event name is user-defined event name.</p>
clientProtocols	ClientProtocols	<p>Optional - specifies which client protocol can trigger the Web PubSub trigger functions.</p> <p>The following case-insensitive values are valid:</p> <ul style="list-style-type: none"> <code>all</code>: Accepts all client protocols. Default value. <code>webPubSub</code>: Accepts only Web PubSub protocols. <code>mqtt</code>: Accepts only MQTT protocols.
connection	Connection	Optional - the name of an app settings or setting collection that specifies the upstream Azure Web PubSub service. The value is used for signature validation. And the value is auto resolved with app settings <code>WebPubSubConnectionString</code> by default. And <code>null</code> means the validation isn't needed and always succeed.

Usages

In C#, `WebPubSubEventRequest` is type recognized binding parameter, rest parameters are bound by parameter name. Check following table for available parameters and types.

In weakly typed language like JavaScript, `name` in `function.json` is used to bind the trigger object regarding following mapping table. And respect `dataType` in `function.json` to convert message accordingly when `name` is set to `data` as the binding object for trigger input. All the parameters can be read from `context.bindingData.<BindingName>` and is `JObject` converted.

[\[+\] Expand table](#)

Binding Name	Binding Type	Description	Properties
request	<code>WebPubSubEventRequest</code>	Describes the upstream request	Property differs by different event types, including derived classes <code>ConnectEventRequest</code> , <code>MqttConnectEventRequest</code> , <code>ConnectedEventRequest</code> , <code>MqttConnectedEventRequest</code> , <code>UserEventRequest</code> , <code>DisconnectedEventRequest</code> , and <code>MqttDisconnectedEventRequest</code> .
connectionContext	<code>WebPubSubConnectionContext</code>	Common request information	<code>EventType</code> , <code>EventName</code> , <code>Hub</code> , <code>ConnectionId</code> , <code>UserId</code> , <code>Headers</code> , <code>Origin</code> , <code>Signature</code> , <code>States</code>
data	<code>BinaryData</code> , <code>string</code> , <code>Stream</code> , <code>byte[]</code>	Request message data from client in user message event	-
dataType	<code>WebPubSubDataType</code>	Request message dataType, which supports <code>binary</code> , <code>text</code> , <code>json</code>	-
claims	<code>IDictionary<string, string[]></code>	User Claims in system	-

Binding Name	Binding Type	Description	Properties
		connect request	
query	<code>IDictionary<string, string[]></code>	User query in system connect request	-
subprotocols	<code>IList<string></code>	Available subprotocols in system connect request	-
clientCertificates	<code>IList<ClientCertificate></code>	A list of certificate thumbprint from clients in system connect request	-
reason	<code>string</code>	Reason in system disconnected request	-

ⓘ Important

In C#, multiple types supported parameter **MUST** be put in the first, i.e. `request` or `data` that other than the default `BinaryData` type to make the function binding correctly.

Return response

`WebPubSubTrigger` respects customer returned response for synchronous events of `connect` and user event. Only matched response is sent back to service, otherwise, it's ignored. Besides, `WebPubSubTrigger` return object supports users to `SetState()` and `ClearStates()` to manage the metadata for the connection. And the extension merges the results from return value with the original ones from `request` `WebPubSubConnectionContext.States`. Value in existing key is overwrite and value in new key is added.

[\[\] Expand table](#)

Return Type	Description	Properties
ConnectEventResponse	Response for <code>connect</code> event	Groups, Roles, UserId, Subprotocol
UserEventResponse	Response for user event	DataType, Data
EventErrorResponse	Error response for the sync event	Code, ErrorMessage
*WebPubSubEventResponse	Base response type of the supported ones used for uncertain return cases	-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Web PubSub input bindings for Azure Functions

Article • 09/03/2024

Our extension provides two input binding targeting different needs.

- [WebPubSubConnection](#)

To let a client connect to Azure Web PubSub Service, it must know the service endpoint URL and a valid access token. The `WebPubSubConnection` input binding produces required information, so client doesn't need to handle this token generation itself. The token is time-limited and can authenticate a specific user to a connection. Therefore, don't cache the token or share it between clients. An HTTP trigger working with this input binding can be used for clients to retrieve the connection information.

- [WebPubSubContext](#)

When using is Static Web Apps, `HttpTrigger` is the only supported trigger and under Web PubSub scenario, we provide the `WebPubSubContext` input binding helps users deserialize upstream http request from service side under Web PubSub protocols. So customers can get similar results comparing to `WebPubSubTrigger` to easily handle in functions. When used with `HttpTrigger`, customer requires to configure the `HttpTrigger` exposed url in event handler accordingly.

WebPubSubConnection

Example

The following example shows an HTTP trigger function that acquires Web PubSub connection information using the input binding and returns it over HTTP. In following example, the `UserId` is passed in through client request query part like `?userid={UserA}`.

Isolated worker model

```
C#
[Function("WebPubSubConnectionInputBinding")]
public static HttpResponseMessage
```

```
Run([HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequestData req,
[WebPubSubConnectionInput(Hub = "<hub>", , UserId = "{query.userid}",
Connection = "<web_pubsub_connection_name>")] WebPubSubConnection
connectionInfo)
{
    var response = req.CreateResponse(HttpStatusCode.OK);
    response.WriteAsJsonAsync(connectionInfo);
    return response;
}
```

Get authenticated user ID

If the function is triggered by an authenticated client, you can add a user ID claim to the generated token. You can easily add authentication to a function app using App Service Authentication.

App Service Authentication sets HTTP headers named `x-ms-client-principal-id` and `x-ms-client-principal-name` that contain the authenticated user's client principal ID and name, respectively.

You can set the `UserId` property of the binding to the value from either header using a binding expression: `{headers.x-ms-client-principal-id}` or `{headers.x-ms-client-principal-name}`.

Isolated worker model

C#

```
[Function("WebPubSubConnectionInputBinding")]
public static HttpResponseMessage
Run([HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequestData req,
[WebPubSubConnectionInput(Hub = "<hub>", , UserId = "{headers.x-ms-
client-principal-id}", Connection = "<web_pubsub_connection_name>")]
WebPubSubConnection connectionInfo)
{
    var response = req.CreateResponse(HttpStatusCode.OK);
    response.WriteAsJsonAsync(connectionInfo);
    return response;
}
```

Configuration

The following table explains the binding configuration properties that you set in the function.json file and the `WebPubSubConnection` attribute.

[\[+\] Expand table](#)

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to <code>webPubSubConnection</code>
<code>direction</code>	n/a	Must be set to <code>in</code>
<code>name</code>	n/a	Variable name used in function code for input connection binding object.
<code>hub</code>	Hub	Required - The value must be set to the name of the Web PubSub hub for the function to be triggered. We support set the value in attribute as higher priority, or it can be set in app settings as a global value.
<code>userId</code>	<code>UserId</code>	Optional - the value of the user identifier claim to be set in the access key token.
<code>clientProtocol</code>	<code>ClientProtocol</code>	Optional - The client protocol type. Valid values include <code>default</code> and <code>mqtt</code> . For MQTT clients, you must set it to <code>mqtt</code> . For other clients, you can omit the property or set it to <code>default</code> .
<code>connection</code>	<code>Connection</code>	Required - The name of the app setting that contains the Web PubSub Service connection string (defaults to "WebPubSubConnectionString").

Usage

`WebPubSubConnection` provides following properties.

[\[+\] Expand table](#)

Binding Name	Binding Type	Description
<code>BaseUri</code>	<code>Uri</code>	Web PubSub client connection uri.
<code>Uri</code>	<code>Uri</code>	Absolute Uri of the Web PubSub connection, contains <code>AccessToken</code> generated base on the request.

Binding Name	Binding Type	Description
AccessToken	string	Generated AccessToken based on request UserId and service information.

More customization of generated token

Limited to the binding parameter types don't support a way to pass list nor array, the `WebPubSubConnection` isn't fully supported with all the parameters server SDK has, especially `roles`, and also includes `groups` and `expiresAfter`.

When customer needs to add roles or delay building the access token in the function, we suggest you to work with [server SDK for C#](#).

Isolated worker model

```
C#
[Function("WebPubSubConnectionCustomRoles")]
public static HttpResponseMessage
Run([HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequestData req)
{
    var serviceClient = new WebPubSubServiceClient(new Uri(endpoint), "<hub>", "<web-pubsub-connection-string>");
    var userId = req.Query["userid"].FirstOrDefault();
    // your method to get custom roles.
    var roles = GetRoles(userId);
    var url = await
    serviceClient.GetClientAccessUriAsync(TimeSpan.FromMinutes(5), userId,
    roles);
    var response = req.CreateResponse(HttpStatusCode.OK);
    response.WriteString(url.ToString());
    return response;
}
```

WebPubSubContext

Example

Isolated worker model

```
C#
```

```

// validate method when upstream set as http://<func-host>/api/{event}
[Function("validate")]
public static HttpResponseMessage Validate(
    [HttpTrigger(AuthorizationLevel.Anonymous, "options")]
    HttpRequestData req,
    [WebPubSubContextInput] WebPubSubContext wpsReq)
{
    return BuildHttpResponseData(req, wpsReq.Response);
}

// Respond AbuseProtection to put header correctly.
private static HttpResponseMessage BuildHttpResponseData(HttpRequestData request, SimpleResponse wpsResponse)
{
    var response = request.CreateResponse();
    response.StatusCode = (HttpStatusCode)wpsResponse.Status;
    response.Body = response.Body;
    foreach (var header in wpsResponse.Headers)
    {
        response.Headers.Add(header.Key, header.Value);
    }
    return response;
}

```

Configuration

The following table explains the binding configuration properties that you set in the `functions.json` file and the `WebPubSubContext` attribute.

[] Expand table

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to <code>webPubSubContext</code> .
<code>direction</code>	n/a	Must be set to <code>in</code> .
<code>name</code>	n/a	Variable name used in function code for input Web PubSub request.
<code>connection</code>	Connection	Optional - the name of an app settings or setting collection that specifies the upstream Azure Web PubSub service. The value is used for Abuse Protection and Signature validation. The value is auto resolved with "WebPubSubConnectionString" by default. And <code>null</code> means the validation isn't needed and always succeed.

Usage

`WebPubSubContext` provides following properties.

[+] Expand table

Binding Name	Binding Type	Description	Properties
request	<code>WebPubSubEventRequest</code>	Request from client, see following table for details.	<code>WebPubSubConnectionContext</code> from request header and other properties deserialized from request body describe the request, for example, <code>Reason</code> for <code>DisconnectedEventRequest</code> .
response	<code>HttpResponseMessage</code>	Extension builds response mainly for <code>AbuseProtection</code> and errors cases.	-
errorMessage	<code>string</code>	Describe the error details when processing the upstream request.	-
hasError	<code>bool</code>	Flag to indicate whether it's a valid Web PubSub upstream request.	-
isPreflight	<code>bool</code>	Flag to indicate whether it's a preflight request of <code>AbuseProtection</code> .	-

For `WebPubSubEventRequest`, it's deserialized to different classes that provide different information about the request scenario. For `PreflightRequest` or not valid cases, user can check the flags `IsPreflight` and `HasError` to know. We suggest you to return system build response `WebPubSubContext.Response` directly, or customer can log errors on demand. In different scenarios, customer can read the request properties as following.

[+] Expand table

Derived Class	Description	Properties
PreflightRequest	Used in <code>AbuseProtection</code> when <code>IsPreflight</code> is <code>true</code>	-
ConnectEventRequest	Used in system <code>Connect</code> event type	Claims, Query, Subprotocols, ClientCertificates
ConnectedEventRequest	Used in system <code>Connected</code> event type	-
UserEventRequest	Used in user event type	Data, DataType
DisconnectedEventRequest	Used in system <code>Disconnected</code> event type	Reason

ⓘ Note

Though the `WebPubSubContext` is a input binding provides similar request deserialize way under `HttpTrigger` comparing to `WebPubSubTrigger`, there's limitations, i.e. connection state post merge isn't supported. The return response is still respected by the service side, but users require to build the response themselves. If users have needs to set the event response, you should return a `HttpResponseMessage` contains `ConnectEventResponse` or messages for user event as **response body** and put connection state with key `ce-connectionstate` in **response header**.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Web PubSub output binding for Azure Functions

Article • 09/03/2024

Use the *Web PubSub* output binding to invoke Azure Web PubSub service to do something. You can send a message to:

- All connected clients
- Connected clients authenticated to a specific user
- Connected clients joined in a specific group
- A specific client connection

The output binding also allows you to manage clients and groups, and grant/revoke permissions targeting specific `connectionId` with group.

- Add connection to group
- Add user to group
- Remove connection from a group
- Remove user from a group
- Remove user from all groups
- Close all client connections
- Close a specific client connection
- Close connections in a group
- Grant permission of a connection
- Revoke permission of a connection

For information on setup and configuration details, see the [overview](#).

Example

Isolated worker model

C#

```
[Function("WebPubSubOutputBinding")]
[WebPubSubOutput(Hub = "<hub>", Connection = "
<web_pubsub_connection_name>")]
public static WebPubSubAction
Run([HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequestData req)
{
    return new SendToAllAction
```

```

    {
        Data = BinaryData.FromString("Hello Web PubSub!"),
        DataType = WebPubSubDataType.Text
    };
}

```

WebPubSubAction

`WebPubSubAction` is the base abstract type of output bindings. The derived types represent the action server want service to invoke.

In C# language, we provide a few static methods under `WebPubSubAction` to help discover available actions. For example, user can create the `SendToAllAction` by call `WebPubSubAction.CreateSendToAllAction()`.

[\[+\] Expand table](#)

Derived Class	Properties
<code>SendToAllAction</code>	Data, DataType, Excluded
<code>SendToGroupAction</code>	Group, Data, DataType, Excluded
<code>SendToUserAction</code>	UserId, Data, DataType
<code>SendToConnectionAction</code>	ConnectionId, Data, DataType
<code>AddUserToGroupAction</code>	UserId, Group
<code>RemoveUserFromGroupAction</code>	UserId, Group
<code>RemoveUserFromAllGroupsAction</code>	UserId
<code>AddConnectionToGroupAction</code>	ConnectionId, Group
<code>RemoveConnectionFromGroupAction</code>	ConnectionId, Group
<code>CloseAllConnectionsAction</code>	Excluded, Reason
<code>CloseClientConnectionAction</code>	ConnectionId, Reason
<code>CloseGroupConnectionsAction</code>	Group, Excluded, Reason
<code>GrantPermissionAction</code>	ConnectionId, Permission, TargetName
<code>RevokePermissionAction</code>	ConnectionId, Permission, TargetName

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `WebPubSub` attribute.

[Expand table](#)

function.json property	Attribute property	Description
<code>type</code>	n/a	Must be set to <code>webPubSub</code>
<code>direction</code>	n/a	Must be set to <code>out</code>
<code>name</code>	n/a	Variable name used in function code for output binding object.
<code>hub</code>	Hub	The value must be set to the name of the Web PubSub hub for the function to be triggered. We support set the value in attribute as higher priority, or it can be set in app settings as a global value.
<code>connection</code>	Connection	The name of the app setting that contains the Web PubSub Service connection string (defaults to "WebPubSubConnectionString").

Troubleshooting

Setting up console logging

You can also easily [enable console logging](#) if you want to dig deeper into the requests you're making against the service.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

AZFW0001: Invalid binding attributes

Article • 12/07/2023

This rule occurs when invalid binding attributes are used in the function definition.

[+] Expand table

Value	
Rule ID	AZFW0001
Severity	Error

Rule description

The Azure Functions .NET language worker uses an [isolated worker model](#) for input and output bindings. This model requires libraries that are incompatible with the original in-process binding model used by both [in-process .NET class library functions](#) and WebJobs.

A new set of packages were introduced to support the existing bindings and triggers using the new .NET isolated worker model. The two package naming conventions are:

.NET isolated worker model: `Microsoft.Azure.Functions.Worker.Extensions.*`. .NET In-process and WebJobs: `Microsoft.Azure.WebJobs.Extensions.*`

How to fix violations

To fix violations, change or add a reference to the appropriate package as described above and use the correct attributes from that package. For more information, see the binding reference articles for your specific binding extensions.

When to suppress the rule

This rule shouldn't be suppressed, as the existing bindings won't work in the isolated model.

AZF0001: Avoid `async void`

Article • 12/07/2023

This rule is triggered when the `void` return type is used in an `async` function definition.

[+] Expand table

Value	
Rule ID	AZF0001
Severity	Error

Rule description

Defining `async` functions with a `void` return type make it impossible for the Functions runtime to track invocation completion or catch and handle exceptions thrown by the function method.

Refer to this article for general `async void` information:

[https://msdn.microsoft.com/magazine/jj991977.aspx ↗](https://msdn.microsoft.com/magazine/jj991977.aspx)

How to fix violations

To fix violations, change the function's return type from `void` to `Task` and make the necessary code changes to appropriately return a `Task`.

When to suppress the rule

This rule should not be suppressed. Use of `async void` will lead to unpredictable behavior.

AZF0002: Inefficient HttpClient usage

Article • 11/17/2021

This rule is triggered when an [HttpClient](#) is instantiated within a `[FunctionName]`-decorated method.

	Value
Rule ID	AZF0002
Category	[Reliability]
Severity	Warning

Rule description

Simple usage of [HttpClient](#) to make HTTP requests presents several issues, including vulnerability to socket exhaustion. In a Function app, calling the `HttpClient` constructor in the body of a function method will create a new instance with every function invocation, amplifying these issues. For apps running on a [Consumption hosting plan](#), inefficient `HttpClient` usage can exhaust the plan's [outbound connection limits](#).

The [recommended best practice](#) is to use an `IHttpClientFactory` with dependency injection or a single static `HttpClient` instance, depending on the nature of your application.

How to fix violations

- Move your `HttpClient` instantiation to a static instance defined outside of the function, such as in [this example](#).
- Supply an `HttpClient` to your functions using `IHttpClientFactory` through [dependency injection](#). This is the recommended approach for fixing this violation.

When to suppress the rule

This rule should not be suppressed.

Next steps

For more information about connection management best practices in Azure Functions, see [Manage connections in Azure Functions](#).

For more information about `HttpClient` behavior issues and management, see [Use `IHttpClientFactory` to implement resilient HTTP requests](#)

AZFD0001: AzureWebJobsStorage app setting is not present.

Article • 12/07/2023

This event occurs when the function app doesn't have the `AzureWebJobsStorage` app setting configured for the function app.

[+] Expand table

	Value
Event ID	AZFD0001
Severity	Error

Event description

The `AzureWebJobsStorage` app setting is used to store the connection string of the Azure Storage account associated with the function app. The Azure Functions runtime uses this connection for core behaviors such as coordinating singleton execution of timer triggers, default app key storage, and storing diagnostic events.

For more information, see [AzureWebJobsStorage](#).

How to resolve the event

Create a new app setting on your function app with name `AzureWebJobsStorage` with a valid storage account connection string as the value. For more information, see [Work with application settings](#).

When to suppress the event

This event shouldn't be suppressed.

AZFD0002: Value of AzureWebJobsStorage app setting is invalid.

Article • 12/07/2023

This event occurs when the value of the `AzureWebJobsStorage` app setting is set to either an invalid Azure Storage account connection string or to a Key Vault reference.

[] Expand table

	Value
Event ID	AZFD0002
Severity	Error

Event description

The `AzureWebJobsStorage` app setting is used to store the connection string of the storage account associated with the function app. The Azure Functions runtime uses this connection for core behaviors such as coordinating singleton execution of timer triggers, default app key storage, and storing diagnostic events. This app setting needs to be set to a valid connection string.

For more information, see [AzureWebJobsStorage](#).

How to resolve the event

Update the value of the `AzureWebJobsStorage` app setting on your function app with a valid storage account connection string. For more information, see [Troubleshoot error: "Azure Functions Runtime is unreachable"](#).

When to suppress the event

You should suppress this event when your function app uses an Azure Key Vault reference in the `AzureWebjobsStorage` app setting instead of a connection string. For more information, see [Source application settings from Key Vault](#)

AZFD0003: Encountered a StorageException while trying to fetch the diagnostic events.

Article • 12/07/2023

This event occurs when the Azure Storage account connection string value in the `AzureWebJobsStorage` app setting either doesn't have permissions to access Azure Table Storage or generates exceptions when trying to connect to storage.

[] Expand table

	Value
Event ID	AZFD0003
Severity	Error

Event description

The `AzureWebJobsStorage` app setting is used to store the connection string of the storage account associated with the function app. The Azure Functions runtime uses this connection for core behaviors such as coordinating singleton execution of timer triggers, default app key storage, and storing diagnostic events.

The connection string set in `AzureWebJobsStorage` must be for an account that has permissions to store and read diagnostic events from Table Storage. The complete set of read, write, delete, add, and create events must be supported.

For more information, see [AzureWebJobsStorage](#).

How to resolve the event

Make sure that the storage account for the connection string stored in `AzureWebJobsStorage` has permissions to read, write, delete, add and create in Table Storage. Clients should be able to access Storage using this connection string without generating exceptions.

When to suppress the event

This event should not be suppressed.

AZFD0004: Host ID collision

Article • 01/30/2023

This event occurs when you have the same host ID assigned to multiple function apps or slots, which also share the same storage account.

	Value
Event ID	AZFD0004
Category	[Usage]
Severity	Error

Event description

A host ID collision can occur when more than one function app or slot uses the same host ID while sharing a storage account. This condition usually occurs due to truncation of similar function app names when the host ID value is generated. For example, if you have multiple apps or slots with names longer than 32 characters and the first 32 characters are shared, both generated host ID values may be the same due to truncation.

You can also have the same collision when you explicitly set the same host ID value on multiple function apps that use the same storage account.

When multiple apps have the same host ID, the resulting collision can cause incorrect behaviors. For example, some triggers, like timer and Blob Storage, store tracking data by host ID. A host ID collision can result in incorrect behavior when the host can't differentiate between apps by host ID. When such a collision is detected, an error (hard failure) is logged and the host is shut down. Before version 4.x of the Functions runtime, a warning was logged, but the host wasn't shut down.

For more information, see [host ID considerations](#).

Options for addressing collisions:

- Connect each function app or slot in the collision to a different storage account by changing the [AzureWebJobsStorage](#) application setting or slot setting.
- Rename your function apps to a name that has fewer than 32 characters. When app names have fewer than 32 characters, unique host IDs can be generated for

each app, which removes the collision.

- Set explicit host ID values for your function apps or slots so they no longer conflict.
For more information, see [host ID considerations](#).

When to suppress the event

This event shouldn't be suppressed.

AZFD0005: External startup exception

Article • 03/01/2023

This event occurs when an external startup class throws an exception during function app initialization.

	Value
Event ID	AZFD0005
Category	[Usage]
Severity	Error

Event description

An external startup class is a class registered with the `FunctionsStartupAttribute`. It is often used to register services or configuration sources for dependency injection. For more information on the feature, see [use dependency injection in .NET Azure Functions](#).

If a call to either of the `Configure()` methods on the `FunctionsStartup` class throws an exception, it will cause the function app initialization to fail. The functions host will catch this exception, log the error, and retry initialization. This retry helps to recover from transient errors, but permanent errors may be logged many times as the host retries.

If `APPLICATIONINSIGHTS_CONNECTION_STRING` is set as an app setting, the error will also be logged as an `exception` in Application Insights.

How to resolve the event

Every occurrence of this event is unique to the application. Investigate the error message and stack trace to see what may need to be done to prevent this error in the future. For example, a timeout may need to be retried; or an error calling an external service may need to be handled.

When to suppress the event

This event shouldn't be suppressed.

AZFD0006: SAS Token Expiring Warning

Article • 12/07/2023

This event occurs when an SAS token in an application setting is set to expire within 45 days.

[+] Expand table

	Value
Event ID	AZFD0006
Severity	Warning

Event description

This warning event occurs when an SAS token within a URI is set to expire within 45 days. If the token has already expired, an error event will be triggered.

Functions currently only checks the following application settings for expiring SAS tokens:

- [WEBSITE_RUN_FROM_PACKAGE](#)
- [AzureWebJobsStorage](#)
- [WEBSITE_CONTENTAZUREFILECONNECTIONSTRING](#) The warning message specifies which variable has the SAS token that's going to expire and how much time is left.

How to resolve the event

There are two ways to resolve this event:

- Renew the SAS token so that it doesn't expire within the next 45 days and set the new value in application settings. For more information, see [Manually uploading a package to Blob Storage](#).
- Switch to using managed identities instead relying on using an SAS URI. For more information, see [Fetch a package from Azure Blob Storage using a managed identity](#).

When to suppress the event

This event shouldn't be suppressed.

AZFD0007: Repository has more than 10 nondecryptable secrets backups

Article • 12/07/2023

This error occurs when you reach the maximum number of backups of the secrets repository file that maintains your function app access keys. At this point, your function app can't start until after you delete one or more of these repository file backups.

[+] Expand table

	Value
Event ID	AZFD0007
Severity	Error

Event description

Azure Functions uses an encrypted repository file (host.json) to securely store [access keys](#) used by your function app. Whenever the Functions host is unable to decrypt this repository file, it regenerates the repository file and creates a backup of the unreadable file with a name like `host.snapshot.<DATE>.json`.

Some reasons for the repository to be regenerated can include:

- Changes to the main decryption key used by the function app
- Recreating a function app with the same name using the same storage account
- [Host ID collisions](#)
- Using [lifecycle management policies](#)

When 10 unreadable secrets repository backup files exist, the secrets repository can't be regenerated and your function app might not start or run correctly.

How to resolve the event

The repository backup files are stored in the storage account set in the [AzureWebJobsStorage](#) application setting. They can be found in a container named `azure-webjobs-secrets` in a subfolder with the same name as your function app.

To resolve this error, delete one or more of the repository backups (`host.snapshot.<DATE>.json`) from the `azure-webjobs-secrets\<FUNCTION_APP_NAME>` container in the storage account used by your function app.

When to suppress the event

This event shouldn't be suppressed.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

AZFD0008: Failed to read from Blob Storage secret repository because its access tier is set to archive

Article • 12/07/2023

This error happens when you set the Azure Blob Storage access tier to `archive` in the file where secrets are stored. This setting prevents the Azure Functions host from accessing key information required by certain functions.

[+] Expand table

	Value
Event ID	AZFD0008
Severity	Error

Event description

By default, the access tier for blobs in Azure Storage is set to an online value (`hot` or `cool`). For more information, see [Set a blob's access tier](#).

Azure Functions uses an encrypted repository file (`host.json`) to securely store [access keys](#) (function or host) used by your function app. When the access tier of this secrets repository file is set to `archive`, functions that require access keys return an error. These functions can include HTTP triggers, Event Grid calls, and durable orchestrations.

When access keys aren't accessible because the secrets repository is in an archived state, you see a 409 warning in the logs like:

This operation is not permitted on an archived blob.

How to resolve the event

To resolve this error, change the access tier of the repository (`host.json`) file used by your function app back to `hot`. To learn how to set the access tier on a file, see [Set a blob's access tier](#).

You can find the host.json file in the storage account used by your function app in a container named `azure-webjobs-secrets` in a subfolder with the same name as your function app, such as: `azure-webjobs-secrets\<FUNCTION_APP_NAME>\host.json`.

To determine the storage account used by your function app, review the [AzureWebJobsStorage](#) application setting.

When to suppress the event

This event shouldn't be suppressed.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

AZFD0009: Unable to parse host configuration file

Article • 12/07/2023

The Azure Functions runtime was unable to parse the host configuration file (host.json) for your function app.

[+] Expand table

Value	
Event ID	AZFD0009
Severity	Error

Event description

For some reason, the Functions runtime couldn't parse the host configuration file (host.json) in the root of your function app. When the Functions host can't read this file, your app can't start.

How to resolve the event

To resolve this error, you should make sure that the host.json file in the root of your function app contains valid JSON in a valid configuration. For more information about the configuration allowed in this file, see the [host.json file](#) reference article.

Because you use the same host.json file when running your functions locally, it could be helpful to debug this configuration file [during local development](#).

When to suppress the event

This event shouldn't be suppressed.

AZFD0010: Linux Consumption Does Not Support TZ & WEBSITE_TIME_ZONE Error

Article • 12/07/2023

This event occurs when a function app running on Linux in an Azure Functions Consumption plan app has either `WEBSITE_TIME_ZONE` or `TZ` set as an application setting.

[] Expand table

	Value
Event ID	AZFD0010
Severity	Error

Event description

Using `WEBSITE_TIME_ZONE` or `TZ` when running on Linux in a Consumption plan can create SSL-related issues and cause metrics to stop working for your app. For more information, see the [WEBSITE_TIME_ZONE](#) reference. You can learn more about NCrontab timezones in the [Timer trigger reference](#).

How to resolve the event

Remove either `WEBSITE_TIME_ZONE` or `TZ` from the [application settings](#). If your scenario requires you to adjust the timezone of your function app, consider using a different [hosting plan](#) or running on Windows, if possible.

When to suppress the event

This event shouldn't be suppressed.

AZFD0011: The FUNCTIONS_WORKER_RUNTIME setting is required

Article • 10/03/2024

This event occurs when a function app doesn't have a value for the `FUNCTIONS_WORKER_RUNTIME` application setting, which is required.

[+] Expand table

	Value
Event ID	AZFD0011
Severity	Warning

Event description

The `FUNCTIONS_WORKER_RUNTIME` application setting indicates the language or language stack on which the function app runs, such as `python`. For more information on valid values, see the [FUNCTIONS_WORKER_RUNTIME](#) reference.

You should always specify a valid `FUNCTIONS_WORKER_RUNTIME` for your function apps. When you don't have this setting and the Functions host can't determine the correct language or language stack, you might see performance degradations, exceptions, or unexpected behaviors. To ensure that your application operates as intended, you should explicitly set it in all of your existing function apps and deployment scripts.

The value of `FUNCTIONS_WORKER_RUNTIME` should align with the language stack used to create the deployed application payload. If these do not align, you may see the [AZFD0013](#) event.

How to resolve the event

In a production application, set [FUNCTIONS_WORKER_RUNTIME to a valid value](#) in the [application settings](#). The value should align with the language stack used to create the application payload.

When running locally in Azure Functions Core Tools, also set **FUNCTIONS_WORKER_RUNTIME** to a valid value in the [local.settings.json file](#). The value should align with the language stack used in the local project.

When to suppress the event

This event shouldn't be suppressed.

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

AZFD0012: A non highly-identifiable secret has been loaded by the application

Article • 07/18/2024

This event occurs when a function app has either the `StrictHISModeWarn` or `StrictHISModeEnabled` feature flag enabled in the [AzureWebJobsFeatureFlags](#) application setting.

[+] Expand table

	Value
Event ID	AZFD0012
Severity	Warning or Error

Event description

The runtime determined that one or more secrets loaded by the host aren't highly identifiable secrets. Highly identifiable secrets are preferred because they include a signature/checksum along with a prefix that enables automated scanning tools to identify the secret as an Azure Functions key value. This identifiability aids in the remediation of inadvertently leaked secrets, such as when they get accidentally checked into a source control repository.

By default, the secret key values generated by Azure Functions are highly identifiable. However, function apps that were created long ago might still have legacy nonidentifiable key values generated before highly identifiable secret generation was enabled.

How to resolve the event

This event indicates the name and type of the key that is in violation. You should regenerate the key to obtain a new value that is highly identifiable. To learn how to regenerate keys, see [Renew access keys](#).

When to suppress the event

This event shouldn't be suppressed.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

AZFD0013: The configured runtime does not match the worker runtime metadata found in the deployed function app artifacts

Article • 10/10/2024

This event occurs when a function app has a `FUNCTIONS_WORKER_RUNTIME` setting specifying a language stack, but a payload for a different stack is deployed to it.

[+] Expand table

	Value
Event ID	AZFD0013
Severity	Warning or Error

Event description

The `FUNCTIONS_WORKER_RUNTIME` application setting indicates the language or language stack on which the function app runs, such as `python`. For more information on valid values, see the [FUNCTIONS_WORKER_RUNTIME](#) reference. The deployed application must correspond with the provided value. If there is a mismatch, it means that either the value of `FUNCTIONS_WORKER_RUNTIME` is incorrect, or that an unexpected payload was deployed to the application.

This event may appear for apps that were previously using inconsistent and undefined behavior to continue running while in a mismatch state. Follow the instructions in this article to resolve the event for these applications. Doing so allows these apps to take advantage of performance enhancements and ensure that they can continue to operate as expected.

.NET apps undergoing a [migration from the in-process model to the isolated worker](#) may encounter this event temporarily during that process. When `FUNCTIONS_WORKER_RUNTIME` is updated to `dotnet-isolated`, but the application is still using an in-process model payload, this event may appear until the migration is completed. See the migration guidance for instructions on using deployment slots to prevent this event from appearing in your production environment.

How to resolve the event

The event message indicates the current value of `FUNCTIONS_WORKER_RUNTIME` and the detected runtime metadata from the app payload. These values must be aligned, either by deploying an application payload of the appropriate type or by updating the setting to an expected value.

For most applications, the correct resolution is to update the value of `FUNCTIONS_WORKER_RUNTIME`. To do so, on your function app in Azure, set the `FUNCTIONS_WORKER_RUNTIME` application setting to the expected value for your application payload. The expected value is not necessarily the same as the detected runtime metadata, though in many cases it will be. Use the following table to determine the correct value to use:

[+] Expand table

Detected payload	Expected <code>FUNCTIONS_WORKER_RUNTIME</code> value
csharp	dotnet
custom	custom
dotnet	dotnet
dotnet-isolated	dotnet-isolated
java	java
node	node
powershell	powershell
python	python
Any multi-stack payload ¹	dotnet

¹ A multi-stack payload is a comma-separated list of stack values. Multi-stack payloads are only supported for [Logic Apps Standard](#).

When running locally in the Azure Functions Core Tools, you should also add `FUNCTIONS_WORKER_RUNTIME` to the [local.settings.json file](#).

For apps following a migration guide, see that guide for relevant instructions. [Migrating .NET applications to the isolated worker model](#) involves first setting `FUNCTIONS_WORKER_RUNTIME` to `dotnet-isolated` before deploying the updated application payload, and this event may appear temporarily between those steps.

When to suppress the event

This event shouldn't be suppressed.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

host.json reference for Azure Functions 2.x and later

Article • 05/16/2024

The host.json metadata file contains configuration options that affect all functions in a function app instance. This article lists the settings that are available starting with version 2.x of the Azure Functions runtime.

ⓘ Note

This article is for Azure Functions 2.x and later versions. For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

Other function app configuration options are managed depending on where the function app runs:

- **Deployed to Azure:** in your [application settings](#)
- **On your local computer:** in the [local.settings.json](#) file.

Configurations in host.json related to bindings are applied equally to each function in the function app.

You can also [override or apply settings per environment](#) using application settings.

Sample host.json file

The following sample *host.json* file for version 2.x+ has all possible options specified (excluding any that are for internal use only).

JSON

```
{  
    "version": "2.0",  
    "aggregator": {  
        "batchSize": 1000,  
        "flushTimeout": "00:00:30"  
    },  
    "concurrency": {  
        "dynamicConcurrencyEnabled": true,  
        "snapshotPersistenceEnabled": true  
    },  
    "extensions": {  
        "blobs": {}  
    }  
}
```

```
"cosmosDb": {},
"durableTask": {},
"eventHubs": {},
"http": {},
"queues": {},
"sendGrid": {},
"serviceBus": {}
},
"extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.0.0, 5.0.0)"
},
"functions": [ "QueueProcessor", "GitHubWebHook" ],
"functionTimeout": "00:05:00",
"healthMonitor": {
    "enabled": true,
    "healthCheckInterval": "00:00:10",
    "healthCheckWindow": "00:02:00",
    "healthCheckThreshold": 6,
    "counterThreshold": 0.80
},
"logging": {
    "fileLoggingMode": "debugOnly",
    "logLevel": {
        "Function.MyFunction": "Information",
        "default": "None"
    },
    "applicationInsights": {
        "samplingSettings": {
            "isEnabled": true,
            "maxTelemetryItemsPerSecond" : 20,
            "evaluationInterval": "01:00:00",
            "initialSamplingPercentage": 100.0,
            "samplingPercentageIncreaseTimeout" : "00:00:01",
            "samplingPercentageDecreaseTimeout" : "00:00:01",
            "minSamplingPercentage": 0.1,
            "maxSamplingPercentage": 100.0,
            "movingAverageRatio": 1.0,
            "excludedTypes" : "Dependency;Event",
            "includedTypes" : "PageView;Trace"
        },
        "dependencyTrackingOptions": {
            "enableSqlCommandTextInstrumentation": true
        },
        "enableLiveMetrics": true,
        "enableDependencyTracking": true,
        "enablePerformanceCountersCollection": true,
        "httpAutoCollectionOptions": {
            "enableHttpTriggerExtendedInfoCollection": true,
            "enableW3CDistributedTracing": true,
            "enableResponseHeaderInjection": true
        },
        "snapshotConfiguration": {
            "agentEndpoint": null,
            "captureSnapshotMemoryWeight": 0.5,
            "captureSnapshotFileWeight": 0.5
        }
    }
}
```

```

        "failedRequestLimit": 3,
        "handleUntrackedExceptions": true,
        "isEnabled": true,
        "isEnabledInDeveloperMode": false,
        "isEnabledWhenProfiling": true,
        "isExceptionSnappointsEnabled": false,
        "isLowPrioritySnapshotUploader": true,
        "maximumCollectionPlanSize": 50,
        "maximumSnapshotsRequired": 3,
        "problemCounterResetInterval": "24:00:00",
        "provideAnonymousTelemetry": true,
        "reconnectInterval": "00:15:00",
        "shadowCopyFolder": null,
        "shareUploaderProcess": true,
        "snapshotInLowPriorityThread": true,
        "snapshotsPerDayLimit": 30,
        "snapshotsPerTenMinutesLimit": 1,
        "tempFolder": null,
        "thresholdForSnapshotting": 1,
        "uploaderProxy": null
    }
},
{
    "managedDependency": {
        "enabled": true
    },
    "singleton": {
        "lockPeriod": "00:00:15",
        "listenerLockPeriod": "00:01:00",
        "listenerLockRecoveryPollingInterval": "00:01:00",
        "lockAcquisitionTimeout": "00:01:00",
        "lockAcquisitionPollingInterval": "00:00:03"
    },
    "telemetryMode": "OpenTelemetry",
    "watchDirectories": [ "Shared", "Test" ],
    "watchFiles": [ "myFile.txt" ]
}

```

The following sections of this article explain each top-level property. All are optional unless otherwise indicated.

aggregator

Specifies how many function invocations are aggregated when [calculating metrics for Application Insights](#).

JSON

```
{
    "aggregator": {
        "batchSize": 1000,
```

```
        "flushTimeout": "00:00:30"
    }
}
```

[+] Expand table

Property	Default	Description
batchSize	1000	Maximum number of requests to aggregate.
flushTimeout	00:00:30	Maximum time period to aggregate.

Function invocations are aggregated when the first of the two limits are reached.

applicationInsights

This setting is a child of [logging](#).

Controls options for Application Insights, including [sampling options](#).

For the complete JSON structure, see the earlier [example host.json file](#).

ⓘ Note

Log sampling may cause some executions to not show up in the Application Insights monitor blade. To avoid log sampling, add `excludedTypes: "Request"` to the `samplingSettings` value.

[+] Expand table

Property	Default	Description
samplingSettings	n/a	See applicationInsights.samplingSettings .
dependencyTrackingOptions	n/a	See applicationInsights.dependencyTrackingOptions .
enableLiveMetrics	true	Enables live metrics collection.
enableDependencyTracking	true	Enables dependency tracking.
enablePerformanceCountersCollection	true	Enables Kudu performance counters collection.
liveMetricsInitializationDelay	00:00:15	For internal use only.

Property	Default	Description
httpAutoCollectionOptions	n/a	See applicationInsights.httpAutoCollectionOptions .
snapshotConfiguration	n/a	See applicationInsights.snapshotConfiguration .

applicationInsights.samplingSettings

For more information about these settings, see [Sampling in Application Insights](#).

[\[\] Expand table](#)

Property	Default	Description
isEnabled	true	Enables or disables sampling.
maxTelemetryItemsPerSecond	20	The target number of telemetry items logged per second on each server host. If your app runs on many hosts, reduce this value to remain within your overall target rate of traffic.
evaluationInterval	01:00:00	The interval at which the current rate of telemetry is reevaluated. Evaluation is performed as a moving average. You might want to shorten this interval if your telemetry is liable to sudden bursts.
initialSamplingPercentage	100.0	The initial sampling percentage applied at the start of the sampling process to dynamically vary the percentage. Don't reduce value while you're debugging.
samplingPercentageIncreaseTimeout	00:00:01	When the sampling percentage value changes, this property determines how soon afterwards Application Insights is allowed to raise sampling percentage again to capture more data.
samplingPercentageDecreaseTimeout	00:00:01	When the sampling percentage value changes, this property determines how soon afterwards Application Insights is allowed to lower sampling percentage again to capture less data.
minSamplingPercentage	0.1	As sampling percentage varies, this property determines the minimum allowed sampling percentage.

Property	Default	Description
maxSamplingPercentage	100.0	As sampling percentage varies, this property determines the maximum allowed sampling percentage.
movingAverageRatio	1.0	In the calculation of the moving average, the weight assigned to the most recent value. Use a value equal to or less than 1. Smaller values make the algorithm less reactive to sudden changes.
excludedTypes	null	A semi-colon delimited list of types that you don't want to be sampled. Recognized types are: Dependency, Event, Exception, PageView, Request, and Trace. All instances of the specified types are transmitted; the types that aren't specified are sampled.
includedTypes	null	A semi-colon delimited list of types that you want to be sampled; an empty list implies all types. Type listed in excludedTypes override types listed here. Recognized types are: Dependency, Event, Exception, PageView, Request, and Trace. Instances of the specified types are sampled; the types that aren't specified or implied are transmitted without sampling.

applicationInsights.httpAutoCollectionOptions

[Expand table](#)

Property	Default	Description
enableHttpTriggerExtendedInfoCollection	true	Enables or disables extended HTTP request information for HTTP triggers: incoming request correlation headers, multi-instrumentation keys support, HTTP method, path, and response.
enableW3CDistributedTracing	true	Enables or disables support of W3C distributed tracing protocol (and turns on legacy correlation schema). Enabled by default if enableHttpTriggerExtendedInfoCollection is true. If enableHttpTriggerExtendedInfoCollection

Property	Default	Description
		is false, this flag applies to outgoing requests only, not incoming requests.
enableResponseHeaderInjection	true	Enables or disables injection of multi-component correlation headers into responses. Enabling injection allows Application Insights to construct an Application Map to when several instrumentation keys are used. Enabled by default if <code>enableHttpTriggerExtendedInfoCollection</code> is true. This setting doesn't apply if <code>enableHttpTriggerExtendedInfoCollection</code> is false.

applicationInsights.dependencyTrackingOptions

[+] Expand table

Property	Default	Description
enableSqlCommandTextInstrumentation	false	Enables collection of the full text of SQL queries, which is disabled by default. For more information on collecting SQL query text, see Advanced SQL tracking to get full SQL query .

applicationInsights.snapshotConfiguration

For more information on snapshots, see [Debug snapshots on exceptions in .NET apps](#) and [Troubleshoot problems enabling Application Insights Snapshot Debugger or viewing snapshots](#).

[+] Expand table

Property	Default	Description
agentEndpoint	null	The endpoint used to connect to the Application Insights Snapshot Debugger service. If null, a default endpoint is used.
captureSnapshotMemoryWeight	0.5	The weight given to the current process memory size when checking if there's enough memory to take a snapshot. The expected value is a greater

Property	Default	Description
		than 0 proper fraction ($0 < \text{CaptureSnapshotMemoryWeight} < 1$).
failedRequestLimit	3	The limit on the number of failed requests to request snapshots before the telemetry processor is disabled.
handleUntrackedExceptions	true	Enables or disables tracking of exceptions that aren't tracked by Application Insights telemetry.
isEnabled	true	Enables or disables snapshot collection
isEnabledInDeveloperMode	false	Enables or disables snapshot collection is enabled in developer mode.
isEnabledWhenProfiling	true	Enables or disables snapshot creation even if the Application Insights Profiler is collecting a detailed profiling session.
isExceptionSnappointsEnabled	false	Enables or disables filtering of exceptions.
isLowPrioritySnapshotUploader	true	Determines whether to run the SnapshotUploader process at below normal priority.
maximumCollectionPlanSize	50	The maximum number of problems that we can track at any time in a range from one to 999.
maximumSnapshotsRequired	3	The maximum number of snapshots collected for a single problem, in a range from one to 999. A problem may be thought of as an individual throw statement in your application. Once the number of snapshots collected for a problem reaches this value, no more snapshots will be collected for that problem until problem counters are reset (see <code>problemCounterResetInterval</code>) and the <code>thresholdForSnapshotting</code> limit is reached again.
problemCounterResetInterval	24:00:00	How often to reset the problem counters in a range from one minute to seven days. When this interval is reached, all problem counts are reset to zero. Existing problems that have already reached the threshold for doing snapshots, but haven't yet generated the number of snapshots in <code>maximumSnapshotsRequired</code> , remain active.
provideAnonymousTelemetry	true	Determines whether to send anonymous usage and error telemetry to Microsoft. This telemetry may be used if you contact Microsoft to help troubleshoot

Property	Default	Description
		problems with the Snapshot Debugger. It's also used to monitor usage patterns.
reconnectInterval	00:15:00	How often we reconnect to the Snapshot Debugger endpoint. Allowable range is one minute to one day.
shadowCopyFolder	null	Specifies the folder to use for shadow copying binaries. If not set, the folders specified by the following environment variables are tried in order: Fabric_Folder_App_Temp, LOCALAPPDATA, APPDATA, TEMP.
shareUploaderProcess	true	If true, only one instance of SnapshotUploader will collect and upload snapshots for multiple apps that share the InstrumentationKey. If set to false, the SnapshotUploader will be unique for each (ProcessName, InstrumentationKey) tuple.
snapshotInLowPriorityThread	true	Determines whether or not to process snapshots in a low IO priority thread. Creating a snapshot is a fast operation but, in order to upload a snapshot to the Snapshot Debugger service, it must first be written to disk as a minidump. That happens in the SnapshotUploader process. Setting this value to true uses low-priority IO to write the minidump, which won't compete with your application for resources. Setting this value to false speeds up minidump creation at the expense of slowing down your application.
snapshotsPerDayLimit	30	The maximum number of snapshots allowed in one day (24 hours). This limit is also enforced on the Application Insights service side. Uploads are rate limited to 50 per day per application (that is, per instrumentation key). This value helps prevent creating additional snapshots that will eventually be rejected during upload. A value of zero removes the limit entirely, which isn't recommended.
snapshotsPerTenMinutesLimit	1	The maximum number of snapshots allowed in 10 minutes. Although there's no upper bound on this value, exercise caution increasing it on production workloads because it could impact the performance of your application. Creating a snapshot is fast, but creating a minidump of the snapshot and uploading it to the Snapshot Debugger service is a much

Property	Default	Description
		slower operation that will compete with your application for resources (both CPU and I/O).
tempFolder	null	Specifies the folder to write minidumps and uploader log files. If not set, then %TEMP%\Dumps is used.
thresholdForSnapshotting	1	How many times Application Insights needs to see an exception before it asks for snapshots.
uploaderProxy	null	Overrides the proxy server used in the Snapshot Uploader process. You may need to use this setting if your application connects to the internet via a proxy server. The Snapshot Collector runs within your application's process and will use the same proxy settings. However, the Snapshot Uploader runs as a separate process and you may need to configure the proxy server manually. If this value is null, then Snapshot Collector will attempt to autodetect the proxy's address by examining <code>System.Net.WebRequest.DefaultWebProxy</code> and passing on the value to the Snapshot Uploader. If this value isn't null, then autodetection isn't used and the proxy server specified here will be used in the Snapshot Uploader.

blobs

Configuration settings can be found in [Storage blob triggers and bindings](#).

console

This setting is a child of [logging](#). It controls the console logging when not in debugging mode.

```
JSON
{
  "logging": {
    ...
    "console": {
      "isEnabled": false,
      "DisableColors": true
    },
    ...
  }
}
```

```
    }  
}
```

[+] Expand table

Property	Default	Description
DisableColors	false	Suppresses log formatting in the container logs on Linux. Set to true if you're seeing unwanted ANSI control characters in the container logs when running on Linux.
isEnabled	false	Enables or disables console logging.

Azure Cosmos DB

Configuration settings can be found in [Azure Cosmos DB triggers and bindings](#).

customHandler

Configuration settings for a custom handler. For more information, see [Azure Functions custom handlers](#).

JSON

```
"customHandler": {  
  "description": {  
    "defaultExecutablePath": "server",  
    "workingDirectory": "handler",  
    "arguments": [ "--port", "%FUNCTIONS_CUSTOMHANDLER_PORT%" ]  
  },  
  "enableForwardingHttpRequest": false  
}
```

[+] Expand table

Property	Default	Description
defaultExecutablePath	n/a	The executable to start as the custom handler process. It's a required setting when using custom handlers and its value is relative to the function app root.
workingDirectory	<i>function app root</i>	The working directory in which to start the custom handler process. It's an optional setting and its value is relative to the function app root.

Property	Default	Description
arguments	n/a	An array of command line arguments to pass to the custom handler process.
enableForwardingHttpRequest	false	If set, all functions that consist of only an HTTP trigger and HTTP output is forwarded the original HTTP request instead of the custom handler request payload .

durableTask

Configuration setting can be found in [bindings for Durable Functions](#).

Concurrency

Enables dynamic concurrency for specific bindings in your function app. For more information, see [Dynamic concurrency](#).

JSON
<pre>{ "concurrency": { "dynamicConcurrencyEnabled": true, "snapshotPersistenceEnabled": true } }</pre>

[\[+\] Expand table](#)

Property	Default	Description
dynamicConcurrencyEnabled	false	Enables dynamic concurrency behaviors for all triggers supported by this feature, which is off by default.
snapshotPersistenceEnabled	true	Learned concurrency values are periodically persisted to storage so new instances start from those values instead of starting from 1 and having to redo the learning.

eventHub

Configuration settings can be found in [Event Hub triggers and bindings](#).

extensions

Property that returns an object that contains all of the binding-specific settings, such as [http](#) and [eventHub](#).

extensionBundle

Extension bundles let you add a compatible set of Functions binding extensions to your function app. To learn more, see [Extension bundles for local development](#).

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[4.0.0, 5.0.0)"  
    }  
}
```

The following properties are available in `extensionBundle`:

[+] [Expand table](#)

Property	Description
<code>id</code>	The namespace for Microsoft Azure Functions extension bundles.
<code>version</code>	The version range of the bundle to install. The Functions runtime always picks the maximum permissible version defined by the version range or interval. For example, a <code>version</code> value range of <code>[4.0.0, 5.0.0)</code> allows all bundle versions from 4.0.0 up to but not including 5.0.0. For more information, see the interval notation for specifying version ranges .

functions

A list of functions that the job host runs. An empty array means run all functions. Intended for use only when [running locally](#). In function apps in Azure, you should instead follow the steps in [How to disable functions in Azure Functions](#) to disable specific functions rather than using this setting.

JSON

```
{  
    "functions": [ "QueueProcessor", "GitHubWebHook" ]  
}
```

functionTimeout

Indicates the timeout duration for all function executions. It follows the timespan string format.

[+] Expand table

Plan type	Default (min)	Maximum (min)
Consumption	5	10
Premium ¹	30	-1 (unbounded) ²
Dedicated (App Service)	30	-1 (unbounded) ²

¹ Premium plan execution is only guaranteed for 60 minutes, but technically unbounded.

² A value of -1 indicates unbounded execution, but keeping a fixed upper bound is recommended.

JSON

```
{  
    "functionTimeout": "00:05:00"  
}
```

healthMonitor

Configuration settings for [Host health monitor](#).

JSON

```
{  
    "healthMonitor": {  
        "enabled": true,  
        "healthCheckInterval": "00:00:10",  
        "healthCheckWindow": "00:02:00",  
        "healthCheckThreshold": 6,  
        "counterThreshold": 0.80  
    }  
}
```

[\[\] Expand table](#)

Property	Default	Description
enabled	true	Specifies whether the feature is enabled.
healthCheckInterval	10 seconds	The time interval between the periodic background health checks.
healthCheckWindow	2 minutes	A sliding time window used in conjunction with the <code>healthCheckThreshold</code> setting.
healthCheckThreshold	6	Maximum number of times the health check can fail before a host recycle is initiated.
counterThreshold	0.80	The threshold at which a performance counter will be considered unhealthy.

http

Configuration settings can be found in [http triggers and bindings](#).

logging

Controls the logging behaviors of the function app, including Application Insights.

JSON

```
"logging": {  
    "fileLoggingMode": "debugOnly",  
    "logLevel": {  
        "Function.MyFunction": "Information",  
        "default": "None"  
    },  
    "console": {  
        ...  
    },  
    "applicationInsights": {  
        ...  
    }  
}
```

[\[\] Expand table](#)

Property	Default	Description
fileLoggingMode	debugOnly	Determines the file logging behavior when running in Azure. Options are <code>never</code> , <code>always</code> , and <code>debugOnly</code> . This setting isn't used when running locally. When possible, you should use Application Insights when debugging your functions in Azure. Using <code>always</code> negatively impacts your app's cold start behavior and data throughput. The default <code>debugOnly</code> setting generates log files when you're debugging using the Azure portal.
logLevel	n/a	Object that defines the log category filtering for functions in the app. This setting lets you filter logging for specific functions. For more information, see Configure log levels .
console	n/a	The <code>console</code> logging setting.
applicationInsights	n/a	The <code>applicationInsights</code> setting.

managedDependency

Managed dependency is a feature that is currently only supported with PowerShell based functions. It enables dependencies to be automatically managed by the service. When the `enabled` property is set to `true`, the `requirements.psd1` file is processed. Dependencies are updated when any minor versions are released. For more information, see [Managed dependency](#) in the PowerShell article.

JSON

```
{
  "managedDependency": {
    "enabled": true
  }
}
```

queues

Configuration settings can be found in [Storage queue triggers and bindings](#).

sendGrid

Configuration setting can be found in [SendGrid triggers and bindings](#).

serviceBus

Configuration setting can be found in [Service Bus triggers and bindings](#).

singleton

Configuration settings for Singleton lock behavior. For more information, see [GitHub issue about singleton support ↗](#).

JSON

```
{  
  "singleton": {  
    "lockPeriod": "00:00:15",  
    "listenerLockPeriod": "00:01:00",  
    "listenerLockRecoveryPollingInterval": "00:01:00",  
    "lockAcquisitionTimeout": "00:01:00",  
    "lockAcquisitionPollingInterval": "00:00:03"  
  }  
}
```

[] [Expand table](#)

Property	Default	Description
lockPeriod	00:00:15	The period that function level locks are taken for. The locks auto-renew.
listenerLockPeriod	00:01:00	The period that listener locks are taken for.
listenerLockRecoveryPollingInterval	00:01:00	The time interval used for listener lock recovery if a listener lock couldn't be acquired on startup.
lockAcquisitionTimeout	00:01:00	The maximum amount of time the runtime will try to acquire a lock.
lockAcquisitionPollingInterval	n/a	The interval between lock acquisition attempts.

telemetryMode

This feature is currently in preview.

Used to enable output of logs and traces in an OpenTelemetry output format to one or more endpoints that support OpenTelemetry. When this setting is set to `OpenTelemetry`, OpenTelemetry output is used. By default without this setting, all logs, traces, and events are sent to Application Insights using the standard outputs. For more information, see [Use OpenTelemetry with Azure Functions](#).

version

This value indicates the schema version of host.json. The version string "version": "2.0" is required for a function app that targets the v2 runtime, or a later version. There are no host.json schema changes between v2 and v3.

watchDirectories

A set of [shared code directories](#) that should be monitored for changes. Ensures that when code in these directories is changed, the changes are picked up by your functions.

JSON

```
{  
  "watchDirectories": [ "Shared" ]  
}
```

watchFiles

An array of one or more names of files that are monitored for changes that require your app to restart. This guarantees that when code in these files is changed, the updates are picked up by your functions.

JSON

```
{  
  "watchFiles": [ "myFile.txt" ]  
}
```

Override host.json values

There may be instances where you wish to configure or modify specific settings in a host.json file for a specific environment, without changing the host.json file itself. You can override specific host.json values by creating an equivalent value as an application setting. When the runtime finds an application setting in the format

AzureFunctionsJobHost__path__to__setting, it overrides the equivalent host.json setting located at path.to.setting in the JSON. When expressed as an application setting, the dot (.) used to indicate JSON hierarchy is replaced by a double underscore (__).

For example, say that you wanted to disable Application Insight sampling when running locally. If you changed the local host.json file to disable Application Insights, this change might get pushed to your production app during deployment. The safer way to do this is to instead create an application setting as

"AzureFunctionsJobHost_logging_applicationInsights_samplingSettings_isEnabled": "false" in the local.settings.json file. You can see this in the following local.settings.json file, which doesn't get published:

```
JSON

{
    "IsEncrypted": false,
    "Values": {
        "AzureWebJobsStorage": "{storage-account-connection-string}",
        "FUNCTIONS_WORKER_RUNTIME": "{language-runtime}",

    "AzureFunctionsJobHost_logging_applicationInsights_samplingSettings_isEnabled": "false"
    }
}
```

Overriding host.json settings using environment variables follows the ASP.NET Core naming conventions. When the element structure includes an array, the numeric array index should be treated as an additional element name in this path. For more information, see [Naming of environment variables](#).

Next steps

[Learn how to update the host.json file](#)

[See global settings in environment variables](#)

host.json reference for Azure Functions 1.x

Article • 03/16/2022

The `host.json` metadata file contains configuration options that affect all functions in a function app instance. This article lists the settings that are available for the version 1.x runtime. The JSON schema is at <http://json.schemastore.org/host>.

ⓘ Note

This article is for Azure Functions 1.x. For a reference of `host.json` in Functions 2.x and later, see [host.json reference for Azure Functions 2.x](#).

Other function app configuration options are managed in your [app settings](#).

Some `host.json` settings are only used when running locally in the [local.settings.json](#) file.

Sample host.json file

The following sample `host.json` files have all possible options specified.

JSON

```
{  
    "aggregator": {  
        "batchSize": 1000,  
        "flushTimeout": "00:00:30"  
    },  
    "applicationInsights": {  
        "sampling": {  
            "isEnabled": true,  
            "maxTelemetryItemsPerSecond" : 5  
        }  
    },  
    "documentDB": {  
        "connectionMode": "Gateway",  
        "protocol": "Https",  
        "leaseOptions": {  
            "leasePrefix": "prefix"  
        }  
    },  
    "eventHub": {  
        "maxBatchSize": 64,  
        "prefetchCount": 256,  
        "batchCheckpointFrequency": 1  
    }  
}
```

```
},
"functions": [ "QueueProcessor", "GitHubWebHook" ],
"functionTimeout": "00:05:00",
"healthMonitor": {
    "enabled": true,
    "healthCheckInterval": "00:00:10",
    "healthCheckWindow": "00:02:00",
    "healthCheckThreshold": 6,
    "counterThreshold": 0.80
},
"http": {
    "routePrefix": "api",
    "maxOutstandingRequests": 20,
    "maxConcurrentRequests": 10,
    "dynamicThrottlesEnabled": false
},
"id": "9f4ea53c5136457d883d685e57164f08",
"logger": {
    "categoryFilter": {
        "defaultLevel": "Information",
        "categoryLevels": {
            "Host": "Error",
            "Function": "Error",
            "Host.Aggregator": "Information"
        }
    }
},
"queues": {
    "maxPollingInterval": 2000,
    "visibilityTimeout" : "00:00:30",
    "batchSize": 16,
    "maxDequeueCount": 5,
    "newBatchThreshold": 8
},
"sendGrid": {
    "from": "Contoso Group <admin@contoso.com>"
},
"serviceBus": {
    "maxConcurrentCalls": 16,
    "prefetchCount": 100,
    "autoRenewTimeout": "00:05:00",
    "autoComplete": true
},
"singleton": {
    "lockPeriod": "00:00:15",
    "listenerLockPeriod": "00:01:00",
    "listenerLockRecoveryPollingInterval": "00:01:00",
    "lockAcquisitionTimeout": "00:01:00",
    "lockAcquisitionPollingInterval": "00:00:03"
},
"tracing": {
    "consoleLevel": "verbose",
    "fileLoggingMode": "debugOnly"
},
```

```
    "watchDirectories": [ "Shared" ],  
}
```

The following sections of this article explain each top-level property. All are optional unless otherwise indicated.

aggregator

Specifies how many function invocations are aggregated when [calculating metrics for Application Insights](#).

JSON

```
{  
  "aggregator": {  
    "batchSize": 1000,  
    "flushTimeout": "00:00:30"  
  }  
}
```

Property	Default	Description
batchSize	1000	Maximum number of requests to aggregate.
flushTimeout	00:00:30	Maximum time period to aggregate.

Function invocations are aggregated when the first of the two limits are reached.

applicationInsights

Controls the [sampling feature in Application Insights](#).

JSON

```
{  
  "applicationInsights": {  
    "sampling": {  
      "isEnabled": true,  
      "maxTelemetryItemsPerSecond" : 5  
    }  
  }  
}
```

Property	Default	Description
----------	---------	-------------

Property	Default	Description
isEnabled	true	Enables or disables sampling.
maxTelemetryItemsPerSecond	5	The threshold at which sampling begins.

DocumentDB

Configuration settings for the [Azure Cosmos DB trigger and bindings](#).

JSON

```
{
  "documentDB": {
    "connectionMode": "Gateway",
    "protocol": "Https",
    "leaseOptions": {
      "leasePrefix": "prefix1"
    }
  }
}
```

Property	Default	Description
GatewayMode	Gateway	The connection mode used by the function when connecting to the Azure Cosmos DB service. Options are <code>Direct</code> and <code>Gateway</code>
Protocol	Https	The connection protocol used by the function when connection to the Azure Cosmos DB service. Read Here for an explanation of both modes
leasePrefix	n/a	Lease prefix to use across all functions in an app.

durableTask

Configuration settings for [Durable Functions](#).

 **Note**

All major versions of Durable Functions are supported on all versions of the Azure Functions runtime. However, the schema of the host.json configuration is slightly different depending on the version of the Azure Functions runtime and the Durable Functions extension version you use. The following examples are for use with Azure Functions 2.0 and 3.0. In both examples, if you're using Azure Functions 1.0, the

available settings are the same, but the "durableTask" section of the host.json should go in the root of the host.json configuration instead of as a field under "extensions".

Durable Functions 2.x

JSON

```
{  
  "extensions": {  
    "durableTask": {  
      "hubName": "MyTaskHub",  
      "storageProvider": {  
        "connectionStringName": "AzureWebJobsStorage",  
        "controlQueueBatchSize": 32,  
        "controlQueueBufferThreshold": 256,  
        "controlQueueVisibilityTimeout": "00:05:00",  
        "maxQueuePollingInterval": "00:00:30",  
        "partitionCount": 4,  
        "trackingStoreConnectionStringName": "TrackingStorage",  
        "trackingStoreNamePrefix": "DurableTask",  
        "useLegacyPartitionManagement": true,  
        "workItemQueueVisibilityTimeout": "00:05:00",  
      },  
      "tracing": {  
        "traceInputsAndOutputs": false,  
        "traceReplayEvents": false,  
      },  
      "notifications": {  
        "eventGrid": {  
          "topicEndpoint": "https://topic_name.westus2-  
1.eventgrid.azure.net/api/events",  
          "keySettingName": "EventGridKey",  
          "publishRetryCount": 3,  
          "publishRetryInterval": "00:00:30",  
          "publishEventTypes": [  
            "Started",  
            "Pending",  
            "Failed",  
            "Terminated"  
          ]  
        }  
      },  
      "maxConcurrentActivityFunctions": 10,  
      "maxConcurrentOrchestratorFunctions": 10,  
      "extendedSessionsEnabled": false,  
      "extendedSessionIdleTimeoutInSeconds": 30,  
      "useAppLease": true,  
      "useGracefulShutdown": false,  
      "maxEntityOperationBatchSize": 50  
    }  
  }
```

```
}
```

Task hub names must start with a letter and consist of only letters and numbers. If not specified, the default task hub name for a function app is **TestHubName**. For more information, see [Task hubs](#).

Property	Default	Description
hubName	TestHubName (DurableFunctionsHub if using Durable Functions 1.x)	Alternate task hub names can be used to isolate multiple Durable Functions applications from each other, even if they're using the same storage backend.
controlQueueBatchSize	32	The number of messages to pull from the control queue at a time.
controlQueueBufferThreshold	Consumption plan for Python: 32 Consumption plan for JavaScript and C#: 128 Dedicated/Premium plan: 256	The number of control queue messages that can be buffered in memory at a time, at which point the dispatcher will wait before dequeuing any additional messages.
partitionCount	4	The partition count for the control queue. May be a positive integer between 1 and 16.
controlQueueVisibilityTimeout	5 minutes	The visibility timeout of dequeued control queue messages.
workItemQueueVisibilityTimeout	5 minutes	The visibility timeout of dequeued work item queue messages.
maxConcurrentActivityFunctions	Consumption plan: 10 Dedicated/Premium plan: 10X the number of processors on the current machine	The maximum number of activity functions that can be processed concurrently on a single host instance.
maxConcurrentOrchestratorFunctions	Consumption plan: 5 Dedicated/Premium plan: 10X the number of processors on the current machine	The maximum number of orchestrator functions that can be processed concurrently on a single host instance.

Property	Default	Description
maxQueuePollingInterval	30 seconds	The maximum control and work-item queue polling interval in the <i>hh:mm:ss</i> format. Higher values can result in higher message processing latencies. Lower values can result in higher storage costs because of increased storage transactions.
connectionName (2.7.0 and later) connectionStringName (2.x) azureStorageConnectionStringName (1.x)	AzureWebJobsStorage	The name of an app setting or setting collection that specifies how to connect to the underlying Azure Storage resources. When a single app setting is provided, it should be an Azure Storage connection string.
trackingStoreConnectionName (2.7.0 and later) trackingStoreConnectionStringName		The name of an app setting or setting collection that specifies how to connect to the History and Instances tables. When a single app setting is provided, it should be an Azure Storage connection string. If not specified, the <code>connectionStringName</code> (Durable 2.x) or <code>azureStorageConnectionStringName</code> (Durable 1.x) connection is used.
trackingStoreNamePrefix		The prefix to use for the History and Instances tables when <code>trackingStoreConnectionStringName</code> is specified. If not set, the default prefix value will be <code>DurableTask</code> . If <code>trackingStoreConnectionStringName</code> is not specified, then the History and Instances tables will use the <code>hubName</code> value as their prefix, and any setting for <code>trackingStoreNamePrefix</code> will be ignored.

Property	Default	Description
traceInputsAndOutputs	false	A value indicating whether to trace the inputs and outputs of function calls. The default behavior when tracing function execution events is to include the number of bytes in the serialized inputs and outputs for function calls. This behavior provides minimal information about what the inputs and outputs look like without bloating the logs or inadvertently exposing sensitive information. Setting this property to true causes the default function logging to log the entire contents of function inputs and outputs.
traceReplayEvents	false	A value indicating whether to write orchestration replay events to Application Insights.
eventGridTopicEndpoint		The URL of an Azure Event Grid custom topic endpoint. When this property is set, orchestration life-cycle notification events are published to this endpoint. This property supports App Settings resolution.
eventGridKeySettingName		The name of the app setting containing the key used for authenticating with the Azure Event Grid custom topic at <code>EventGridTopicEndpoint</code> .
eventGridPublishRetryCount	0	The number of times to retry if publishing to the Event Grid Topic fails.
eventGridPublishRetryInterval	5 minutes	The Event Grid publishes retry interval in the <i>hh:mm:ss</i> format.
eventGridPublishEventTypes		A list of event types to publish to Event Grid. If not specified, all event types will be published. Allowed values include <code>Started</code> , <code>Completed</code> , <code>Failed</code> , <code>Terminated</code> .

Property	Default	Description
useAppLease	true	When set to <code>true</code> , apps will require acquiring an app-level blob lease before processing task hub messages. For more information, see the disaster recovery and geo-distribution documentation. Available starting in v2.3.0.
useLegacyPartitionManagement	false	When set to <code>false</code> , uses a partition management algorithm that reduces the possibility of duplicate function execution when scaling out. Available starting in v2.3.0.
useGracefulShutdown	false	(Preview) Enable gracefully shutting down to reduce the chance of host shutdowns failing in-process function executions.
maxEntityOperationBatchSize(2.6.1)	Consumption plan: 50 Dedicated/Premium plan: 5000	The maximum number of entity operations that are processed as a batch . If set to 1, batching is disabled, and each operation message is processed by a separate function invocation.

Many of these settings are for optimizing performance. For more information, see [Performance and scale](#).

eventHub

Configuration settings for [Event Hub triggers and bindings](#).

functions

A list of functions that the job host runs. An empty array means run all functions. Intended for use only when [running locally](#). In function apps in Azure, you should instead follow the steps in [How to disable functions in Azure Functions](#) to disable specific functions rather than using this setting.

JSON

```
{
  "Functions": [ "QueueProcessor", "GitHubWebHook" ]
```

```
}
```

functionTimeout

Indicates the timeout duration for all functions. In a serverless Consumption plan, the valid range is from 1 second to 10 minutes, and the default value is 5 minutes. In an App Service plan, there is no overall limit and the default is *null*, which indicates no timeout.

JSON

```
{  
    "functionTimeout": "00:05:00"  
}
```

healthMonitor

Configuration settings for [Host health monitor](#).

```
{  
    "healthMonitor": {  
        "enabled": true,  
        "healthCheckInterval": "00:00:10",  
        "healthCheckWindow": "00:02:00",  
        "healthCheckThreshold": 6,  
        "counterThreshold": 0.80  
    }  
}
```

Property	Default	Description
enabled	true	Specifies whether the feature is enabled.
healthCheckInterval	10 seconds	The time interval between the periodic background health checks.
healthCheckWindow	2 minutes	A sliding time window used in conjunction with the <code>healthCheckThreshold</code> setting.
healthCheckThreshold	6	Maximum number of times the health check can fail before a host recycle is initiated.
counterThreshold	0.80	The threshold at which a performance counter will be considered unhealthy.

http

Configuration settings for [http triggers and bindings](#).

JSON

```
{  
    "http": {  
        "routePrefix": "api",  
        "maxOutstandingRequests": 200,  
        "maxConcurrentRequests": 100,  
        "dynamicThrottlesEnabled": true  
    }  
}
```

Property	Default	Description
dynamicThrottlesEnabled	false	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels.
maxConcurrentRequests	unbounded (-1)	The maximum number of HTTP functions that will be executed in parallel. This allows you to control concurrency, which can help manage resource utilization. For example, you might have an HTTP function that uses a lot of system resources (memory/cpu/sockets) such that it causes issues when concurrency is too high. Or you might have a function that makes outbound requests to a third party service, and those calls need to be rate limited. In these cases, applying a throttle here can help.
maxOutstandingRequests	unbounded (-1)	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows callers to employ time-based retry strategies, and also helps you to control maximum request latencies. This only controls queuing that occurs within the script host execution path. Other queues such as the ASP.NET request queue will still be in effect and unaffected by this setting.

Property	Default	Description
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.

id

The unique ID for a job host. Can be a lower case GUID with dashes removed. Required when running locally. When running in Azure, we recommend that you not set an ID value. An ID is generated automatically in Azure when `id` is omitted.

If you share a Storage account across multiple function apps, make sure that each function app has a different `id`. You can omit the `id` property or manually set each function app's `id` to a different value. The timer trigger uses a storage lock to ensure that there will be only one timer instance when a function app scales out to multiple instances. If two function apps share the same `id` and each uses a timer trigger, only one timer will run.

JSON

```
{
  "id": "9f4ea53c5136457d883d685e57164f08"
}
```

logger

Controls filtering for logs written by an [ILogger](#) object or by `context.log`.

JSON

```
{
  "logger": {
    "categoryFilter": {
      "defaultLevel": "Information",
      "categoryLevels": {
        "Host": "Error",
        "Function": "Error",
        "Host.Aggregator": "Information"
      }
    }
  }
}
```

Property	Default	Description
categoryFilter	n/a	Specifies filtering by category
defaultLevel	Information	For any categories not specified in the <code>categoryLevels</code> array, send logs at this level and above to Application Insights.
categoryLevels	n/a	An array of categories that specifies the minimum log level to send to Application Insights for each category. The category specified here controls all categories that begin with the same value, and longer values take precedence. In the preceding sample <code>host.json</code> file, all categories that begin with "Host.Aggregator" log at <code>Information</code> level. All other categories that begin with "Host", such as "Host.Executor", log at <code>Error</code> level.

queues

Configuration settings for [Storage queue triggers and bindings](#).

JSON

```
{
  "queues": {
    "maxPollingInterval": 2000,
    "visibilityTimeout" : "00:00:30",
    "batchSize": 16,
    "maxDequeueCount": 5,
    "newBatchThreshold": 8
  }
}
```

Property	Default	Description
maxPollingInterval	60000	The maximum interval in milliseconds between queue polls.
visibilityTimeout	0	The time interval between retries when processing of a message fails.

Property	Default	Description
batchSize	16	<p>The number of queue messages that the Functions runtime retrieves simultaneously and processes in parallel. When the number being processed gets down to the <code>newBatchThreshold</code>, the runtime gets another batch and starts processing those messages. So the maximum number of concurrent messages being processed per function is <code>batchSize</code> plus <code>newBatchThreshold</code>. This limit applies separately to each queue-triggered function.</p> <p>If you want to avoid parallel execution for messages received on one queue, you can set <code>batchSize</code> to 1. However, this setting eliminates concurrency only so long as your function app runs on a single virtual machine (VM). If the function app scales out to multiple VMs, each VM could run one instance of each queue-triggered function.</p> <p>The maximum <code>batchSize</code> is 32.</p>
maxDequeueCount	5	The number of times to try processing a message before moving it to the poison queue.
newBatchThreshold	<code>batchSize/2</code>	Whenever the number of messages being processed concurrently gets down to this number, the runtime retrieves another batch.

SendGrid

Configuration setting for the [SendGrid output binding](#)

JSON
<pre>{ "sendGrid": { "from": "Contoso Group <admin@contoso.com>" } }</pre>

Property	Default	Description
from	n/a	The sender's email address across all functions.

serviceBus

Configuration setting for [Service Bus triggers and bindings](#).

JSON

```
{  
  "serviceBus": {  
    "maxConcurrentCalls": 16,  
    "prefetchCount": 100,  
    "autoRenewTimeout": "00:05:00",  
    "autoComplete": true  
  }  
}
```

Property	Default	Description
maxConcurrentCalls	16	The maximum number of concurrent calls to the callback that the message pump should initiate. By default, the Functions runtime processes multiple messages concurrently. To direct the runtime to process only a single queue or topic message at a time, set <code>maxConcurrentCalls</code> to 1.
prefetchCount	n/a	The default PrefetchCount that will be used by the underlying MessageReceiver.
autoRenewTimeout	00:05:00	The maximum duration within which the message lock will be renewed automatically.
autoComplete	true	When true, the trigger will complete the message processing automatically on successful execution of the operation. When false, it is the responsibility of the function to complete the message before returning.

singleton

Configuration settings for Singleton lock behavior. For more information, see [GitHub issue about singleton support ↗](#).

JSON

```
{  
  "singleton": {  
    "lockPeriod": "00:00:15",  
    "listenerLockPeriod": "00:01:00",  
    "listenerLockRecoveryPollingInterval": "00:01:00",  
    "lockAcquisitionTimeout": "00:01:00",  
    "lockAcquisitionPollingInterval": "00:00:03"  
  }  
}
```

Property	Default	Description
lockPeriod	00:00:15	The period that function level locks are taken for. The locks auto-renew.
listenerLockPeriod	00:01:00	The period that listener locks are taken for.
listenerLockRecoveryPollingInterval	00:01:00	The time interval used for listener lock recovery if a listener lock couldn't be acquired on startup.
lockAcquisitionTimeout	00:01:00	The maximum amount of time the runtime will try to acquire a lock.
lockAcquisitionPollingInterval	n/a	The interval between lock acquisition attempts.

tracing

Version 1.x

Configuration settings for logs that you create by using a `TraceWriter` object. To learn more, see [C# Logging].

JSON

```
{
  "tracing": {
    "consoleLevel": "verbose",
    "fileLoggingMode": "debugOnly"
  }
}
```

Property	Default	Description
consoleLevel	info	The tracing level for console logging. Options are: <code>off</code> , <code>error</code> , <code>warning</code> , <code>info</code> , and <code>verbose</code> .
fileLoggingMode	debugOnly	The tracing level for file logging. Options are <code>never</code> , <code>always</code> , <code>debugOnly</code> .

watchDirectories

A set of [shared code directories](#) that should be monitored for changes. Ensures that when code in these directories is changed, the changes are picked up by your functions.

JSON

```
{  
  "watchDirectories": [ "Shared" ]  
}
```

Next steps

[Learn how to update the host.json file](#)

[Persist settings in environment variables](#)

Azure Functions monitoring data reference

Article • 08/12/2024

This article contains all the monitoring reference information for this service.

See [Monitor Azure Functions](#) for details on the data you can collect for Azure Functions and how to use it.

See [Monitor executions in Azure Functions](#) for details on using Application Insights to collect and analyze log data from individual functions in your function app.

Metrics

This section lists all the automatically collected platform metrics for this service. These metrics are also part of the global list of [all platform metrics supported in Azure Monitor](#).

For information on metric retention, see [Azure Monitor Metrics overview](#).

Hosting plans that allow your apps to scale dynamically support extra Functions-specific metrics:

Consumption plan

These metrics are used specifically when [estimating Consumption plan costs](#).

[Expand table](#)

Metric	Description
FunctionExecutionCount	Function execution count indicates the number of times your function app executed. This value correlates to the number of times a function runs in your app. This metric isn't currently supported for Premium and Dedicated (App Service) plans running on Linux.
FunctionExecutionUnits	Function execution units are a combination of execution time and your memory usage. Memory data isn't a metric currently available through Azure Monitor. However, if you want to optimize the memory usage of your app, can use the performance counter data collected by Application Insights. This metric isn't currently supported for Premium and Dedicated (App Service) plans running on Linux.

Supported metrics for Microsoft.Web/sites

The following table lists the metrics available for the Microsoft.Web/sites resource type. Most of these metrics apply to both function app and web apps, which both run on App Service.

Note

These metrics aren't available when your function app runs on Linux in a [Consumption plan](#).

- All columns might not be present in every table.
- Some columns might be beyond the viewing area of the page. Select **Expand table** to view all available columns.

Table headings

- **Category** - The metrics group or classification.
- **Metric** - The metric display name as it appears in the Azure portal.
- **Name in REST API** - The metric name as referred to in the [REST API](#).
- **Unit** - Unit of measure.

- **Aggregation** - The default [aggregation](#) type. Valid values: Average (Avg), Minimum (Min), Maximum (Max), Total (Sum), Count.
- **Dimensions** - [Dimensions](#) available for the metric.
- **Time Grains** - [Intervals](#) at which the metric is sampled. For example, `PT1M` indicates that the metric is sampled every minute, `PT30M` every 30 minutes, `PT1H` every hour, and so on.
- **DS Export**- Whether the metric is exportable to Azure Monitor Logs via diagnostic settings. For information on exporting metrics, see [Create diagnostic settings in Azure Monitor](#).

 Expand table

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Always Ready Function Execution Count	<code>AlwaysReadyFunctionExecutionCount</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Always Ready Function Execution Count. For Flex Consumption FunctionApps only.						
Always Ready Function Execution Units	<code>AlwaysReadyFunctionExecutionUnits</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Always Ready Function Execution Units. For Flex Consumption FunctionApps only.						
Always Ready Units	<code>AlwaysReadyUnits</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Always Ready Units. For Flex Consumption FunctionApps only.						
Connections	<code>AppConnections</code>	Count	Average, Count, Maximum, Minimum	<code>Instance</code>	PT1M	Yes
The number of bound sockets existing in the sandbox (w3wp.exe and its child processes). A bound socket is created by calling bind()/connect() APIs and remains until said socket is closed with CloseHandle()/closesocket(). For WebApps and FunctionApps.						
Average memory working set	<code>AverageMemoryWorkingSet</code>	Bytes	Average	<code>Instance</code>	PT1M	Yes
The average amount of memory used by the app, in megabytes (MiB). For WebApps and FunctionApps.						
Average Response Time (deprecated)	<code>AverageResponseTime</code>	Seconds	Average	<code>Instance</code>	PT1M	Yes
The average time taken for the app to serve requests,						

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
in seconds. For WebApps and FunctionApps.						
Data In	BytesReceived	Bytes	Total (Sum)	Instance	PT1M	Yes
The amount of incoming bandwidth consumed by the app, in MiB. For WebApps and FunctionApps.						
Data Out	BytesSent	Bytes	Total (Sum)	Instance	PT1M	Yes
The amount of outgoing bandwidth consumed by the app, in MiB. For WebApps and FunctionApps.						
CPU Time	CpuTime	Seconds	Count, Total (Sum), Minimum, Maximum	Instance	PT1M	Yes
The amount of CPU consumed by the app, in seconds. For more information about this metric. Please see https://aka.ms/website-monitor-cpu-time-vs-cpu-percentage (CPU time vs CPU percentage). For WebApps only.						
Current Assemblies	CurrentAssemblies	Count	Average	Instance	PT1M	Yes
The current number of Assemblies loaded across all AppDomains in this application. For WebApps and FunctionApps.						
File System Usage	FileSystemUsage	Bytes	Average	<none>	PT6H, PT12H, P1D	Yes
Percentage of filesystem quota consumed by the app. For WebApps and FunctionApps.						
Function Execution Count	FunctionExecutionCount	Count	Total (Sum)	Instance	PT1M	Yes
Function Execution Count. For FunctionApps only.						
Function Execution Units	FunctionExecutionUnits	Count	Total (Sum)	Instance	PT1M	Yes
Function Execution Units. For FunctionApps only.						
Gen 0 Garbage Collections	Gen0Collections	Count	Total (Sum)	Instance	PT1M	Yes
The number of times the generation 0 objects are garbage collected since the start of the app process.						

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Higher generation GCs include all lower generation GCs. For WebApps and FunctionApps.						
Gen 1 Garbage Collections The number of times the generation 1 objects are garbage collected since the start of the app process. Higher generation GCs include all lower generation GCs. For WebApps and FunctionApps.	<code>Gen1Collections</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Gen 2 Garbage Collections The number of times the generation 2 objects are garbage collected since the start of the app process. For WebApps and FunctionApps.	<code>Gen2Collections</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Handle Count The total number of handles currently open by the app process. For WebApps and FunctionApps.	<code>Handles</code>	Count	Average	<code>Instance</code>	PT1M	Yes
Health check status Health check status. For WebApps and FunctionApps.	<code>HealthCheckStatus</code>	Count	Average	<code>Instance</code>	PT5M, PT1H, P1D	Yes
Http 101 The count of requests resulting in an HTTP status code 101. For WebApps and FunctionApps.	<code>Http101</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Http 2xx The count of requests resulting in an HTTP status code ≥ 200 but < 300 . For WebApps and FunctionApps.	<code>Http2xx</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
Http 3xx The count of requests resulting in an HTTP status code ≥ 300 but < 400 . For WebApps and FunctionApps.	<code>Http3xx</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Http 401	<code>Http401</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The count of requests resulting in HTTP 401 status code. For WebApps and FunctionApps.						
Http 403	<code>Http403</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The count of requests resulting in HTTP 403 status code. For WebApps and FunctionApps.						
Http 404	<code>Http404</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The count of requests resulting in HTTP 404 status code. For WebApps and FunctionApps.						
Http 406	<code>Http406</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The count of requests resulting in HTTP 406 status code. For WebApps and FunctionApps.						
Http 4xx	<code>Http4xx</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The count of requests resulting in an HTTP status code ≥ 400 but < 500 . For WebApps and FunctionApps.						
Http Server Errors	<code>Http5xx</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The count of requests resulting in an HTTP status code ≥ 500 but < 600 . For WebApps and FunctionApps.						
Response Time	<code>HttpResponseTime</code>	Seconds	Average	<code>Instance</code>	PT1M	Yes
The time taken for the app to serve requests, in seconds. For WebApps and FunctionApps.						
Automatic Scaling Instance Count	<code>InstanceCount</code>	Count	Average	<code><none></code>	PT1M	Yes
The number of instances on which this app is running.						
IO Other Bytes Per Second	<code>IoOtherBytesPerSecond</code>	BytesPerSecond	Total (Sum)	<code>Instance</code>	PT1M	Yes
The rate at which the app process is issuing bytes to I/O operations that don't involve data, such as						

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
control operations. For WebApps and FunctionApps.						
IO Other Operations Per Second	<code>IoOtherOperationsPerSecond</code>	BytesPerSecond	Total (Sum)	Instance	PT1M	Yes
The rate at which the app process is issuing I/O operations that aren't read or write operations. For WebApps and FunctionApps.						
IO Read Bytes Per Second	<code>IoReadBytesPerSecond</code>	BytesPerSecond	Total (Sum)	Instance	PT1M	Yes
The rate at which the app process is reading bytes from I/O operations. For WebApps and FunctionApps.						
IO Read Operations Per Second	<code>IoReadOperationsPerSecond</code>	BytesPerSecond	Total (Sum)	Instance	PT1M	Yes
The rate at which the app process is issuing read I/O operations. For WebApps and FunctionApps.						
IO Write Bytes Per Second	<code>IoWriteBytesPerSecond</code>	BytesPerSecond	Total (Sum)	Instance	PT1M	Yes
The rate at which the app process is writing bytes to I/O operations. For WebApps and FunctionApps.						
IO Write Operations Per Second	<code>IoWriteOperationsPerSecond</code>	BytesPerSecond	Total (Sum)	Instance	PT1M	Yes
The rate at which the app process is issuing write I/O operations. For WebApps and FunctionApps.						
Memory working set	<code>MemoryWorkingSet</code>	Bytes	Average	Instance	PT1M	Yes
The current amount of memory used by the app, in MiB. For WebApps and FunctionApps.						
On Demand Function Execution Count	<code>OnDemandFunctionExecutionCount</code>	Count	Total (Sum)	Instance	PT1M	Yes
On Demand Function Execution Count. For Flex Consumption FunctionApps only.						
On Demand Function Execution Units	<code>OnDemandFunctionExecutionUnits</code>	Count	Total (Sum)	Instance	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
On Demand Function Execution Units. For Flex Consumption FunctionApps only.						
Private Bytes	<code>PrivateBytes</code>	Bytes	Average	<code>Instance</code>	PT1M	Yes
Private Bytes is the current size, in bytes, of memory that the app process has allocated that can't be shared with other processes. For WebApps and FunctionApps.						
Requests	<code>Requests</code>	Count	Total (Sum)	<code>Instance</code>	PT1M	Yes
The total number of requests regardless of their resulting HTTP status code. For WebApps and FunctionApps.						
Requests In Application Queue	<code>RequestsInApplicationQueue</code>	Count	Average	<code>Instance</code>	PT1M	Yes
The number of requests in the application request queue. For WebApps and FunctionApps.						
Thread Count	<code>Threads</code>	Count	Average	<code>Instance</code>	PT1M	Yes
The number of threads currently active in the app process. For WebApps and FunctionApps.						
Total App Domains	<code>TotalAppDomains</code>	Count	Average	<code>Instance</code>	PT1M	Yes
The current number of AppDomains loaded in this application. For WebApps and FunctionApps.						
Total App Domains Unloaded	<code>TotalAppDomainsUnloaded</code>	Count	Average	<code>Instance</code>	PT1M	Yes
The total number of AppDomains unloaded since the start of the application. For WebApps and FunctionApps.						
Workflow Action Completed Count	<code>WorkflowActionsCompleted</code>	Count	Total (Sum)	<code>workflowName, status</code>	PT1M	Yes
Workflow Action Completed Count. For LogicApps only.						

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Workflow Actions Failure Rate	WorkflowActionsFailureRate	Percent	Total (Sum)	workflowName	PT1M	Yes
Workflow Actions Failure Rate. For LogicApps only.						
Logic App Job Pull Rate Per Second	WorkflowAppJobPullRate	CountPerSecond	Total (Sum)	accountName	PT1M	Yes
Logic Job Pull Rate per second. For LogicApps only.						
Workflow Job Execution Delay	WorkflowJobExecutionDelay	Seconds	Average	workflowName	PT1M	Yes
Workflow Job Execution Delay. For LogicApps only.						
Workflow Job Execution Duration	WorkflowJobExecutionDuration	Seconds	Average	workflowName	PT1M	Yes
Workflow Job Execution Duration. For LogicApps only.						
Workflow Runs Completed Count	WorkflowRunsCompleted	Count	Total (Sum)	workflowName, status	PT1M	Yes
Workflow Runs Completed Count. For LogicApps only.						
Workflow Runs dispatched Count	WorkflowRunsDispatched	Count	Total (Sum)	workflowName	PT1M	Yes
Workflow Runs Dispatched Count. For LogicApps only.						
Workflow Runs Failure Rate	WorkflowRunsFailureRate	Percent	Total (Sum)	workflowName	PT1M	Yes
Workflow Runs Failure Rate. For LogicApps only.						
Workflow Runs Started Count	WorkflowRunsStarted	Count	Total (Sum)	workflowName	PT1M	Yes
Workflow Runs Started Count. For LogicApps only.						
Workflow Triggers Completed Count	WorkflowTriggersCompleted	Count	Total (Sum)	workflowName, status	PT1M	Yes
Workflow Triggers Completed Count. For LogicApps only.						
Workflow Triggers Failure Rate	WorkflowTriggersFailureRate	Percent	Total (Sum)	workflowName	PT1M	Yes
Workflow Triggers Failure Rate. For LogicApps only.						

Metric dimensions

For information about what metric dimensions are, see [Multi-dimensional metrics](#).

This service doesn't have any metrics that contain dimensions.

Resource logs

This section lists the types of resource logs you can collect for this service. The section pulls from the list of [all resource logs category types supported in Azure Monitor](#).

Supported resource logs for Microsoft.Web/sites

[Expand table](#)

Category	Category display name	Log table	Supports basic log plan	Supports ingestion-time transformation	Example queries	Costs to export
AppServiceAntivirusScanAuditLogs	Report Antivirus Audit Logs	AppServiceAntivirusScanAuditLogs	No	Yes		No
		Report on any discovered virus or infected files that have been uploaded to their site.				
AppServiceAppLogs	App Service Application Logs	AppServiceAppLogs	No	Yes	Queries	No
		Logs generated through your application.				
AppServiceAuditLogs	Access Audit Logs	AppServiceAuditLogs	No	Yes	Queries	No
		Logs generated when publishing users successfully log on via one of the App Service publishing protocols.				
AppServiceAuthenticationLogs	App Service Authentication logs (preview)	AppServiceAuthenticationLogs	No	No	Queries	Yes
		Logs generated through App Service Authentication for your application.				
AppServiceConsoleLogs	App Service Console Logs	AppServiceConsoleLogs	No	Yes	Queries	No
		Console logs generated from application or container.				
AppServiceFileAuditLogs	Site Content Change Audit Logs	AppServiceFileAuditLogs	No	Yes	Queries	No
		Logs generated when app service content is modified.				
AppServiceHTTPLogs	HTTP logs	AppServiceHTTPLogs	No	Yes	Queries	No
		Incoming HTTP requests on App Service. Use these logs to monitor application health, performance and usage patterns.				
AppServiceIPSecAuditLogs	IPSecurity Audit logs	AppServiceIPSecAuditLogs	No	Yes		No
		Logs generated through your application and pushed to Azure				

Category	Category display name	Log table	Supports basic log plan	Supports ingestion-time transformation	Example queries	Costs to export
Monitoring.						
AppServicePlatformLogs	App Service Platform logs	AppServicePlatformLogs	No	Yes		No
		Logs generated through AppService platform for your application.				
FunctionAppLogs	Function Application Logs	FunctionAppLogs	No	Yes	Queries	No
		Log generated by Function Apps. It includes logs emitted by the Functions host and logs emitted by customer code. Use these logs to monitor application health, performance, and behavior.				
WorkflowRuntime	Workflow Runtime Logs	LogicAppWorkflowRuntime	No	No	Queries	Yes
		Logs generated during Logic Apps workflow runtime.				

The log specific to Azure Functions is **FunctionAppLogs**.

For more information, see the [App Service monitoring data reference](#).

Azure Monitor Logs tables

This section lists the Azure Monitor Logs tables relevant to this service, which are available for query by Log Analytics using Kusto queries. The tables contain resource log data and possibly more depending on what is collected and routed to them.

App Services

Microsoft.Web/sites

- [FunctionAppLogs](#)

Activity log

The linked table lists the operations that can be recorded in the activity log for this service. These operations are a subset of [all the possible resource provider operations in the activity log](#).

For more information on the schema of activity log entries, see [Activity Log schema](#).

The following table lists operations related to Azure Functions that might be created in the activity log.

[!\[\]\(ec437be14e3e3be8b62440a9f6b75890_img.jpg\) Expand table](#)

Operation	Description
Microsoft.web/sites/functions/listkeys/action	Return the keys for the function .
Microsoft.Web/sites/host/listkeys/action	Return the host keys for the function app .
Microsoft.Web/sites/host-sync/action	Sync triggers operation.

Operation	Description
Microsoft.Web/sites/start/action	Function app started.
Microsoft.Web/sites/stop/action	Function app stopped.
Microsoft.Web/sites/write	Change a function app setting, such as runtime version or enable remote debugging.

You may also find logged operations that relate to the underlying App Service behaviors. For a more complete list, see [Microsoft.Web resource provider operations](#).

Related content

- See [Monitor Azure Functions](#) for a description of monitoring Azure Functions.
- See [Monitor Azure resources with Azure Monitor](#) for details on monitoring Azure resources.

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Frequently asked questions about networking in Azure Functions

FAQ

This article lists frequently asked questions about networking in Azure Functions. For a more comprehensive overview, see [Functions networking options](#).

How do I set a static IP in Functions?

Deploying a function in an App Service Environment is the primary way to have static inbound and outbound IP addresses for your functions. For details on using an App Service Environment, start with the article [Create and use an internal load balancer with an App Service Environment](#).

You can also use a virtual network NAT gateway to route outbound traffic through a public IP address that you control. To learn more, see [Tutorial: Control Azure Functions outbound IP with an Azure virtual network NAT gateway](#).

How do I restrict internet access to my function?

You can restrict internet access in a couple of ways:

- [Private endpoints](#): Restrict inbound traffic to your function app by private link over your virtual network, effectively blocking inbound traffic from the public internet.
- [IP restrictions](#): Restrict inbound traffic to your function app by IP range.
 - Under IP restrictions, you are also able to configure [Service Endpoints](#), which restrict your Function to only accept inbound traffic from a particular virtual network.
- Removal of all HTTP triggers. For some applications, it's enough to simply avoid HTTP triggers and use any other event source to trigger your function.

Keep in mind that the Azure portal editor requires direct access to your running function. Any code changes through the Azure portal will require the device you're using to browse the portal to have its IP added to the approved list. But you can still use anything under the platform features tab with network restrictions in place.

How do I restrict my function app to a virtual network?

You are able to restrict **inbound** traffic for a function app to a virtual network using [Service Endpoints](#). This configuration still allows the function app to make outbound calls to the internet.

To completely restrict a function such that all traffic flows through a virtual network, you can use a [private endpoints](#) with outbound virtual network integration or an App Service Environment. To learn more, see [Integrate Azure Functions with an Azure virtual network by using private endpoints](#).

How can I access resources in a virtual network from a function app?

You can access resources in a virtual network from a running function by using virtual network integration. For more information, see [Virtual network integration](#).

How do I access resources protected by service endpoints?

By using virtual network integration you can access service-endpoint-secured resources from a running function. For more information, see [virtual network integration](#).

How can I trigger a function from a resource in a virtual network?

You are able to allow HTTP triggers to be called from a virtual network using [Service Endpoints](#) or [Private Endpoint connections](#).

You can also trigger a function from all other resources in a virtual network by deploying your function app to a Premium plan, App Service plan, or App Service Environment. See [non-HTTP virtual network triggers](#) for more information

How can I deploy my function app in a virtual network?

Deploying to an App Service Environment is the only way to create a function app that's wholly inside a virtual network. For details on using an internal load balancer with an App Service Environment, start with the article [Create and use an internal load balancer with an App Service Environment](#).

For scenarios where you need only one-way access to virtual network resources, or less comprehensive network isolation, see the [Functions networking overview](#).

Next steps

To learn more about networking and functions:

- Follow the tutorial about getting started with virtual network integration
- Learn more about the networking options in Azure Functions
- Learn more about virtual network integration with App Service and Functions
- Learn more about virtual networks in Azure
- Enable more networking features and control with App Service Environments

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Web applications architecture design

Article • 09/23/2024

Today's web apps are expected to be available all day, every day from anywhere in the world, and usable from virtually any device or screen size. Web applications must be secure, flexible, and scalable to meet spikes in demand.

This article provides an overview of Azure web app technologies, guidance, solution ideas, and reference architectures.

Azure provides a wide range of tools and capabilities for creating, hosting, and monitoring web apps. These are just some of the key web app services available in Azure:

- [Azure App Service](#) enables you to easily create enterprise-ready web and mobile apps for any platform or device and deploy them on a scalable cloud infrastructure.
- [Azure Web Application Firewall](#) provides powerful protection for web apps.
- [Azure Monitor](#) provides full observability into your applications, infrastructure, and network. Monitor includes [Application Insights](#), which provides application performance management and monitoring for live web apps.
- [Azure SignalR Service](#) enables you to easily add real-time web functionalities.
- [Static Web Apps](#) provides streamlined full-stack development, from source code to global high availability.
- [Web App for Containers](#) enables you to run containerized web apps on Windows and Linux.
- [Azure Service Bus](#) enables you to integrate with other web apps using loosely coupled event-driven patterns.

Introduction to web apps on Azure

If you're new to creating and hosting web apps on Azure, the best way to learn more is with [Microsoft Learn training](#). This free online platform provides interactive training for Microsoft products and more.

These are a few good starting points to consider:

- [Create Azure App Service web apps](#)
- [Deploy and run a containerized web app with Azure App Service](#)
- [Azure Static Web Apps](#)

Path to production

Consider these patterns, guidelines, and architectures as you plan and implement your deployment:

- Basic web application
- Baseline zone-redundant web application
- Multi-region active-passive web application
- Common web application architectures
- Design principles for Azure applications
- Design and implementation patterns - Cloud Design Patterns
- Enterprise deployment using App Services Environment
- High availability enterprise deployment using App Services Environment

Best practices

For a good overview, see [Characteristics of modern web applications](#).

For information specific to Azure App Service, see:

- [Azure App Service and operational excellence](#)
- [App Service deployment best practices](#)
- [Security recommendations for App Service](#)
- [Azure security baseline for App Service](#)

Web app architectures

The following sections, organized by category, provide links to sample web app architectures.

E-commerce

- [Intelligent product search engine for e-commerce](#)
- [E-commerce website running in secured App Service Environment](#)
- [Scalable e-commerce web app](#)

Healthcare

- [Clinical insights with Microsoft Cloud for Healthcare](#)
- [Consumer health portal on Azure](#)
- [Virtual health on Microsoft Cloud for Healthcare](#)

Modernization

- Choose between traditional web apps and single-page apps
- ASP.NET architectural principles
- Common client-side web technologies
- Development process for Azure
- Azure hosting recommendations for ASP.NET Core web apps

Multi-tier apps

- Multi-tier web application built for HA/DR

Multi-region apps

- Highly available multi-region web application

Scalability

- Baseline web application with zone redundancy

Security

- Improved-security access to multitenant web apps from an on-premises network
- Protect APIs with Application Gateway and API Management

SharePoint

- Highly available SharePoint farm

Stay current with web development

Get the latest [updates on Azure web app products and features](#).

Additional resources

Example solutions

Here are some additional implementations to consider:

- Eventual consistency between multiple Power Apps instances
- App Service networking features
- Migrate a web app using Azure APIM
- Sharing location in real time using low-cost serverless Azure services
- Serverless web application

AWS or Google Cloud professionals

- AWS to Azure services comparison - Web applications
 - Google Cloud to Azure services comparison - Application services
-

Feedback

Was this page helpful?

 Yes

 No

Language runtime support policy

Article • 08/06/2024

This article explains Azure functions language runtime support policy.

Retirement process

Azure Functions runtime is built around various components, including operating systems, the Azure Functions host, and language-specific workers. To maintain full-support coverages for function apps, Functions support aligns with end-of-life support for a given language. To achieve this goal, Functions implements a phased reduction in support as programming language versions reach their end-of-life dates. For most language versions, the retirement date coincides with the community end-of-life date.

Notification phase

The Functions team sends notification emails to function app users about upcoming language version retirements. When you receive the notification, you should prepare to upgrade functions apps to use to a supported version.

Retirement phase

After the language end-of-life date, function apps that use retired language versions can still be created and deployed, and they continue to run on the platform. However your apps aren't eligible for new features, security patches, and performance optimizations until you upgrade them to a supported language version.

Important

You're highly encouraged to upgrade the language version of your affected function apps to a supported version. If you're running functions apps using an unsupported runtime or language version, you may encounter issues and performance implications and will be required to upgrade before receiving support for your function app.

Retirement policy exceptions

Any Azure Functions supported exceptions to language-specific retirement policies are documented here.

There are currently no exceptions to the general retirement policy.

Language version support timeline

To learn more about specific language version support policy timeline, visit the following external resources:

- .NET - dotnet.microsoft.com ↗
- Node - github.com ↗
- Java - [Microsoft technical documentation](#)
- PowerShell - [Microsoft technical documentation](#)
- Python - devguide.python.org ↗

Configuring language versions

[] Expand table

Language	Configuration guides
C# (isolated worker model)	link
C# (in-process model)	link
Java	link
Node	link
PowerShell	link
Python	link

Retired runtime versions

This historical table shows the highest language level for specific Azure Functions runtime versions that are no longer supported:

[] Expand table

Language	2.x	3.x
C#	GA (.NET Core 2.1)	GA (.NET Core 3.1 & .NET 5*)
JavaScript/TypeScript	GA (Node.js 10 & 8)	GA (Node.js 14, 12, & 10)
Java	GA (Java 8)	GA (Java 11 & 8)
PowerShell	N/A	N/A
Python	GA (Python 3.7)	GA (Python 3.9, 3.8, 3.7)
TypeScript	GA	GA

*.NET 5 was only supported for C# apps running in the [isolated worker model](#).

For the language levels currently supported by Azure Functions, see [Languages by runtime version](#).

Next steps

To learn more about how to upgrade your functions apps language versions, see the following resources:

- [Currently supported language versions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure Functions hosting options

Article • 07/16/2024

When you create a function app in Azure, you must choose a hosting option for your app. Azure provides you with these hosting options for your function code:

[+] Expand table

Hosting option	Service	Availability	Container support
Consumption plan	Azure Functions	Generally available (GA)	None
Flex Consumption plan	Azure Functions	Preview	None
Premium plan	Azure Functions	GA	Linux
Dedicated plan	Azure Functions	GA	Linux
Container Apps	Azure Container Apps	GA	Linux

Azure Functions hosting options are facilitated by Azure App Service infrastructure on both Linux and Windows virtual machines. The hosting option you choose dictates the following behaviors:

- How your function app is scaled.
- The resources available to each function app instance.
- Support for advanced functionality, such as Azure Virtual Network connectivity.
- Support for Linux containers.

The plan you choose also impacts the costs for running your function code. For more information, see [Billing](#).

This article provides a detailed comparison between the various hosting options. To learn more about running and managing your function code in Linux containers, see [Linux container support in Azure Functions](#).

Overview of plans

The following is a summary of the benefits of the various options for Azure Functions hosting:

[+] Expand table

Option	Benefits
Consumption plan	<p>Pay for compute resources only when your functions are running (pay-as-you-go) with automatic scale.</p> <p>On the Consumption plan, instances of the Functions host are dynamically added and removed based on the number of incoming events.</p> <ul style="list-style-type: none"> ✓ Default hosting plan that provides true <i>serverless</i> hosting. ✓ Pay only when your functions are running. ✓ Scales automatically, even during periods of high load.
Flex Consumption plan	<p>Get high scalability with compute choices, virtual networking, and pay-as-you-go billing.</p> <p>On the Flex Consumption plan, instances of the Functions host are dynamically added and removed based on the configured per instance concurrency and the number of incoming events.</p> <ul style="list-style-type: none"> ✓ Reduce cold starts by specifying a number of pre-provisioned (always ready) instances. ✓ Supports virtual networking for added security. ✓ Pay when your functions are running. ✓ Scales automatically, even during periods of high load.
Premium plan	<p>Automatically scales based on demand using prewarmed workers, which run applications with no delay after being idle, runs on more powerful instances, and connects to virtual networks.</p> <p>Consider the Azure Functions Premium plan in the following situations:</p> <ul style="list-style-type: none"> ✓ Your function apps run continuously, or nearly continuously. ✓ You want more control of your instances and want to deploy multiple function apps on the same plan with event-driven scaling. ✓ You have a high number of small executions and a high execution bill, but low GB seconds in the Consumption plan. ✓ You need more CPU or memory options than are provided by consumption plans. ✓ Your code needs to run longer than the maximum execution time allowed on the Consumption plan. ✓ You require virtual network connectivity. ✓ You want to provide a custom Linux image in which to run your functions.
Dedicated plan	<p>Run your functions within an App Service plan at regular App Service plan rates.</p> <p>Best for long-running scenarios where Durable Functions can't be used.</p> <p>Consider an App Service plan in the following situations:</p> <ul style="list-style-type: none"> ✓ You have existing and underutilized virtual machines that are already running

Option	Benefits
	<p>other App Service instances.</p> <ul style="list-style-type: none"> ✓ You must have fully predictable billing, or you need to manually scale instances. ✓ You want to run multiple web apps and function apps on the same plan ✓ You need access to larger compute size choices. ✓ Full compute isolation and secure network access provided by an App Service Environment (ASE). ✓ Very high memory usage and high scale (ASE).
Container Apps	<p>Create and deploy containerized function apps in a fully managed environment hosted by Azure Container Apps.</p> <p>Use the Azure Functions programming model to build event-driven, serverless, cloud native function apps. Run your functions alongside other microservices, APIs, websites, and workflows as container-hosted programs. Consider hosting your functions on Container Apps in the following situations:</p> <ul style="list-style-type: none"> ✓ You want to package custom libraries with your function code to support line-of-business apps. ✓ You need to migrate code execution from on-premises or legacy apps to cloud native microservices running in containers. ✓ When you want to avoid the overhead and complexity of managing Kubernetes clusters and dedicated compute. ✓ Your functions need high-end processing power provided by dedicated GPU compute resources.

The remaining tables in this article compare hosting options based on various features and behaviors.

Operating system support

This table shows operating system support for the hosting options.

[\[\]](#) Expand table

Hosting	Linux ¹ deployment	Windows ² deployment
Consumption plan	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only <input type="checkbox"/> Container (not supported) 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only
Flex Consumption plan	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only <input type="checkbox"/> Container (not supported) 	<ul style="list-style-type: none"> <input type="checkbox"/> Not supported
Premium plan	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only <input checked="" type="checkbox"/> Container 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Code-only

Hosting	Linux ¹ deployment	Windows ² deployment
Dedicated plan	<input checked="" type="checkbox"/> Code-only <input checked="" type="checkbox"/> Container	<input checked="" type="checkbox"/> Code-only
Container Apps	<input checked="" type="checkbox"/> Container-only	 Not supported

1. Linux is the only supported operating system for the [Python runtime stack](#).
2. Windows deployments are code-only. Functions doesn't currently support Windows containers.

Function app timeout duration

The timeout duration for functions in a function app is defined by the `functionTimeout` property in the [host.json](#) project file. This property applies specifically to function executions. After the trigger starts function execution, the function needs to return/respond within the timeout duration. For more information, see [Improve Azure Functions performance and reliability](#).

The following table shows the default and maximum values (in minutes) for specific plans:

[+] [Expand table](#)

Plan	Default	Maximum ¹
Consumption plan	5	10
Flex Consumption plan	30	Unlimited ³
Premium plan	30 ²	Unlimited ³
Dedicated plan	30 ²	Unlimited ³
Container Apps	30 ⁵	Unlimited ³

1. Regardless of the function app timeout setting, 230 seconds is the maximum amount of time that an HTTP triggered function can take to respond to a request. This is because of the [default idle timeout of Azure Load Balancer](#). For longer processing times, consider using the [Durable Functions async pattern](#) or [defer the actual work and return an immediate response](#).
2. The default timeout for version 1.x of the Functions runtime is *unlimited*.
3. Guaranteed for up to 60 minutes. [OS and runtime patching](#), vulnerability patching, and [scale in behaviors](#) can still cancel function executions so [ensure to write robust](#)

functions.

4. In a Flex Consumption plan, the host doesn't enforce an execution time limit. However, there are currently no guarantees because the platform might need to terminate your instances during scale-in, deployments, or to apply updates.
5. When the [minimum number of replicas](#) is set to zero, the default timeout depends on the specific triggers used in the app.

Language support

For details on current native language stack support in Functions, see [Supported languages in Azure Functions](#).

Scale

The following table compares the scaling behaviors of the various hosting plans. Maximum instances are given on a per-function app (Consumption) or per-plan (Premium/Dedicated) basis, unless otherwise indicated.

[+] [Expand table](#)

Plan	Scale out	Max # instances
Consumption plan	Event driven . Scales out automatically, even during periods of high load. Functions infrastructure scales CPU and memory resources by adding more instances of the Functions host, based on the number of incoming trigger events.	Windows: 200 Linux: 100 ¹
Flex Consumption plan	Per-function scaling . Event-driven scaling decisions are calculated on a per-function basis, which provides a more deterministic way of scaling the functions in your app. With the exception of HTTP, Blob storage (Event Grid), and Durable Functions, all other function trigger types in your app scale on independent instances. All HTTP triggers in your app scale together as a group on the same instances, as do all Blob storage (Event Grid) triggers. All Durable Functions triggers also share instances and scale together.	Limited only by total memory usage of all instances across a given region. For more information, see Instance memory .
Premium plan	Event driven . Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding more instances of the Functions host, based on the number of events that its functions are triggered on.	Windows: 100 Linux: 20-100 ²

Plan	Scale out	Max # instances
Dedicated plan ³	Manual/autoscale	10-30 100 (ASE)
Container Apps	Event driven. Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding more instances of the Functions host, based on the number of events that its functions are triggered on.	300-1000 ⁴

1. During scale-out, there's currently a limit of 500 instances per subscription per hour for Linux apps on a Consumption plan.
2. In some regions, Linux apps on a Premium plan can scale to 100 instances. For more information, see the [Premium plan article](#).
3. For specific limits for the various App Service plan options, see the [App Service plan limits](#).
4. On Container Apps, the default is 10 instances, but you can set the [maximum number of replicas](#), which has an overall maximum of 1000. This setting is honored as long as there's enough cores quota available. When you create your function app from the Azure portal you're limited to 300 instances.

Cold start behavior

[+] [Expand table](#)

Plan	Details
Consumption plan	Apps can scale to zero when idle, meaning some requests might have more latency at startup. The consumption plan does have some optimizations to help decrease cold start time, including pulling from prewarmed placeholder functions that already have the host and language processes running.
Flex Consumption plan	Supports always ready instances to reduce the delay when provisioning new instances.
Premium plan	Supports always ready instances to avoid cold starts by letting you maintain one or more <i>perpetually warm</i> instances.
Dedicated plan	When running in a Dedicated plan, the Functions host can run continuously on a prescribed number of instances, which means that cold start isn't really an issue.
Container Apps	Depends on the minimum number of replicas : <ul style="list-style-type: none"> • When set to zero: apps can scale to zero when idle and some requests might

Plan	Details
	<p>have more latency at startup.</p> <ul style="list-style-type: none"> When set to one or more: the host process runs continuously, which means that cold start isn't an issue.

Service limits

[\[\]](#) Expand table

Resource	Consumption plan	Flex Consumption plan ¹³	Premium plan	Dedicated plan/ASE	Container Apps
Default timeout duration (min)	5	30	30	30 ¹	30 ¹⁷
Max timeout duration (min)	10	unbounded ¹⁶	unbounded ⁸	unbounded ²	unbounded ¹⁸
Max outbound connections (per instance)	600 active (1200 total)	unbounded	unbounded	unbounded	unbounded
Max request size (MB) ³	100	100	100	100	100
Max query string length ³	4096	4096	4096	4096	4096
Max request URL length ³	8192	8192	8192	8192	8192
ACU per instance	100	varies	210-840	100-840/210-250 ⁹	varies
Max memory (GB per instance)	1.5	4 ¹⁴	3.5-14	1.75-14/3.5-14	varies
Max instance count (Windows/Linux)	200/100	1000 ¹⁵	100/20	varies by SKU/100 ¹⁰	10-300 ¹⁹
Function apps per plan ¹²	100	100	100	unbounded ⁴	unbounded ⁴

Resource	Consumption plan	Flex Consumption plan ¹³	Premium plan	Dedicated plan/ASE	Container Apps
App Service plans	100 per region ¹²	n/a	100 per resource group	100 per resource group	n/a
Deployment slots per app ¹¹	2	n/a	3	1-20 ¹⁰	not supported
Storage (temporary) ⁵	0.5 GB	0.8 GB	21-140 GB	11-140 GB	n/a
Storage (persisted)	1 GB ⁶	0 GB ⁶	250 GB	10-1000 GB ¹⁰	n/a
Custom domains per app	500 ⁷	500	500	500	not supported
Custom domain SSL support	unbounded SNI SSL connection included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included	not supported

Notes on service limits:

1. By default, the timeout for the Functions 1.x runtime in an App Service plan is unbounded.
2. Requires the App Service plan be set to [Always On](#). Pay at standard [rates](#).
3. These limits are [set in the host](#).
4. The actual number of function apps that you can host depends on the activity of the apps, the size of the machine instances, and the corresponding resource utilization.
5. The storage limit is the total content size in temporary storage across all apps in the same App Service plan. For Consumption plans on Linux, the storage is currently 1.5 GB.
6. Consumption plan uses an Azure Files share for persisted storage. When you provide your own Azure Files share, the specific share size limits depend on the storage account you set for [WEBSITE_CONTENTAZUREFILECONNECTIONSTRING](#). On Linux, you must [explicitly mount your own Azure Files share](#) for both Flex Consumption and Consumption plans.
7. When your function app is hosted in a [Consumption plan](#), only the CNAME option is supported. For function apps in a [Premium plan](#) or an [App Service plan](#), you can

- map a custom domain using either a CNAME or an A record.
8. Guaranteed for up to 60 minutes.
 9. Workers are roles that host customer apps. Workers are available in three fixed sizes: One vCPU/3.5 GB RAM; Two vCPU/7 GB RAM; Four vCPU/14 GB RAM.
 10. See [App Service limits](#) for details.
 11. Including the production slot.
 12. There's currently a limit of 5000 function apps in a given subscription.
 13. The Flex Consumption plan is currently in preview.
 14. Flex Consumption plan instance sizes are currently defined as either 2,048 MB or 4,096 MB. For more information, see [Instance memory](#).
 15. Flex Consumption plan during preview has a regional subscription quota that limits the total memory usage of all instances across a given region. For more information, see [Instance memory](#).
 16. In a Flex Consumption plan, the host doesn't enforce an execution time limit. However, there are currently no guarantees because the platform might need to terminate your instances during scale-in, deployments, or to apply updates.
 17. When the [minimum number of replicas](#) is set to zero, the default timeout depends on the specific triggers used in the app.
 18. When the [minimum number of replicas](#) is set to one or more.
 19. On Container Apps, you can set the [maximum number of replicas](#), which is honored as long as there's enough cores quota available.

Networking features

[] Expand table

Feature	Consumption plan	Flex Consumption plan	Premium plan	Dedicated plan/ASE	Container Apps*
Inbound IP restrictions	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
Inbound Private Endpoints	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Virtual network integration	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes (Regional)	<input checked="" type="checkbox"/> Yes (Regional)	<input checked="" type="checkbox"/> Yes (Regional and Gateway)	<input checked="" type="checkbox"/> Yes
Virtual network triggers (non-HTTP)	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes

Feature	Consumption plan	Flex Consumption plan	Premium plan	Dedicated plan/ASE	Container Apps*
Hybrid connections (Windows only)	✗ No	✗ No	✓ Yes	✓ Yes	✗ No
Outbound IP restrictions	✗ No	✓ Yes	✓ Yes	✓ Yes	✓ Yes

*For more information, see [Networking in Azure Container Apps environment](#).

Billing

[+] [Expand table](#)

Plan	Details
Consumption plan	Pay only for the time your functions run. Billing is based on number of executions, execution time, and memory used.
Flex Consumption plan	Billing is based on number of executions, the memory of instances when they're actively executing functions, plus the cost of any always ready instances . For more information, see Flex Consumption plan billing .
Premium plan	Premium plan is based on the number of core seconds and memory used across needed and prewarmed instances. At least one instance per plan must always be kept warm. This plan provides the most predictable pricing.
Dedicated plan	You pay the same for function apps in an App Service Plan as you would for other App Service resources, like web apps. For an ASE, there's a flat monthly rate that pays for the infrastructure and doesn't change with the size of the environment. There's also a cost per App Service plan vCPU. All apps hosted in an ASE are in the Isolated pricing SKU. For more information, see the ASE overview article .
Container Apps	Billing in Azure Container Apps is based on your plan type. For more information, see Billing in Azure Container Apps .

For a direct cost comparison between dynamic hosting plans (Consumption, Flex Consumption, and Premium), see the [Azure Functions pricing page](#). For pricing of the various Dedicated plan options, see the [App Service pricing page](#). For pricing Container Apps hosting, see [Azure Container Apps pricing](#).

Limitations for creating new function apps in an existing resource group

In some cases, when trying to create a new hosting plan for your function app in an existing resource group you might receive one of the following errors:

- The pricing tier is not allowed in this resource group
- <SKU_name> workers are not available in resource group
<resource_group_name>

This can happen when the following conditions are met:

- You create a function app in an existing resource group that has ever contained another function app or web app. For example, Linux Consumption apps aren't supported in the same resource group as Linux Dedicated or Linux Premium plans.
- Your new function app is created in the same region as the previous app.
- The previous app is in some way incompatible with your new app. This error can happen between SKUs, operating systems, or due to other platform-level features, such as availability zone support.

The reason this happens is due to how function app and web app plans are mapped to different pools of resources when being created. Different SKUs require a different set of infrastructure capabilities. When you create an app in a resource group, that resource group is mapped and assigned to a specific pool of resources. If you try to create another plan in that resource group and the mapped pool does not have the required resources, this error occurs.

When this error occurs, instead create your function app and hosting plan in a new resource group.

Next steps

- [Deployment technologies in Azure Functions](#)
- [Azure Functions developer guide](#)

Feedback

Was this page helpful?

