

Contents

[Functions Documentation](#)

[Overview](#)

[About Azure Functions](#)

[Getting started](#)

[Durable Functions](#)

[Serverless comparison](#)

[Hosting plan options](#)

[Quickstarts](#)

[Create your first function](#)

[Visual Studio Code](#)

[C#](#)

[Java](#)

[JavaScript](#)

[PowerShell](#)

[Python](#)

[TypeScript](#)

[Visual Studio](#)

[Command line](#)

[C#](#)

[Java](#)

[JavaScript](#)

[PowerShell](#)

[Python](#)

[TypeScript](#)

[ARM template](#)

[Connect to storage](#)

[Visual Studio Code](#)

[Visual Studio](#)

[Command line](#)

Tutorials

- [Functions with Logic Apps](#)
- [Develop Python functions with VS Code](#)
- [OpenAPI definition for serverless APIs](#)
- [Connect to a Virtual Network](#)
- [Establish private site access](#)
- [Image resize with Event Grid](#)
- [Create a serverless web app](#)
- [Machine learning with TensorFlow](#)
- [Image classification with PyTorch](#)
- [Create a custom Linux image](#)
- [Functions on IoT Edge device](#)
- [Java with Azure Cosmos DB and Event Hubs](#)

Samples

[Azure Serverless Community Library](#)

[Azure Samples](#)

[C#](#)

[Java](#)

[JavaScript](#)

[PowerShell](#)

[Python](#)

[TypeScript](#)

[Azure CLI](#)

[CLI sample index](#)

[Create function app](#)

[Serverless function app](#)

[Serverless Python app](#)

[Scalable Premium plan app](#)

[Dedicated \(App Service\) plan app](#)

[Integrate services](#)

[Connect Azure Storage](#)

[Connect Azure Cosmos DB](#)

- Python mount Files share
- Continuous deployment
- GitHub deployment
- DevOps deployment

Concepts

- Compare runtime versions
- Premium plan
- Deployments
- Events and messaging
 - Reliable event processing
 - Designing for identical input
- Triggers and bindings
 - About triggers and bindings
 - Binding example
 - Register binding extensions
 - Binding expression patterns
 - Use binding return values
 - Handle binding errors

Languages

- Supported languages
- C# (class library)
- C# script (.csx)
- F#
- JavaScript
- Java
- PowerShell
- Python
- TypeScript

Frameworks

- Express.js

Security

- Security overview

- [Security baseline](#)
- [Monitoring](#)
- [Diagnostics](#)
- [Consumption plan costs](#)
- [Performance considerations](#)
- [Storage considerations](#)
- [Functions Proxies](#)
- [Networking options](#)
- [IP addresses](#)
- [Custom handlers](#)
- [How-to guides](#)
 - [Develop](#)
 - [Developer guide](#)
 - [Local development](#)
 - [Develop and debug locally](#)
 - [Visual Studio Code development](#)
 - [Visual Studio development](#)
 - [Core Tools development](#)
 - [Create functions](#)
 - [HTTP trigger](#)
 - [Azure portal](#)
 - [Command line](#)
 - [Visual Studio](#)
 - [Visual Studio Code](#)
 - [Java using Gradle](#)
 - [Java using Eclipse](#)
 - [Java using IntelliJ IDEA](#)
 - [Kotlin using Maven](#)
 - [Kotlin using IntelliJ](#)
 - [Linux App Service plan](#)
 - [Linux Consumption plan](#)
 - [Azure for Students Starter](#)

- [Azure Cosmos DB trigger](#)
- [Blob storage trigger](#)
- [Queue storage trigger](#)
- [Timer trigger](#)
- [Add bindings](#)
 - [How to add a binding](#)
 - [Azure Cosmos DB - portal](#)
 - [Storage - Python](#)
 - [Storage - portal](#)
 - [Storage - Visual Studio Code](#)
 - [Storage - Visual Studio](#)
 - [Storage - Java](#)
- [Debug and test](#)
 - [Test functions](#)
 - [Debug local PowerShell functions](#)
 - [Debug Event Grid trigger locally](#)
- [Dependency injection](#)
- [Manage connections](#)
- [Error handling and retries](#)
- [Manually run a non HTTP-triggered function](#)
- [Deploy](#)
 - [Continuous deployment](#)
 - [Deployment slots](#)
 - [Build and deploy using Azure Pipelines](#)
 - [Build and deploy using GitHub Actions](#)
 - [Zip deployment](#)
 - [Run from package](#)
 - [Functions in Kubernetes](#)
 - [Automate resource deployment](#)
 - [On-premises functions](#)
 - [Deploy using the Jenkins plugin](#)
- [Configure](#)

[Manage a function app](#)

[Set the runtime version](#)

[Disable a function](#)

[Geo-disaster recovery](#)

[Monitor](#)

[Configure monitoring](#)

[Analyze telemetry data](#)

[Streaming logs](#)

[Diagnostic logs](#)

[Secure](#)

[Add SSL cert](#)

[Authenticate users](#)

[Authenticate with Azure AD](#)

[Authenticate with Facebook](#)

[Authenticate with Google](#)

[Authenticate with Microsoft account](#)

[Authenticate with Twitter](#)

[Authenticate with an OpenID Connect provider \(Preview\)](#)

[Authenticate using Sign in with Apple \(Preview\)](#)

[Advanced auth](#)

[Restrict IPs](#)

[Use a managed identity](#)

[Reference secrets from Key Vault](#)

[Encrypt site data](#)

[Integrate](#)

[Add bindings](#)

[Azure Cosmos DB - portal](#)

[Storage - Python](#)

[Storage - portal](#)

[Storage - Visual Studio Code](#)

[Storage - Visual Studio](#)

[Connect to SQL Database](#)

- [Connect to a virtual Network](#)
- [Create an Open API 2.0 definition](#)
- [Export to PowerApps and Microsoft Flow](#)
- [Use a managed identity](#)
- [Customize HTTP function endpoint](#)
- [Manage on-premises resources](#)
- [Troubleshoot](#)
 - [Troubleshoot storage](#)
 - [Troubleshoot Python functions](#)
- [Reference](#)
 - [API references](#)
 - [ARM template](#)
 - [Azure CLI](#)
 - [Azure PowerShell](#)
 - [Java](#)
 - [Python](#)
- [App settings reference](#)
- [Triggers and bindings](#)
 - [Blob storage](#)
 - [Overview](#)
 - [Trigger](#)
 - [Input](#)
 - [Output](#)
 - [Azure Cosmos DB](#)
 - [Functions 1.x](#)
 - [Functions 2.x and higher](#)
 - [Overview](#)
 - [Trigger](#)
 - [Input](#)
 - [Output](#)
 - [Dapr](#)
 - [Event Grid](#)

[Overview](#)

[Trigger](#)

[Output](#)

[Event Hubs](#)

[Overview](#)

[Trigger](#)

[Output](#)

[IoT Hub](#)

[Overview](#)

[Trigger](#)

[Output](#)

[Kafka](#)

[HTTP and webhooks](#)

[Overview](#)

[Trigger](#)

[Output](#)

[Mobile Apps](#)

[Notification Hubs](#)

[Queue storage](#)

[Overview](#)

[Trigger](#)

[Output](#)

[RabbitMQ](#)

[SendGrid](#)

[Service Bus](#)

[Overview](#)

[Trigger](#)

[Output](#)

[SignalR Service](#)

[Overview](#)

[Trigger](#)

[Input](#)

[Output](#)

[Table storage](#)

[Overview](#)

[Input](#)

[Output](#)

[Timer](#)

[Twilio](#)

[Warmup](#)

[host.json 2.x reference](#)

[host.json 1.x reference](#)

[Networking FAQ](#)

[OpenAPI reference](#)

[Resources](#)

[Build your skills with Microsoft Learn](#)

[Architecture guidance](#)

[Azure Roadmap](#)

[Pricing](#)

[Pricing calculator](#)

[Quota information](#)

[Regional availability](#)

[Videos](#)

[Microsoft Q&A question page](#)

[Stack Overflow](#)

[Twitter](#)

[Provide product feedback](#)

[Azure Functions GitHub repository](#)

[Azure updates](#)

Introduction to Azure Functions

12/4/2020 • 2 minutes to read • [Edit Online](#)

We often build systems to react to a series of critical events. Whether you're building a web API, responding to database changes, processing IoT data streams, or even managing message queues - every application needs a way to run some code as these events occur.

To meet this need, Azure Functions provides "compute on-demand" in two significant ways.

First, Azure Functions allows you to implement your system's logic into readily available blocks of code. These code blocks are called "functions". Different functions can run anytime you need to respond to critical events.

Second, as requests increase, Azure Functions meets the demand with as many resources and function instances as necessary - but only while needed. As requests fall, any extra resources and application instances drop off automatically.

Where do all the compute resources come from? Azure Functions [provides as many or as few compute resources as needed](#) to meet your application's demand.

Providing compute resources on-demand is the essence of [serverless computing](#) in Azure Functions.

Scenarios

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.

| IF YOU WANT TO... | THEN... |
|---------------------------------------|--|
| Build a web API | Implement an endpoint for your web applications using the HTTP trigger |
| Process file uploads | Run code when a file is uploaded or changed in blob storage |
| Build a serverless workflow | Chain a series of functions together using durable functions |
| Respond to database changes | Run custom logic when a document is created or updated in Cosmos DB |
| Run scheduled tasks | Execute code at set times |
| Create reliable message queue systems | Process message queues using Queue Storage , Service Bus , or Event Hubs |
| Analyze IoT data streams | Collect and process data from IoT devices |
| Process data in real time | Use Functions and SignalR to respond to data in the moment |

As you build your functions, you have the following options and resources available:

- **Use your preferred language:** Write functions in [C#](#), [Java](#), [JavaScript](#), [PowerShell](#), or [Python](#), or use a [custom handler](#) to use virtually any other language.

- **Automate deployment:** From a tools-based approach to using external pipelines, there's a [myriad of deployment options](#) available.
- **Troubleshoot a function:** Use [monitoring tools](#) and [testing strategies](#) to gain insights into your apps.
- **Flexible pricing options:** With the [Consumption](#) plan, you only pay while your functions are running, while the [Premium](#) and [App Service](#) plans offer features for specialized needs.

Next Steps

Get started through lessons, samples, and interactive tutorials

Getting started with Azure Functions

12/4/2020 • 2 minutes to read • [Edit Online](#)

Introduction

Azure Functions allows you to implement your system's logic into readily-available blocks of code. These code blocks are called "functions".

Use the following resources to get started.

| ACTION | RESOURCES |
|---------------------------------|---|
| Create your first function | <p>Using one of the following tools:</p> <ul style="list-style-type: none">• Visual Studio• Visual Studio Code• Command line |
| See a function running | <ul style="list-style-type: none">• Azure Samples Browser• Azure Community Library |
| Explore an interactive tutorial | <ul style="list-style-type: none">• Choose the best Azure serverless technology for your business scenario• Well-Architected Framework - Performance efficiency• Execute an Azure Function with triggers <p>See Microsoft Learn for a full listing of interactive tutorials.</p> |
| Learn more in-depth | <ul style="list-style-type: none">• Learn how functions automatically increase or decrease instances to match demand• Explore the different deployment methods available• Use built-in monitoring tools to help analyze your functions• Read the C# language reference |
| ACTION | RESOURCES |
| Create your first function | <p>Using one of the following tools:</p> <ul style="list-style-type: none">• Visual Studio Code• Java/Maven function with terminal/command prompt• Gradle• Eclipse• IntelliJ IDEA |
| See a function running | <ul style="list-style-type: none">• Azure Samples Browser• Azure Community Library |
| Explore an interactive tutorial | <ul style="list-style-type: none">• Choose the best Azure serverless technology for your business scenario• Well-Architected Framework - Performance efficiency• Develop an App using the Maven Plugin for Azure Functions <p>See Microsoft Learn for a full listing of interactive tutorials.</p> |

| ACTION | RESOURCES |
|---------------------------------|--|
| Learn more in-depth | <ul style="list-style-type: none"> Learn how functions automatically increase or decrease instances to match demand Explore the different deployment methods available Use built-in monitoring tools to help analyze your functions Read the Java language reference |
| ACTION | RESOURCES |
| Create your first function | <p>Using one of the following tools:</p> <ul style="list-style-type: none"> Visual Studio Code Node.js terminal/command prompt |
| See a function running | <ul style="list-style-type: none"> Azure Samples Browser Azure Community Library |
| Explore an interactive tutorial | <ul style="list-style-type: none"> Choose the best Azure serverless technology for your business scenario Well-Architected Framework - Performance efficiency Build Serverless APIs with Azure Functions Create serverless logic with Azure Functions Refactor Node.js and Express APIs to Serverless APIs with Azure Functions <p>See Microsoft Learn for a full listing of interactive tutorials.</p> |
| Learn more in-depth | <ul style="list-style-type: none"> Learn how functions automatically increase or decrease instances to match demand Explore the different deployment methods available Use built-in monitoring tools to help analyze your functions Read the JavaScript or TypeScript language reference |
| ACTION | RESOURCES |
| Create your first function | <ul style="list-style-type: none"> Using Visual Studio Code |
| See a function running | <ul style="list-style-type: none"> Azure Samples Browser Azure Community Library |
| Explore an interactive tutorial | <ul style="list-style-type: none"> Choose the best Azure serverless technology for your business scenario Well-Architected Framework - Performance efficiency Build Serverless APIs with Azure Functions Create serverless logic with Azure Functions Execute an Azure Function with triggers <p>See Microsoft Learn for a full listing of interactive tutorials.</p> |
| Learn more in-depth | <ul style="list-style-type: none"> Learn how functions automatically increase or decrease instances to match demand Explore the different deployment methods available Use built-in monitoring tools to help analyze your functions Read the PowerShell language reference |

| ACTION | RESOURCES |
|---------------------------------|--|
| Create your first function | <p>Using one of the following tools:</p> <ul style="list-style-type: none">• Visual Studio Code• Terminal/command prompt |
| See a function running | <ul style="list-style-type: none">• Azure Samples Browser• Azure Community Library |
| Explore an interactive tutorial | <ul style="list-style-type: none">• Choose the best Azure serverless technology for your business scenario• Well-Architected Framework - Performance efficiency• Build Serverless APIs with Azure Functions• Create serverless logic with Azure Functions <p>See Microsoft Learn for a full listing of interactive tutorials.</p> |
| Learn more in-depth | <ul style="list-style-type: none">• Learn how functions automatically increase or decrease instances to match demand• Explore the different deployment methods available• Use built-in monitoring tools to help analyze your functions• Read the Python language reference |

Next steps

[Learn about the anatomy of an Azure Functions application](#)

What are Durable Functions?

11/2/2020 • 18 minutes to read • [Edit Online](#)

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 currently supports the following languages:

- **C#**: both [precompiled class libraries](#) and [C# script](#).
- **JavaScript**: supported only for version 2.x of the Azure Functions runtime. Requires version 1.7.0 of the Durable Functions extension, or a later version.
- **Python**: requires version 1.8.5 of the Durable Functions extension, or a later version. Support for Durable Functions is currently in public preview.
- **F#**: precompiled class libraries and F# script. F# script is only supported for version 1.x of the Azure Functions runtime.
- **PowerShell**: support for Durable Functions is currently in public preview. Supported only for version 3.x of the Azure Functions runtime and PowerShell 7. Requires version 2.2.2 of the Durable Functions extension, or a later version. Only the following patterns are currently supported: [Function chaining](#), [Fan-out/fan-in](#), [Async HTTP APIs](#).

Durable Functions has a goal of supporting all [Azure Functions languages](#). See the [Durable Functions issues list](#) for the latest status of work to support additional languages.

Like Azure Functions, there are templates to help you develop Durable Functions using [Visual Studio 2019](#), [Visual Studio Code](#), and the [Azure portal](#).

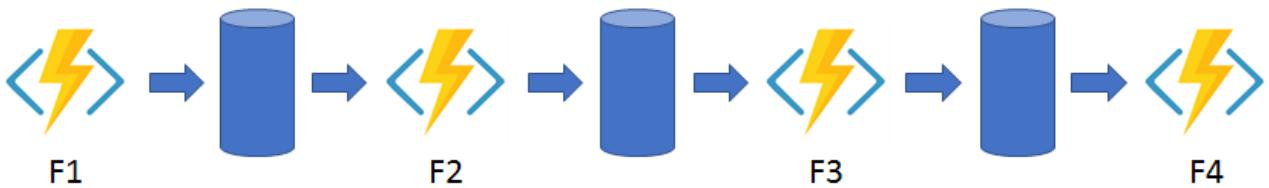
Application patterns

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.



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.

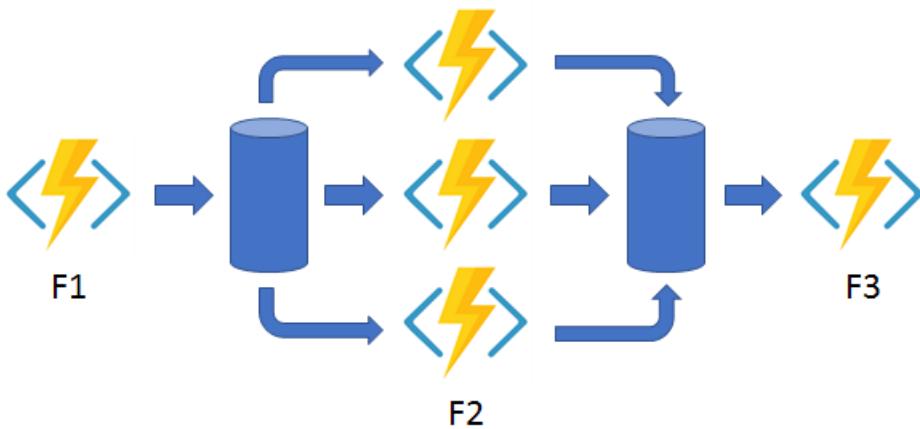
- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

```
[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#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

```
[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.

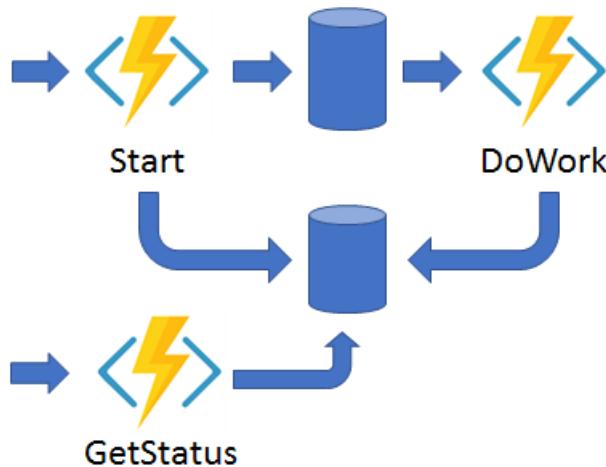
The automatic checkpointing that happens at the `await` call on `Task.WhenAll` ensures that a potential midway crash or reboot doesn't require restarting an already completed task.

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#](#) and [JavaScript](#)) 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/b79baf67f717453ca9e86c5da21e03ec

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

> curl
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec -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 [orchestration 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

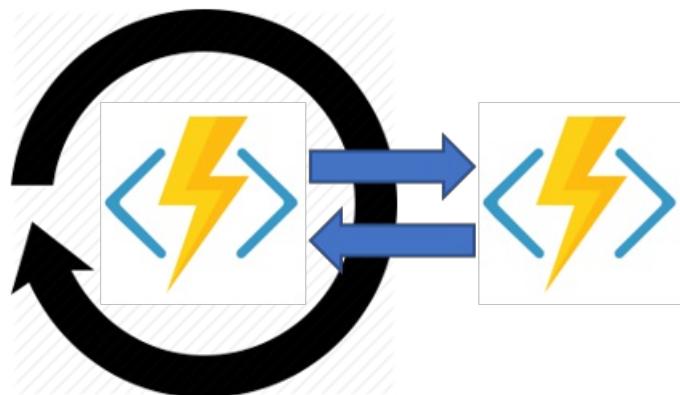
instead of the built-in HTTP APIs that use a generated key for authentication.

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#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

```

[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", machineId);
            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 a condition is met and the loop is exited. A durable timer controls the polling interval. Then, more work can be performed, or the orchestration can end. When `nextCheck` exceeds `expiryTime`, the monitor ends.

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#](#)
- [JavaScript](#)
- [Python](#)

- [PowerShell](#)

```
[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).

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}/raiseEvent/ApprovalEvent -H
"Content-Type: application/json"
```

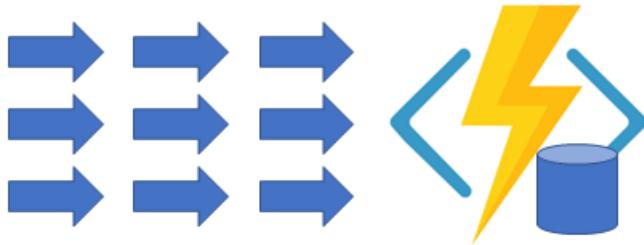
An event can also be raised using the durable orchestration client from another function in the same function app:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

```
[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#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

```
[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.

```
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#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

```
[FunctionName("EventHubTriggerCSharp")]
public static async Task Run(
    [EventHubTrigger("device-sensor-events")] EventData eventData,
    [DurableClient] IDurableOrchestrationClient 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# and JavaScript.

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](#)
- [Python using Visual Studio Code](#)

- [PowerShell using Visual Studio Code](#)

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.

Learn more

The following video highlights the benefits of Durable Functions:

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):

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

11/2/2020 • 6 minutes to read • [Edit Online](#)

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.

If you're looking for a more general comparison between Azure Functions and other Azure compute options, see [Criteria for choosing an Azure compute service](#) and [Choosing an Azure compute option for microservices](#).

Compare Microsoft Power Automate and Azure Logic Apps

Power Automate and 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 top of Logic Apps. They share the same workflow designer and the same [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. Accordingly, you can start with a flow at first, and then convert it to a logic app as needed.

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 and Visual Studio , Code view available |
| Application lifecycle management (ALM) | Design and test in non-production environments, promote to production when ready | 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 |

| | POWER AUTOMATE | LOGIC APPS |
|----------|---|--|
| Security | Microsoft 365 security audit logs, DLP, encryption at rest for sensitive data | Security assurance of Azure: Azure security , Azure Security Center , 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 provides serverless workflows. Both can create complex *orchestrations*. An orchestration is a collection of functions or steps, called *actions* in Logic Apps, that are executed to accomplish 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 Logic Apps, you create orchestrations by using a GUI or editing configuration files.

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

| | DURABLE FUNCTIONS | 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 |
| Management | REST API, Visual Studio | Azure portal, REST API, PowerShell, Visual Studio |
| Execution context | Can run locally or in the cloud | Runs only in the cloud |

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 C#, Java, JavaScript, Bash, .cmd, .bat, PowerShell, PHP, TypeScript, Python, and more. This is not a comprehensive list. 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 as well as they do with external services.

A flow can call a logic app. A logic app can call a function, and a function can call a logic app. See, for example, [Create a function that integrates with Azure Logic Apps](#).

The integration between Power Automate, Logic Apps, and Functions continues to improve over time. You can build something in one service and use it 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, or function app. Select any of the following links:

- [Get started with Power Automate](#)
- [Create a logic app](#)
- [Create your first Azure function](#)

Azure Functions scale and hosting

12/4/2020 • 19 minutes to read • [Edit Online](#)

When you create a function app in Azure, you must choose a hosting plan for your app. There are three basic hosting plans available for Azure Functions: [Consumption plan](#), [Premium plan](#), and [Dedicated \(App Service\) plan](#). All hosting plans are generally available (GA) on both Linux and Windows virtual machines.

The hosting plan you choose dictates the following behaviors:

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

Both Consumption and Premium plans automatically add compute power when your code is running. Your app is scaled out when needed to handle load, and scaled in when code stops running. For the Consumption plan, you also don't have to pay for idle VMs or reserve capacity in advance.

Premium plan provides additional features, such as premium compute instances, the ability to keep instances warm indefinitely, and VNet connectivity.

App Service plan allows you to take advantage of dedicated infrastructure, which you manage. Your function app doesn't scale based on events, which means it never scales in to zero. (Requires that [Always on](#) is enabled.)

For a detailed comparison between the various hosting plans (including Kubernetes-based hosting), see the [Hosting plans comparison section](#).

Consumption plan

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. This serverless plan scales automatically, and 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.

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](#).

The Consumption plan is the default hosting plan and offers the following benefits:

- Pay only when your functions are running
- Scale out automatically, even during periods of high load

Function apps in the same region can be assigned to the same Consumption plan. There's no downside or impact to having multiple apps running in the same Consumption plan. Assigning multiple apps to the same Consumption plan has no impact on resilience, scalability, or reliability of each app.

To learn more about how to estimate costs when running in a Consumption plan, see [Understanding Consumption plan costs](#).

Premium plan

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. Premium plan supports the following features:

- Perpetually warm instances to avoid any cold start

- VNet connectivity
- Unlimited execution duration (60 minutes guaranteed)
- Premium instance sizes (one core, two core, and four core instances)
- More predictable pricing
- High-density app allocation for plans with multiple function apps

To learn how you can create a function app in a Premium plan, see [Azure Functions Premium plan](#).

Instead of billing per execution and memory consumed, billing for the Premium plan is based on the number of core seconds and memory allocated across instances. There is no execution charge with the Premium plan. At least one instance must be allocated at all times per plan. This 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.

Consider the Azure Functions Premium plan in the following situations:

- Your function apps run continuously, or nearly continuously.
- You have a high number of small executions and have a high execution bill but low GB second bill in the Consumption plan.
- You need more CPU or memory options than what is provided by the Consumption plan.
- Your code needs to run longer than the [maximum execution time allowed](#) on the Consumption plan.
- You require features that are only available on a Premium plan, such as virtual network connectivity.

Dedicated (App Service) plan

Your function apps can also run on the same dedicated VMs as other App Service apps (Basic, Standard, Premium, and Isolated SKUs).

Consider an 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.

You pay the same for function apps in an App Service Plan as you would for other App Service resources, like web apps. For details about how the App Service plan works, see the [Azure App Service plans in-depth overview](#).

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](#).

When running JavaScript 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](#).

Running in an [App Service Environment](#) (ASE) lets you fully isolate your functions and take advantage of higher number of instances than an App Service Plan.

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. Always on is available only on an App Service plan. On a Consumption plan, the platform activates function apps automatically.

Function app timeout duration

The timeout duration of a function app is defined by the `functionTimeout` property in the `host.json` project file. The following table shows the default and maximum values in minutes for both plans and the different runtime versions:

| PLAN | RUNTIME VERSION | DEFAULT | MAXIMUM |
|-------------|-----------------|-----------|-----------|
| Consumption | 1.x | 5 | 10 |
| Consumption | 2.x | 5 | 10 |
| Consumption | 3.x | 5 | 10 |
| Premium | 1.x | 30 | Unlimited |
| Premium | 2.x | 30 | Unlimited |
| Premium | 3.x | 30 | Unlimited |
| App Service | 1.x | Unlimited | Unlimited |
| App Service | 2.x | 30 | Unlimited |
| App Service | 3.x | 30 | Unlimited |

NOTE

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](#).

Even with Always On enabled, the execution timeout for individual functions is controlled by the `functionTimeout` setting in the `host.json` project file.

Determine the hosting plan of an existing application

To determine the hosting plan used by your function app, see [App Service plan](#) in the Overview tab for the function app in the [Azure portal](#). To see the pricing tier, select the name of the [App Service Plan](#), and then select [Properties](#) from the left pane.

The screenshot shows the Azure portal's Overview tab for a function app named "myfunctionapp". The left sidebar lists several sections: Overview (highlighted with a red box), Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions, Functions (under Functions), App keys, App files, and Proxies. The main content area 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) (highlighted with a red box)
- Properties:** See More
- Runtime version:** 3.0.13139.0

At the bottom of the main content area, there are links for Metrics, Features (8), Notifications (0), and Quickstart.

You can also use the Azure CLI to determine the plan, as follows:

```
appServicePlanId=$(az functionapp show --name <my_function_app_name> --resource-group <my_resource_group> --query appServicePlanId --output tsv)
az appservice plan list --query "[?id=='$appServicePlanId'].sku.tier" --output tsv
```

When the output from this command is `dynamic`, your function app is in the Consumption plan. When the output from this command is `ElasticPremium`, your function app is in the Premium plan. All other values indicate different tiers of an App Service plan.

Storage account requirements

On any plan, a function app requires a general Azure Storage account, which supports Azure Blob, Queue, Files, and Table storage. This is because Azure Functions relies on Azure Storage for operations such as managing triggers and logging function executions, but some storage accounts don't support queues and tables. These accounts, which include blob-only storage accounts (including premium storage) and general-purpose storage accounts with zone-redundant storage replication, are filtered-out from your existing **Storage Account** selections when you create a function app.

The same storage account used by your function app can also be used by your triggers and bindings to store your application data. However, for storage-intensive operations, you should use a separate storage account.

It's possible for multiple function apps to share the same storage account without any issues. (A good example of this is when you develop multiple apps in your local environment using the Azure Storage Emulator, which acts like one storage account.)

To learn more about storage account types, see [Introducing the Azure Storage services](#).

In Region Data Residency

When necessary for all customer data to 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 would also need to be used with [Azure Durable Functions](#) for Durable Functions.

Other platform-managed customer data will only be stored within the region when hosting in an Internal Load Balancer App Service Environment (or ILB ASE). Details can be found in [ASE zone redundancy](#).

How the Consumption and Premium plans work

In the Consumption and Premium plans, the Azure Functions infrastructure scales CPU and memory resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. Each instance of the Functions host in the Consumption plan is limited to 1.5 GB of memory and one CPU. An instance of the host is the entire function app, meaning all functions within a function app share resource within an instance and scale at the same time. Function apps that share the same Consumption plan are scaled independently. In the Premium plan, your plan size will determine the available memory and CPU for all apps in that plan on that instance.

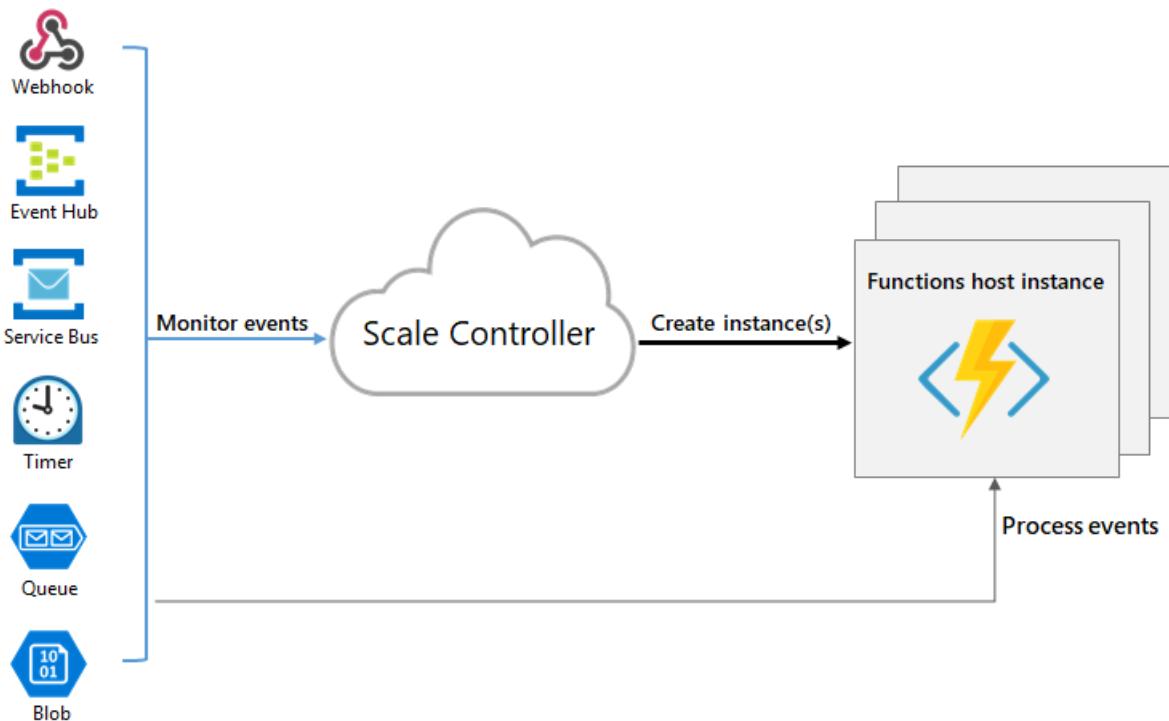
Function code files are stored on Azure Files shares on the function's main storage account. When you delete the main storage account of the function app, the function code files are deleted and cannot be recovered.

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 scales based on the queue length and the age of the oldest queue message.

The unit of scale for Azure Functions is the function app. When the function app is scaled out, additional 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* to zero when no functions are running within a function app.



Cold Start

After your function app has been idle for a number of minutes, the platform may 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 that must be loaded by your function app can impact 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 running in a Premium plan or in a Dedicated plan with Always on enabled.

Understanding scaling behaviors

Scaling can vary on a number of factors, and scale differently based on the trigger and language selected. There are a few intricacies of scaling behaviors to be aware of:

- A single function app only scales out to a maximum of 200 instances. A single instance may process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions. You can [specify a lower maximum](#) to throttle scale as required.
- 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](#).
- For Service Bus triggers, use *Manage* rights on resources for the most efficient scaling. With *Listen* rights, scaling isn't as accurate because the queue 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](#).
- For Event Hub triggers, see the [scaling guidance](#) in the reference article.

Limit scale out

You may wish to restrict the number of instances an app scales out to. This is most common for cases where a downstream component like a database has limited throughput. By default, consumption plan functions will scale out to as many as 200 instances, and premium plan functions will scale out to as many as 100 instances. You can specify a lower maximum for a specific app by modifying the `functionAppScaleLimit` value. The `functionAppScaleLimit` can be set to 0 or null for unrestricted, or a valid value between 1 and the app maximum.

```
az resource update --resource-type Microsoft.Web/sites -g <resource_group> -n <function_app_name>/config/web --set properties.functionAppScaleLimit=<scale_limit>
```

Best practices and patterns for scalable apps

There are many aspects of a function app that will impact how well it will scale, 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](#).

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](#).

Billing model

Billing for the different plans is described in detail on the [Azure Functions pricing page](#). Usage is aggregated at the function app level and counts only the time that function code is executed. The following are units for billing:

- **Resource consumption in gigabyte-seconds (GB-s).** Computed as a combination of memory size and execution time for all functions within a function app.
- **Executions.** Counted each time a function is executed in response to an event trigger.

Useful queries and information on how to understand your consumption bill can be found [on the billing FAQ](#).

Hosting plans comparison

The following comparison table shows all important aspects to help the decision of Azure Functions App hosting plan choice:

Plan summary

| | |
|-------------------------|--|
| Consumption plan | Scale automatically and only pay for compute resources when your functions are running. On the Consumption plan, instances of the Functions host are dynamically added and removed based on the number of incoming events. <ul style="list-style-type: none">✓ Default hosting plan.✓ Pay only when your functions are running.✓ scale-out automatically, even during periods of high load. |
| Premium plan | While automatically scaling based on demand, use pre-warmed workers to run applications with no delay after being idle, run on more powerful instances, and connect to VNETs. Consider the Azure Functions Premium plan in the following situations, in addition to all features of the App Service plan: <ul style="list-style-type: none">✓ Your function apps run continuously, or nearly continuously.✓ You have a high number of small executions and have a high execution bill but low GB second bill in the Consumption plan.✓ You need more CPU or memory options than what is provided by the Consumption plan.✓ Your code needs to run longer than the maximum execution time allowed on the Consumption plan.✓ You require features that are only available on a Premium plan, such as virtual network connectivity. |

| | |
|------------------------------------|---|
| Dedicated plan ¹ | Run your functions within an App Service plan at regular App Service plan rates. Good fit for long running operations, as well as when more predictive scaling and costs are required. Consider an App Service plan in the following situations: <ul style="list-style-type: none">✓ 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. |
| ASE ¹ | App Service Environment (ASE) is an App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale. ASEs are appropriate for application workloads that require: <ul style="list-style-type: none">✓ Very high scale.✓ Full compute isolation and secure network access.✓ High memory utilization. |
| Kubernetes | Kubernetes provides a fully isolated and dedicated environment running on top of the Kubernetes platform. Kubernetes is appropriate for application workloads that require: <ul style="list-style-type: none">✓ Custom hardware requirements.✓ Isolation and secure network access.✓ Ability to run in hybrid or multi-cloud environment.✓ Run alongside existing Kubernetes applications and services. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Operating system/runtime

| | LINUX ¹ CODE-ONLY | WINDOWS ² CODE-ONLY | LINUX ^{1,3} DOCKER CONTAINER |
|------------------------------------|--|---|---|
| Consumption plan | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | No support |
| Premium plan | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | .NET Core Node.js Java PowerShell Core Python |
| Dedicated plan ⁴ | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | .NET Core Node.js Java PowerShell Core Python |
| ASE ⁴ | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | .NET Core Node.js Java PowerShell Core Python |

| | LINUX CODE-ONLY | WINDOWS CODE-ONLY | LINUX DOCKER CONTAINER |
|-------------------|--------------------|----------------------|---|
| Kubernetes | n/a | n/a | .NET Core Node.js Java PowerShell Core Python |

¹Linux is the only supported operating system for the Python runtime stack.

²Windows is the only supported operating system for the PowerShell runtime stack.

³Linux is the only supported operating system for Docker containers.⁴ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Scale

| | SCALE OUT | MAX # INSTANCES |
|-----------------------------------|--|--------------------|
| Consumption plan | Event driven. Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. | 200 |
| Premium plan | Event driven. Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. | 100 |
| Dedicated plan¹ | Manual/autoscale | 10-20 |
| ASE¹ | Manual/autoscale | 100 |
| Kubernetes | Event-driven autoscale for Kubernetes clusters using KEDA . | Varies by cluster. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Cold start behavior

| | |
|-----------------------------------|--|
| Consumption plan | Apps may scale to zero if idle for a period of time, meaning some requests may have additional latency at startup. The consumption plan does have some optimizations to help decrease cold start time, including pulling from pre-warmed placeholder functions that already have the function host and language processes running. |
| Premium plan | Perpetually warm instances to avoid any cold start. |
| Dedicated plan¹ | When running in a Dedicated plan, the Functions host can run continuously, which means that cold start isn't really an issue. |

| | |
|------------------------|---|
| ASE¹ | When running in a Dedicated plan, the Functions host can run continuously, which means that cold start isn't really an issue. |
| Kubernetes | Depends on KEDA configuration. Apps can be configured to always run and never have cold start, or configured to scale to zero, which results in cold start on new events. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Service limits

| RESOURCE | CONSUMPTION PLAN | PREMIUM PLAN | DEDICATED PLAN | ASE | KUBERNETES |
|--|--|--|--|--|-----------------------------|
| Default timeout duration (min) | 5 | 30 | 30 ¹ | 30 | 30 |
| Max timeout duration (min) | 10 | unbounded ⁷ | unbounded ² | unbounded | unbounded |
| Max outbound connections (per instance) | 600 active (1200 total) | unbounded | unbounded | unbounded | unbounded |
| Max request size (MB) ³ | 100 | 100 | 100 | 100 | Depends on cluster |
| Max query string length ³ | 4096 | 4096 | 4096 | 4096 | Depends on cluster |
| Max request URL length ³ | 8192 | 8192 | 8192 | 8192 | Depends on cluster |
| ACU per instance | 100 | 210-840 | 100-840 | 210-250 ⁸ | AKS pricing |
| Max memory (GB per instance) | 1.5 | 3.5-14 | 1.75-14 | 3.5 - 14 | Any node is supported |
| Function apps per plan | 100 | 100 | unbounded ⁴ | unbounded | unbounded |
| App Service plans | 100 per region | 100 per resource group | 100 per resource group | - | - |
| Storage ⁵ | 5 TB | 250 GB | 50-1000 GB | 1 TB | n/a |
| Custom domains per app | 500 ⁶ | 500 | 500 | 500 | n/a |
| 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 | n/a |

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.

Consumption plan uses Azure Files for temporary storage.

⁶ 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.

⁷ Guaranteed for up to 60 minutes.

⁸ 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.

Networking features

| FEATURE | CONSUMPTION PLAN | PREMIUM PLAN | DEDICATED PLAN | ASE | KUBERNETES |
|---|---|--|--|---|---|
| Inbound IP restrictions and private site access | <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 |
| Virtual network integration | <input type="checkbox"/> No | <input checked="" type="checkbox"/> Yes (Regional) | <input checked="" type="checkbox"/> Yes (Regional and Gateway) | <input checked="" type="checkbox"/> Yes | <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 |
| Hybrid connections (Windows only) | <input type="checkbox"/> No | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes |
| Outbound IP restrictions | <input type="checkbox"/> No | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes |

Billing

| | |
|-----------------------------------|--|
| Consumption plan | Pay only for the time your functions run. Billing is based on number of executions, execution time, and memory used. |
| Premium plan | Premium plan is based on the number of core seconds and memory used across needed and pre-warmed instances. At least one instance per plan must be kept warm at all times. This plan provides more predictable pricing. |
| Dedicated plan¹ | You pay the same for function apps in an App Service Plan as you would for other App Service resources, like web apps. |
| ASE¹ | there's a flat monthly rate for an ASE that pays for the infrastructure and doesn't change with the size of the ASE. In addition, there's a cost per App Service plan vCPU. All apps hosted in an ASE are in the Isolated pricing SKU. |

Kubernetes

You pay only the costs of your Kubernetes cluster; no additional billing for Functions. Your function app runs as an application workload on top of your cluster, just like a regular app.

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Next steps

- [Quickstart: Create an Azure Functions project using Visual Studio Code](#)
- [Deployment technologies in Azure Functions](#)
- [Azure Functions developer guide](#)

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

12/4/2020 • 6 minutes to read • [Edit Online](#)

In this article, you use Visual Studio Code to create a C# class library-based 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

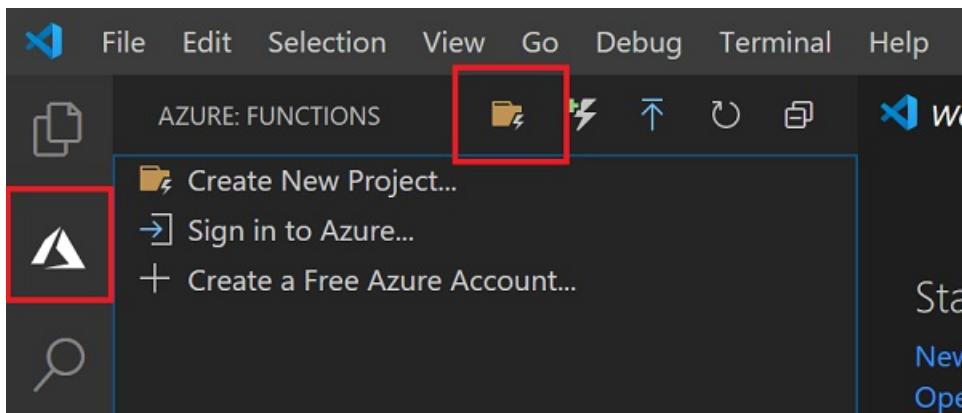
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js](#), required by Windows for npm. Only [Active LTS](#) and [Maintenance LTS versions](#). Use the `node --version` command to check your version. Not required for local development on macOS and Linux.
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [C# extension](#) for Visual Studio Code.
- The [Azure Functions extension](#) for Visual Studio Code.

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. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Provide the following information at the prompts:

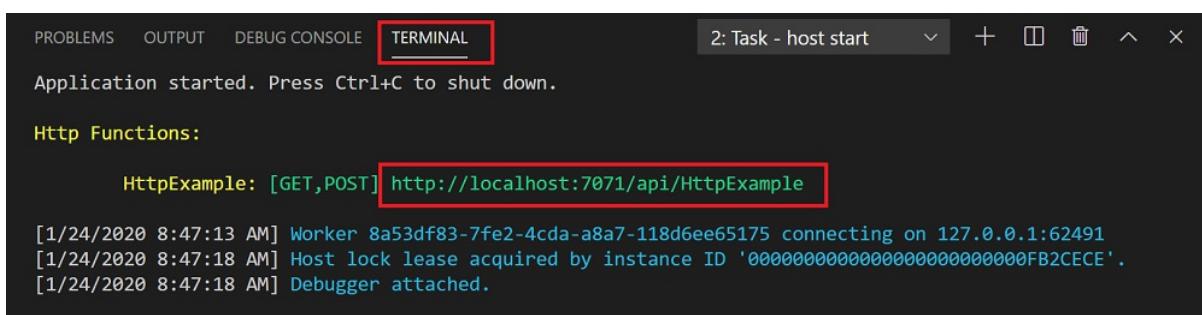
- Select a language for your function project: Choose `C#`.
- Select a template for your project's first function: Choose `HTTP trigger`.
- Provide a function name: Type `HttpExample`.
- Provide a namespace: Type `My.Functions`.
- Authorization level: Choose `Anonymous`, which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
- Select how you would like to open your project: Choose `Add to workspace`.

4. 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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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
2: Task - host start + □ ×
Application started. Press Ctrl+C to shut down.

Http Functions:

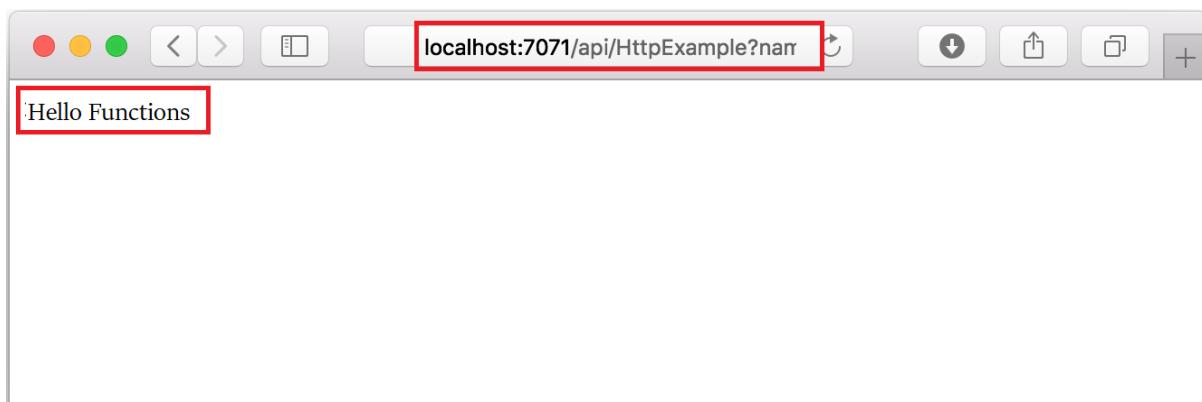
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '00000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.
```

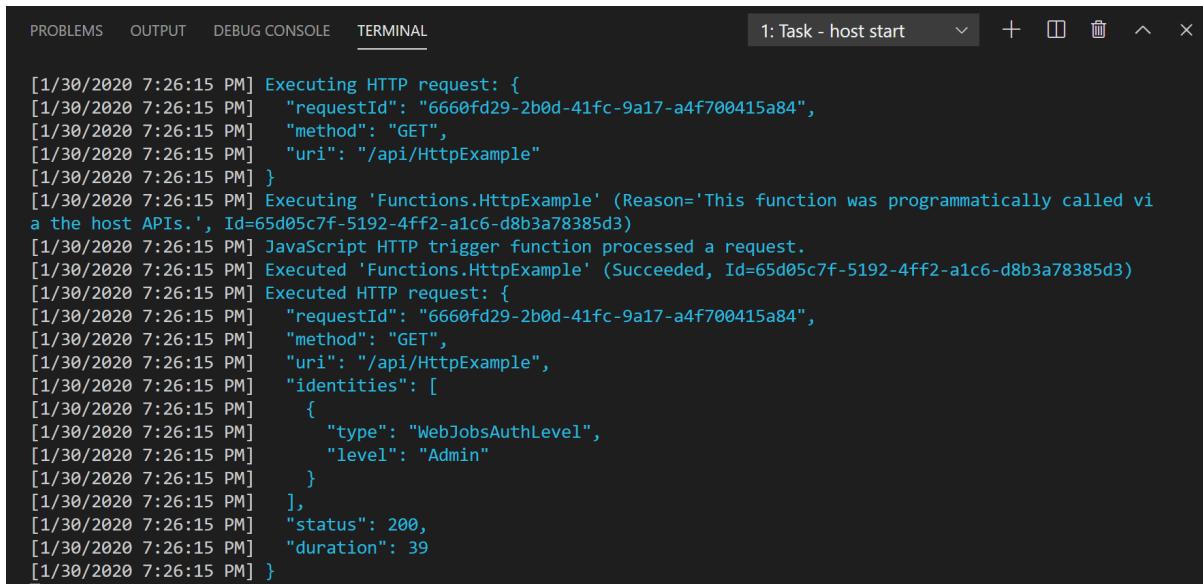
3. With Core Tools running, navigate to the following URL to execute a GET request, which includes `?name=Functions` query string.

```
http://localhost:7071/api/HttpExample?name=Functions
```

4. A response is returned, which looks like the following in a browser:



5. 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 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]   "status": 200,
[1/30/2020 7:26:15 PM]   "duration": 39
[1/30/2020 7:26:15 PM]
```

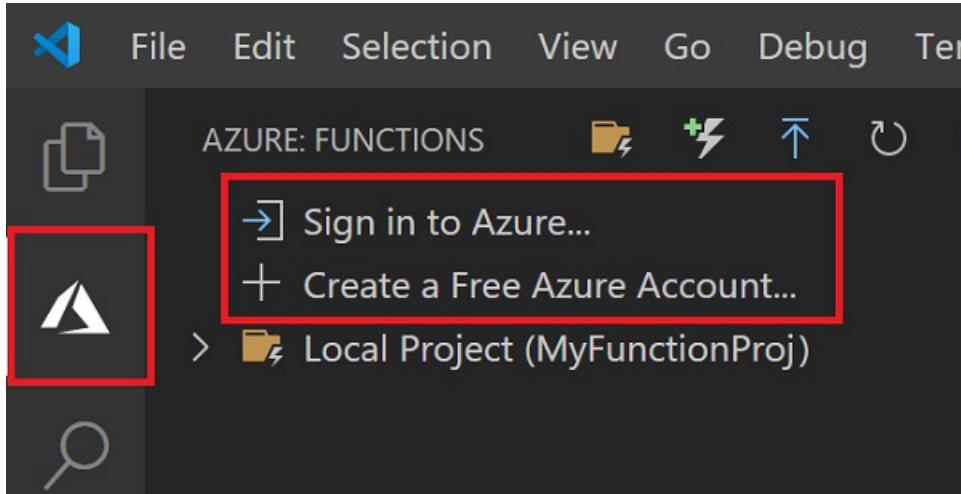
6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure**.... If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

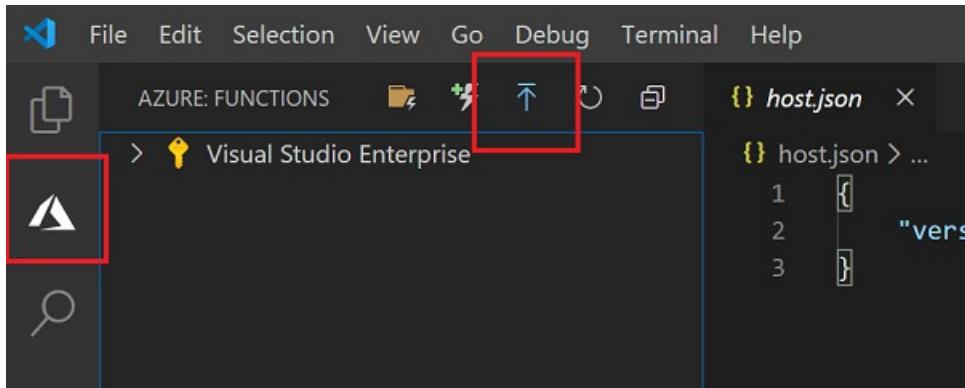
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to function app...** button.



2. Provide the following information at the prompts:

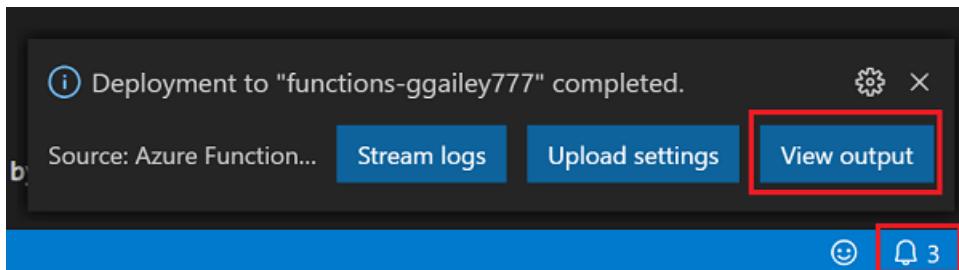
- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose (Don't choose the option, which isn't covered in this article.)
- **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.

3. When completed, the following Azure resources are created in your subscription, using names 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 consumption plan, which defines the underlying host for your serverless function app.
- 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 Application Insights instance connected to the function app, which tracks usage of your serverless function.

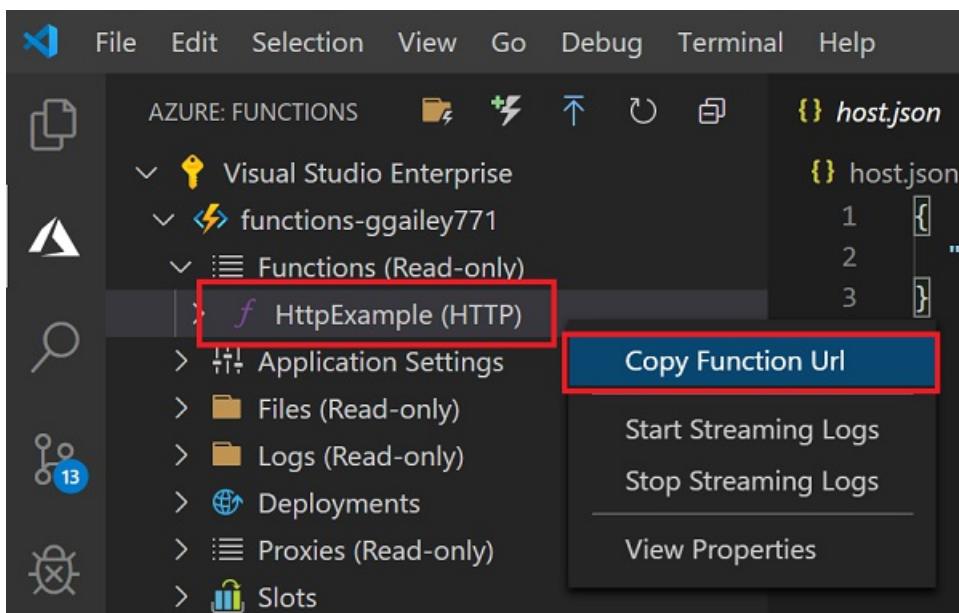
A notification is displayed after your function app is created and the deployment package is applied.

4. Select **View Output** in this notification 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 **Azure: Functions** area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl + click (macOS) on **HttpExample**, and then choose **Copy function URL**.



2. Paste this URL for the HTTP request into your browser's address bar, add the `?name=Functions` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



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 Functions: 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 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 titled 'myfunctionapp' and shows the 'Overview' tab selected. A red box highlights the 'Resource group (change)' section, which displays 'myResourceGroup'. To the right, detailed information is provided: 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), and Runtime version (3.0.13139.0).

4. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 6 minutes to read • [Edit Online](#)

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.

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

NOTE

If Visual Studio Code isn't your preferred development tool, check out our similar tutorials for Java developers using [Maven](#), [Gradle](#) and [IntelliJ IDEA](#).

Configure your environment

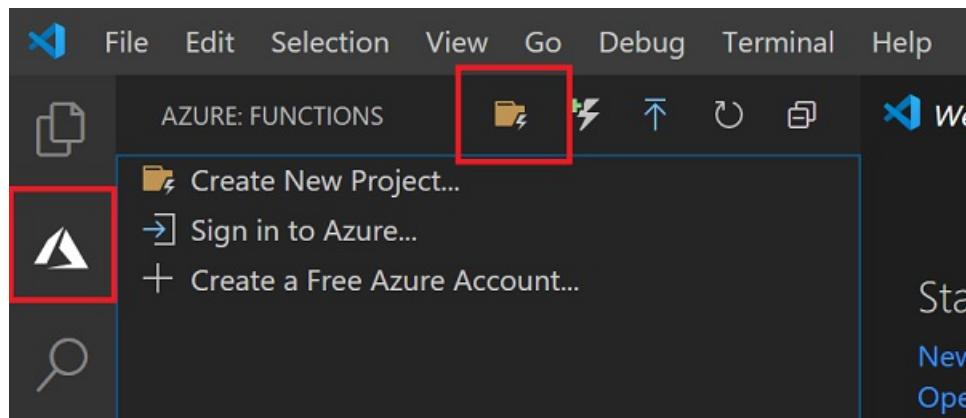
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- The [Java Developer Kit](#), version 8 or 11.
- [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.

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. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Provide the following information at the prompts:

- Select a language for your function project: Choose Java .
 - Select a version of Java: Choose Java 8 or Java 11 , the Java version on which your functions run in Azure. Choose a Java version that you've verified locally.
 - Provide a group ID: Choose com.function .
 - Provide an artifact ID: Choose myFunction .
 - Provide a version: Choose 1.0-SNAPSHOT .
 - Provide a package name: Choose com.function .
 - Provide an app name: Choose myFunction-12345 .
 - Authorization level: Choose Anonymous , which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
 - Select how you would like to open your project: Choose Add to workspace .

- 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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
 2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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 message "Application started. Press Ctrl+C to shut down." followed by "Http Functions:" and "HttpExample: [GET,POST] http://localhost:7071/api/HttpExample". Below this, three log entries are shown: "Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491", "Host lock lease acquired by instance ID '0000000000000000000000000000FB2CECE'.", and "Debugger attached.".

```
Application started. Press Ctrl+C to shut down.

Http Functions:

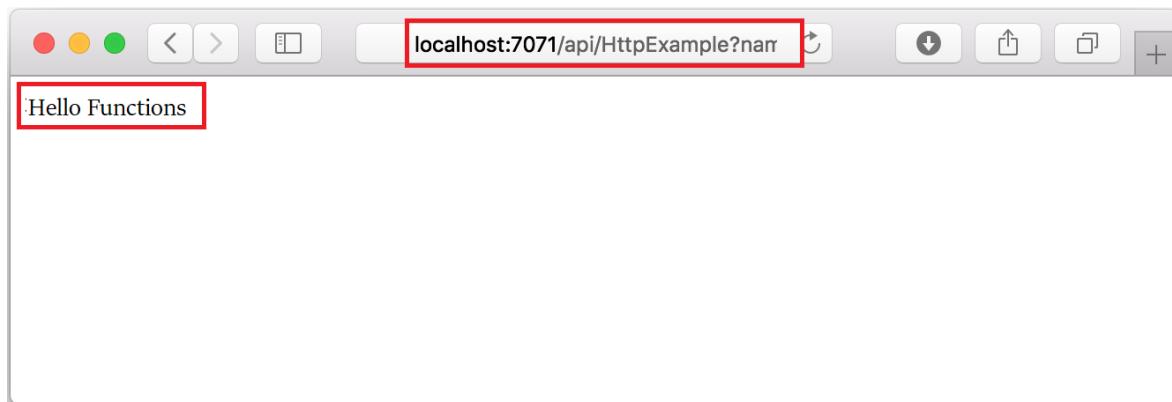
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '0000000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.
```

3. 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>

4. A response is returned, which looks like the following in a browser:



5. 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 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] }
```

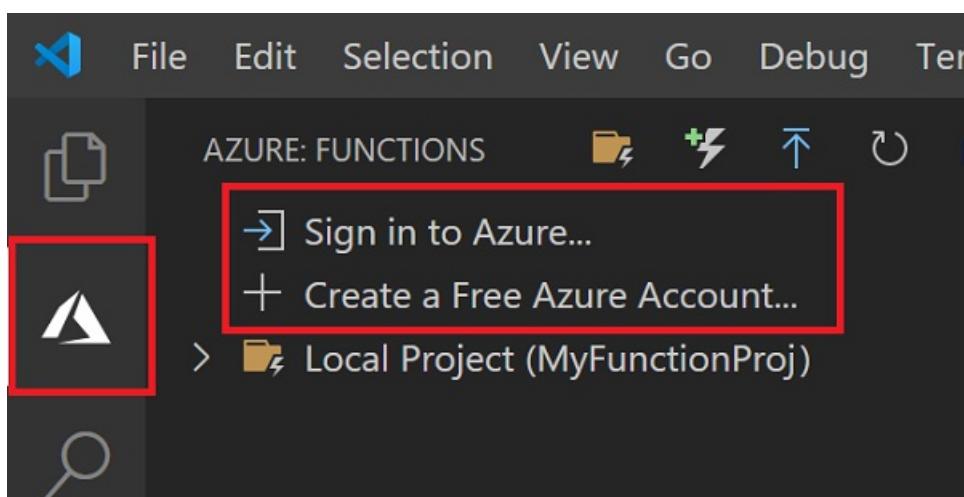
6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure**.... If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

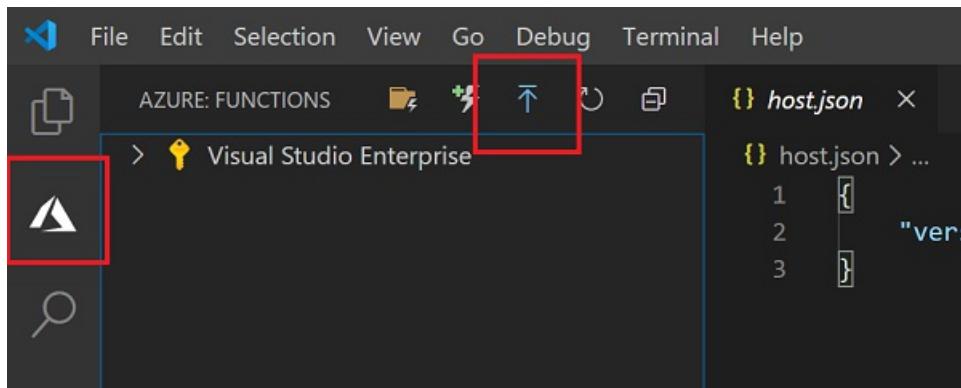
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to function app...** button.



2. Provide the following information at the prompts:

- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose (Don't choose the Advanced option, which isn't covered in this article.)
- **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.

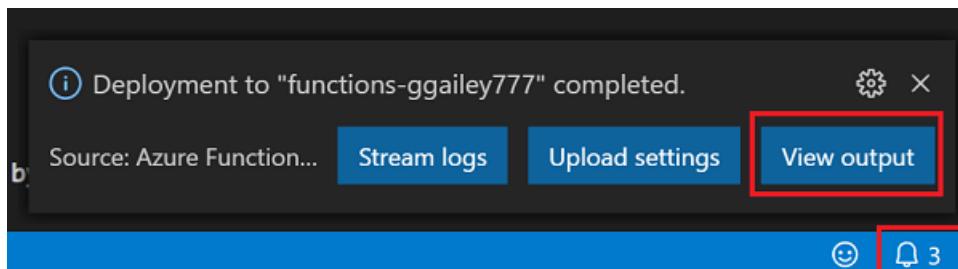
3. When completed, the following Azure resources are created in your subscription, using names based on your function app name:

- 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 consumption plan, which defines the underlying host for your serverless function app.
- 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 Application Insights instance connected to the function app, which tracks usage of your serverless function.

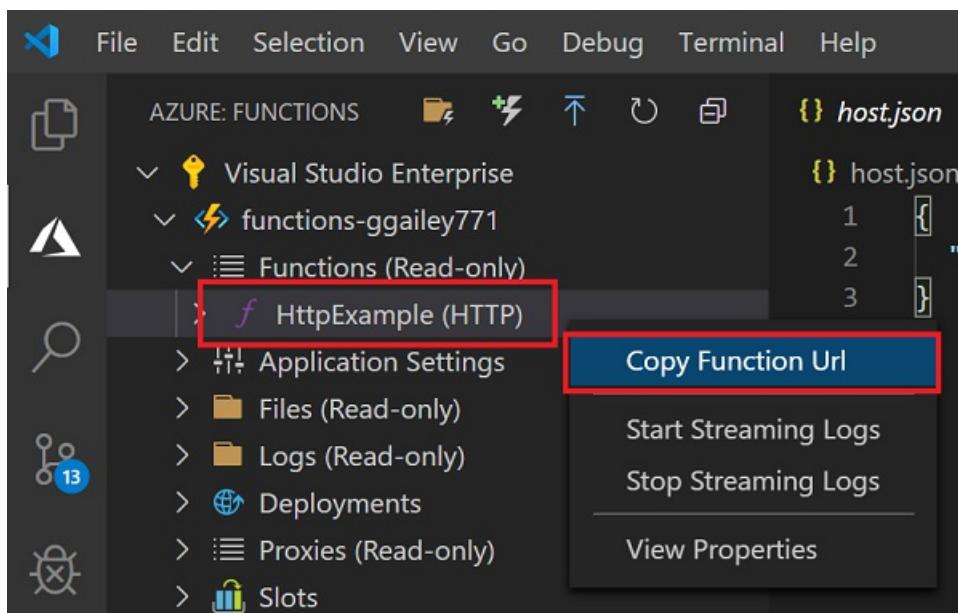
A notification is displayed after your function app is created and the deployment package is applied.

- Select **View Output** in this notification 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

- Back in the **Azure: Functions** area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl - click (macOS) on **HttpExample**, and then choose **Copy function URL**.



- Paste this URL for the HTTP request into your browser's address bar, add the `name` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



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 Functions: 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**.

A screenshot of the Azure portal's Overview page for a function app named "myfunctionapp". The left sidebar shows navigation links like 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 displays basic app 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 Resource group section. The URL https://myfunctionapp.azurewebsites.net is shown at the top right.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 6 minutes to read • [Edit Online](#)

In this article, you use Visual Studio Code 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.

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

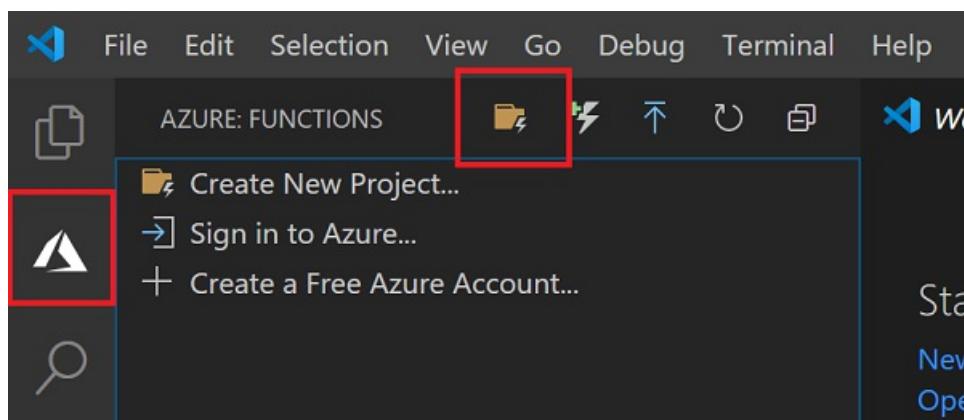
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js](#), Active LTS and Maintenance LTS versions (10.14.1 recommended). Use the `node --version` command to check your version.
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [Azure Functions extension](#) for Visual Studio Code.

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'll publish your function code to Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Provide the following information at the prompts:

- Select a language for your function project: Choose `JavaScript`.
 - Select a template for your project's first function: Choose `HTTP trigger`.
 - Provide a function name: Type `HttpExample`.
 - Authorization level: Choose `Anonymous`, which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
 - Select how you would like to open your project: Choose `Add to workspace`.

4. 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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
 2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



Application started. Press Ctrl+C to shut down.

Http Functions:

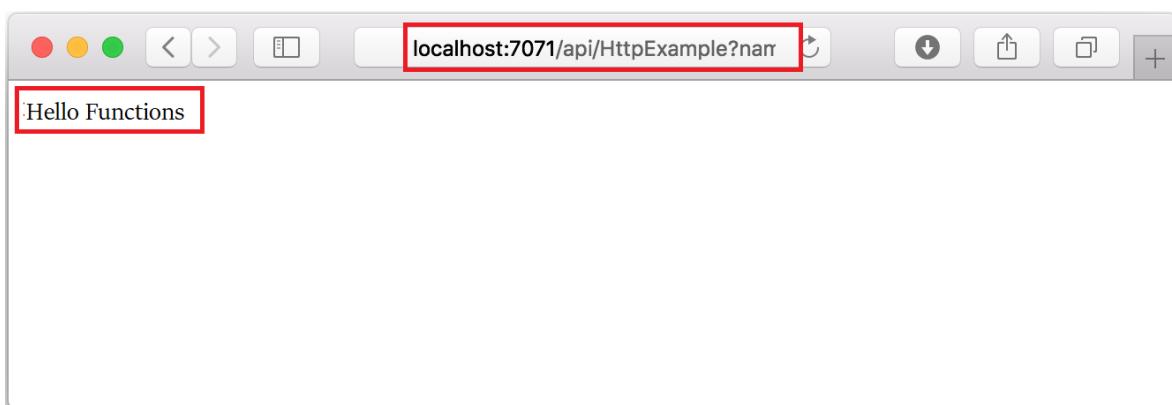
 HttpExample: [GET,POST] <http://localhost:7071/api/HttpExample>

```
[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '0000000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.
```

3. 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>

4. A response is returned, which looks like the following in a browser:



5. Information about the request is shown in **Terminal** panel.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Task - host start + □ ⌂ ⌄ ×

[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 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]   {
[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]
```

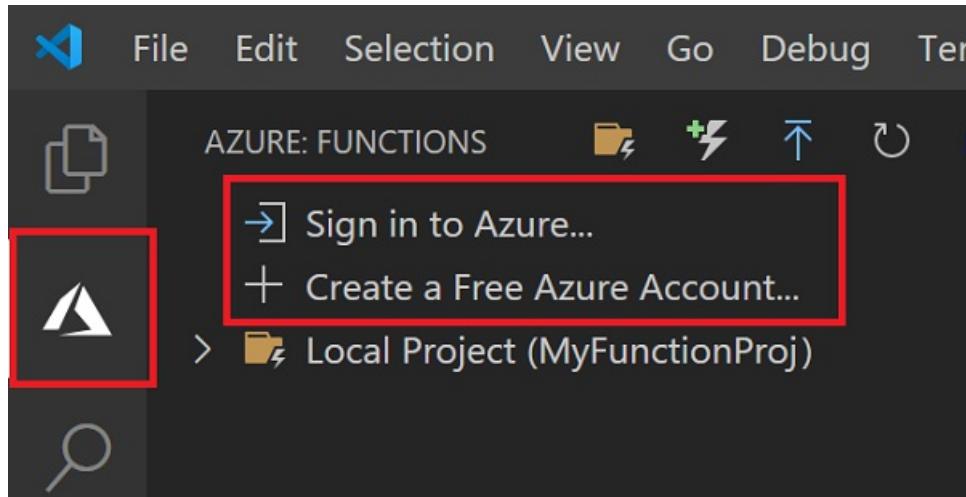
6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure**.... If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

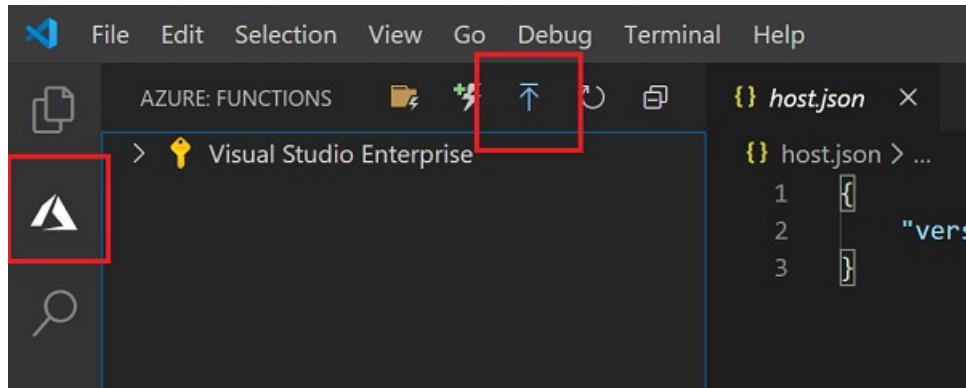
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the Azure: Functions area, choose the Deploy to function app... button.



2. Provide the following information at the prompts:

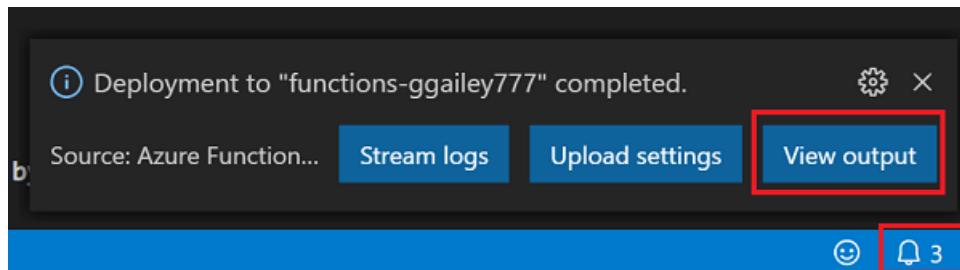
- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose `+ Create new Function App`. (Don't choose the `Advanced` option, which isn't covered in this article.)
- **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:** Choose the version of Node.js you've been running on locally. You can use the `node --version` command to check your version.
- **Select a location for new resources:** For better performance, choose a `region` near you.

3. When completed, the following Azure resources are created in your subscription, using names 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 consumption plan, which defines the underlying host for your serverless function app.
- 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 Application Insights instance connected to the function app, which tracks usage of your serverless function.

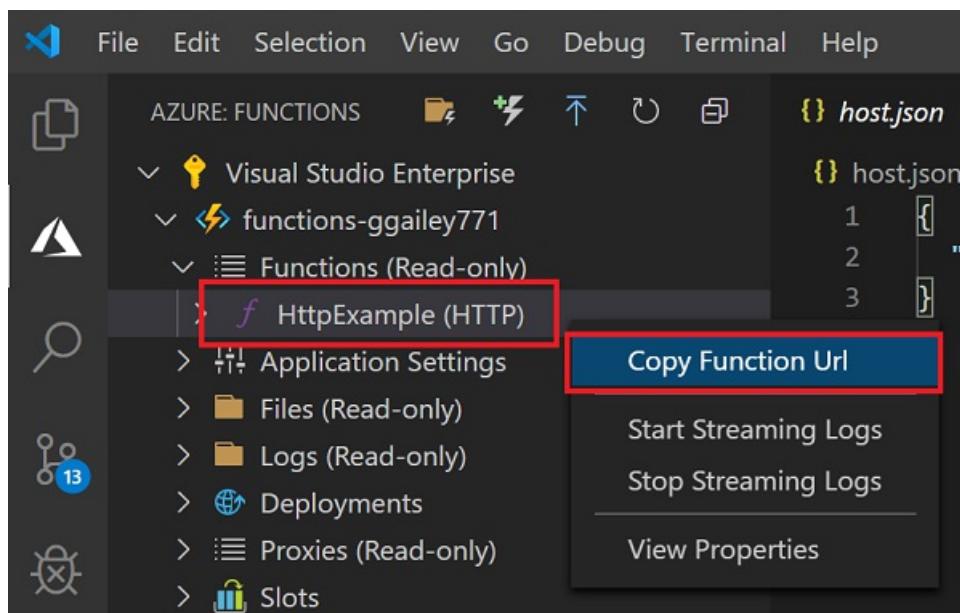
A notification is displayed after your function app is created and the deployment package is applied.

4. Select **View Output** in this notification 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

- Back in the Azure: Functions area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl - click (macOS) on **HttpExample**, and then choose **Copy function URL**.



- Paste this URL for the HTTP request into your browser's address bar, add the `name` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



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 Functions: 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 a "Overview" section selected, indicated by a red box. Below it is a list of other tabs: Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions, Functions, App keys, App files, and Proxies. The main content area shows the following details:

| Setting | Value |
|-------------------------|---|
| 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. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 6 minutes to read • [Edit Online](#)

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

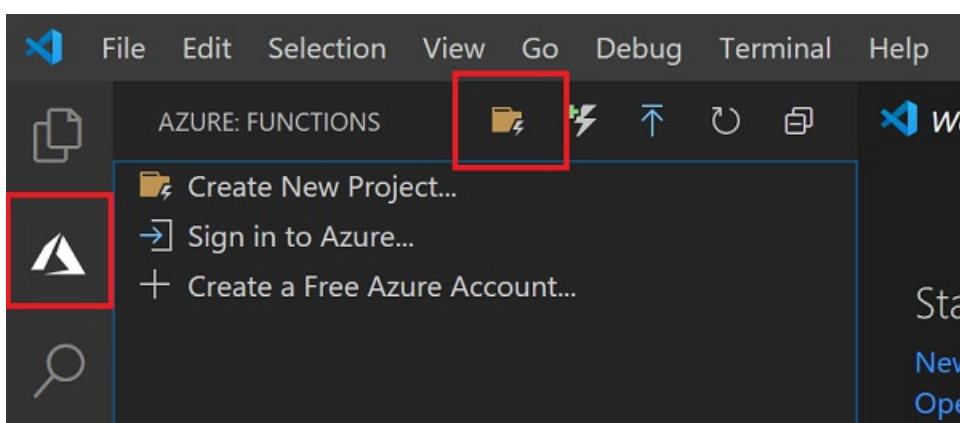
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js](#), required by Windows for npm. Only [Active LTS and Maintenance LTS versions](#). Use the `node --version` command to check your version. Not required for local development on macOS and Linux.
- [PowerShell 7](#)
- Both [.NET Core 3.1 runtime](#) and [.NET Core 2.1 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.

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. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Provide the following information at the prompts:

- Select a language for your function project: Choose `PowerShell`.
 - Select a template for your project's first function: Choose `HTTP trigger`.
 - Provide a function name: Type `HttpExample`.
 - Authorization level: Choose `Anonymous`, which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
 - Select how you would like to open your project: Choose `Add to workspace`.

- 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

Azure Functions Core Tools integrates with Visual Studio Code to let you run and debug an Azure Functions project locally. For details on how to debug in Visual Studio Code, see [Debug PowerShell Azure Functions locally](#).

1. Press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
 2. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

4: Task - host start ▾ + ⚡ 🗑️ ⌂ ⌄ ×

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

Http Functions:

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

```
[4/20/2019 6:19:06 AM] System Log: {
[4/20/2019 6:19:06 AM]   Log-Message: The enforced concurrency level (pool size limit) is '1'.
[4/20/2019 6:19:06 AM] }
[4/20/2019 6:19:06 AM] System Log: {
```

unctions (MyFunctionProj) 🔥 Ln 10, Col 28 (27 selected) Spaces: 4 UTF-8 LF PowerShell ✘ 6.2 😊 🔔 1

3. Append the query string `?name=<yourusername>` to this URL, and then use `Invoke-RestMethod` in a second PowerShell command prompt to execute the request, as follows:

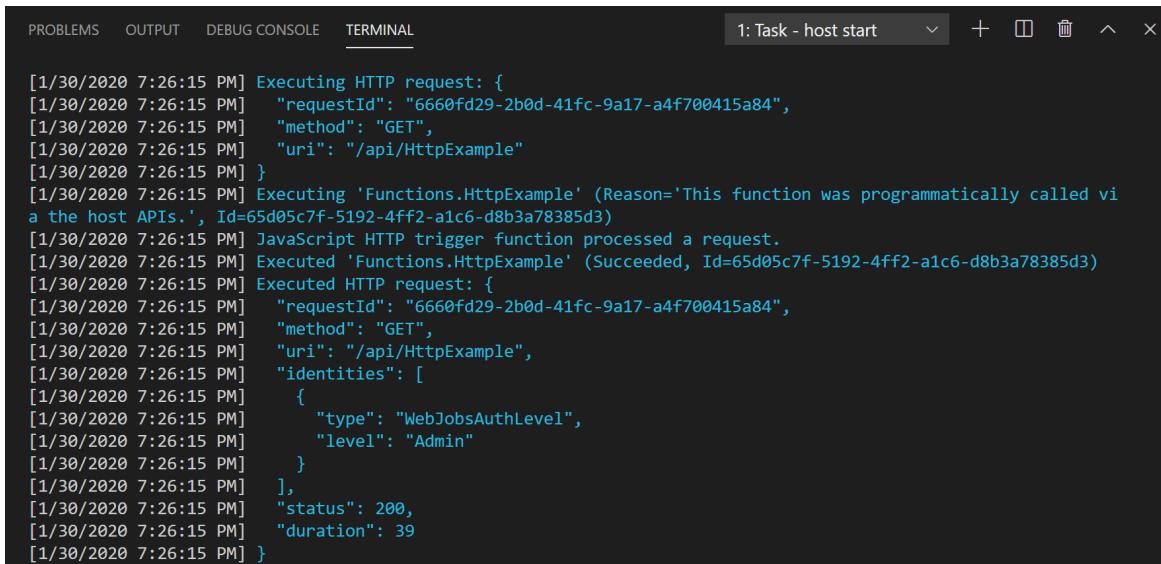
```
PS > Invoke-RestMethod -Method Get -Uri http://localhost:7071/api/HttpTrigger?name=PowerShell  
Hello PowerShell
```

You can also execute the GET request from a browser from the following URL:

<http://localhost:7071/api/HttpExample?name=PowerShell>

When you call the `HttpTrigger` endpoint without passing a `name` parameter either as a query parameter or in the body, the function returns a `BadRequest` error. When you review the code in `run.ps1`, you see that this error occurs by design.

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 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]   {
[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. When done, press **Ctrl + C** to stop Core Tools.

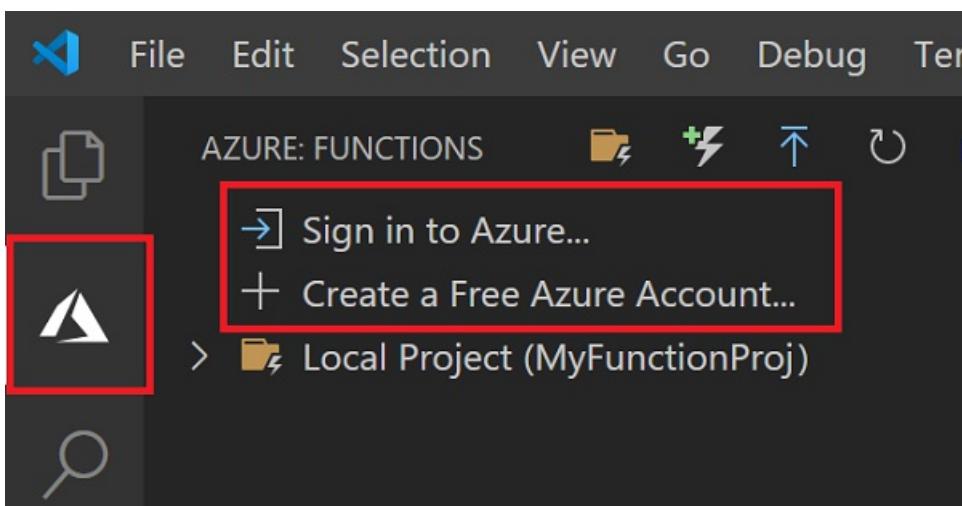
After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure....** If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

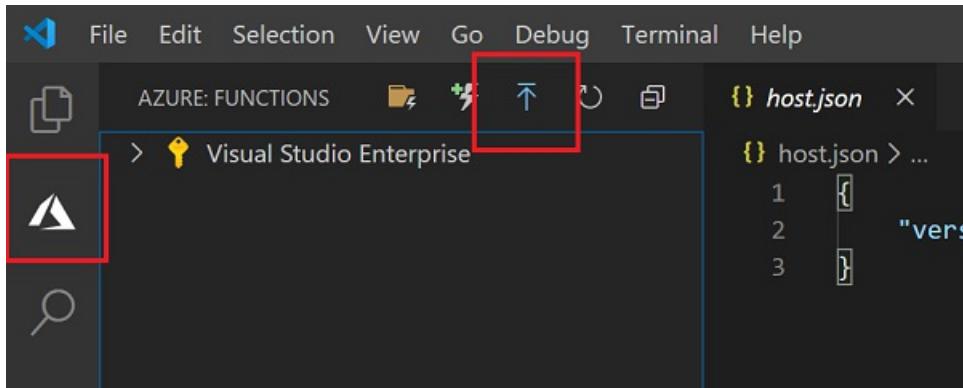
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to function app...** button.



2. Provide the following information at the prompts:

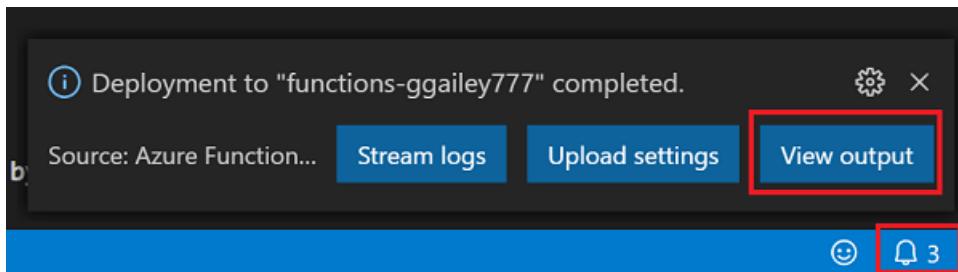
- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose . (Don't choose the Advanced option, which isn't covered in this article.)
- **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.

3. When completed, the following Azure resources are created in your subscription, using names 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 consumption plan, which defines the underlying host for your serverless function app.
- 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 Application Insights instance connected to the function app, which tracks usage of your serverless function.

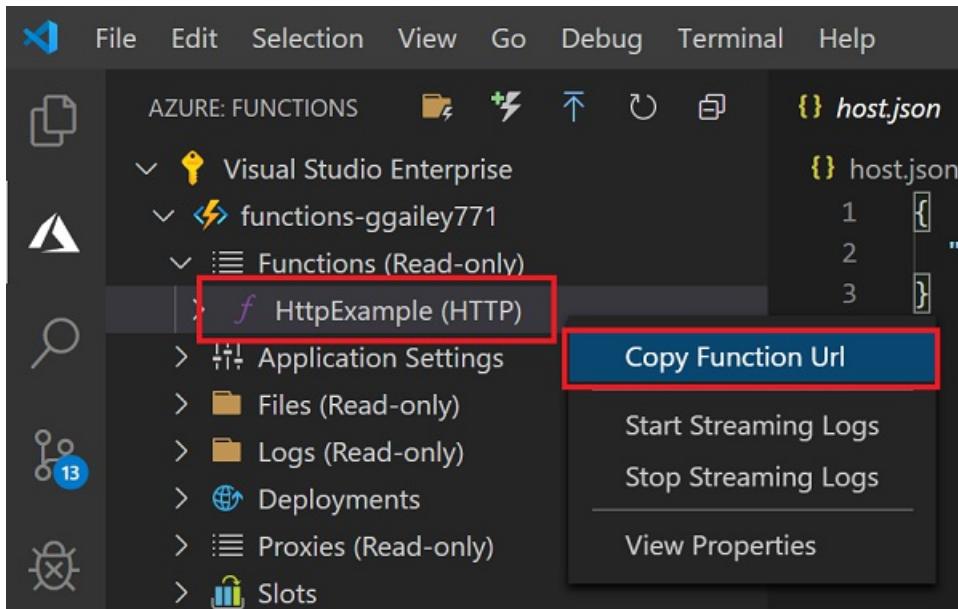
A notification is displayed after your function app is created and the deployment package is applied.

4. Select **View Output** in this notification 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

- Back in the Azure: Functions area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl1 - click (macOS) on **HttpExample**, and then choose **Copy function URL**.



- Paste this URL for the HTTP request into your browser's address bar, add the `name` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



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 Functions: 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 a "Resource group (change)" link highlighted with a red box, pointing to "myResourceGroup". The main content area displays various details about the function app, including its status (Running), location (Central US), subscription (Visual Studio Enterprise), and runtime version (3.0.13139.0). The URL is https://myfunctionapp.azurewebsites.net. The bottom navigation bar includes Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

Quickstart: Create a function in Azure with Python using Visual Studio Code

12/4/2020 • 6 minutes to read • [Edit Online](#)

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.

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

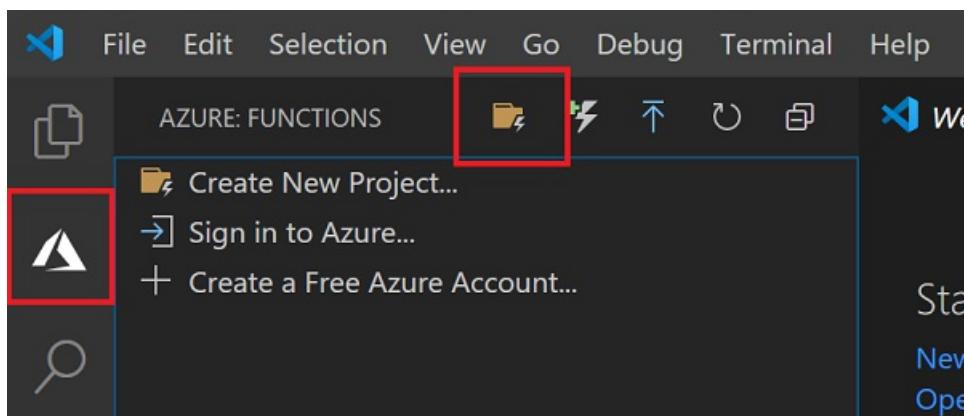
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js](#), required by Windows for npm. Only [Active LTS and Maintenance LTS versions](#). Use the `node --version` command to check your version. Not required for local development on macOS and Linux.
- [Python 3.8, Python 3.7, Python 3.6](#) are supported by Azure Functions (x64).
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [Python extension](#) for Visual Studio Code.
- The [Azure Functions extension](#) for Visual Studio Code.

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'll publish your function code to Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Provide the following information at the prompts:

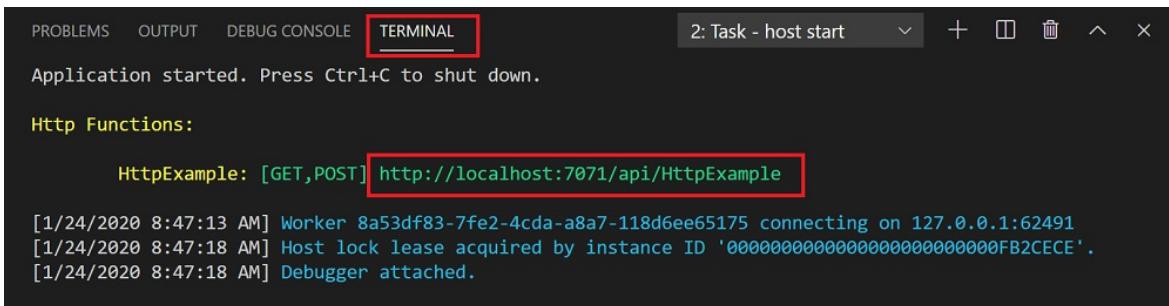
- **Select a language for your function project:** Choose `Python`.
- **Select a Python alias to create a virtual environment:** Choose the location of your Python interpreter. If the location isn't shown, type in the full path to your Python binary.
- **Select a template for your project's first function:** Choose `HTTP trigger`.
- **Provide a function name:** Type `HttpExample`.
- **Authorization level:** Choose `Anonymous`, which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
- **Select how you would like to open your project:** Choose `Add to workspace`.

4. 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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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
2: Task - host start + □ □ ×
Application started. Press Ctrl+C to shut down.

Http Functions:

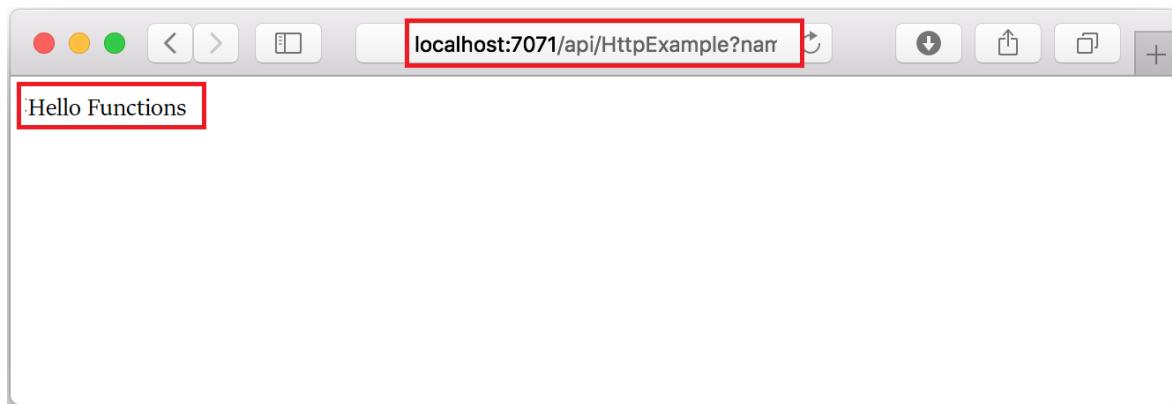
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '00000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.
```

3. 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`

4. A response is returned, which looks like the following in a browser:



5. 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 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] }
```

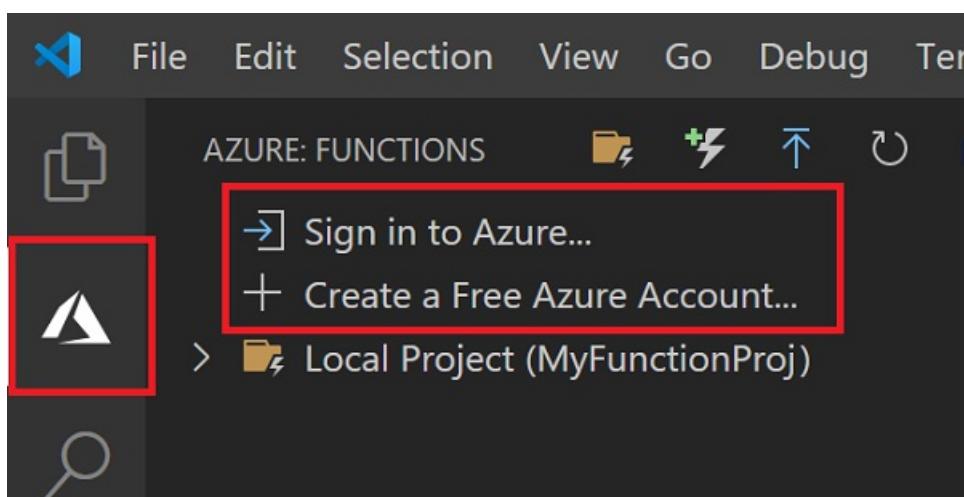
6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure**.... If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

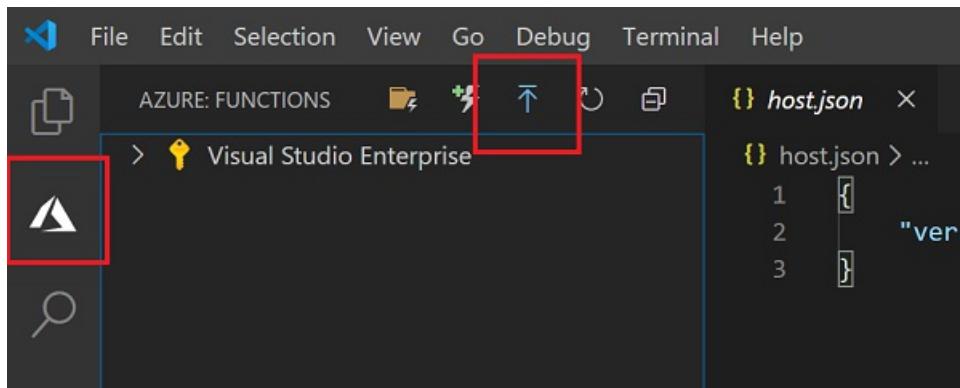
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to function app...** button.



2. Provide the following information at the prompts:

- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose `+ Create new Function App`. (Don't choose the `Advanced` option, which isn't covered in this article.)
- **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:** Choose the version of Python you've been running on locally. You can use the `python --version` command to check your version.
- **Select a location for new resources:** For better performance, choose a [region](#) near you.

3. When completed, the following Azure resources are created in your subscription, using names based on your function app name:

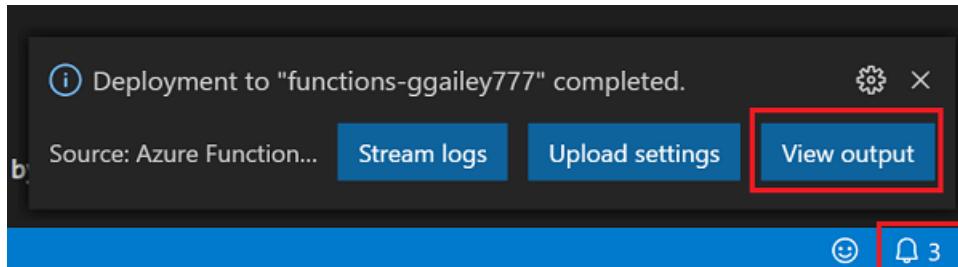
- A resource group, which is a logical container for related resources.
- A standard Azure Storage account, which maintains state and other information about your projects.
- A consumption plan, which defines the underlying host for your serverless function app.
- A function app, which provides the environment 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 Application Insights instance connected to the function app, which tracks usage of your serverless function.

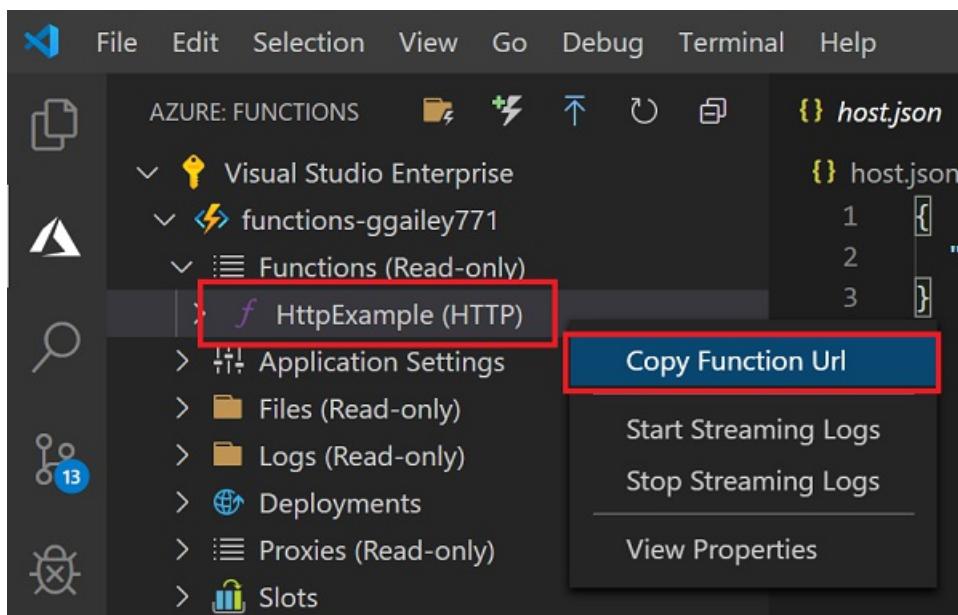
A notification is displayed after your function app is created and the deployment package is applied.

4. Select **View Output** in this notification 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 **Azure: Functions** area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl1 - click (macOS) on **HttpExample**, and then choose **Copy function URL**.



2. Paste this URL for the HTTP request into your browser's address bar, add the `?name=Functions` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



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 Functions: 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**.

A screenshot of the Azure portal's Overview page for a function app named "myfunctionapp". The left sidebar shows navigation links like 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 displays the app's status as "Running", location as "Central US", and subscription as "Visual Studio Enterprise". It also shows the URL "https://myfunctionapp.azurewebsites.net", operating system as "Windows", and app service plan as "ASP-myResourceGroup-a285 (Y1: 0)". A "Resource group (change)" link is highlighted with a red box. Below it, the resource group name "myResourceGroup" is shown. At the bottom of the page, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

Having issues? Let us know.

Quickstart: Create a function in Azure with TypeScript using Visual Studio Code

12/4/2020 • 6 minutes to read • [Edit Online](#)

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.

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

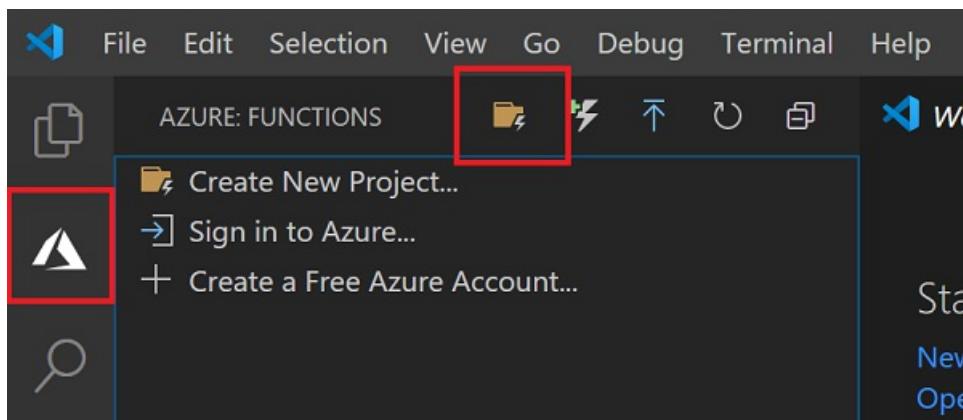
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js](#), Active LTS and Maintenance LTS versions (10.14.1 recommended). Use the `node --version` command to check your version.
- [Visual Studio Code](#) on one of the supported platforms.
- The [Azure Functions extension](#) for Visual Studio Code.

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'll publish your function code to Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Provide the following information at the prompts:

- Select a language for your function project: Choose `TypeScript`.
 - Select a template for your project's first function: Choose `HTTP trigger`.
 - Provide a function name: Type `HttpExample`.
 - Authorization level: Choose `Anonymous`, which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
 - Select how you would like to open your project: Choose `Add to workspace`.

4. 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

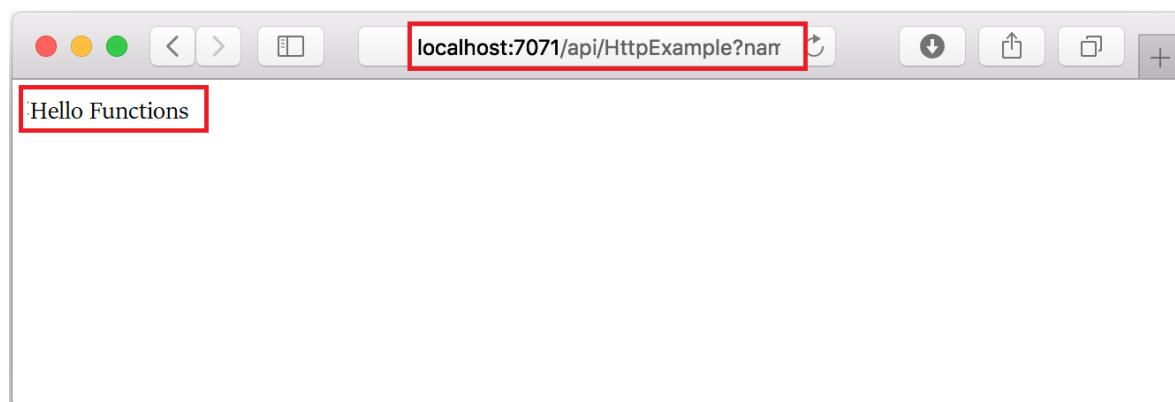
Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

1. To call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
 2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.

3. 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>

4. A response is returned, which looks like the following in a browser:



5. Information about the request is shown in Terminal panel.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Task - host start + □ ⌂ ⌄ ×

[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 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]   {
[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]
```

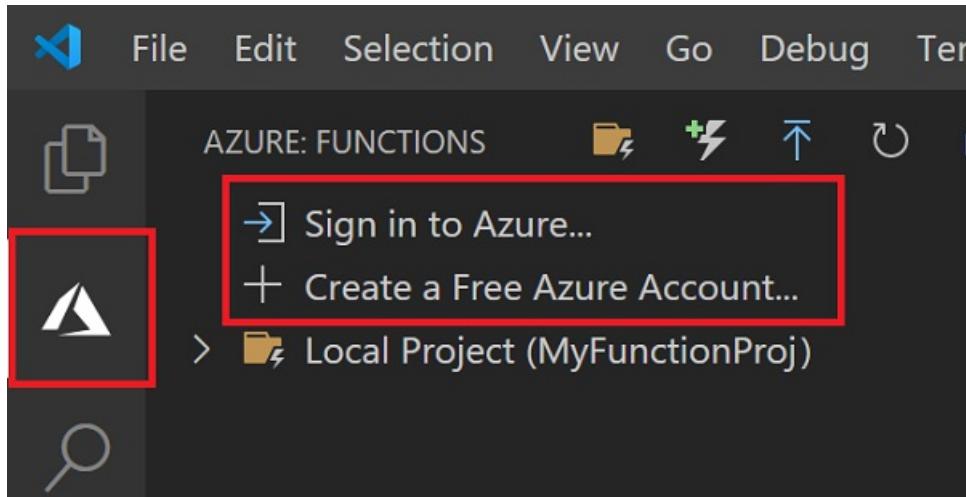
6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure...**. If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

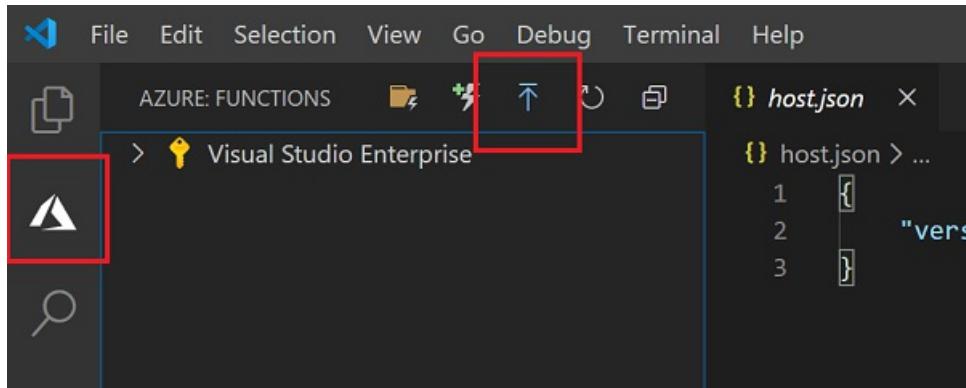
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to function app...** button.



2. Provide the following information at the prompts:

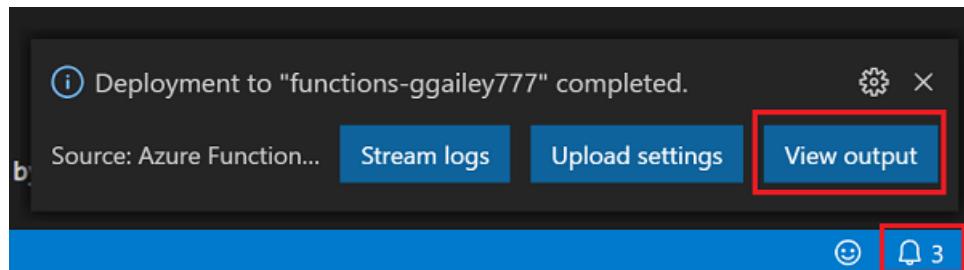
- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose `+ Create new Function App`. (Don't choose the `Advanced` option, which isn't covered in this article.)
- **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:** Choose the version of Node.js you've been running on locally. You can use the `node --version` command to check your version.
- **Select a location for new resources:** For better performance, choose a `region` near you.

3. When completed, the following Azure resources are created in your subscription, using names 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 consumption plan, which defines the underlying host for your serverless function app.
- 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 Application Insights instance connected to the function app, which tracks usage of your serverless function.

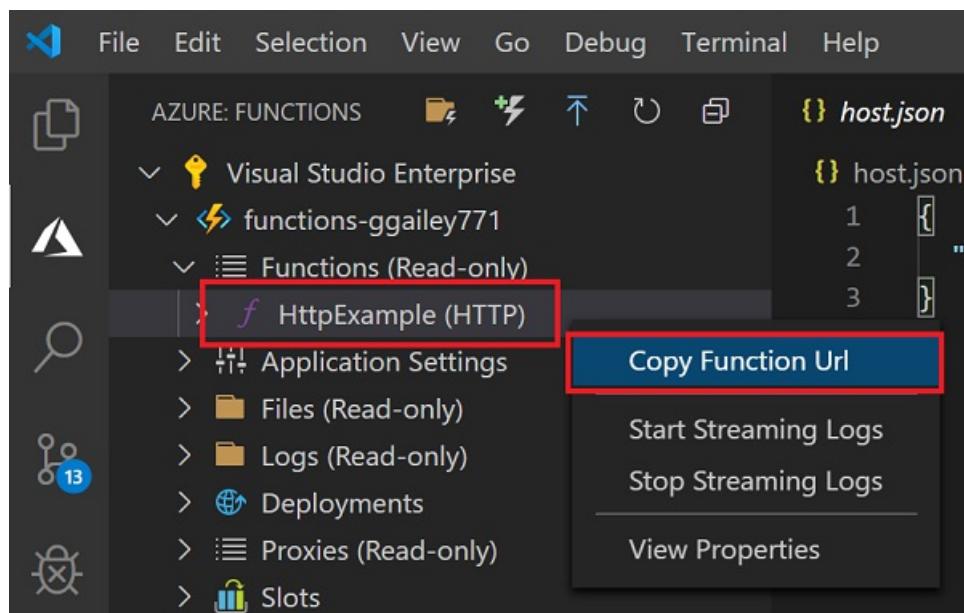
A notification is displayed after your function app is created and the deployment package is applied.

4. Select **View Output** in this notification 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 **Azure: Functions** area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl - click (macOS) on **HttpExample**, and then choose **Copy function URL**.



2. Paste this URL for the HTTP request into your browser's address bar, add the `name` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



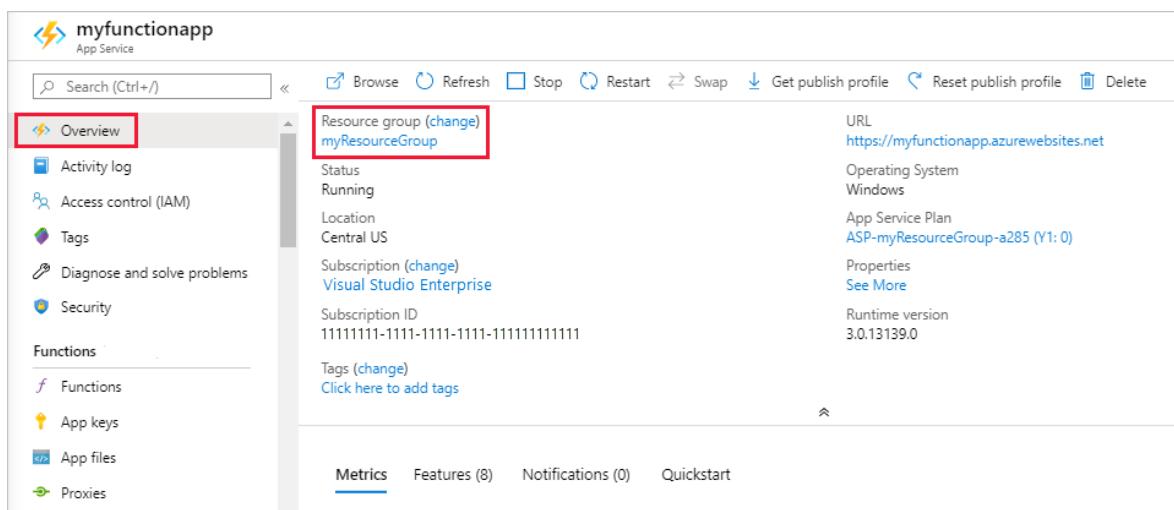
Clean up resources

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 Functions: 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 a "Overview" tab selected, indicated by a red box. The main content area displays various details about the app, including its status (Running), location (Central US), subscription (Visual Studio Enterprise), and runtime version (3.0.13139.0). A "Resource group" link is highlighted with a red box and labeled "myResourceGroup". The URL of the app is https://myfunctionapp.azurewebsites.net. The bottom navigation bar includes Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

Quickstart: Create your first function in Azure using Visual Studio

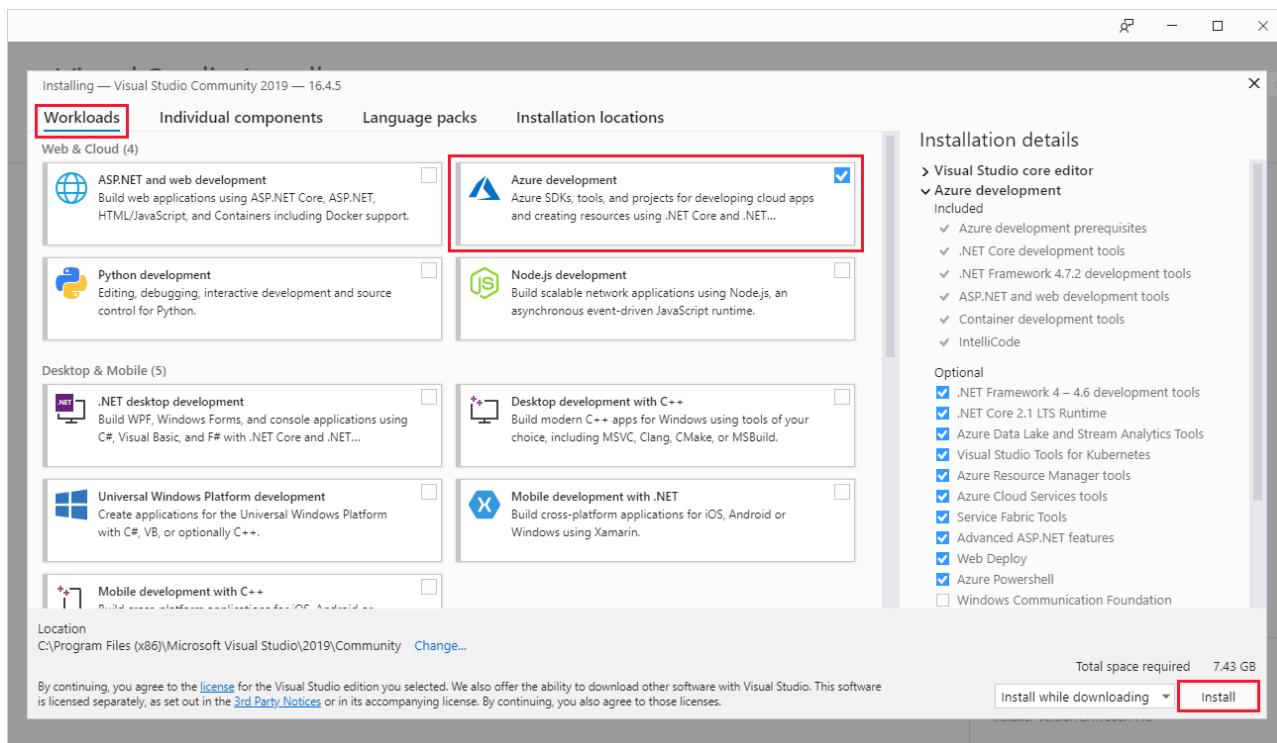
12/4/2020 • 7 minutes to read • [Edit Online](#)

In this article, you use Visual Studio to create a C# class library-based 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.

Prerequisites

To complete this tutorial, first install [Visual Studio 2019](#). Ensure you select the **Azure development** workload during installation. If you want to create an Azure Functions project by using Visual Studio 2017 instead, you must first install the [latest Azure Functions tools](#).



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

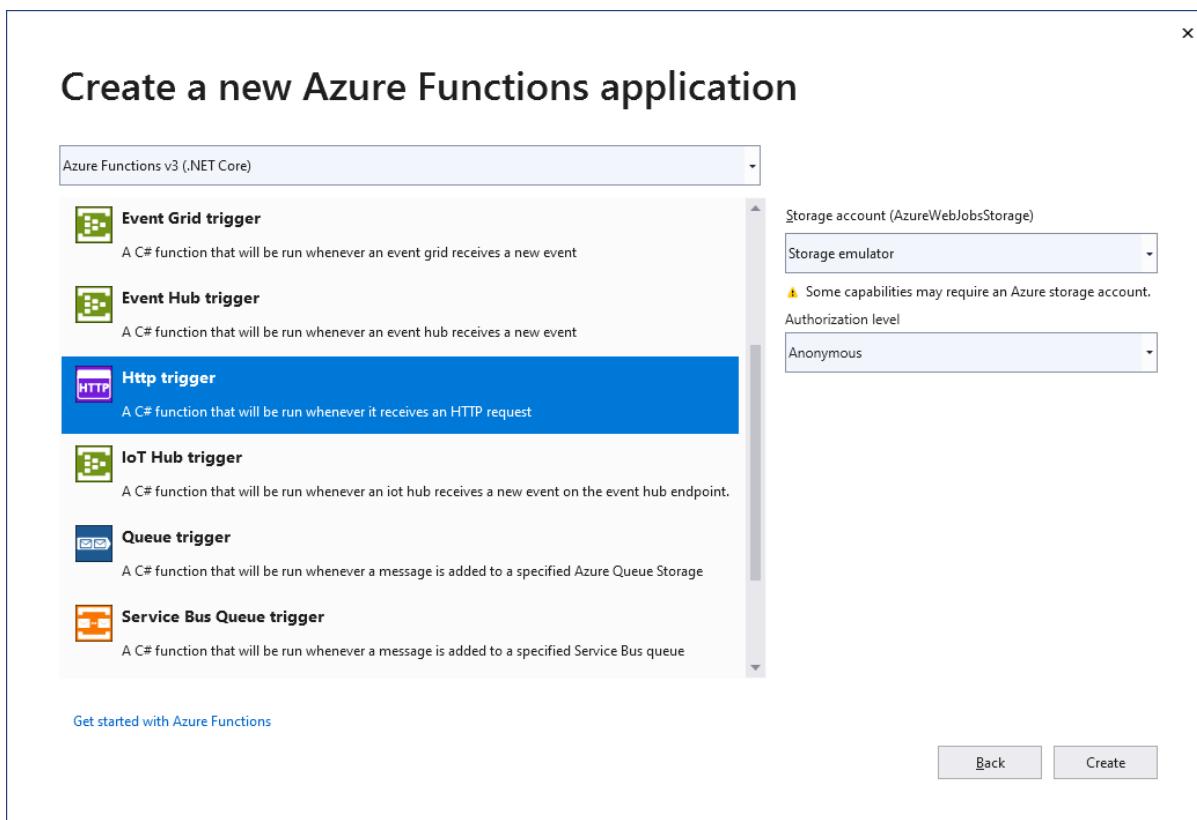
Create a function app project

The Azure Functions project template in Visual Studio creates a 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:

| SETTING | VALUE | DESCRIPTION |
|---------------------------------------|--------------------------------|--|
| Functions runtime | Azure Functions v3 (.NET Core) | This value creates a function project that uses the version 3.x runtime of Azure Functions, which supports .NET Core 3.x. Azure Functions 1.x supports the .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 an Azure Function 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.

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 `FunctionName` 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 `HttpTrigger` method named `Run`, rename the `FunctionName` method attribute to `HttpExample`.

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.

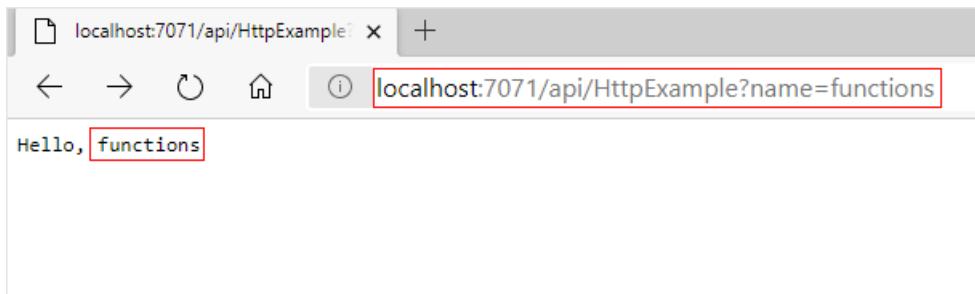
```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.49.0\cli_x64\func.exe
[5/27/2020 7:53:39 AM] Loading functions metadata
[5/27/2020 7:53:39 AM] 1 functions loaded
[5/27/2020 7:53:39 AM] Generating 1 job function(s)
[5/27/2020 7:53:40 AM] Found the following functions:
[5/27/2020 7:53:40 AM] 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. Append the query string `?name=<YOUR_NAME>` to this URL 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.

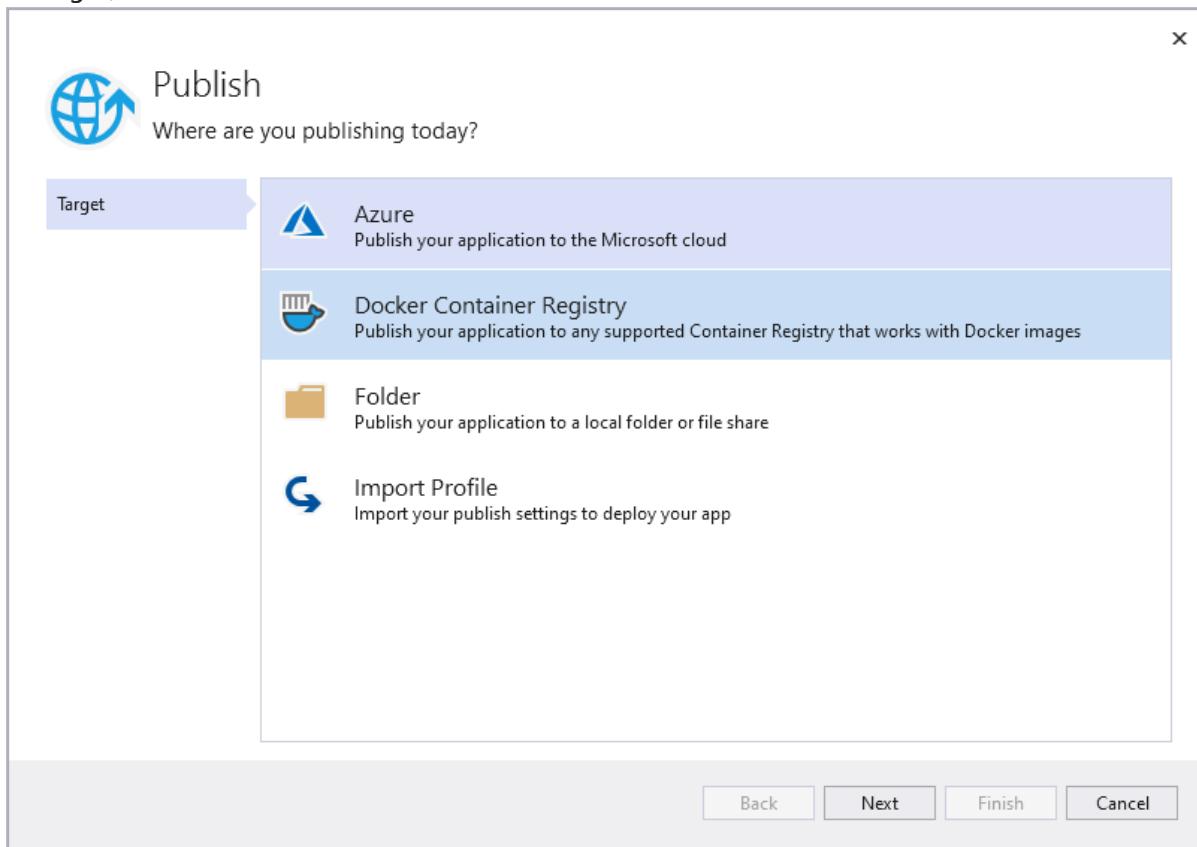
After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Publish the project to Azure

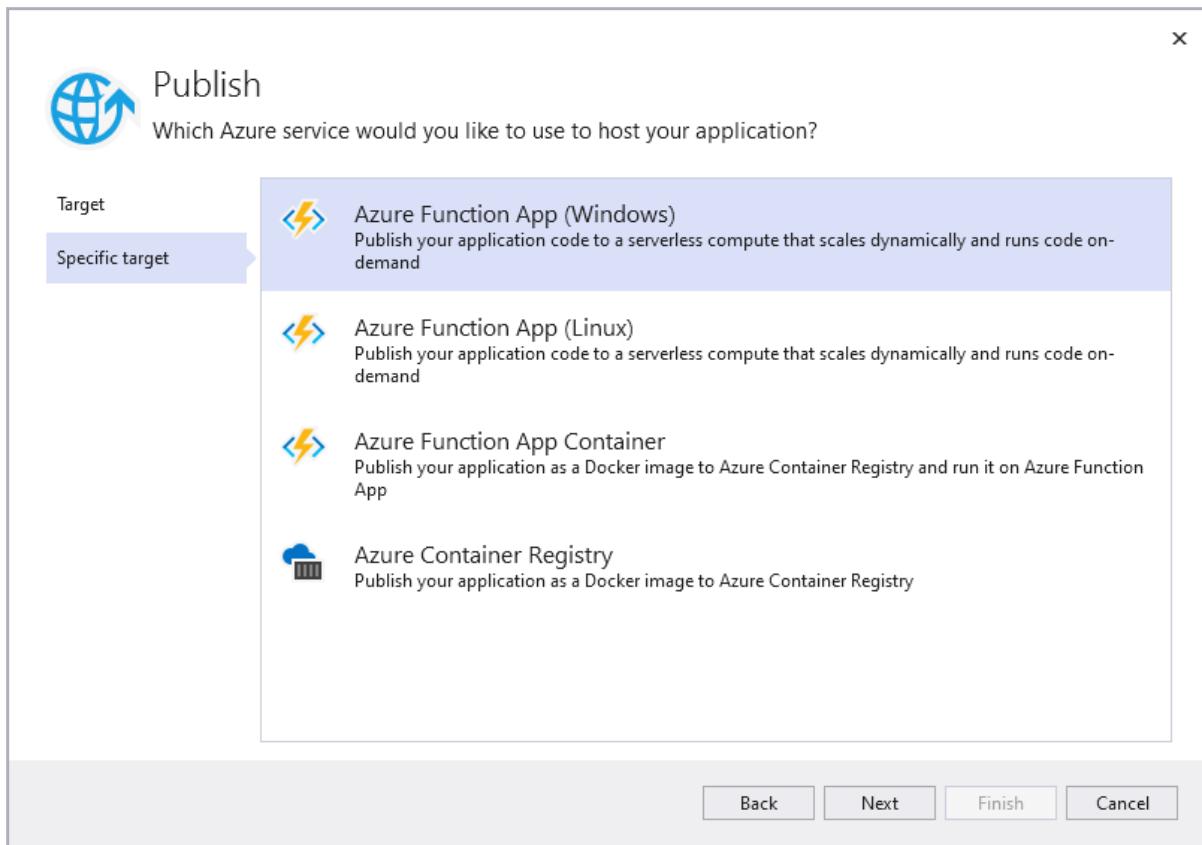
Before you can publish your project, you must have a function app in your Azure subscription. Visual Studio publishing creates a function app for you the first time you publish your project.

1. In **Solution Explorer**, right-click the project and select **Publish**.

2. In **Target**, select **Azure**



3. In **Specific target**, select **Azure Function App (Windows)**



4. In **Function Instance**, select **Create a new Azure Function...** and then use the values specified in the following 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: <code>a-z</code> , <code>0-9</code> , and <code>-</code> . |
| 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 app service | Choose a Location in a region near you or other services your functions access. |

| SETTING | VALUE | DESCRIPTION |
|---------------|---------------------------------|--|
| 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 an existing account that meets the storage account requirements . |

Function App (Windows)

Name: FunctionApp220200930124040

Subscription: Visual Studio Enterprise

Resource group: azure-docs-first-function* [New...](#)

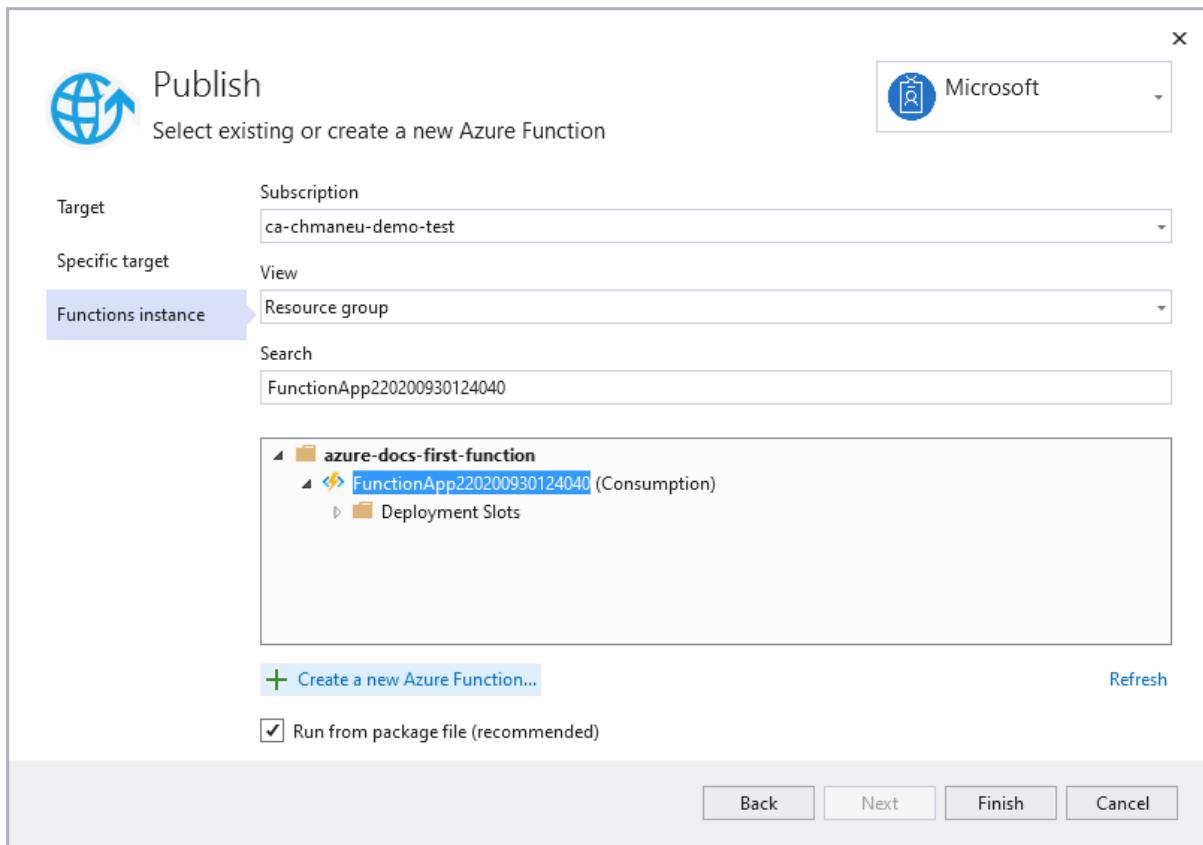
Plan Type: Consumption

Location: France Central

Azure Storage: functiondemochris* (France Central) [New...](#)

Export... Create Cancel

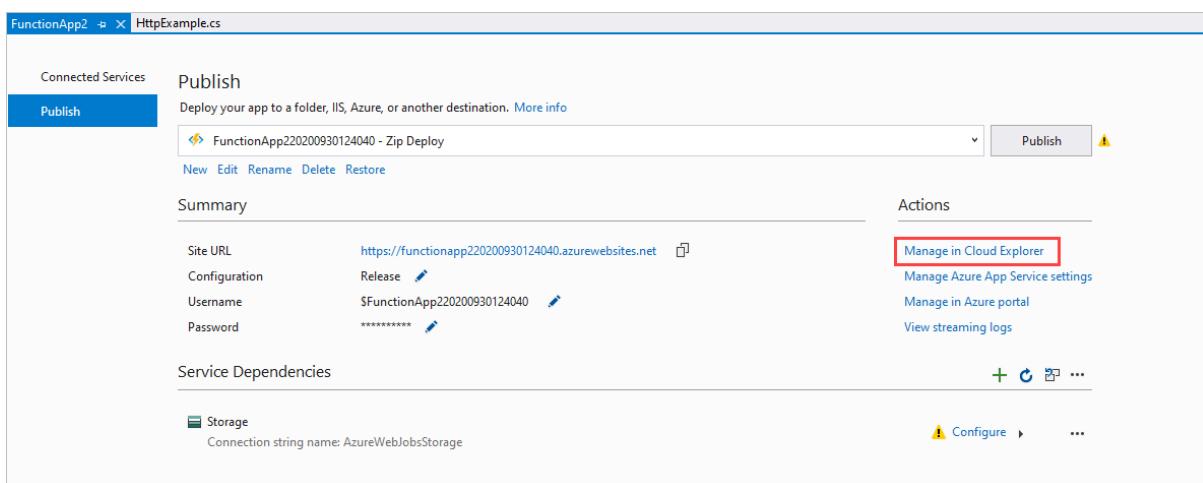
- Select **Create** to create a function app and its related resources in Azure.
- In the **Functions instance**, make sure that **Run from package file** is checked. Your function app is deployed using [Zip Deploy](#) with [Run-From-Package](#) mode enabled. This is the recommended deployment method for your functions project, since it results 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.

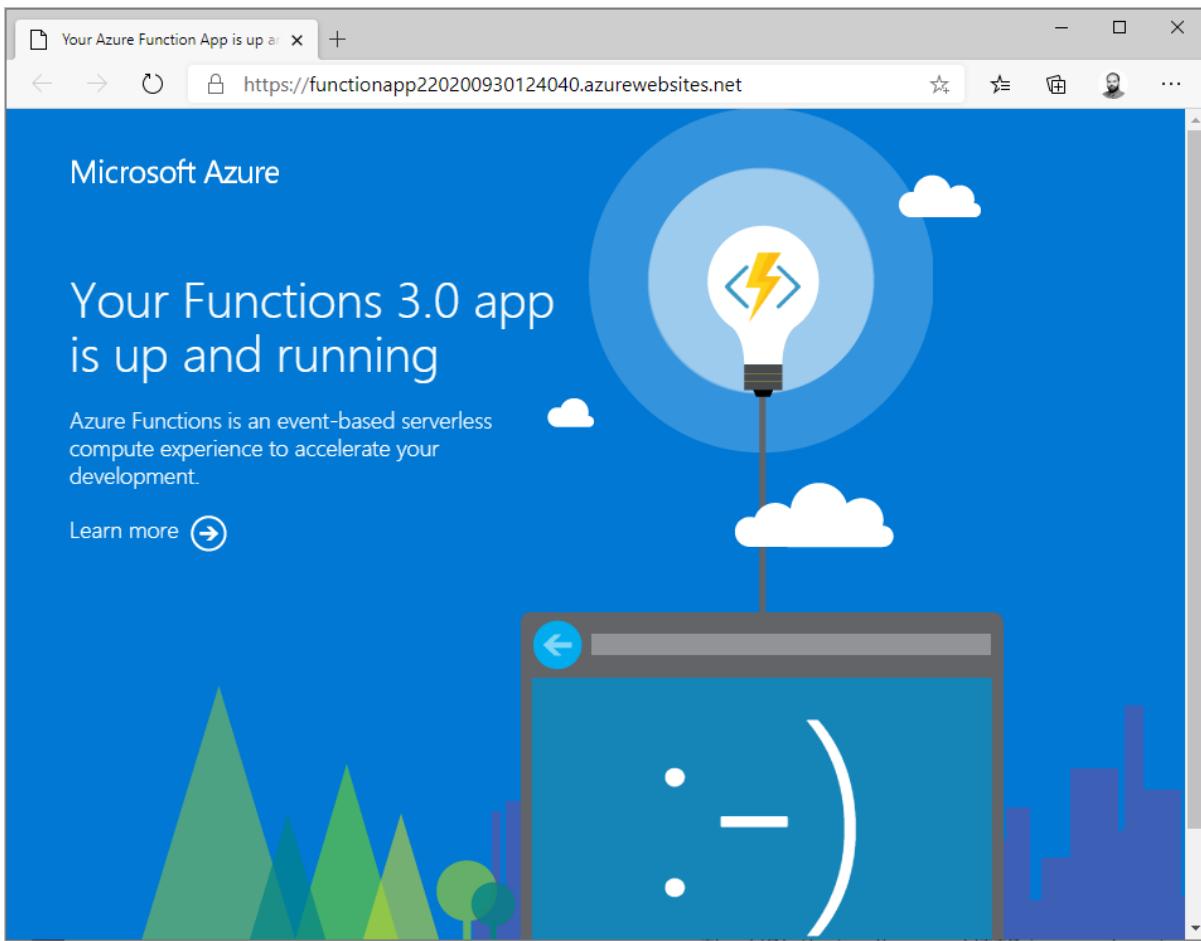
8. In the Publish tab, choose **Manage in Cloud Explorer**. This opens the new function app Azure resource in Cloud Explorer.



Cloud Explorer lets you use Visual Studio to view the contents of the site, start and stop the function app, and browse directly to function app resources on Azure and in the Azure portal.

Test your function in Azure

1. In Cloud Explorer, your new function app should be selected. If not, expand your subscription > **App Services**, and select your new function app.
2. Right-click the function app and choose **Open in Browser**. This opens the root of your function app in your default web browser and displays the page that indicates your function app is running.

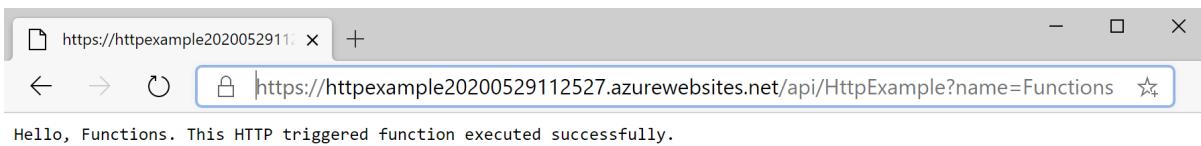


3. In the address bar in the browser, append the string `/api/HttpExample?name=Functions` to the base URL and run the request.

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

```
http://<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

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 have created in this quickstart, do not 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 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 the Cloud Explorer, expand your subscription > **App Services**, right-click your function app, and choose **Open in Portal**.

2. In 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 links for 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 main area has tabs for Overview (highlighted with a red box), Resource group (change) (highlighted with a red box), and Metrics. The Resource group section displays details: 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), and Runtime version (3.0.13139.0).

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.

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](#)

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

12/4/2020 • 7 minutes to read • [Edit Online](#)

In this article, you use command-line tools to create a C# class library-based 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](#).
- The [.NET Core SDK 3.1](#)
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - [Azure PowerShell](#) version 5.0 or later.

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
- [Azure PowerShell](#)
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
- Run `az --version` to check that the Azure CLI version is 2.4 or later.
- Run `az login` to sign in to Azure and verify an active subscription.
- Run `dotnet --list-sdks` to check that .NET Core SDK version 3.1x is installed

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:

```
func init LocalFunctionProj --dotnet
```

2. Navigate into the project folder:

```
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).

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

`func new` creates a `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 is an [HttpRequest](#) that's decorated with the [HttpTriggerAttribute](#), which defines the trigger behavior.

```
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 LocalFunctionProj
{
    public static class HttpExample
    {
        [FunctionName("HttpExample")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "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;

            return name != null
                ? (ActionResult)new OkObjectResult($"Hello, {name}")
                : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
        }
    }
}
```

The return object is an [ActionResult](#) that returns a response message as either an [OkObjectResult](#) (200) or a [BadRequestObjectResult](#) (400). 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:

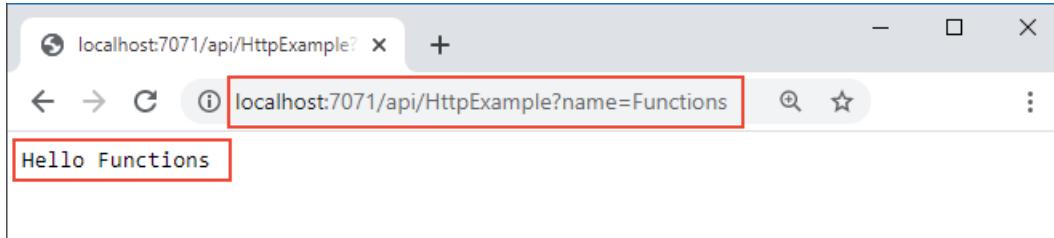
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
...
```

NOTE

If *HttpExample* doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. 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 maintains 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 you haven't done so already, sign in to Azure:

- [Azure CLI](#)

- [Azure PowerShell](#)

```
az login
```

The [az login](#) command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The [az group create](#) command creates a resource group. You generally create your resource group and resources in a region near you, using an available region returned from the [az account list-locations](#) command.

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

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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](#).

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location westeurope --runtime dotnet --functions-version 3 --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.

```
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.

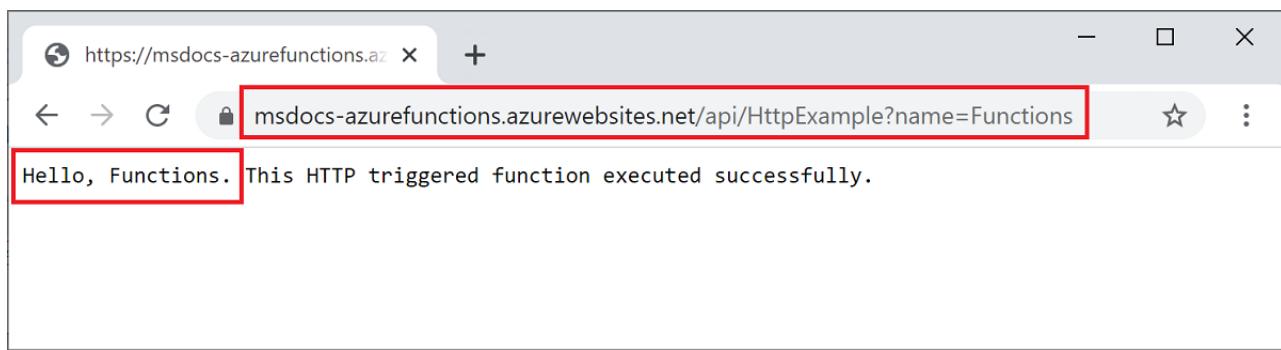
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
    HttpExample - [httpTrigger]
        Invoke url: https://msdocs-azurefunctions-qs.azurewebsites.net/api/httpexample
```

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](#)
- [curl](#)

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](#):

```
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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 7 minutes to read • [Edit Online](#)

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.

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

NOTE

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

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 Functions Core Tools](#) version 3.x..
- 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.

Prerequisite check

- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
- Run `az --version` to check that the Azure CLI version is 2.4 or later.
- Run `az login` to sign in to Azure and verify an active subscription.

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](#).
 - [Bash](#)
 - [PowerShell](#)
 - [Cmd](#)

```
mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure -DarchetypeArtifactId=azure-functions-archetype -DjavaVersion=8
```

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:

```
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.

```

/**
 * 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.FunctionName;
import com.microsoft.azure.functions.annotation.HttpTrigger;

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();
        }
    }
}

```

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.

```

<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>${functionResourceGroup}</resourceGroup>
        <!-- function app service plan name -->
        <appServicePlanName>java-functions-app-service-plan</appServicePlanName>
        <!-- function app region-->
        <!-- refers https://github.com/microsoft/azure-maven-plugins/wiki/Azure-Functions:-Configuration-Details#supported-regions for all valid values -->
        <region>${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></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>~3</value>
            </property>
        </appSettings>
    </configuration>
    <executions>
        <execution>
            <id>package-functions</id>
            <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:

```

mvn clean package
mvn azure-functions:run

```

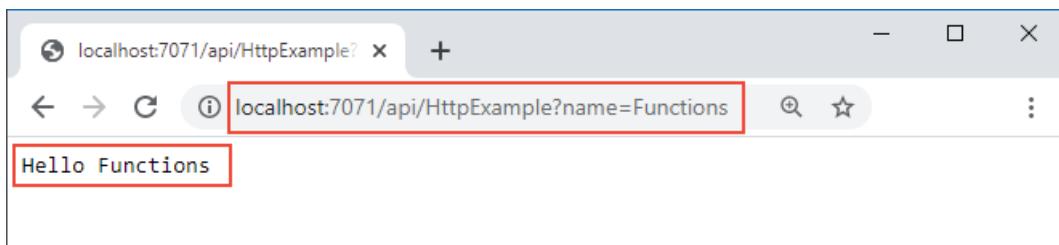
Toward the end of the output, the following lines should appear:

```
...  
  
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 below, 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 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.

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

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

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

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

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.

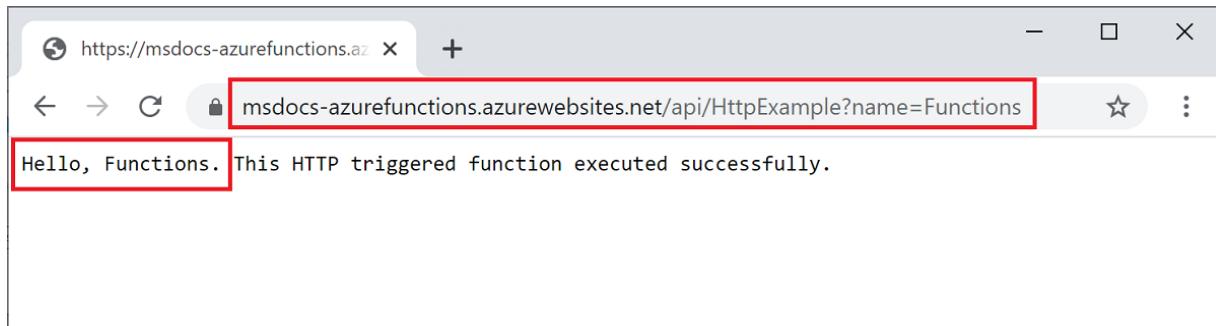
The deployment packages the project files and deploys them to the new function app using [zip deployment](#). The code runs from the deployment package in Azure.

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](#)
- [curl](#)

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](#):

```
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 PowerShell

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

Next steps

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 8 minutes to read • [Edit Online](#)

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.

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](#).
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - [Azure PowerShell](#) version 5.0 or later.
- [Node.js](#) version 12. Node.js version 10 is also supported.

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
 - [Azure PowerShell](#)
-
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
 - Run `az --version` to check that the Azure CLI version is 2.4 or later.
 - Run `az login` to sign in to Azure and verify an active subscription.

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:

```
func init LocalFunctionProj --javascript
```

2. Navigate into the project folder:

```
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).

```
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.

index.js

index.js exports a function that's triggered according to the configuration in *function.json*.

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    const name = (req.query.name || (req.body && req.body.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.";

    context.res = {
        // status: 200, /* Defaults to 200 */
        body: responseMessage
    };
}
```

For an HTTP trigger, the function receives request data in the variable `req` as defined in *function.json*. The return object, defined as `$return` in *function.json*, is the response. To learn more, see [Azure Functions HTTP triggers and bindings](#).

function.json

function.json is a configuration file that defines the input and output `bindings` for the function, including the trigger type.

```
{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```

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:

```
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.

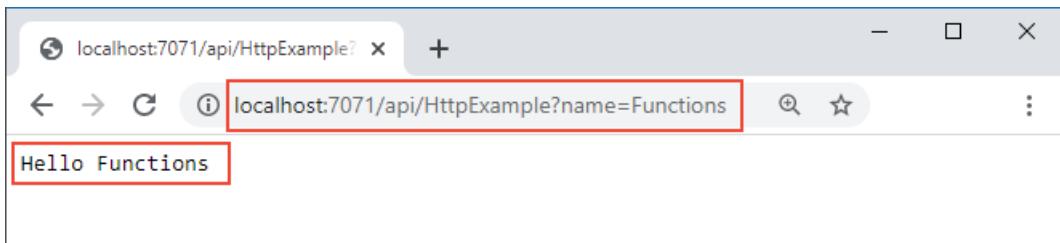
Http Functions:

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

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. 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 maintains 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 you haven't done so already, sign in to Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

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

2. Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The `az group create` command creates a resource group. You generally create your resource group and resources in a region near you, using an available region returned from the `az account list-locations` command.

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

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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](#).

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location westeurope --runtime node --runtime-version 12 --functions-version 3 --name <APP_NAME> --storage-account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure. If you're using Node.js 10, also change `--runtime-version` to `10`.

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.

```
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
```

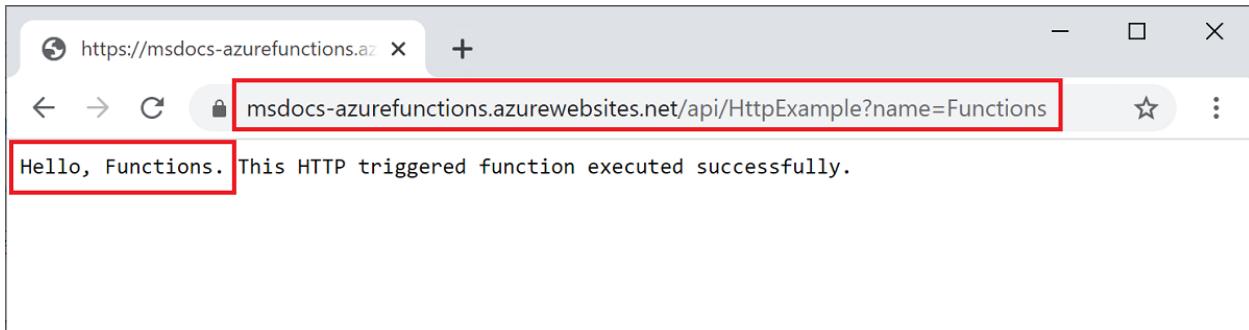
Invoke the function on Azure

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](#)

- [curl](#)

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](#):

```
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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 8 minutes to read • [Edit Online](#)

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](#).
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure PowerShell](#) version 5.0 or later.
 - [Azure CLI](#) version 2.4 or later.
- The [.NET Core SDK 3.1](#)

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
 - [Azure PowerShell](#)
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
 - Run `az --version` to check that the Azure CLI version is 2.4 or later.
 - Run `az login` to sign in to Azure and verify an active subscription.

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:

```
func init LocalFunctionProj --powershell
```

2. Navigate into the project folder:

```
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).

```
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*.

```
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.

```
{
  "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:

```
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.

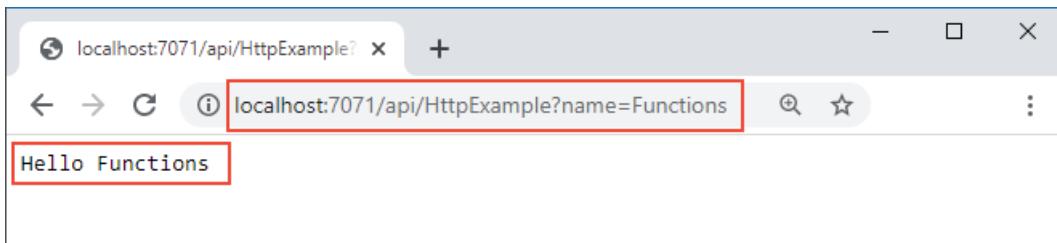
Http Functions:

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

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. 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 maintains 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 you haven't done so already, sign in to Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

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

2. Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The `az group create` command creates a resource group. You generally create your resource group and resources in a region near you, using an available region returned from the `az account list-locations` command.

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

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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](#).

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location  
westeurope --runtime powershell --functions-version 3 --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.

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

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

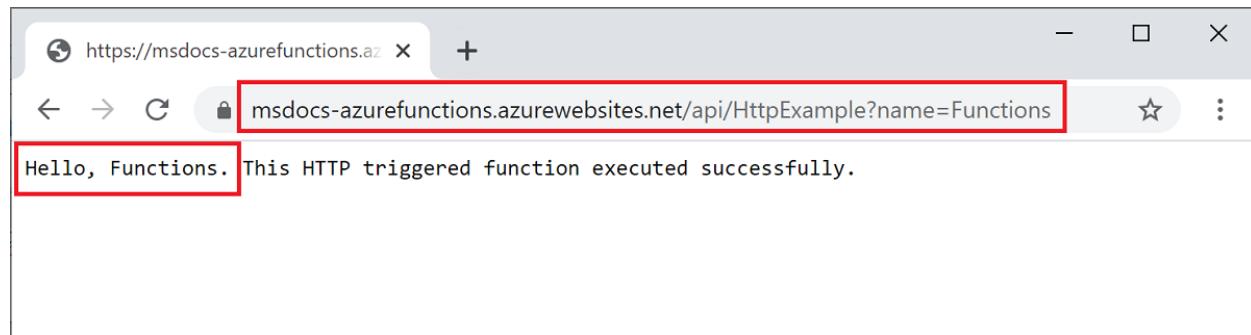
```
...  
  
Getting site publishing info...  
Creating archive for current directory...  
Performing remote build for functions project.  
  
...  
  
Deployment successful.  
Remote build succeeded!  
Syncing triggers...  
Functions in msdocs-azurefunctions-qs:  
  HttpExample - [httpTrigger]  
    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](#)
- [curl](#)

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](#):

```
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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 9 minutes to read • [Edit Online](#)

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.

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](#).
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - [Azure PowerShell](#) version 5.0 or later.
- [Python 3.8 \(64-bit\)](#), [Python 3.7 \(64-bit\)](#), [Python 3.6 \(64-bit\)](#), which are all supported by version 3.x of Azure Functions.

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
- [Azure PowerShell](#)
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
- Run `az --version` to check that the Azure CLI version is 2.4 or later.
- Run `az login` to sign in to Azure and verify an active subscription.
- Run `python --version` (Linux/macOS) or `py --version` (Windows) to check your Python version reports 3.8.x, 3.7.x or 3.6.x.

Create and activate a virtual environment

In a suitable folder, run the following commands to create and activate a virtual environment named `.venv`. Be sure to use Python 3.8, 3.7 or 3.6, which are supported by Azure Functions.

- [bash](#)
- [PowerShell](#)
- [Cmd](#)

```
python -m venv .venv
```

```
source .venv/bin/activate
```

If Python didn't install the venv package on your Linux distribution, run the following command:

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment.

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:

```
func init LocalFunctionProj --python
```

2. Navigate into the project folder:

```
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).

```
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.

`__init__.py`

`__init__.py` contains a `main()` Python function that's triggered according to the configuration in `function.json`.

```

import logging

import azure.functions as func


def main(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
        )

```

For an HTTP trigger, the function receives request data in the variable `req` as defined in `function.json`. `req` is an instance of the [azure.functions.HttpRequest class](#). The return object, defined as `$return` in `function.json`, is an instance of [azure.functions.HttpResponse class](#). To learn more, see [Azure Functions HTTP triggers and bindings](#).

function.json

`function.json` is a configuration file that defines the input and output `bindings` for the function, including the trigger type.

You can change `scriptFile` to invoke a different Python file if desired.

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    }
  ]
}
```

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:

```
func start
```

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

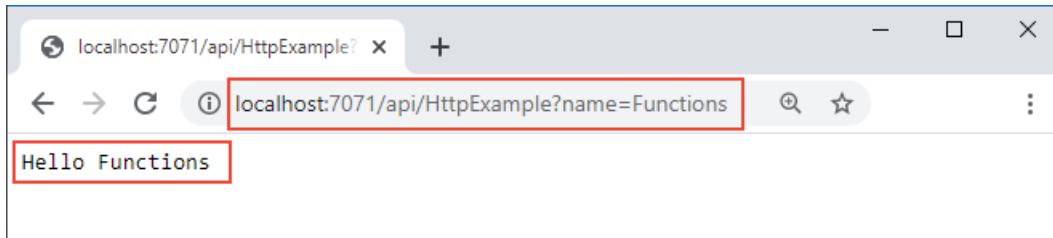
```
...  
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.  
  
Http Functions:  
  
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample  
...
```

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. 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 maintains 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 you haven't done so already, sign in to Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

The [az login](#) command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region.

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The [az group create](#) command creates a resource group. You generally create your resource group and resources in a region near you, using an available region 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 PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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 storage account incurs only a few cents (USD) for this quickstart.

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location  
westeurope --runtime python --runtime-version 3.8 --functions-version 3 --name <APP_NAME> --storage-  
account <STORAGE_NAME> --os-type linux
```

The [az functionapp create](#) command creates the function app in Azure. If you are using Python 3.7 or 3.6, change `--runtime-version` to `3.7` or `3.6`, respectively.

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.

```
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.

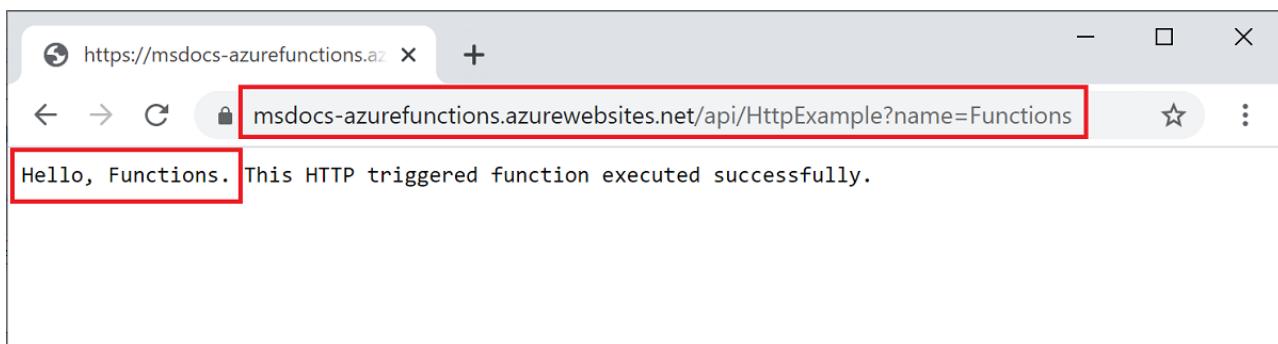
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
    HttpExample - [httpTrigger]
        Invoke url: https://msdocs-azurefunctions-qs.azurewebsites.net/api/httpexample
```

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](#)
- [curl](#)

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](#) in Application Insights in the Azure portal:

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

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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

[Having issues? Let us know.](#)

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

12/4/2020 • 8 minutes to read • [Edit Online](#)

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.

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](#).
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - [Azure PowerShell](#) version 5.0 or later.
- [Node.js](#), Active LTS and Maintenance LTS versions (8.11.1 and 10.14.1 recommended).

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
 - [Azure PowerShell](#)
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
 - Run `az --version` to check that the Azure CLI version is 2.4 or later.
 - Run `az login` to sign in to Azure and verify an active subscription.

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:

```
func init LocalFunctionProj --typescript
```

2. Navigate into the project folder:

```
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).

```
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.

index.ts

index.ts exports a function that's triggered according to the configuration in *function.json*.

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.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.";

    context.res = {
        // status: 200, /* Defaults to 200 */
        body: responseMessage
    };
}

export default httpTrigger;
```

For an HTTP trigger, the function receives request data in the variable `req` of type `HttpRequest` as defined in *function.json*. The return object, defined as `$return` in *function.json*, is the response.

function.json

function.json is a configuration file that defines the input and output `bindings` for the function, including the trigger type.

```
{  
  "bindings": [  
    {  
      "authLevel": "function",  
      "type": "httpTrigger",  
      "direction": "in",  
      "name": "req",  
      "methods": [  
        "get",  
        "post"  
      ]  
    },  
    {  
      "type": "http",  
      "direction": "out",  
      "name": "res"  
    }  
  ]  
}
```

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:

```
npm install  
npm start
```

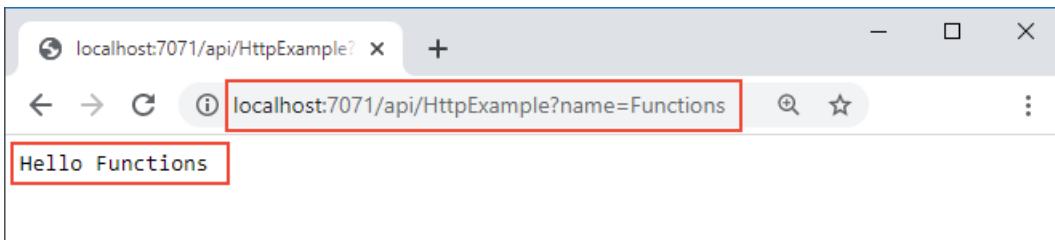
Toward the end of the output, the following lines should appear:

```
...  
  
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 below, 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.

- 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 maintains 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.

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

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

The [az login](#) command signs you into your Azure account.

- Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The [az group create](#) command creates a resource group. You generally create your resource group and resources in a region near you, using an available region returned from the [az account list-locations](#) command.

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

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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](#).

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location westeurope --runtime node --runtime-version 12 --functions-version 3 --name <APP_NAME> --storage-account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure. If you're using Node.js 10, also change `--runtime-version` to `10`.

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

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:

```
npm run build:production
```

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 `func azure functionapp publish` command. In the following example, replace `<APP_NAME>` with the name of your app.

```
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.

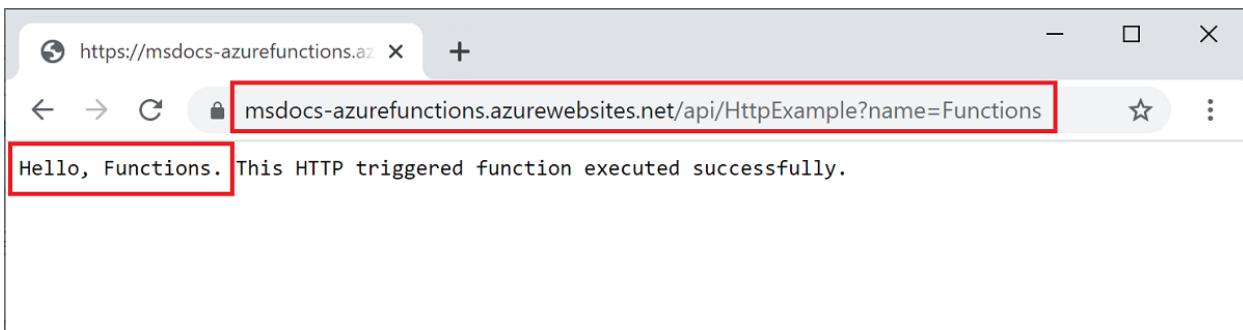
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-qs.azurewebsites.net/api/httpexample?
code=KYHrydo4GFe9y0000000qRgRJ8NdLFKpkakGJQfcC3izYVidzzDN4gQ==
```

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](#)
- [curl](#)

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](#):

```
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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

Quickstart: Create and deploy Azure Functions resources from an ARM template

12/4/2020 • 5 minutes to read • [Edit Online](#)

In this article, you use an Azure Resource Manager template (ARM template) to create a function that responds to HTTP requests.

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

An [ARM 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.



Prerequisites

Azure account

Before you begin, you must have an Azure account with an active subscription. [Create an account for free](#).

Create a local functions project

This article requires a local functions code project to run on the Azure resources that you create. If you don't first create a project to publish, you won't be able to complete the deployment section of this article.

Choose one of the following tabs, follow the link, and complete the section to create a function app in the language of your choice:

- [Visual Studio Code](#)
- [Visual Studio](#)
- [Command line](#)

Create your local functions project in your chosen language in Visual Studio Code:

- [C#](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)
- [TypeScript](#)

After you've created your project locally, you create the resources required to run your new function in Azure.

Review the template

The template used in this quickstart is from [Azure Quickstart Templates](#).

```
{
```

```
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
```

```
"contentVersion": "1.0.0.0",
"parameters": {
    "appName": {
        "type": "string",
        "defaultValue": "[concat('fnapp', 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": "The language worker runtime to load in the function app."
        }
    }
},
"variables": {
    "functionAppName": "[parameters('appName')]",
    "hostingPlanName": "[parameters('appName')]",
    "applicationInsightsName": "[parameters('appName')]",
    "storageAccountName": "[concat(uniquestring(resourceGroup().id), 'azfunctions')]",
    "functionWorkerRuntime": "[parameters('runtime')]"
},
"resources": [
    {
        "type": "Microsoft.Storage/storageAccounts",
        "apiVersion": "2019-06-01",
        "name": "[variables('storageAccountName')]",
        "location": "[parameters('location')]",
        "sku": {
            "name": "[parameters('storageAccountType')]"
        },
        "kind": "Storage"
    },
    {
        "type": "Microsoft.Web/serverfarms",
        "apiVersion": "2020-06-01",
        "name": "[variables('hostingPlanName')]"
    }
]
```

```

    "name": "[variables('hostingPlanName')]",
    "location": "[parameters('location')]",
    "sku": {
        "name": "Y1",
        "tier": "Dynamic"
    },
    "properties": {
        "name": "[variables('hostingPlanName')]",
        "computeMode": "Dynamic"
    }
},
{
    "type": "Microsoft.Web/sites",
    "apiVersion": "2020-06-01",
    "name": "[variables('functionAppName')]",
    "location": "[parameters('location')]",
    "kind": "functionapp",
    "dependsOn": [
        "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
    ],
    "properties": {
        "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
        "siteConfig": {
            "appSettings": [
                {
                    "name": "AzureWebJobsStorage",
                    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'), ';EndpointSuffix=', environment().suffixes.storage, ';AccountKey=', listKeys(resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName')), '2019-06-01').keys[0].value)]"
                },
                {
                    "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",
                    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'), ';EndpointSuffix=', environment().suffixes.storage, ';AccountKey=', listKeys(resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName')), '2019-06-01').keys[0].value)]"
                },
                {
                    "name": "WEBSITE_CONTENTSHARE",
                    "value": "[toLower(variables('functionAppName'))]"
                },
                {
                    "name": "FUNCTIONS_EXTENSION_VERSION",
                    "value": "~2"
                },
                {
                    "name": "WEBSITE_NODE_DEFAULT_VERSION",
                    "value": "~10"
                },
                {
                    "name": "APPINSIGHTS_INSTRUMENTATIONKEY",
                    "value": "[reference(resourceId('microsoft.insights/components', variables('applicationInsightsName')), '2020-02-02-preview').InstrumentationKey]"
                },
                {
                    "name": "FUNCTIONS_WORKER_RUNTIME",
                    "value": "[variables('functionWorkerRuntime')]"
                }
            ]
        }
    },
    {
        "type": "microsoft.insights/components",
        "apiVersion": "2020-02-02-preview",
        "name": "[variables('applicationInsightsName')]",
        "location": "[parameters('appInsightsLocation')]",
        "tags": [

```

```

    "tags": 1
    "[concat('hidden-link:', resourceId('Microsoft.Web/sites', variables('applicationInsightsName')))]":
"Resource"
},
"properties": {
    "ApplicationId": "[variables('applicationInsightsName')]",
    "Request_Source": "IbizaWebAppExtensionCreate"
}
}
]
}

```

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.

Deploy the template

- [Azure CLI](#)
- [PowerShell](#)

```

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/101-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

```

Validate the deployment

Next you validate the function app hosting resources you created by publishing your project to Azure and calling the HTTP endpoint of the function.

Publish the function project to Azure

Use the following steps to publish your project to the new Azure resources:

- [Visual Studio Code](#)
 - [Visual Studio](#)
 - [Command line](#)
1. In Visual Studio Code, select F1 to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app**.
 2. If you're not signed in, you're prompted to **Sign in to Azure**. After you sign in from the browser, go back to Visual Studio Code. If you have multiple subscriptions, **Select a subscription** that contains your function app.
 3. Select your existing function app in Azure. When you're warned about overwriting all files in the function app, select **Deploy** to acknowledge the warning and continue.

The project is rebuilt, repackaged, and uploaded to Azure. The existing project is replaced by the new package, and the function app restarts.

In the output, copy the URL of the HTTP trigger. You use this to test your function running in Azure.

Invoke the function on Azure

Paste the URL you copied for the HTTP request into your browser's address bar, make sure that the `name` query string as `?name=Functions` has been appended to the end of this URL, and then execute the request.

You should see a response like:

```
Hello Functions!
```

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.

```
az group delete --name <RESOURCE_GROUP_NAME>
```

Replace `<RESOURCE_GROUP_NAME>` with the name of your resource group.

Next steps

Now that you've publish your first function, learn more by adding an output binding to your function.

- [Visual Studio Code](#)
- [Visual Studio](#)
- [Command line](#)

[Connect to an Azure Storage queue](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

12/4/2020 • 16 minutes to read • [Edit Online](#)

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 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 start this article, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool 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](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- 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 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 the F1 key to open the command palette, then search for and run the command

- Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

IMPORTANT

Because it contains secrets, the local.settings.json file never gets published, and is excluded from source control.

- 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 usage is enabled in the host.json file at the root of the project, which appears as follows:

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage
```

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 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.json file. 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:

| 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. |

| PROMPT | VALUE | DESCRIPTION |
|---|---------------------|--|
| The name used to identify this binding in your code | msg | Name that identifies the binding parameter referenced in your code. |
| The queue to which the message will be sent | outqueue | 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" | AzureWebJobsStorage | 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:

```
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
```

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};
```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```

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
})

```

Update `HttpExample__init__.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```

import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )

```

The `msg` parameter is an instance of the [azure.functions.InputStream class](#). Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```
if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}
```

At this point, your function should look as follows:

```
[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}
```

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.

```
// Write the name to the message queue.
```

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:

```

@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.

```

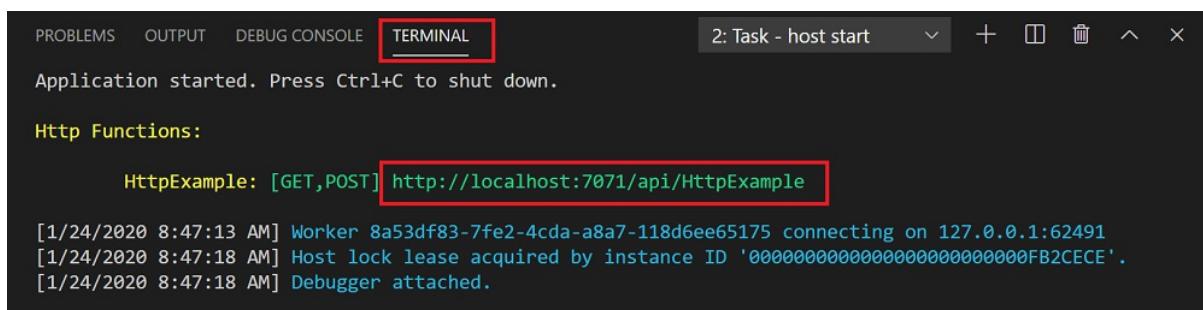
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);

```

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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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
2: Task - host start
Application started. Press Ctrl+C to shut down.

Http Functions:

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

[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '0000000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.

```

3. 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
```

4. A response is returned, which looks like the following in a browser:



5. 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 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] }
```

The screenshot shows the Visual Studio Code interface with the "TERMINAL" tab selected. The terminal window displays a log of events from an Azure Function. It starts with an HTTP request being executed, followed by the function being triggered via a host API. The terminal then shows the execution of the JavaScript HTTP trigger function, which processes a request and executes an HTTP request back to itself. The log concludes with a status code of 200 and a duration of 39 milliseconds.

6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

Run the function locally

Azure Functions Core Tools integrates with Visual Studio Code to let you run and debug an Azure Functions project locally. For details on how to debug in Visual Studio Code, see [Debug PowerShell Azure Functions locally](#).

1. Press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The title bar says "TERMINAL". The terminal window displays the following text:

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

Http Functions:

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

[4/20/2019 6:19:06 AM] System Log: {
[4/20/2019 6:19:06 AM]   Log-Message: The enforced concurrency level (pool size limit) is '1'.
[4/20/2019 6:19:06 AM] }
[4/20/2019 6:19:06 AM] System Log: {
```

Below the terminal window, the status bar shows "unctions (MyFunctionProj)" and "Ln 10, Col 28 (27 selected)".

- Append the query string `?name=<yourname>` to this URL, and then use `Invoke-RestMethod` in a second PowerShell command prompt to execute the request, as follows:

```
PS > Invoke-RestMethod -Method Get -Uri http://localhost:7071/api/HttpTrigger?name=PowerShell
Hello PowerShell
```

You can also execute the GET request from a browser from the following URL:

```
http://localhost:7071/api/HttpExample?name=PowerShell
```

When you call the `HttpTrigger` endpoint without passing a `name` parameter either as a query parameter or in the body, the function returns a `BadRequest` error. When you review the code in `run.ps1`, you see that this error occurs by design.

- Information about the request is shown in `Terminal` panel.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The title bar says "TERMINAL". The terminal window displays the following detailed log output:

```
[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 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] }
```

- When done, press **Ctrl + C** to stop Core Tools.

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

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.

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.

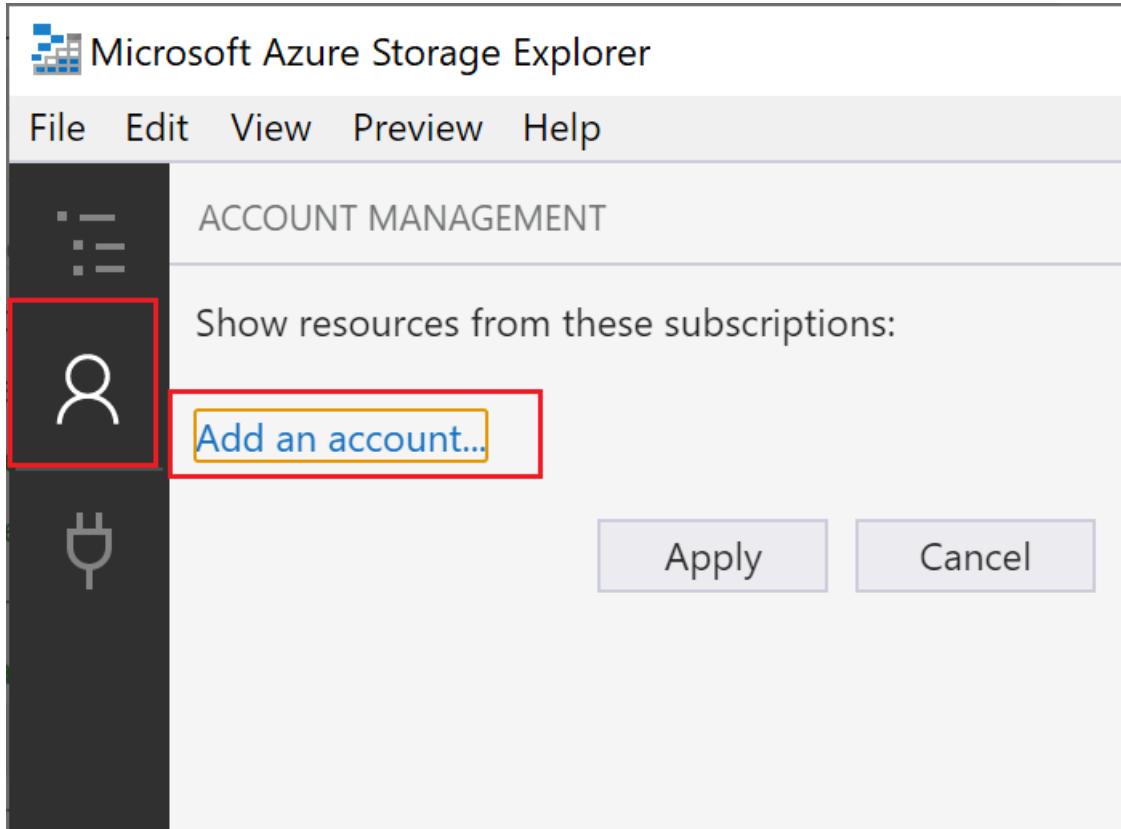
Browse to the location of your test code under `src/test/java`, open the `Function.java` project file, and replace the line of code under `//Invoke` with the following code.

```
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);
```

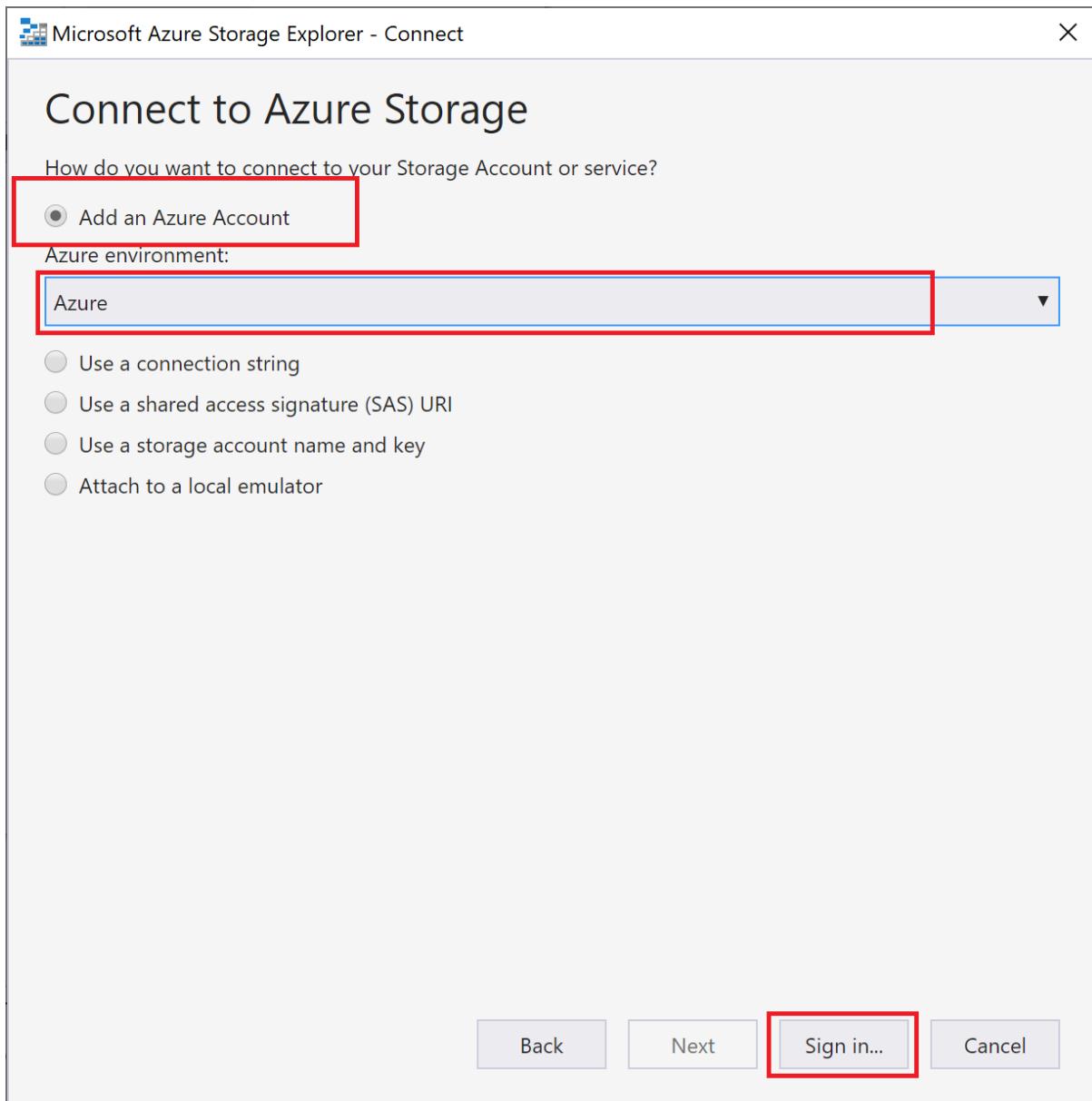
Connect Storage Explorer to your account

Skip this section if you have 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 select **Sign in....**

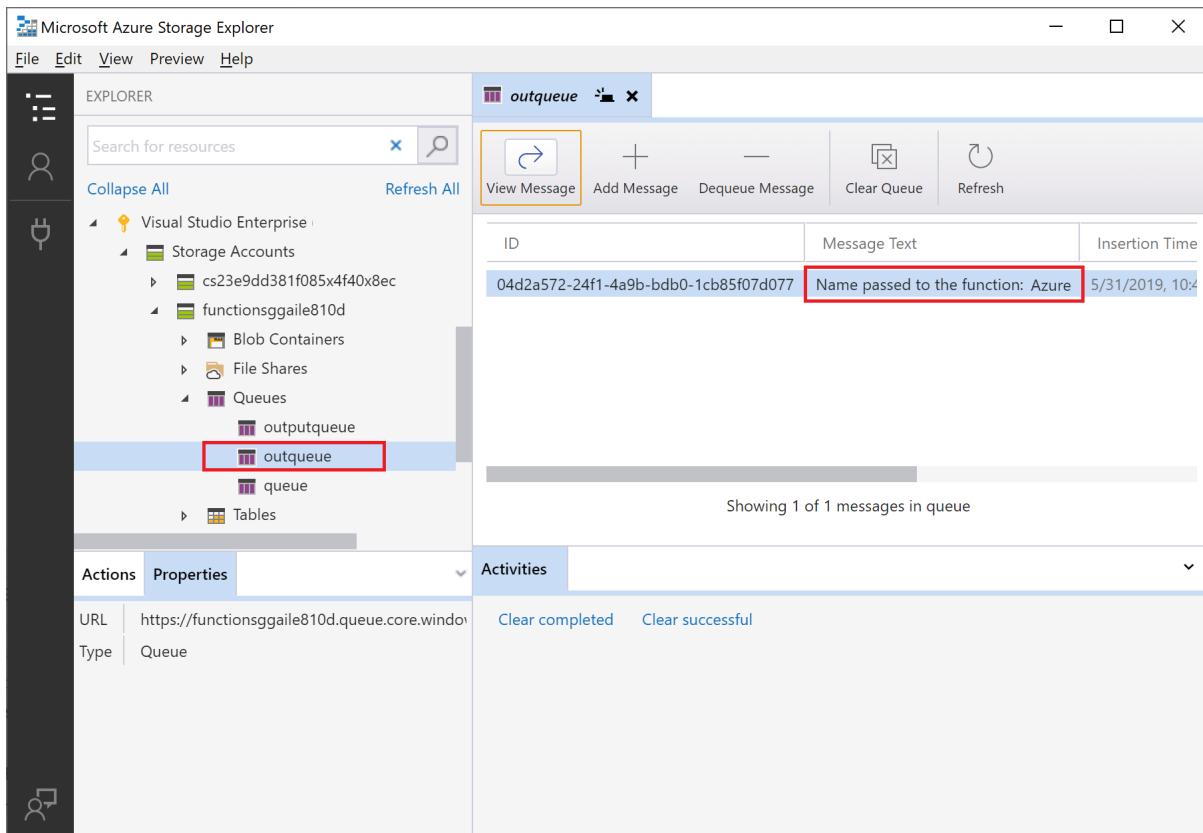


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account.

Examine the output queue

1. In Visual Studio Code, press the F1 key 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 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'll see a new message appear 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 deployment completes, you can again use cURL or a browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL, as in the following example:

```
curl https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....&name=<yourname>
```

- 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

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:

- In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Open in portal`.
- 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'. On the left, there's a sidebar with links like 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 panel has a title bar with 'myfunctionapp' and 'App Service'. Below the title bar are buttons for Browse, Refresh, Stop, Restart, Swap, Get publish profile, Reset publish profile, and Delete. The 'Overview' tab is selected. A red box highlights the 'Resource group (change)' field, which contains 'myResourceGroup'. To the right of this field are details: 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), and Runtime version (3.0.13139.0). At the bottom of the main panel are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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](#)
- [Examples of complete Function projects in JavaScript.](#)
- [Azure Functions JavaScript developer guide](#)
- [Examples of complete Function projects in Java.](#)
- [Azure Functions Java developer guide](#)
- [Examples of complete Function projects in TypeScript.](#)
- [Azure Functions TypeScript developer guide](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)
- [Examples of complete Function projects in PowerShell.](#)
- [Azure Functions PowerShell developer guide](#)

Connect functions to Azure Storage using Visual Studio

12/4/2020 • 6 minutes to read • [Edit Online](#)

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

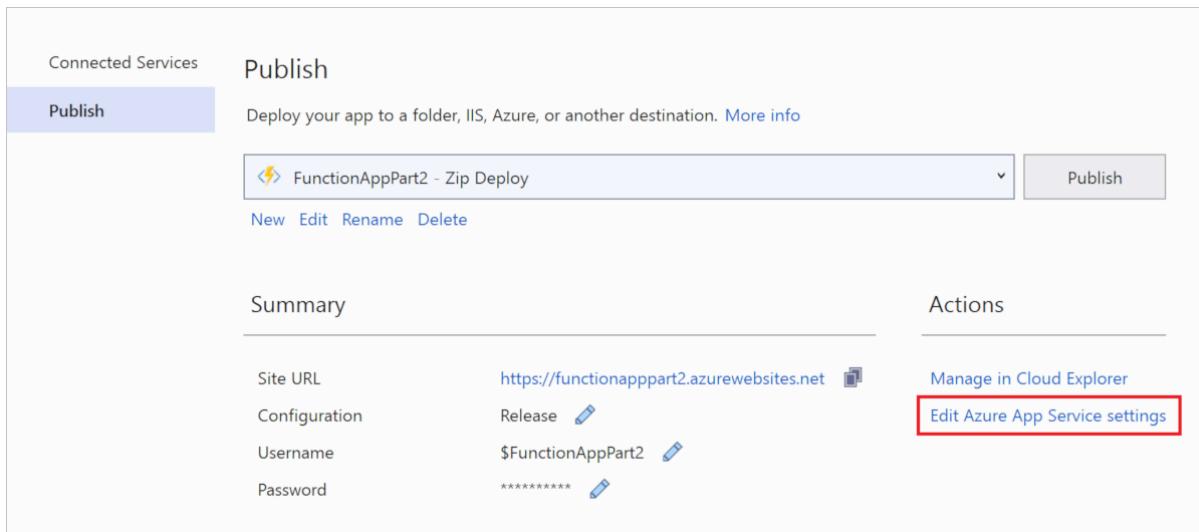
Before you start this article, you must:

- Complete [part 1 of the Visual Studio quickstart](#).
- 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. Under **Actions**, select **Edit 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:

```
Install-Package Microsoft.Azure.WebJobs.Extensions.Storage -Version 3.0.6
```

Now, you can add the storage 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. 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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The Run method definition should now look like the following:

```
[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)
```

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 `msg` output binding object to create a queue message. Add this code before the method returns.

```

if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}

```

At this point, your function should look as follows:

```

[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

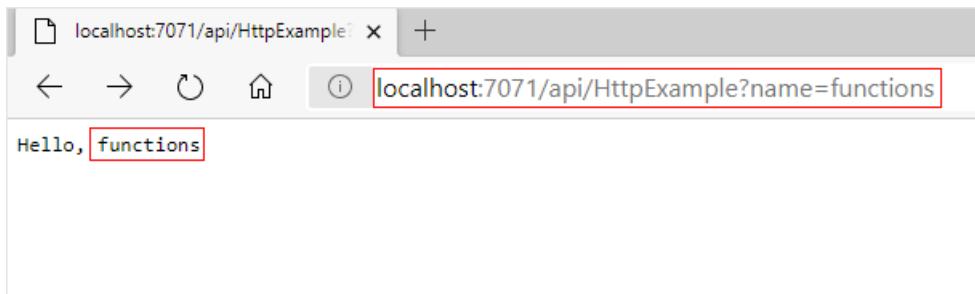
Http Functions:

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

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

```

3. Paste the URL for the HTTP request into your browser's address bar. Append the query string `?name=<YOUR_NAME>` to this URL 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.

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 Cloud Explorer to verify that the queue was created along with the new message.

Examine the output queue

1. In Visual Studio from the **View** menu, select **Cloud Explorer**.
2. In **Cloud Explorer**, expand your Azure subscription and **Storage Accounts**, then expand the storage account used by your function. If you can't remember the storage account name, check the `AzureWebJobsStorage` connection string setting in the `local.settings.json` file.
3. Expand the **Queues** node, and then double-click the queue named **outqueue** to view the contents of the queue in Visual Studio.

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*.

| ID | Message Text | Insertion Time |
|--------------------------------------|------------------------------------|---------------------|
| 04d2a572-24f1-4a9b-bdb0-1cb85f07d077 | Name passed to the function: Azure | 5/31/2019, 10:42:23 |

4. Run the function again, send another request, and you'll see a new message appear in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

1. In **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
2. After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourusername>` to the URL.
3. 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

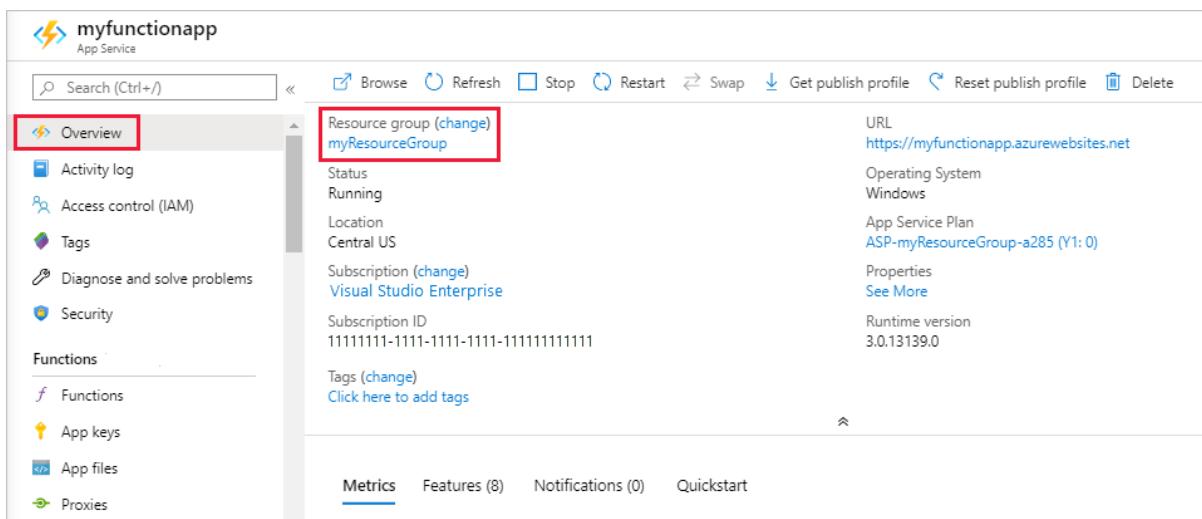
Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you have created in this quickstart, do not clean up the resources.

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 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 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 "Resource group (change)" section with a red box around it, containing the text "myResourceGroup". The main content area displays various details about the function app, such as Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), App Service Plan (ASP-myResourceGroup-a285), 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 quickstart.

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 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. 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

12/4/2020 • 16 minutes to read • [Edit Online](#)

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.

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.

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.

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.

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.

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 use by the function app. 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 that connection write to a Storage queue in the same account when running the function locally.

- From the root of the project, run the following command, replacing `<app_name>` with the name of your function app from the previous quickstart. This command will overwrite any existing values in the file.

```
func azure functionapp fetch-app-settings <app_name>
```

- Open `local.settings.json` 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 `local.settings.json` 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

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage --version 3.0.4
```

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 let 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:

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "res"
  }
]
```

```
"scriptFile": "__init__.py",
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  }
]
```

```

"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "Response"
  }
]

```

Each binding has at least a type, a direction, and a name. In the example above, the first binding is of type `httpTrigger` with the direction `in`. For the `in` direction, `name` specifies the name of an input parameter that's sent to the function when invoked by the trigger.

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:

```

{
  "authLevel": "function",
  "type": "httpTrigger",
  "direction": "in",
  "name": "req",
  "methods": [
    "get",
    "post"
  ]
},
{
  "type": "http",
  "direction": "out",
  "name": "res"
},
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
]
}

```

The second binding in the collection is of type `http` with the direction `out`, in which case the special `name` of `$return` indicates that this binding uses the function's return value rather than providing an input parameter.

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:

```

"bindings": [
  {
    "authLevel": "anonymous",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  },
  {
    "type": "queue",
    "direction": "out",
    "name": "msg",
    "queueName": "outqueue",
    "connection": "AzureWebJobsStorage"
  }
]

```

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:

```

{
  "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"
  }
]
}

```

In this case, `msg` is given to the function as an output argument. For a `queue` type, you must also specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from *local.settings.json*) in `connection`.

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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.

Update `HttpExample_init_.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```

import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )

```

The `msg` parameter is an instance of the `azure.functions.InputStream class`. Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```

module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```

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
})

```

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```

if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}

```

At this point, your function should look as follows:

```

[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

```
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:

```

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.

```
@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:

```
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:

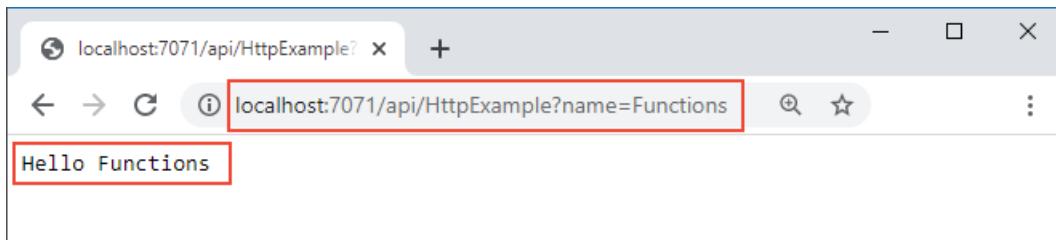
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
...
```

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. When you're done, use `Ctrl+C` and choose `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:

```
{  
    "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`, pasting 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](#)
 - [PowerShell](#)
 - [Azure CLI](#)

```
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 should include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
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 first name you used when testing the function earlier. The command reads and removes the first message from the queue.

- [bash](#)
- [PowerShell](#)
- [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>
```

In the local project folder, use the following Maven command to republish your project:

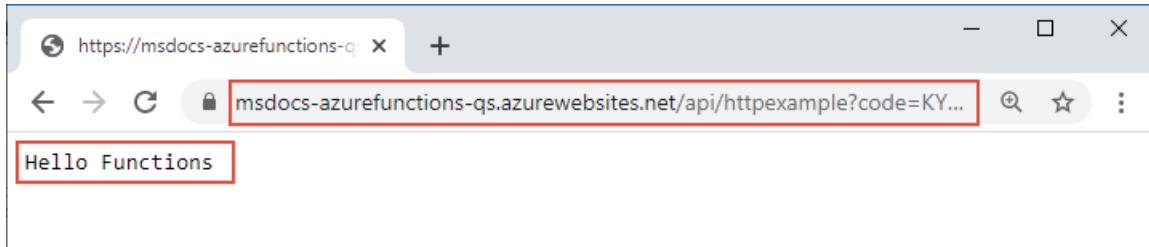
```
mvn azure-functions:deploy
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

- [Browser](#)
- [curl](#)

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.



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.

```
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](#)

- [Examples of complete Function projects in JavaScript.](#)
- [Azure Functions JavaScript developer guide](#)
- [Examples of complete Function projects in TypeScript.](#)
- [Azure Functions TypeScript developer guide](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)
- [Examples of complete Function projects in PowerShell.](#)
- [Azure Functions PowerShell developer guide](#)

Create a function that integrates with Azure Logic Apps

11/2/2020 • 10 minutes to read • [Edit Online](#)

Azure Functions integrates with Azure Logic Apps in the Logic Apps Designer. This integration lets you use the computing power of Functions in orchestrations with other Azure and third-party services.

This tutorial shows you how to use Azure Functions with Logic Apps and Cognitive Services on Azure to run sentiment analysis from Twitter posts. An HTTP trigger function categorizes tweets as green, yellow, or red based on the sentiment score. An email is sent when poor sentiment is detected.

The screenshot shows the Microsoft Azure Logic Apps Designer interface. On the left, there's a sidebar with various icons and links like Dashboard, Activity log, Access control (IAM), Tags, Development Tools (Logic App Designer, Logic App Code View, Versions, API Connections, Quick Start Guides, Release notes), Settings (Workflow settings, Access keys, Properties), and Essentials. The main area is titled 'TweetSentiment' and shows the 'Overview' tab selected. It displays the following details:

- Trigger:** TWITTER, When a new tweet is posted.
- Actions:** COUNT, 4 actions, with a link to 'View in Logic Apps designer'.
- EVALUATION:** Evaluated 0 times, fired 100 times in the last 24 hours, with a link to 'See trigger history'.
- Runs history:** A table showing three successful runs:

| STATUS | START TIME | IDENTIFIER | DURATION |
|-----------|--------------------|-----------------------------------|------------------|
| Succeeded | 11/5/2018, 1:59 PM | 08586601625339934553355234860CU24 | 671 Milliseconds |
| Succeeded | 11/5/2018, 1:59 PM | 08586601625339934553355234860CU24 | 1.04 Seconds |
| Succeeded | 11/5/2018, 1:58 PM | 08586601625840915865006727030CU15 | 591 Milliseconds |

In this tutorial, you learn how to:

- Create a Cognitive Services API Resource.
- Create a function that categorizes tweet sentiment.
- Create a logic app that connects to Twitter.
- 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 [Twitter](#) account.
- An [Outlook.com](#) account (for sending notifications).

NOTE

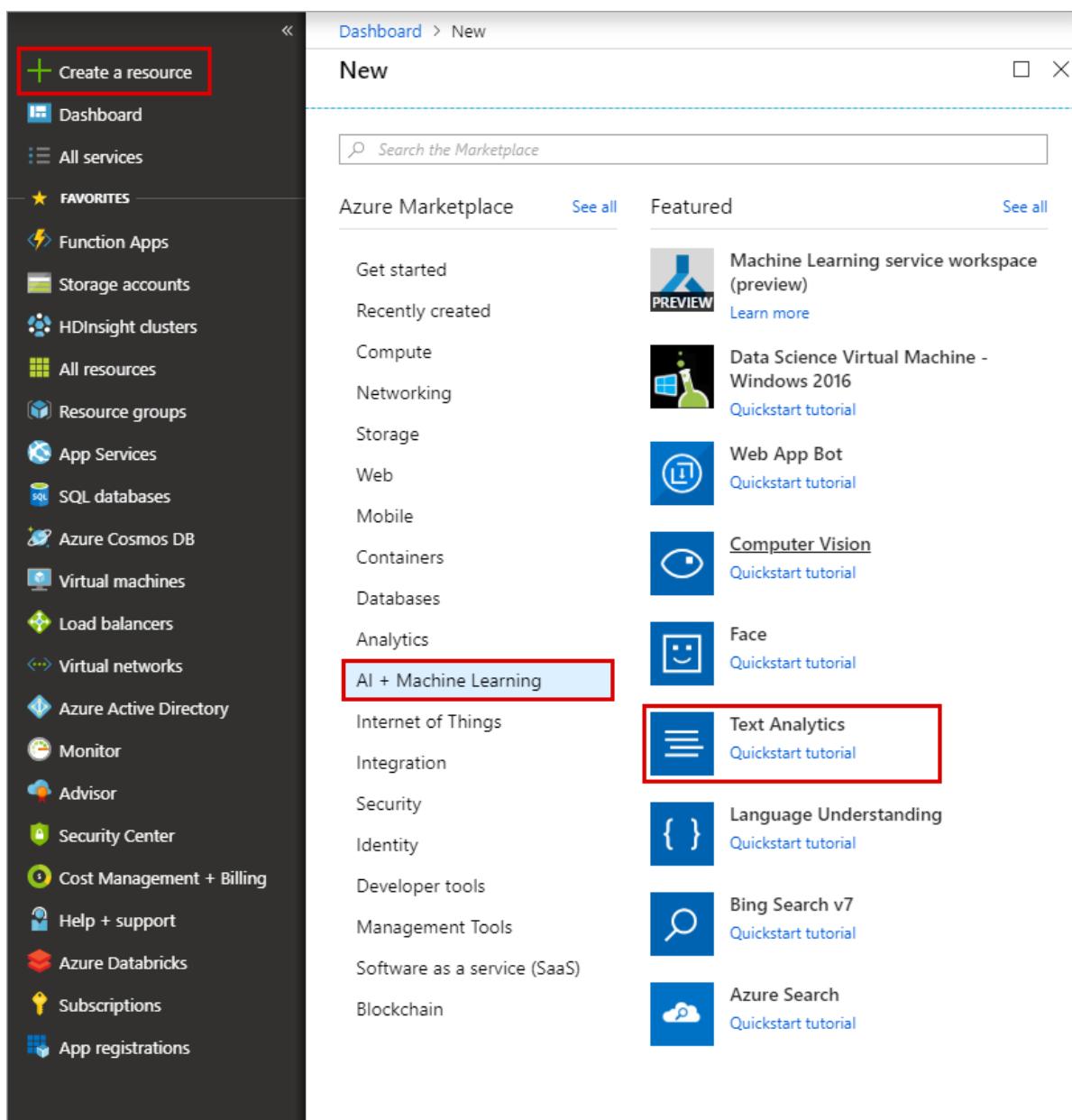
If you want to use the Gmail connector, only G-Suite business accounts can use this connector without restrictions in logic apps. If you have a Gmail consumer account, you can use the Gmail connector with only specific Google-approved apps and services, or you can [create a Google client app to use for authentication in your Gmail connector](#). For more information, see [Data security and privacy policies for Google connectors in Azure Logic Apps](#).

- This article uses as its starting point the resources created in [Create your first function from the Azure portal](#). If you haven't already done so, complete these steps now to create your function app.

Create a Cognitive Services resource

The Cognitive Services APIs are available in Azure as individual resources. Use the Text Analytics API to detect the sentiment of the tweets being monitored.

1. Sign in to the [Azure portal](#).
2. Click **Create a resource** in the upper left-hand corner of the Azure portal.
3. Click **AI + Machine Learning > Text Analytics**. Then, use the settings as specified in the table to create the resource.



| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------|--------------------------|--|
| Name | MyCognitiveServicesAccnt | Choose a unique account name. |
| Location | West US | Use the location nearest you. |
| Pricing tier | F0 | Start with the lowest tier. If you run out of calls, scale to a higher tier. |
| Resource group | myResourceGroup | Use the same resource group for all services in this tutorial. |

4. Click **Create** to create your resource.
5. Click **Overview** and copy the value of the **Endpoint** to a text editor. This value is used when creating a connection to the Cognitive Services API.

The screenshot shows the Azure portal interface for a Cognitive Services account named "MyCognitiveServicesAccnt". The left sidebar has a "Keys" item highlighted with a red box. The main content area shows the "Overview" tab selected. The "Essentials" section displays various account details, and the "Endpoint" field is highlighted with a red box, containing the URL <https://westcentralus.api.cognitive.microsoft.com/text/analytics/v2.0>.

6. In the left navigation column, click **Keys**, and then copy the value of **Key 1** and set it aside in a text editor. You use the key to connect the logic app to your Cognitive Services API.

The screenshot shows the Azure portal interface for the "MyCognitiveServicesAccnt - Keys" page. The left sidebar has a "Keys" item highlighted with a red box. The main content area shows the "KEY 1" field, which contains a series of asterisks (****) and is highlighted with a red box.

Create the function app

Azure Functions provides a great way to offload processing tasks in a logic apps workflow. This tutorial uses an HTTP trigger function to process tweet sentiment scores from Cognitive Services and return a category value.

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 a runtime that supports your favorite function programming language. Choose .NET Core for C# and F# functions. |
| 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. |

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 * [Visual Studio Enterprise](#)

Resource Group * [\(New\) myResourceGroup](#) [Create new](#)

Instance Details

Function App name * [myfunctionapp](#) [.azurewebsites.net](#)

Publish * [Code](#) [Docker Container](#)

Runtime stack * [.NET Core](#)

Version * [3.1](#)

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 | 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. When you run in an App Service plan, you must manage the scaling of your function app . |

Function App X

[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 * ▼
[Create new](#)

Operating system

The Operating System has been recommended for you based on 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 * Consumption (Serverless) ○ ▼

[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. |

Function App X

Basics Hosting **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

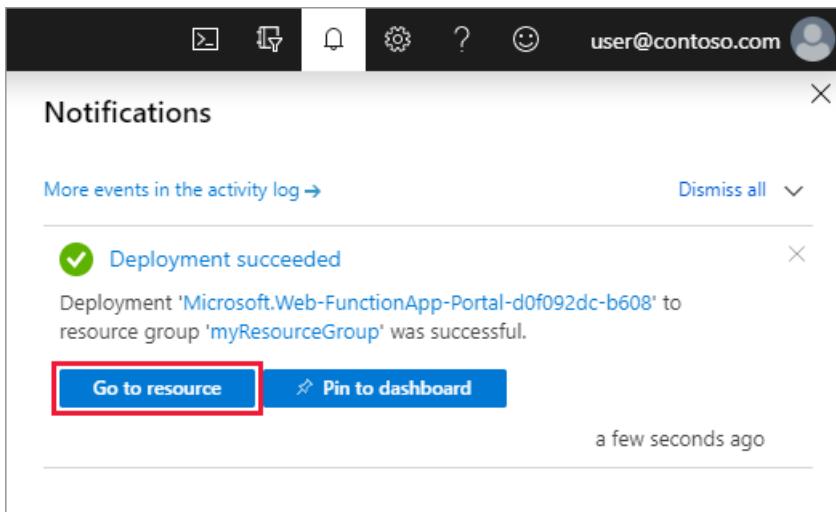
Enable Application Insights * No Yes

Application Insights * (New) myfunctionapp (Central US) ▼
[Create new](#)

Region Central US

Review + create < Previous Next : Tags >

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 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**.

A screenshot of the Azure Functions blade for a function app named "myfunctionapp". The left sidebar shows "Functions" selected. The main area is titled "New Function" with a sub-header "Create a new function in this Function App. Start by selecting a template below." It shows four templates: "HTTP trigger" (selected and highlighted with a red box), "Timer trigger", "Azure Queue Storage trigger", and "Azure Service Bus Queue trigger".

3. From the **New Function** page, select **Create Function**.
4. In your new HTTP trigger function, select **Code + Test** from the left menu, replace the contents of the `run.csx` file with the following code, and then select **Save**:

```

#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 category = "GREEN";

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    log.LogInformation(string.Format("The sentiment score received is '{0}'.", requestBody));

    double score = Convert.ToDouble(requestBody);

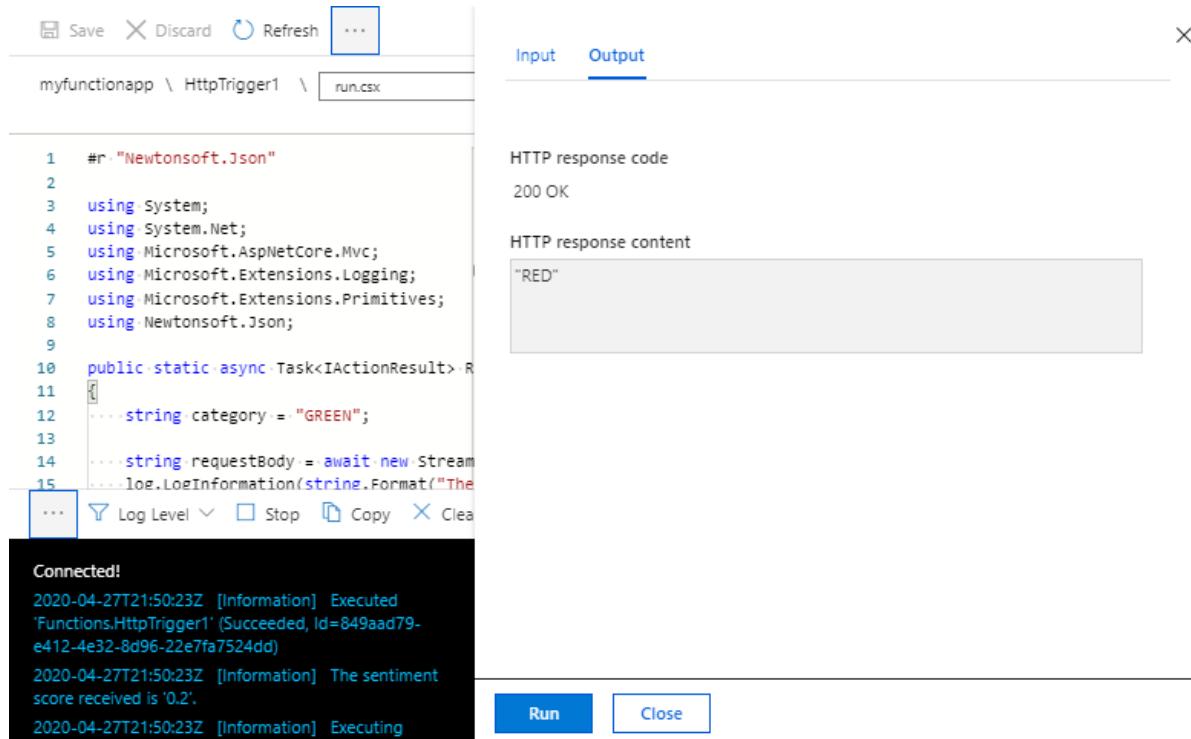
    if(score < .3)
    {
        category = "RED";
    }
    else if (score < .6)
    {
        category = "YELLOW";
    }

    return requestBody != null
        ? (ActionResult)new OkObjectResult(category)
        : new BadRequestObjectResult("Please pass a value on the query string or in the request body");
}

```

This function code returns a color category based on the sentiment score received in the request.

- To test the function, select **Test** from the top menu. On the **Input** tab, enter a value of `0.2` in the **Body**, and then select **Run**. A value of **RED** is returned in the **HTTP response content** on the **Output** tab.



Now you have a function that categorizes sentiment scores. Next, you create a logic app that integrates your function with your Twitter and Cognitive Services API.

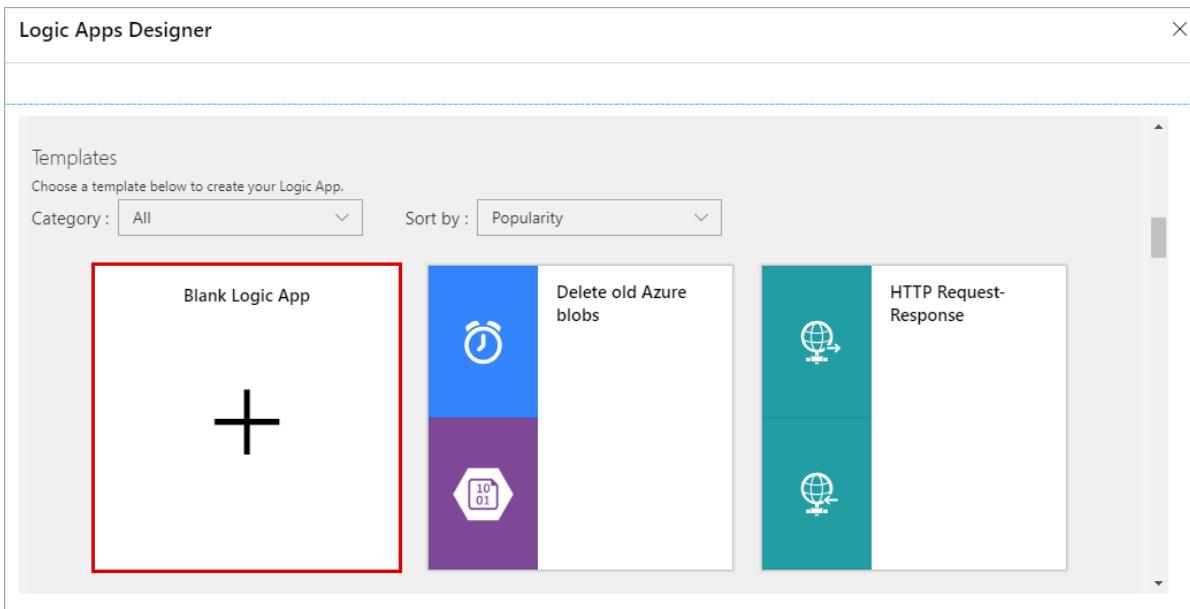
Create a logic app

1. In the Azure portal, click the **Create a resource** button found on the upper left-hand corner of the Azure portal.
2. Click **Web > Logic App**.
3. Then, type a value for **Name** like `TweetSentiment`, and use the settings as specified in the table.

The screenshot shows the Azure portal's 'New' blade for creating a new resource. On the left, the 'Create a resource' button is highlighted with a red box. The main area shows the 'Azure Marketplace' with various service categories: Get started, Recently created, Compute, Networking, Storage, Web (which is highlighted with a red box), Mobile, Containers, Databases, Analytics, AI + Machine Learning, Internet of Things, Integration, Security, Identity, Developer tools, Management Tools, Software as a service (SaaS), and Blockchain. On the right, the 'Featured' section lists 'Web App', 'Logic App' (which is highlighted with a red box), 'Web App for Containers', 'CDN', 'Media Services', 'Azure Search', 'API App', 'Template deployment', and 'API management'. Each item has a 'Quickstart tutorial' link next to it.

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------|-----------------|--|
| Name | TweetSentiment | Choose an appropriate name for your app. |
| Resource group | myResourceGroup | Choose the same existing resource group as before. |
| Location | East US | Choose a location close to you. |

4. Once you have entered the proper settings values, click **Create** to create your logic app.
5. After the app is created, click your new logic app pinned to the dashboard. Then in the Logic Apps Designer, scroll down and click the **Blank Logic App** template.



You can now use the Logic Apps Designer to add services and triggers to your app.

Connect to Twitter

First, create a connection to your Twitter account. The logic app polls for tweets, which trigger the app to run.

- In the designer, click the **Twitter** service, and click the **When a new tweet is posted** trigger. Sign in to your Twitter account and authorize Logic Apps to use your account.
- Use the Twitter trigger settings as specified in the table.

The screenshot shows the Logic Apps Designer with the Twitter trigger configuration. The 'Save' button is highlighted with a red box. The trigger is set to 'When a new tweet is posted'. The 'Search text' field contains '#azure'. The 'Interval' field is set to '15' and the 'Frequency' field is set to 'Minute'.

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|-------------|-----------------|--|
| Search text | #Azure | Use a hashtag that is popular enough to generate new tweets in the chosen interval. When using the Free tier and your hashtag is too popular, you can quickly use up the transaction quota in your Cognitive Services API. |
| Interval | 15 | The time elapsed between Twitter requests, in frequency units. |
| Frequency | Minute | The frequency unit used for polling Twitter. |

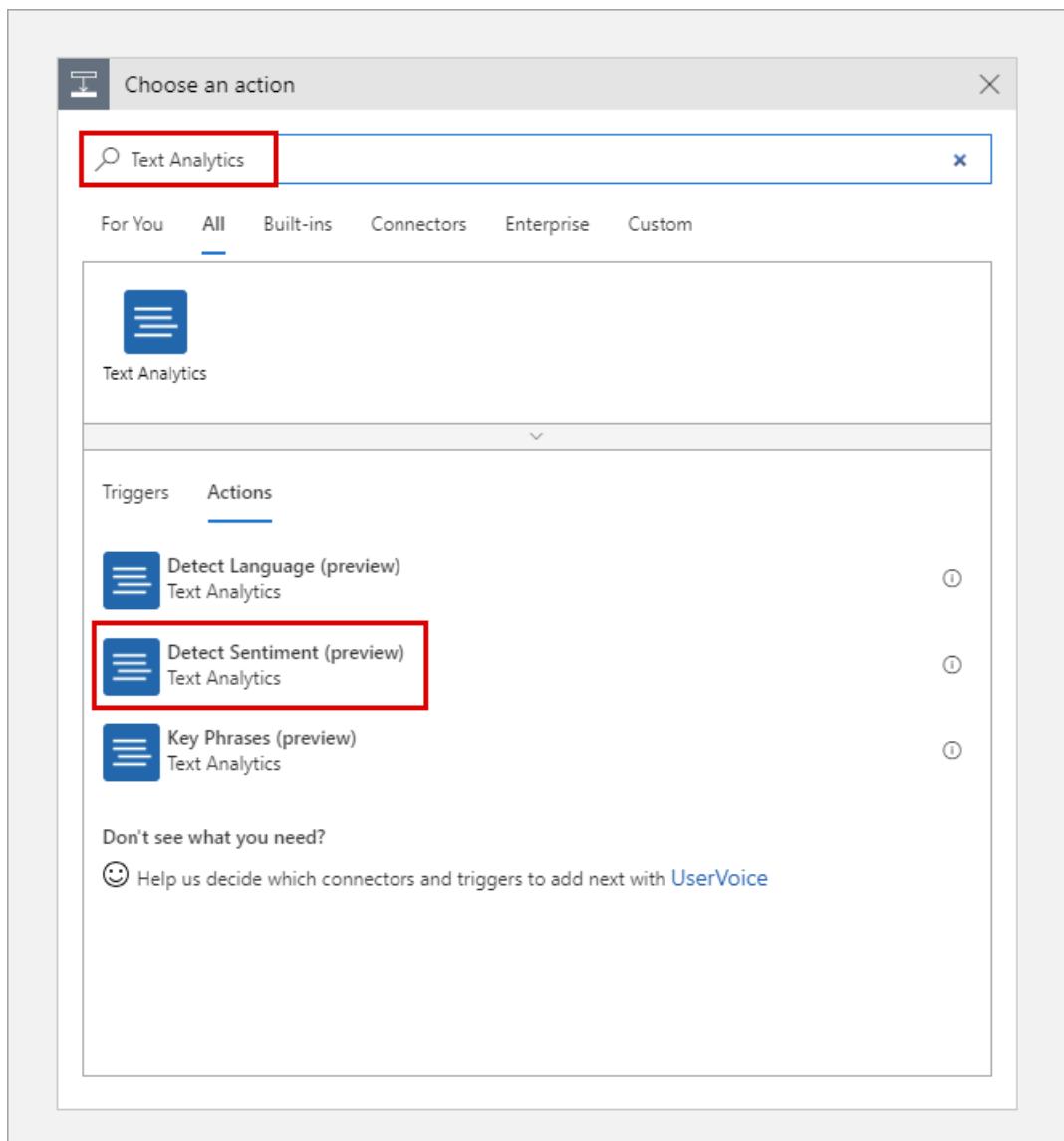
3. Click **Save** to connect to your Twitter account.

Now your app is connected to Twitter. Next, you connect to text analytics to detect the sentiment of collected tweets.

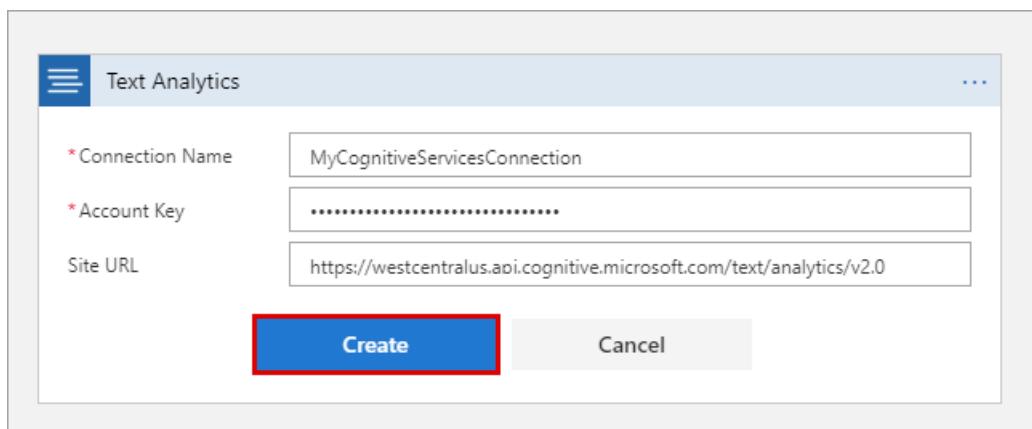
Add sentiment detection

1. Click **New Step**, and then **Add an action**.

2. In **Choose an action**, type **Text Analytics**, and then click the **Detect sentiment** action.



3. Type a connection name such as `MyCognitiveServicesConnection`, paste the key for your Cognitive Services API and the Cognitive Services endpoint you set aside in a text editor, and click **Create**.



4. Next, enter **Tweet text** in the text box and then click **New Step**.

The screenshot shows the Logic Apps Designer interface. A trigger step 'When a new tweet is posted' is at the top. Below it is a 'Detect Sentiment (Preview)' step. In the 'Text' input field of this step, there is a placeholder 'Tweet text' with a small 'x' icon to its right. To the right of the step, a vertical pane titled 'Dynamic content' lists several tweet properties. One item, 'Tweet text', is highlighted with a red rectangular border.

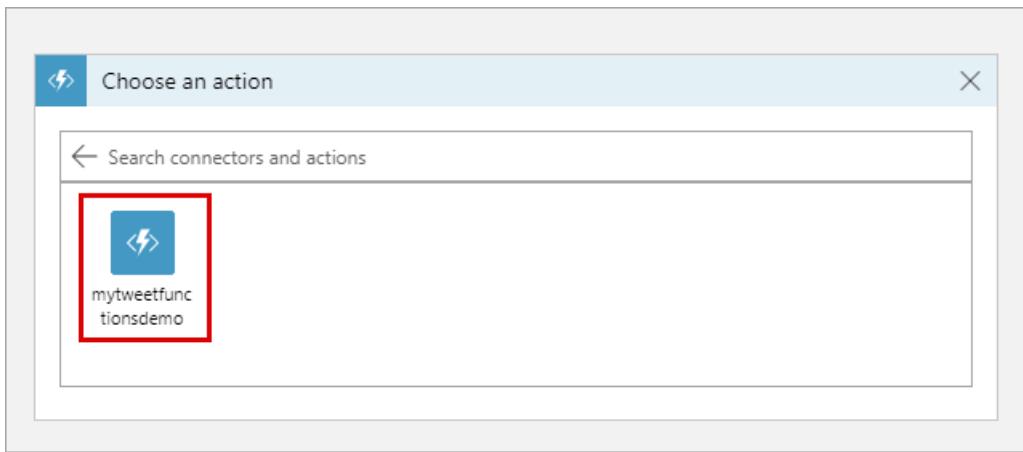
Now that sentiment detection is configured, you can add a connection to your function that consumes the sentiment score output.

Connect sentiment output to your function

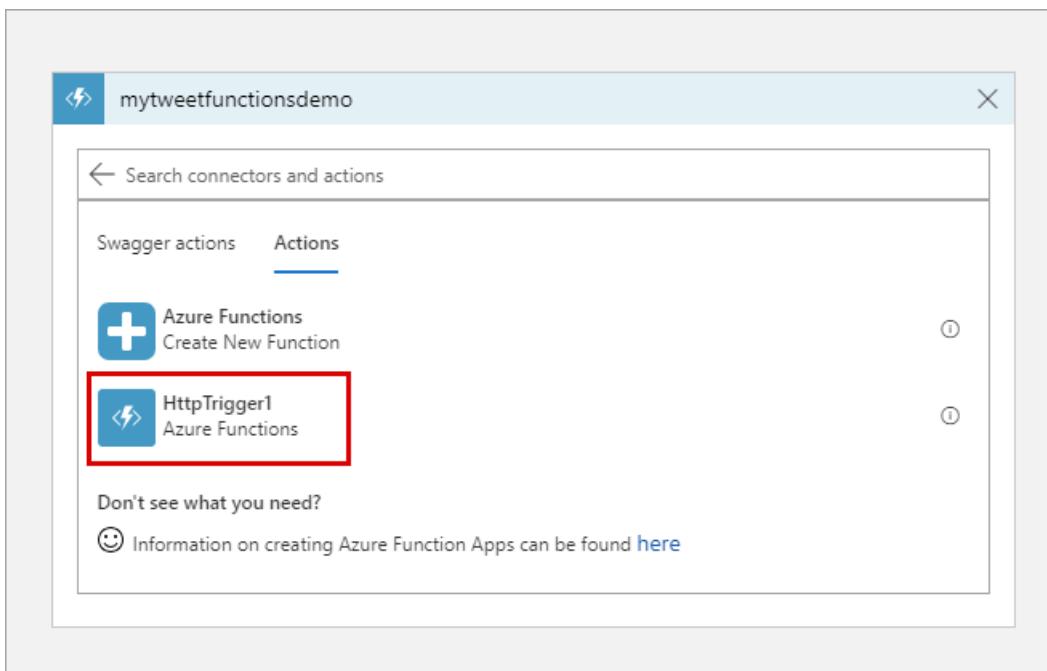
1. In the Logic Apps Designer, click **New step > Add an action**, filter on **Azure Functions** and click **Choose an Azure function**.

The screenshot shows the 'Choose an action' dialog. At the top, there's a search bar with 'Azure Functions' typed into it. Below the search bar are tabs for 'For You', 'All', 'Built-ins', 'Connectors', 'Enterprise', and 'Custom'. The 'All' tab is selected. In the main list area, there's a category for 'Azure Functions' which contains a single item: 'Choose an Azure function'. This item has a blue lightning bolt icon and is also highlighted with a red rectangular box. Below this list, there are two buttons: 'Triggers' and 'Actions', with 'Actions' being the active tab. At the bottom of the dialog, there are two links: 'Don't see what you need?' and 'Help us decide which connectors and triggers to add next with UserVoice'.

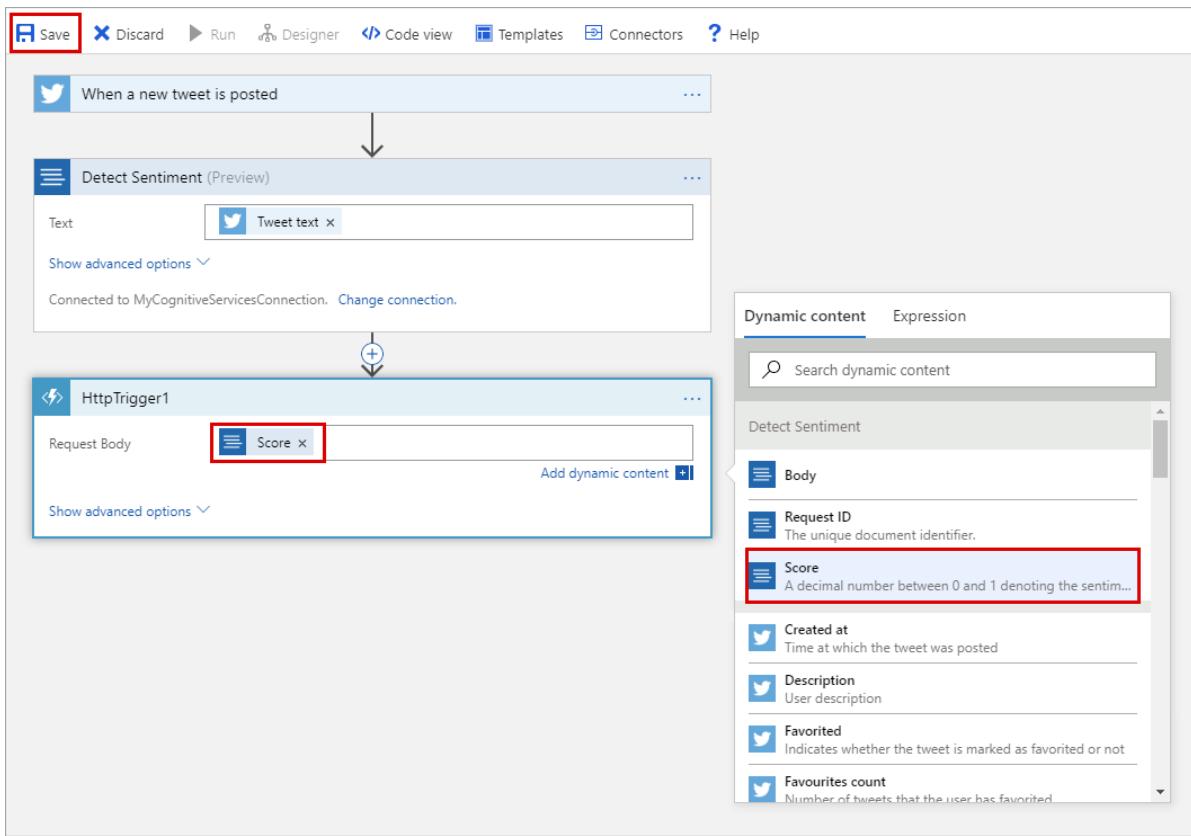
2. Select the function app you created earlier.



3. Select the function you created for this tutorial.



4. In Request Body, click Score and then Save.

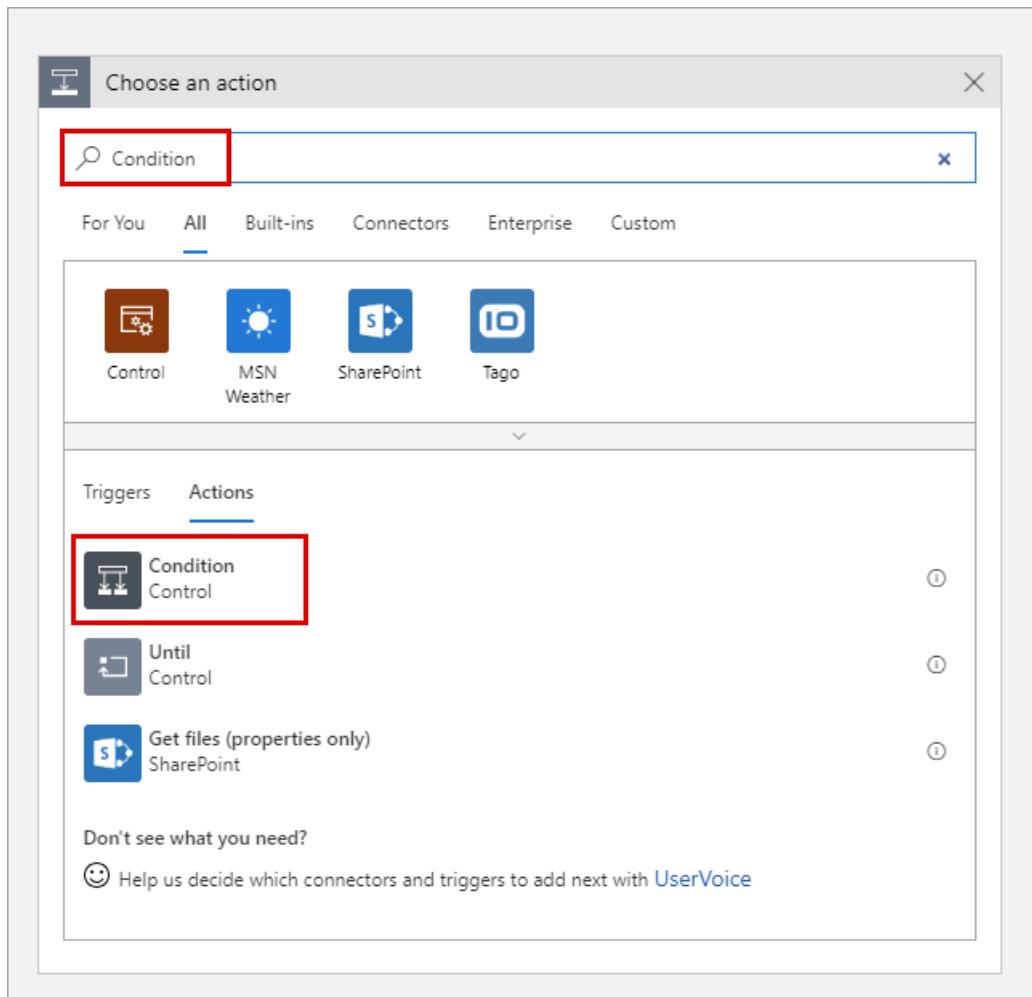


Now, your function is triggered when a sentiment score is sent from the logic app. A color-coded category is returned to the logic app by the function. Next, you add an email notification that is sent when a sentiment value of RED is returned from the function.

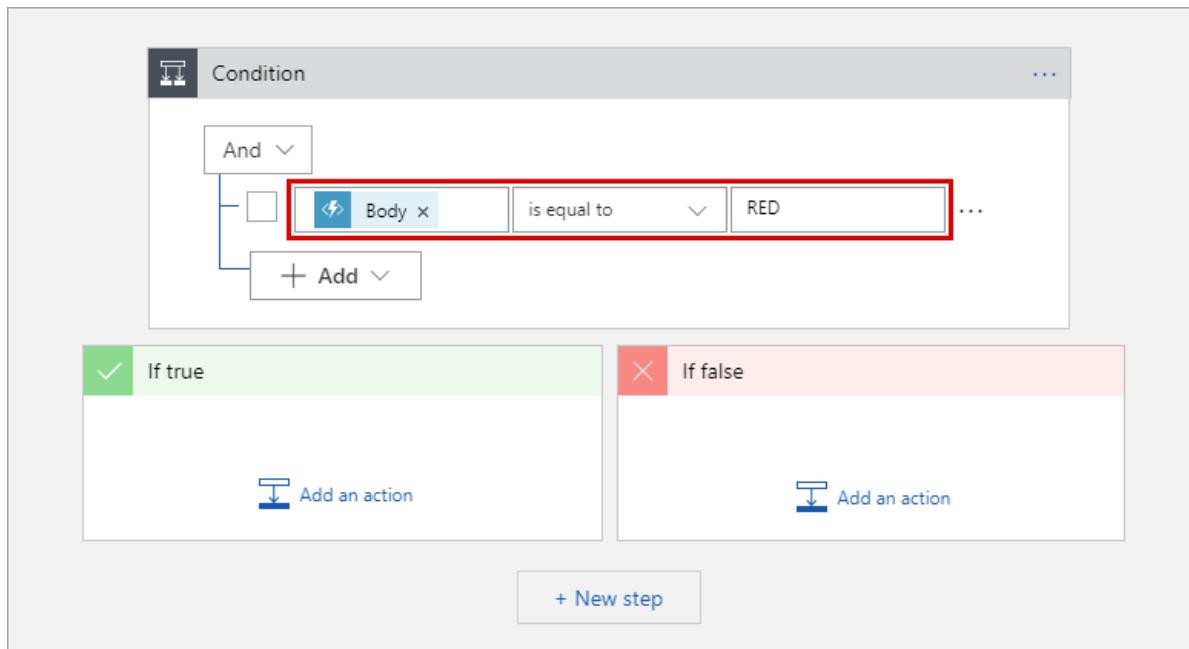
Add email notifications

The last part of the workflow is to trigger an email when the sentiment is scored as *RED*. This article uses an Outlook.com connector. You can perform similar steps to use a Gmail or Office 365 Outlook connector.

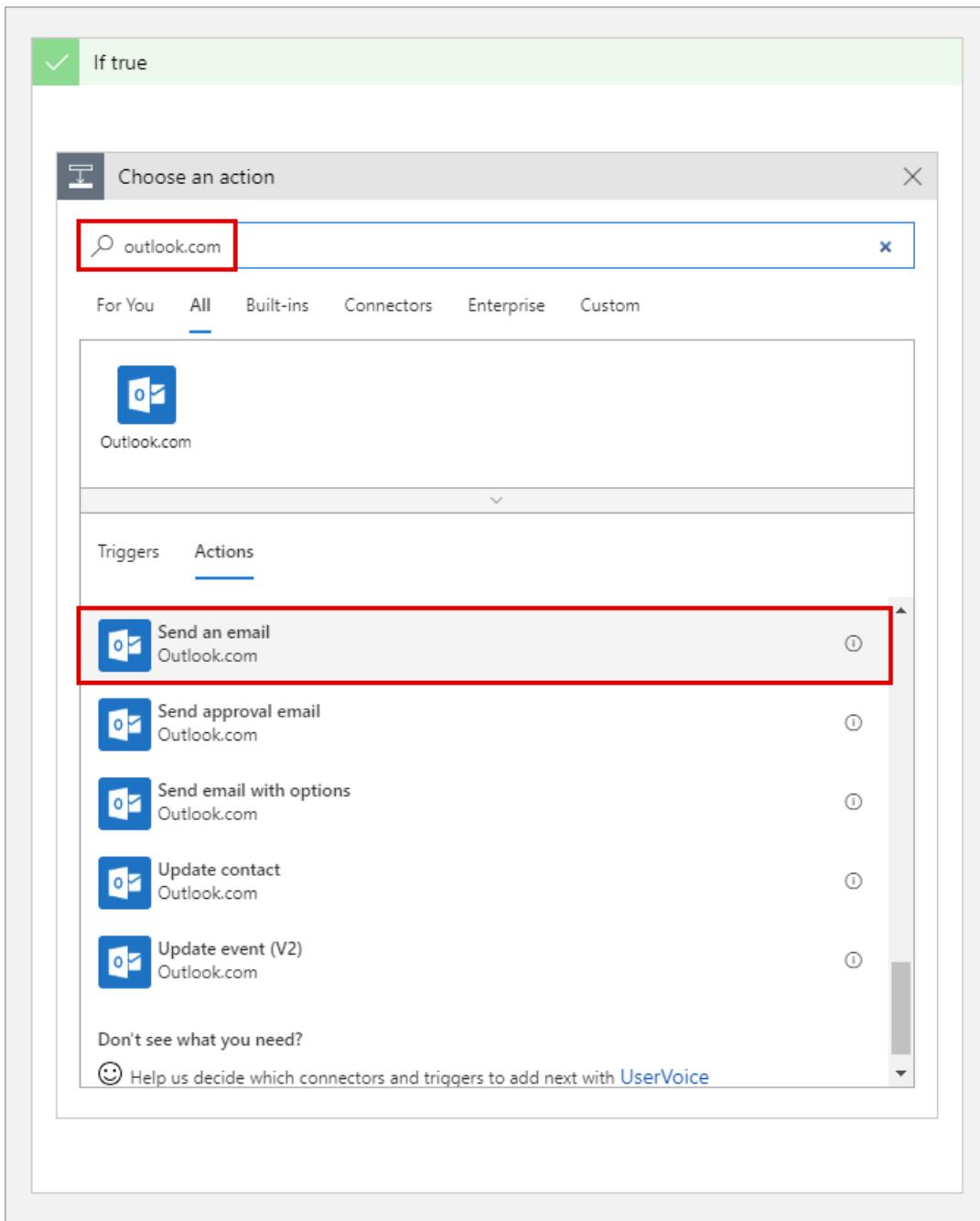
1. In the Logic Apps Designer, click **New step > Add a condition**.



2. Click **Choose a value**, then click **Body**. Select **is equal to**, click **Choose a value** and type **RED**, and click **Save**.



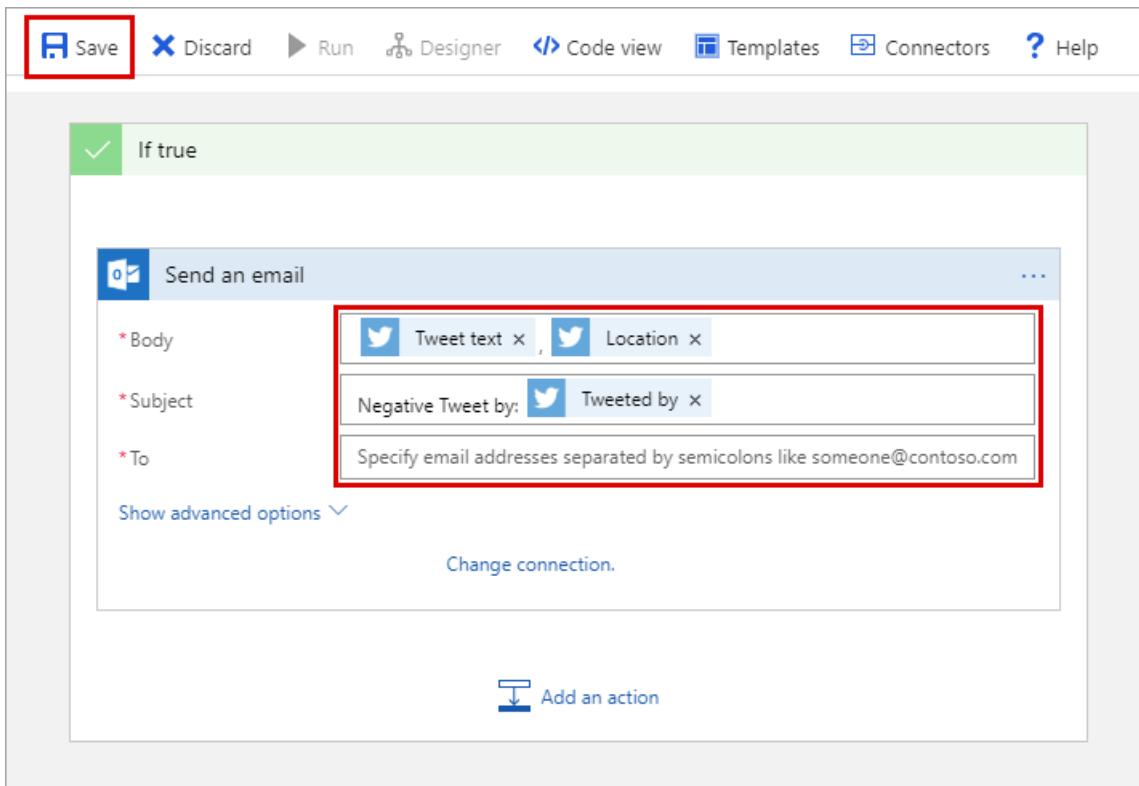
3. In IF TRUE, click **Add an action**, search for **outlook.com**, click **Send an email**, and sign in to your Outlook.com account.



NOTE

If you don't have an Outlook.com account, you can choose another connector, such as Gmail or Office 365 Outlook.

4. In the **Send an email** action, use the email settings as specified in the table.



| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|-----------------------------------|---|
| To | Type your email address | The email address that receives the notification. |
| Subject | Negative tweet sentiment detected | The subject line of the email notification. |
| Body | Tweet text, Location | Click the Tweet text and Location parameters. |

1. Click **Save**.

Now that the workflow is complete, you can enable the logic app and see the function at work.

Test the workflow

1. In the Logic App Designer, click **Run** to start the app.
2. In the left column, click **Overview** to see the status of the logic app.

| STATUS | START TIME | IDENTIFIER | DURATION |
|-----------|--------------------|-----------------------------------|------------------|
| Succeeded | 11/5/2018, 1:59 PM | 08586601625339934552355234860CU24 | 671 Milliseconds |
| Succeeded | 11/5/2018, 1:59 PM | 08586601625339934553355234860CU24 | 1.04 Seconds |
| Succeeded | 11/5/2018, 1:58 PM | 08586601625840915865006727030CU15 | 591 Milliseconds |
| Succeeded | 11/5/2018, 1:57 PM | 08586601626216029997972591193CU23 | 765 Milliseconds |
| Succeeded | 11/5/2018, 1:57 PM | 08586601626216029998972591193CU23 | 764 Milliseconds |
| Succeeded | 11/5/2018, 1:56 PM | 08586601626817889036422295301CU11 | 1.06 Seconds |
| Succeeded | 11/5/2018, 1:56 PM | 08586601626817889037422295301CU11 | 1.1 Seconds |
| Succeeded | 11/5/2018, 1:56 PM | 08586601626999922865795875911CU10 | 892 Milliseconds |
| Succeeded | 11/5/2018, 1:56 PM | 08586601626999922866795875911CU10 | 902 Milliseconds |
| Succeeded | 11/5/2018, 1:56 PM | 08586601626999922867795875911CU10 | 845 Milliseconds |

3. (Optional) Click one of the runs to see details of the execution.

4. Go to your function, view the logs, and verify that sentiment values were received and processed.

```

Logs
2017-05-13T17:04:47 No new trace in the past 4 min(s).
2017-05-13T17:05:47 No new trace in the past 5 min(s).
2017-05-13T17:05:57.816 Function started (Id=318e055a-ec07-4667-ae55-e0313e10ea1b)
2017-05-13T17:05:57.848 The sentiment score received is '0.908686587323299'.
2017-05-13T17:05:57.848 Function completed (Success, Id=318e055a-ec07-4667-ae55-e0313e10ea1b, Duration=33ms)
2017-05-13T17:05:57.863 Function started (Id=3fea3662-b468-4efd-bbbd-6f88cf3361a7)
2017-05-13T17:05:57.863 The sentiment score received is '0.5'.
2017-05-13T17:05:57.863 Function completed (Success, Id=3fea3662-b468-4efd-bbbd-6f88cf3361a7, Duration=0ms)
2017-05-13T17:05:57.942 Function started (Id=7ea7351d-7894-46hf-h5f9-f4a3e79hd9f9)

```

5. When a potentially negative sentiment is detected, you receive an email. If you haven't received an email, you can change the function code to return RED every time:

```
return (ActionResult)new OkObjectResult("RED");
```

After you have verified email notifications, change back to the original code:

```
return requestBody != null
? (ActionResult)new OkObjectResult(category)
: new BadRequestObjectResult("Please pass a value on the query string or in the request body");
```

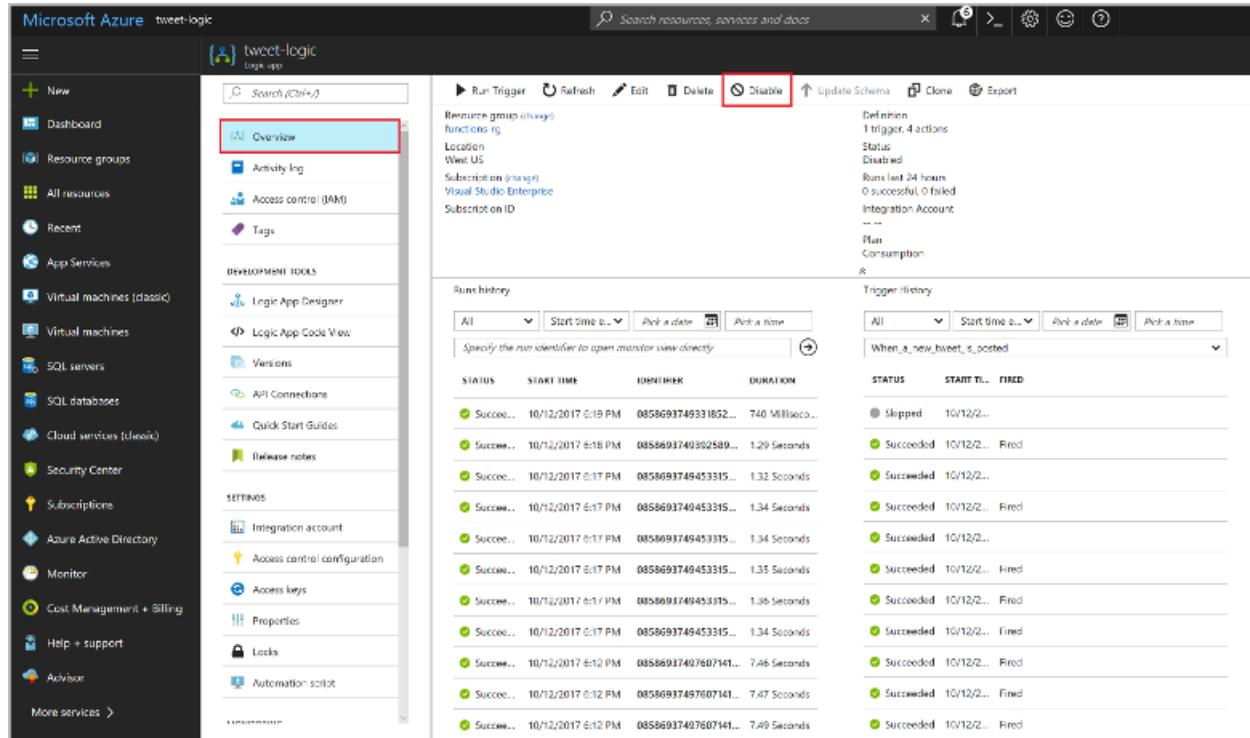
IMPORTANT

After you have completed this tutorial, you should disable the logic app. By disabling the app, you avoid being charged for executions and using up the transactions in your Cognitive Services API.

Now you've seen how easy it is to integrate Functions into a Logic Apps workflow.

Disable the logic app

To disable the logic app, click **Overview** and then click **Disable** at the top of the screen. Disabling the app stops it from running and incurring charges without deleting the app.



The screenshot shows the Azure portal interface for a logic app named 'tweet-logic'. On the left, there's a sidebar with various service links like New, Dashboard, Resource groups, etc. The main area has a title bar with the logic app name and a search bar. Below the title bar, there are tabs for Run Trigger, Refresh, Edit, Delete, and Disable. The 'Disable' button is highlighted with a red box. To the right of the tabs, there's a summary section with details like Resource group, Location, Subscription, and Integration Account. Below this is a 'Runs history' table showing several successful executions. Further down is a 'Trigger history' table showing multiple successful executions for the 'When_a_new_tweet_is_posted' trigger.

| Status | Start Time | Duration |
|-----------|--------------------|------------------|
| Succeeded | 10/12/2017 6:19 PM | 740 Milliseconds |
| Succeeded | 10/12/2017 6:19 PM | 1.29 Seconds |
| Succeeded | 10/12/2017 6:17 PM | 1.32 Seconds |
| Succeeded | 10/12/2017 6:17 PM | 1.34 Seconds |
| Succeeded | 10/12/2017 6:17 PM | 1.34 Seconds |
| Succeeded | 10/12/2017 6:17 PM | 1.35 Seconds |
| Succeeded | 10/12/2017 6:17 PM | 1.36 Seconds |
| Succeeded | 10/12/2017 6:17 PM | 1.34 Seconds |
| Succeeded | 10/12/2017 6:12 PM | 7.46 Seconds |
| Succeeded | 10/12/2017 6:12 PM | 7.47 Seconds |
| Succeeded | 10/12/2017 6:12 PM | 7.49 Seconds |

| Status | Start Time |
|-----------|------------------|
| Skipped | 10/12/2... |
| Succeeded | 10/12/2... Fired |
| Succeeded | 10/12/2... |
| Succeeded | 10/12/2... Fired |

Next steps

In this tutorial, you learned how to:

- Create a Cognitive Services API Resource.
- Create a function that categorizes tweet sentiment.
- Create a logic app that connects to Twitter.
- Add sentiment detection to the logic app.
- Connect the logic app to the function.
- Send an email based on the response from the function.

Advance to the next tutorial to learn how to create a serverless API for your function.

[Create a serverless API using Azure Functions](#)

To learn more about Logic Apps, see [Azure Logic Apps](#).

Create an OpenAPI definition for a serverless API using Azure API Management

11/2/2020 • 7 minutes to read • [Edit Online](#)

REST APIs are often described using an OpenAPI definition. This definition contains information about what operations are available in an API and how the request and response data for the API should be structured.

In this tutorial, you create a function that determines whether an emergency repair on a wind turbine is cost-effective. You then create an OpenAPI definition for the function app using [Azure API Management](#) so that the function can be called from other apps and services.

In this tutorial, you learn how to:

- Create a function in Azure
- Generate an OpenAPI definition using Azure API Management
- Test the definition by calling the function
- Download the OpenAPI definition

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.

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. Choose .NET Core for C# and F# functions. |
| Version | Version number | Choose the version of your installed runtime. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|------------------|--|
| Region | Preferred region | Choose a region near you or near other services your functions access. |

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 * | <input type="text" value="Visual Studio Enterprise"/> |
| Resource Group * | <input type="text" value="(New) myResourceGroup"/> Create new |

Instance Details

| | |
|---------------------|--|
| Function App name * | <input type="text" value="myfunctionapp"/> .azurewebsites.net |
| Publish * | <input checked="" type="radio"/> Code <input type="radio"/> Docker Container |
| Runtime stack * | <input type="text" value=".NET Core"/> |
| Version * | <input type="text" value="3.1"/> |
| Region * | <input type="text" value="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. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|--------------------------|--|
| Plan | Consumption (Serverless) | <p>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. When you run in an App Service plan, you must manage the scaling of your function app.</p> |

Function App X

[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 * ▼
[Create new](#)

Operating system
The Operating System has been recommended for you based on 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 * ▼

[Review + create](#) < Previous Next : Monitoring >

5. Select **Next : Monitoring**. On the **Monitoring** page, enter the following settings.

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------------|-----------------|--|
| Application Insights | Default | <p>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.</p> |

Function App X

Basics Hosting **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes (New) myfunctionapp (Central US)

Create new

Application Insights *

Region Central US

Review + create < Previous Next : Tags >

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.

More events in the activity log → Dismiss all

Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource Pin to dashboard

a few seconds ago

Create the function

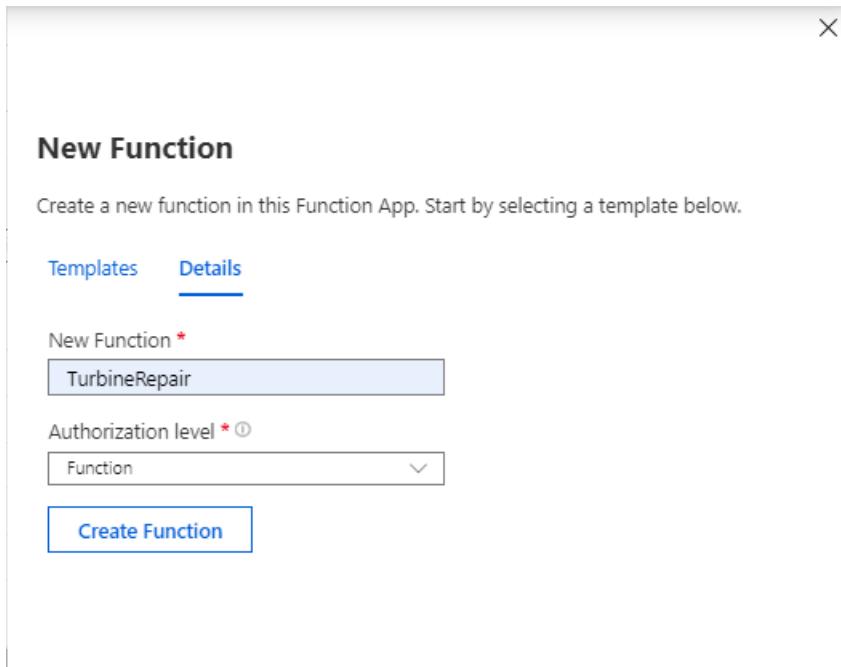
This tutorial uses an HTTP triggered function that takes two parameters:

- The estimated time to make a turbine repair, in hours.

- The capacity of the turbine, in kilowatts.

The function then calculates how much a repair will cost, and how much revenue the turbine could make in a 24-hour period. To create the HTTP triggered function in the [Azure portal](#):

1. From the left menu of your functions app, select **Functions**, and then select **Add** from the top menu.
2. In the **New Function** window, select **Http trigger**.
3. For **New Function**, enter `TurbineRepair`.
4. Choose **Function** from the **Authorization level** drop-down list, and then select **Create Function**.



5. Select **Code + Test**, and then select `run.csx` from the drop-down list. Replace the contents of the `run.csx` C# script file with the following code, then choose **Save**:

```

#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

const double revenuePerkW = 0.12;
const double technicianCost = 250;
const double turbineCost = 100;

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    // Get query strings if they exist
    int tempVal;
    int? hours = Int32.TryParse(req.Query["hours"], out tempVal) ? tempVal : (int?)null;
    int? capacity = Int32.TryParse(req.Query["capacity"], out tempVal) ? tempVal : (int?)null;

    // Get request body
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);

    // Use request body if a query was not sent
    capacity = capacity ?? data?.capacity;
    hours = 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 on the query string or 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 (ActionResult)new OkObjectResult(new{
        message = repairTurbine,
        revenueOpportunity = "$"+ revenueOpportunity,
        costToFix = "$"+ costToFix
    });
}

```

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.

6. To test the function, select **Test**, select the **Input** tab, enter the following input for the **Body**, and then select **Run**:

```
{
"hours": "6",
"capacity": "2500"
}
```

The screenshot shows the Azure Functions Test interface. On the left, the 'Code + Test' tab is selected. The main area displays the following C# code:

```

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  const double revenuePerKW = 0.12;
9  const double technicianCost = 250;
10 const double turbineCost = 100;
11
12 public static async Task<IActionResult> Run(HttpRequest req)
13 {
14     // Get query strings if they exist
15     int? tempVal;
16     int? hours = Int32.TryParse(req.Query["hours"], out tempVal) ? tempVal : null;
17     int? capacity = Int32.TryParse(req.Query["capacity"], out tempVal) ? tempVal : null;
18
19     // Get request body
20     string requestBody = await new StreamReader(req.Body).ReadToEnd();
    
```

On the right, the 'Input' tab is selected, showing the following JSON input parameters:

```

1  {
2      "hours": "6",
3      "capacity": "2500"
4  }
    
```

A red box highlights the 'Run' button at the bottom of the interface.

The following output is returned in the Output tab:

```
{"message": "Yes", "revenueOpportunity": "$7200", "costToFix": "$1600"}
```

Now you have a function that determines the cost-effectiveness of emergency repairs. Next, you generate an OpenAPI definition for the function app.

Generate the OpenAPI definition

To generate the OpenAPI definition:

1. Select the function app, choose **API Management** from the left menu, and then select **Create new** under **API Management**.

The screenshot shows the 'API Management' blade in the Azure portal. The 'Development Tools' sidebar has 'API Management' selected. The main area is titled 'API Management' and contains the following steps:

- Expose your HTTP trigger Functions through Azure API Management - manage, protect, secure, and publish the APIs. [Learn more](#)
- Select API Management instance and API for your App.
- API Management ***: A dropdown menu with 'Create new' selected.
- API**: A list of options including 'Import Functions' (checked), 'Enable Application Insights' (unchecked), and 'Link API'.

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. |
| Subscription | Your subscription | The subscription under which this new resource is created. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------------|-----------------|--|
| Resource group | myResourceGroup | The same resource as your function app, which should get set for you. |
| Location | West US | Choose the West US location. |
| 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 □ X

* Name
myfunctionapp-apim ✓
.azure-api.net

* Subscription
Visual Studio Enterprise ▼

* Resource group
myResourceGroup ▼
[Create new](#)

* Location
North Central US ▼

* Organization name i
Contoso ✓

* Administrator email i
admin@contoso.com ✓

Pricing tier ([View full pricing details](#))
Consumption (99.9 SLA, %) ▼

[Create](#)
[Automation options](#)

3. Choose **Create** 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, and then select **Link API**.

5. The Import Azure Functions opens with the **TurbineRepair** function highlighted. Choose **Select** to continue.

Import Azure Functions

API Management service

Don't see an Azure Function? Azure API Management requires Azure Functions to use the HTTP trigger and Function or Anonymous authorization level setting.

Search to filter items...

| NAME | HTTP METHODS | URL TEMPLATE |
|---|--------------|---------------|
| <input checked="" type="checkbox"/> TurbineRepair | GET, POST | TurbineRepair |

Select

6. 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.

Test the API

Before you use the OpenAPI definition, you should verify that the API works.

1. On your function app page, select **API Management**, select the **Test** tab, and then select **POST TurbineRepair**.
2. Enter the following code in the **Request body**:

```
{  
  "hours": "6",  
  "capacity": "2500"  
}
```

3. Select **Send**, and then view the **HTTP response**.

The screenshot shows the Azure portal interface for managing APIs. On the left, there's a sidebar with various options like 'App Service plan', 'Development Tools', and 'API'. Under 'API', 'API Management' is selected, which is also highlighted with a red box. The main area shows a 'Test' blade for a 'TurbineRepair' API. At the top, it says 'REVISION 1 UPDATED Apr 23, 2019, 12:06:13 PM'. Below that, there are tabs for 'Design', 'Settings', 'Test' (which is selected and highlighted with a red box), 'Revisions', and 'Change log'. The 'Test' tab has sections for 'Search operations', 'Filter by tags', 'Group by tag', 'GET TurbineRepair ...', 'POST TurbineRepair ...' (which is highlighted with a red box), and '...'. The 'Request body' section shows a JSON payload: { "hours": "6", "capacity": "2500" } (also highlighted with a red box). At the bottom right of the test blade, there's a large blue 'Send' button.

Download the OpenAPI definition

If your API works as expected, you can download the OpenAPI definition.

1. Select **Download OpenAPI definition** at the top of the page.



2. Save the downloaded JSON file, and then open it. Review the definition.

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

You have used API Management integration to generate an OpenAPI definition of your functions. You can now edit the definition in API Management in the portal. You can also [learn more about API Management](#).

[Edit the OpenAPI definition in API Management](#)

Tutorial: integrate Functions with an Azure virtual network

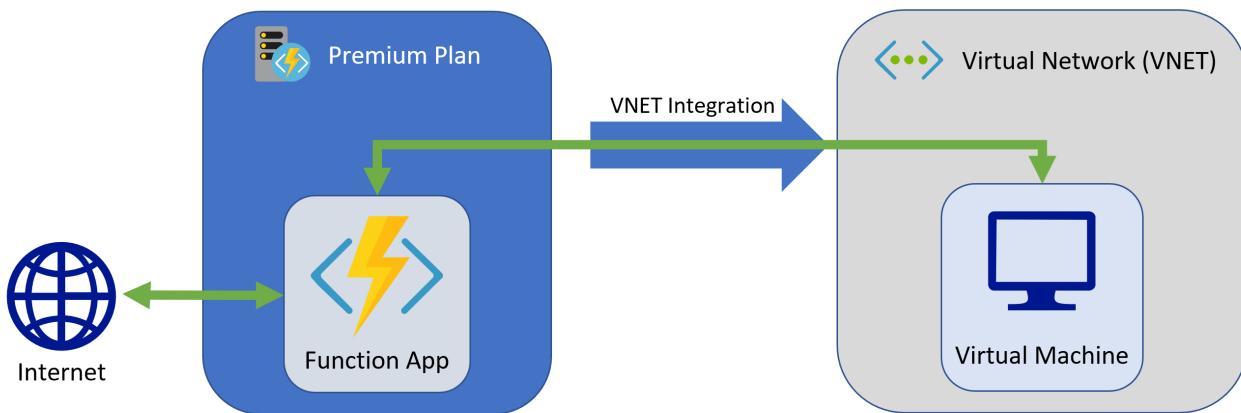
11/2/2020 • 9 minutes to read • [Edit Online](#)

This tutorial shows you how to use Azure Functions to connect to resources in an Azure virtual network. You'll create a function that has access to both the internet and to a VM running WordPress in virtual network.

- Create a function app in the Premium plan
- Deploy a WordPress site to VM in a virtual network
- Connect the function app to the virtual network
- Create a function proxy to access WordPress resources
- Request a WordPress file from inside the virtual network

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.

Create a function app in a Premium plan

First, you create a function app in the [Premium plan](#). This plan provides serverless scale while supporting virtual network integration.

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 a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions. |
| Region | Preferred region | Choose a region near you or near other services your functions access. |

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 * ⓘ

Resource Group * ⓘ [Create new](#)

Instance Details

Function App name * [.azurewebsites.net](#)

Publish * [Code](#) [Docker Container](#)

Runtime stack *

Region *

[Review + create](#) [< Previous](#) [Next : Hosting >](#)

4. Select **Next: Hosting**. On the **Hosting** page, enter the following settings:

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|-----------------|-------------|
|---------|-----------------|-------------|

| 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. |
| Plan | Premium | <p>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.</p> <p>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.</p> |

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 *

(New) storageaccountmyres93a6

[Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System *

[Linux](#) [Windows](#)

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * ⓘ

Premium

ⓘ Cannot find your App service plan? [Try a different location in Basics tab.](#)

Windows Plan (Central US) * ⓘ

(New) ASP-myResourceGroup-8ddf

[Create new](#)

Sku and size *

Elastic Premium EP1

210 total ACU, 3.5 GB memory

[Change size](#)

[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, you can change the New resource name or choose a different Location in an Azure geography to store your data. |

Function App

Basics Hosting Monitoring **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

Create new

Region

Central US

Review + create

< Previous Next : Tags >

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.

More events in the activity log → Dismiss all

Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource Pin to dashboard

a few seconds ago

You can pin the function app to the dashboard by selecting the pin icon in the upper right-hand corner. Pinning makes it easier to return to this function app after you create your VM.

Create a VM inside a virtual network

Next, create a preconfigured VM that runs WordPress inside a virtual network ([WordPress LEMP7 Max Performance](#) by Jetware). A WordPress VM is used because of its low cost and convenience. This same scenario works with any resource in a virtual network, such as REST APIs, App Service Environments, and other Azure services.

1. In the portal, choose **+ Create a resource** on the left navigation pane, in the search field type **WordPress LEMP7 Max Performance**, and press Enter.
2. Choose **Wordpress LEMP Max Performance** in the search results. Select a software plan of **Wordpress LEMP Max Performance for CentOS** as the **Software Plan** and select **Create**.
3. In the **Basics** tab, use the VM settings as specified in the table below the image:

Home > New > Wordpress LEMP Max Performance > Create a virtual machine

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 Enterprise](#)
└─ * Resource group [myResourceGroup](#) [Create new](#)

INSTANCE DETAILS

* Virtual machine name [VNET-WordPress](#)
* Region [\(Europe\) West Europe](#)

Availability options [No infrastructure redundancy required](#)

* Image [Wordpress LEMP7 Max Performance on CentOS](#) [Browse all images](#)

* Size [Standard B1s](#)
1 vcpu, 1 GB memory
[Change size](#)

ADMINISTRATOR ACCOUNT

Authentication type [Password](#) [SSH public key](#)
* Username [myusername](#)
* Password [.....](#)
* Confirm password [.....](#)

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|--------------|-------------------|--|
| Subscription | Your subscription | The subscription under which your resources are created. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------------|----------------------|---|
| Resource group | myResourceGroup | Choose <code>myResourceGroup</code> , or the resource group you created with your function app. Using the same resource group for the function app, WordPress VM, and hosting plan makes it easier to clean up resources when you're done with this tutorial. |
| Virtual machine name | VNET-Wordpress | The VM name needs to be unique in the resource group |
| Region | (Europe) West Europe | Choose a region near you or near the functions that access the VM. |
| Size | B1s | Choose Change size and then select the B1s standard image, which has 1 vCPU and 1 GB of memory. |
| Authentication type | Password | To use password authentication, you must also specify a Username , a secure Password , and then Confirm password . For this tutorial, you won't need to sign in to the VM unless you need to troubleshoot. |

4. Choose the **Networking** tab and under Configure virtual networks select **Create new**.

5. 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

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).

| ADDRESS RANGE | ADDRESSES | OVERLAP |
|---|---|--|
| <input type="checkbox"/> 10.10.0.0/16 ✓ | 10.10.0.0 - 10.10.255.255 (65536 addresses) | None Delete ... |
| <input type="checkbox"/> | (0 Addresses) | None |

SUBNETS

The subnet's address range in CIDR notation. It must be contained by the address space of the virtual network.

| SUBNET NAME | ADDRESS RANGE | ADDRESSES |
|---|--------------------------|---|
| <input type="checkbox"/> Tutorial-Net ✓ | 10.10.1.0/24 ✓ | 10.10.1.0 - 10.10.1.255 (256 addresses) Delete ... |
| <input type="checkbox"/> | <input type="checkbox"/> | (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-Net | 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 WordPress site. A /24 subnet provides 254 host addresses. |

6. Select **OK** to create the virtual network.
7. Back in the **Networking** tab, choose **None** for **Public IP**.
8. Choose the **Management** tab, then in **Diagnostics storage account**, choose the Storage account you created with your function app.
9. Select **Review + create**. After validation completes, select **Create**. The VM create process takes a few minutes. The created VM can only access the virtual network.
10. After the VM is created, choose **Go to resource** to view the page for your new VM, then choose **Networking** under **Settings**.
11. Verify that there's no **Public IP**. Make a note the **Private IP**, which you use to connect to the VM from your function app.

| PRIORITY | NAME | PORT | PROTOCOL | SOURCE |
|----------|------------------|------|----------|--------|
| 1010 | HTTP | 80 | TCP | Any |
| 1020 | HTTPS | 443 | TCP | Any |
| 1030 | HTTP_web_control | 1999 | TCP | Any |

You now have a WordPress site deployed entirely within your virtual network. This site isn't accessible from the public internet.

Connect your function app to the virtual network

With a WordPress site running in a VM in a virtual network, you can now connect your function app to that virtual network.

1. In your new function app, select **Networking** in the left menu.
2. Under **VNet Integration**, select **Click here to configure**.

 myfunctionapp | Networking
App Service

Search (Ctrl+/
)

Settings

- Configuration
- Authentication / Authorizati...
- Application Insights
- Identity
- Backups
- Custom domains
- TLS/SSL settings
- Networking**

 VNet Integration

Securely access resources available in or through your Azure VNet.
[Learn More](#)

Click here to configure

 Hybrid connections

Securely access applications in private networks
[Learn More](#)

[Configure your hybrid connection endpoints](#)

 Azure Front Door with Web Application Firewall

Scalable and secure entry point for accelerated delivery of your web applications
[Learn More](#)

[Configure Azure Front Door with WAF for your app](#)

 Azure CDN

Secure, reliable content delivery with broad global reach and rich feature set
[Learn More](#)

3. On the **VNET Integration** page, select Add VNet.

 VNet Integration
myfunctionapp

 Disconnect  Refresh

 **VNet Configuration**

Securely access resources available in or through your Azure VNet. [Learn more](#)

Add VNet

VNet Details

| | |
|-----------|----------------|
| VNet NAME | Not Configured |
| LOCATION | Not Configured |

VNet Address Space

| Start Address | End Address |
|----------------|-------------|
| Not Configured | |

4. In **Network Feature Status**, use the settings in the table below the image:

Network Feature Status

vnet-function1

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. It doesn't matter that your functions use a different subnet than your VM. The virtual network automatically routes traffic between the two subnets. |
| Subnet name | Function-Net | Name of the new subnet. |
| Virtual network address block | 10.10.0.0/16 | Choose the same address block used by the WordPress site. You should only have one address block defined. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------|-----------------|---|
| Address range | 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. |

- 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 where the WordPress site is running. Next, you use [Azure Functions Proxies](#) to return a file from the WordPress site.

Create a proxy to access VM resources

With VNet Integration enabled, you can create a proxy in your function app to forward requests to the VM running in the virtual network.

- In your function app, select **Proxies** from the left menu, and then select **Add**. Use the proxy settings in the table below the image:

New proxy

Name: Plant

Route template: /plant

Allowed HTTP methods: All methods

Backend URL: http://10.10.1.4/wp-content/themes/twentyseventeen/assets/images/header.jpg

Request override

Response override

Create

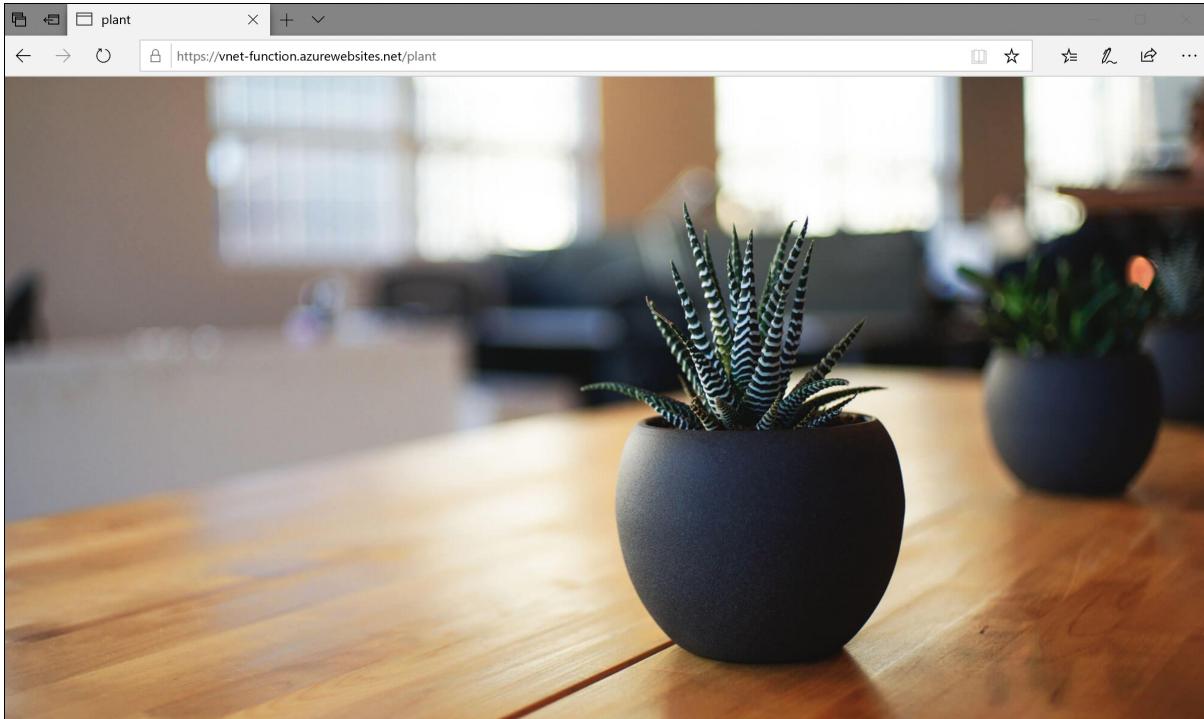
| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------|--|---|
| Name | Plant | The name can be any value. It's used to identify the proxy. |
| Route Template | /plant | Route that maps to a VM resource. |
| Backend URL | http://<YOUR_VM_IP>/wp-content/themes/twentyseventeen/assets/images/header.jpg | Replace <YOUR_VM_IP> with the IP address of your WordPress VM that you created earlier. This mapping returns a single file from the site. |

- Select **Create** to add the proxy to your function app.

Try it out

- In your browser, try to access the URL you used as the **Backend URL**. As expected, the request times out. A timeout occurs because your WordPress site is connected only to your virtual network and not the internet.

2. Copy the **Proxy URL** value from your new proxy and paste it into the address bar of your browser. The returned image is from the WordPress site running inside your virtual network.



Your function app is connected to both the internet and your virtual network. The proxy is receiving a request over the public internet, and then acting as a simple HTTP proxy to forward that request to the connected virtual network. The proxy then relays the response back to you publicly over the internet.

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, the WordPress site serves as an API that is called by using a proxy in the function app. This scenario makes a good tutorial because it's easy to set up and visualize. You could use any other API deployed within a virtual network. You could also have created a function with code that calls APIs deployed within the virtual network. A more realistic scenario is a function that uses data client APIs to call a SQL Server instance deployed in the virtual network.

Functions running in a Premium plan share the same underlying App Service infrastructure as web apps on PremiumV2 plans. All the documentation for [web apps in Azure App Service](#) applies to your Premium plan functions.

[Learn more about the networking options in Functions](#)

Tutorial: Establish Azure Functions private site access

12/4/2020 • 9 minutes to read • [Edit Online](#)

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.

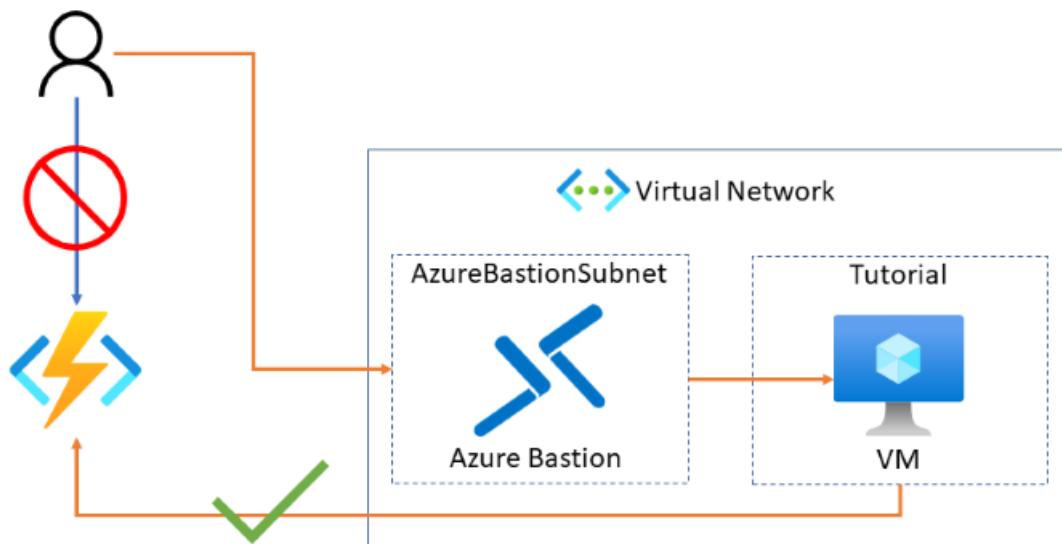
In this tutorial, you learn how to configure private site access for your function app:

- 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:

The screenshot shows the 'Create a virtual machine' Basics tab. The 'Basics' tab is selected. The configuration includes:

- Subscription:** Visual Studio Ultimate
- Resource group:** (New) myResourceGroup
- Virtual machine name:** myVM
- Region:** (US) North Central US
- Image:** Windows Server 2019 Datacenter
- Azure Spot instance:** No
- Size:** Standard DS1 v2 (1 vcpu, 3.5 GiB memory (\$53.29/month))
- Administrator account:** Username: myusername, Password: (redacted), Confirm password: (redacted)
- Inbound port rules:** Public inbound ports: None

A note at the bottom states: "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 |
|---------|-----------------|-------------|
|---------|-----------------|-------------|

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|-----------------------------|-----------------------|--|
| <i>Subscription</i> | Your subscription | The subscription under which your resources are created. |
| <i>Resource group</i> | 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. |
| <i>Virtual machine name</i> | myVM | The VM name needs to be unique in the resource group |
| <i>Region</i> | (US) North Central US | Choose a region near you or near the functions to be accessed. |
| <i>Public inbound ports</i> | 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 * | <input type="text" value="(new) myResourceGroup-vnet"/> Create new |
| Subnet * | <input type="text" value="(new) Tutorial (10.10.1.0/24)"/> |
| Public IP | <input type="text" value="None"/> Create new |
| NIC network security group | <input type="radio"/> None <input checked="" type="radio"/> Basic <input type="radio"/> Advanced |
| Public inbound ports * | <input checked="" type="radio"/> None <input type="radio"/> Allow selected ports |
| Select inbound ports | <input type="text" value="Select one or more ports"/> |
| <p>Info All traffic from the internet will be blocked by default. You will be able to change inbound port rules in the VM > Networking page.</p> | |
| Accelerated networking | <input type="radio"/> On <input checked="" type="radio"/> 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? | <input type="radio"/> Yes <input checked="" type="radio"/> 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 |
| Region | North Central US | Choose a region near you or near other services your functions access. |
| Virtual network | myResourceGroup-vnet | The virtual network in which the Bastion resource will be created in |
| Subnet | 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.

Create a Bastion

[Basics](#) [Tags](#) [Review + create](#)

Bastion allows web based RDP access to your vnet VM. [Learn more.](#)

Project details

Subscription *

Resource group * [Create new](#)

Instance details

Name * ✓

Region *

Configure virtual networks

Virtual network * [Create new](#)

Subnet * [Manage subnet configuration](#)

Public IP address

Public IP address * Create new Use existing

Public IP address name * ✓

Public IP address SKU Standard

Assignment Dynamic Static

[Review + create](#) [Previous](#) [Next : Tags >](#) [Download a template for automation](#)

- 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.

- In the portal, choose **Add** at the top of the resource group view.
- Select **Compute > Function App**
- On the *Basics* section, use the function app settings as specified in the table below.

| 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. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------------|--------------------|--|
| <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.

- 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. |

- Select **Review + Create** to review the app configuration selections.

- 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.

- Within the function app, select the **Networking** link under the *Settings* section header.
- The *Networking* page is the starting point to configure Azure Front Door, the Azure CDN, and also Access Restrictions.
- Select **Configure Access Restrictions** to configure private site access.
- 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.

- In the *Add Access Restriction* pane, provide a *Name*, *Priority*, and *Description* for the new rule.
- 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.

- 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

- Return to the previously created function app. In the *Overview* section, copy the URL.

The screenshot shows the Azure portal's Overview page for a function app named "tutorial-private-function". The URL field, which contains "https://tutorial-private-function.azurewebsites.net", is highlighted with a red box. Other visible details 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.

- 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.
- Select **Connect** and then choose **Bastion**.
- Provide the required username and password to log into the virtual machine.
- 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}".

- 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.

| Name | Trigger | Status |
|--------------------|---------|---------|
| HttpTriggerCSharp1 | HTTP | Enabled |

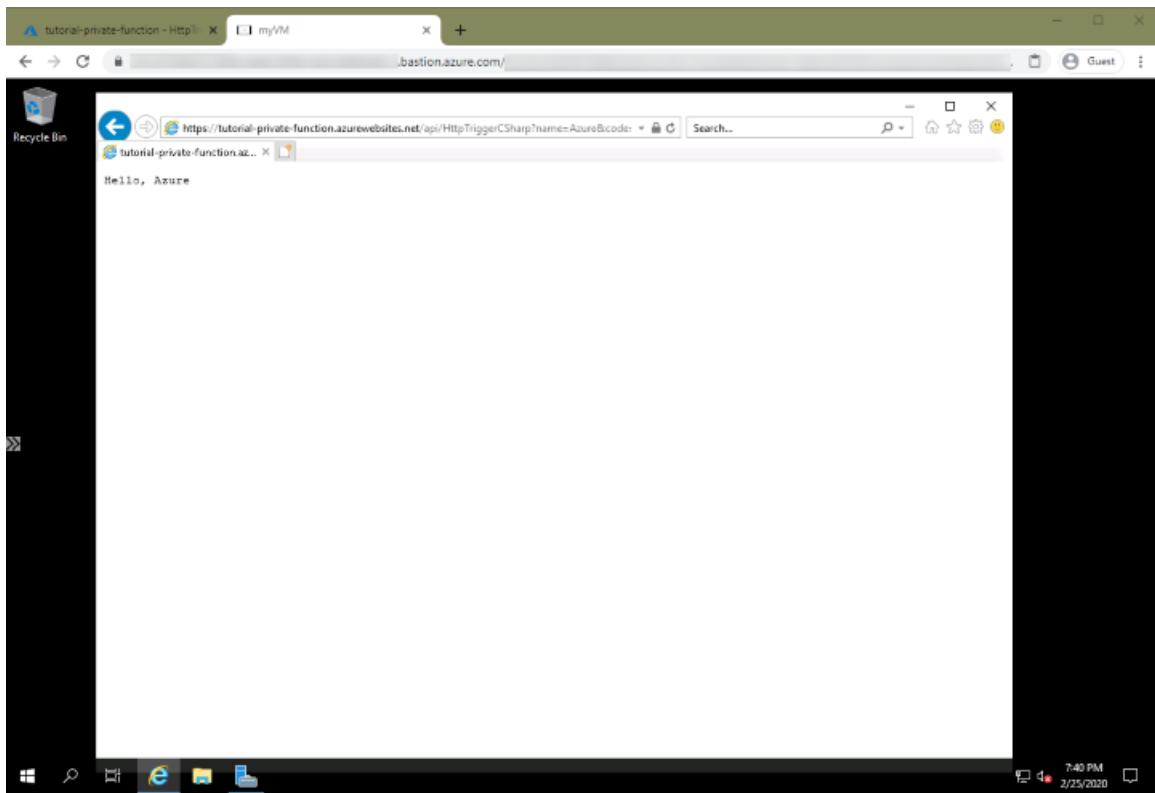
Invoke the function directly

- 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.

- 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: Automate resizing uploaded images using Event Grid

11/2/2020 • 9 minutes to read • [Edit Online](#)

Azure Event Grid is an eventing service for the cloud. Event Grid enables you to create subscriptions to events raised by Azure services or third-party resources.

This tutorial is part two of a series of Storage tutorials. It extends the [previous Storage tutorial](#) to add serverless automatic thumbnail generation using Azure Event Grid and Azure Functions. Event Grid enables [Azure Functions](#) to respond to [Azure Blob storage](#) events and generate thumbnails of uploaded images. An event subscription is created against the Blob storage create event. When a blob is added to a specific Blob storage container, a function endpoint is called. Data passed to the function binding from Event Grid is used to access the blob and generate the thumbnail image.

You use the Azure CLI and the Azure portal to add the resizing functionality to an existing image upload app.

- [.NET v12 SDK](#)
- [Node.js V10 SDK](#)

The screenshot shows a web browser window titled "Home Page - ImageResizer". The main content area has a header "ImageResizer" and a section titled "Upload photos" with a dashed blue border. Inside this border is a message "Drop files here or click to upload.". Below this is a section titled "Generated Thumbnails" containing a thumbnail image of a person working at a desk. At the bottom left, there is a note about privacy policy.

Drop files here or click to upload.

Generated Thumbnails

This app has no official privacy policy. Your data will be uploaded to a service in order to produce a picture. Your images will be public once you upload them and there is no automated way to remove them.

In this tutorial, you learn how to:

- Create an Azure Storage account
- Deploy serverless code using Azure Functions
- Create a Blob storage event subscription in Event Grid

Prerequisites

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

To complete this tutorial:

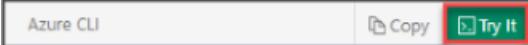
You must have completed the previous Blob storage tutorial: [Upload image data in the cloud with Azure Storage](#).

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

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

If you choose to install and use the CLI locally, this tutorial requires the Azure CLI version 2.0.14 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

If you are not using Cloud Shell, you must first sign in using `az login`.

If you've not previously registered the Event Grid resource provider in your subscription, make sure it's registered.

```
az provider register --namespace Microsoft.EventGrid
```

```
az provider register --namespace Microsoft.EventGrid
```

Create an Azure Storage account

Azure Functions requires a general storage account. In addition to the Blob storage account you created in the previous tutorial, create a separate general storage account in the resource group by using the `az storage account create` command. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

1. Set a variable to hold the name of the resource group that you created in the previous tutorial.

```
resourceGroupName="myResourceGroup"
```

```
$resourceGroupName="myResourceGroup"
```

2. Set a variable to hold the location for resources to be created.

```
location="eastus"
```

```
$location="eastus"
```

3. Set a variable for the name of the new storage account that Azure Functions requires.

```
functionstorage=<name of the storage account to be used by the function>
```

```
$functionstorage=<name of the storage account to be used by the function>"
```

4. Create the storage account for the Azure function.

```
az storage account create --name $functionstorage --location $location \
--resource-group $resourceGroupName --sku Standard_LRS --kind StorageV2
```

```
az storage account create --name $functionstorage --location $location \
--resource-group $resourceGroupName --sku Standard_LRS --kind StorageV2
```

Create a function app

You must have a function app to host the execution of your function. The function app provides an environment for serverless execution of your function code. Create a function app by using the [az functionapp create](#) command.

In the following command, provide your own unique function app name. The function app name is used as the default DNS domain for the function app, and so the name needs to be unique across all apps in Azure.

1. Specify a name for the function app that's to be created.

```
functionapp=<name of the function app>"
```

```
$functionapp=<name of the function app>"
```

2. Create the Azure function.

```
az functionapp create --name $functionapp --storage-account $functionstorage \
--resource-group $resourceGroupName --consumption-plan-location $location \
--functions-version 2
```

```
az functionapp create --name $functionapp --storage-account $functionstorage \
--resource-group $resourceGroupName --consumption-plan-location $location \
--functions-version 2
```

Now configure the function app to connect to the Blob storage account you created in the [previous tutorial](#).

Configure the function app

The function needs credentials for the Blob storage account, which are added to the application settings of the function app using the [az functionapp config appsettings set](#) command.

- [.NET v12 SDK](#)
- [Node.js V10 SDK](#)

```
storageConnectionString=$(az storage account show-connection-string --resource-group $resourceGroupName \
--name $blobStorageAccount --query connectionString --output tsv)

az functionapp config appsettings set --name $functionapp --resource-group $resourceGroupName \
--settings AzureWebJobsStorage=$storageConnectionString THUMBNAIL_CONTAINER_NAME=thumbnails \
THUMBNAIL_WIDTH=100 FUNCTIONS_EXTENSION_VERSION=~2
```

```
$storageConnectionString=$(az storage account show-connection-string --resource-group $resourceGroupName \
--name $blobStorageAccount --query connectionString --output tsv)

az functionapp config appsettings set --name $functionapp --resource-group $resourceGroupName \
--settings AzureWebJobsStorage=$storageConnectionString THUMBNAIL_CONTAINER_NAME=thumbnails \
THUMBNAIL_WIDTH=100 FUNCTIONS_EXTENSION_VERSION=~2
```

The `FUNCTIONS_EXTENSION_VERSION=~2` setting makes the function app run on version 2.x of the Azure Functions runtime.

You can now deploy a function code project to this function app.

Deploy the function code

- [.NET v12 SDK](#)
- [Node.js V10 SDK](#)

The sample C# resize function is available on [GitHub](#). Deploy this code project to the function app by using the [az functionapp deployment source config](#) command.

```
az functionapp deployment source config --name $functionapp --resource-group $resourceGroupName \
--branch master --manual-integration \
--repo-url https://github.com/Azure-Samples/function-image-upload-resize
```

```
az functionapp deployment source config --name $functionapp --resource-group $resourceGroupName \
--branch master --manual-integration \
--repo-url https://github.com/Azure-Samples/function-image-upload-resize
```

The image resize function is triggered by HTTP requests sent to it from the Event Grid service. You tell Event Grid that you want to get these notifications at your function's URL by creating an event subscription. For this tutorial you subscribe to blob-created events.

The data passed to the function from the Event Grid notification includes the URL of the blob. That URL is in turn passed to the input binding to obtain the uploaded image from Blob storage. The function generates a thumbnail image and writes the resulting stream to a separate container in Blob storage.

This project uses `EventGridTrigger` for the trigger type. Using the Event Grid trigger is recommended over generic HTTP triggers. Event Grid automatically validates Event Grid Function triggers. With generic HTTP triggers, you must implement the [validation response](#).

- .NET v12 SDK
- Node.js V10 SDK

To learn more about this function, see the [function.json](#) and [run.csx](#) files.

The function project code is deployed directly from the public sample repository. To learn more about deployment options for Azure Functions, see [Continuous deployment for Azure Functions](#).

Create an event subscription

An event subscription indicates which provider-generated events you want sent to a specific endpoint. In this case, the endpoint is exposed by your function. Use the following steps to create an event subscription that sends notifications to your function in the Azure portal:

1. In the [Azure portal](#), at the top of the page search for and select **Function App** and choose the function app that you just created. Select **Functions** and choose the **Thumbnail** function.

The screenshot shows the Azure portal's Functions blade. On the left, there's a sidebar with 'Functions' selected, and under it, 'App keys', 'App files', and 'Proxies'. On the right, a list of functions is shown with columns for Name, Trigger, and Status. The 'Thumbnail' function is listed, with its name and trigger type ('EventGrid') also highlighted with a red box.

2. Select **Integration** then choose the **Event Grid Trigger** and select **Create Event Grid subscription**.

The screenshot shows the 'Thumbnail | Integration' blade. On the left, the 'Developer' menu has 'Integration' selected. The main area shows the 'Integration' section with a note to edit triggers. Under 'Trigger', 'Event Grid Trigger (eventGridEvent)' is selected and highlighted with a red box. On the right, an 'Edit Trigger' pane is open with fields for 'Binding Type' (set to 'Event Grid Trigger'), 'Event Trigger parameter name*' (set to 'eventGridEvent'), and a 'Create Event Grid subscription' button highlighted with a red box.

3. Use the event subscription settings as specified in the table.



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

imageresizesub



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

[spblogstorageaccount0608 \(change\)](#)

System Topic Name

imagestoragesystopic



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

Azure Function [\(change\)](#)

Endpoint

[Thumbnail \(change\)](#)

Create

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|-------------------|---------------------------|---|
| Name | imageresizersub | Name that identifies your new event subscription. |
| Topic type | Storage accounts | Choose the Storage account event provider. |
| Subscription | Your Azure subscription | By default, your current Azure subscription is selected. |
| Resource group | myResourceGroup | Select Use existing and choose the resource group you have been using in this tutorial. |
| Resource | Your Blob storage account | Choose the Blob storage account you created. |
| System Topic Name | imagestoragesystopic | Specify a name for the system topic. To learn about system topics, see System topics overview . |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------|-----------------|---|
| Event types | Blob created | Uncheck all types other than Blob created . Only event types of <code>Microsoft.Storage.BlobCreated</code> are passed to the function. |
| Endpoint type | autogenerated | Pre-defined as Azure Function . |
| Endpoint | autogenerated | Name of the function. In this case, it's Thumbnail . |

4. Switch to the **Filters** tab, and do the following actions:

- a. Select **Enable subject filtering** option.
- b. For **Subject begins with**, enter the following value :
`/blobServices/default/containers/images/blobs/`.

Create Event Subscription
Event Grid

Basic **Filters** **Additional Features**

SUBJECT FILTERS
Apply filters to the subject of each event. Only events with matching subjects get delivered. [Learn more](#)

Enable subject filtering

Subject Begins With

Subject Ends With

Case-sensitive subject matching

ADVANCED FILTERS
Filter on attributes of each event. Only events that match all filters get delivered. Up to 5 filters can be specified. All string comparisons are case-insensitive. [Learn more](#)

Valid keys for currently selected event schema:

- id, topic, subject, eventtype, dataversion
- Custom properties at most one level inside the data payload, using "." as the nesting separator. (e.g. data, data.key are valid, data.key.key is not)

| KEY | OPERATOR | VALUE |
|--------------------------------|----------|-------|
| No results | | |
| Add new filter | | |

Create

5. Select **Create** to add the event subscription. This creates an event subscription that triggers the `Thumbnail` function when a blob is added to the `images` container. The function resizes the images and adds them to the `thumbnails` container.

Now that the backend services are configured, you test the image resize functionality in the sample web app.

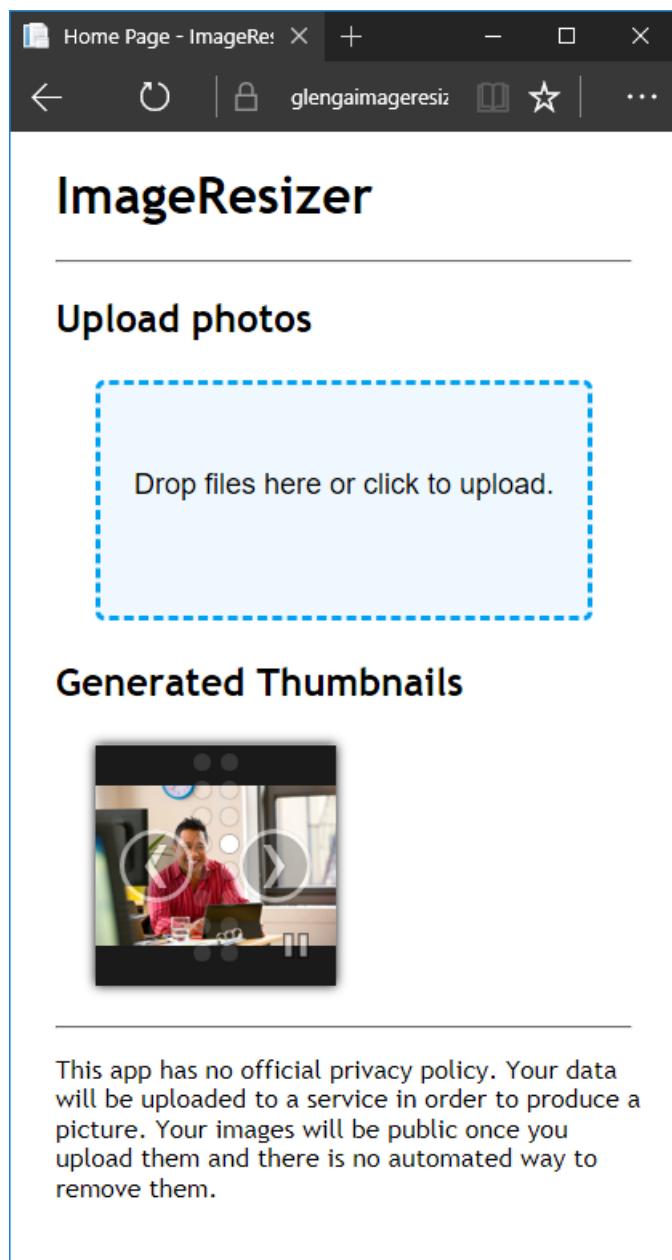
Test the sample app

To test image resizing in the web app, browse to the URL of your published app. The default URL of the web app is
`https://<web_app>.azurewebsites.net`.

- .NET v12 SDK
- Node.js V10 SDK

Click the **Upload photos** region to select and upload a file. You can also drag a photo to this region.

Notice that after the uploaded image disappears, a copy of the uploaded image is displayed in the **Generated Thumbnails** carousel. This image was resized by the function, added to the *thumbnails* container, and downloaded by the web client.



Next steps

In this tutorial, you learned how to:

- Create a general Azure Storage account
- Deploy serverless code using Azure Functions
- Create a Blob storage event subscription in Event Grid

Advance to part three of the Storage tutorial series to learn how to secure access to the storage account.

[Secure access to an applications data in the cloud](#)

- To learn more about Event Grid, see [An introduction to Azure Event Grid](#).

- To try another tutorial that features Azure Functions, see [Create a function that integrates with Azure Logic Apps](#).

Tutorial: Apply machine learning models in Azure Functions with Python and TensorFlow

12/4/2020 • 8 minutes to read • [Edit Online](#)

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`

- [PowerShell](#)
- [Cmd](#)

```
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:

```
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](#)
- [PowerShell](#)
- [Cmd](#)

```
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](#)
- [PowerShell](#)
- [Cmd](#)

```
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:

```
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 MutableDenseHashTable.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:

```
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:

```
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](#)
- [PowerShell](#)
- [Cmd](#)

```
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.

Dog or Cat?

<https://raw.githubusercontent.com/Azure-Samples/fun>



Cat



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

12/4/2020 • 7 minutes to read • [Edit Online](#)

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 is 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](#)
- [PowerShell](#)

- [Cmd](#)

```
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:

```
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` 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`.

```
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
```

- 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.

- 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.

- In the `start` folder, run the following command to copy the prediction code and labels into the `classify` folder.

- [bash](#)
- [PowerShell](#)
- [Cmd](#)

```
cp ../resources/predict.py classify  
cp ../resources/labels.txt classify
```

- 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.
- Open `start/requirements.txt` in a text editor and add the dependencies required by the helper code, which should look like the following:

```
azure-functions  
requests  
-f https://download.pytorch.org/whl/torch_stable.html  
torch==1.5.0+cpu  
torchvision==0.6.0+cpu
```

- 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

- 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:

```
import logging
import azure.functions as func
import json

# Import helper script
from .predict import predict_image_from_url
```

- Replace the entire contents of the `main` function with the following code:

```
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.

- 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
```

- 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
```

- 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.

- Open a new terminal or command prompt and activate the virtual environment (as described earlier under [Create and activate a Python virtual environment](#)).
- Navigate to the repository's `frontend` folder.

3. Start an HTTP server with Python:

- [bash](#)
- [PowerShell](#)
- [Cmd](#)

```
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/fun
```

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

Create a function on Linux using a custom container

12/4/2020 • 28 minutes to read • [Edit Online](#)

In this tutorial, you create and deploy your code to Azure Functions as a custom Docker container using a Linux base image. You typically use a custom image when your functions require a specific language version or have a specific dependency or configuration that isn't provided by the built-in image.

Deploying your function code in a custom Linux container requires [Premium plan](#) or a [Dedicated \(App Service\) plan](#) hosting. Completing this tutorial incurs costs of a few US dollars in your Azure account, which you can minimize by [cleaning-up resources](#) when you're done.

You can also use a default Azure App Service container as described on [Create your first function hosted on Linux](#). Supported base images for Azure Functions are found in the [Azure Functions base images repo](#).

In this tutorial, you learn how to:

- Create a function app and Dockerfile using the Azure Functions Core Tools.
- Build a custom image using Docker.
- Publish a custom image to a container registry.
- Create supporting resources in Azure for the function app
- Deploy a function app from Docker Hub.
- Add application settings to the function app.
- Enable continuous deployment.
- Enable SSH connections to the container.
- Add a Queue storage output binding.

You can follow this tutorial on any computer running Windows, macOS, or Linux.

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 Functions Core Tools](#) version 2.7.1846 or a later 2.x version.
- The Azure Functions Core Tools version that corresponds to your installed Python version:

| PYTHON VERSION | CORE TOOLS VERSION |
|--------------------------|---|
| Python 3.8 | version 3.x |
| Python 3.6 Python 3.7 | Version 2.7.1846 or a later version |

- The [Azure CLI](#) version 2.4 or later.
- [Node.js](#), Active LTS and Maintenance LTS versions (8.11.1 and 10.14.1 recommended).
- [Python 3.8 \(64-bit\)](#), [Python 3.7 \(64-bit\)](#), [Python 3.6 \(64-bit\)](#), which are supported by Azure Functions.
- The [.NET Core SDK 3.1](#)

- The [Java Developer Kit](#), version 8 or 11.
- [Apache Maven](#), version 3.0 or above.
- [Docker](#)
- A [Docker ID](#)

Prerequisite check

- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 2.7.1846 or later.
- Run `az --version` to check that the Azure CLI version is 2.0.76 or later.
- Run `az login` to sign in to Azure and verify an active subscription.
- Run `python --version` (Linux/MacOS) or `py --version` (Windows) to check your Python version reports 3.8.x, 3.7.x or 3.6.x.
- Run `docker login` to sign in to Docker. This command fails if Docker isn't running, in which case start docker and retry the command.

Create and activate a virtual environment

In a suitable folder, run the following commands to create and activate a virtual environment named `.venv`. Be sure to use Python 3.8, 3.7 or 3.6, which are supported by Azure Functions.

- [bash](#)
- [PowerShell](#)
- [Cmd](#)

```
python -m venv .venv
```

```
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment.

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 a folder named `LocalFunctionsProject`.

```
func init LocalFunctionsProject --worker-runtime dotnet --docker
```

```
func init LocalFunctionsProject --worker-runtime node --language javascript --docker
```

```
func init LocalFunctionsProject --worker-runtime powershell --docker
```

```
func init LocalFunctionsProject --worker-runtime python --docker
```

```
func init LocalFunctionsProject --worker-runtime node --language typescript --docker
```

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

- [Bash](#)
- [PowerShell](#)
- [Cmd](#)

```
mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure -DarchetypeArtifactId=azure-functions-archetype  
-DjavaVersion=8 -Ddocker
```

The `-DjavaVersion` parameter tells the Functions runtime which version of Java to use. Use `-DjavaVersion=11` if you want your functions to run on Java 11. When you don't specify `-DjavaVersion`, Maven defaults to Java 8. For more information, see [Java versions](#).

IMPORTANT

The `JAVA_HOME` environment variable must be set to the install location of the correct version of the JDK to complete this article.

Maven asks you for values needed to finish generating the project on deployment.

Provide the following values when prompted:

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

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`.

The `--docker` option generates a `Dockerfile` for the project, which defines a suitable custom container for use with Azure Functions and the selected runtime.

Navigate into the project folder:

```
cd LocalFunctionsProject
```

```
cd fabrikam-functions
```

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` 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`.

```
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:

```
func start --build
```

```
func start
```

```
npm install  
npm start
```

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

Once you see the `HttpExample` endpoint appear in the output, navigate to

`http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a "hello" message that echoes back `Functions`, the value supplied to the `name` query parameter.

Use `Ctrl-C` to stop the host.

Build the container image and test 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, and provide a name, `azurefunctionsimage`, and tag, `v1.0.0`. Replace `<DOCKER_ID>` with your Docker Hub account ID. This command builds the Docker image for the container.

```
docker build --tag <DOCKER_ID>/azurefunctionsimage:v1.0.0 .
```

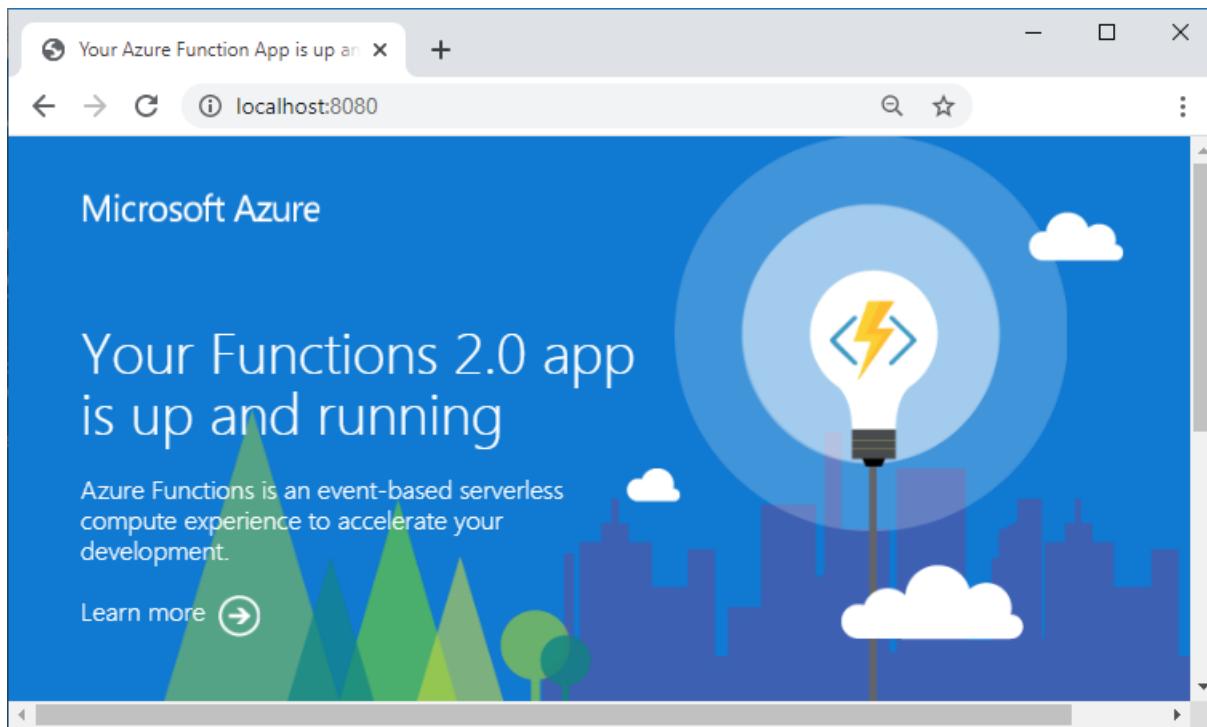
When the command completes, you can run the new container locally.

To test the build, run the image in a local container using the `docker run` command, replacing again `<DOCKER_ID>` with your Docker ID and adding the ports argument, `-p 8080:80`:

```
docker run -p 8080:80 -it <docker_id>/azurefunctionsimage:v1.0.0
```

Once the image is running in a local container, open a browser to `http://localhost:8080`, which should display the placeholder image shown below. The image appears at this point because your function is running in the local container, as it would in Azure, which means that it's protected by an access key as defined in `function.json` with

the `"authLevel": "function"` property. The container hasn't yet been published to a function app in Azure, however, so the key isn't yet available. If you want to test against the local container, stop docker, change the authorization property to `"authLevel": "anonymous"`, rebuild the image, and restart docker. Then reset `"authLevel": "function"` in *function.json*. For more information, see [authorization keys](#).



Once the image is running in a local container, browse to <http://localhost:8080/api/HttpExample?name=Functions>, which should display the same "hello" message as before. Because the Maven archetype generates an HTTP triggered function that uses anonymous authorization, you can still call the function even though it's running in the container.

After you've verified the function app in the container, stop docker with **Ctrl+C**.

Push the image to Docker Hub

Docker Hub is a container registry that hosts images and provides image and container services. To share your image, which includes deploying to Azure, you must push it to a registry.

1. If you haven't already signed in to Docker, do so with the [docker login](#) command, replacing `<docker_id>` with your Docker ID. This command prompts you for your username and password. A "Login Succeeded" message confirms that you're signed in.

```
docker login
```

2. After you've signed in, push the image to Docker Hub by using the [docker push](#) command, again replacing `<docker_id>` with your Docker ID.

```
docker push <docker_id>/azurefunctionsimage:v1.0.0
```

3. Depending on your network speed, pushing the image the first time might take a few minutes (pushing subsequent changes is much faster). While you're waiting, you can proceed to the next section and create Azure resources in another terminal.

Create supporting Azure resources for your function

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

- A resource group, which is a logical container for related resources.
- An Azure Storage account, which maintains state and other information about your projects.
- An Azure functions 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.

You use Azure CLI commands to create these items. Each command provides JSON output upon completion.

1. Sign in to Azure with the [az login](#) command:

```
az login
```

2. Create a resource group with the [az group create](#) command. The following example creates a resource group named `AzureFunctionsContainers-rg` in the `westeurope` region. (You generally create your resource group and resources in a region near you, using an available region from the [az account list-locations](#) command.)

```
az group create --name AzureFunctionsContainers-rg --location westeurope
```

NOTE

You can't host Linux and Windows apps in the same resource group. If you have an existing resource group named `AzureFunctionsContainers-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 by using the [az storage account create](#) command. In the following example, replace `<storage_name>` with a globally unique name appropriate to you. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a typical general-purpose account.

```
az storage account create --name <storage_name> --location westeurope --resource-group AzureFunctionsContainers-rg --sku Standard_LRS
```

The storage account incurs only a few USD cents for this tutorial.

4. Use the command to create a Premium plan for Azure Functions named `myPremiumPlan` in the **Elastic Premium 1** pricing tier (`--sku EP1`), in the West Europe region (`-location westeurope`, or use a suitable region near you), and in a Linux container (`--is-linux`).

```
az functionapp plan create --resource-group AzureFunctionsContainers-rg --name myPremiumPlan --location westeurope --number-of-workers 1 --sku EP1 --is-linux
```

We use the Premium plan here, which can scale as needed. To learn more about hosting, see [Azure Functions hosting plans comparison](#). To calculate costs, see the [Functions pricing page](#).

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.

Create and configure a function app on Azure with the image

A function app on Azure manages the execution of your functions in your hosting plan. In this section, you use the Azure resources from the previous section to create a function app from an image on Docker Hub and configure it with a connection string to Azure Storage.

1. Create the Functions app using the [az functionapp create](#) command. In the following 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>` with your Docker ID.

```
az functionapp create --name <app_name> --storage-account <storage_name> --resource-group AzureFunctionsContainers-rg --plan myPremiumPlan --runtime <functions runtime stack> --deployment-container-image-name <docker_id>/azurefunctionsimage:v1.0.0
```

The `deployment-container-image-name` parameter specifies the image to use for the function app. You can use the [az functionapp config container show](#) command to view information about the image used for deployment. You can also use the [az functionapp config container set](#) command to deploy from a different image.

2. Retrieve the connection string for the storage account you created by using the [az storage account show-connection-string](#) command, assigning it to a shell variable `storageConnectionString`:

```
az storage account show-connection-string --resource-group AzureFunctionsContainers-rg --name <storage_name> --query connectionString --output tsv
```

3. Add this setting to the function app by using the [az functionapp config appsettings set](#) command. In the following command, replace `<app_name>` with the name of your function app, and replace `<connection_string>` with the connection string from the previous step (a long encoded string that begins with "DefaultEndpointProtocol="):

```
az functionapp config appsettings set --name <app_name> --resource-group AzureFunctionsContainers-rg --settings AzureWebJobsStorage=<connection_string>
```

4. The function can now use this connection string to access the storage account.

TIP

In Bash, you can use a shell variable to capture the connection string instead of using the clipboard. First, use the following command to create a variable with the connection string:

```
storageConnectionString=$(az storage account show-connection-string --resource-group AzureFunctionsContainers-rg --name <storage_name> --query connectionString --output tsv)
```

Then refer to the variable in the second command:

```
az functionapp config appsettings set --name <app_name> --resource-group AzureFunctionsContainers-rg --settings AzureWebJobsStorage=$storageConnectionString
```

NOTE

If you publish your custom image to a private container account, you should use environment variables in the Dockerfile for the connection string instead. For more information, see the [ENV instruction](#). You should also set the variables `DOCKER_REGISTRY_SERVER_USERNAME` and `DOCKER_REGISTRY_SERVER_PASSWORD`. To use the values, then, you must rebuild the image, push the image to the registry, and then restart the function app on Azure.

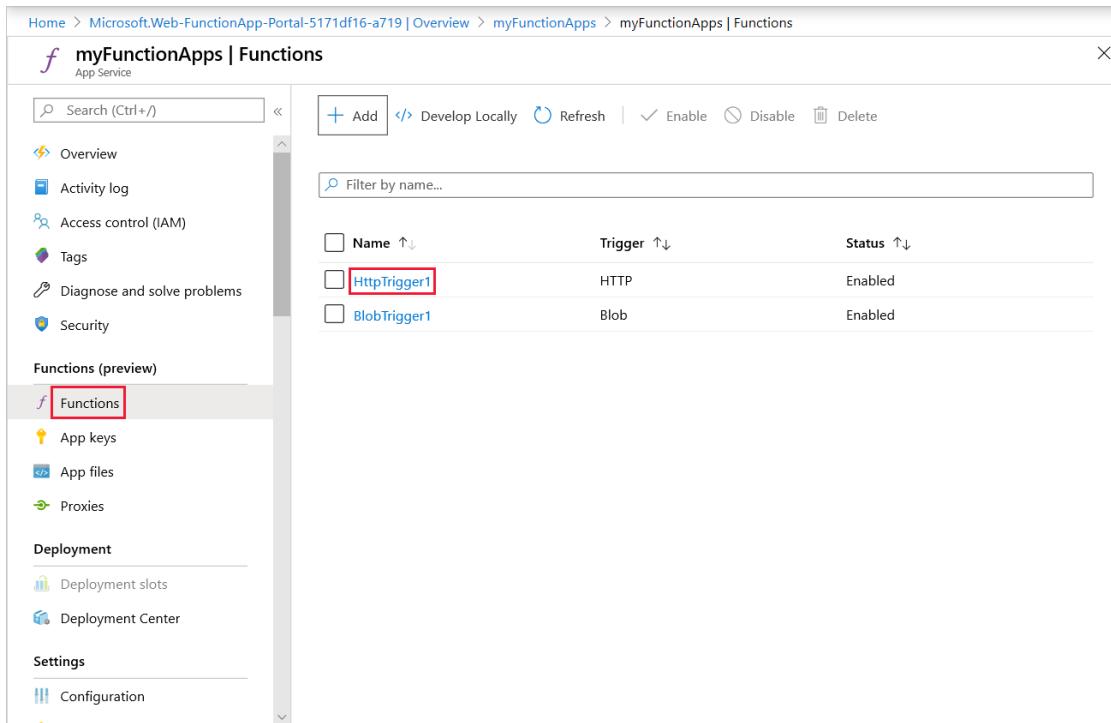
Verify your functions on Azure

With the image deployed to the function app on Azure, you can now invoke the function through HTTP requests. Because the `function.json` definition includes the property `"authLevel": "function"`, you must first obtain the access key (also called the "function key") and include it as a URL parameter in any requests to the endpoint.

1. Retrieve the function URL with the access (function) key by using the Azure portal, or by using the Azure CLI with the `az rest` command.)

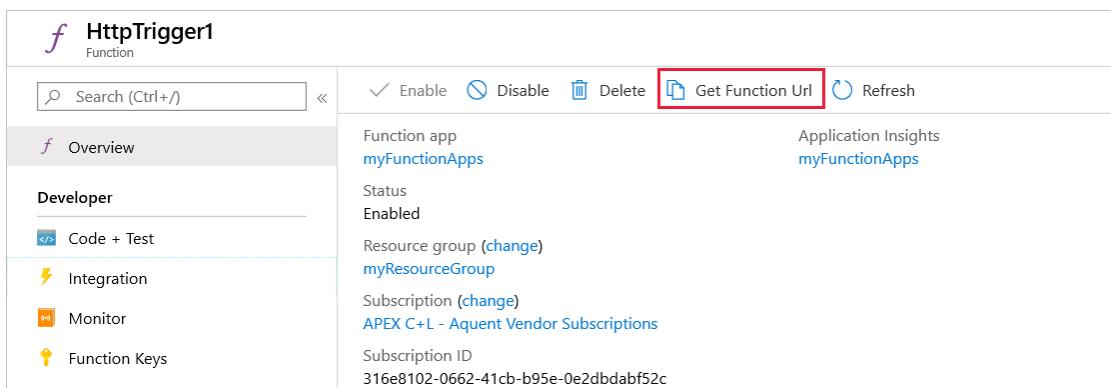
- [Portal](#)
- [Azure CLI](#)

1. Sign in to the Azure portal, then search for and select **Function App**.
2. Select the function you want to verify.
3. In the left navigation panel, select **Functions**, and then select the function you want to verify.



| Name | Trigger | Status |
|--------------|---------|---------|
| HttpTrigger1 | HTTP | Enabled |
| BlobTrigger1 | Blob | Enabled |

4. Select **Get Function Url**.



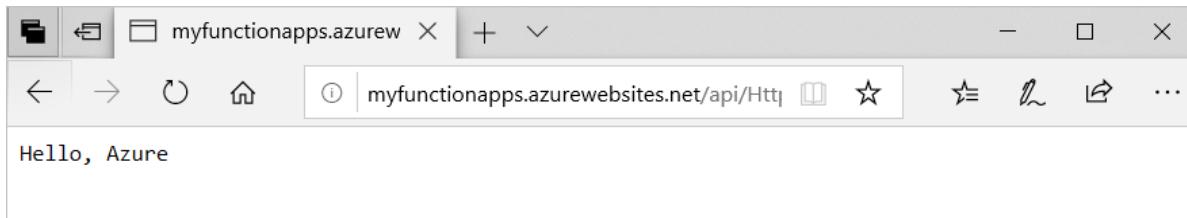
5. In the pop-up window, select **default (function key)** and then copy the URL to the clipboard. The key is the string of characters following `?code=`.



NOTE

Because your function app is deployed as a container, you can't make changes to your function code in the portal. You must instead update the project in the local image, push the image to the registry again, and then redeploy to Azure. You can set up continuous deployment in a later section.

2. Paste the function URL into your browser's address bar, adding the parameter `&name=Azure` to the end of this URL. Text like "Hello, Azure" should appear in the browser.



3. To test authorization, remove the `code=` parameter from the URL and verify that you get no response from the function.

Enable continuous deployment to Azure

You can enable Azure Functions to automatically update your deployment of an image whenever you update the image in the registry.

1. Enable continuous deployment by using `az functionapp deployment container config` command, replacing `<app_name>` with the name of your function app:

```
az functionapp deployment container config --enable-cd --query CI_CD_URL --output tsv --name <app_name>
--resource-group AzureFunctionsContainers-rg
```

This command enables continuous deployment and returns the deployment webhook URL. (You can retrieve this URL at any later time by using the `az functionapp deployment container show-cd-url` command.)

2. Copy the deployment webhook URL to the clipboard.
3. Open [Docker Hub](#), sign in, and select **Repositories** on the nav bar. Locate and select image, select the **Webhooks** tab, specify a **Webhook name**, paste your URL in **Webhook URL**, and then select **Create**:

The screenshot shows the Docker Hub interface for a repository named 'kraigb / azurefunctionsimage'. The 'Webhooks' tab is selected and highlighted with a red box. Below the tabs, there's a section titled 'Webhooks' with a brief description: 'A webhook is an HTTP call-back triggered by a specific event. You can create a single webhook to start and connect multiple webhooks to further build out your workflow.' It also mentions that 'When an image is pushed to this repo, your workflows will kick off based on your specified webhooks.' A link to 'Learn More' is provided. At the bottom of this section is a 'New Webhook' button. Below it, a 'ContinuousDeployment' webhook is listed with its URL: 'https://\$msdocsfunctionscontainer:ArMheeqTlhz70Kwl'. A blue 'Create' button is to the right of the URL.

- With the webhook set, Azure Functions redeloys your image whenever you update it in Docker Hub.

Enable SSH connections

SSH enables secure communication between a container and a client. With SSH enabled, you can connect to your container using App Service Advanced Tools (Kudu). To make it easy to connect to your container using SSH, Azure Functions provides a base image that has SSH already enabled. You need only edit your Dockerfile, then rebuild and redeploy the image. You can then connect to the container through the Advanced Tools (Kudu).

- In your Dockerfile, append the string `-appservice` to the base image in your `FROM` instruction:

```
FROM mcr.microsoft.com/azure-functions/dotnet:3.0-appservice
```

```
FROM mcr.microsoft.com/azure-functions/node:2.0-appservice
```

```
FROM mcr.microsoft.com/azure-functions/powershell:2.0-appservice
```

```
FROM mcr.microsoft.com/azure-functions/python:2.0-python3.7-appservice
```

```
FROM mcr.microsoft.com/azure-functions/node:2.0-appservice
```

- Rebuild the image by using the `docker build` command again, replacing `<docker_id>` with your Docker ID:

```
docker build --tag <docker_id>/azurefunctionsimage:v1.0.0 .
```

- Push the updated image to Docker Hub, which should take considerably less time than the first push only the updated segments of the image need to be uploaded.

```
docker push <docker_id>/azurefunctionsimage:v1.0.0
```

- Azure Functions automatically redeploys the image to your functions app; the process takes place in less than a minute.

- In a browser, open `https://<app_name>.scm.azurewebsites.net/`, replacing `<app_name>` with your unique

name. This URL is the Advanced Tools (Kudu) endpoint for your function app container.

6. Sign in to your Azure account, and then select the SSH to establish a connection with the container. Connecting may take a few moments if Azure is still updating the container image.
7. After a connection is established with your container, run the `top` command to view the currently running processes.

```
top - 22:05:11 up 22:51, 2 users, load average: 0.26, 0.34, 0.30
Tasks: 10 total, 1 running, 9 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3.0 us, 2.0 sy, 0.0 ni, 79.2 id, 15.8 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3512112 total, 189752 free, 909172 used, 2413188 buff/cache
KiB Swap: 1910780 total, 1867300 free, 43480 used. 2261580 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 1 root 20 0 21392 3104 2820 S 0.0 0.1 0:00.03 bash
 18 root 20 0 69960 3024 2256 S 0.0 0.1 0:00.00 sshd
 19 root 20 0 7163216 155468 66032 S 0.0 4.4 0:06.85 Microsoft.+
 49 root 20 0 722064 33228 12336 S 0.0 0.9 0:00.62 python
 74 root 20 0 69960 5876 5068 S 0.0 0.2 0:00.04 sshd
 76 root 20 0 18172 3252 2732 S 0.0 0.1 0:00.00 bash
 86 root 20 0 41016 3080 2624 S 0.0 0.1 0:00.01 top
 88 root 20 0 69960 6156 5348 S 0.0 0.2 0:00.01 sshd
 90 root 20 0 18172 3100 2688 S 0.0 0.1 0:00.00 bash
 95 root 20 0 41020 3080 2624 R 0.0 0.1 0:00.00 top
```

Write to an Azure Storage queue

Azure Functions lets you connect your functions to other Azure services and resources 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 section shows you how to integrate your function 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.

Retrieve the Azure Storage connection string

Earlier, you created an Azure Storage account for use by the function app. 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 that connection 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, replacing `<app_name>` with the name of your function app from the previous quickstart. This command will overwrite any existing values in the file.

```
func azure functionapp fetch-app-settings <app_name>
```

2. Open `local.settings.json` 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 `local.settings.json` 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

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage --version 3.0.4
```

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 let 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:

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "res"
  }
]
```

```
"scriptFile": "__init__.py",
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  }
]
```

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "Response"
  }
]
```

Each binding has at least a type, a direction, and a name. In the example above, the first binding is of type `httpTrigger` with the direction `in`. For the `in` direction, `name` specifies the name of an input parameter that's sent to the function when invoked by the trigger.

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:

```

{
  "authLevel": "function",
  "type": "httpTrigger",
  "direction": "in",
  "name": "req",
  "methods": [
    "get",
    "post"
  ],
},
{
  "type": "http",
  "direction": "out",
  "name": "res"
},
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
]
}

```

The second binding in the collection is of type `http` with the direction `out`, in which case the special `name` of `$return` indicates that this binding uses the function's return value rather than providing an input parameter.

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:

```

"bindings": [
{
  "authLevel": "anonymous",
  "type": "httpTrigger",
  "direction": "in",
  "name": "req",
  "methods": [
    "get",
    "post"
  ]
},
{
  "type": "http",
  "direction": "out",
  "name": "$return"
},
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
]

```

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:

```
{
  "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"
}
]
```

In this case, `msg` is given to the function as an output argument. For a `queue` type, you must also specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from `local.settings.json`) in `connection`.

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `storageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `storageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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 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_init_.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```
import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )
```

The `msg` parameter is an instance of the `azure.functions.InputStream class`. Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};
```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```
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
})
```

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```
if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}
```

At this point, your function should look as follows:

```

[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

```
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:

```

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.

```
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);
```

Update the image in the registry

1. In the root folder, run `docker build` again, and this time update the version in the tag to `v1.0.1`. As before, replace `<docker_id>` with your Docker Hub account ID:

```
docker build --tag <docker_id>/azurefunctionsimage:v1.0.1 .
```

2. Push the updated image back to the repository with `docker push`:

```
docker push <docker_id>/azurefunctionsimage:v1.0.1
```

3. Because you configured continuous delivery, updating the image in the registry again automatically updates your function app in Azure.

View the message in the Azure Storage queue

In a browser, use the same URL as before to invoke your function. The browser should display the same response as before, because you didn't modify that part of the function code. The added code, however, wrote a message using the `name` URL parameter to the `outqueue` 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`, pasting 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](#)
- [PowerShell](#)
- [Azure CLI](#)

```
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 should include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
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 first name you used when testing the function earlier. The command reads and removes the first message from

the queue.

- [bash](#)
- [PowerShell](#)
- [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.

Clean up resources

If you want to continue working with Azure Function using the resources you created in this tutorial, 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 `AzureFunctionsContainer-rg` resource group to clean up all the resources in that group:

```
az group delete --name AzureFunctionsContainer-rg
```

Next steps

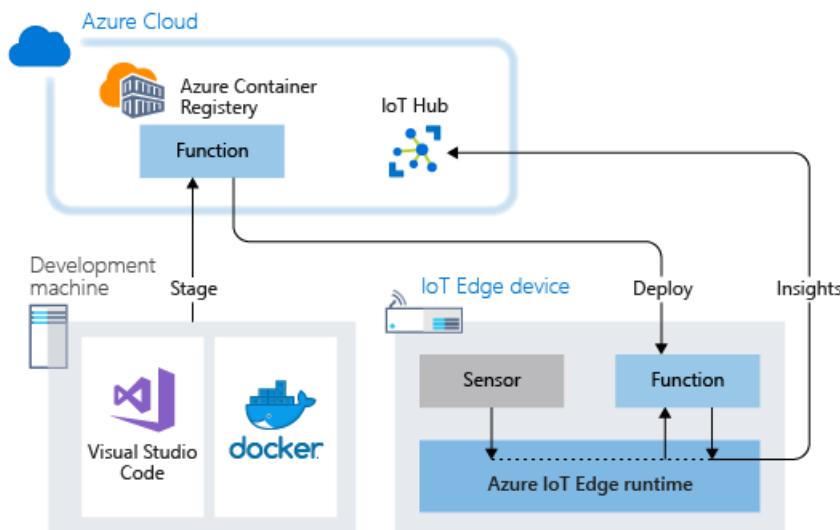
- [Monitoring functions](#)
- [Scale and hosting options](#)
- [Kubernetes-based serverless hosting](#)

Tutorial: Deploy Azure Functions as IoT Edge modules

12/4/2020 • 9 minutes to read • [Edit Online](#)

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 Deploy Azure IoT Edge on a simulated device on [Windows](#) or [Linux](#) quickstarts. In this tutorial, you learn how to:

- Use Visual Studio Code to create an Azure Function.
- Use VS 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 a [free account](#) before you begin.

Prerequisites

Before beginning this tutorial, you should have gone through the previous tutorial to set up your development environment for Linux container development: [Develop IoT Edge modules for Linux devices](#). By completing that tutorial, you should have the following prerequisites in place:

- A free or standard-tier [IoT Hub](#) in Azure.
- A [Linux device running Azure IoT Edge](#)
- A container registry, like [Azure Container Registry](#).
- [Visual Studio Code](#) configured with the [Azure IoT Tools](#).
- [Docker CE](#) configured to run Linux containers.

To develop an IoT Edge module in with Azure Functions, install the following additional prerequisites on your development machine:

- [C# for Visual Studio Code \(powered by OmniSharp\) extension](#).

- [The .NET Core 2.1 SDK](#).

Create a function project

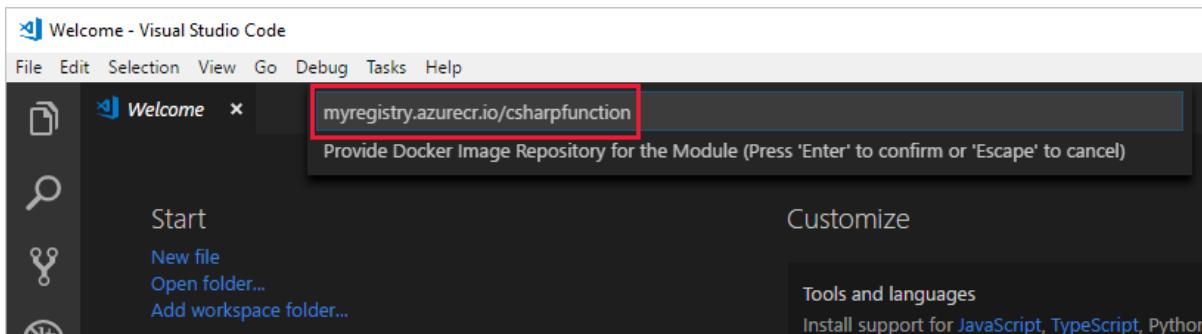
The Azure IoT Tools for Visual Studio Code that you installed in the prerequisites provides management capabilities as well as some code templates. In this section, you use Visual Studio Code to create an IoT Edge solution that contains an Azure Function.

Create a new project

Create a C# Function solution template that you can customize with your own code.

1. Open Visual Studio Code on your development machine.
2. Open the VS Code command palette by selecting **View > Command Palette**.
3. In the command palette, enter and run the command **Azure IoT Edge: New IoT Edge solution**. Follow the prompts in the command palette to create your solution.

| FIELD | VALUE |
|--|--|
| Select folder | Choose the location on your development machine for VS Code to create the solution files. |
| Provide a solution name | Enter a descriptive name for your solution, like FunctionSolution , or accept the default. |
| Select module template | Choose Azure Functions - C# . |
| Provide a module name | Name your module CSharpFunction . |
| Provide 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 prepopulated 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 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 the IoT Edge device.

The IoT Edge extension tries to pull your container registry credentials from Azure and populate them in the environment file. Check to see if your credentials are already included. If not, add them now:

1. In the VS Code explorer, open the **.env** file.

2. Update the fields with the **username** and **password** values that you copied from your Azure container registry.
3. Save this file.

Select your target architecture

Currently, Visual Studio Code can develop C modules for Linux AMD64 and Linux ARM32v7 devices. You need to select which architecture you're targeting with each solution, because the container is built and run differently for each architecture type. The default is Linux AMD64.

1. Open the command palette and search for **Azure IoT Edge: Set Default Target Platform for Edge Solution**, or select the shortcut icon in the side bar at the bottom of the window.
2. In the command palette, select the target architecture from the list of options. For this tutorial, we're using an Ubuntu virtual machine as the IoT Edge device, so will keep the default **amd64**.

Update the module with custom code

Let's add some additional code so that the module processes the messages at the edge before forwarding them to IoT Hub.

1. In Visual Studio Code, 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.

```
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.Extensions.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)
                        
```

```

        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;}
}
}
}

```

- 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.

- Open the VS Code integrated terminal by selecting **View > Terminal**.
- 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.

```
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.

- In the VS 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.

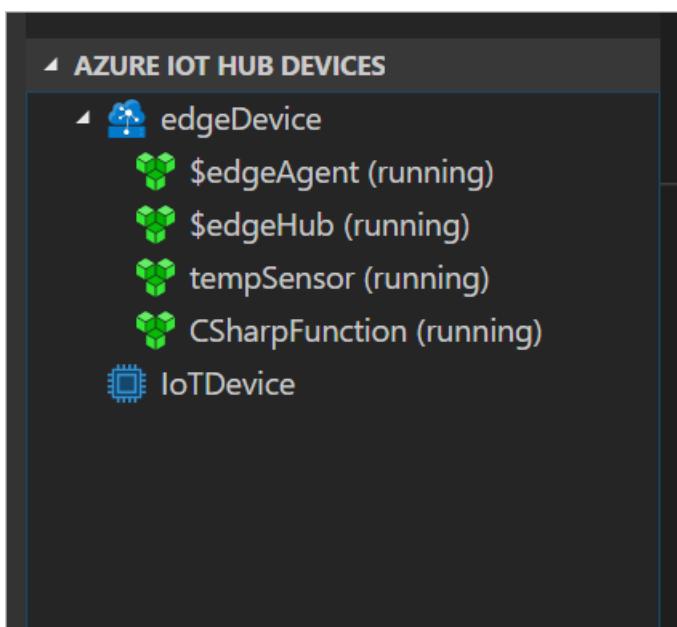
1. In the Azure portal, browse to your Azure container registry.
2. Select **Repositories**.
3. You should see the **csharpfunction** repository in the list. Select this repository to see more details.
4. In the **Tags** section, you should see the **0.0.1-amd64** tag. This tag indicates the version 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 quickstarts. You can also deploy and monitor modules from within Visual Studio Code. The following sections use the Azure IoT Tools for VS Code that was listed in the prerequisites. Install the extension now, if you didn'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. Click 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 by running **Azure IoT Hub: Start 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 and select **Start Monitoring Built-in Event Endpoint**.

To stop monitoring messages, run the command **Azure IoT Hub: Stop Monitoring Built-in Event Endpoint** in the command palette.

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, instead of deleting the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#) and select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources 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. When you're ready to build your own modules, you can learn more about how to [Develop with Azure IoT Edge for Visual Studio Code](#).

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](#)

Tutorial: Create a function in Java with an Event Hub trigger and an Azure Cosmos DB output binding

12/4/2020 • 13 minutes to read • [Edit Online](#)

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.

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 a [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 CLI](#) if you prefer not to use Cloud Shell
- [Azure Functions Core Tools](#) version 2.6.666 or above

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.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

| OPTION | EXAMPLE/LINK |
|--|--|
| Select Try It in the upper-right corner of a code block. Selecting Try It doesn't automatically copy the code to Cloud Shell. |  |
| Go to https://shell.azure.com , or select the Launch Cloud Shell button to open Cloud Shell in your browser. |  |

| OPTION | EXAMPLE/LINK |
|--|--|
| Select the Cloud Shell button on the menu bar at the upper right in the Azure portal . |  |

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

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
- A 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.

Log in to Azure

If you're not using Cloud Shell, you'll need to use the Azure CLI locally to access your account. Use the `az login` command from the Bash prompt to launch the browser-based login experience. If you have access to more than one Azure subscription, set the default with `az account set --subscription` followed by the subscription ID.

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](#)
- [Cmd](#)

```
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](#)
- [Cmd](#)

```
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](#)
- [Cmd](#)

```
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](#)
- [Cmd](#)

```
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 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](#)
- [Cmd](#)

```
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 2
```

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 Cosmos DB connection strings and save them in environment variables:

- [Bash](#)
- [Cmd](#)

```

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](#)
- [Cmd](#)

```

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](#)

- [Cmd](#)

```
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](#)
- [Cmd](#)

```
mvn archetype:generate --batch-mode \
-DarchetypeGroupId=com.microsoft.azure \
-DarchetypeArtifactId=azure-functions-archetype \
-DappName=$FUNCTION_APP \
-DresourceGroup=$RESOURCE_GROUP \
-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 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](#)
- [Cmd](#)

```
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](#)
- [Cmd](#)

```
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.

```
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 = "*/*10 * * * *") // 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",
            collectionName = "TelemetryInfo",
            connectionStringSetting = "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:

```

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](#)
- [Cmd](#)

```
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:

```
[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 Azure Data Explorer interface for the 'weatherdata123 - Data Explorer' account. The left sidebar has a red box around the 'Data Explorer' item under the 'weather' resource group. The main area shows the 'SQL API' view with the 'TelemetryDb' database selected. Under 'TelemetryDb', 'TelemetryInfo' is expanded, and 'Items' is selected, also highlighted with a red box. A table titled 'Items' displays several rows of sensor data. A specific row is selected and expanded to show its detailed JSON structure. The JSON includes fields like 'temperature', 'pressure', 'isNormalPressure', 'temperatureStatus', 'id', 'rid', '_self', '_etag', '_attachments', and '_ts'. The JSON structure is as follows:

```
1  "temperature": 47.967774187747615,  
2  "pressure": 19.17957577377465,  
3  "isNormalPressure": true,  
4  "temperatureStatus": "WARM",  
5  "id": "3d5e7408-4536-40c1-a849-5cc2b92",  
6  "rid": "3fRaAK+T9pIEAAAAAAA==",  
7  "_self": "obs/3fRaAA=/colls/3fRaAK+T9pIEAAAAAAA==",  
8  "_etag": "\\"3002e2ef-0000-0700-0000-5c2b92",  
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:

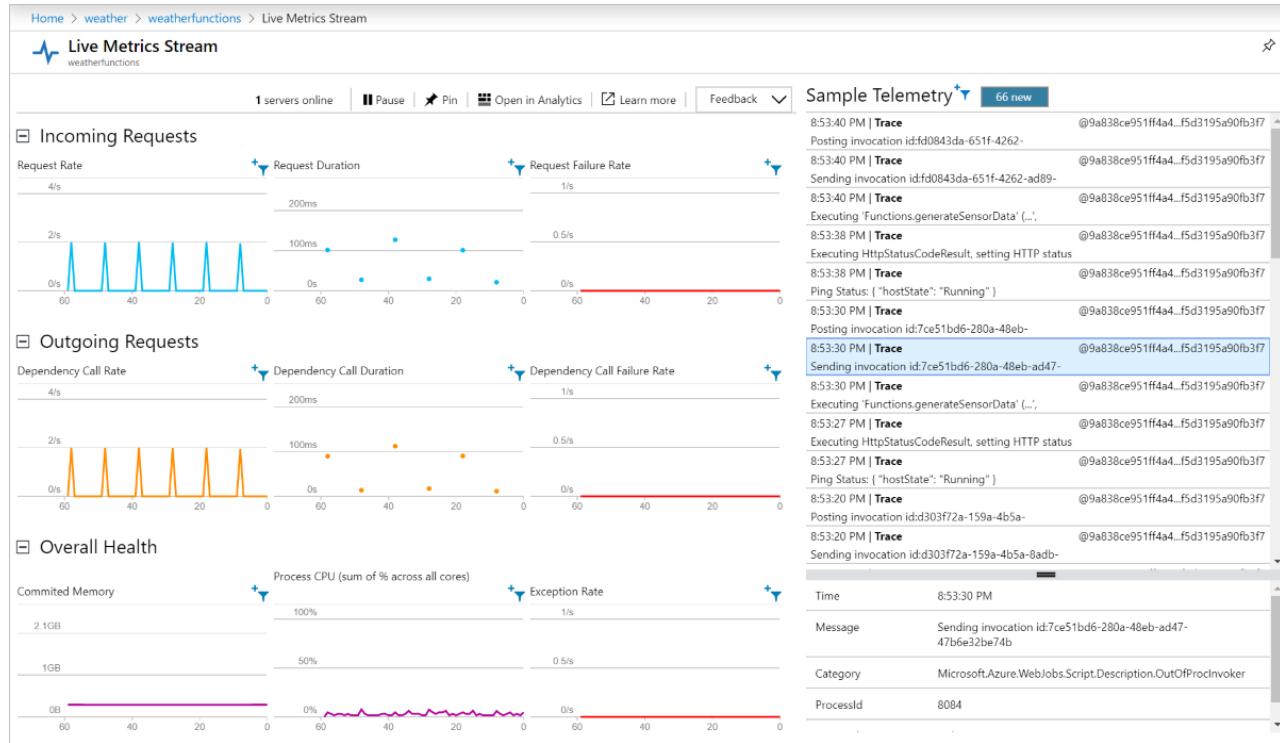
- [Bash](#)
- [Cmd](#)

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

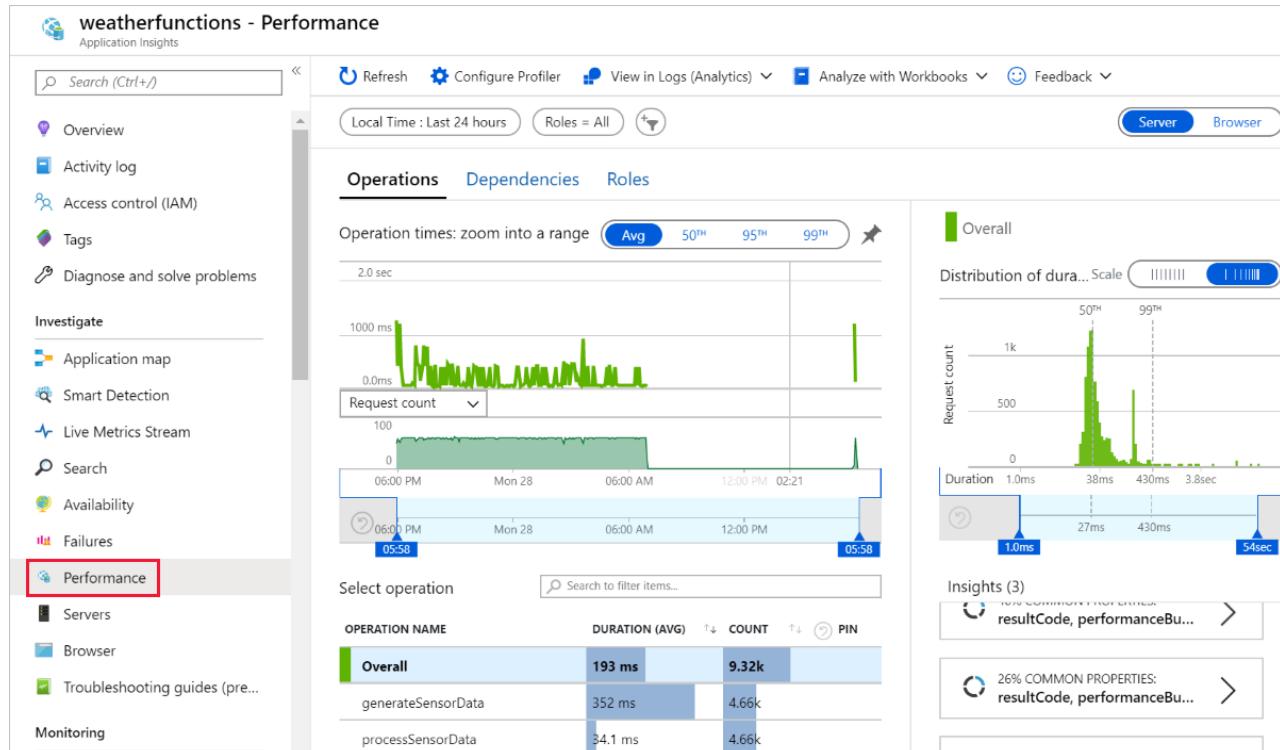
Your functions now run in Azure, and continue to accumulate data in your Azure Cosmos DB. You can view your

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](#)
- [Cmd](#)

```
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 a Cosmos DB. 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

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following table includes links to bash scripts for Azure Functions that use the Azure CLI.

| 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. |

| INTEGRATE | DESCRIPTION |
|--|--|
| Create a function app and connect to a storage account | Create a function app and connect it to a storage account. |
| Create a function app and connect to an Azure Cosmos DB | Create a function app and connect it to an Azure Cosmos DB. |
| Create a Python function app and mount a 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. |

| CONTINUOUS DEPLOYMENT | DESCRIPTION |
|--|---|
| Deploy from GitHub | Create a function app that deploys from a GitHub repository. |
| Deploy from Azure DevOps | Create a function app that deploys from an Azure DevOps repository. |

Create a function app for serverless code execution

12/4/2020 • 2 minutes to read • [Edit Online](#)

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 a [free account](#) before you begin.

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.
[Launch Cloud Shell](#)
- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This script creates an Azure Function app using the [Consumption plan](#).

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName=myserverlessfunc$RANDOM
region=westeurope

# Create a resource group.
az group create --name myResourceGroup --location $region

# Create an Azure storage account in the resource group.
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create a serverless function app in the resource group.
az functionapp create \
--name $functionAppName \
--storage-account $storageName \
--consumption-plan-location $region \
--resource-group myResourceGroup \
--functions-version 2

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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

12/4/2020 • 2 minutes to read • [Edit Online](#)

This Azure Functions sample script creates a function app, which is a container for your functions.

NOTE

The function app created runs on Python version 3.6. Python version 3.7 is also supported by Azure Functions.

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

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.

 [Launch Cloud Shell](#)

- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This script creates an Azure Function app using the [Consumption plan](#).

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName=myserverlessfunc$RANDOM
region=westeurope
pythonVersion=3.6 #3.7 also supported

# Create a resource group.
az group create --name myResourceGroup --location $region

# Create an Azure storage account in the resource group.
az storage account create \
    --name $storageName \
    --location $region \
    --resource-group myResourceGroup \
    --sku Standard_LRS

# Create a serverless function app in the resource group.
az functionapp create \
    --name $functionAppName \
    --storage-account $storageName \
    --consumption-plan-location $region \
    --resource-group myResourceGroup \
    --os-type Linux \
    --runtime python \
    --runtime-version $pythonVersion \
    --functions-version 2

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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

12/4/2020 • 2 minutes to read • [Edit Online](#)

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 a [free account](#) before you begin.

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.
[Launch Cloud Shell](#)
- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This script creates a function app using a [Premium plan](#).

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName=myappsvcpfunc$RANDOM
region=westeurope

# Create a resource resourceGroupName
az group create \
--name myResourceGroup \
--location $region

# Create an azure storage account
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create a Premium plan
az functionapp plan create \
--name mypremiumplan \
--resource-group myResourceGroup \
--location $region \
--sku EP1

# Create a Function App
az functionapp create \
--name $functionAppName \
--storage-account $storageName \
--plan mypremiumplan \
--resource-group myResourceGroup \
--functions-version 2

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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

12/4/2020 • 2 minutes to read • [Edit Online](#)

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 a [free account](#) before you begin.

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.
[!\[\]\(c0e0d05448c1ad4b43d7ac27c443233c_img.jpg\) Launch Cloud Shell](#)
- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This script creates an Azure Function app using a dedicated [App Service plan](#).

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName=myappsvcpfunc$RANDOM
region=westeurope

# Create a resource resourceGroupName
az group create \
--name myResourceGroup \
--location $region

# Create an azure storage account
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create an App Service plan
az functionapp plan create \
--name myappserviceplan \
--resource-group myResourceGroup \
--location $region \
--sku B1

# Create a Function App
az functionapp create \
--name $functionAppName \
--storage-account $storageName \
--plan myappserviceplan \
--resource-group myResourceGroup \
--functions-version 2

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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

12/4/2020 • 2 minutes to read • [Edit Online](#)

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 connection can be used with a [storage trigger or binding](#).

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

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.

 [Launch Cloud Shell](#)

- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This sample creates an Azure Function app and adds the storage connection string to an app setting.

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName="mystorageaccount$RANDOM"
functionAppName="myfuncwithstorage$RANDOM"
region=westeurope

# Create a resource group with location.
az group create \
--name myResourceGroup \
--location $region

# Create a storage account in the resource group.
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create a serverless function app in the resource group.
az functionapp create \
--name $functionAppName \
--resource-group myResourceGroup \
--storage-account $storageName \
--consumption-plan-location $region \
--functions-version 2

# Get the storage account connection string.
connstr=$(az storage account show-connection-string --name $storageName --resource-group myResourceGroup --query connectionString --output tsv)

# Update function app settings to connect to the storage account.
az functionapp config appsettings set \
--name $functionAppName \
--resource-group myResourceGroup \
--settings StorageConStr=$connstr

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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. |

| COMMAND | NOTES |
|--|---|
| <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](#).

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

12/4/2020 • 2 minutes to read • [Edit Online](#)

This Azure Functions sample script creates a function app and connects the function to an Azure Cosmos DB database. 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 a [free account](#) before you begin.

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.
[!\[\]\(4d97d23546cff6fa205bed4fb57555f2_img.jpg\) Launch Cloud Shell](#)
- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This sample creates an Azure Function app and adds a Cosmos DB endpoint and access key to app settings.

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName="myfuncwithcosmosdb$RANDOM"
region=westeurope

# Create a resource group with location.
az group create \
--name myResourceGroup \
--location $region

# Create a storage account for the function app.
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create a serverless function app in the resource group.
az functionapp create \
--name $functionAppName \
--resource-group myResourceGroup \
--storage-account $storageName \
--consumption-plan-location $region \
--functions-version 2

# Create an Azure Cosmos DB database using the same function app name.
az cosmosdb create \
--name $functionAppName \
--resource-group myResourceGroup

# Get the Azure Cosmos DB connection string.
endpoint=$(az cosmosdb show \
--name $functionAppName \
--resource-group myResourceGroup \
--query documentEndpoint \
--output tsv)

key=$(az cosmosdb list-keys \
--name $functionAppName \
--resource-group myResourceGroup \
--query primaryMasterKey \
--output tsv)

# Configure function app settings to use the Azure Cosmos DB connection string.
az functionapp config appsettings set \
--name $functionAppName \
--resource-group myResourceGroup \
--setting CosmosDB_Endpoint=$endpoint CosmosDB_Key=$key

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

This script uses the following commands: Each command in the table links to command specific documentation.

| COMMAND | NOTES |
|--|---|
| <code>az group create</code> | Create a resource group with location |
| <code>az storage accounts create</code> | Create a storage account |
| <code>az functionapp create</code> | Creates a function app in the serverless Consumption plan . |
| <code>az cosmosdb create</code> | Create an Azure Cosmos DB database. |
| <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](#).

Additional 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

12/4/2020 • 2 minutes to read • [Edit Online](#)

This Azure Functions sample script creates a function app 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 3.7. Azure Functions also [supports Python versions 3.6 and 3.8](#).

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

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.

 [Launch Cloud Shell](#)

- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This script creates an Azure Function app using the [Consumption plan](#).

```

#!/bin/bash

# Function app and storage account names must be unique.
export AZURE_STORAGE_ACCOUNT=mystorageaccount$RANDOM
functionAppName=myserverlessfunc$RANDOM
region=westeurope
pythonVersion=3.7 #3.6 also supported
shareName=myfileshare
directoryName=mydir
shareId=myshare$RANDOM
mountPath=/mounted

# Create a resource group.
az group create --name myResourceGroup --location $region

# Create an Azure storage account in the resource group.
az storage account create \
--name $AZURE_STORAGE_ACCOUNT \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Set the storage account key as an environment variable.
export AZURE_STORAGE_KEY=$(az storage account keys list -g myResourceGroup -n $AZURE_STORAGE_ACCOUNT --query '[0].value' -o tsv)

# Create a serverless function app in the resource group.
az functionapp create \
--name $functionAppName \
--storage-account $AZURE_STORAGE_ACCOUNT \
--consumption-plan-location $region \
--resource-group myResourceGroup \
--os-type Linux \
--runtime python \
--runtime-version $pythonVersion \
--functions-version 2

# Work with Storage account using the set env variables.
# Create a share in Azure Files.
az storage share create \
--name $shareName

# Create a directory in the share.
az storage directory create \
--share-name $shareName \
--name $directoryName

az webapp config storage-account add \
--resource-group myResourceGroup \
--name $functionAppName \
--custom-id $shareId \
--storage-type AzureFiles \
--share-name $shareName \
--account-name $AZURE_STORAGE_ACCOUNT \
--mount-path $mountPath \
--access-key $AZURE_STORAGE_KEY

az webapp config storage-account list \
--resource-group myResourceGroup \
--name $functionAppName

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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

12/4/2020 • 2 minutes to read • [Edit Online](#)

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 GitHub repository.

In this sample, you need:

- A GitHub repository with functions code, that you have administrative permissions for.
- A [Personal Access Token \(PAT\)](#) for your GitHub account.

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

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.

 [Launch Cloud Shell](#)

- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This sample creates an Azure Function app and deploys function code from GitHub.

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName=mygithubfunc$RANDOM
region=westeurope

# TODO:
gitrepo=<Replace with your GitHub repo URL e.g. https://github.com/Azure-Samples/functions-quickstart.git>
token=<Replace with a GitHub access token>

# Enable authenticated git deployment in your subscription from a private repo.
az functionapp deployment source update-token \
--git-token $token

# Create a resource group.
az group create \
--name myResourceGroup \
--location $region

# Create an Azure storage account in the resource group.
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create a function app with source files deployed from the specified GitHub repo.
az functionapp create \
--name $functionAppName \
--storage-account $storageName \
--consumption-plan-location $region \
--resource-group myResourceGroup \
--deployment-source-url $gitrepo \
--deployment-source-branch master \
--functions-version 2

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

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 a function in Azure that is deployed from Azure DevOps

12/4/2020 • 2 minutes to read • [Edit Online](#)

This topic shows you how to use Azure Functions to create a [serverless](#) function app using the [Consumption plan](#). The function app, which is a container for your functions, is continuously deployed from an Azure DevOps repository.

To complete this topic, you must have:

- An Azure DevOps repository that contains your function app project and to which you have administrative permissions.
- A [personal access token \(PAT\)](#) to access your Azure DevOps repository.

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

Prerequisites

- Use [Azure Cloud Shell](#) using the bash environment.
[Launch Cloud Shell](#)
- If you prefer, [install](#) Azure CLI to run CLI reference commands.
 - If you're using a local install, sign in with Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for additional sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- This tutorial requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Sample script

This sample creates an Azure Function app and deploys function code from Azure DevOps.

```

#!/bin/bash

# Function app and storage account names must be unique.
storageName=mystorageaccount$RANDOM
functionAppName=mygithubfunc$RANDOM
region=westeurope

# TODO:
gitrepo=<Replace with your VSTS repo URL, like
https://samples.visualstudio.com/DefaultCollection/_git/Function-Quickstart>
token=<Replace with a Visual Studio Team Services personal access token>

# Create a resource group.
az group create \
--name myResourceGroup \
--location $region

# Create an Azure storage account.
az storage account create \
--name $storageName \
--location $region \
--resource-group myResourceGroup \
--sku Standard_LRS

# Create a serverless function app.
az functionapp create \
--name $functionAppName \
--storage-account $storageName \
--consumption-plan-location $region \
--resource-group myResourceGroup \
--functions-version 2

# Set the deployment source to the VSTS repo using the token.
az functionapp deployment source config \
--name $functionAppName \
--resource-group myResourceGroup \
--repo-url $gitrepo \
--branch master \
--git-token $token

```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Script explanation

This script uses the following commands to create a resource group, storage account, function app, and all related resources. Each command in the table links to command specific documentation.

| 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 . |

| COMMAND | NOTES |
|---|---|
| az functionapp deployment source config | Associates a function app 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](#).

Azure Functions runtime versions overview

12/4/2020 • 10 minutes to read • [Edit Online](#)

Azure Functions currently supports three versions of the runtime host: 1.x, 2.x, and 3.x. All three versions are supported for production scenarios.

IMPORTANT

Version 1.x is in maintenance mode and only supports development in the Azure portal, Azure Stack Hub portal, or locally on Windows computers. Enhancements are provided only in later versions.

This article details some of the differences between the various versions, how you can create each version, and how to change versions.

Languages

Starting with version 2.x, the runtime uses a language extensibility model, and all functions in a function app must share the same language. The language of functions in a function app is chosen when creating the app and is maintained in the [FUNCTIONS_WORKER_RUNTIME](#) setting.

The following table indicates which programming languages are currently supported in each runtime version.

| LANGUAGE | 1.X | 2.X | 3.X |
|------------|-------------------------|------------------------|--|
| C# | GA (.NET Framework 4.7) | GA (.NET Core 2.2) | GA (.NET Core 3.1) |
| JavaScript | GA (Node 6) | GA (Node 10 & 8) | GA (Node 12 & 10) Preview (Node 14) |
| F# | GA (.NET Framework 4.7) | GA (.NET Core 2.2) | GA (.NET Core 3.1) |
| Java | N/A | GA (Java 8) | GA (Java 11 & 8) |
| PowerShell | N/A | GA (PowerShell Core 6) | GA (PowerShell 7 & Core 6) |
| Python | N/A | GA (Python 3.7 & 3.6) | GA (Python 3.8, 3.7, & 3.6) |
| TypeScript | N/A | GA* | GA* |

* Supported through transpiling to JavaScript.

See the language-specific developer guide article for more details about supported language versions.

For information about planned changes to language support, see [Azure roadmap](#).

Run on a specific version

By default, function apps created in the Azure portal and by the Azure CLI are set to version 3.x. You can modify this version as needed. You can only change the runtime version to 1.x after you create your function app but before you add any functions. Moving between 2.x and 3.x is allowed even with apps that have functions, but it is still recommended to test in a new app first.

Migrating from 1.x to later versions

You may choose to migrate an existing app written to use the version 1.x runtime to instead use a newer version. Most of the changes you need to make are related to changes in the language runtime, such as C# API changes between .NET Framework 4.7 and .NET Core. You'll also need to make sure your code and libraries are compatible with the language runtime you choose. Finally, be sure to note any changes in trigger, bindings, and features highlighted below. For the best migration results, you should create a new function app in a new version and port your existing version 1.x function code to the new app.

While it's possible to do an "in-place" upgrade by manually updating the app configuration, going from 1.x to a higher version includes some breaking changes. For example, in C#, the debugging object is changed from `TraceWriter` to `ILogger`. By creating a new version 3.x project, you start off with updated functions based on the latest version 3.x templates.

Changes in triggers and bindings after version 1.x

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 Hub `path` property is now `eventHubName`. See the [existing binding table](#) for links to documentation for each binding.

Changes in features and functionality after version 1.x

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 File storage by default. When upgrading an app from version 1.x to version 2.x, existing secrets that are in file storage 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.
- The host configuration file (`host.json`) should be empty or have the string `"version": "2.0"`.
- 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 in the `maxConcurrentRequests` setting in the `host.json` file.
- Because of [.NET Core limitations](#), support for F# script (.fsx) functions has been removed. Compiled F# functions (.fs) are still supported.
- The URL format of Event Grid trigger webhooks has been changed to

`https://{{app}}/runtime/webhooks/{{triggerName}}`.

Migrating from 2.x to 3.x

Azure Functions version 3.x is highly backwards compatible to version 2.x. Many apps should be able to safely upgrade to 3.x without any code changes. While moving to 3.x is encouraged, be sure to run extensive tests before changing the major version in production apps.

Breaking changes between 2.x and 3.x

The following are the changes to be aware of before upgrading a 2.x app to 3.x.

JavaScript

- Output bindings assigned through `context.done` or return values now behave the same as setting in `context.bindings`.
- Timer trigger object is camelCase instead of PascalCase
- Event Hub triggered functions with `dataType` binary will receive an array of `binary` instead of `string`.
- The HTTP request payload can no longer be accessed via `context.bindingData.req`. It can still be accessed as an input parameter, `context.req`, and in `context.bindings`.
- Node.js 8 is no longer supported and will not execute in 3.x functions.

.NET

- [Synchronous server operations are disabled by default.](#)

Changing version of apps in Azure

The version of the Functions runtime used by published apps in Azure is dictated by the

[`FUNCTIONS_EXTENSION_VERSION`](#) application setting. The following major runtime version values are supported:

| VALUE | RUNTIME TARGET |
|-----------------|----------------|
| <code>~3</code> | 3.x |
| <code>~2</code> | 2.x |
| <code>~1</code> | 1.x |

IMPORTANT

Don't arbitrarily change this setting, because other app setting changes and changes to your function code may be required.

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 three 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:

`Version 1.x`

```
<TargetFramework>net461</TargetFramework>
<AzureFunctionsVersion>v1</AzureFunctionsVersion>
```

Version 2.x

```
<TargetFramework>netcoreapp2.1</TargetFramework>
<AzureFunctionsVersion>v2</AzureFunctionsVersion>
```

Version 3.x

```
<TargetFramework>netcoreapp3.1</TargetFramework>
<AzureFunctionsVersion>v3</AzureFunctionsVersion>
```

NOTE

Azure Functions 3.x and .NET requires the `Microsoft.NET.Sdk.Functions` extension be at least `3.0.0`.

Updating 2.x apps to 3.x in Visual Studio

You can open an existing function targeting 2.x and move to 3.x by editing the `.csproj` file and updating the values above. Visual Studio manages runtime versions automatically for you based on project metadata. However, it's possible if you have never created a 3.x app before that Visual Studio doesn't yet have the templates and runtime for 3.x on your machine. This may present itself with an error like "no Functions runtime available that matches the version specified in the project." To fetch the latest templates and runtime, go through the experience to create a new function project. When you get to the version and template select screen, wait for Visual Studio to complete fetching the latest templates. Once the latest .NET Core 3 templates are available and displayed you should be able to run and debug any project configured for version 3.x.

IMPORTANT

Version 3.x functions can only be developed in Visual Studio if using Visual Studio version 16.4 or newer.

VS 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. To develop against version 3.x, install version 3.x of the Core Tools. Version 2.x development requires version 2.x of the Core Tools, and so on. For more information, see [Install the Azure Functions Core Tools](#).

For Visual Studio Code development, you may 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. To create apps in `~3` you would update the `azureFunctions.projectRuntime` user setting to `~3`.

Azure Functions: Project Runtime

The default runtime to use when performing operations in the Azure Functions extension (e.g. "Create New Function").

`~3`



Maven and Java apps

You can migrate Java apps from version 2.x to 3.x by [installing the 3.x version of the core tools](#) required to run locally. After verifying that your app works correctly running locally on version 3.x, update the app's `POM.xml` file to modify the `FUNCTIONS_EXTENSION_VERSION` setting to `~3`, as in the following example:

```

<configuration>
  <resourceGroup>${functionResourceGroup}</resourceGroup>
  <appName>${functionAppName}</appName>
  <region>${functionAppRegion}</region>
  <appSettings>
    <property>
      <name>WEBSITE_RUN_FROM_PACKAGE</name>
      <value>1</value>
    </property>
    <property>
      <name>FUNCTIONS_EXTENSION_VERSION</name>
      <value>~3</value>
    </property>
  </appSettings>
</configuration>

```

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.

With the exception of 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:

| TYPE | 1.X | 2.X AND HIGHER ¹ | TRIGGER | INPUT | OUTPUT |
|--------------------|-----|-----------------------------|---------|-------|--------|
| Blob storage | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure Cosmos DB | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dapr ³ | | ✓ | ✓ | ✓ | ✓ |
| Event Grid | ✓ | ✓ | ✓ | | ✓ |
| Event Hubs | ✓ | ✓ | ✓ | | ✓ |
| HTTP & webhooks | ✓ | ✓ | ✓ | | ✓ |
| IoT Hub | ✓ | ✓ | ✓ | | ✓ |
| Kafka ² | | ✓ | ✓ | | ✓ |
| Mobile Apps | ✓ | | | ✓ | ✓ |

| Type | 1.x | 2.x and higher | Trigger | Input | Output |
|-----------------------|-----|----------------|---------|-------|--------|
| Notification Hubs | ✓ | | | | ✓ |
| Queue storage | ✓ | ✓ | ✓ | | ✓ |
| RabbitMQ ² | | ✓ | ✓ | | ✓ |
| SendGrid | ✓ | ✓ | | | ✓ |
| Service Bus | ✓ | ✓ | ✓ | | ✓ |
| SignalR | | ✓ | | ✓ | ✓ |
| Table storage | ✓ | ✓ | | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | | |
| Twilio | ✓ | ✓ | | | ✓ |

¹ Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

² Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

³ Supported only in Kubernetes, IoT Edge, and other self-hosted modes only.

Function app timeout duration

The timeout duration of a function app is defined by the `functionTimeout` property in the `host.json` project file. The following table shows the default and maximum values in minutes for both plans and the different runtime versions:

| Plan | Runtime Version | Default | Maximum |
|-------------|-----------------|-----------|-----------|
| Consumption | 1.x | 5 | 10 |
| Consumption | 2.x | 5 | 10 |
| Consumption | 3.x | 5 | 10 |
| Premium | 1.x | 30 | Unlimited |
| Premium | 2.x | 30 | Unlimited |
| Premium | 3.x | 30 | Unlimited |
| App Service | 1.x | Unlimited | Unlimited |
| App Service | 2.x | 30 | Unlimited |
| App Service | 3.x | 30 | Unlimited |

NOTE

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](#).

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 Premium plan

11/2/2020 • 10 minutes to read • [Edit Online](#)

The Azure Functions Premium plan (sometimes referred to as Elastic Premium plan) is a hosting option for function apps. The Premium plan provides features like VNet connectivity, no cold start, and premium hardware. 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. For a comparison of the Premium plan and other plan and hosting types, see [function scale and hosting options](#).

Create a Premium plan

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. Choose .NET for C# and F# functions. |
| Region | Preferred region | Choose a region near you or near other services your functions access. |

Function App

X

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 * ⓘ

Visual Studio Enterprise

Resource Group * ⓘ

(New) myResourceGroup

[Create new](#)

Instance Details

Function App name *

myfunctionapp



.azurewebsites.net

Publish *

[Code](#) Docker Container

Runtime stack *

.NET Core

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 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. |
| 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 . |

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 *

(New) storageaccountmyres93a6

Create new

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System *

Linux Windows

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type *

Premium

Cannot find your App service plan? [Try a different location in Basics tab.](#)

Windows Plan (Central US) *

(New) ASP-myResourceGroup-8ddf

Create new

Sku and size *

Elastic Premium EP1

210 total ACU, 3.5 GB memory

Change size

[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, you can change the New resource name or choose a different Location in an Azure geography to store your data. |

Function App

Basics Hosting Monitoring **Tags** Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

Create new

Region

Central US

Review + create < Previous Next : Tags >

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.

You can also create a Premium plan using `az functionapp plan create` in the Azure CLI. The following example creates an *Elastic Premium 1* tier plan:

```
az functionapp plan create --resource-group <RESOURCE_GROUP> --name <PLAN_NAME> \
--location <REGION> --sku EP1
```

In this example, replace `<RESOURCE_GROUP>` with your resource group and `<PLAN_NAME>` with a name for your plan that is unique in the resource group. Specify a supported `<REGION>`. To create a Premium plan that supports Linux, include the `--is-linux` option.

With the plan created, you can use `az functionapp create` to create your function app. In the portal, both the plan and the app are created at the same time. For an example of a complete Azure CLI script, see [Create a function app in a Premium plan](#).

Features

The following features are available to function apps deployed to a Premium plan.

Always ready instances

If no events and executions occur today in the Consumption plan, your app may scale in to zero instances. When new events come in, a new instance needs to be specialized with your app running on it. Specializing new instances may take some time depending on the app. This additional latency on the first call is often called app cold start.

In the Premium plan, you can have your app always ready on a specified number of instances. The maximum number of always ready instances is 20. When events begin to trigger the app, they are routed to the always ready instances first. As the function becomes active, additional instances will be warmed as a buffer. This buffer prevents cold start for new instances required during scale. These buffered instances are called [pre-warmed instances](#). With the combination of the always ready instances and a pre-warmed buffer, your app can effectively eliminate cold start.

NOTE

Every premium plan will have at least one active (billed) instance at all times.

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.

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

You can also configure always ready instances for an app with the Azure CLI.

```
az resource update -g <resource_group> -n <function_app_name>/config/web --set  
properties.minimumElasticInstanceCount=<desired_always_ready_count> --resource-type Microsoft.Web/sites
```

Pre-warmed instances

Pre-warmed instances are the number of instances warmed as a buffer during scale and activation events. Pre-warmed instances continue to buffer until the maximum scale-out limit is reached. The default pre-warmed instance count is 1, and for most scenarios should remain as 1. If an app has a long warm up (like a custom container image), you may wish to increase this buffer. A pre-warmed instance will become active only after all active instances have been sufficiently utilized.

Consider this example of how always ready instances and pre-warmed instances work together. A premium function app has five always ready instances configured, and the default of one pre-warmed instance. When the app is idle and no events are triggering, the app will be provisioned and running on five instances. At this time, you will not be billed for a pre-warmed instance as the always ready instances aren't used, and no pre-warmed instance is even allocated.

As soon as the first trigger comes in, the five always ready instances become active, and a pre-warmed instance is allocated. The app is now running with six provisioned instances: the five now-active always ready instances, and the sixth pre-warmed and inactive buffer. If the rate of executions continues to increase, the five active instances will eventually be utilized. When the platform decides to scale beyond five instances, it will scale into the pre-warmed instance. When that happens, there will now be six active instances, and a seventh instance will instantly be provisioned and fill the pre-warmed buffer. This sequence of scaling and pre-warming will continue until the maximum instance count for the app is reached. No instances will be pre-warmed or activated beyond the maximum.

You can modify the number of pre-warmed instances for an app using the Azure CLI.

```
az resource update -g <resource_group> -n <function_app_name>/config/web --set  
properties.preWarmedInstanceCount=<desired_prewarmed_count> --resource-type Microsoft.Web/sites
```

Maximum instances for an app

In addition to the [plan maximum instance count](#), you can configure a per-app maximum. The app maximum can be configured using the [app scale limit](#).

Private network connectivity

Azure Functions deployed to a Premium plan takes advantage of [new VNet integration for web apps](#). When configured, your app can communicate with resources within your VNet 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 VNet](#).

Rapid elastic scale

Additional 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 [Function scale and hosting](#).

Longer run duration

Azure 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 (guaranteed 60 minutes).

Plan and SKU 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 maximum burst limit. You are billed for instances beyond your plan size only while they are running and allocated to you, on a per-second basis. We will make a best effort at scaling your app out to its defined maximum limit.

You can configure the plan size and maximums in the Azure portal by selecting the **Scale Out** options in the plan or a function app deployed to that plan (under **Platform Features**).

You can also increase the maximum burst limit from the Azure CLI:

```
az functionapp plan update -g <resource_group> -n <premium_plan_name> --max-burst <desired_max_burst>
```

The minimum for every plan will be at least one instance. The actual minimum number of instances will be autoconfigured 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 will be calculated as five. App A will be running on all 5, and app B will only be 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 should be sufficient. However, scaling beyond the minimum occurs at a best effort. It is possible, though unlikely, that at a specific time scale-out could be delayed if additional instances are unavailable. By setting a minimum higher than the autocalculated minimum, you reserve instances in advance of scale-out.

Increasing the calculated minimum for a plan can be done using the Azure CLI.

```
az functionapp plan update -g <resource_group> -n <premium_plan_name> --min-instances <desired_min_instances>
```

Available instance SKUs

When creating or scaling your plan, you can choose between three instance sizes. You will be 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.

| SKU | CORES | MEMORY | STORAGE |
|-----|-------|--------|---------|
| EP1 | 1 | 3.5GB | 250GB |
| EP2 | 2 | 7GB | 250GB |
| EP3 | 4 | 14GB | 250GB |

Memory utilization considerations

Running on a machine with more memory does not always mean that your function app will use 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>`.

Region Max Scale Out

Below are the currently supported maximum scale-out values for a single plan in each region and OS configuration. To request an increase, please open a support ticket.

See the complete regional availability of Functions here: Azure.com

| REGION | WINDOWS | LINUX |
|----------------------|---------------|---------------|
| Australia Central | 100 | Not Available |
| Australia Central 2 | 100 | Not Available |
| Australia East | 100 | 20 |
| Australia Southeast | 100 | 20 |
| Brazil South | 100 | 20 |
| Canada Central | 100 | 20 |
| Central US | 100 | 20 |
| China East 2 | 100 | 20 |
| China North 2 | 100 | 20 |
| East Asia | 100 | 20 |
| East US | 100 | 20 |
| East US 2 | 100 | 20 |
| France Central | 100 | 20 |
| Germany West Central | 100 | Not Available |
| Japan East | 100 | 20 |
| Japan West | 100 | 20 |
| Korea Central | 100 | 20 |
| Korea South | Not Available | 20 |
| North Central US | 100 | 20 |

| REGION | WINDOWS | LINUX |
|-------------------|---------|---------------|
| North Europe | 100 | 20 |
| Norway East | 100 | 20 |
| South Central US | 100 | 20 |
| South India | 100 | Not Available |
| Southeast Asia | 100 | 20 |
| Switzerland North | 100 | Not Available |
| Switzerland West | 100 | Not Available |
| UK South | 100 | 20 |
| UK West | 100 | 20 |
| USGov Arizona | 100 | 20 |
| USGov Virginia | 100 | 20 |
| USNat East | 100 | Not Available |
| USNat West | 100 | Not Available |
| West Europe | 100 | 20 |
| West India | 100 | 20 |
| West Central US | 100 | 20 |
| West US | 100 | 20 |
| West US 2 | 100 | 20 |

Next steps

[Understand Azure Functions scale and hosting options](#)

Deployment technologies in Azure Functions

12/4/2020 • 10 minutes to read • [Edit Online](#)

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 Azure is generally determined by the way in which you publish your app. The appropriate deployment method is determined by 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 are more likely to publish continuously from source control or by using an automated publishing pipeline, which includes additional validation and testing.

The following table describes the available deployment methods for your Function project.

| DEPLOYMENT TYPE | METHODS | BEST FOR... |
|---------------------|---|--|
| Tools-based | <ul style="list-style-type: none">• Visual Studio Code publish• Visual Studio publish• Core Tools publish | Deployments during development and other ad-hoc deployments. Deployments are managed locally by the tooling. |
| 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">• DevOps Pipelines• GitHub actions | Production and DevOps pipelines that include additional validation, testing, and other actions be run as part of an automated deployment. Deployments are managed by the pipeline. |

While specific Functions deployments use the best technology based on their context, most deployment methods are based on [zip deployment](#).

Deployment technology availability

Azure Functions supports cross-platform local development and hosting on Windows and Linux. Currently, three hosting plans are available:

- [Consumption](#)
- [Premium](#)
- [Dedicated \(App Service\)](#)

Each plan has different behaviors. Not all deployment technologies are available for each flavor of Azure Functions. The following chart shows which deployment technologies are supported for each combination of operating system and hosting plan:

| DEPLOYMENT TECHNOLOGY | WINDOWS CONSUMPTION | WINDOWS PREMIUM | WINDOWS DEDICATED | LINUX CONSUMPTION | LINUX PREMIUM | LINUX DEDICATED |
|-----------------------------------|---------------------|-----------------|-------------------|-------------------|----------------|-----------------|
| External package URL ¹ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Zip deploy | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Docker container | | | | | ✓ | ✓ |
| Web Deploy | ✓ | ✓ | ✓ | | | |
| Source control | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Local Git ¹ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Cloud sync ¹ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| FTP ¹ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Portal editing | ✓ | ✓ | ✓ | | ✓ ² | ✓ ² |

¹ Deployment technology that requires [manual trigger syncing](#).² Portal editing is enabled only for HTTP and Timer triggers for Functions on Linux using Premium and Dedicated plans.

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. When you deploy your updates by referencing an external package URL, local Git, cloud sync, or FTP, you must manually sync your triggers. 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.

Remote build

Azure Functions can automatically perform builds on the code it receives after zip deployments. These builds behave slightly differently depending on whether your app is running on Windows or Linux. Remote builds are not performed when an app has previously been set to run in [Run From Package](#) mode. To learn how to use remote build, navigate to [zip deploy](#).

NOTE

If you're having issues with remote build, it might be because your app was created before the feature was made available (August 1, 2019). Try creating a new function app, or running `az functionapp update -g <RESOURCE_GROUP_NAME> -n <APP_NAME>` to update your function app. This command might take two tries to succeed.

Remote build on Windows

All function apps running on Windows have a small management app, the SCM (or [Kudu](#)) site. 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.

Remote build on Linux

To enable remote build on Linux, the following [application settings](#) must be set:

- `ENABLE_ORYX_BUILD=true`
- `SCM_DO_BUILD_DURING_DEPLOYMENT=true`

By default, both [Azure Functions Core Tools](#) and the [Azure Functions Extension for Visual Studio Code](#) perform remote builds when deploying to Linux. Because of this, both tools automatically create these settings for you in Azure.

When apps are built remotely on Linux, they [run from the deployment package](#).

Consumption plan

Linux function apps running in the Consumption plan don't have an SCM/Kudu site, which limits the deployment options. However, function apps on Linux running in the Consumption plan do support remote builds.

Dedicated and Premium plans

Function apps running on Linux in the [Dedicated \(App Service\) plan](#) and the [Premium plan](#) also have a limited SCM/Kudu site.

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. When you update the package file that a function app references, you must [manually sync triggers](#) to tell Azure that your application has changed.

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.

Docker container

You can deploy a Linux container image that contains your function app.

How to use it: Create a Linux function app in the Premium or Dedicated plan and specify which container image to run from. You can do this in two ways:

- Create a Linux function app on an Azure App Service plan in the Azure portal. For **Publish**, select **Docker Image**, and then configure the container. Enter the location where the image is hosted.
- Create a Linux function app on an App Service plan by using the Azure CLI. To learn how, see [Create a function on Linux by using a custom image](#).

To deploy to an existing app by using a custom container, in [Azure Functions Core Tools](#), use the `func 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. This deployment mechanism is available only for Functions running on Linux.

Web Deploy (MSDeploy)

Web Deploy packages and deploys your Windows applications to any IIS server, including your function apps running on Windows in Azure.

How to use it: Use [Visual Studio tools for Azure Functions](#). Clear the **Run from package file (recommended)** check box.

You can also download [Web Deploy 3.6](#) and call `MSDeploy.exe` directly.

When to use it: Web Deploy is supported and has no issues, but the preferred mechanism is [zip deploy with Run From Package enabled](#). To learn more, see the [Visual Studio development guide](#).

Source control

Use source control to connect your function app to a Git repository. An update to code in that repository triggers deployment. For more information, see the [Kudu Wiki](#).

How to use it: Use Deployment Center in the Functions area of the portal to set up publishing from source control. 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.

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: In general, we recommend that you use a different deployment method. When you publish from local Git, you must [manually sync triggers](#).

Cloud sync

Use cloud sync to sync your content from Dropbox and OneDrive to Azure Functions.

How to use it: Follow the instructions in [Sync content from a cloud folder](#).

When to use it: In general, we recommend other deployment methods. When you publish by using cloud sync, you must [manually sync triggers](#).

FTP

You can use FTP to directly transfer files to Azure Functions.

How to use it: Follow the instructions in [Deploy content by using FTP/s](#).

When to use it: In general, we recommend other deployment methods. When you publish by using FTP, you must [manually sync triggers](#).

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 intense development work, we recommend that you use one of the following client tools:

- [Visual Studio Code](#)
- [Azure Functions Core Tools \(command line\)](#)
- [Visual Studio](#)

The following table shows the operating systems and languages that support portal editing:

| LANGUAGE | WINDOWS CONSUMPTION | WINDOWS PREMIUM | WINDOWS DEDICATED | LINUX CONSUMPTION | LINUX PREMIUM | LINUX DEDICATED |
|----------------------|---------------------|-----------------|-------------------|-------------------|---------------|-----------------|
| C# | | | | | | |
| C# Script | ✓ | ✓ | ✓ | | ✓* | ✓* |
| F# | | | | | | |
| Java | | | | | | |
| JavaScript (Node.js) | ✓ | ✓ | ✓ | | ✓* | ✓* |
| Python (Preview) | | | | | | |
| PowerShell (Preview) | ✓ | ✓ | ✓ | | | |
| TypeScript (Node.js) | | | | | | |

* Portal editing is enabled only for HTTP and Timer triggers for Functions on Linux using Premium and Dedicated plans.

Deployment behaviors

When you do a deployment, all existing executions are allowed to complete or time out, after which the new code is loaded to begin processing requests.

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. 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 DevOps](#)
- [Zip deployments for Azure Functions](#)
- [Run your Azure Functions from a package file](#)
- [Automate resource deployment for your function app in Azure Functions](#)

Azure Functions reliable event processing

12/4/2020 • 6 minutes to read • [Edit Online](#)

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](#)

Designing Azure Functions for identical input

11/20/2019 • 2 minutes to read • [Edit Online](#)

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.

Azure Functions triggers and bindings concepts

12/4/2020 • 5 minutes to read • [Edit Online](#)

In this article you learn the high-level concepts surrounding functions triggers and bindings.

Triggers are what cause a function to run. A trigger defines how a function is invoked and a function must have exactly one trigger. Triggers have associated data, which is often provided as the payload of the function.

Binding to a function is a way of declaratively connecting another resource to the function; bindings may be connected as *input bindings*, *output bindings*, or both. Data from bindings is provided to the function as parameters.

You can mix and match different bindings to suit your needs. 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.

| EXAMPLE SCENARIO | TRIGGER | INPUT BINDING | OUTPUT BINDING |
|---|------------|----------------------------|-----------------|
| A new queue message arrives which runs a function to write to another queue. | Queue* | <i>None</i> | Queue* |
| A scheduled job reads Blob Storage contents and creates a new Cosmos DB document. | Timer | Blob Storage | Cosmos DB |
| The Event Grid is used to read an image from Blob Storage and a document from Cosmos DB to send an email. | Event Grid | Blob Storage and Cosmos DB | SendGrid |
| A webhook that uses Microsoft Graph to update an Excel sheet. | HTTP | <i>None</i> | Microsoft Graph |

* Represents different queues

These examples are not meant to be exhaustive, but are provided to illustrate how you can use triggers and bindings together.

Trigger and binding definitions

Triggers and bindings are defined differently depending on the development approach.

| PLATFORM | TRIGGERS AND BINDINGS ARE CONFIGURED BY... |
|-------------------------------------|--|
| C# class library | decorating methods and parameters with C# attributes |
| All others (including Azure portal) | updating function.json (schema) |

The portal provides a UI for this configuration, but you can edit the file directly by opening the **Advanced editor** available via the **Integrate** tab of your function.

In .NET, the parameter type defines the data type for input data. For instance, use `string` to bind to the text of a queue trigger, a byte array to read as binary and a custom type to de-serialize to an object.

For languages that are dynamically typed such as JavaScript, use the `dataType` property in the `function.json` file. For example, to read the content of an HTTP request in binary format, set `dataType` to `binary`:

```
{
  "dataType": "binary",
  "type": "httpTrigger",
  "name": "req",
  "direction": "in"
}
```

Other options for `dataType` are `stream` and `string`.

Binding direction

All triggers and bindings have a `direction` property in the `function.json` file:

- For triggers, the direction is always `in`
- Input and output bindings use `in` and `out`
- Some bindings support a special direction `inout`. If you use `inout`, only the **Advanced editor** is available via the **Integrate** tab in the portal.

When you use [attributes in a class library](#) to configure triggers and bindings, the direction is provided in an attribute constructor or inferred from the parameter type.

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:

| TYPE | 1.X | 2.X AND HIGHER ¹ | TRIGGER | INPUT | OUTPUT |
|-----------------|-----|-----------------------------|---------|-------|--------|
| Blob storage | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure Cosmos DB | ✓ | ✓ | ✓ | ✓ | ✓ |

| Type | 1.x | 2.x and higher | Trigger | Input | Output |
|-----------------------|-----|----------------|---------|-------|--------|
| Dapr ³ | | ✓ | ✓ | ✓ | ✓ |
| Event Grid | ✓ | ✓ | ✓ | | ✓ |
| Event Hubs | ✓ | ✓ | ✓ | | ✓ |
| HTTP & webhooks | ✓ | ✓ | ✓ | | ✓ |
| IoT Hub | ✓ | ✓ | ✓ | | ✓ |
| Kafka ² | | ✓ | ✓ | | ✓ |
| Mobile Apps | ✓ | | | ✓ | ✓ |
| Notification Hubs | ✓ | | | | ✓ |
| Queue storage | ✓ | ✓ | ✓ | | ✓ |
| RabbitMQ ² | | ✓ | ✓ | | ✓ |
| SendGrid | ✓ | ✓ | | | ✓ |
| Service Bus | ✓ | ✓ | ✓ | | ✓ |
| SignalR | | ✓ | | ✓ | ✓ |
| Table storage | ✓ | ✓ | | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | | |
| Twilio | ✓ | ✓ | | | ✓ |

¹ Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

² Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

³ Supported only 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](#).

Bindings code examples

Use the following table to find 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.

- [C#](#)
- [Java](#)
- [JavaScript](#)

- [PowerShell](#)
- [Python](#)

| SERVICE | EXAMPLES | SAMPLES |
|-----------------|--|----------------------|
| Blob storage | Trigger Input Output | Link |
| Azure Cosmos DB | 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 |
| SendGrid | Output | |
| 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](#).

Resources

- [Binding expressions and patterns](#)
- [Using the Azure Function return value](#)
- [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](#)

Next steps

[Register Azure Functions binding extensions](#)

Azure Functions trigger and binding example

11/2/2020 • 3 minutes to read • [Edit Online](#)

This article demonstrates how to configure a [trigger and bindings](#) in an Azure Function.

Suppose you want to write a new row to Azure Table storage whenever a new message appears in Azure Queue storage. This scenario can be implemented using an Azure Queue storage trigger and an Azure Table storage output binding.

Here's a *function.json* file for this scenario.

```
{  
  "bindings": [  
    {  
      "type": "queueTrigger",  
      "direction": "in",  
      "name": "order",  
      "queueName": "myqueue-items",  
      "connection": "MY_STORAGE_ACCT_APP_SETTING"  
    },  
    {  
      "type": "table",  
      "direction": "out",  
      "name": "$return",  
      "tableName": "outTable",  
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING"  
    }  
  ]  
}
```

The first element in the `bindings` array is the Queue storage trigger. The `type` and `direction` properties identify the trigger. The `name` property identifies the function parameter that receives the queue message content. The name of the queue to monitor is in `queueName`, and the connection string is in the app setting identified by `connection`.

The second element in the `bindings` array is the Azure Table Storage output binding. The `type` and `direction` properties identify the binding. The `name` property specifies how the function provides the new table row, in this case by using the function return value. The name of the table is in `tableName`, and the connection string is in the app setting identified by `connection`.

To view and edit the contents of *function.json* in the Azure portal, click the **Advanced editor** option on the **Integrate** tab of your function.

NOTE

The value of `connection` is the name of an app setting that contains the connection string, not the connection string itself. Bindings use connection strings stored in app settings to enforce the best practice that *function.json* does not contain service secrets.

C# script example

Here's C# script code that works with this trigger and binding. Notice that the name of the parameter that provides the queue message content is `order`; this name is required because the `name` property value in *function.json* is

order

```
#r "Newtonsoft.Json"

using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;

// From an incoming queue message that is a JSON object, add fields and write to Table storage
// The method return value creates a new row in Table Storage
public static Person Run(JObject order, ILogger log)
{
    return new Person() {
        PartitionKey = "Orders",
        RowKey = Guid.NewGuid().ToString(),
        Name = order["Name"].ToString(),
        MobileNumber = order["MobileNumber"].ToString() };
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

JavaScript example

The same *function.json* file can be used with a JavaScript function:

```
// From an incoming queue message that is a JSON object, add fields and write to Table Storage
// The second parameter to context.done is used as the value for the new row
module.exports = function (context, order) {
    order.PartitionKey = "Orders";
    order.RowKey = generateRandomId();

    context.done(null, order);
};

function generateRandomId() {
    return Math.random().toString(36).substring(2, 15) +
        Math.random().toString(36).substring(2, 15);
}
```

Class library example

In a class library, the same trigger and binding information — queue and table names, storage accounts, function parameters for input and output — is provided by attributes instead of a *function.json* file. Here's an example:

```
public static class QueueTriggerTableOutput
{
    [FunctionName("QueueTriggerTableOutput")]
    [return: Table("outTable", Connection = "MY_TABLE_STORAGE_ACCT_APP_SETTING")]
    public static Person Run(
        [QueueTrigger("myqueue-items", Connection = "MY_STORAGE_ACCT_APP_SETTING")] JObject order,
        ILogger log)
    {
        return new Person() {
            PartitionKey = "Orders",
            RowKey = Guid.NewGuid().ToString(),
            Name = order["Name"].ToString(),
            MobileNumber = order["MobileNumber"].ToString() };
    }
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

You now have a working function that is triggered by an Azure Queue and outputs data to Azure Table storage.

Next steps

[Azure Functions binding expression patterns](#)

Register Azure Functions binding extensions

11/2/2020 • 5 minutes to read • [Edit Online](#)

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 allows 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.

| DEVELOPMENT ENVIRONMENT | REGISTRATION IN FUNCTIONS 1.X | REGISTRATION IN FUNCTIONS 3.X/2.X |
|---|---------------------------------|--|
| 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.

Access extensions in non-.NET languages

For Java, JavaScript, PowerShell, Python, and Custom Handler function apps, we recommended using extension bundles to access bindings. In cases where extension bundles cannot be used, you can explicitly install binding extensions.

Extension bundles

Extension bundles is a way to add a compatible set of binding extensions to your function app. You enable extension bundles in the app's *host.json* file.

You can use extension bundles with version 2.x and later versions of the Functions runtime.

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.

To add an extension bundle to your function app, add the `extensionBundle` section to *host.json*. In many cases, Visual Studio Code and Azure Functions Core Tools will automatically add it for you.

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

The following properties are available in `extensionBundle`:

| PROPERTY | DESCRIPTION |
|----------------------|---|
| <code>id</code> | The namespace for Microsoft Azure Functions extension bundles. |
| <code>version</code> | The version of the bundle to install. The Functions runtime always picks the maximum permissible version defined by the version range or interval. The version value above allows all bundle versions from 1.0.0 up to but not including 2.0.0. For more information, see the interval notation for specifying version ranges . |

The following table lists the currently available versions of the default `Microsoft.Azure.Functions.ExtensionBundle` bundle and links to the extensions they include.

| BUNDLE VERSION | VERSION IN HOST.JSON | INCLUDED EXTENSIONS |
|----------------|----------------------------|---|
| 1.x | <code>[1.* , 2.0.0)</code> | See extensions.json used to generate the bundle |
| 2.x | <code>[2.* , 3.0.0)</code> | See extensions.json used to generate the bundle |

NOTE

While you can specify a custom version range in `host.json`, we recommend you use a version value from this table.

Explicitly install extensions

If you aren't able to use extension bundles, you can use Azure Functions Core Tools locally to install the specific extension packages required by your project.

IMPORTANT

You can't explicitly install extensions in a function app that is using extension bundles. Remove the `extensionBundle` section in `host.json` before explicitly installing extensions.

The following items describe some reasons 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.

NOTE

To manually install extensions by using Core Tools, you must have the [.NET Core 2.x SDK](#) installed. The .NET Core SDK is used by Azure Functions Core Tools to install extensions from NuGet. You don't need to know .NET to use Azure Functions extensions.

When you explicitly install extensions, a .NET project file named `extensions.csproj` is added to the root of your 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 the

file.

There are several ways to use Core Tools to install the required extensions in your local project.

Install all extensions

Use the following command to automatically add all extension packages used by the bindings in your local project:

```
func extensions install
```

The command reads the *function.json* file to see which packages you need, installs them, and rebuilds the extensions project (*extensions.csproj*). It adds any new bindings at the current version but does not update existing bindings. Use the `--force` option to update existing bindings to the latest version when installing new ones.

If your function app uses bindings that Core Tools does not recognize, you must manually install the specific extension.

Install a specific extension

Use the following command to install a specific extension package at a specific version, in this case the Storage extension:

```
func extensions install --package Microsoft.Azure.WebJobs.Extensions.Storage --version 4.0.2
```

Install extensions from NuGet in .NET languages

For a C# class library-based functions project, you should install extensions directly. Extension bundles are designed specifically for projects that aren't C# class library-based.

C# class library with Visual Studio

In **Visual Studio**, you can install packages from the Package Manager Console using the [Install-Package](#) command, as shown in the following example:

```
Install-Package Microsoft.Azure.WebJobs.Extensions.ServiceBus -Version <TARGET_VERSION>
```

The name of the package used for a given binding is provided in the reference article for that binding. For an example, see the [Packages section of the Service Bus binding reference article](#).

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 Functions runtime 1.x or 2.x are specified in the reference article for the binding.

If you use `Install-Package` to reference a binding, you don't need to use [extension bundles](#). This approach is specific for class libraries built in Visual Studio.

C# class library with Visual Studio Code

In **Visual Studio Code**, install packages for a C# class library project from the command prompt using the [dotnet add package](#) command in the .NET Core CLI. The following example demonstrates how you add a binding:

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.<BINDING_TYPE_NAME> --version <TARGET_VERSION>
```

The .NET Core CLI can only be used for Azure Functions 2.x development.

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 Functions runtime 1.x or 2.x are specified in the reference article for the binding.

Next steps

[Azure Function trigger and binding example](#)

Azure Functions binding expression patterns

11/2/2020 • 6 minutes to read • [Edit Online](#)

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.

```
{
  "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:

```
[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`:

```
{
  "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:

```
...
{
  "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# 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:

```
[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`

```
{
    "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:

```

"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

When a trigger payload is JSON, you can refer to its properties in configuration for other bindings in the same function and in function code.

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}" )
```

```
{
  "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:

```

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.

```

module.exports = function (context, info) {
    if ('BlobName' in info) {
        context.res = {
            body: { 'data': context.bindings.blobContents }
        }
    } else {
        context.res = {
            status: 404
        };
    }
    context.done();
}

```

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. For example, suppose your JSON looks like this:

```
{
    "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:

```
"path": "strings/{BlobName.FileName}.{BlobName.Extension}",
```

In C#, you would need two classes:

```
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`.

```
{
    "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`.

```
{
    "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](#)

Using the Azure Function return value

12/4/2020 • 2 minutes to read • [Edit Online](#)

This article explains how return values work inside a function.

In languages that have a return value, you can bind a function [output binding](#) to the return value:

- In a C# class library, apply the output binding attribute to the method return value.
- In Java, apply the output binding annotation to the function method.
- In other languages, set the `name` property in `function.json` to `$return`.

If there are multiple output bindings, use the return value for only one of them.

In C# and C# script, alternative ways to send data to an output binding are `out` parameters and [collector objects](#).

- [C#](#)
- [C# Script](#)
- [F#](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)
- [Java](#)

Here's C# code that uses the return value for an output binding, followed by an async example:

```
[FunctionName("QueueTrigger")]
[return: Blob("output-container/{id}")]
public static string Run([QueueTrigger("inputqueue")]WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return json;
}
```

```
[FunctionName("QueueTrigger")]
[return: Blob("output-container/{id}")]
public static Task<string> Run([QueueTrigger("inputqueue")]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);
}
```

Next steps

[Handle Azure Functions binding errors](#)

Handle Azure Functions binding errors

12/4/2020 • 6 minutes to read • [Edit Online](#)

Errors raised in an Azure Functions can come from any of the following origins:

- Use of built-in Azure Functions [triggers and bindings](#)
- Calls to APIs of underlying Azure services
- Calls to REST endpoints
- Calls to client libraries, packages, or third-party APIs

Following good error handling practices is important to avoid loss of data or missed messages. Recommended error handling practices include the following actions:

- [Enable Application Insights](#)
- [Use structured error handling](#)
- [Design for idempotency](#)
- [Implement retry policies](#) (where appropriate)

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 try/catch block. In the catch block, you can capture and log errors.

Retry policies (preview)

A retry policy can be defined on any function for any trigger type in your function app. The retry policy re-executes a function until either successful execution or until the maximum number of retries occur. Retry policies can be defined for all functions in an app or for individual functions. By default, a function app won't retry messages (aside from the [specific triggers that have a retry policy on the trigger source](#)). A retry policy is evaluated whenever an execution results in an uncaught exception. As a best practice, you should catch all exceptions in your code and rethrow any errors that should result in a retry. Event Hubs and Azure Cosmos DB checkpoints won't be written until the retry policy for the execution has completed, meaning progressing on that partition is paused until the current batch has completed.

Retry policy options

The following options are available for defining a retry policy.

Max Retry Count is the maximum number of times an execution is retried before eventual failure. A value of `-1` means to retry indefinitely. The current retry count is stored in memory of the instance. It's possible that an instance has a failure between retry attempts. When an instance fails during a retry policy, the retry count is lost. When there are instance failures, triggers like Event Hubs, Azure Cosmos DB, and Queue storage are able to resume processing and retry the batch on a new instance, with the retry count reset to zero. Other triggers, like HTTP and timer, don't resume on a new instance. This means that the max retry count is a best effort, and in some rare cases an execution could be retried more than the maximum, or for triggers like HTTP and timer be retried less than the maximum.

Retry Strategy controls how retries behave. The following are two supported retry options:

| OPTION | DESCRIPTION |
|--------|-------------|
|--------|-------------|

| OPTION | DESCRIPTION |
|--------------------|---|
| fixedDelay | A specified amount of time is allowed to elapse between each retry. |
| exponentialBackoff | The first retry waits for the minimum delay. On subsequent retries, time is added exponentially to the initial duration for each retry, until the maximum delay is reached. Exponential back-off adds some small randomization to delays to stagger retries in high-throughput scenarios. |

App level configuration

A retry policy can be defined for all functions in an app using the [host.json](#) file.

Function level configuration

A retry policy can be defined for a specific function. Function-specific configuration takes priority over app-level configuration.

Fixed delay retry

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)
- [PowerShell](#)

Retries require NuGet package [Microsoft.Azure.WebJobs](#) >= 3.0.23

```
[FunctionName("EventHubTrigger")]
[FixedDelayRetry(5, "00:00:10")]
public static async Task Run([EventHubTrigger("myHub", Connection = "EventHubConnection")] EventData[] events,
ILogger log)
{
// ...
}
```

Exponential backoff retry

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)
- [PowerShell](#)

Retries require NuGet package [Microsoft.Azure.WebJobs](#) >= 3.0.23

```
[FunctionName("EventHubTrigger")]
[ExponentialBackoffRetry(5, "00:00:04", "00:15:00")]
public static async Task Run([EventHubTrigger("myHub", Connection = "EventHubConnection")] EventData[] events,
ILogger log)
{
// ...
}
```

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| strategy | n/a | Required. The retry strategy to use. Valid values are <code>fixedDelay</code> or <code>exponentialBackoff</code> . |
| maxRetryCount | n/a | Required. The maximum number of retries allowed per function execution. <code>-1</code> means to retry indefinitely. |
| delayInterval | n/a | The delay that will be used between retries when using <code>fixedDelay</code> strategy. |
| minimumInterval | n/a | The minimum retry delay when using <code>exponentialBackoff</code> strategy. |
| maximumInterval | n/a | The maximum retry delay when using <code>exponentialBackoff</code> strategy. |

Retry limitations during preview

- For .NET projects, you may need to manually pull in a version of [Microsoft.Azure.WebJobs](#) >= 3.0.23.
- In the consumption plan, the app may be scaled down to zero while retrying the final messages in a queue.
- In the consumption plan, the app may be scaled down while performing retries. For best results, choose a retry interval <= 00:01:00 and <= 5 retries.

Using retry support on top of trigger resilience

The function app retry policy is independent of any retries or resiliency that the trigger provides. The function retry policy will only layer on top of a trigger resilient retry. For example, if using Azure Service Bus, by default queues have a message delivery count of 10. The default delivery count means after 10 attempted deliveries of a queue message, Service Bus will dead-letter the message. You can define a retry policy for a function that has a Service Bus trigger, but the retries will layer on top of the Service Bus delivery attempts.

For instance, if you used the default Service Bus delivery count of 10, and defined a function retry policy of 5. The message would first dequeue, incrementing the service bus delivery account to 1. If every execution failed, after five attempts to trigger the same message, that message would be marked as abandoned. Service Bus would immediately requeue the message, it would trigger the function and increment the delivery count to 2. Finally, after 50 eventual attempts (10 service bus deliveries * five function retries per delivery), the message would be abandoned and trigger a dead-letter on service bus.

WARNING

It is not recommended to set the delivery count for a trigger like Service Bus Queues to 1, meaning the message would be dead-lettered immediately after a single function retry cycle. This is because triggers provide resiliency with retries, while the function retry policy is best effort and may result in less than the desired total number of retries.

Triggers with additional resiliency or retries

The following triggers support retries at the trigger source:

- [Azure Blob storage](#)
- [Azure Queue storage](#)
- [Azure Service Bus \(queue/topic\)](#)

By default, most triggers retry requests up to five times. After the fifth retry, both the Azure Queue storage will write a message to a [poison queue](#). The default Service Bus queue and topic policy will write a message to a [dead-letter queue](#) after 10 attempts.

For information on errors returned by services supported by Functions, see the [Binding error codes](#) section of the [Azure Functions error handling](#) overview article.

Supported languages in Azure Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

This article explains the levels of support offered for languages that you can use with Azure Functions. It also describes strategies for creating functions using languages not natively supported.

Levels of support

There are two levels of support:

- **Generally available (GA)** - Fully supported and approved for production use.
- **Preview** - Not yet supported but is expected to reach GA status in the future.

Languages by runtime version

Three versions of the Azure Functions runtime are available. The following table shows which languages are supported in each runtime version.

| LANGUAGE | 1.X | 2.X | 3.X |
|------------|-------------------------|------------------------|--|
| C# | GA (.NET Framework 4.7) | GA (.NET Core 2.2) | GA (.NET Core 3.1) |
| JavaScript | GA (Node 6) | GA (Node 10 & 8) | GA (Node 12 & 10) Preview (Node 14) |
| F# | GA (.NET Framework 4.7) | GA (.NET Core 2.2) | GA (.NET Core 3.1) |
| Java | N/A | GA (Java 8) | GA (Java 11 & 8) |
| PowerShell | N/A | GA (PowerShell Core 6) | GA (PowerShell 7 & Core 6) |
| Python | N/A | GA (Python 3.7 & 3.6) | GA (Python 3.8, 3.7, & 3.6) |
| TypeScript | N/A | GA* | GA* |

* Supported through transpiling to JavaScript.

See the language-specific developer guide article for more details about supported language versions.

For information about planned changes to language support, see [Azure roadmap](#).

Custom handlers (preview)

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 \(preview\)](#).

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

To learn more about how to develop functions in the supported languages, see the following resources:

- [C# class library developer reference](#)
- [C# script developer reference](#)
- [Java developer reference](#)
- [JavaScript developer reference](#)
- [PowerShell developer reference](#)
- [Python developer reference](#)
- [TypeScript developer reference](#)

Azure Functions C# developer reference

12/4/2020 • 17 minutes to read • [Edit Online](#)

This article is an introduction to developing Azure Functions by using C# in .NET class libraries.

As a C# developer, you may also be interested in one of the following articles:

| GETTING STARTED | CONCEPTS | GUIDED LEARNING/SAMPLES |
|---|---|---|
| <ul style="list-style-type: none">• Using Visual Studio• Using Visual Studio Code• Using command line tools | <ul style="list-style-type: none">• Hosting options• Performance considerations• Visual Studio development• Dependency injection | <ul style="list-style-type: none">• Create serverless applications• C# 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 work with specific versions of .NET. The following table shows the highest level of .NET Core and .NET Framework and .NET Core that can be used with a specific version of Functions in your project.

| FUNCTIONS RUNTIME VERSION | MAX .NET VERSION |
|---------------------------|--------------------|
| Functions 3.x | .NET Core 3.1 |
| Functions 2.x | .NET Core 2.2 |
| Functions 1.x | .NET Framework 4.7 |

To learn more, see [Azure Functions runtime versions overview](#)

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 static method with a `[FunctionName]` and a trigger attribute, as shown in the following example:

```
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 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 additional 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 does not 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 binding example

The following example modifies the preceding one by adding an output queue binding. The function writes the queue message that triggers the function to a new queue message in a different queue.

```
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;
    }
}
```

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.

```
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 is not 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 or 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:

```
{
  "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 same package is used for both version 1.x and 2.x of the Functions runtime. The target framework is what differentiates a 1.x project from a 2.x project. Here are the relevant parts of `.csproj` files, showing different target frameworks and the same `Sdk` package:

- `v2.x+`
- `v1.x`

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
  <AzureFunctionsVersion>v2</AzureFunctionsVersion>
</PropertyGroup>
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Sdk.Functions" Version="1.0.8" />
</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 2.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](#).

Runtime version

Visual Studio uses the [Azure Functions Core Tools](#) to run Functions projects. The Core Tools is a command-line interface for the Functions runtime.

If you install the Core Tools by using npm, that 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 2.x, the Core Tools are included in the [Azure Functions and Web Jobs Tools](#) extension.

For both 1.x and 2.x, you can see what version is being used in the console output when you run a Functions project:

```
[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 3.0 and requires [version 3.0 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.

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
  <AzureFunctionsVersion>v3</AzureFunctionsVersion>
  <PublishReadyToRun>true</PublishReadyToRun>
  <RuntimeIdentifier>win-x86</RuntimeIdentifier>
</PropertyGroup>
```

IMPORTANT

ReadyToRun currently doesn't support cross-compilation. You must build your app on the same platform as the deployment target. Also, pay attention to the "bitness" that is configured in your function app. For example, if your function app in Azure is Windows 64-bit, you must compile your app on Windows with `win-x64` as the [runtime identifier](#).

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`:

```
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.

```
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.

The following example shows how to check for impending function termination.

```

public static class CancellationTokenExample
{
    public static void Run(
        [QueueTrigger("inputqueue")] 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);
        }
    }
}

```

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..`:

```

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 are 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:

```

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:

```
{  
    "customDimensions": {  
        "prop__{OriginalFormat}": "C# Queue trigger function processed: {message}",  
        "Category": "Function",  
        "LogLevel": "Information",  
        "prop__message": "c9519cbf-b1e6-4b9b-bf24-cb7d10b1bb89"  
    }  
}
```

Log custom telemetry in C# functions

There is 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](#)
- [PowerShell](#)

```
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.

- [v2.x+](#)
- [v1.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.

```

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.ContentLength);

            // 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.

Environment variables

To get an environment variable or an app setting value, use

`System.Environment.GetEnvironmentVariable`, as shown in the following code example:

```
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"));
    }

    public 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.

```
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` cannot 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.

```
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 attribute 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:

```

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:

| TYPE | 1.X | 2.X AND HIGHER ¹ | TRIGGER | INPUT | OUTPUT |
|-----------------------|-----|-----------------------------|---------|-------|--------|
| Blob storage | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure Cosmos DB | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dapr ³ | | ✓ | ✓ | ✓ | ✓ |
| Event Grid | ✓ | ✓ | ✓ | | ✓ |
| Event Hubs | ✓ | ✓ | ✓ | | ✓ |
| HTTP & webhooks | ✓ | ✓ | ✓ | | ✓ |
| IoT Hub | ✓ | ✓ | ✓ | | ✓ |
| Kafka ² | | ✓ | ✓ | | ✓ |
| Mobile Apps | ✓ | | | ✓ | ✓ |
| Notification Hubs | ✓ | | | | ✓ |
| Queue storage | ✓ | ✓ | ✓ | | ✓ |
| RabbitMQ ² | | ✓ | ✓ | | ✓ |

| Type | 1.x | 2.x and higher | Trigger | Input | Output |
|---------------|-----|----------------|---------|-------|--------|
| SendGrid | ✓ | ✓ | | | ✓ |
| Service Bus | ✓ | ✓ | ✓ | | ✓ |
| SignalR | | ✓ | | ✓ | ✓ |
| Table storage | ✓ | ✓ | | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | | |
| Twilio | ✓ | ✓ | | | ✓ |

¹ Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

² Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

³ 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](#)

Azure Functions C# script (.csx) developer reference

11/2/2020 • 11 minutes to read • [Edit Online](#)

This article is an introduction to developing Azure Functions by using C# script (.csx).

Azure Functions supports C# and C# script programming languages. If you're looking for guidance on using C# in a Visual Studio class library project, see [C# developer reference](#).

This article assumes that you've already read the [Azure Functions developers guide](#).

How .csx works

The C# script experience for Azure Functions is based on the [Azure WebJobs SDK](#). 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 are not.

Folder structure

The folder structure for a C# script project looks like the following:

```
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 developing 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.

```
{
    "disabled": false,
    "bindings": [
        {
            "type": "queueTrigger",
            "direction": "in",
            "name": "myQueueItem",
            "queueName": "myqueue-items",
            "connection": "MyStorageConnectionAppSetting"
        }
    ]
}
```

```
#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.

```
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*:

```
#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*:

```
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:

```
#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:

```
#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...");
    log.LogInformation(myQueueItem.ToString());

    outputQueueItem = myQueueItem;
}
```

Example *order.csx*:

```

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`. 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`:

```

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.

```

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:

```
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.

```
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.

```
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.

```
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.

```
#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:

- `mscorlib`
- `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 (for example, `#r "AssemblyName"`):

- `Newtonsoft.Json`
- `Microsoft.WindowsAzure.Storage`
- `Microsoft.ServiceBus`
- `Microsoft.AspNet.WebHooks.Receivers`
- `Microsoft.AspNet.WebHooks.Common`
- `Microsoft.Azure.NotificationHubs`

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 your `function app root folder` (`wwwroot`).

- 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

To use NuGet packages in a 2.x and later C# function, upload a `function.proj` file to the function's folder in the function app's file system. Here is an example `function.proj` file that adds a reference to `Microsoft.ProjectOxford.Face` version `1.1.0`:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.ProjectOxford.Face" Version="1.1.0" />
  </ItemGroup>
</Project>
```

To use a custom NuGet feed, specify the feed in a `Nuget.Config` file in the Function App root. For more information, see [Configuring NuGet behavior](#).

NOTE

In 1.x C# functions, NuGet packages are referenced with a `project.json` file instead of a `function.proj` file.

For 1.x functions, use a `project.json` file instead. Here is an example `project.json` file:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Microsoft.ProjectOxford.Face": "1.1.0"
      }
    }
  }
}
```

Using a `function.proj` file

1. Open the function in the Azure portal. The logs tab displays the package installation output.
2. To upload a `function.proj` file, use one of the methods described in the [How to update function app files](#) in the Azure Functions developer reference topic.
3. After the `function.proj` file is uploaded, you see output like the following example in your function's streaming log:

```
2018-12-14T22:00:48.658 [Information] Restoring packages.
2018-12-14T22:00:48.681 [Information] Starting packages restore
2018-12-14T22:00:57.064 [Information] Restoring packages for D:\local\Temp\9e814101-fe35-42aa-ada5-f8435253eb83\function.proj...
2016-04-04T19:02:50.511 Restoring packages for D:\home\site\wwwroot\HttpTriggerCSharp1\function.proj...
2018-12-14T22:01:00.844 [Information] Installing Newtonsoft.Json 10.0.2.
2018-12-14T22:01:01.041 [Information] Installing Microsoft.ProjectOxford.Common.DotNetStandard 1.0.0.
2018-12-14T22:01:01.140 [Information] Installing Microsoft.ProjectOxford.Face.DotNetStandard 1.0.0.
2018-12-14T22:01:09.799 [Information] Restore completed in 5.79 sec for D:\local\Temp\9e814101-fe35-42aa-ada5-f8435253eb83\function.proj.
2018-12-14T22:01:10.905 [Information] Packages restored.
```

Environment variables

To get an environment variable or an app setting value, use `System.Environment.GetEnvironmentVariable`, as shown in the following code example:

```
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);
}
```

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.

```
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` cannot 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.

```
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 attribute 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:

```

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.Write("Hello World!");
    }
}

```

The following table lists the .NET attributes for each binding type and the packages in which they are defined.

| BINDING | ATTRIBUTE | ADD REFERENCE |
|-------------------|--|---|
| Cosmos DB | Microsoft.Azure.WebJobs.DocumentDBAttribute #r "Microsoft.Azure.WebJobs.Extensions.CosmosDB" | |
| Event Hubs | Microsoft.Azure.WebJobs.ServiceBus.EventHubAttribute #r "Microsoft.Azure.Jobs.ServiceBus" Microsoft.Azure.WebJobs.ServiceBusAccountAttribute | |
| Mobile Apps | Microsoft.Azure.WebJobs.MobileTableAttribute #r "Microsoft.Azure.WebJobs.Extensions.MobileApp" | |
| Notification Hubs | Microsoft.Azure.WebJobs.NotificationHubAttribute #r "Microsoft.Azure.WebJobs.Extensions.NotificationHub" | |
| Service Bus | Microsoft.Azure.WebJobs.ServiceBusAttribute #r "Microsoft.Azure.WebJobs.ServiceBus" Microsoft.Azure.WebJobs.ServiceBusAccountAttribute | |
| 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" | |

Next steps

[Learn more about triggers and bindings](#)

[Learn more about best practices for Azure Functions](#)

Azure Functions F# Developer Reference

11/2/2020 • 7 minutes to read • [Edit Online](#)

F# for Azure Functions is a solution for easily running small pieces of code, or "functions," in the cloud. Data flows into your F# function via function arguments. Argument names are specified in `function.json`, and there are predefined names for accessing things like the function logger and cancellation tokens.

IMPORTANT

F# script (.fsx) is only supported by [version 1.x](#) of the Azure Functions runtime. If you want to use F# with version 2.x and later versions of the runtime, you must use a precompiled F# class library project (.fs). You create, manage, and publish an F# class library project using Visual Studio as you would a [C# class library project](#). For more information about Functions versions, see [Azure Functions runtime versions overview](#).

This article assumes that you've already read the [Azure Functions developer reference](#).

How .fsx works

An `.fsx` file is an F# script. It can be thought of as an F# project that's contained in a single file. The file contains both the code for your program (in this case, your Azure Function) and directives for managing dependencies.

When you use an `.fsx` for an Azure Function, commonly required assemblies are automatically included for you, allowing you to focus on the function rather than "boilerplate" code.

Folder structure

The folder structure for an F# script project looks like the following:

```
FunctionsProject
| - MyFirstFunction
| | - run.fsx
| | - function.json
| | - function.proj
| - MySecondFunction
| | - run.fsx
| | - 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 (`.fsx`) 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 developing functions in the Azure portal, this registration is done for you.

Binding to arguments

Each binding supports some set of arguments, as detailed in the [Azure Functions triggers and bindings developer reference](#). For example, one of the argument bindings a blob trigger supports is a POCO, which can be expressed

using an F# record. For example:

```
type Item = { Id: string }

let Run(blob: string, output: byref<Item>) =
    let item = { Id = "Some ID" }
    output <- item
```

Your F# Azure Function will take one or more arguments. When we talk about Azure Functions arguments, we refer to *input* arguments and *output* arguments. An input argument is exactly what it sounds like: input to your F# Azure Function. An *output* argument is mutable data or a `byref<>` argument that serves as a way to pass data back *out* of your function.

In the example above, `blob` is an input argument, and `output` is an output argument. Notice that we used `byref<>` for `output` (there's no need to add the `[<Out>]` annotation). Using a `byref<>` type allows your function to change which record or object the argument refers to.

When an F# record is used as an input type, the record definition must be marked with `[<CLIMutable>]` in order to allow the Azure Functions framework to set the fields appropriately before passing the record to your function. Under the hood, `[<CLIMutable>]` generates setters for the record properties. For example:

```
[<CLIMutable>]
type TestObject =
    { SenderName : string
      Greeting : string }

let Run(req: TestObject, log: ILogger) =
    { req with Greeting = sprintf "Hello, %s" req.SenderName }
```

An F# class can also be used for both in and out arguments. For a class, properties will usually need getters and setters. For example:

```
type Item() =
    member val Id = "" with get, set
    member val Text = "" with get, set

let Run(input: string, item: byref<Item>) =
    let result = Item(Id = input, Text = "Hello from F#!")
    item <- result
```

Logging

To log output to your [streaming logs](#) in F#, your function should take an argument of type `ILogger`. For consistency, we recommend this argument is named `log`. For example:

```
let Run(blob: string, output: byref<string>, log: ILogger) =
    log.LogInformation(sprintf "F# Azure Function processed a blob: %s" blob)
    output <- input
```

Async

The `async` workflow can be used, but the result needs to return a `Task`. This can be done with `Async.StartAsTask`, for example:

```
let Run(req: HttpRequestMessage) =
    async {
        return new HttpResponseMessage(HttpStatusCode.OK)
    } |> Async.StartAsTask
```

Cancellation Token

If your function needs to handle shutdown gracefully, you can give it a `CancellationToken` argument. This can be combined with `async`, for example:

```
let Run(req: HttpRequestMessage, token: CancellationToken)
    let f = async {
        do! Async.Sleep(10)
        return new HttpResponseMessage(HttpStatusCode.OK)
    }
    Async.StartAsTask(f, token)
```

Importing namespaces

Namespaces can be opened in the usual way:

```
open System.Net
open System.Threading.Tasks
open Microsoft.Extensions.Logging

let Run(req: HttpRequestMessage, log: ILogger) =
    ...
```

The following namespaces are automatically opened:

- `System`
- `System.Collections.Generic`
- `System.IO`
- `System.Linq`
- `System.Net.Http`
- `System.Threading.Tasks`
- `Microsoft.Azure.WebJobs`
- `Microsoft.Azure.WebJobs.Host`.

Referencing External Assemblies

Similarly, framework assembly references can be added with the `#r "AssemblyName"` directive.

```
#r "System.Web.Http"

open System.Net
open System.Net.Http
open System.Threading.Tasks
open Microsoft.Extensions.Logging

let Run(req: HttpRequestMessage, log: ILogger) =
    ...
```

The following assemblies are automatically added by the Azure Functions hosting environment:

- `mscorlib`,
- `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`.

In addition, the following assemblies are special cased and may be referenced by `simplename` (e.g.

```
#r "AssemblyName"):
```

- `Newtonsoft.Json`
- `Microsoft.WindowsAzure.Storage`
- `Microsoft.ServiceBus`
- `Microsoft.AspNet.WebHooks.Receivers`
- `Microsoft.AspNet.WebHooks.Common`.

If you need to reference a private assembly, you can upload the assembly file into a `bin` folder relative to your function and reference it by using the file name (e.g. `#r "MyAssembly.dll"`). For information on how to upload files to your function folder, see the following section on package management.

Editor Prelude

An editor that supports F# Compiler Services will not be aware of the namespaces and assemblies that Azure Functions automatically includes. As such, it can be useful to include a prelude that helps the editor find the assemblies you are using, and to explicitly open namespaces. For example:

```
#if !COMPILED
#I "../../bin/Binaries/WebJobs.Script.Host"
#r "Microsoft.Azure.WebJobs.Host.dll"
#endif

open System
open Microsoft.Azure.WebJobs.Host
open Microsoft.Extensions.Logging

let Run(blob: string, output: byref<string>, log: ILogger) =
    ...
```

When Azure Functions executes your code, it processes the source with `COMPILED` defined, so the editor prelude will be ignored.

Package management

To use NuGet packages in an F# function, add a `project.json` file to the function's folder in the function app's file system. Here is an example `project.json` file that adds a NuGet package reference to `Microsoft.ProjectOxford.Face` version 1.1.0:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Microsoft.ProjectOxford.Face": "1.1.0"
      }
    }
  }
}
```

Only the .NET Framework 4.6 is supported, so make sure that your `project.json` file specifies `net46` as shown here.

When you upload a `project.json` file, the runtime gets the packages and automatically adds references to the package assemblies. You don't need to add `#r "AssemblyName"` directives. Just add the required `open` statements to your `.fsx` file.

You may wish to put automatically references assemblies in your editor prelude, to improve your editor's interaction with F# Compile Services.

How to add a `project.json` file to your Azure Function

1. Begin by making sure your function app is running, which you can do by opening your function in the Azure portal. This also gives access to the streaming logs where package installation output will be displayed.
2. To upload a `project.json` file, use one of the methods described in [how to update function app files](#). If you are using [Continuous Deployment for Azure Functions](#), you can add a `project.json` file to your staging branch in order to experiment with it before adding it to your deployment branch.
3. After the `project.json` file is added, you will see output similar to the following example in your function's streaming log:

```
2016-04-04T19:02:48.745 Restoring packages.
2016-04-04T19:02:48.745 Starting NuGet restore
2016-04-04T19:02:50.183 MSBuild auto-detection: using msbuild version '14.0' from 'D:\Program Files (x86)\MSBuild\14.0\bin'.
2016-04-04T19:02:50.261 Feeds used:
2016-04-04T19:02:50.261 C:\DWASFiles\Sites\facaivalfunctest\LocalAppData\NuGet\Cache
2016-04-04T19:02:50.261 https://api.nuget.org/v3/index.json
2016-04-04T19:02:50.261
2016-04-04T19:02:50.511 Restoring packages for D:\home\site\wwwroot\HttpTriggerCSharp1\Project.json...
2016-04-04T19:02:52.800 Installing Newtonsoft.Json 6.0.8.
2016-04-04T19:02:52.800 Installing Microsoft.ProjectOxford.Face 1.1.0.
2016-04-04T19:02:57.095 All packages are compatible with .NETFramework,Version=v4.6.
2016-04-04T19:02:57.189
2016-04-04T19:02:57.189
2016-04-04T19:02:57.455 Packages restored.
```

Environment variables

To get an environment variable or an app setting value, use `System.Environment.GetEnvironmentVariable`, for example:

```
open System.Environment
open Microsoft.Extensions.Logging

let Run(timer: TimerInfo, log: ILogger) =
    log.LogInformation("Storage = " + GetEnvironmentVariable("AzureWebJobsStorage"))
    log.LogInformation("Site = " + GetEnvironmentVariable("WEBSITE_SITE_NAME"))
```

Reusing .fsx code

You can use code from other `.fsx` files by using a `#load` directive. For example:

`run.fsx`

```
#load "logger.fsx"

let Run(timer: TimerInfo, log: ILogger) =
    mylog log (sprintf "Timer: %s" DateTime.Now.ToString())
```

`logger.fsx`

```
let mylog(log: ILogger, text: string) =
    log.LogInformation(text);
```

Paths provided to the `#load` directive are relative to the location of your `.fsx` file.

- `#load "logger.fsx"` loads a file located in the function folder.
- `#load "package\logger.fsx"` loads a file located in the `package` folder in the function folder.
- `#load "..\shared\mylogger.fsx"` loads a file located in the `shared` folder at the same level as the function folder, that is, directly under `wwwroot`.

The `#load` directive only works with `.fsx` (F# script) files, and not with `.fs` files.

Next steps

For more information, see the following resources:

- [F# Guide](#)
- [Best Practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)
- [Azure Functions testing](#)
- [Azure Functions scaling](#)

Azure Functions JavaScript developer guide

12/4/2020 • 24 minutes to read • [Edit Online](#)

This guide contains detailed information to help you succeed developing Azure Functions using JavaScript.

As an Express.js, Node.js, or JavaScript developer, if you are new to Azure Functions, please consider first reading one of the following articles:

| GETTING STARTED | CONCEPTS | GUIDED LEARNING |
|---|---|---|
| <ul style="list-style-type: none">• Node.js function using Visual Studio Code• Node.js function with terminal/command prompt | <ul style="list-style-type: none">• Developer guide• Hosting options• TypeScript functions• Performance considerations | <ul style="list-style-type: none">• Create serverless applications• Refactor Node.js and Express APIs to Serverless APIs |

JavaScript function basics

A JavaScript (Node.js) function is an exported `function` that executes when triggered ([triggers are configured in `function.json`](#)). The first argument passed to every function is a `context` object, which is used for receiving and sending binding data, logging, and communicating with the runtime.

Folder structure

The required folder structure for a JavaScript project looks like the following. This default can be changed. For more information, see the `scriptFile` section below.

```
FunctionsProject
| - MyFirstFunction
| | - index.js
| | - function.json
| - MySecondFunction
| | - index.js
| | - function.json
| - SharedCode
| | - myFirstHelperFunction.js
| | - mySecondHelperFunction.js
| - node_modules
| - host.json
| - package.json
| - extensions.csproj
```

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 (.js) and binding configuration file (`function.json`). The name of `function.json`'s parent directory is always the name of your function.

The binding extensions required in [version 2.x](#) 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.

Exporting a function

JavaScript functions must be exported via `module.exports` (or `exports`). Your exported function should be a

JavaScript function that executes when triggered.

By default, the Functions runtime looks for your function in `index.js`, where `index.js` shares the same parent directory as its corresponding `function.json`. In the default case, your exported function should be the only export from its file or the export named `run` or `index`. To configure the file location and export name of your function, read about [configuring your function's entry point](#) below.

Your exported function is passed a number of arguments on execution. The first argument it takes is always a `context` object. If your function is synchronous (doesn't return a Promise), you must pass the `context` object, as calling `context.done` is required for correct use.

```
// You should include context, other arguments are optional
module.exports = function(context, myTrigger, myInput, myOtherInput) {
    // function logic goes here :)
    context.done();
};
```

Exporting an async function

When using the `async function` declaration or plain JavaScript [Promises](#) in version 2.x of the Functions runtime, you do not need to explicitly call the `context.done` callback to signal that your function has completed. Your function completes when the exported async function/Promise completes. For functions targeting the version 1.x runtime, you must still call `context.done` when your code is done executing.

The following example is a simple function that logs that it was triggered and immediately completes execution.

```
module.exports = async function (context) {
    context.log('JavaScript trigger function processed a request.');
};
```

When exporting an async function, you can also configure an output binding to take the `return` value. This is recommended if you only have one output binding.

To assign an output using `return`, change the `name` property to `$return` in `function.json`.

```
{
  "type": "http",
  "direction": "out",
  "name": "$return"
}
```

In this case, your function should look like the following example:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');
    // You can call and await an async method here
    return {
        body: "Hello, world!"
    };
}
```

Bindings

In JavaScript, [bindings](#) are configured and defined in a function's `function.json`. Functions interact with bindings a number of ways.

Inputs

Input are divided into two categories in Azure Functions: one is the trigger input and the other is the additional input. Trigger and other input bindings (bindings of `direction === "in"`) can be read by a function in three ways:

- **[Recommended] As parameters passed to your function.** They are passed to the function in the same order that they are defined in `function.json`. The `name` property defined in `function.json` does not need to match the name of your parameter, although it should.

```
module.exports = async function(context, myTrigger, myInput, myOtherInput) { ... };
```

- **As members of the `context.bindings` object.** Each member is named by the `name` property defined in `function.json`.

```
module.exports = async function(context) {
    context.log("This is myTrigger: " + context.bindings.myTrigger);
    context.log("This is myInput: " + context.bindings.myInput);
    context.log("This is myOtherInput: " + context.bindings.myOtherInput);
};
```

- **As inputs using the JavaScript `arguments` object.** This is essentially the same as passing inputs as parameters, but allows you to dynamically handle inputs.

```
module.exports = async function(context) {
    context.log("This is myTrigger: " + arguments[1]);
    context.log("This is myInput: " + arguments[2]);
    context.log("This is myOtherInput: " + arguments[3]);
};
```

Outputs

Outputs (bindings of `direction === "out"`) can be written to by a function in a number of ways. In all cases, the `name` property of the binding as defined in `function.json` corresponds to the name of the object member written to in your function.

You can assign data to output bindings in one of the following ways (don't combine these methods):

- **[Recommended for multiple outputs] Returning an object.** If you are using an `async/Promise` returning function, you can return an object with assigned output data. In the example below, the output bindings are named "httpResponse" and "queueOutput" in `function.json`.

```
module.exports = async function(context) {
    let retMsg = 'Hello, world!';
    return {
        httpResponse: {
            body: retMsg
        },
        queueOutput: retMsg
    };
};
```

If you are using a synchronous function, you can return this object using `context.done` (see example).

- **[Recommended for single output] Returning a value directly and using the `$return` binding name.** This only works for `async/Promise` returning functions. See example in [exporting an async function](#).
- **Assigning values to `context.bindings`** You can assign values directly to `context.bindings`.

```

module.exports = async function(context) {
    let retMsg = 'Hello, world!';
    context.bindings.httpResponse = {
        body: retMsg
    };
    context.bindings.queueOutput = retMsg;
    return;
};

```

Bindings data type

To define the data type for an input binding, use the `dataType` property in the binding definition. For example, to read the content of an HTTP request in binary format, use the type `binary`:

```
{
    "type": "httpTrigger",
    "name": "req",
    "direction": "in",
    "dataType": "binary"
}
```

Options for `dataType` are: `binary`, `stream`, and `string`.

context object

The runtime uses a `context` object to pass data to and from your function and the runtime. Used to read and set data from bindings and for writing to logs, the `context` object is always the first parameter passed to a function.

For functions featuring synchronous code, the context object includes the `done` callback which you call when the function is done processing. Explicitly calling `done` is unnecessary when writing asynchronous code; the `done` callback is called implicitly.

```

module.exports = (context) => {

    // Function logic goes here

    context.log("The function has executed.");

    context.done();
};

```

The context passed into your function exposes an `executionContext` property, which is an object with the following properties:

| PROPERTY NAME | TYPE | DESCRIPTION |
|--------------------------------|--------|--|
| <code>invocationId</code> | String | Provides a unique identifier for the specific function invocation. |
| <code>functionName</code> | String | Provides the name of the running function |
| <code>functionDirectory</code> | String | Provides the functions app directory. |

The following example shows how to return the `invocationId`.

```
module.exports = (context, req) => {
    context.res = {
        body: context.executionContext.invocationId
    };
    context.done();
};
```

context.bindings property

```
context.bindings
```

Returns a named object that is used to read or assign binding data. Input and trigger binding data can be accessed by reading properties on `context.bindings`. Output binding data can be assigned by adding data to `context.bindings`.

For example, the following binding definitions in your function.json let you access the contents of a queue from `context.bindings.myInput` and assign outputs to a queue using `context.bindings.myOutput`.

```
{
    "type": "queue",
    "direction": "in",
    "name": "myInput"
    ...
},
{
    "type": "queue",
    "direction": "out",
    "name": "myOutput"
    ...
}
```

```
// myInput contains the input data, which may have properties such as "name"
var author = context.bindings.myInput.name;
// Similarly, you can set your output data
context.bindings.myOutput = {
    some_text: 'hello world',
    a_number: 1
};
```

You can choose to define output binding data using the `context.done` method instead of the `context.binding` object (see below).

context.bindingData property

```
context.bindingData
```

Returns a named object that contains trigger metadata and function invocation data (`invocationId`, `sys.methodName`, `sys.utcNow`, `sys.randGuid`). For an example of trigger metadata, see this [event hubs example](#).

context.done method

```
context.done([err],[propertyBag])
```

Lets the runtime know that your code has completed. When your function uses the `async function` declaration, you do not need to use `context.done()`. The `context.done` callback is implicitly called. Async functions are available in Node 8 or a later version, which requires version 2.x of the Functions runtime.

If your function is not an async function, you must call `context.done` to inform the runtime that your function is complete. The execution times out if it is missing.

The `context.done` method allows you to pass back both a user-defined error to the runtime and a JSON object containing output binding data. Properties passed to `context.done` overwrite anything set on the `context.bindings` object.

```
// Even though we set myOutput to have:  
// -> text: 'hello world', number: 123  
context.bindings.myOutput = { text: 'hello world', number: 123 };  
// If we pass an object to the done function...  
context.done(null, { myOutput: { text: 'hello there, world', noNumber: true }});  
// the done method overwrites the myOutput binding to be:  
// -> text: 'hello there, world', noNumber: true
```

context.log method

```
context.log(message)
```

Allows you to write to the streaming function logs at the default trace level, with other logging levels available. Trace logging is described in detail in the next section.

Write trace output to logs

In Functions, you use the `context.log` methods to write trace output to the logs and the console. When you call `context.log()`, your message is written to the logs at the default trace level, which is the *info* trace level. 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 telemetry and your trace outputs. To learn more, see [monitoring Azure Functions](#).

The following example writes a log at the info trace level, including the invocation ID:

```
context.log("Something has happened. " + context.invocationId);
```

All `context.log` methods support the same parameter format that's supported by the Node.js [util.format](#) method. Consider the following code, which writes function logs by using the default trace level:

```
context.log('Node.js HTTP trigger function processed a request. RequestUri=' + req.originalUrl);  
context.log('Request Headers = ' + JSON.stringify(req.headers));
```

You can also write the same code in the following format:

```
context.log('Node.js HTTP trigger function processed a request. RequestUri=%s', req.originalUrl);  
context.log('Request Headers = ', JSON.stringify(req.headers));
```

NOTE

Don't use `console.log` to write trace outputs. Because output from `console.log` is captured at the function app level, it's not tied to a specific function invocation and isn't displayed in a specific function's logs. Also, version 1.x of the Functions runtime doesn't support using `console.log` to write to the console.

Trace levels

In addition to the default level, the following logging methods are available that let you write function logs at specific trace levels.

| METHOD | DESCRIPTION |
|-------------------------------|---|
| <code>error(message)</code> | Writes an error-level event to the logs. |
| <code>warn(message)</code> | Writes a warning-level event to the logs. |
| <code>info(message)</code> | Writes to info level logging, or lower. |
| <code>verbose(message)</code> | Writes to verbose level logging. |

The following example writes the same log at the warning trace level, instead of the info level:

```
context.log.warn("Something has happened. " + context.invocationId);
```

Because `error` is the highest trace level, this trace is written to the output at all trace levels as long as logging is enabled.

Configure the trace level for logging

Functions lets you define the threshold trace level for writing to the logs or the console. The specific threshold settings depend on your version of the Functions runtime.

- [v2.x+](#)
- [v1.x](#)

To set the threshold for traces written to the logs, use the `logging.level` property in the host.json file. This JSON object lets you define a default threshold for all functions in your function app, plus you can define specific thresholds for individual functions. To learn more, see [How to configure monitoring for Azure Functions](#).

Log custom telemetry

By default, Functions writes output as traces to Application Insights. For more control, you can instead use the [Application Insights Node.js SDK](#) to send custom telemetry data to your Application Insights instance.

- [v2.x+](#)
- [v1.x](#)

```

const appInsights = require("applicationinsights");
appInsights.setup();
const client = appInsights.defaultClient;

module.exports = function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    // Use this with 'tagOverrides' to correlate custom telemetry to the parent function invocation.
    var operationIdOverride = {"ai.operation.id":context.traceContext.traceparent};

    client.trackEvent({name: "my custom event", tagOverrides:operationIdOverride, properties: {customProperty2: "custom property value"}});
    client.trackException({exception: new Error("handled exceptions can be logged with this method"), tagOverrides:operationIdOverride});
    client.trackMetric({name: "custom metric", value: 3, tagOverrides:operationIdOverride});
    client.trackTrace({message: "trace message", tagOverrides:operationIdOverride});
    client.trackDependency({target:"http://dbname", name:"select customers proc", data:"SELECT * FROM Customers", duration:231, resultCode:0, success: true, dependencyTypeName: "ZSQL", tagOverrides:operationIdOverride});
    client.trackRequest({name:"GET /customers", url:"http://myserver/customers", duration:309, resultCode:200, success:true, tagOverrides:operationIdOverride});

    context.done();
};

```

The `tagOverrides` parameter sets the `operation_Id` to the function's invocation ID. This setting enables you to correlate all of the automatically generated and custom telemetry for a given function invocation.

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 `context.req` (request) object has the following properties:

| PROPERTY | DESCRIPTION |
|--------------------------|--|
| <code>body</code> | An object that contains the body of the request. |
| <code>headers</code> | An object that contains the request headers. |
| <code>method</code> | The HTTP method of the request. |
| <code>originalUrl</code> | The URL of the request. |
| <code>params</code> | An object that contains the routing parameters of the request. |
| <code>query</code> | An object that contains the query parameters. |
| <code>rawBody</code> | The body of the message as a string. |

Response object

The `context.res` (response) object has the following properties:

| PROPERTY | DESCRIPTION |
|----------------|---|
| <i>body</i> | An object that contains the body of the response. |
| <i>headers</i> | An object that contains the response headers. |
| <i>isRaw</i> | Indicates that formatting is skipped for the response. |
| <i>status</i> | The HTTP status code of the response. |
| <i>cookies</i> | An array of HTTP cookie objects that are set in the response. An HTTP cookie object has a <code>name</code> , <code>value</code> , and other cookie properties, such as <code>maxAge</code> or <code>sameSite</code> . |

Accessing the request and response

When you work with HTTP triggers, you can access the HTTP request and response objects in a number of ways:

- From `req` and `res` properties on the `context` object. In this way, you can use the conventional pattern to access HTTP data from the context object, instead of having to use the full `context.bindings.name` pattern. The following example shows how to access the `req` and `res` objects on the `context`:

```
// You can access your HTTP request off the context ...
if(context.req.body.emoji === ':pizza:') context.log('Yay!');
// and also set your HTTP response
context.res = { status: 202, body: 'You successfully ordered more coffee!' };
```

- From the named input and output bindings. In this way, the HTTP trigger and bindings work the same as any other binding. The following example sets the response object by using a named `response` binding:

```
{
  "type": "http",
  "direction": "out",
  "name": "response"
}
```

```
context.bindings.response = { status: 201, body: "Insert succeeded." };
```

- [Response only] By calling `context.res.send(body?: any)`. An HTTP response is created with input `body` as the response body. `context.done()` is implicitly called.
- [Response only] By calling `context.done()`. A special type of HTTP binding returns the response that is passed to the `context.done()` method. The following HTTP output binding defines a `$return` output parameter:

```
{
  "type": "http",
  "direction": "out",
  "name": "$return"
}
```

```
// Define a valid response object.
res = { status: 201, body: "Insert succeeded." };
context.done(null, res);
```

Scaling and concurrency

By default, Azure Functions automatically monitors the load on your application and creates additional host instances for Node.js as needed. 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.

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.

The [FUNCTIONS_WORKER_PROCESS_COUNT](#) applies to each host that Functions creates when scaling out your application to meet demand.

Node version

The following table shows current supported Node.js versions for each major version of the Functions runtime, by operating system:

| FUNCTIONS VERSION | NODE VERSION (WINDOWS) | NODE VERSION (LINUX) |
|-------------------|---|---|
| 1.x | 6.11.2 (locked by the runtime) | n/a |
| 2.x | <div style="display: flex; justify-content: space-around;"> ~8 ~10 (recommended) ~12 </div> | <div style="display: flex; justify-content: space-around;"> node 8 node 10 (recommended) </div> |
| 3.x | <div style="display: flex; justify-content: space-around;"> ~10 ~12 (recommended) ~14 (preview) </div> | <div style="display: flex; justify-content: space-around;"> node 10 node 12 (recommended) node 14 (preview) </div> |

You can see the current version that the runtime is using by logging `process.version` from any function.

Setting the Node version

For Windows function apps, target the version in Azure by setting the [WEBSITE_NODE_DEFAULT_VERSION](#) app setting to a supported LTS version, such as `~12`.

For Linux function apps, run the following Azure CLI command to update the Node version.

```
az functionapp config set --linux-fx-version "node|12" --name "<MY_APP_NAME>" --resource-group "<MY_RESOURCE_GROUP_NAME>"
```

Dependency management

In order to use community libraries in your JavaScript code, as is shown in the below example, you need to ensure that all dependencies are installed on your Function App in Azure.

```
// Import the underscore.js library
var _ = require('underscore');
var version = process.version; // version === 'v6.5.0'

module.exports = function(context) {
    // Using our imported underscore.js library
    var matched_names = _
        .where(context.bindings.myInput.names, {first: 'Carla'});
}
```

NOTE

You should define a `package.json` file at the root of your Function App. Defining the file lets all functions in the app share the same cached packages, which gives the best performance. If a version conflict arises, you can resolve it by adding a `package.json` file in the folder of a specific function.

When deploying Function Apps from source control, any `package.json` file present in your repo, will trigger an `npm install` in its folder during deployment. But when deploying via the Portal or CLI, you will have to manually install the packages.

There are two ways to install packages on your Function App:

Deploying with Dependencies

1. Install all requisite packages locally by running `npm install`.
2. Deploy your code, and ensure that the `node_modules` folder is included in the deployment.

Using Kudu

1. Go to https://<function_app_name>.scm.azurewebsites.net.
2. Click **Debug Console > CMD**.
3. Go to `D:\home\site\wwwroot`, and then drag your `package.json` file to the `wwwroot` folder at the top half of the page.
You can upload files to your function app in other ways also. For more information, see [How to update function app files](#).
4. After the `package.json` file is uploaded, run the `npm install` command in the **Kudu remote execution console**.
This action downloads the packages indicated in the `package.json` file and restarts the function app.

Environment variables

Add your own environment variables to a function app, in both your local and cloud environments, such as operational secrets (connection strings, keys, and endpoints) or environmental settings (such as profiling variables). Access these settings using `process.env` in your function code.

In local development environment

When running locally, your functions project includes a `local.settings.json` file, where you store your environment variables in the `Values` object.

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "",  
        "FUNCTIONS_WORKER_RUNTIME": "node",  
        "translatorTextEndPoint": "https://api.cognitive.microsofttranslator.com/",  
        "translatorTextKey": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
        "languageWorkers_node_arguments": "--prof"  
    }  
}
```

In Azure cloud environment

When running in Azure, the function app lets you set uses [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](#).

Access environment variables in code

Access application settings as environment variables using `process.env`, as shown here in the second and third calls to `context.log()` where we log the `AzureWebJobsStorage` and `WEBSITE_SITE_NAME` environment variables:

```
module.exports = async function (context, myTimer) {  
  
    context.log("AzureWebJobsStorage: " + process.env["AzureWebJobsStorage"]);  
    context.log("WEBSITE_SITE_NAME: " + process.env["WEBSITE_SITE_NAME"]);  
};
```

Configure function entry point

The `function.json` properties `scriptFile` and `entryPoint` can be used to configure the location and name of your exported function. These properties can be important when your JavaScript is transpiled.

Using `scriptFile`

By default, a JavaScript function is executed from `index.js`, a file that shares the same parent directory as its corresponding `function.json`.

`scriptFile` can be used to get a folder structure that looks like the following example:

```
FunctionApp  
| - host.json  
| - myNodeFunction  
| | - function.json  
| - lib  
| | - sayHello.js  
| - node_modules  
| | - ... packages ...  
| - package.json
```

The `function.json` for `myNodeFunction` should include a `scriptFile` property pointing to the file with the exported function to run.

```
{  
  "scriptFile": "../lib/sayHello.js",  
  "bindings": [  
    ...  
  ]  
}
```

Using `entryPoint`

In `scriptFile` (or `index.js`), a function must be exported using `module.exports` in order to be found and run. By default, the function that executes when triggered is the only export from that file, the export named `run`, or the export named `index`.

This can be configured using `entryPoint` in `function.json`, as in the following example:

```
{  
  "entryPoint": "logFoo",  
  "bindings": [  
    ...  
  ]  
}
```

In Functions v2.x, which supports the `this` parameter in user functions, the function code could then be as in the following example:

```
class MyObj {  
  constructor() {  
    this.foo = 1;  
  };  
  
  logFoo(context) {  
    context.log("Foo is " + this.foo);  
    context.done();  
  }  
}  
  
const myObj = new MyObj();  
module.exports = myObj;
```

In this example, it is important to note that although an object is being exported, there are no guarantees for preserving state between executions.

Local Debugging

When started with the `--inspect` parameter, a Node.js process listens for a debugging client on the specified port. In Azure Functions 2.x, you can specify arguments to pass into the Node.js process that runs your code by adding the environment variable or App Setting `languageWorkers:node:arguments = <args>`.

To debug locally, add `"languageWorkers:node:arguments": "--inspect=5858"` under `Values` in your `local.settings.json` file and attach a debugger to port 5858.

When debugging using VS Code, the `--inspect` parameter is automatically added using the `port` value in the project's `launch.json` file.

In version 1.x, setting `languageWorkers:node:arguments` will not work. The debug port can be selected with the `--nodeDebugPort` parameter on Azure Functions Core Tools.

TypeScript

When you target version 2.x of the Functions runtime, both [Azure Functions for Visual Studio Code](#) and the [Azure Functions Core Tools](#) let you create function apps using a template that support TypeScript function app projects. The template generates `package.json` and `tsconfig.json` project files that make it easier to transpile, run, and publish JavaScript functions from TypeScript code with these tools.

A generated `.funcignore` file is used to indicate which files are excluded when a project is published to Azure.

TypeScript files (.ts) are transpiled into JavaScript files (js) in the `dist` output directory. TypeScript templates use the `scriptFile` parameter in `function.json` to indicate the location of the corresponding js file in the `dist` folder. The output location is set by the template by using `outDir` parameter in the `tsconfig.json` file. If you change this setting or the name of the folder, the runtime is not able to find the code to run.

The way that you locally develop and deploy from a TypeScript project depends on your development tool.

Visual Studio Code

The [Azure Functions for Visual Studio Code](#) extension lets you develop your functions using TypeScript. The Core Tools is a requirement of the Azure Functions extension.

To create a TypeScript function app in Visual Studio Code, choose `TypeScript` as your language when you create a function app.

When you press F5 to run the app locally, transpilation is done before the host (func.exe) is initialized.

When you deploy your function app to Azure using the **Deploy to function app...** button, the Azure Functions extension first generates a production-ready build of JavaScript files from the TypeScript source files.

Azure Functions Core Tools

There are several ways in which a TypeScript project differs from a JavaScript project when using the Core Tools.

Create project

To create a TypeScript function app project using Core Tools, you must specify the TypeScript language option when you create your function app. You can do this in one of the following ways:

- Run the `func init` command, select `node` as your language stack, and then select `typescript`.
- Run the `func init --worker-runtime typescript` command.

Run local

To run your function app code locally using Core Tools, use the following commands instead of `func host start`:

```
npm install  
npm start
```

The `npm start` command is equivalent to the following commands:

- `npm run build`
- `func extensions install`
- `tsc`
- `func start`

Publish to Azure

Before you use the `func azure functionapp publish` command to deploy to Azure, you create a production-ready build of JavaScript files from the TypeScript source files.

The following commands prepare and publish your TypeScript project using Core Tools:

```
npm run build:production
func azure functionapp publish <APP_NAME>
```

In this command, replace `<APP_NAME>` with the name of your function app.

Considerations for JavaScript functions

When you work with JavaScript functions, be aware of the considerations in the following sections.

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 JavaScript functions more efficiently on single-vCPU VMs, and using larger VMs does not 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](#).

Cold Start

When developing Azure Functions in the serverless hosting model, cold starts are a reality. *Cold start* refers to the fact that when your function app starts for the first time after a period of inactivity, it takes longer to start up. For JavaScript functions 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 the run from package model by default, but if you're experiencing large cold starts and are not running this way, this change can offer a significant improvement.

Connection Limits

When you use a service-specific client in an Azure Functions application, don't create a new client with every function invocation. 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 JavaScript, 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 are not properly awaited.

In the example below, the asynchronous method `fs.readFile` is invoked with an error-first callback function as its second parameter. This code causes both of the issues mentioned above. An exception that is not explicitly caught in the correct scope crashed the entire process (issue #1). Calling `context.done()` outside of the scope of the callback function means that the function invocation may end before the file is read (issue #2). In this example, calling `context.done()` too early results in missing log entries starting with `Data from file:`.

```
// NOT RECOMMENDED PATTERN
const fs = require('fs');

module.exports = function (context) {
    fs.readFile('./hello.txt', (err, data) => {
        if (err) {
            context.log.error('ERROR', err);
            // BUG #1: This will result in an uncaught exception that crashes the entire process
            throw err;
        }
        context.log(`Data from file: ${data}`);
        // context.done() should be called here
    });
    // BUG #2: Data is not guaranteed to be read before the Azure Function's invocation ends
    context.done();
}
```

Using the `async` and `await` keywords helps avoid both of these errors. You should use the Node.js utility function `util.promisify` to turn error-first callback-style functions into awaitable functions.

In the example below, any unhandled exceptions thrown during the function execution only fail the individual invocation that raised an exception. The `await` keyword means that steps following `readFileAsync` only execute after `readFile` is complete. With `async` and `await`, you also don't need to call the `context.done()` callback.

```
// Recommended pattern
const fs = require('fs');
const util = require('util');
const readFileAsync = util.promisify(fs.readFile);

module.exports = async function (context) {
    let data;
    try {
        data = await readFileAsync('./hello.txt');
    } catch (err) {
        context.log.error('ERROR', err);
        // This rethrown exception will be handled by the Functions Runtime and will only fail the individual
        // invocation
        throw err;
    }
    context.log(`Data from file: ${data}`);
}
```

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 Java developer guide

12/4/2020 • 13 minutes to read • [Edit Online](#)

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, please consider first reading one of the following articles:

| GETTING STARTED | CONCEPTS |
|--|--|
| <ul style="list-style-type: none">• Java function using Visual Studio Code• Java/Maven function with terminal/command prompt• Java function using Gradle• Java function using Eclipse• Java function using IntelliJ IDEA | <ul style="list-style-type: none">• Developer guide• Hosting options• Performance considerations |

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. When running 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](#)

The article links above 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](#)
- [Cmd](#)

```
mvn archetype:generate \
-DarchetypeGroupId=com.microsoft.azure \
-DarchetypeArtifactId=azure-functions-archetype
```

To get started using this archetype, see the [Java quickstart](#).

Folder structure

Here is 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:

```

public class Function {
    public String echo(@HttpTrigger(name = "req",
        methods = {HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
        String req, ExecutionContext context) {
        return String.format(req);
    }
}

```

Here is the generated corresponding `function.json` by the [azure-functions-maven-plugin](#):

```
{
  "scriptFile": "azure-functions-example.jar",
  "entryPoint": "com.example.Function.echo",
  "bindings": [
    {
      "type": "httpTrigger",
      "name": "req",
      "direction": "in",
      "authLevel": "anonymous",
      "methods": [ "post" ]
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ]
}
```

Java versions

The version of Java used when creating the function app on which functions 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 have 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:

| FUNCTIONS VERSION | JAVA VERSIONS (WINDOWS) | JAVA VERSIONS (LINUX) |
|-------------------|-------------------------|-----------------------|
| 3.x | 11 8 | 11 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` or `11`.

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:

| ELEMENT | JAVA 8 VALUE | JAVA 11 VALUE | DESCRIPTION |
|--------------|--------------|---------------|--|
| Java.version | 1.8 | 11 | Version of Java used by the maven-compiler-plugin. |
| JavaVersion | 8 | 11 | 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

```
<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.version>
  <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

```
<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.

| ELEMENT | WINDOWS | LINUX | DOCKER |
|---------|---------|-------|--------|
| os | windows | linux | docker |

The following example shows the operating system setting in the `runtime` section of the pom.xml file:

```

<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

For local development of Java function apps, download and use the appropriate [Azul Zulu Enterprise for Azure](#) Java JDKs from [Azul Systems](#). Azure Functions uses an Azul Java JDK runtime when you deploy your function app to the cloud.

[Azure support](#) for issues with the JDKs and function apps is available with a [qualified support plan](#).

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 additional arguments in an app setting named `JAVA_OPTS`. You can add app settings to your function app deployed to Azure in the Azure portal or the Azure CLI.

IMPORTANT

In the Consumption plan, you must also add the `WEBSITE_USE_PLACEHOLDER` setting with a value of 0 for the customization to work. This setting does increase the cold start times for Java functions.

Azure portal

In the [Azure portal](#), use the [Application Settings tab](#) to add the `JAVA_OPTS` setting.

Azure CLI

You can use the `az functionapp config appsettings set` command to set `JAVA_OPTS`, as in the following example:

- [Consumption plan](#)
- [Dedicated plan / Premium plan](#)

```

az functionapp config appsettings set \
    --settings "JAVA_OPTS=-Djava.awt.headless=true" \
    "WEBSITE_USE_PLACEHOLDER=0" \
    --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`:

```
@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

```

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>`.

```

@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 is 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`.

```
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.

```
@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`. It writes multiple values to Queue storage.

HttpRequestMessage and HttpResponseMessage

These are defined in `azure-functions-java-library`. They are helper types to work with `HttpTrigger` functions.

| SPECIALIZED TYPE | TARGET | TYPICAL USAGE |
|--|--------------|----------------------------------|
| <code>HttpRequestMessage<T></code> | HTTP Trigger | Gets method, headers, or queries |

| SPECIALIZED TYPE | TARGET | TYPICAL USAGE |
|---------------------|---------------------|-------------------------------|
| HttpResponseMessage | 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.

```
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.

```
@FunctionName("QueueTriggerMetadata")
public void QueueTriggerMetadata(
    @QueueTrigger(name = "message", queueName = "test-input-java-metadata", 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:

```
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, as well as other application logging.

Here's how to configure your function app to write application logging by using the Azure CLI:

- [Bash](#)
- [Cmd](#)

```
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](#)
- [Cmd](#)

```
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:

```
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`:

```
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);  
    }  
}
```

NOTE

The value of AppSetting FUNCTIONS_EXTENSION_VERSION should be ~2 or ~3 for an optimized cold start experience.

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

12/4/2020 • 20 minutes to read • [Edit Online](#)

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:

```
# $TriggerMetadata is optional here. If you don't need it, you can safely remove it from the param block
param($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:

```
$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`.

```
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:

```
param($MyFirstInputBinding, $MySecondInputBinding)

Push-OutputBinding -Name myQueue -Value $myValue
```

You can also pass in a value for a specific binding through the pipeline.

```
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>-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":

```
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.

```
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":

```
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":

```
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".

```
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:

```
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:

```
Get-OutputBinding
```

| Name | Value |
|--------------|--------|
| ---- | ----- |
| MyQueue | myData |
| MyOtherQueue | myData |

`Get-OutputBinding` also contains a parameter called `-Name`, which can be used to filter the returned binding, as in the following example:

```
Get-OutputBinding -Name MyQ*
```

| 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> | Information | Writes to <i>Information</i> level logging. |
| 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.level.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`:

```
{  
  "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 |

| PROPERTY | DESCRIPTION | TYPE |
|----------|--|----------------------------|
| Headers | A dictionary that contains the request headers. | Dictionary<string,string>* |
| Method | The HTTP method of the request. | string |
| Params | An object that contains the routing parameters of the request. | Dictionary<string,string>* |
| Query | An object that contains the query parameters. | Dictionary<string,string>* |
| Url | 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 `HttpResponseContext`, which has the following properties:

| PROPERTY | DESCRIPTION | TYPE |
|-------------|---|-------------------------|
| Body | An object that contains the body of the response. | object |
| ContentType | A short hand for setting the content type for the response. | string |
| Headers | An object that contains the response headers. | Dictionary or Hashtable |
| StatusCode | 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 `HttpResponseContext` object to return a response, as shown in the following:

`function.json`

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "anonymous"
    },
    {
      "type": "http",
      "direction": "out"
    }
  ]
}
```

`run.ps1`

```

param($req, $TriggerMetadata)

$name = $req.Query.Name

Push-OutputBinding -Name res -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:

```
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 |
|-------------------|---|--------------------------------|
| 3.x (recommended) | PowerShell 7 (recommended) PowerShell Core 6 | .NET Core 3.1 .NET Core 2.1 |
| 2.x | PowerShell Core 6 | .NET Core 2.2 |

You can see the current version by printing `$PSVersionTable` from any function.

Running local on a specific version

When running locally the Azure Functions runtime defaults to using PowerShell Core 6. To instead use PowerShell 7 when running locally, you need to add the setting `"FUNCTIONS_WORKER_RUNTIME_VERSION" : "~7"` to

the `Values` array in the local.setting.json file in the project root. When running locally on PowerShell 7, your local.settings.json file looks like the following example:

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "",  
        "FUNCTIONS_WORKER_RUNTIME": "powershell",  
        "FUNCTIONS_WORKER_RUNTIME_VERSION" : "~7"  
    }  
}
```

Changing the PowerShell version

Your function app must be running on version 3.x to be able to upgrade from PowerShell Core 6 to PowerShell 7. 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](#)
- [PowerShell](#)

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 portal's Configuration page for a function app. The 'General settings' tab is active. The 'PowerShell Core Version' dropdown is open, showing three options: 'PowerShell 7.0' (selected), 'PowerShell 7.0', and 'PowerShell Core 6.2'. Other tabs visible include 'Application settings' and 'Function runtime settings'. A red box highlights the 'Save' button at the top right, and another red box highlights the 'General settings' tab.

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:

```
{
  "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:

```
@{
  Az = '1.*'
 SqlServer = '21.1.18147'
}
```

When you update the requirements.psd1 file, updated modules are installed after a restart.

NOTE

Managed dependencies requires access to www.powershellgallery.com to download modules. When running locally, make sure that the runtime can access this URL by adding any required firewall rules.

NOTE

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`.

The following application settings can be used to change how the managed dependencies are downloaded and installed. Your app upgrade starts within `MDMaxBackgroundUpgradePeriod`, and the upgrade process completes within approximately the `MDNewSnapshotCheckPeriod`.

| FUNCTION APP SETTING | DEFAULT VALUE | DESCRIPTION |
|---|----------------------------------|---|
| <code>MDMaxBackgroundUpgradePeriod</code> | <code>7.00:00:00</code> (7 days) | Each PowerShell worker process initiates checking for module upgrades on the PowerShell Gallery on process start and every <code>MDMaxBackgroundUpgradePeriod</code> 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 may also delay delivering new module versions to your app. |

| FUNCTION APP SETTING | DEFAULT VALUE | DESCRIPTION |
|---|---------------------------------|---|
| <code>MDNewSnapshotCheckPeriod</code> | <code>01:00:00</code> (1 hour) | 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 may use either the old or the new module versions. Restarting all PowerShell workers complete within <code>MDNewSnapshotCheckPeriod</code> . Increasing this value decreases the frequency of interruptions, but may also increase the period of time when function invocations use either the old or the new module versions non-deterministically. |
| <code>MDMinBackgroundUpgradePeriod</code> | <code>1.00:00:00</code> (1 day) | 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 <code>MDMinBackgroundUpgradePeriod</code> . |

Leveraging your own custom modules is a little different than how you would do it normally.

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.

```
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.ps1
```

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:

```
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](#).

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 amount of runspaces created will match the `PSWorkerInProcConcurrencyUpperBound` application setting. 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.

```
{  
  "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:

```
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.

```
{  
  "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

12/4/2020 • 19 minutes to read • [Edit Online](#)

This article is an introduction to developing Azure Functions using Python. The content below assumes that you've already read the [Azure Functions developers guide](#).

As a Python developer, you may also be interested in one of the following articles:

| GETTING STARTED | CONCEPTS | SCENARIOS/SAMPLES |
|---|--|--|
| <ul style="list-style-type: none">• Python function using Visual Studio Code• Python function with terminal/command prompt | <ul style="list-style-type: none">• Developer guide• Hosting options• Performance considerations | <ul style="list-style-type: none">• Image classification with PyTorch• Azure automation sample• Machine learning with TensorFlow• Browse Python samples |

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 called `main()` in the `__init__.py` file. You can also [specify an alternate entry point](#).

Data from triggers and bindings is bound to the function via method attributes using the `name` property defined in the `function.json` file. For example, the `function.json` below describes a simple function triggered by an HTTP request named `req`:

```
{  
    "scriptFile": "__init__.py",  
    "bindings": [  
        {  
            "authLevel": "function",  
            "type": "httpTrigger",  
            "direction": "in",  
            "name": "req",  
            "methods": [  
                "get",  
                "post"  
            ]  
        },  
        {  
            "type": "http",  
            "direction": "out",  
            "name": "$return"  
        }  
    ]  
}
```

Based on this definition, the `__init__.py` file that contains the function code might look like the following example:

```
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 using Python type annotations. This helps you use the intellisense and autocomplete features provided by many Python code editors.

```
import azure.functions

def main(req: azure.functions.HttpRequest) -> str:
    user = req.params.get('user')
    return f'Hello, {user}!'
```

Use the Python annotations included in the [azure.functions.*](#) package to bind input and outputs to your methods.

Alternate entry point

You can change the default behavior of a function by optionally specifying the `scriptFile` and `entryPoint` properties in the `function.json` file. For example, the `function.json` below tells the runtime to use the `customentry()` method in the `main.py` file, as the entry point for your Azure Function.

```
{
  "scriptFile": "main.py",
  "entryPoint": "customentry",
  "bindings": [
    ...
  ]
}
```

Folder structure

The recommended folder structure for a Python Functions project looks like the following example:

```
<project_root>/
| - .venv/
| - .vscode/
| - my_first_function/
| | - __init__.py
| | - function.json
| | - example.py
| - my_second_function/
| | - __init__.py
| | - function.json
| - shared_code/
| | - __init__.py
| | - my_first_helper_function.py
| | - my_second_helper_function.py
| - tests/
| | - test_my_second_function.py
| - .funcignore
| - host.json
| - local.settings.json
| - requirements.txt
| - Dockerfile
```

The main project folder (<project_root>) can contain the following files:

- *local.settings.json*: Used to store app settings and connection strings when running locally. This file doesn't get published to Azure. To learn more, see [local.settings.file](#).
- *requirements.txt*: Contains the list of Python packages the system installs when publishing to Azure.
- *host.json*: Contains global configuration options that affect all functions in a function app. This file does get published to Azure. Not all options are supported when running locally. To learn more, see [host.json](#).
- *.vscode/*: (Optional) Contains store VSCode configuration. To learn more, see [VSCode setting](#).
- *.venv/*: (Optional) Contains a Python virtual environment used by local development.
- *Dockerfile*: (Optional) Used when publishing your project in a [custom container](#).
- *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.

Each function has its own code file and binding configuration file (*function.json*).

When deploying your project to a function app in Azure, the entire contents of the main project (<project_root>) folder should be included in the package, but not the folder itself, which means `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](#).

Import behavior

You can import modules in your function code using both absolute and relative references. Based on the folder structure shown above, the following imports work from within the function file `<project_root>\my_first_function__init__.py`:

```
from shared_code import my_first_helper_function #(absolute)
```

```
import shared_code.my_second_helper_function #(absolute)
```

```
from . import example #(relative)
```

NOTE

The `shared_code/` folder needs to contain an `__init__.py` file to mark it as a Python package when using absolute import syntax.

The following `__app__` import and beyond top-level relative import are deprecated, since it is not supported by static type checker and not supported by Python test frameworks:

```
from __app__.shared_code import my_first_helper_function #(deprecated __app__ import)
```

```
from ..shared_code import my_first_helper_function #(deprecated beyond top-level relative import)
```

Triggers and Inputs

Inputs are divided into two categories in Azure Functions: trigger input and additional input. Although they are different in the `function.json` file, usage is identical in Python code. Connection strings or secrets for trigger and input sources map to values in the `local.settings.json` file when running locally, and the application settings when running in Azure.

For example, the following code demonstrates the difference between the two:

```
// function.json
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "req",
      "direction": "in",
      "type": "httpTrigger",
      "authLevel": "anonymous",
      "route": "items/{id}"
    },
    {
      "name": "obj",
      "direction": "in",
      "type": "blob",
      "path": "samples/{id}",
      "connection": "AzureWebJobsStorage"
    }
  ]
}
```

```
// local.settings.json
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "AzureWebJobsStorage": "<azure-storage-connection-string>"
  }
}
```

```
# __init__.py
import azure.functions as func
import logging

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 will be retrieved from the Azure Blob Storage based on the `/ID` in the route URL and made available as `obj` in the function body. Here, the storage account specified is the connection string found in the `AzureWebJobsStorage` app setting, which is the same storage account used by the function app.

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 use the return value of a function as the value of an output binding, the `name` property of the binding should be set to `$return` in `function.json`.

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.

```
{  
    "scriptFile": "__init__.py",  
    "bindings": [  
        {  
            "name": "req",  
            "direction": "in",  
            "type": "httpTrigger",  
            "authLevel": "anonymous"  
        },  
        {  
            "name": "msg",  
            "direction": "out",  
            "type": "queue",  
            "queueName": "outqueue",  
            "connection": "AzureWebJobsStorage"  
        },  
        {  
            "name": "$return",  
            "direction": "out",  
            "type": "http"  
        }  
    ]  
}
```

```
import azure.functions as func  
  
def main(req: func.HttpRequest,  
        msg: func.Out[func.QueueMessage]) -> 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 encountered during the function execution.

The following example logs an info message when the function is invoked via an HTTP trigger.

```
import logging  
  
def main(req):  
    logging.info('Python HTTP trigger function processed a request.')
```

Additional logging methods are available that let you write to the console at different trace levels:

| METHOD | DESCRIPTION |
|----------------------------------|--|
| <code>critical(_message_)</code> | Writes a message with level CRITICAL on the root logger. |

| METHOD | DESCRIPTION |
|---------------------------------|---|
| <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](#).

HTTP Trigger and bindings

The HTTP trigger is defined in the function.json file. The `name` of the binding must match the named parameter in the function. In the previous examples, a binding name `req` is used. This parameter is an [HttpRequest](#) object, and an [HttpResponse](#) object is returned.

From the [HttpRequest](#) object, you can get request headers, query parameters, route parameters, and the message body.

The following example is from the [HTTP trigger template for Python](#).

```
def main(req: func.HttpRequest) -> func.HttpResponse:
    headers = {"my-http-header": "some-value"}

    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}!", headers=headers)
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            headers=headers, status_code=400
        )
```

In this function, the value of the `name` query parameter is obtained from the `params` parameter of the [HttpRequest](#) object. The JSON-encoded message body is read using the `get_json` method.

Likewise, you can set the `status_code` and `headers` for the response message in the returned [HttpResponse](#) object.

Scaling and Performance

It's important to understand how your functions perform and how that performance affects the way your function app gets scaled. This is particularly important when designing highly performant apps. The following are several factors to consider when designing, writing and configuring your functions apps.

Horizontal scaling

By default, Azure Functions automatically monitors the load on your application and creates additional host

instances for Python as needed. 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 [How the Consumption and Premium plans work](#).

Improving throughput performance

A key to improving performance is understanding how your app uses resources and being able to configure your function app accordingly.

Understanding your workload

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 are running.

| | I/O-BOUND WORKLOAD | CPU-BOUND WORKLOAD |
|-------------------------------------|--|---|
| Function app characteristics | <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">App does long-running computations, such as image resizing.App does data transformation. |
| Examples | <ul style="list-style-type: none">Web APIs | <ul style="list-style-type: none">Data processingMachine learning inference |

NOTE

As real world functions workload are most often a mix of I/O and CPU bound, we recommend to profile the workload under realistic production loads.

Performance-specific configurations

After understanding the workload profile of your function app, the following are configurations that you can use to improve the throughput performance of your functions.

Async

Because [Python is a single-threaded runtime](#), a host instance for Python can process only one function invocation at a time. 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 `aiohttp` directly:

```
async def main():
    await some_nonblocking_socket_io_op()
```

Here is an example of a function with HTTP trigger that uses `aiohttp` http client:

```
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 an asyncio thread-pool:

```
# Runs in an asyncio thread-pool

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.

Here are a few examples of client libraries that has implemented `async` pattern:

- [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

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 worker 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 impact overall performance due to the increased number of required context switches.

The `FUNCTIONS_WORKER_PROCESS_COUNT` applies to each host that Functions creates when scaling out your application to meet demand.

Context

To get the invocation context of a function during execution, include the `context` argument in its signature.

For example:

```
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:

`function_directory` The directory in which the function is running.

`function_name` Name of the function.

`invocation_id` ID of the current function invocation.

Global variables

It is not 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. In order to cache the results of an expensive computation, declare it as a global variable.

```
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 Functions, [application settings](#), such as service connection strings, are exposed as environment variables during execution. You can access these settings by declaring `import os` and then using,

`setting = os.environ["setting-name"]`.

The following example gets the [application setting](#), with the key named `myAppSetting`:

```
import logging
import os
import azure.functions as func

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:

| FUNCTIONS VERSION | PYTHON* VERSIONS |
|-------------------|-------------------|
| 3.x | 3.8 3.7 3.6 |
| 2.x | 3.7 3.6 |

*Official CPython 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 can't be changed.

When running locally, the runtime uses the available Python version.

Package management

When developing locally using the Azure Functions Core Tools or Visual Studio Code, add the names and versions of the required packages to the `requirements.txt` file and install them using `pip`.

For example, the following requirements file and pip command can be used to install the `requests` package from PyPI.

```
requests==2.19.1
```

```
pip install -r requirements.txt
```

Publishing to Azure

When you're ready to publish, make sure that all your publicly available dependencies are listed in the `requirements.txt` file, which is located at the root of your project directory.

Project files and folders that are excluded from publishing, including the virtual environment folder, are listed in the `.funcignore` file.

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 using continuous delivery (CD). To learn more, see [Continuous delivery by using Azure DevOps](#).

Remote build

When using remote build, dependencies restored on the server and native dependencies match the production environment. This results in a smaller deployment package to upload. Use remote build when 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, the Azure Functions Core Tools requests a remote build when you use the following `func azure functionapp publish` command to publish your Python project to Azure.

```
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.

```
func azure functionapp publish <APP_NAME> --build local
```

Remember to replace `<APP_NAME>` with the name of your function app in Azure.

Using 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, dependencies in your requirements.txt file can't be acquired by Core Tools, you must use the custom dependencies option for publishing.

We don't recommend using local builds when developing locally on Windows.

Custom dependencies

When your project has dependencies not found in the [Python Package Index](#), there are two ways to build the project. 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 publishing, make 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` 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 Python documentation.

Install local packages

If your project uses packages not publicly available to our tools, you can make them available to your app by putting them in the `__app__/python_packages` directory. Before publishing, run the following command to install the dependencies locally:

```
pip install --target="/python_packages/lib/site-packages" -r requirements.txt
```

When using custom dependencies, you should use the `--no-build` publishing option, since you have already installed the dependencies into the project folder.

```
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 written in Python can be tested like other Python code 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 is not immediately available, be sure to install it via your `requirements.txt` file as described in the [package management](#) section above.

Take `my_second_function` as an example, following is a mock test of an HTTP triggered function:

First we need to create `<project_root>/my_second_function/function.json` file and define this function as an http trigger.

```
{
  "scriptFile": "__init__.py",
  "entryPoint": "main",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    }
  ]
}
```

Now, we can implement the *my_second_function* and the *shared_code.my_second_helper_function*.

```
# <project_root>/my_second_function/__init__.py
import azure.functions as func
import logging

# Use absolute import to resolve shared_code modules
from shared_code import my_second_helper_function

# Define an http trigger which accepts ?value=<int> query parameter
# Double the value and return the result in HttpResponseMessage
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
    )
```

```
# <project_root>/shared_code/__init__.py
# Empty __init__.py file marks shared_code folder as a Python package
```

```
# <project_root>/shared_code/my_second_helper_function.py

def double(value: int) -> int:
    return value * 2
```

We can start writing test cases for our http trigger.

```

# <project_root>/tests/test_my_second_function.py
import unittest

import azure.functions as func
from my_second_function 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.
        resp = main(req)

        # Check the output.
        self.assertEqual(
            resp.get_body(),
            b'21 * 2 = 42',
        )

```

Inside your `.venv` Python virtual environment, install your favorite Python test framework (e.g. `pip install pytest`). Simply 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 generated and used by your functions during execution.

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`):

```

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 separate from the project folder. This keeps you from deploying test code with your app.

Preinstalled libraries

There are a few libraries come with the Python Functions runtime.

Python Standard Library

The Python Standard Library contain a list of built-in Python modules that are shipped with each Python

distribution. Most of these libraries help you access system functionality, like file I/O. On Windows systems, these libraries are installed with Python. On the Unix-based systems, they are provided by package collections.

To view the full details of the list of these libraries, please visit the links below:

- [Python 3.6 Standard Library](#)
- [Python 3.7 Standard Library](#)
- [Python 3.8 Standard Library](#)

Azure Functions Python worker dependencies

The 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 may not be available to your code when running outside of Azure Functions. You can find a detailed list of dependencies in the `install_requires` section in the [setup.py](#) file.

NOTE

If your function app's requirements.txt contains an `azure-functions-worker` entry, remove it. The functions worker is automatically managed by Azure Functions platform, and we regularly update it with new features and bug fixes. Manually installing an old version of worker in requirements.txt may cause unexpected issues.

Azure Functions Python library

Every Python worker update includes a new version of [Azure Functions Python library \(azure.functions\)](#). This approach makes it easier to continuously update your Python function apps, because each update is backwards-compatible. A list of releases of this library can be found in [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:

```
getattr(azure.functions, '__version__', '< 1.2.1')
```

Runtime system libraries

For a list of preinstalled system libraries in Python worker Docker images, please follow the links below:

| FUNCTIONS RUNTIME | DEBIAN VERSION | PYTHON VERSIONS |
|-------------------|----------------|--|
| Version 2.x | Stretch | Python 3.6 Python 3.7 |
| Version 3.x | Buster | Python 3.6 Python 3.7 Python 3.8 |

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](#).

CORS is fully supported for Python function apps.

Known issues and FAQ

Following is a list of troubleshooting guides for common issues:

- [ModuleNotFoundError and ImportError](#)
- [Cannot import 'cygrpc'](#)

All known issues and feature requests are tracked using [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 trigger](#)

[Having issues? Let us know.](#)

Azure Functions JavaScript developer guide

12/4/2020 • 24 minutes to read • [Edit Online](#)

This guide contains detailed information to help you succeed developing Azure Functions using JavaScript.

As an Express.js, Node.js, or JavaScript developer, if you are new to Azure Functions, please consider first reading one of the following articles:

| GETTING STARTED | CONCEPTS | GUIDED LEARNING |
|---|---|---|
| <ul style="list-style-type: none">• Node.js function using Visual Studio Code• Node.js function with terminal/command prompt | <ul style="list-style-type: none">• Developer guide• Hosting options• TypeScript functions• Performance considerations | <ul style="list-style-type: none">• Create serverless applications• Refactor Node.js and Express APIs to Serverless APIs |

JavaScript function basics

A JavaScript (Node.js) function is an exported `function` that executes when triggered ([triggers are configured in function.json](#)). The first argument passed to every function is a `context` object, which is used for receiving and sending binding data, logging, and communicating with the runtime.

Folder structure

The required folder structure for a JavaScript project looks like the following. This default can be changed. For more information, see the [scriptFile](#) section below.

```
FunctionsProject
| - MyFirstFunction
| | - index.js
| | - function.json
| - MySecondFunction
| | - index.js
| | - function.json
| - SharedCode
| | - myFirstHelperFunction.js
| | - mySecondHelperFunction.js
| - node_modules
| - host.json
| - package.json
| - extensions.csproj
```

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 (js) and binding configuration file (function.json). The name of `function.json`'s parent directory is always the name of your function.

The binding extensions required in [version 2.x](#) 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.

Exporting a function

JavaScript functions must be exported via `module.exports` (or `exports`). Your exported function should be a JavaScript function that executes when triggered.

By default, the Functions runtime looks for your function in `index.js`, where `index.js` shares the same parent directory as its corresponding `function.json`. In the default case, your exported function should be the only export from its file or the export named `run` or `index`. To configure the file location and export name of your function, read about [configuring your function's entry point](#) below.

Your exported function is passed a number of arguments on execution. The first argument it takes is always a `context` object. If your function is synchronous (doesn't return a Promise), you must pass the `context` object, as calling `context.done` is required for correct use.

```
// You should include context, other arguments are optional
module.exports = function(context, myTrigger, myInput, myOtherInput) {
    // function logic goes here :)
    context.done();
};
```

Exporting an async function

When using the `async function` declaration or plain JavaScript [Promises](#) in version 2.x of the Functions runtime, you do not need to explicitly call the `context.done` callback to signal that your function has completed. Your function completes when the exported async function/Promise completes. For functions targeting the version 1.x runtime, you must still call `context.done` when your code is done executing.

The following example is a simple function that logs that it was triggered and immediately completes execution.

```
module.exports = async function (context) {
    context.log('JavaScript trigger function processed a request.');
};
```

When exporting an async function, you can also configure an output binding to take the `return` value. This is recommended if you only have one output binding.

To assign an output using `return`, change the `name` property to `$return` in `function.json`.

```
{
  "type": "http",
  "direction": "out",
  "name": "$return"
}
```

In this case, your function should look like the following example:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');
    // You can call and await an async method here
    return {
        body: "Hello, world!"
    };
}
```

Bindings

In JavaScript, [bindings](#) are configured and defined in a function's `function.json`. Functions interact with

bindings a number of ways.

Inputs

Input are divided into two categories in Azure Functions: one is the trigger input and the other is the additional input. Trigger and other input bindings (bindings of `direction === "in"`) can be read by a function in three ways:

- **[Recommended] As parameters passed to your function.** They are passed to the function in the same order that they are defined in `function.json`. The `name` property defined in `function.json` does not need to match the name of your parameter, although it should.

```
module.exports = async function(context, myTrigger, myInput, myOtherInput) { ... };
```

- **As members of the `context.bindings` object.** Each member is named by the `name` property defined in `function.json`.

```
module.exports = async function(context) {
    context.log("This is myTrigger: " + context.bindings.myTrigger);
    context.log("This is myInput: " + context.bindings.myInput);
    context.log("This is myOtherInput: " + context.bindings.myOtherInput);
};
```

- **As inputs using the JavaScript `arguments` object.** This is essentially the same as passing inputs as parameters, but allows you to dynamically handle inputs.

```
module.exports = async function(context) {
    context.log("This is myTrigger: " + arguments[1]);
    context.log("This is myInput: " + arguments[2]);
    context.log("This is myOtherInput: " + arguments[3]);
};
```

Outputs

Outputs (bindings of `direction === "out"`) can be written to by a function in a number of ways. In all cases, the `name` property of the binding as defined in `function.json` corresponds to the name of the object member written to in your function.

You can assign data to output bindings in one of the following ways (don't combine these methods):

- **[Recommended for multiple outputs] Returning an object.** If you are using an `async/Promise` returning function, you can return an object with assigned output data. In the example below, the output bindings are named "httpResponse" and "queueOutput" in `function.json`.

```
module.exports = async function(context) {
    let retMsg = 'Hello, world!';
    return {
        httpResponse: {
            body: retMsg
        },
        queueOutput: retMsg
    };
};
```

If you are using a synchronous function, you can return this object using `context.done` (see example).

- **[Recommended for single output] Returning a value directly and using the `$return`**

binding name. This only works for async/Promise returning functions. See example in [exporting an async function](#).

- **Assigning values to** `context.bindings` You can assign values directly to context.bindings.

```
module.exports = async function(context) {
    let retMsg = 'Hello, world!';
    context.bindings.httpResponse = {
        body: retMsg
    };
    context.bindings.queueOutput = retMsg;
    return;
};
```

Bindings data type

To define the data type for an input binding, use the `dataType` property in the binding definition. For example, to read the content of an HTTP request in binary format, use the type `binary`:

```
{
    "type": "httpTrigger",
    "name": "req",
    "direction": "in",
    "dataType": "binary"
}
```

Options for `dataType` are: `binary`, `stream`, and `string`.

context object

The runtime uses a `context` object to pass data to and from your function and the runtime. Used to read and set data from bindings and for writing to logs, the `context` object is always the first parameter passed to a function.

For functions featuring synchronous code, the context object includes the `done` callback which you call when the function is done processing. Explicitly calling `done` is unnecessary when writing asynchronous code; the `done` callback is called implicitly.

```
module.exports = (context) => {

    // function logic goes here

    context.log("The function has executed.");

    context.done();
};
```

The context passed into your function exposes an `executionContext` property, which is an object with the following properties:

| PROPERTY NAME | TYPE | DESCRIPTION |
|---------------------------|--------|--|
| <code>invocationId</code> | String | Provides a unique identifier for the specific function invocation. |
| <code>functionName</code> | String | Provides the name of the running function |

| PROPERTY NAME | TYPE | DESCRIPTION |
|--------------------------------|--------|---------------------------------------|
| <code>functionDirectory</code> | String | Provides the functions app directory. |

The following example shows how to return the `invocationId`.

```
module.exports = (context, req) => {
    context.res = {
        body: context.executionContext.invocationId
    };
    context.done();
};
```

context.bindings property

```
context.bindings
```

Returns a named object that is used to read or assign binding data. Input and trigger binding data can be accessed by reading properties on `context.bindings`. Output binding data can be assigned by adding data to `context.bindings`.

For example, the following binding definitions in your function.json let you access the contents of a queue from `context.bindings.myInput` and assign outputs to a queue using `context.bindings.myOutput`.

```
{
  "type": "queue",
  "direction": "in",
  "name": "myInput"
  ...
},
{
  "type": "queue",
  "direction": "out",
  "name": "myOutput"
  ...
}
```

```
// myInput contains the input data, which may have properties such as "name"
var author = context.bindings.myInput.name;
// Similarly, you can set your output data
context.bindings.myOutput = {
    some_text: 'hello world',
    a_number: 1
};
```

You can choose to define output binding data using the `context.done` method instead of the `context.binding` object (see below).

context.bindingData property

```
context.bindingData
```

Returns a named object that contains trigger metadata and function invocation data (`invocationId`, `sys.methodName`, `sys.utcNow`, `sys.randGuid`). For an example of trigger metadata, see this [event hubs example](#).

context.done method

```
context.done([err],[propertyBag])
```

Lets the runtime know that your code has completed. When your function uses the [async function](#) declaration, you do not need to use `context.done()`. The `context.done` callback is implicitly called. Async functions are available in Node 8 or a later version, which requires version 2.x of the Functions runtime.

If your function is not an async function, **you must call** `context.done` to inform the runtime that your function is complete. The execution times out if it is missing.

The `context.done` method allows you to pass back both a user-defined error to the runtime and a JSON object containing output binding data. Properties passed to `context.done` overwrite anything set on the `context.bindings` object.

```
// Even though we set myOutput to have:  
//   -> text: 'hello world', number: 123  
context.bindings.myOutput = { text: 'hello world', number: 123 };  
// If we pass an object to the done function...  
context.done(null, { myOutput: { text: 'hello there, world', noNumber: true }});  
// the done method overwrites the myOutput binding to be:  
//   -> text: 'hello there, world', noNumber: true
```

context.log method

```
context.log(message)
```

Allows you to write to the streaming function logs at the default trace level, with other logging levels available. Trace logging is described in detail in the next section.

Write trace output to logs

In Functions, you use the `context.log` methods to write trace output to the logs and the console. When you call `context.log()`, your message is written to the logs at the default trace level, which is the *info* trace level. 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 telemetry and your trace outputs. To learn more, see [monitoring Azure Functions](#).

The following example writes a log at the info trace level, including the invocation ID:

```
context.log("Something has happened. " + context.invocationId);
```

All `context.log` methods support the same parameter format that's supported by the Node.js [util.format](#) method. Consider the following code, which writes function logs by using the default trace level:

```
context.log('Node.js HTTP trigger function processed a request. RequestUri=' + req.originalUrl);  
context.log('Request Headers = ' + JSON.stringify(req.headers));
```

You can also write the same code in the following format:

```
context.log('Node.js HTTP trigger function processed a request. RequestUri=%s', req.originalUrl);  
context.log('Request Headers = ', JSON.stringify(req.headers));
```

NOTE

Don't use `console.log` to write trace outputs. Because output from `console.log` is captured at the function app level, it's not tied to a specific function invocation and isn't displayed in a specific function's logs. Also, version 1.x of the Functions runtime doesn't support using `console.log` to write to the console.

Trace levels

In addition to the default level, the following logging methods are available that let you write function logs at specific trace levels.

| METHOD | DESCRIPTION |
|-------------------------------|---|
| <code>error(message)</code> | Writes an error-level event to the logs. |
| <code>warn(message)</code> | Writes a warning-level event to the logs. |
| <code>info(message)</code> | Writes to info level logging, or lower. |
| <code>verbose(message)</code> | Writes to verbose level logging. |

The following example writes the same log at the warning trace level, instead of the info level:

```
context.log.warn("Something has happened. " + context.invocationId);
```

Because `error` is the highest trace level, this trace is written to the output at all trace levels as long as logging is enabled.

Configure the trace level for logging

Functions lets you define the threshold trace level for writing to the logs or the console. The specific threshold settings depend on your version of the Functions runtime.

- [v2.x+](#)
- [v1.x](#)

To set the threshold for traces written to the logs, use the `logging.level` property in the host.json file. This JSON object lets you define a default threshold for all functions in your function app, plus you can define specific thresholds for individual functions. To learn more, see [How to configure monitoring for Azure Functions](#).

Log custom telemetry

By default, Functions writes output as traces to Application Insights. For more control, you can instead use the [Application Insights Node.js SDK](#) to send custom telemetry data to your Application Insights instance.

- [v2.x+](#)
- [v1.x](#)

```

const appInsights = require("applicationinsights");
appInsights.setup();
const client = appInsights.defaultClient;

module.exports = function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    // Use this with 'tagOverrides' to correlate custom telemetry to the parent function invocation.
    var operationIdOverride = {"ai.operation.id":context.traceContext.traceparent};

    client.trackEvent({name: "my custom event", tagOverrides:operationIdOverride, properties: {customProperty2: "custom property value"}});
    client.trackException({exception: new Error("handled exceptions can be logged with this method"), tagOverrides:operationIdOverride});
    client.trackMetric({name: "custom metric", value: 3, tagOverrides:operationIdOverride});
    client.trackTrace({message: "trace message", tagOverrides:operationIdOverride});
    client.trackDependency({target:"http://dbname", name:"select customers proc", data:"SELECT * FROM Customers", duration:231, resultCode:0, success: true, dependencyTypeName: "ZSQL", tagOverrides:operationIdOverride});
    client.trackRequest({name:"GET /customers", url:"http://myserver/customers", duration:309, resultCode:200, success:true, tagOverrides:operationIdOverride});

    context.done();
};

```

The `tagOverrides` parameter sets the `operation_Id` to the function's invocation ID. This setting enables you to correlate all of the automatically generated and custom telemetry for a given function invocation.

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 `context.req` (request) object has the following properties:

| PROPERTY | DESCRIPTION |
|--------------------------|--|
| <code>body</code> | An object that contains the body of the request. |
| <code>headers</code> | An object that contains the request headers. |
| <code>method</code> | The HTTP method of the request. |
| <code>originalUrl</code> | The URL of the request. |
| <code>params</code> | An object that contains the routing parameters of the request. |
| <code>query</code> | An object that contains the query parameters. |
| <code>rawBody</code> | The body of the message as a string. |

Response object

The `context.res` (response) object has the following properties:

| PROPERTY | DESCRIPTION |
|----------------|--|
| <i>body</i> | An object that contains the body of the response. |
| <i>headers</i> | An object that contains the response headers. |
| <i>isRaw</i> | Indicates that formatting is skipped for the response. |
| <i>status</i> | The HTTP status code of the response. |
| <i>cookies</i> | An array of HTTP cookie objects that are set in the response. An HTTP cookie object has a <code>name</code> , <code>value</code> , and other cookie properties, such as <code>maxAge</code> or <code>sameSite</code> . |

Accessing the request and response

When you work with HTTP triggers, you can access the HTTP request and response objects in a number of ways:

- From `req` and `res` properties on the `context` object. In this way, you can use the conventional pattern to access HTTP data from the context object, instead of having to use the full `context.bindings.name` pattern. The following example shows how to access the `req` and `res` objects on the `context`:

```
// You can access your HTTP request off the context ...
if(context.req.body.emoji === ':pizza:') context.log('Yay!');
// and also set your HTTP response
context.res = { status: 202, body: 'You successfully ordered more coffee!' };
```

- From the named input and output bindings. In this way, the HTTP trigger and bindings work the same as any other binding. The following example sets the response object by using a named `response` binding:

```
{
  "type": "http",
  "direction": "out",
  "name": "response"
}
```

```
context.bindings.response = { status: 201, body: "Insert succeeded." };
```

- [Response only] By calling `context.res.send(body?: any)`. An HTTP response is created with input `body` as the response body. `context.done()` is implicitly called.
- [Response only] By calling `context.done()`. A special type of HTTP binding returns the response that is passed to the `context.done()` method. The following HTTP output binding defines a `$return` output parameter:

```
{
  "type": "http",
  "direction": "out",
  "name": "$return"
}
```

```
// Define a valid response object.
res = { status: 201, body: "Insert succeeded." };
context.done(null, res);
```

Scaling and concurrency

By default, Azure Functions automatically monitors the load on your application and creates additional host instances for Node.js as needed. 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.

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.

The [FUNCTIONS_WORKER_PROCESS_COUNT](#) applies to each host that Functions creates when scaling out your application to meet demand.

Node version

The following table shows current supported Node.js versions for each major version of the Functions runtime, by operating system:

| FUNCTIONS VERSION | NODE VERSION (WINDOWS) | NODE VERSION (LINUX) |
|-------------------|---|---|
| 1.x | 6.11.2 (locked by the runtime) | n/a |
| 2.x | <div style="display: flex; justify-content: space-around;"> ~8 ~10 (recommended) ~12 </div> | <div style="display: flex; justify-content: space-around;"> node 8 node 10 (recommended) </div> |
| 3.x | <div style="display: flex; justify-content: space-around;"> ~10 ~12 (recommended) ~14 (preview) </div> | <div style="display: flex; justify-content: space-around;"> node 10 node 12 (recommended) node 14 (preview) </div> |

You can see the current version that the runtime is using by logging `process.version` from any function.

Setting the Node version

For Windows function apps, target the version in Azure by setting the [WEBSITE_NODE_DEFAULT_VERSION](#) [app setting](#) to a supported LTS version, such as `~12`.

For Linux function apps, run the following Azure CLI command to update the Node version.

```
az functionapp config set --linux-fx-version "node|12" --name "<MY_APP_NAME>" --resource-group "<MY_RESOURCE_GROUP_NAME>"
```

Dependency management

In order to use community libraries in your JavaScript code, as is shown in the below example, you need to

ensure that all dependencies are installed on your Function App in Azure.

```
// Import the underscore.js library
var _ = require('underscore');
var version = process.version; // version === 'v6.5.0'

module.exports = function(context) {
    // Using our imported underscore.js library
    var matched_names = _
        .where(context.bindings.myInput.names, {first: 'Carla'});
```

NOTE

You should define a `package.json` file at the root of your Function App. Defining the file lets all functions in the app share the same cached packages, which gives the best performance. If a version conflict arises, you can resolve it by adding a `package.json` file in the folder of a specific function.

When deploying Function Apps from source control, any `package.json` file present in your repo, will trigger an `npm install` in its folder during deployment. But when deploying via the Portal or CLI, you will have to manually install the packages.

There are two ways to install packages on your Function App:

Deploying with Dependencies

1. Install all requisite packages locally by running `npm install`.
2. Deploy your code, and ensure that the `node_modules` folder is included in the deployment.

Using Kudu

1. Go to https://<function_app_name>.scm.azurewebsites.net.
2. Click **Debug Console > CMD**.
3. Go to `D:\home\site\wwwroot`, and then drag your `package.json` file to the `wwwroot` folder at the top half of the page.
You can upload files to your function app in other ways also. For more information, see [How to update function app files](#).
4. After the `package.json` file is uploaded, run the `npm install` command in the **Kudu remote execution console**.
This action downloads the packages indicated in the `package.json` file and restarts the function app.

Environment variables

Add your own environment variables to a function app, in both your local and cloud environments, such as operational secrets (connection strings, keys, and endpoints) or environmental settings (such as profiling variables). Access these settings using `process.env` in your function code.

In local development environment

When running locally, your functions project includes a `local.settings.json` file, where you store your environment variables in the `Values` object.

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "",  
        "FUNCTIONS_WORKER_RUNTIME": "node",  
        "translatorTextEndPoint": "https://api.cognitive.microsofttranslator.com/",  
        "translatorTextKey": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
        "languageWorkers_node_arguments": "--prof"  
    }  
}
```

In Azure cloud environment

When running in Azure, the function app lets you set uses [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](#).

Access environment variables in code

Access application settings as environment variables using `process.env`, as shown here in the second and third calls to `context.log()` where we log the `AzureWebJobsStorage` and `WEBSITE_SITE_NAME` environment variables:

```
module.exports = async function (context, myTimer) {  
  
    context.log("AzureWebJobsStorage: " + process.env["AzureWebJobsStorage"]);  
    context.log("WEBSITE_SITE_NAME: " + process.env["WEBSITE_SITE_NAME"]);  
};
```

Configure function entry point

The `function.json` properties `scriptFile` and `entryPoint` can be used to configure the location and name of your exported function. These properties can be important when your JavaScript is transpiled.

Using `scriptFile`

By default, a JavaScript function is executed from `index.js`, a file that shares the same parent directory as its corresponding `function.json`.

`scriptFile` can be used to get a folder structure that looks like the following example:

```
FunctionApp  
| - host.json  
| - myNodeFunction  
| | - function.json  
| - lib  
| | - sayHello.js  
| - node_modules  
| | - ... packages ...  
| - package.json
```

The `function.json` for `myNodeFunction` should include a `scriptFile` property pointing to the file with the exported function to run.

```
{  
  "scriptFile": "../lib/sayHello.js",  
  "bindings": [  
    ...  
  ]  
}
```

Using `entryPoint`

In `scriptFile` (or `index.js`), a function must be exported using `module.exports` in order to be found and run. By default, the function that executes when triggered is the only export from that file, the export named `run`, or the export named `index`.

This can be configured using `entryPoint` in `function.json`, as in the following example:

```
{  
  "entryPoint": "logFoo",  
  "bindings": [  
    ...  
  ]  
}
```

In Functions v2.x, which supports the `this` parameter in user functions, the function code could then be as in the following example:

```
class MyObj {  
  constructor() {  
    this.foo = 1;  
  };  
  
  logFoo(context) {  
    context.log("Foo is " + this.foo);  
    context.done();  
  }  
}  
  
const myObj = new MyObj();  
module.exports = myObj;
```

In this example, it is important to note that although an object is being exported, there are no guarantees for preserving state between executions.

Local Debugging

When started with the `--inspect` parameter, a Node.js process listens for a debugging client on the specified port. In Azure Functions 2.x, you can specify arguments to pass into the Node.js process that runs your code by adding the environment variable or App Setting `languageWorkers:node:arguments = <args>`.

To debug locally, add `"languageWorkers:node:arguments": "--inspect=5858"` under `Values` in your `local.settings.json` file and attach a debugger to port 5858.

When debugging using VS Code, the `--inspect` parameter is automatically added using the `port` value in the project's `launch.json` file.

In version 1.x, setting `languageWorkers:node:arguments` will not work. The debug port can be selected with the `--nodeDebugPort` parameter on Azure Functions Core Tools.

TypeScript

When you target version 2.x of the Functions runtime, both [Azure Functions for Visual Studio Code](#) and the [Azure Functions Core Tools](#) let you create function apps using a template that support TypeScript function app projects. The template generates `package.json` and `tsconfig.json` project files that make it easier to transpile, run, and publish JavaScript functions from TypeScript code with these tools.

A generated `.funcignore` file is used to indicate which files are excluded when a project is published to Azure.

TypeScript files (.ts) are transpiled into JavaScript files (.js) in the `dist` output directory. TypeScript templates use the `scriptFile` parameter in `function.json` to indicate the location of the corresponding .js file in the `dist` folder. The output location is set by the template by using `outDir` parameter in the `tsconfig.json` file. If you change this setting or the name of the folder, the runtime is not able to find the code to run.

The way that you locally develop and deploy from a TypeScript project depends on your development tool.

Visual Studio Code

The [Azure Functions for Visual Studio Code](#) extension lets you develop your functions using TypeScript. The Core Tools is a requirement of the Azure Functions extension.

To create a TypeScript function app in Visual Studio Code, choose `TypeScript` as your language when you create a function app.

When you press **F5** to run the app locally, transpilation is done before the host (`func.exe`) is initialized.

When you deploy your function app to Azure using the **Deploy to function app...** button, the Azure Functions extension first generates a production-ready build of JavaScript files from the TypeScript source files.

Azure Functions Core Tools

There are several ways in which a TypeScript project differs from a JavaScript project when using the Core Tools.

Create project

To create a TypeScript function app project using Core Tools, you must specify the TypeScript language option when you create your function app. You can do this in one of the following ways:

- Run the `func init` command, select `node` as your language stack, and then select `typescript`.
- Run the `func init --worker-runtime typescript` command.

Run local

To run your function app code locally using Core Tools, use the following commands instead of `func host start`:

```
npm install  
npm start
```

The `npm start` command is equivalent to the following commands:

- `npm run build`
- `func extensions install`
- `tsc`
- `func start`

Publish to Azure

Before you use the `func azure functionapp publish` command to deploy to Azure, you create a production-ready build of JavaScript files from the TypeScript source files.

The following commands prepare and publish your TypeScript project using Core Tools:

```
npm run build:production
func azure functionapp publish <APP_NAME>
```

In this command, replace `<APP_NAME>` with the name of your function app.

Considerations for JavaScript functions

When you work with JavaScript functions, be aware of the considerations in the following sections.

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 JavaScript functions more efficiently on single-vCPU VMs, and using larger VMs does not 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](#).

Cold Start

When developing Azure Functions in the serverless hosting model, cold starts are a reality. *Cold start* refers to the fact that when your function app starts for the first time after a period of inactivity, it takes longer to start up. For JavaScript functions 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 the run from package model by default, but if you're experiencing large cold starts and are not running this way, this change can offer a significant improvement.

Connection Limits

When you use a service-specific client in an Azure Functions application, don't create a new client with every function invocation. 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 JavaScript, 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 are not properly awaited.

In the example below, the asynchronous method `fs.readFile` is invoked with an error-first callback function as its second parameter. This code causes both of the issues mentioned above. An exception that is not explicitly caught in the correct scope crashed the entire process (issue #1). Calling `context.done()` outside of the scope of the callback function means that the function invocation may end before the file is read (issue #2). In this example, calling `context.done()` too early results in missing log entries starting with `Data from file: .`

```
// NOT RECOMMENDED PATTERN
const fs = require('fs');

module.exports = function (context) {
    fs.readFile('./hello.txt', (err, data) => {
        if (err) {
            context.log.error('ERROR', err);
            // BUG #1: This will result in an uncaught exception that crashes the entire process
            throw err;
        }
        context.log(`Data from file: ${data}`);
        // context.done() should be called here
    });
    // BUG #2: Data is not guaranteed to be read before the Azure Function's invocation ends
    context.done();
}
```

Using the `async` and `await` keywords helps avoid both of these errors. You should use the Node.js utility function `util.promisify` to turn error-first callback-style functions into awaitable functions.

In the example below, any unhandled exceptions thrown during the function execution only fail the individual invocation that raised an exception. The `await` keyword means that steps following `readFileAsync` only execute after `readFile` is complete. With `async` and `await`, you also don't need to call the `context.done()` callback.

```
// Recommended pattern
const fs = require('fs');
const util = require('util');
const readFileAsync = util.promisify(fs.readFile);

module.exports = async function (context) {
    let data;
    try {
        data = await readFileAsync('./hello.txt');
    } catch (err) {
        context.log.error('ERROR', err);
        // This rethrown exception will be handled by the Functions Runtime and will only fail the
        // individual invocation
        throw err;
    }
    context.log(`Data from file: ${data}`);
}
```

Next steps

For more information, see the following resources:

- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)

Shifting from Express.js to Azure Functions

11/2/2020 • 3 minutes to read • [Edit Online](#)

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.

```
// 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.

```
| - hello  
| - function.json  
| - index.js
```

The following example implements the same result as the above Express.js endpoint, but with Azure Functions.

- [JavaScript](#)
- [TypeScript](#)

```
// 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 `context.done()` to complete execution, while `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.

```
{  
    "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

12/4/2020 • 17 minutes to read • [Edit Online](#)

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 [Azure 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.

Security Center

Security Center 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 the real-time security features of Security Center, for an additional 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 activity and logging analytics. Functions integrates with Application Insights to collects 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 Azure Sentinel to this workspace. To learn more, see [What is Azure Sentinel](#).

For more security recommendations for observability, see the [Azure security baseline for Azure Functions](#).

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 \(TSL\)](#).

Function access keys

Functions lets you use keys to make it harder to access your HTTP function endpoints during development. Unless the HTTP access level on an HTTP triggered function is set to `anonymous`, requests must include an API access key in the request.

While keys provide a default security mechanism, you may want to consider additional options to secure an HTTP endpoint in production. For example, it's generally not a good practice to distribute shared secret in public apps. If your function is being called from a public client, you may want to consider implementing another security mechanism. To learn more, see [Secure an HTTP endpoint in production](#).

When you renew your function key values, you must manually redistribute the updated key values to all clients that call your function.

Authorization scopes (function-level)

There are two access scopes for function-level keys:

- **Function:** These keys apply only to the specific functions under which they are defined. When used as an API key, these only allow access to that function.
- **Host:** Keys with a host scope can be used to access all functions within the function app. When used as an API key, these allow access to any function within the function app.

Each key is named for reference, and there is 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.

Master key (admin-level)

Each function app also has an admin-level host key named `_master`. In addition to providing host-level access to all functions in the app, the master key also provides administrative access to the runtime REST APIs. This key cannot be revoked. When you set an access level of `admin`, requests must use the master key; any other key results in access failure.

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 native client applications. Use caution when choosing the admin access level.

System key

Specific extensions may require a system-managed key to access webhook endpoints. System keys are designed for extension-specific function endpoints that 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](#).

The scope of system keys is determined by the extension, but it generally applies to the entire function app. 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.

Keys comparison

The following table compares the uses for various kinds of access keys:

| ACTION | SCOPE | VALID KEYS |
|---|---------------------------|---------------------|
| Execute a function | Specific function | Function |
| Execute a function | Any function | Function or host |
| Call an admin endpoint | Function app | Host (master only) |
| Call Durable Task extension APIs | Function app ¹ | System ² |
| Call an extension-specific Webhook (internal) | Function app ¹ | system ² |

¹Scope determined by the extension.

²Specific names set by extension.

To learn more about access keys, see the [HTTP trigger binding article](#).

Secret repositories

By default, keys are stored in a Blob storage container in the account provided by the `AzureWebJobsStorage` setting.

You can use specific application settings to override this behavior and store keys in a different location.

| LOCATION | SETTING | VALUE | DESCRIPTION |
|---------------------------|--|-----------------------------------|--|
| Different storage account | <code>AzureWebJobsSecretStorageSas</code> | <code><BLOB_SAS_URL></code> | Stores keys in Blob storage of a second storage account, based on the provided SAS URL. Keys are encrypted before being stored using a secret unique to your function app. |
| File system | <code>AzureWebJobsSecretStorageType</code> | <code>files</code> | Keys are persisted on the file system, encrypted before storage using a secret unique to your function app. |
| Azure Key Vault | <code>AzureWebJobsSecretStorageType</code> | <code>keyvault</code> | 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: <code>Get</code> , <code>Set</code> , <code>List</code> , and <code>Delete</code> . When running locally, the developer identity is used, and settings must be in the local.settings.json file . |

| LOCATION | SETTING | VALUE | DESCRIPTION |
|--------------------|---------|---|---|
| Kubernetes Secrets | | AzureWebJobsSecretStorageType kubernetes AzureWebJobsKubernetesSecretName <SECRETS_RESOURCE> (optional) | Supported only when running the Functions runtime in 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. |

Authentication/authorization

While function 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.

Enable App Service Authentication/Authorization

The App Service platform lets you use Azure Active Directory (AAD) 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 a variety of 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. 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 Azure Active Directory (Azure AD) allows your app to easily access other Azure AD-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 Azure AD, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to your application and is deleted if your 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.

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. But, 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. To learn more, see the `IsEncrypted` property in 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](#).

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 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](#).

Private site access

[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.

When creating an inbound private endpoint connection for functions, you will also need a DNS record to resolve the private address. By default a private DNS record will be created for you when creating a private endpoint using the Azure portal.

To learn more, see [using Private Endpoints for Web Apps](#).

Deploy your function app in isolation

Azure App Service Environment (ASE) provides a dedicated hosting environment in which to run your functions. ASE lets 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](#)

Azure security baseline for Azure Functions

12/4/2020 • 35 minutes to read • [Edit Online](#)

The Azure Security Baseline for Azure Functions contains recommendations that will help you improve the security posture of your deployment.

The baseline for this service is drawn from the [Azure Security Benchmark version 1.0](#), which provides recommendations on how you can secure your cloud solutions on Azure with our best practices guidance.

For more information, see the [Azure security baselines overview](#).

Network security

For more information, see [Security control: Network security](#).

1.1: Protect resources using Network Security Groups or Azure Firewall on your Virtual Network

Guidance: Integrate your Azure Functions apps with an Azure virtual network. Function apps running in the Premium plan have the same hosting capabilities as web apps in Azure App Service, which includes the "VNet Integration" feature. Azure virtual networks allow you to place many of your Azure resources, such as Azure Functions, in a non-internet routable network.

- [How to integrate Functions with an Azure Virtual Network](#)
- [Understand Vnet Integration for Azure Functions and Azure App Service](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.2: Monitor and log the configuration and traffic of Vnets, Subnets, and NICs

Guidance: Use Azure Security Center and follow network protection recommendations to help secure network resources and network configurations related to your Azure Functions apps.

If using Network Security groups (NSGs) with your Azure Functions implementation, enable NSG flow logs and send logs into an Azure Storage Account for traffic audits. You may also send NSG flow logs to a Log Analytics workspace and use Traffic Analytics to provide insights into traffic flow in your Azure cloud. Some advantages of Traffic Analytics are the ability to visualize network activity and identify hot spots, identify security threats, understand traffic flow patterns, and pinpoint network misconfigurations.

- [Understand Network Security provided by Azure Security Center](#)
- [How to Enable NSG Flow Logs](#)
- [How to Enable and use Traffic Analytics](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

1.3: Protect critical web applications

Guidance: To fully secure your Azure Function endpoints in production, you should consider implementing one of the following function app-level security options:

- Turn on App Service Authentication / Authorization for your function app,
- Use Azure API Management (APIM) to authenticate requests, or

- Deploy your function app to an Azure App Service Environment.

In addition, ensure remote debugging has been disabled for your production Azure Functions. Furthermore, Cross-Origin Resource Sharing (CORS) should not allow all domains to access your Azure Function app. Allow only required domains to interact with your Azure Function app.

Consider deploying Azure Web Application Firewall (WAF) as part of the networking configuration for additional inspection of incoming traffic. Enable Diagnostic Setting for WAF and ingest logs into a Storage Account, Event Hub, or Log Analytics Workspace.

- [How to secure Azure Function endpoints in production](#)
- [How to deploy Azure WAF](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

1.4: Deny communications with known malicious IP addresses

Guidance: Enable DDoS Protection Standard on the Virtual Networks associated with your functions apps to guard against DDoS attacks. Use Azure Security Center Integrated Threat Intelligence to deny communications with known malicious or unused public IP addresses. In addition, configure a front-end gateway, such as Azure Web Application Firewall, to authenticate all incoming requests and filter out malicious traffic. Azure Web Application Firewall can help secure your Azure Function apps by inspecting inbound web traffic to block SQL injections, Cross-Site Scripting, malware uploads, and DDoS attacks. Introduction of a WAF requires either an App Service Environment or use of Private Endpoints (Preview). Ensure that Private Endpoints are no longer in (Preview) before using them with production workloads.

- [Azure Functions networking options](#)
- [Azure Functions Premium Plan](#)
- [Introduction to the App Service Environments](#)
- [Networking considerations for an App Service Environment](#)
- [How to configure DDoS protection](#)
- [How to deploy Azure Firewall](#)
- [Understand Azure Security Center Integrated Threat Intelligence](#)
- [Understand Azure Security Center Adaptive Network Hardening](#)
- [Understand Azure Security Center Just In Time Network Access Control](#)
- [Using Private Endpoints for Azure Functions](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

1.5: Record network packets and flow logs

Guidance: If using Network Security groups (NSGs) with your Azure Functions implementation, enable Network Security Group flow logs and send logs into a storage account for traffic audit. You may also send flow logs to a Log Analytics workspace and use Traffic Analytics to provide insights into traffic flow in your Azure cloud. Some advantages of Traffic Analytics are the ability to visualize network activity and identify hot spots, identify security threats, understand traffic flow patterns, and pinpoint network misconfigurations.

- [How to Enable NSG Flow Logs](#)

- [How to Enable and use Traffic Analytics](#)
- [How to enable Network Watcher](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

1.6: Deploy network based intrusion detection/intrusion prevention systems (IDS/IPS)

Guidance: Configure a front-end gateway such as Azure Web Application Firewall to authenticate all incoming requests and filter out malicious traffic. Azure Web Application Firewall can help secure your function apps by inspecting inbound web traffic to block SQL injections, Cross-Site Scripting, malware uploads, and DDoS attacks. Introduction of a WAF requires either an App Service Environment or use of Private Endpoints (Preview). Ensure that Private Endpoints are no longer in (Preview) before using them with production workloads.

Alternatively, there are multiple marketplace options like the Barracuda WAF for Azure that are available on the Azure Marketplace which include IDS/IPS features.

- [Azure Functions networking options](#)
- [Azure Functions Premium Plan](#)
- [Introduction to the App Service Environments](#)
- [Networking considerations for an App Service Environment](#)
- [Understand Azure Web Application Firewall](#)
- [Using Private Endpoints for Azure Functions](#)
- [Understand Barracuda WAF Cloud Service](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.7: Manage traffic to web applications

Guidance: Configure a front-end gateway for your network such as Azure Web Application Firewall with end-to-end TLS encryption. Introduction of a WAF requires either an App Service Environment or use of Private Endpoints (Preview). Ensure that Private Endpoints are no longer in (Preview) before using them with production workloads.

- [Azure Functions networking options](#)
- [Azure Functions Premium Plan](#)
- [Introduction to the App Service Environments](#)
- [Networking considerations for an App Service Environment](#)
- [Understand Azure Web Application Firewall](#)
- [How to configure end-to-end TLS by using Application Gateway with the portal](#)
- [Using Private Endpoints for Azure Functions](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.8: Minimize complexity and administrative overhead of network security rules

Guidance: Use Virtual Network service tags to define network access controls on Network Security Groups or Azure Firewall. You can use service tags in place of specific IP addresses when creating security rules. By specifying

the service tag name (e.g., AzureAppService) in the appropriate source or destination field of a rule, you can allow or deny the traffic for the corresponding service. Microsoft manages the address prefixes encompassed by the service tag and automatically updates the service tag as addresses change.

- [For more information about using service tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

1.9: Maintain standard security configurations for network devices

Guidance: Define and implement standard security configurations for network settings related to your Azure Functions. Use Azure Policy aliases in the "Microsoft.Web" and "Microsoft.Network" namespaces to create custom policies to audit or enforce the network configuration of your Azure Functions. You may also make use of built-in policy definitions for Azure Functions, such as:

- CORS should not allow every resource to access your Function Apps
- Function App should only be accessible over HTTPS
- Latest TLS version should be used in your Function App

You may also use Azure Blueprints to simplify large-scale Azure deployments by packaging key environment artifacts, such as Azure Resource Manager templates, Azure role-based access control (Azure RBAC), and policies in a single blueprint definition. You can easily apply the blueprint to new subscriptions, environments, and fine-tune control and management through versioning.

- [How to configure and manage Azure Policy](#)
- [How to create an Azure Blueprint](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

1.10: Document traffic configuration rules

Guidance: If using Network Security groups (NSGs) with your Azure Functions implementation, use tags for the NSGs and other resources related to network security and traffic flow. For individual NSG rules, use the "Description" field to specify business need and/or duration (etc.) for any rules that allow traffic to/from a network.

Use any of the built-in Azure policy definitions related to tagging, such as "Require tag and its value" to ensure that all resources are created with tags and to notify you of existing untagged resources.

You may use Azure PowerShell or Azure CLI to look-up or perform actions on resources based on their tags.

- [How to create and use tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

1.11: Use automated tools to monitor network resource configurations and detect changes

Guidance: Use Azure Activity Log to monitor network resource configurations and detect changes for network settings and resources related to your Azure Functions deployments. Create alerts within Azure Monitor that will trigger when changes to critical network settings or resources takes place.

- [How to view and retrieve Azure Activity Log events](#)
- [How to create alerts in Azure Monitor](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Logging and monitoring

For more information, see [Security control: Logging and monitoring](#).

2.1: Use approved time synchronization sources

Guidance: Microsoft maintains the time source used for Azure resources such as Azure Functions for timestamps in the logs.

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

2.2: Configure central security log management

Guidance: For control plane audit logging, enable Azure Activity Log diagnostic settings and send the logs to a Log Analytics workspace, Azure event hub, or Azure storage account for archive. Using Azure Activity Log data, you can determine the "what, who, and when" for any write operations (PUT, POST, DELETE) performed at the control plane level for your Azure resources.

Azure Functions also offers built-in integration with Azure Application Insights to monitor functions. Application Insights collects log, performance, and error data. It automatically detects performance anomalies and includes powerful analytics tools to help you diagnose issues and to understand how your functions are used.

If you have built-in custom security/audit logging within your Azure Function app, enable the diagnostics setting "FunctionAppLogs" and send the logs to a Log Analytics workspace, Azure event hub, or Azure storage account for archive.

Optionally, you may enable and on-board data to Azure Sentinel or a third-party SIEM.

- [How to enable Diagnostic Settings for Azure Activity Log](#)
- [How to set up Azure Functions with Azure Application Insights](#)
- [How to enable Diagnostic Settings \(user-generated logs\) for Azure Functions](#)
- [How to onboard Azure Sentinel](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

2.3: Enable audit logging for Azure resources

Guidance: For control plane audit logging, enable Azure Activity Log diagnostic settings and send the logs to a Log Analytics workspace, Azure event hub, or Azure storage account for archive. Using Azure Activity Log data, you can determine the "what, who, and when" for any write operations (PUT, POST, DELETE) performed at the control plane level for your Azure resources.

If you have built-in custom security/audit logging within your Azure Function app, enable the diagnostics setting "FunctionAppLogs" and send the logs to a Log Analytics workspace, Azure event hub, or Azure storage account for archive.

- [How to enable Diagnostic Settings for Azure Activity Log](#)
- [How to enable Diagnostic Settings \(user-generated logs\) for Azure Functions](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

2.4: Collect security logs from operating systems

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.5: Configure security log storage retention

Guidance: In Azure Monitor, set log retention period for Log Analytics workspaces associated with your Azure Functions apps according to your organization's compliance regulations.

- [How to set log retention parameters](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.6: Monitor and review Logs

Guidance: Enable Azure Activity Log diagnostic settings as well as the diagnostic settings for your Azure Functions app and send the logs to a Log Analytics workspace. Perform queries in Log Analytics to search terms, identify trends, analyze patterns, and provide many other insights based on the collected data.

Enable Application Insights for your Azure Functions apps to collect log, performance, and error data. You can view the telemetry data collected by Application Insights within the Azure portal.

If you have built-in custom security/audit logging within your Azure Function app, enable the diagnostics setting "FunctionAppLogs" and send the logs to a Log Analytics workspace, Azure event hub, or Azure storage account for archive.

Optionally, you may enable and on-board data to Azure Sentinel or a third-party SIEM.

- [How to enable diagnostic settings for Azure Activity Log](#)
- [How to enable diagnostic settings for Azure Functions](#)
- [How to set up Azure Functions with Azure Application Insights and view the telemetry data](#)
- [How to onboard Azure Sentinel](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.7: Enable alerts for anomalous activity

Guidance: Enable Azure Activity Log diagnostic settings as well as the diagnostic settings for your Azure Functions app and send the logs to a Log Analytics workspace. Perform queries in Log Analytics to search terms, identify trends, analyze patterns, and provide many other insights based on the collected data. You can create alerts based on your Log Analytics workspace queries.

Enable Application Insights for your Azure Functions apps to collect log, performance, and error data. You can view the telemetry data collected by Application Insights and create alerts within the Azure portal.

Optionally, you may enable and on-board data to Azure Sentinel or a third-party SIEM.

- [How to enable diagnostic settings for Azure Activity Log](#)
- [How to enable diagnostic settings for Azure Functions](#)
- [How to enable Application Insights for Azure Functions](#)
- [How to create alerts within Azure](#)

- [How to onboard Azure Sentinel](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.8: Centralize anti-malware logging

Guidance: Not applicable; Azure Functions apps do not process or produce anti-malware related logs.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.9: Enable DNS query logging

Guidance: Not applicable; Azure Functions apps do not process or produce user accessible DNS-related logs.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.10: Enable command-line audit logging

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Identity and access control

For more information, see [Security control: Identity and access control](#).

3.1: Maintain an inventory of administrative accounts

Guidance: Azure Active Directory (AD) has built-in roles that must be explicitly assigned and are queryable. Use the Azure AD PowerShell module to perform ad hoc queries to discover accounts that are members of administrative groups.

- [How to get a directory role in Azure AD with PowerShell](#)
- [How to get members of a directory role in Azure AD with PowerShell](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.2: Change default passwords where applicable

Guidance: Control plane access to Azure Functions is controlled through Azure Active Directory (AD). Azure AD does not have the concept of default passwords.

Data plane access can be controlled through several means, including authorization keys, network restrictions, and validating an Azure AD identity. Authorization keys are used by the clients connecting to your Azure Functions HTTP endpoints and can be regenerated at any time. These keys are generated for new HTTP endpoints by default.

Multiple deployment methods are available to function apps, some of which may leverage a set of generated credentials. Review the deployment methods that will be used for your application.

- [Secure an HTTP endpoint](#)
- [How to obtain and regenerate authorization keys](#)
- [Deployment technologies in Azure Functions](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.3: Use dedicated administrative accounts

Guidance: Create standard operating procedures around the use of dedicated administrative accounts. Use Azure Security Center Identity and Access Management to monitor the number of administrative accounts.

Additionally, to help you keep track of dedicated administrative accounts, you may use recommendations from Azure Security Center or built-in Azure Policies, such as: There should be more than one owner assigned to your subscription Deprecated accounts with owner permissions should be removed from your subscription External accounts with owner permissions should be removed from your subscription

- [How to use Azure Security Center to monitor identity and access \(Preview\)](#)
- [How to use Azure Policy](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.4: Use single sign-on (SSO) with Azure Active Directory

Guidance: Wherever possible, use Azure Active Directory SSO instead than configuring individual stand-alone credentials for data access to your function app. Use Azure Security Center Identity and Access Management recommendations. Implement single sign-on for your Azure Functions apps using the App Service Authentication / Authorization feature.

- [Understand authentication and authorization in Azure Functions](#)
- [Understand SSO with Azure AD](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.5: Use multi-factor authentication for all Azure Active Directory based access

Guidance: Enable Azure Active Directory (AD) Multi-Factor Authentication (MFA) and follow Azure Security Center Identity and Access Management recommendations.

- [How to enable MFA in Azure](#)
- [How to monitor identity and access within Azure Security Center](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.6: Use dedicated machines (Privileged Access Workstations) for all administrative tasks

Guidance: Use privileged access workstations (PAW) with Multi-Factor Authentication (MFA) configured to log into and configure Azure resources.

- [Learn about Privileged Access Workstations](#)
- [How to enable MFA in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.7: Log and alert on suspicious activity from administrative accounts

Guidance: Use Azure Active Directory (AD) Privileged Identity Management (PIM) for generation of logs and alerts

when suspicious or unsafe activity occurs in the environment.

In addition, use Azure AD risk detections to view alerts and reports on risky user behavior.

- [How to deploy Privileged Identity Management \(PIM\)](#)
- [Understand Azure AD risk detections](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.8: Manage Azure resources from only approved locations

Guidance: Use Conditional Access Named Locations to allow access to the Azure portal from only specific logical groupings of IP address ranges or countries/regions.

- [How to configure Named Locations in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.9: Use Azure Active Directory

Guidance: Use Azure Active Directory (AD) as the central authentication and authorization system for your Azure Functions apps. Azure AD protects data by using strong encryption for data at rest and in transit. Azure AD also salts, hashes, and securely stores user credentials.

- [How to configure your Azure Functions app to use Azure AD login](#)
- [How to create and configure an Azure AD instance](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.10: Regularly review and reconcile user access

Guidance: Azure Active Directory (AD) provides logs to help you discover stale accounts. In addition, use Azure Identity Access Reviews to efficiently manage group memberships, access to enterprise applications, and role assignments. User access can be reviewed on a regular basis to make sure only the right Users have continued access.

- [Understand Azure AD reporting](#)
- [How to use Azure Identity Access Reviews](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.11: Monitor attempts to access deactivated accounts

Guidance: Use Azure Active Directory (AD) as the central authentication and authorization system for your Azure Function apps. Azure AD protects data by using strong encryption for data at rest and in transit. Azure AD also salts, hashes, and securely stores user credentials.

You have access to Azure AD sign-in activity, audit and risk event log sources, which allow you to integrate with Azure Sentinel or a third-party SIEM.

You can streamline this process by creating diagnostic settings for Azure AD user accounts and sending the audit logs and sign-in logs to a Log Analytics workspace. You can configure desired log alerts within Log Analytics.

- [How to configure your Azure Functions app to use Azure AD login](#)

- [How to integrate Azure Activity Logs into Azure Monitor](#)
- [How to on-board Azure Sentinel](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.12: Alert on account login behavior deviation

Guidance: Use Azure Active Directory (AD) as the central authentication and authorization system for your Azure Functions apps. For account login behavior deviation on the control plane (the Azure portal), use Azure Active Directory (AD) Identity Protection and risk detection features to configure automated responses to detected suspicious actions related to user identities. You can also ingest data into Azure Sentinel for further investigation.

- [How to view Azure AD risky sign-ins](#)
- [How to configure and enable Identity Protection risk policies](#)
- [How to onboard Azure Sentinel](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.13: Provide Microsoft with access to relevant customer data during support scenarios

Guidance: Not currently available; Customer Lockbox is not currently supported for Azure Functions.

- [List of Customer Lockbox-supported services](#)

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Data protection

For more information, see [Security control: Data protection](#).

4.1: Maintain an inventory of sensitive Information

Guidance: Use tags to assist in tracking Azure resources that store or process sensitive information.

- [How to create and use tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.2: Isolate systems storing or processing sensitive information

Guidance: Implement separate subscriptions and/or management groups for development, test, and production. Azure Function apps should be separated by virtual network (VNet)/subnet and tagged appropriately.

You may also use Private Endpoints to perform network isolation. An Azure Private Endpoint is a network interface that connects you privately and securely to a service (for example: Azure Functions app HTTPs endpoint) powered by Azure Private Link. Private Endpoint uses a private IP address from your VNet, effectively bringing the service into your VNet. Private endpoints are in (Preview) for function apps running in the Premium plan. Ensure that Private Endpoints are no longer in (Preview) before using them with production workloads.

- [How to create additional Azure subscriptions](#)
- [How to create Management Groups](#)
- [How to create and use tags](#)

- [Azure Functions networking options](#)
- [Azure Functions Premium Plan](#)
- [Understand Private Endpoint](#)
- [Using Private Endpoints for Azure Functions](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.3: Monitor and block unauthorized transfer of sensitive information

Guidance: Deploy an automated tool on network perimeters that monitors for unauthorized transfer of sensitive information and blocks such transfers while alerting information security professionals.

Microsoft manages the underlying infrastructure for Azure Functions and has implemented strict controls to prevent the loss or exposure of customer data.

- [Understand customer data protection in Azure](#)

Azure Security Center monitoring: Currently not available

Responsibility: Not applicable

4.4: Encrypt all sensitive information in transit

Guidance: In the Azure portal for your Azure Function apps, under "Platform Features: Networking: SSL", enable the "HTTPs Only" setting and set the minimum TLS version to 1.2.

Azure Security Center monitoring: Yes

Responsibility: Customer

4.5: Use an active discovery tool to identify sensitive data

Guidance: Not currently available; data identification, classification, and loss prevention features are not currently available for Azure Functions. Tag Function apps that may be processing sensitive information as such and implement third-party solution if required for compliance purposes.

For the underlying platform which is managed by Microsoft, Microsoft treats all customer content as sensitive and goes to great lengths to guard against customer data loss and exposure. To ensure customer data within Azure remains secure, Microsoft has implemented and maintains a suite of robust data protection controls and capabilities.

- [Understand customer data protection in Azure](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

4.6: Use Azure RBAC to control access to resources

Guidance: Use Azure role-based access control (Azure RBAC) to control access to the Azure Function control plane (the Azure portal).

- [How to configure Azure RBAC](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.7: Use host-based data loss prevention to enforce access control

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Microsoft manages the underlying infrastructure for Azure Functions and has implemented strict controls to prevent the loss or exposure of customer data.

- [Understand customer data protection in Azure](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

4.8: Encrypt sensitive information at rest

Guidance: When creating a function app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. This is because Functions relies on Azure Storage for operations such as managing triggers and logging function executions. Azure Storage encrypts all data in a storage account at rest. By default, data is encrypted with Microsoft-managed keys. For additional control over encryption keys, you can supply customer-managed keys for encryption of blob and file data. These keys must be present in Azure Key Vault for the function app to be able to access the storage account.

- [Understand storage considerations for Azure Functions](#)
- [Understand Azure storage encryption for data at rest](#)

Azure Security Center monitoring: Not applicable

Responsibility: Shared

4.9: Log and alert on changes to critical Azure resources

Guidance: Use Azure Monitor with the Azure Activity log to create alerts for when changes take place to production Azure Function apps as well as other critical or related resources.

- [How to create alerts for Azure Activity Log events](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Vulnerability management

For more information, see [Security control: Vulnerability management](#).

5.1: Run automated vulnerability scanning tools

Guidance: Adopt a DevSecOps practice to ensure your Azure Functions applications are secure and remain as secure as possible throughout the duration of their life-cycle. DevSecOps incorporates your organization's security team and their capabilities into your DevOps practices making security a responsibility of everyone on the team.

In addition, follow recommendations from Azure Security Center to help secure your Azure Function apps.

- [How to add continuous security validation to your CI/CD pipeline](#)
- [How to implement Azure Security Center vulnerability assessment recommendations](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

5.2: Deploy an automated operating system patch management solution

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.3: Deploy automated third-party software patch management solution

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.4: Compare back-to-back vulnerability scans

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.5: Use a risk-rating process to prioritize the remediation of discovered vulnerabilities

Guidance: Microsoft performs vulnerability management on the underlying systems that support Azure Functions, however you may use the severity of the recommendations within Azure Security Center as well as the Secure Score to measure risk within your environment. Your Secure Score is based on how many Security Center recommendations you have mitigated. To prioritize the recommendations to resolve first, consider the severity of each.

- [Security recommendations reference guide](#)

Azure Security Center monitoring: Yes

Responsibility: Shared

Inventory and asset management

For more information, see [Security control: Inventory and asset management](#).

6.1: Use Azure Asset Discovery

Guidance: Use Azure Resource Graph to query/discover all resources (such as compute, storage, network, ports, and protocols etc.) within your subscription(s). Ensure appropriate (read) permissions in your tenant and enumerate all Azure subscriptions as well as resources within your subscriptions.

Although classic Azure resources may be discovered via Resource Graph, it is highly recommended to create and use Azure Resource Manager resources going forward.

- [How to create queries with Azure Resource Graph](#)
- [How to view your Azure Subscriptions](#)
- [Understand Azure RBAC](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.2: Maintain asset metadata

Guidance: Apply tags to Azure resources giving metadata to logically organize them into a taxonomy.

- [How to create and use tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.3: Delete unauthorized Azure resources

Guidance: Use tagging, management groups, and separate subscriptions, where appropriate, to organize and

track Azure resources. Reconcile inventory on a regular basis and ensure unauthorized resources are deleted from the subscription in a timely manner.

In addition, use Azure policy to put restrictions on the type of resources that can be created in customer subscription(s) using the following built-in policy definitions: Not allowed resource types Allowed resource types

- [How to create additional Azure subscriptions](#)
- [How to create Management Groups](#)
- [How to create and use Tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.4: Maintain an inventory of approved Azure resources and software titles

Guidance: Define approved Azure resources and approved software for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.5: Monitor for unapproved Azure resources

Guidance: Use Azure Policy to put restrictions on the type of resources that can be created in your subscription(s).

Use Azure Resource Graph to query/discover resources within their subscription(s). Ensure that all Azure resources present in the environment are approved.

- [How to configure and manage Azure Policy](#)
- [How to create queries with Azure Graph](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.6: Monitor for unapproved software applications within compute resources

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.7: Remove unapproved Azure resources and software applications

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.8: Use only approved applications

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.9: Use only approved Azure services

Guidance: Use Azure Policy to put restrictions on the type of resources that can be created in customer subscription(s) using the following built-in policy definitions: Not allowed resource types Allowed resource types

- [How to configure and manage Azure Policy](#)
- [How to deny a specific resource type with Azure Policy](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.10: Implement approved application list

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.11: Limit users' ability to interact with Azure Resources Manager via scripts

Guidance: Configure Azure Conditional Access to limit users' ability to interact with Azure Resource Manager by configuring "Block access" for the "Microsoft Azure Management" App.

- [How to configure Conditional Access to block access to Azure Resource Manager](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.12: Limit users' ability to execute scripts within compute resources

Guidance: Not applicable; this recommendation is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.13: Physically or logically segregate high risk applications

Guidance: For sensitive or high risk Azure Function apps, implement separate subscriptions and/or management groups to provide isolation.

Deploy high risk Azure Function apps into their own Virtual Network (VNet). Perimeter security in Azure Functions is achieved through VNets. Functions running in the Premium plan or App Service Environment (ASE) can be integrated with VNets. Choose the best architecture for your use case.

- [Azure Functions networking options](#)
- [Azure Functions Premium Plan](#)
- [Networking considerations for an App Service Environment](#)
- [How to create an external ASE](#)

How to create an internal ASE:

- <https://docs.microsoft.com/azure/app-service/environment/create-ilb-as>
- [How to create an NSG with a security config](#)

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Secure configuration

For more information, see [Security control: Secure configuration](#).

7.1: Establish secure configurations for all Azure resources

Guidance: Define and implement standard security configurations for your Azure Function app with Azure Policy. Use Azure Policy aliases in the "Microsoft.Web" namespace to create custom policies to audit or enforce the configuration of your Azure Functions apps. You may also make use of built-in policy definitions such as:

- Managed identity should be used in your Function App
- Remote debugging should be turned off for Function Apps
- Function App should only be accessible over HTTPS
- [How to view available Azure Policy Aliases](#)
- [How to configure and manage Azure Policy](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.2: Establish secure operating system configurations

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.3: Maintain secure Azure resource configurations

Guidance: Use Azure policy [deny] and [deploy if not exist] to enforce secure settings across your Azure resources.

- [How to configure and manage Azure Policy](#)
- [Understand Azure Policy Effects](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.4: Maintain secure operating system configurations

Guidance: Not applicable; while it is possible to deploy on-premises functions, this guideline is intended for IaaS compute resources. When deploying on premises functions, you are responsible for the secure configuration of your environment.

- [Understand on-premises functions](#)

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.5: Securely store configuration of Azure resources

Guidance: Store and manage ARM templates and custom Azure policy definitions securely in source control.

- [What is infrastructure as code](#)
- [Design policy as code workflows](#)
- [How to store code in Azure DevOps](#)
- [Azure Repos Documentation](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.6: Securely store custom operating system images

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.7: Deploy system configuration management tools

Guidance: Use built-in Azure Policy definitions as well as Azure Policy aliases in the "Microsoft.Web" namespace to create custom policies to alert, audit, and enforce system configurations. Additionally, develop a process and pipeline for managing policy exceptions.

- [How to configure and manage Azure Policy](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.8: Deploy system configuration management tools for operating systems

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.9: Implement automated configuration monitoring for Azure services

Guidance: Use built-in Azure Policy definitions as well as Azure Policy aliases in the "Microsoft.Web" namespace to create custom policies to alert, audit, and enforce system configurations. Use Azure policy [audit], [deny], and [deploy if not exist] to automatically enforce configurations for your Azure resources.

- [How to configure and manage Azure Policy](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.10: Implement automated configuration monitoring for operating systems

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.11: Manage Azure secrets securely

Guidance: Use Managed Identities in conjunction with Azure Key Vault to simplify and secure secret management for your cloud applications. Managed Identities allows your function app to authenticate to any service that supports Azure AD authentication, including Key Vault, without any credentials in your code.

- [How to create a Key Vault](#)
- [How to use managed identities for App Service and Azure Functions](#)
- [How to authenticate to Key Vault](#)
- [How to assign a Key Vault access policy](#)
- [Use Key Vault references for App Service and Azure Functions](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

7.12: Manage identities securely and automatically

Guidance: Use Managed Identities to provide your Azure Function app with an automatically managed identity in Azure AD. Managed Identities allows you to authenticate to any service that supports Azure AD authentication, including Key Vault, without any credentials in your code.

- [How to use managed identities for App Service and Azure Functions](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

7.13: Eliminate unintended credential exposure

Guidance: Implement Credential Scanner to identify credentials within code. Credential Scanner will also encourage moving discovered credentials to more secure locations such as Azure Key Vault.

- [How to setup Credential Scanner](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Malware defense

For more information, see [Security control: Malware defense](#).

8.1: Use centrally managed anti-malware software

Guidance: Not applicable; this guideline is intended for IaaS compute resources.

Microsoft anti-malware is enabled on the underlying host that supports Azure services (for example, Azure Functions), however it does not run on customer content.

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

8.2: Pre-scan files to be uploaded to non-compute Azure resources

Guidance: Not applicable; this recommendation is intended for non-compute resources designed to store data.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

8.3: Ensure anti-malware software and signatures are updated

Guidance: Not applicable; this recommendation is intended for non-compute resources designed to store data.

Microsoft anti-malware is enabled on the underlying host that supports Azure services (for example, Azure Functions), however it does not run on customer content.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Data recovery

For more information, see [Security control: Data recovery](#).

9.1: Ensure regular automated back ups

Guidance: Use the Backup and Restore feature to schedule regular backups of your app. Function apps running in the Premium plan have the same hosting capabilities as web apps in Azure App Service, which includes the

"Backup and Restore" feature.

Also make use of a source control solution such as Azure Repos and Azure DevOps to securely store and manage your code. Azure DevOps Services leverages many of the Azure storage features to ensure data availability in the case of hardware failure, service disruption, or region disaster. Additionally, the Azure DevOps team follows procedures to protect data from accidental or malicious deletion.

- [Back up your app in Azure](#)
- [Understand data availability in Azure DevOps](#)
- [How to store code in Azure DevOps](#)
- [Azure Repos Documentation](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

9.2: Perform complete system backups and backup any customer managed keys

Guidance: Use the Backup and Restore feature to schedule regular backups of your app. Function apps running in the Premium plan have the same hosting capabilities as web apps in Azure App Service, which includes the "Backup and Restore" feature. Backup customer managed keys within Azure Key Vault.

Also make use of a source control solution such as Azure Repos and Azure DevOps to securely store and manage your code. Azure DevOps Services leverages many of the Azure storage features to ensure data availability in the case of hardware failure, service disruption, or region disaster. Additionally, the Azure DevOps team follows procedures to protect data from accidental or malicious deletion.

- [Back up your app in Azure](#)
- [How to backup key vault keys in Azure](#)
- [Understand data availability in Azure DevOps](#)
- [How to store code in Azure DevOps](#)
- [Azure Repos Documentation](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

9.3: Validate all backups including customer managed keys

Guidance: Ensure ability to periodically perform restoration from the Backup and Restore feature. If using another offline location to backup your code, periodically ensure ability to perform complete restorations. Test restoration of backed up customer managed keys.

- [Restore an app in Azure from a backup](#)
- [Restore an app in Azure from a snapshot](#)
- [How to restore key vault keys in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

9.4: Ensure protection of backups and customer managed keys

Guidance: Backups from the Backup and Restore feature use an Azure Storage account in your subscription. Azure Storage encrypts all data in a storage account at rest. By default, data is encrypted with Microsoft-managed keys.

For additional control over encryption keys, you can supply customer-managed keys for encryption of storage data.

If you are using customer-managed-keys, ensure Soft-Delete in Key Vault is enabled to protect keys against accidental or malicious deletion.

- [Azure Storage encryption at rest](#)
- [How to enable Soft-Delete in Key Vault](#)

Azure Security Center monitoring: Yes

Responsibility: Shared

Incident response

For more information, see [Security control: Incident response](#).

10.1: Create an incident response guide

Guidance: Build out an incident response guide for your organization. Ensure that there are written incident response plans that define all roles of personnel as well as phases of incident handling/management from detection to post-incident review.

- [How to configure Workflow Automations within Azure Security Center](#)
- [Guidance on building your own security incident response process](#)
- [Microsoft Security Response Center's Anatomy of an Incident](#)
- [Customer may also leverage NIST's Computer Security Incident Handling Guide to aid in the creation of their own incident response plan](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.2: Create an incident scoring and prioritization procedure

Guidance: Security Center assigns a severity to each alert to help you prioritize which alerts should be investigated first. The severity is based on how confident Security Center is in the finding or the analytic used to issue the alert as well as the confidence level that there was malicious intent behind the activity that led to the alert.

Additionally, clearly mark subscriptions (for ex. production, non-prod) and create a naming system to clearly identify and categorize Azure resources.

Azure Security Center monitoring: Yes

Responsibility: Shared

10.3: Test security response procedures

Guidance: Conduct exercises to test your systems' incident response capabilities on a regular cadence. Identify weak points and gaps and revise plan as needed.

- [Refer to NIST's publication: Guide to Test, Training, and Exercise Programs for IT Plans and Capabilities](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.4: Provide security incident contact details and configure alert notifications for security incidents

Guidance: Security incident contact information will be used by Microsoft to contact you if the Microsoft Security Response Center (MSRC) discovers that the customer's data has been accessed by an unlawful or unauthorized party. Review incidents after the fact to ensure that issues are resolved.

- [How to set the Azure Security Center Security Contact](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

10.5: Incorporate security alerts into your incident response system

Guidance: Export your Azure Security Center alerts and recommendations using the Continuous Export feature. Continuous Export allows you to export alerts and recommendations either manually or in an ongoing, continuous fashion. You may use the Azure Security Center data connector to stream the alerts to Azure Sentinel.

- [How to configure continuous export](#)
- [How to stream alerts into Azure Sentinel](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.6: Automate the response to security alerts

Guidance: Use the Workflow Automation feature in Azure Security Center to automatically trigger responses to security alerts and recommendations with Logic Apps.

- [How to configure Workflow Automation and Logic Apps](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Penetration tests and red team exercises

For more information, see [Security control: Penetration tests and red team exercises](#).

11.1: Conduct regular penetration testing of your Azure resources and ensure remediation of all critical security findings

Guidance: Follow the Microsoft Rules of Engagement to ensure your Penetration Tests are not in violation of Microsoft policies. Use Microsoft's strategy and execution of Red Teaming and live site penetration testing against Microsoft-managed cloud infrastructure, services, and applications.

- [Penetration Testing Rules of Engagement](#)
- [Microsoft Cloud Red Teaming](#)

Azure Security Center monitoring: Not applicable

Responsibility: Shared

Next steps

- See the [Azure security benchmark](#)
- Learn more about [Azure security baselines](#)

Monitor Azure Functions

11/2/2020 • 7 minutes to read • [Edit Online](#)

Azure Functions offers built-in integration with [Azure Application Insights](#) to monitor functions. 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](#).

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 [Manage pricing and data volume in Application Insights](#).

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](#).

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 |
|-------------|------|--|
| Trace | 0 | Logs that contain the most detailed messages. These messages may 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#](#) and [JavaScript](#), you can use an Application Insights SDK to write custom telemetry data.

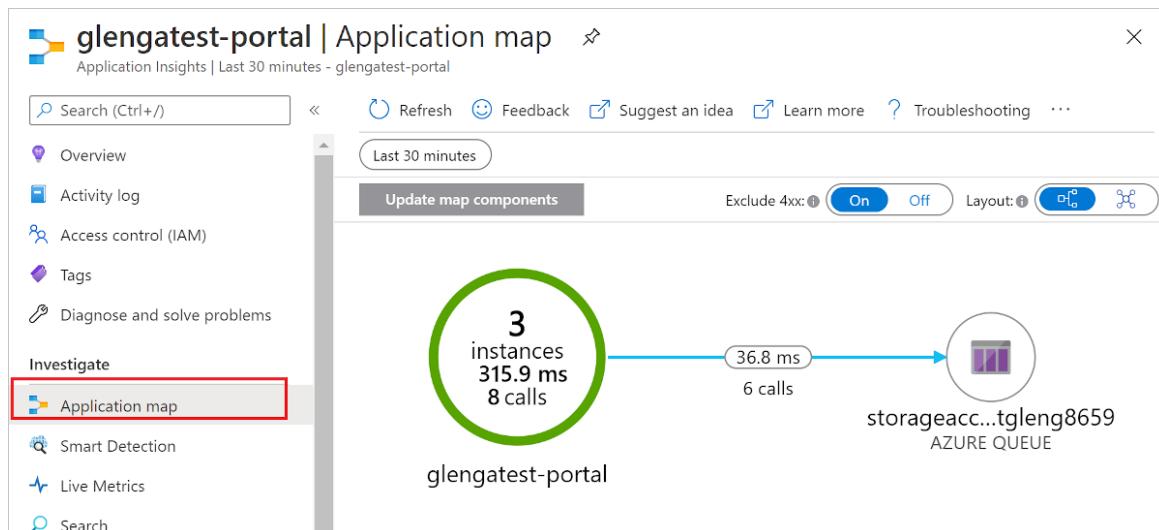
Dependencies

Starting with version 2.x of Functions, the runtime automatically collects data on dependencies for bindings that use certain client SDKs. 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](#)

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](#)

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

This feature is in preview.

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

This feature is in preview.

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](#).

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](#)

Azure Functions diagnostics overview

5/12/2020 • 2 minutes to read • [Edit Online](#)

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.

Home > myFunctionApp-dma | Diagnose and solve problems

myFunctionApp-dma | Diagnose and solve problems

App Service

Search (Ctrl+ /)

execution

5 Results

Function App Health Check
Functions Performance
Detects information about the latest execution.

Function Compilation Error (.csx)
Functions Performance
Detects information about compilation errors in the latest execution.

Function Execution Performance
Availability and Performance
Analyze performance issues for your functions.

Function Executions and Errors
Functions Performance
Detects execution statistics and errors for every function inside the function app.

Timer Trigger Issue Analysis
Availability and Performance
This will help check timer trigger configurations and analyze execution failures.

Azure Functions Diagnostics

Use Azure Functions Diagnostics to investigate how your function app is performing, diagnose issues, and discover how to improve your function app. Select the problem category that best matches the information or tool that you're interested in:

Availability and Performance

Is your Function App performing slower than normal? Investigate performance issues or just check the health of your Function App.

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' tab selected in the top navigation bar. A message box says, 'Hello! Welcome to Azure Functions Diagnostics! My name is Genie and I'm here to help you diagnose and solve problems.' Below it, another message box says, '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 ten blue tiles represents different checks:

- 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 'Function App Down or Reporting Errors' topic list. A button on the right says, 'I am interested in Function App Down or Reporting Errors'. The topics listed are:

- Bad Async Function Pattern
- Check RunFromPackage Logs
- Function App General Information
- Function App Offline History
- Function App Settings Check

Each topic has a 'View Full Report' link to its right.

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 plan costs

11/2/2020 • 8 minutes to read • [Edit Online](#)

There are currently three types of hosting plans for an app that runs in Azure Functions, with each plan having its own pricing model:

| 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. |
| Premium | Provides you with the same features and scaling mechanism as the Consumption plan, but with enhanced performance and VNET access. 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. |

You chose the plan that best supports your function performance and cost requirements. To learn more, see [Azure Functions scale and hosting](#).

This article deals only with the Consumption plan, since this plan results in variable costs. This article supersedes the [Consumption plan cost billing FAQ](#) article.

Durable Functions can also run in a Consumption plan. To learn more about the cost considerations when using Durable Functions, see [Durable Functions billing](#).

Consumption plan costs

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 prior to a function processing the request. A 5xx response generated by the platform after your function code has started to execute is still counted as an execution, even if the error isn't raised by 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 calculating pricing for function apps, any triggers and bindings you have that integrate with other Azure services require you to create and pay for those additional services.

For functions running in a Consumption plan, the total cost is the execution cost of your functions, plus the cost of bandwidth and additional services.

When estimating the overall costs of your function app and related services, use the [Azure pricing calculator](#).

| 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 don't pay for data transfer between Azure services in the same region. However, you can incur costs for outbound data transfers to another region or outside of Azure. To learn more, see Bandwidth pricing details . |

Behaviors affecting execution time

The following behaviors of your functions can impact 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.

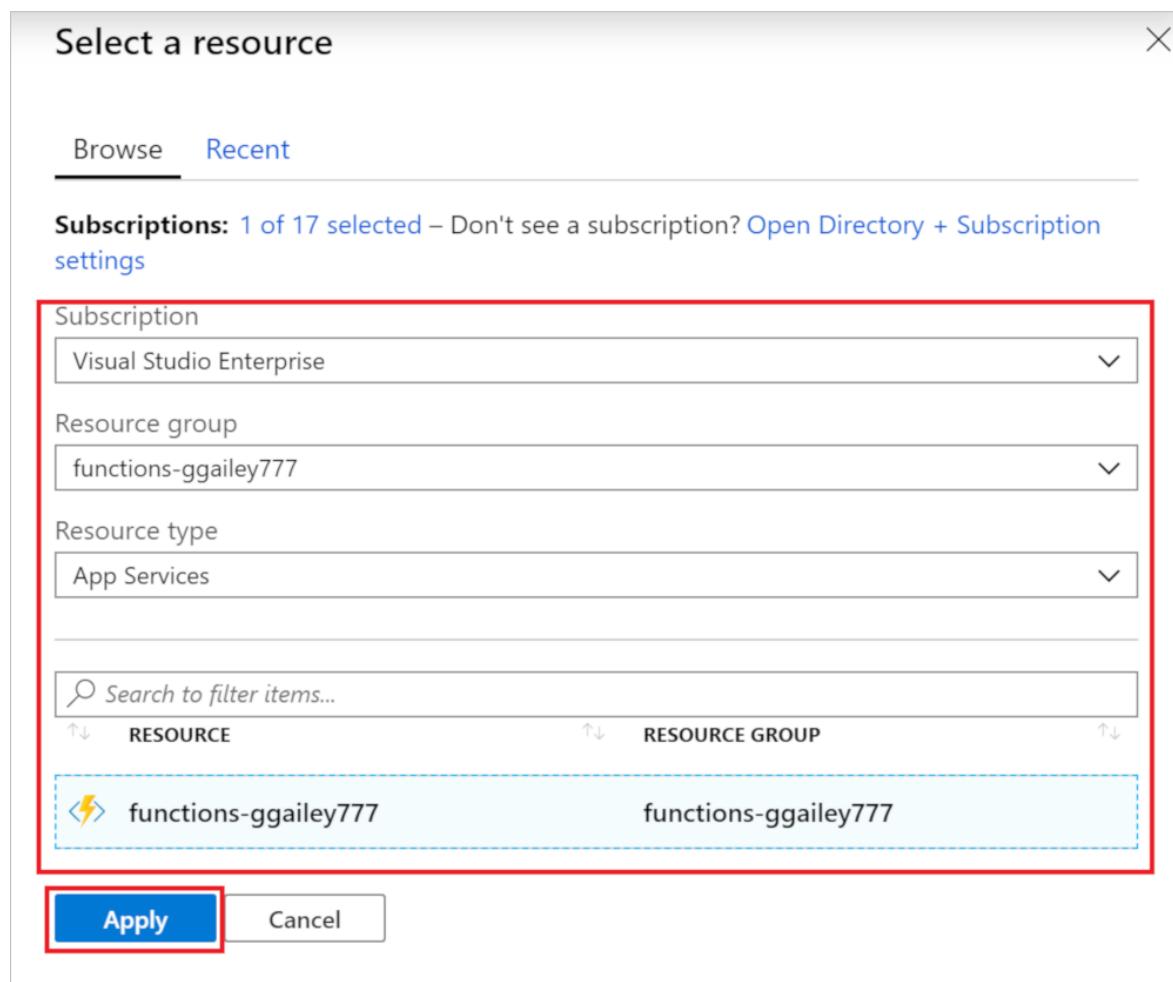
Function app-level metrics

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. You can use either [Azure Monitor metrics explorer](#) in the [Azure portal](#) or REST APIs to get this data.

Monitor metrics explorer

Use [Azure Monitor metrics explorer](#) to view cost-related data for your Consumption plan function apps in a graphical format.

1. At the top of the [Azure portal](#) in **Search services, resources, and docs** search for `monitor` and select **Monitor** under Services.
2. At the left, select **Metrics > Select a resource**, then use the settings below the image to choose your function app.



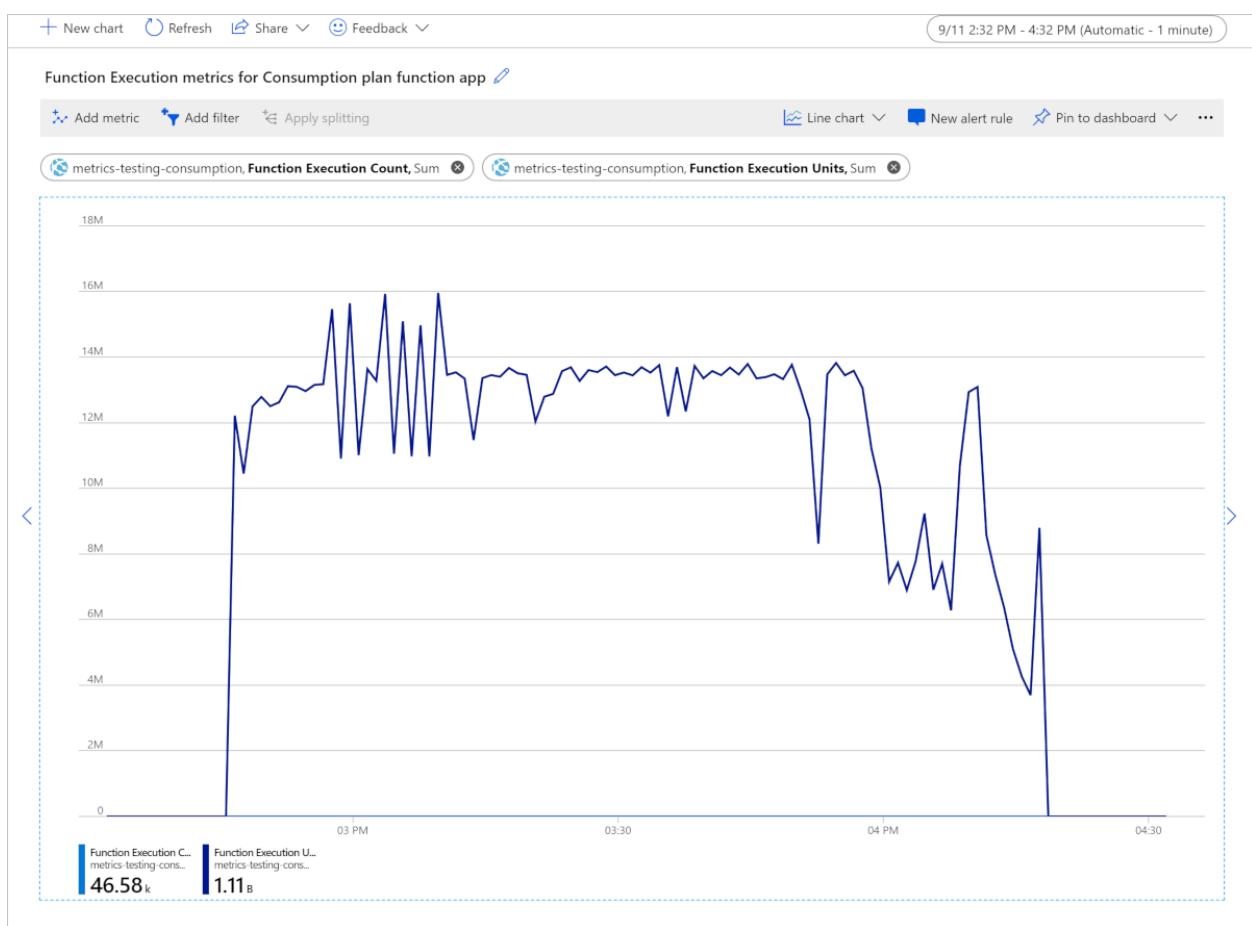
| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------|---------------------|---|
| Subscription | Your subscription | The subscription with your function app. |
| Resource group | Your resource group | The resource group that contains your function app. |
| Resource type | App Services | Function apps are shown as App Services instances in Monitor. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------|-------------------|------------------------------|
| Resource | Your function app | The function app to monitor. |

3. Select **Apply** to choose your function app as the resource to monitor.
4. From **Metric**, choose **Function Execution Count** and **Sum** for **Aggregation**. This adds the sum of the execution counts during chosen period to the chart.

5. 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.

The [Azure CLI](#) has commands for retrieving metrics. You can use the CLI from a local command environment or directly from the portal using [Azure Cloud Shell](#). For example, the following `az monitor metrics list` command returns hourly data over same time period used before.

Make sure to replace `<AZURE_SUBSCRIPTION_ID>` with your Azure subscription ID running the command.

```
az monitor metrics list --resource /subscriptions/<AZURE_SUBSCRIPTION_ID>/resourceGroups/metrics-testing-consumption/providers/Microsoft.Web/sites/metrics-testing-consumption --metric FunctionExecutionUnits,FunctionExecutionCount --aggregation Total --interval PT1H --start-time 2019-09-11T21:46:00Z --end-time 2019-09-11T23:18:00Z
```

This command returns a JSON payload that looks like the following example:

```
{
  "cost": 0.0,
  "interval": "1:00:00",
  "namespace": "Microsoft.Web/sites",
  "resourceregion": "centralus",
  "timespan": "2019-09-11T21:46:00Z/2019-09-11T23:18:00Z",
  "value": [
    {
      "id": "/subscriptions/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX/resourceGroups/metrics-testing-consumption/providers/Microsoft.Web/sites/metrics-testing-consumption/providers/Microsoft.Insights/metrics/FunctionExecutionUnits",
      "name": {
        "localizedValue": "Function Execution Units",
        "value": "FunctionExecutionUnits"
      },
      "resourceGroup": "metrics-testing-consumption",
      "timeseries": [
        {
          "data": [
            {
              "average": null,
              "count": null,
              "maximum": null,
              "minimum": null,
              "timeStamp": "2019-09-11T21:46:00+00:00",
              "total": 793294592.0
            },
            {
              "average": null,
              "count": null,
              "maximum": null,
              "minimum": null,
              "timeStamp": "2019-09-11T22:46:00+00:00",
              "total": 316576256.0
            }
          ],
          "metadatavalues": []
        }
      ],
      "type": "Microsoft.Insights/metrics",
      "unit": "Count"
    },
    {
      "id": "/subscriptions/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX/resourceGroups/metrics-testing-consumption/providers/Microsoft.Web/sites/metrics-testing-consumption/providers/Microsoft.Insights/metrics/FunctionExecutionCount",
      "name": {
        "localizedValue": "Function Execution Count",
        "value": "FunctionExecutionCount"
      },
      "resourceGroup": "metrics-testing-consumption",
      "timeseries": [
        {
          "data": [
            {
              "average": null,
              "count": null,
              "maximum": null,
              "minimum": null,
              "timeStamp": "2019-09-11T21:46:00+00:00",
              "total": 793294592.0
            },
            {
              "average": null,
              "count": null,
              "maximum": null,
              "minimum": null,
              "timeStamp": "2019-09-11T22:46:00+00:00",
              "total": 316576256.0
            }
          ],
          "metadatavalues": []
        }
      ],
      "type": "Microsoft.Insights/metrics",
      "unit": "Count"
    }
  ]
}
```

```

{
  "data": [
    {
      "average": null,
      "count": null,
      "maximum": null,
      "minimum": null,
      "timeStamp": "2019-09-11T21:46:00+00:00",
      "total": 33538.0
    },
    {
      "average": null,
      "count": null,
      "maximum": null,
      "minimum": null,
      "timeStamp": "2019-09-11T22:46:00+00:00",
      "total": 13040.0
    }
  ],
  "metadatavalues": []
}
],
"type": "Microsoft.Insights/metrics",
"unit": "Count"
]
}

```

This particular response shows that from `2019-09-11T21:46` to `2019-09-11T23:18`, the app consumed 1110000000 MB-milliseconds (1083.98 GB-seconds).

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](#).

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 |

| TIMESTAMP [UTC] | NAME | VALUE |
|---------------------------|---------------|-------------|
| 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 function apps](#)

Optimize the performance and reliability of Azure Functions

2/19/2020 • 6 minutes to read • [Edit Online](#)

This article provides guidance to improve the performance and reliability of your [serverless](#) function apps.

General 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.

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 1 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.

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](#).

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 two 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](#)

Storage considerations for Azure Functions

11/2/2020 • 4 minutes to read • [Edit Online](#)

Azure Functions requires an Azure Storage account when you create a function app instance. The following storage services may be used by your function app:

| STORAGE SERVICE | FUNCTIONS USAGE |
|---------------------|--|
| Azure Blob storage | Maintain bindings state and function keys. Also used by task hubs in Durable Functions . |
| Azure Files | File share used to store and run your function app code in a Consumption Plan and Premium Plan . |
| Azure Queue storage | Used by task hubs in Durable Functions . |
| Azure Table storage | Used by task hubs in Durable Functions . |

IMPORTANT

When using the Consumption/Premium hosting plan, your function code and binding configuration files are stored in Azure File storage in the main storage account. When you delete the main storage account, this content is deleted and cannot be recovered.

Storage account requirements

When creating a function app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. This is because Functions relies on Azure Storage for operations such as managing triggers and logging function executions. Some storage accounts don't support queues and tables. These accounts include blob-only storage accounts, Azure Premium Storage, and general-purpose storage accounts with ZRS replication. These unsupported accounts are filtered out of from the Storage Account blade when creating a function app.

To learn more about storage account types, see [Introducing the Azure Storage Services](#).

While you can use an existing storage account with your function app, you must make sure that it meets these requirements. Storage accounts created as part of the function app create flow are guaranteed to meet these storage account requirements.

Storage account guidance

Every function app requires a storage account to operate. If that account is deleted your function app won't run. To troubleshoot storage-related issues, see [How to troubleshoot storage-related issues](#). The following additional considerations apply to the Storage account used by function apps.

Storage account connection setting

The storage account connection is maintained in the [AzureWebJobsStorage application setting](#).

The 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 Azure 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.

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.

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](#).

Mount file shares (Linux)

You can mount existing Azure Files shares to your Linux function apps. By mounting a share to your Linux function app, you can leverage existing machine learning models or other data in your functions. You can use the `az webapp config storage-account add` command to mount an existing share to your Linux function app.

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 a 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 may increase the cold start time by at least 200-300ms, 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:

```
import os
...
files_in_share = os.listdir("/path/to/mount")
```

Next steps

Learn more about Azure Functions hosting options.

[Azure Functions scale and hosting](#)

Work with Azure Functions Proxies

12/4/2020 • 9 minutes to read • [Edit Online](#)

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.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: [C#, JavaScript, Java, or Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#, C# script, F#, Java, JavaScript, or Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

NOTE

Standard Functions billing applies to proxy executions. For more information, see [Azure Functions pricing](#).

Create a proxy

This section shows you how to create a proxy in the Functions portal.

1. Open the [Azure portal](#), and then go to your function app.
2. In the left pane, select **New proxy**.
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 does not need to be static, and it can reference [application settings](#) and [parameters from the original client request](#).
6. Click **Create**.

Your proxy now exists as a new endpoint on your function app. From a client perspective, it is equivalent to an `HttpTrigger` in Azure 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

With Azure Functions 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 requests can be modified in the portal by expanding the *response override* section of the proxy detail page.

Use variables

The configuration for a proxy does not 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 is not 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 is not 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 is not 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 will 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 will not be able to access the trace contents without your login credentials, but generating the trace consumes resources and exposes that you are 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:

```
{
    "$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 is not 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 is 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 is not 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.

```
{
    "$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 are 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. Note that 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. Note that 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:

```
{
  "$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 is not 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:

```
{  
    "$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.

Azure Functions networking options

12/4/2020 • 24 minutes to read • [Edit Online](#)

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.

You can host function apps in a couple of ways:

- You can choose from plan options that run on a multitenant infrastructure, with various levels of virtual network connectivity and scaling options:
 - The [Consumption plan](#) scales dynamically in response to load and offers minimal network isolation options.
 - The [Premium plan](#) also scales dynamically and offers more comprehensive network isolation.
 - The Azure [App Service plan](#) operates at a fixed scale and offers network isolation similar to the Premium plan.
- You can run functions in an [App Service Environment](#). This method deploys your function into your virtual network and offers full network control and isolation.

Matrix of networking features

| FEATURE | CONSUMPTION PLAN | PREMIUM PLAN | DEDICATED PLAN | ASE | KUBERNETES |
|---|------------------|------------------|------------------------------|-------|------------|
| Inbound IP restrictions and private site access | ✓ Yes | ✓ Yes | ✓ Yes | ✓ Yes | ✓ Yes |
| Virtual network integration | ✗ No | ✓ Yes (Regional) | ✓ Yes (Regional and Gateway) | ✓ Yes | ✓ Yes |
| Virtual network triggers (non-HTTP) | ✗ No | ✓ Yes | ✓ Yes | ✓ Yes | ✓ Yes |
| Hybrid connections (Windows only) | ✗ No | ✓ Yes | ✓ Yes | ✓ Yes | ✓ Yes |
| Outbound IP restrictions | ✗ No | ✓ Yes | ✓ Yes | ✓ Yes | ✓ Yes |

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 [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](#).

Use service endpoints

By using service endpoints, you can restrict access to selected Azure virtual network subnets. 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'll be 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 will be configured for service endpoints in anticipation of having them enabled later on the subnet.

Add IP Restriction

Name (Required)

Action
 Allow Deny

* Priority

Description

Type

* Subscription

* Virtual Network

* Subnet

 Selected subnet 'networking-demos-vnet/vnet-integration-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

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](#).

Private endpoint connections

[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.

When creating an inbound private endpoint connection for functions, you will also need a DNS record to resolve the private address. By default a private DNS record will be created for you when creating a private endpoint using the Azure portal.

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](#).

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 multitenant systems that support the full range of pricing plans except Isolated.
- The App Service Environment, which deploys into your VNet and supports Isolated pricing plan apps.

The VNet Integration feature is used in multitenant apps. If your app is in [App Service Environment](#), then it's already in a VNet and doesn't require use of the VNet Integration feature to reach resources in the same VNet. For more information on all of the networking features, see [App Service networking features](#).

VNet Integration gives your app access to resources in your VNet, but it doesn't grant inbound private access to your app from the VNet. Private site access refers to making an app accessible only from a private network, such as from within an Azure virtual network. VNet Integration is used only to make outbound calls from your app into your VNet. The VNet Integration feature behaves differently when it's used with VNet in the same region and with VNet in other regions. The VNet Integration feature has two variations:

- **Regional VNet Integration:** When you connect to Azure Resource Manager virtual networks in the same region, you must have a dedicated subnet in the VNet you're integrating with.
- **Gateway-required VNet Integration:** When you connect to VNet in other regions or to a classic virtual network in the same region, you need an Azure Virtual Network gateway provisioned in the target VNet.

The VNet Integration features:

- Require a Standard, Premium, PremiumV2, PremiumV3, or Elastic Premium pricing plan.
- Support TCP and UDP.
- Work with Azure App Service apps and function apps.

There are some things that VNet Integration doesn't support, like:

- Mounting a drive.
- Active Directory integration.
- NetBIOS.

Gateway-required VNet Integration provides access to resources only in the target VNet or in networks connected to the target VNet with peering or VPNs. Gateway-required VNet Integration doesn't enable access to resources available across Azure ExpressRoute connections or works with service endpoints.

Regardless of the version used, VNet Integration gives your app access to resources in your VNet, but it doesn't

grant inbound private access to your app from the VNet. Private site access refers to making your app accessible only from a private network, such as from within an Azure VNet. VNet Integration is only for making outbound calls from your app into your VNet.

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 [Integrate a function app with an Azure virtual network](#).

Regional virtual network integration

Using regional VNet Integration enables your app to access:

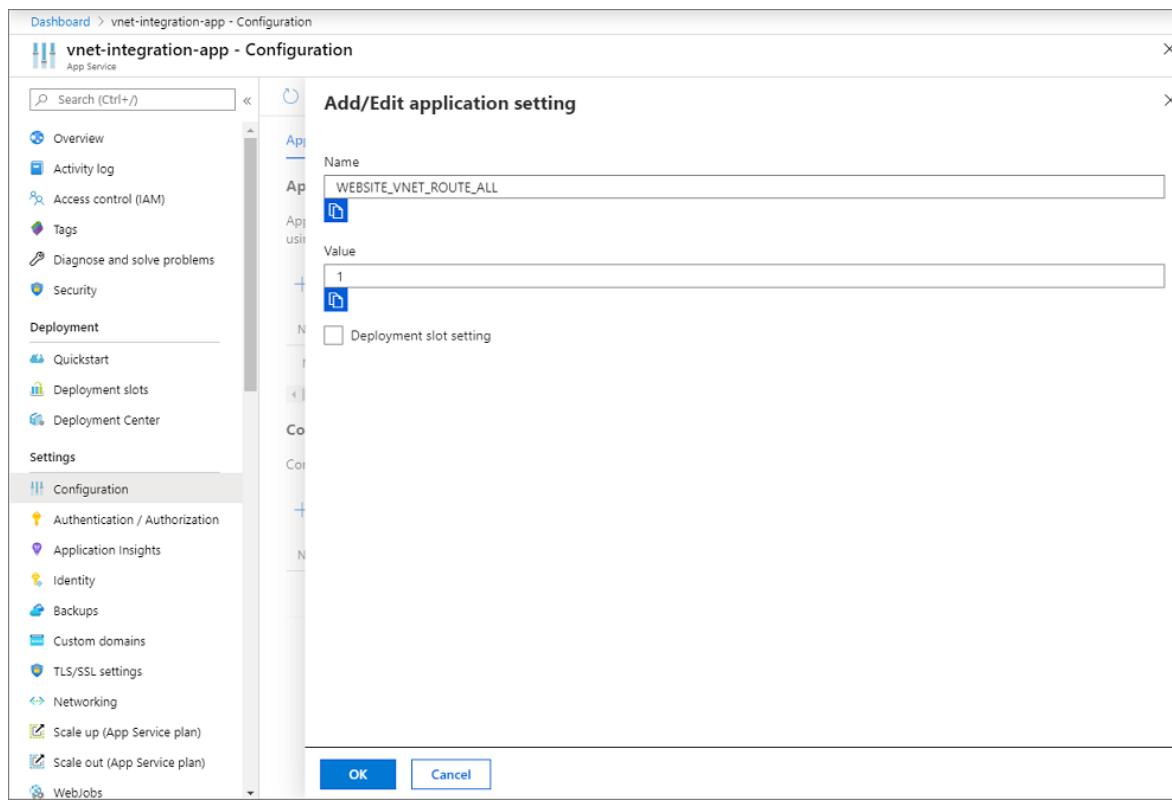
- Resources in a VNet in the same region as your app.
- Resources in VNets peered to the VNet your app is integrated with.
- Service endpoint secured services.
- Resources across Azure ExpressRoute connections.
- Resources in the VNet you're integrated with.
- Resources across peered connections, which include Azure ExpressRoute connections.
- Private endpoints

When you use VNet Integration with VNets in the same region, 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 VNet 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.

By default, your app routes only RFC1918 traffic into your VNet. If you want to route all of your outbound traffic into your VNet, apply the app setting WEBSITE_VNET_ROUTE_ALL to your app. To configure the app setting:

1. Go to the **Configuration** UI in your app portal. Select **New application setting**.
2. Enter **WEBSITE_VNET_ROUTE_ALL** in the **Name** box, and enter **1** in the **Value** box.



3. Select **OK**.

4. Select **Save**.

NOTE

If you route all of your outbound traffic into your VNet, it's subject to the NSGs and UDRs that are applied to your integration subnet. When you route all of your outbound traffic into your VNet, your outbound addresses are still the outbound addresses that are listed in your app properties unless you provide routes to send the traffic elsewhere.

There are some limitations with using VNet Integration with VNets in the same region:

- You can't reach resources across global peering connections.
- The feature is available from all App Service scale units in Premium V2 and Premium V3. It is also available in Standard but only from newer App Service scale units. If you are on an older scale unit you can only use the feature from a Premium V2 App Service plan. If you want to be certain of being able to 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 scale units. 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 VNet.
- The app and the VNet must be in the same region.
- You can't delete a VNet with an integrated app. Remove the integration before you delete the VNet.
- You can only integrate with VNets in the same subscription as the app.
- You can have only one regional VNet Integration per App Service plan. Multiple apps in the same App Service plan can use the same VNet.
- You can't change the subscription of an app or a plan while there's an app that's using regional VNet Integration.
- Your app cannot resolve addresses in Azure DNS Private Zones without configuration changes

VNet Integration depends on use of a dedicated subnet. When you provision a subnet, the Azure subnet loses 5

IPs from the start. One address is used from the integration subnet for each plan instance. If you scale your app to four instances, then four addresses are used. The debit of 5 addresses from the subnet size mean that the maximum available addresses per CIDR block are:

- /28 has 11 addresses
- /27 has 27 address
- /26 has 59 addresses

If you scale up or down in size, you need double your address need for a short period of time. The limits in size means that the real available supported instances per subnet size are, if your subnet is a:

- /28, your maximum horizontal scale is 5 instances
- /27, your maximum horizontal scale is 13 instances
- /26, your maximum horizontal scale is 29 instances

The limits noted on maximum horizontal scale assumes that you will 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, a /26 with 64 addresses is the recommended size.

If you want your apps in another plan to reach a VNet that's already connected to by apps in another plan, select a different subnet than the one being used by the preexisting VNet 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.

Service endpoints

Regional VNet Integration enables you to use service endpoints. To use service endpoints with your app, use regional VNet Integration to connect to a selected VNet and then configure service endpoints with the destination service on the subnet you used for the integration. If you then wanted to access a service over service endpoints:

1. configure regional VNet Integration with your web app
2. go to the destination service and configure service endpoints against the subnet used for integration

Network security groups

You can use network security groups to block inbound and outbound traffic to resources in a VNet. An app that uses regional VNet Integration can use a [network security group](#) to block outbound traffic to resources in your VNet or the internet. To block traffic to public addresses, you must have the application setting WEBSITE_VNET_ROUTE_ALL set to 1. The inbound rules in an NSG don't apply to your app because VNet 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 WEBSITE_VNET_ROUTE_ALL is set to 1 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. If WEBSITE_VNET_ROUTE_ALL isn't set, 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. If you set WEBSITE_VNET_ROUTE_ALL to 1, all of your outbound calls are affected. 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 will be affected. By default, BGP routes affect only your RFC1918 destination traffic. If WEBSITE_VNET_ROUTE_ALL is set to 1, all outbound traffic can be affected by your BGP routes.

Azure DNS Private Zones

After your app integrates with your VNet, it uses the same DNS server that your VNet is configured with. You can override this behavior on your app by configuring the app setting WEBSITE_DNS_SERVER with the address of your desired DNS server. If you had a custom DNS server configured with your VNet but wanted to have your app use Azure DNS private zones, you should set WEBSITE_DNS_SERVER with value 168.63.129.16.

Private endpoints

If you want to make calls to [Private Endpoints](#), then you need to ensure that your DNS lookups will resolve to the private endpoint. To ensure that the DNS lookups from your app will point to your private endpoints you can:

- integrate with Azure DNS Private Zones. If your VNet doesn't have a custom DNS server, this will be automatic
- manage the private endpoint in the DNS server used by your app. To do this you need to know the private endpoint address and then point the endpoint you are trying to reach to that address with an A record.
- configure your own DNS server to forward to Azure DNS private zones

Connect to service endpoint secured resources

To provide a higher level of security, you can restrict a number of Azure services to a virtual network by using service endpoints. You must then integrate your function app with that virtual network to access the resource. This configuration is supported on all plans that support virtual network integration.

To learn more, see [Virtual network service endpoints](#).

Restrict your storage account to a virtual network (preview)

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 endpoint. This preview feature currently only works with Windows Premium plans in West Europe. To set up a function with a storage account restricted to a private network:

NOTE

Restricting the storage account only currently works for Premium functions using Windows in West Europe

1. Create a function with a storage account that does not have service endpoints enabled.
2. Configure the function to connect to your virtual network.
3. Create or configure a different storage account. This will be the storage account we secure with service endpoints and connect our function.
4. [Create a file share](#) in the secured storage account.
5. Enable service endpoints or private endpoint for the storage account.
 - If using private endpoint connections, the storage account will need a private endpoint for the `file` and `blob` subresources. If using certain capabilities like Durable Functions, you will also need `queue` and `table` accessible through a private endpoint connection.

- If using service endpoints, enable the subnet dedicated to your function apps for storage accounts.
6. (Optional) Copy the file and blob content from the function app storage account to the secured storage account and file share.
 7. Copy the connection string for this storage account.
 8. Update the **Application Settings** under **Configuration** for the function app to the following:
 - `AzureWebJobsStorage` to the connection string for the secured storage account.
 - `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` to the connection string for the secured storage account.
 - `WEBSITE_CONTENTSHARE` to the name of the file share created in the secured storage account.
 - Create a new setting with the name `WEBSITE_CONTENTOVERVNET` and value of `1`.
 - If the storage account is using private endpoint connections, verify or add the following settings
 - `WEBSITE_VNET_ROUTE_ALL` with a value of `1`.
 - `WEBSITE_DNS_SERVER` with a value of `168.63.129.16`
 9. Save the application settings.

The function app will restart and will now be connected to a secured storage account.

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.

Currently, [Key Vault references](#) won't work if your key vault is secured with service endpoints. To connect to a key vault by using virtual network integration, you need to call Key Vault in your application code.

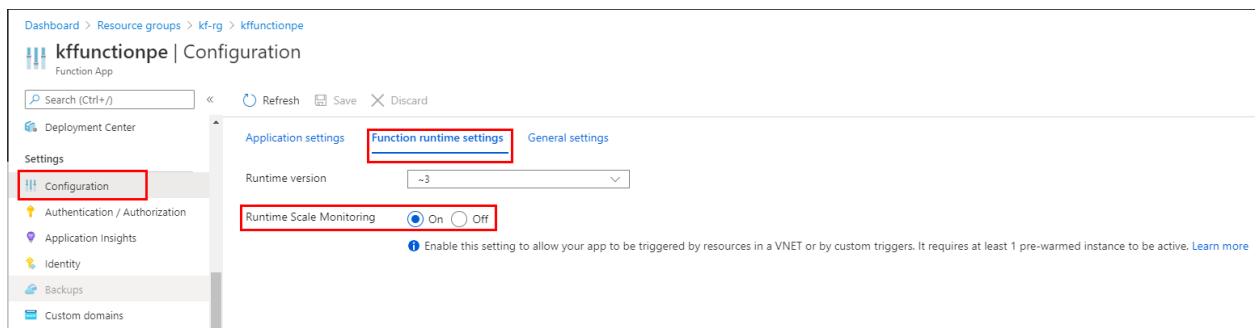
Virtual network triggers (non-HTTP)

Currently, you can use non-HTTP trigger functions from within a virtual network in one of two ways:

- Run your function app in a Premium plan and enable virtual network trigger support.
- Run your function app in an App Service plan or App Service Environment.

Premium plan with virtual network triggers

When you run a Premium plan, you can connect non-HTTP trigger functions to services that run inside a virtual network. To do this, you must enable virtual network trigger support for your function app. The **Runtime Scale Monitoring** setting is found in the [Azure portal](#) under **Configuration > Function runtime settings**.



You can also enable virtual network triggers by using the following Azure CLI command:

```
az resource update -g <resource_group> -n <function_app_name>/config/web --set properties.functionsRuntimeScaleMonitoringEnabled=1 --resource-type Microsoft.Web/sites
```

TIP

Enabling virtual network triggers may have an impact on the performance of your application since your App Service plan instances will need to monitor your triggers to determine when to scale. This impact is likely to be very small.

Virtual network triggers are supported in version 2.x and above of the Functions runtime. The following non-HTTP trigger types are supported.

| EXTENSION | MINIMUM VERSION |
|--|-----------------|
| Microsoft.Azure.WebJobs.Extensions.Storage | 3.0.10 or above |
| Microsoft.Azure.WebJobs.Extensions.EventHubs | 4.1.0 or above |
| Microsoft.Azure.WebJobs.Extensions.ServiceBus | 3.2.0 or above |
| Microsoft.Azure.WebJobs.Extensions.CosmosDB | 3.0.5 or above |
| Microsoft.Azure.WebJobs.Extensions.DurableTask | 2.0.0 or above |

IMPORTANT

When you enable virtual network trigger support, only the trigger types shown in the previous table scale dynamically with your application. You can still use triggers that aren't in the table, but they're not scaled beyond their pre-warmed instance count. For the complete list of triggers, 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 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 a Premium plan or an App Service plan with a virtual network, the app can still make outbound calls to the internet by default. By adding the application setting `WEBSITE_VNET_ROUTE_ALL=1`, you force all outbound traffic to be sent into your virtual network, where network security group rules can be used to restrict traffic.

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](#).

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:

```
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

`nameresolver.exe` 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:

```
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 VNet Integration

- Is your destination a non-RFC1918 address and you don't have `WEBSITE_VNET_ROUTE_ALL` set to 1?
- 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 VNet 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 VNet 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 VNet 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.

Additional 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:

```
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 VNet Integration feature.

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](#)

IP addresses in Azure Functions

11/2/2020 • 4 minutes to read • [Edit Online](#)

This article explains the following topics related to IP addresses of function apps:

- How to find the IP addresses currently in use by a function app.
- What causes a function app's IP addresses to be changed.
- How to restrict the IP addresses that can access a function app.
- How to get 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 has a single inbound IP address. To find that IP address:

1. Sign in to the [Azure portal](#).
2. Navigate to the function app.
3. Select **Platform features**.
4. Select **Properties**, and the inbound IP address appears under **Virtual IP address**.

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.

To find the outbound IP addresses available to a function app:

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](#).

An alternative way to find the available outbound IP addresses is by using the [Cloud Shell](#):

```
az webapp show --resource-group <group_name> --name <app_name> --query outboundIpAddresses --output tsv  
az webapp show --resource-group <group_name> --name <app_name> --query possibleOutboundIpAddresses --output tsv
```

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 may need to add the entire data center to an allow list.

Data center outbound IP addresses

If you need to add the outbound IP addresses used by your function apps to an allow list, another option is to add the function apps' data center (Azure region) to an allow list. 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, this is what the Western Europe JSON fragment might look like:

```
{  
  "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 re-create 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 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 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](#).

When your function app runs in a [Consumption plan](#) or in a [Premium plan](#), the outbound IP address might also change even when you haven't taken any actions such as the ones [listed above](#).

To deliberately force an outbound IP address change:

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

If you need static, dedicated IP addresses, 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:

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**.

As an alternative, you can use the [Cloud Shell](#):

```
az webapp show --resource-group <group_name> --name <app_name> --query sku --output tsv
```

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 (preview)

11/2/2020 • 12 minutes to read • [Edit Online](#)

Every Functions app is executed by a language-specific handler. While Azure Functions supports 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.

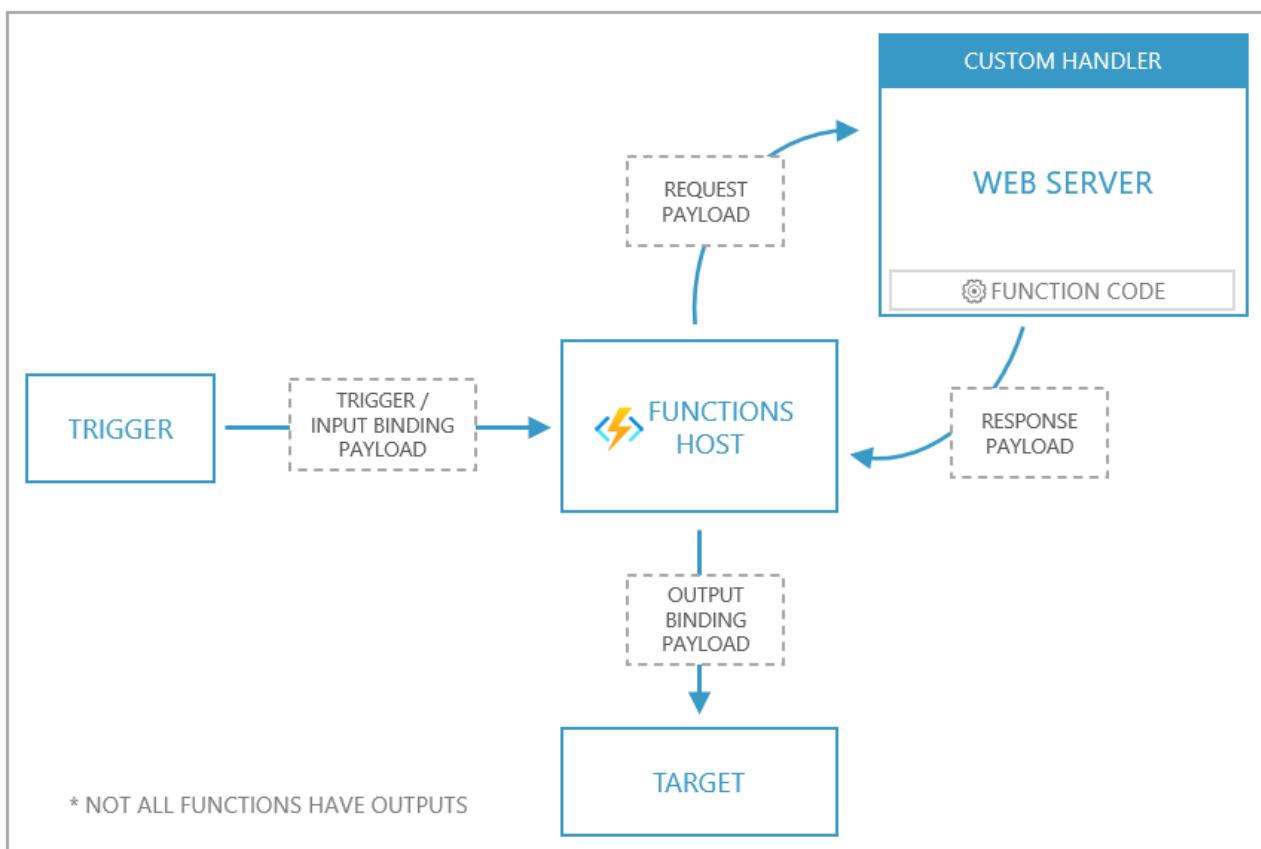
Custom handlers are best suited for situations where you want to:

- Implement a function app in a language that's not currently supported, such as Go and Rust.
- Implement a function app in a runtime that's not currently supported, such as Deno.

With custom handlers, you can use [triggers and input and output bindings](#) via extension bundles.

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 an custom handler executable named *handler.exe*.

```
| /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.

```
{  
  "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`.

```
{
  "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`.

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "Custom"
  }
}
```

NOTE

`Custom` may not be recognized as a valid runtime on the Linux Premium or App Service plans. If that is your deployment target, set `FUNCTIONS_WORKER_RUNTIME` to an empty string.

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

```
{
  "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](#).

```
{
  "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",
    "NextVisibleTime": "2019-10-09T18:08:32+00:00",
    "PopReceipt": "AgAAAAMAAAAAAgtnj8x+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 |
|-------------|-----------|---------|
| | | |

| 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.

```
{
  "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`

```
{  
  "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).

```
{  
  "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.

```
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:

```
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.

```

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.

```
{  
  "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

```
{  
  "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`.

```
{  
  "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.

```
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.

```

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. A "Custom" stack for custom handlers will be added in the future.

To deploy a custom handler app using Azure Functions Core Tools, run the following command.

```
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`.

```
{
  "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.

Azure Functions developer guide

12/4/2020 • 6 minutes to read • [Edit Online](#)

In Azure Functions, specific functions share a few core technical concepts and components, regardless of the language or binding you use. Before you jump into learning details specific to a given language or binding, be sure to read through this overview that applies to all of them.

This article assumes that you've already read the [Azure Functions overview](#).

Function code

A *function* is the primary concept in Azure Functions. A function contains two important pieces - your code, which can be written in a variety of languages, and some config, the `function.json` file. For compiled languages, this config file is generated automatically from annotations in your code. For scripting languages, you must provide the config file yourself.

The `function.json` file defines the function's trigger, bindings, and other configuration settings. Every function has one and only one trigger. The runtime uses this config file to determine the events to monitor and how to pass data into and return data from a function execution. The following is an example `function.json` file.

```
{  
    "disabled":false,  
    "bindings": [  
        // ... bindings here  
        {  
            "type": "bindingType",  
            "direction": "in",  
            "name": "myParamName",  
            // ... more depending on binding  
        }  
    ]  
}
```

For more information, see [Azure Functions triggers and bindings concepts](#).

The `bindings` property is where you configure both triggers and bindings. Each binding shares a few common settings and some settings which are specific to a particular type of binding. Every binding requires the following settings:

| PROPERTY | VALUES/TYPES | COMMENTS |
|------------------------|--------------|---|
| <code>type</code> | string | Binding type. For example, <code>queueTrigger</code> . |
| <code>direction</code> | 'in', 'out' | Indicates whether the binding is for receiving data into the function or sending data from the function. |
| <code>name</code> | string | The name that is used for the bound data in the function. For C#, this is an argument name; for JavaScript, it's the key in a key/value list. |

Function app

A function app provides an execution context in Azure in which your functions run. As such, it is the unit of deployment and management for your functions. A function app is comprised 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. Think of a function app as a way to organize and collectively manage your functions. To learn more, see [How to manage a function app](#).

NOTE

All functions in a function app must be authored in the same language. In [previous versions](#) of the Azure Functions runtime, this wasn't required.

Folder structure

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file and one or more subfolders. Each subfolder contains the code for a separate function. The folder structure is shown in the following representation:

```
FunctionApp
| - host.json
| - MyFirstFunction
| | - function.json
| | - ...
| - MySecondFunction
| | - function.json
| | - ...
| - SharedCode
| - bin
```

In version 2.x and higher of the Functions runtime, all functions in the function app must share the same language stack.

The [host.json](#) file contains runtime-specific configurations and is in the root folder of the function app. A *bin* folder contains packages and other library files that the function app requires. See the language-specific requirements for a function app project:

- [C# class library \(.csproj\)](#)
- [C# script \(.csx\)](#)
- [F# script](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

The above is the default (and recommended) folder structure for a Function app. If you wish to change the file location of a function's code, modify the `scriptFile` section of the `function.json` file. We also recommend using [package deployment](#) to deploy your project to your function app in Azure. You can also use existing tools like [continuous integration and deployment](#) and Azure DevOps.

NOTE

If deploying a package manually, make sure to deploy your `host.json` file and function folders directly to the `wwwroot` folder. Do not include the `wwwroot` folder in your deployments. Otherwise, you end up with `wwwroot\wwwroot` folders.

Use local tools and publishing

Function apps can be authored and published using a variety of tools, including [Visual Studio](#), [Visual Studio Code](#), [IntelliJ](#), [Eclipse](#), and the [Azure Functions Core Tools](#). For more information, see [Code and test Azure Functions locally](#).

How to edit functions in the Azure portal

The Functions editor built into the Azure portal lets you update your code and your `function.json` file directly inline. This is recommended only for small changes or proofs of concept - best practice is to use a local development tool like VS Code.

Parallel execution

When multiple triggering events occur faster than a single-threaded function runtime can process them, the runtime may invoke the function multiple times in parallel. If a function app is using the [Consumption hosting plan](#), the function app could scale out automatically. Each instance of the function app, whether the app runs on the Consumption hosting plan or a regular [App Service hosting plan](#), might process concurrent function invocations in parallel using multiple threads. The maximum number of concurrent function invocations in each function app instance varies based on the type of trigger being used as well as the resources used by other functions within the function app.

Functions runtime versioning

You can configure the version of the Functions runtime using the `FUNCTIONS_EXTENSION_VERSION` app setting. For example, the value "`~3`" indicates that your function app will use 3.x as its major version. Function apps are upgraded to each new minor version as they are released. For more information, including how to view the exact version of your function app, see [How to target Azure Functions runtime versions](#).

Repositories

The code for Azure Functions is open source and stored in GitHub repositories:

- [Azure Functions](#)
- [Azure Functions host](#)
- [Azure Functions portal](#)
- [Azure Functions templates](#)
- [Azure WebJobs SDK](#)
- [Azure WebJobs SDK Extensions](#)

Bindings

Here is a table of all supported bindings.

This table shows the bindings that are supported in the major versions of the Azure Functions runtime:

| TYPE | 1.X | 2.X AND HIGHER ¹ | TRIGGER | INPUT | OUTPUT |
|-------------------|-----|-----------------------------|---------|-------|--------|
| Blob storage | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure Cosmos DB | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dapr ³ | | ✓ | ✓ | ✓ | ✓ |

| Type | 1.x | 2.x and higher | Trigger | Input | Output |
|-----------------------|-----|----------------|---------|-------|--------|
| Event Grid | ✓ | ✓ | ✓ | | ✓ |
| Event Hubs | ✓ | ✓ | ✓ | | ✓ |
| HTTP & webhooks | ✓ | ✓ | ✓ | | ✓ |
| IoT Hub | ✓ | ✓ | ✓ | | ✓ |
| Kafka ² | | ✓ | ✓ | | ✓ |
| Mobile Apps | ✓ | | | ✓ | ✓ |
| Notification Hubs | ✓ | | | | ✓ |
| Queue storage | ✓ | ✓ | ✓ | | ✓ |
| RabbitMQ ² | | ✓ | ✓ | | ✓ |
| SendGrid | ✓ | ✓ | | | ✓ |
| Service Bus | ✓ | ✓ | ✓ | | ✓ |
| SignalR | | ✓ | | ✓ | ✓ |
| Table storage | ✓ | ✓ | | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | | |
| Twilio | ✓ | ✓ | | | ✓ |

¹ Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

² Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

³ Supported only in Kubernetes, IoT Edge, and other self-hosted modes only.

Having issues with errors coming from the bindings? Review the [Azure Functions Binding Error Codes](#) documentation.

Reporting Issues

| Item | Description | Link |
|-----------|--|-------------------------------|
| Runtime | Script Host, Triggers & Bindings, Language Support | File an Issue |
| Templates | Code Issues with Creation Template | File an Issue |

| ITEM | DESCRIPTION | LINK |
|--------|------------------------------------|-------------------------------|
| Portal | User Interface or Experience Issue | File an Issue |

Next steps

For more information, see the following resources:

- [Azure Functions triggers and bindings](#)
- [Code and test Azure Functions locally](#)
- [Best Practices for Azure Functions](#)
- [Azure Functions C# developer reference](#)
- [Azure Functions Node.js developer reference](#)

Code and test Azure Functions locally

12/4/2020 • 2 minutes to read • [Edit Online](#)

While you're able to develop and test Azure Functions in the [Azure portal](#), many developers prefer a local development experience. Functions makes it easy to use your favorite code editor and development tools to create and test functions on your local computer. Your local functions can connect to live Azure services, and you can debug them on your local computer using the full Functions runtime.

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:

| ENVIRONMENT | LANGUAGES | DESCRIPTION |
|----------------------------|--|---|
| Visual Studio Code | C# (class library), C# script (.csx), 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# (class library), C# script (.csx), 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 2019 | C# (class library) | The Azure Functions tools are included in the Azure development workload of Visual Studio 2019 and later versions. Lets you compile functions in a class library and publish the .dll to Azure. Includes the Core Tools for local testing. To learn more, see Develop Azure Functions using Visual Studio . |
| Maven (various) | Java | Integrates with 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 |

IMPORTANT

Do not mix local development with portal development in the same function app. When you create and publish functions from a local project, you should not try to maintain or modify project code in the portal.

Each of these local development environments lets you create function app projects and use predefined Functions 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.

Next steps

- To learn more about local development of compiled C# functions using Visual Studio 2019, 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 [Deploy Azure Functions from VS Code](#).
- To learn more about developing functions from the command prompt or terminal, see [Work with Azure Functions Core Tools](#).

Develop Azure Functions by using Visual Studio Code

12/4/2020 • 27 minutes to read • [Edit Online](#)

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.

The extension can be used with the following languages, which are supported by the Azure Functions runtime starting with version 2.x:

- [C# compiled](#)
- [C# script*](#)
- [JavaScript](#)
- [Java](#)
- [PowerShell](#)
- [Python](#)

*Requires that you [set C# script as your default project language](#).

In this article, examples are currently available only for JavaScript (Node.js) and C# class library functions.

This article provides details about how to use the Azure Functions extension to develop functions and publish them to Azure. Before you read this article, you should [create your first function by using Visual Studio Code](#).

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

Before you install and run the [Azure Functions extension](#), you must meet these requirements:

- [Visual Studio Code](#) installed on one of the [supported platforms](#).
- An active Azure subscription.

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

Other resources that you need, like an Azure storage account, are created in your subscription when you [publish by using Visual Studio Code](#).

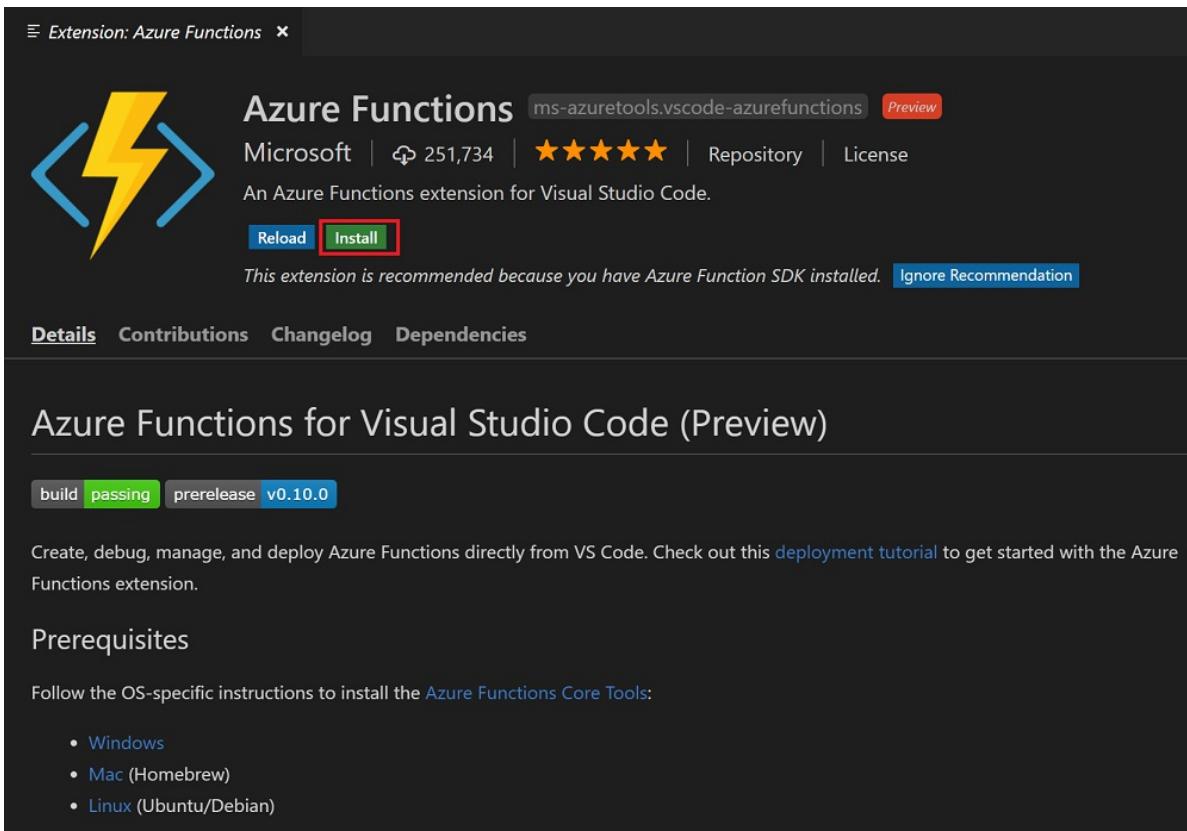
IMPORTANT

You can develop functions locally and publish them to Azure without having to start and run them locally. To run your functions locally, you'll need to meet some additional requirements, including an automatic download of Azure Functions Core Tools. To learn more, see [Additional requirements for running a project locally](#).

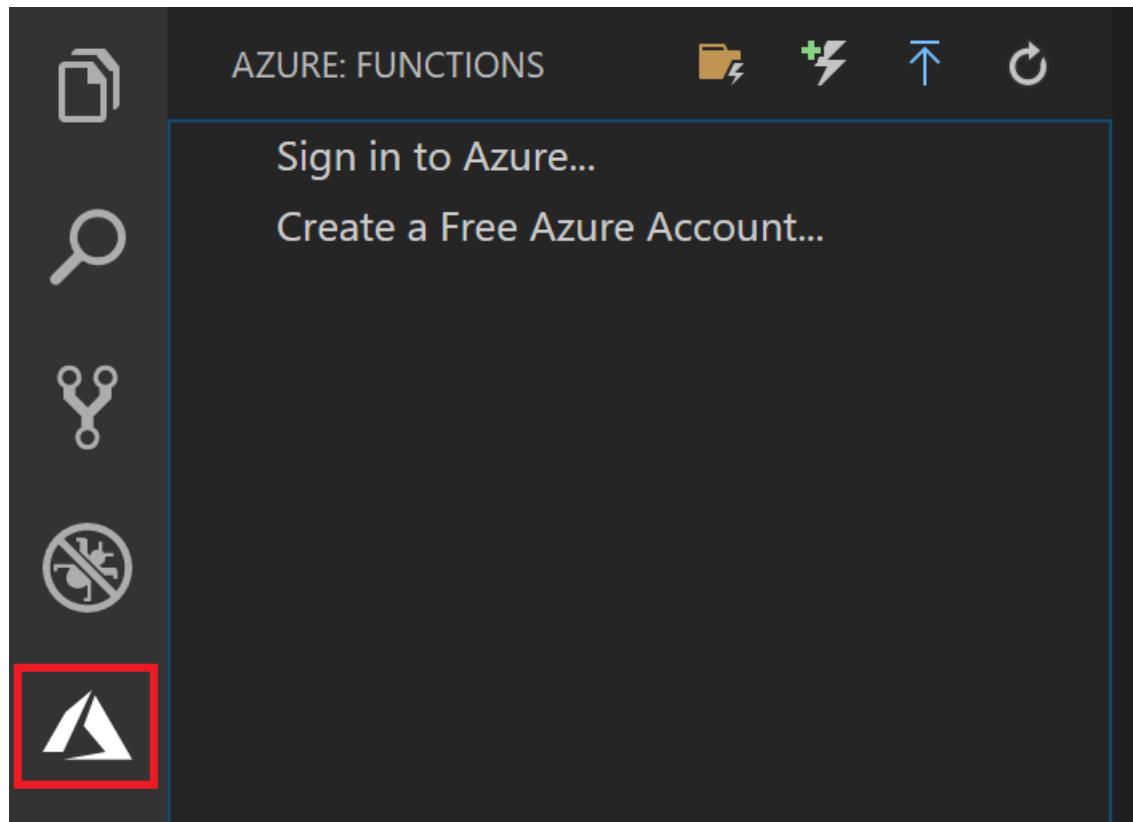
Install the Azure Functions extension

You can use the Azure Functions extension to create and test functions and deploy them to Azure.

1. In Visual Studio Code, open **Extensions** and search for **azure functions**, or select this link in Visual Studio Code: `vscode:extension/ms-azurertools.vscode-azurefunctions`.
2. Select **Install** to install the extension for Visual Studio Code:



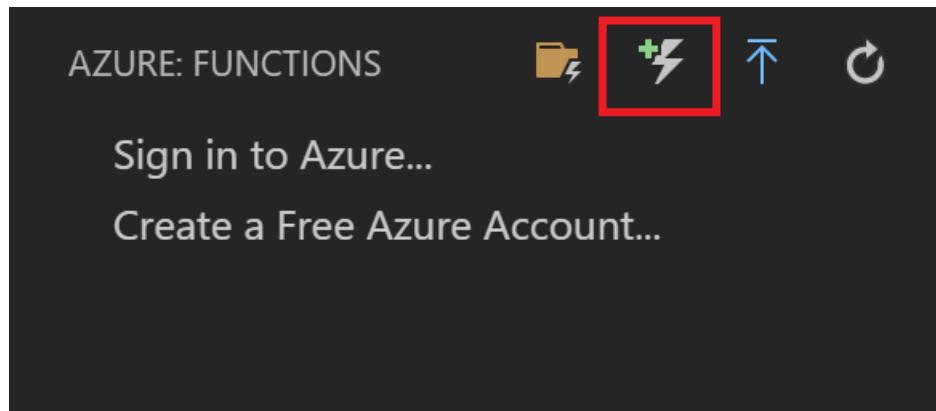
3. After installation, select the Azure icon on the Activity bar. You should see an Azure Functions area in the Side Bar.



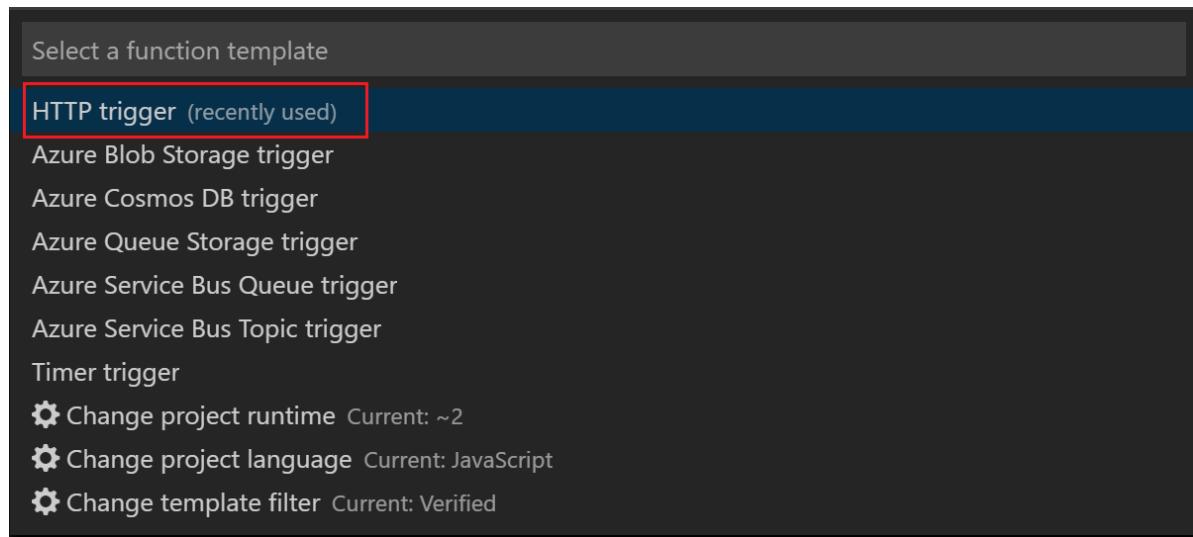
Create an Azure Functions project

The Functions extension lets you create a function app project, along with your first function. The following steps show how to create an HTTP-triggered function in a new Functions project. [HTTP trigger](#) is the simplest function trigger template to demonstrate.

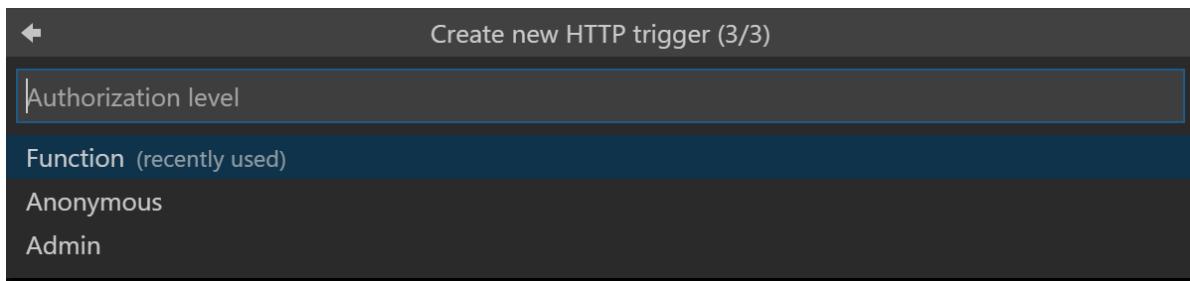
1. From Azure: Functions, select the Create Function icon:



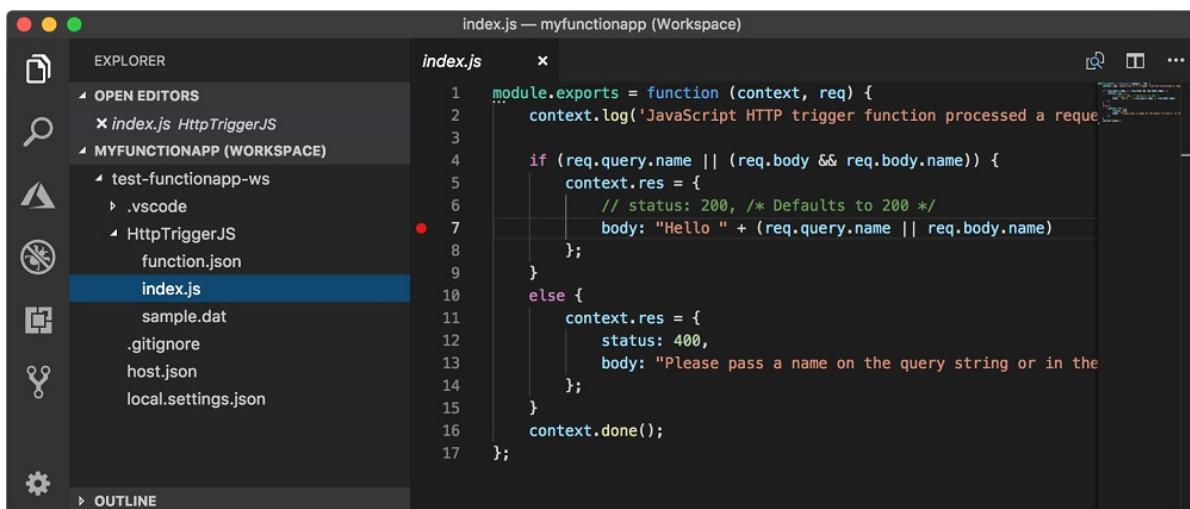
2. Select the folder for your function app project, and then [Select a language for your function project](#).
3. If you haven't already installed the Core Tools, you are asked to [Select a version](#) of the Core Tools to install. Choose version 2.x or a later version.
4. Select the **HTTP trigger** 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.



5. Type **HttpExample** for the function name and select Enter, and then select **Function** authorization. This authorization level requires you to provide a [function key](#) when you call the function endpoint.



A function is created 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 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 running functions locally. 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, you need to exclude it from your project source control.

Depending on your language, these other files are created:

- [C#](#)
- [JavaScript](#)
- [HttpExample.cs class library file](#) that implements the function.

At this point, you can add input and output bindings to your function by [adding a parameter to a C# class library function](#).

You can also [add a new function to your project](#).

Install binding extensions

Except for HTTP and timer triggers, bindings are implemented in extension packages. You must install the extension packages for the triggers and bindings that need them. The process for installing binding extensions depends on your project's language.

- [C#](#)
- [JavaScript](#)

Run the [dotnet add package](#) command in the Terminal window to install the extension packages that you need in your project. The following command installs the Azure Storage extension, which implements bindings for Blob, Queue, and Table storage.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage --version 3.0.4
```

Add a function to your project

You can add a new function to an existing project by using one of the predefined Functions trigger templates. To add a new function trigger, select F1 to open the command palette, and then search for 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 it ready before you create the function trigger.

The results of this action depend on your project's language:

- [C#](#)
- [JavaScript](#)

A new C# class library (.cs) file is added to your project.

Add input and output bindings

You can expand your function by adding input and output bindings. The process for adding bindings depends on your project's language. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

The following examples connect to a storage queue named `outqueue`, where the connection string for the storage account is set in the `MyStorageConnection` application setting in local.settings.json.

- [C#](#)
- [JavaScript](#)

Update the function method to add the following parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("MyStorageConnection")] ICollector<string> msg
```

This code requires you to add the following `using` statement:

```
using Microsoft.Azure.WebJobs.Extensions.Storage;
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written to an output binding when the function completes. You add one or more messages to the collection. These messages are sent to the queue when the function completes.

To learn more, see the [Queue storage output binding](#) documentation.

This table shows the bindings that are supported in the major versions of the Azure Functions runtime:

| TYPE | 1.X | 2.X AND HIGHER ¹ | TRIGGER | INPUT | OUTPUT |
|-----------------------|-----|-----------------------------|---------|-------|--------|
| Blob storage | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure Cosmos DB | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dapr ³ | | ✓ | ✓ | ✓ | ✓ |
| Event Grid | ✓ | ✓ | ✓ | | ✓ |
| Event Hubs | ✓ | ✓ | ✓ | | ✓ |
| HTTP & webhooks | ✓ | ✓ | ✓ | | ✓ |
| IoT Hub | ✓ | ✓ | ✓ | | ✓ |
| Kafka ² | | ✓ | ✓ | | ✓ |
| Mobile Apps | ✓ | | | ✓ | ✓ |
| Notification Hubs | ✓ | | | | ✓ |
| Queue storage | ✓ | ✓ | ✓ | | ✓ |
| RabbitMQ ² | | ✓ | ✓ | | ✓ |
| SendGrid | ✓ | ✓ | | | ✓ |
| Service Bus | ✓ | ✓ | ✓ | | ✓ |
| SignalR | | ✓ | | ✓ | ✓ |
| Table storage | ✓ | ✓ | | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | | |

| Type | 1.x | 2.x and higher | Trigger | Input | Output |
|--------|-----|----------------|---------|-------|--------|
| Twilio | ✓ | ✓ | | | ✓ |

¹ Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

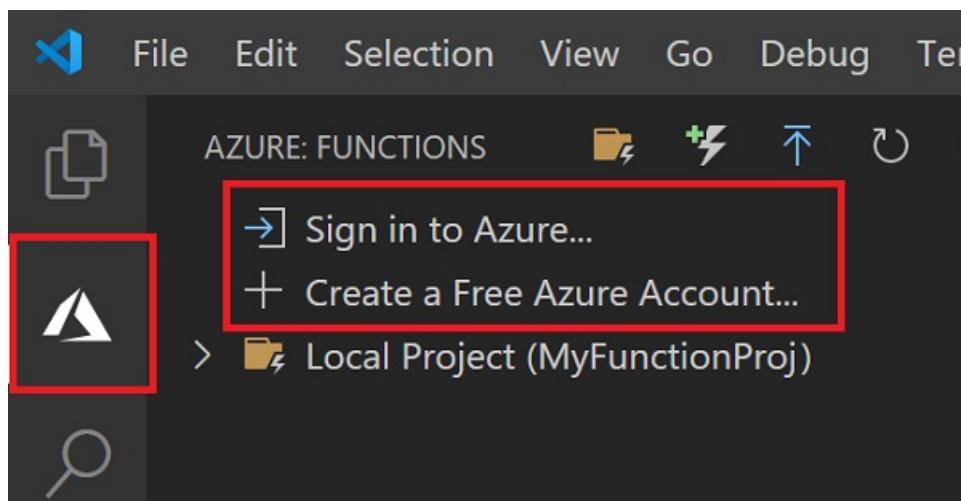
² Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

³ Supported only in Kubernetes, IoT Edge, and other self-hosted modes only.

Sign in to Azure

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

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure....** If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

Publish to Azure

Visual Studio Code lets you publish your Functions project directly to Azure. In the process, you create a function app and related resources in your Azure subscription. The function app provides an execution context for your functions. The project is packaged and deployed to the new function app in your Azure subscription.

When you publish from Visual Studio Code to a new function app in Azure, you are offered both a quick function app create path and an advanced path.

When you publish from Visual Studio Code, you take advantage of the [Zip deploy](#) technology.

Quick function app create

When you choose **+ Create new function app in Azure...**, the extension automatically generates values for the Azure resources needed by your function app. These values are based on the function app name that you choose. For an example of using defaults to publish your project to a new function app in Azure, see the [Visual Studio Code](#)

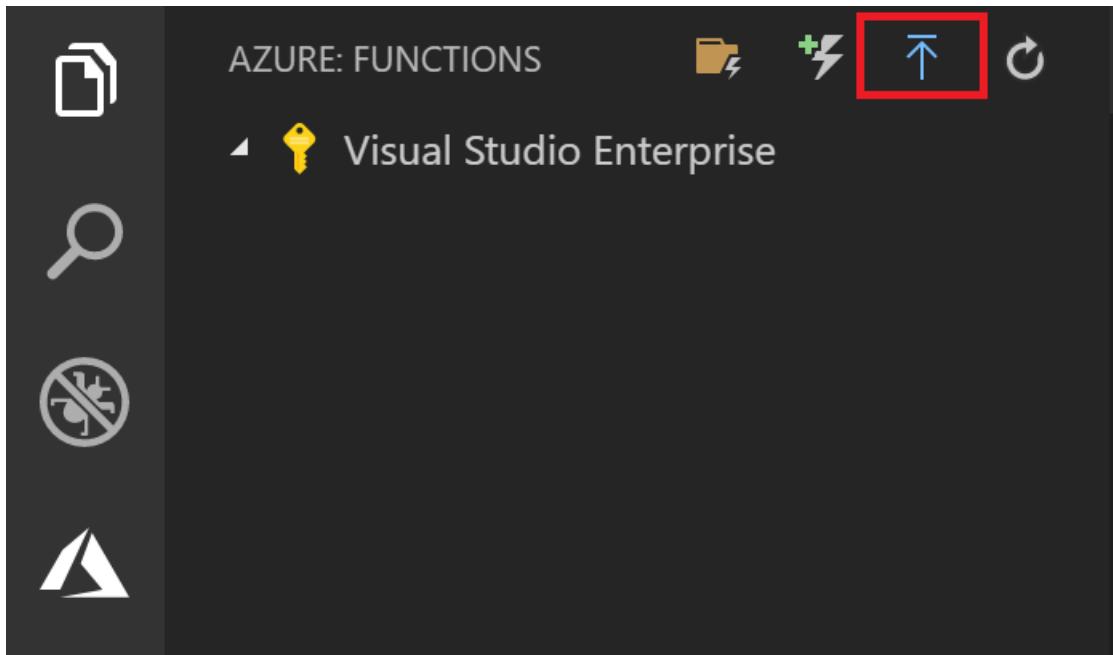
quickstart article.

If you want to provide explicit names for the created resources, you must choose the advanced create path.

Publish a project to a new function app in Azure by using advanced options

The following steps publish your project to a new function app created with advanced create options:

1. In the Azure: Functions area, select the Deploy to Function App icon.



2. If you're not signed in, you're prompted to [Sign in to Azure](#). You can also [Create a free Azure account](#). After signing in from the browser, go back to Visual Studio Code.
3. If you have multiple subscriptions, [Select a subscription](#) for the function app, and then select [+ Create New Function App in Azure... Advanced](#). This *Advanced* option gives you more control over the resources you create in Azure.

4. Following the prompts, provide this information:

| PROMPT | VALUE | DESCRIPTION |
|---|----------------------------------|--|
| Select function app in Azure | Create New Function App in Azure | At the next prompt, 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 an OS | Windows | The function app runs on Windows. |
| Select a hosting plan | Consumption plan | A serverless Consumption plan hosting is used. |
| Select a runtime for your new app | Your project language | The runtime must match the project that you're publishing. |
| Select a resource group for new resources | Create New Resource Group | At the next prompt, type a resource group name, like <code>myResourceGroup</code> , and then select enter. You can also select an existing resource group. |

| PROMPT | VALUE | DESCRIPTION |
|-------------------------------------|----------------------------|--|
| Select a storage account | Create new storage account | At the next prompt, type a globally unique name for the new storage account used by your function app and then select Enter. Storage account names must be between 3 and 24 characters long and can contain only numbers and lowercase letters. You can also select an existing account. |
| Select a location for new resources | region | Select a location in a region near you or near other services that your functions access. |

A notification appears after your function app is created and the deployment package is applied. Select **View Output** in this notification to view the creation and deployment results, including the Azure resources that you created.

Republish project files

When you set up [continuous deployment](#), your function app in Azure is updated whenever source files are updated in the connected source location. We recommend continuous deployment, but you can also republish your project file updates from Visual Studio Code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. In Visual Studio Code, select F1 to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app**.
2. If you're not signed in, you're prompted to **Sign in to Azure**. After you sign in from the browser, go back to Visual Studio Code. If you have multiple subscriptions, **Select a subscription** that contains your function app.
3. Select your existing function app in Azure. When you're warned about overwriting all files in the function app, select **Deploy** to acknowledge the warning and continue.

The project is rebuilt, repackaged, and uploaded to Azure. The existing project is replaced by the new package, and the function app restarts.

Get the URL of the deployed function

To call an HTTP-triggered function, you need the URL of the function when it's deployed to your function app. This URL includes any required [function keys](#). You can use the extension to get these URLs for your deployed functions.

1. Select F1 to open the command palette, and then search for 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 for GET requests to the remote function.

Run functions locally

The Azure Functions extension lets you run a Functions project on your local development computer. The local runtime is the same runtime that hosts your function app in Azure. Local settings are read from the [local.settings.json file](#).

Additional requirements for running a project locally

To run your Functions project locally, you must meet these additional requirements:

- Install version 2.x or later of [Azure Functions Core Tools](#). The Core Tools package is downloaded and installed automatically when you start the project locally. Core Tools includes the entire Azure Functions runtime, so download and installation might take some time.
- Install the specific requirements for your chosen language:

| LANGUAGE | REQUIREMENT |
|------------|---|
| C# | C# extension .NET Core CLI tools |
| Java | Debugger for Java extension Java 8 Maven 3 or later |
| JavaScript | Node.js* |
| Python | Python extension Python 3.6.8 recommended |

*Active LTS and Maintenance LTS versions (8.11.1 and 10.14.1 recommended).

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](#).

Debugging functions locally

To debug your functions, select F5. If you haven't already downloaded [Core Tools](#), you're prompted to do so. When Core Tools is installed and running, output is shown in the Terminal. This is the same as running the `func host start` Core Tools command from the Terminal, but with additional build tasks and an attached debugger.

When the project is running, you can trigger your functions as you would when the project is deployed to Azure. When the project is running in debug mode, breakpoints are hit in Visual Studio Code, as expected.

The request URL for HTTP triggers is displayed in the output in the Terminal. Function keys for HTTP triggers aren't used when a project is running locally. For more information, see [Strategies for testing your code in Azure Functions](#).

To learn more, see [Work with Azure Functions Core Tools](#).

Local settings file

The local.settings.json file stores app settings, connection strings, and settings used by local development tools. Settings in the local.settings.json file are used only when you're running projects locally. The local settings file has this structure:

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "<language worker>",
    "AzureWebJobsStorage": "<connection-string>",
    "AzureWebJobsDashboard": "<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:

| 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 automatically decrypts settings when it runs. Use the <code>func settings decrypt</code> command before trying to read locally encrypted settings. |

| SETTING | DESCRIPTION |
|-------------------|--|
| Values | <p>Array of application settings and connection strings 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 Connection for the Blob storage trigger. For these properties, you need an application setting defined in the Values 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 colon (<code>:</code>) or a double underline (<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 host.json 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. |
| 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 ConnectionStrings section of a configuration file, like Entity Framework . Connection strings in this object are added to the environment with the provider type of System.Data.SqlClient . Items in this collection aren't published to Azure with other app settings. You must explicitly add these values to the Connection strings collection of your function app settings. If you're creating a SqlConnection 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 |
|---------|--------|-------------|
| | | |

| SETTING | VALUES | DESCRIPTION |
|---------------------------------------|--|--|
| AzureWebJobsStorage | 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 Azure Storage Emulator installed locally and you set <code>AzureWebJobsStorage</code> to <code>UseDevelopmentStorage=true</code> , Core Tools uses the emulator. The emulator is useful during development, but you should test with an actual storage connection before deployment. |
| AzureWebJobs.<FUNCTION_NAME>.Disabled | true false | 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 |
| FUNCTIONS_WORKER_RUNTIME | dotnet node java powershell python | 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. |
| FUNCTIONS_WORKER_RUNTIME_VERSION | ~7 | Indicates that PowerShell 7 be used when running locally. If not set, then PowerShell Core 6 is used. This setting is only used when running locally. When running in Azure, the PowerShell runtime version is determined by the <code>powershellVersion</code> site configuration setting, which can be set in the portal . |

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.

The function application settings values can also be read in your code as environment variables. For more information, see the Environment variables sections of these language-specific reference articles:

- [C# precompiled](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)

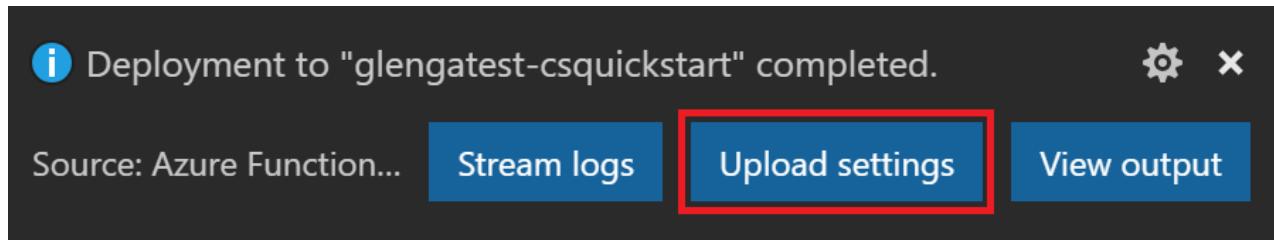
Application settings in Azure

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 must also be added 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:



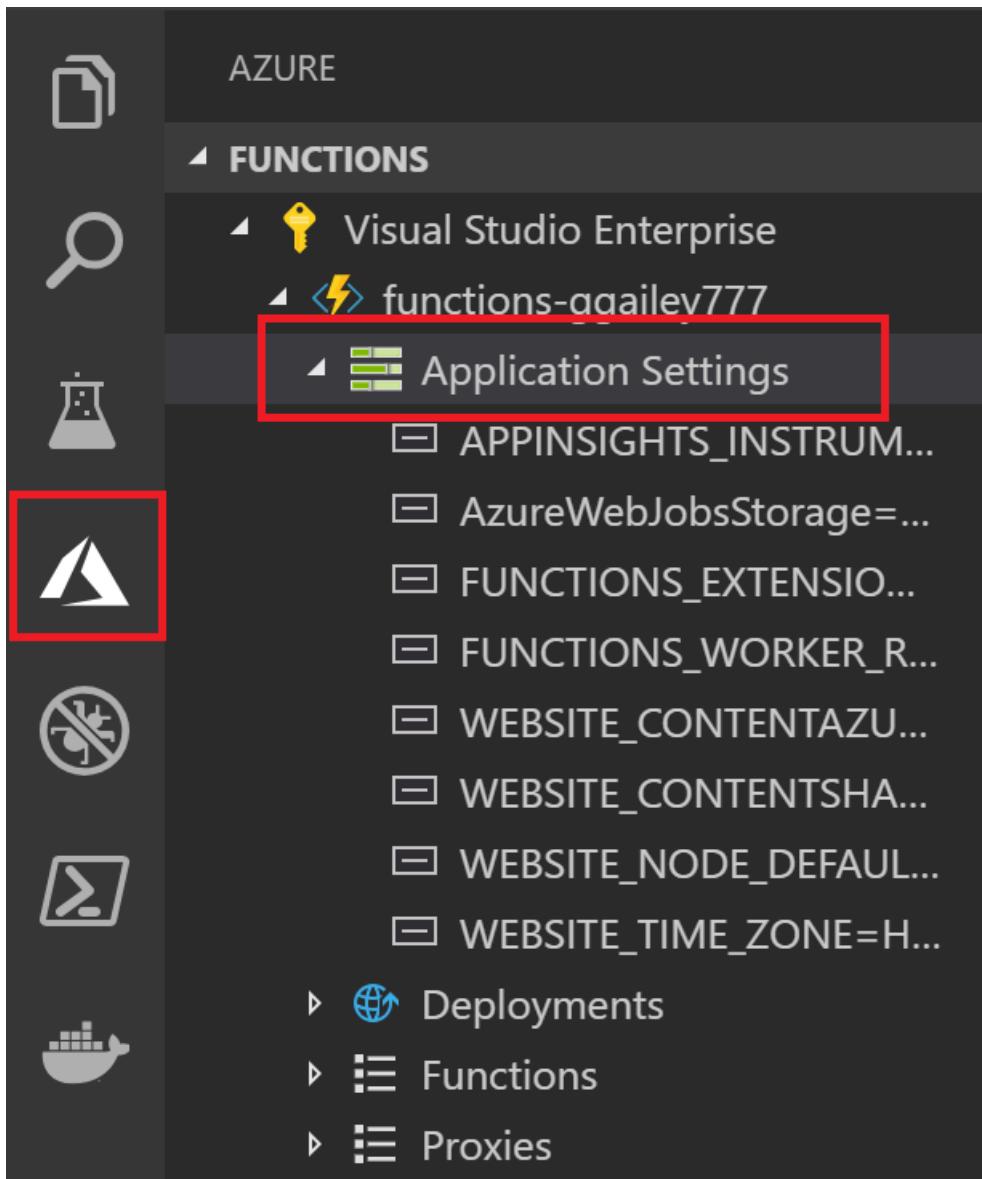
You can also publish settings by using the **Azure Functions: Upload Local Setting** command in the command palette. You can add individual settings to application settings in Azure by using the **Azure Functions: Add New Setting** command.

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.

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 either connect to streaming logs in Azure to see near-real-time log data, or you can 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. 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 output window displays log messages from an Azure Function named 'HttpTrigger1'. The logs include welcome messages, execution details, and host status information, such as the host ID and version.

```
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](#).

To turn on the streaming logs for your function app in Azure:

1. Select F1 to open the command palette, and then search for and run the command **Azure Functions: Start Streaming Logs**.
2. Select your function app in Azure, and then select **Yes** to enable application logging for the function app.
3. Trigger your functions in Azure. Notice that log data is displayed in the Output window in Visual Studio Code.
4. When you're done, remember to run the command **Azure Functions: Stop Streaming Logs** to disable logging for the function app.

NOTE

Streaming logs support only a single instance of the Functions host. When your function is scaled to multiple instances, data from other instances isn't shown in the log stream. [Live Metrics Stream](#) in Application Insights does support multiple instances. While also in near-real time, streaming analytics is based on [sampled data](#).

Application Insights

We recommend that you 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:

| 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: 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. |

| AZURE FUNCTIONS COMMAND | DESCRIPTION |
|--|---|
| 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 will 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 timer-triggered function in Azure. This command is used for testing. To learn more about triggering non-HTTP functions in Azure, see Manually run a non HTTP-triggered function . |
| 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. |
| 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 . |

| AZURE FUNCTIONS COMMAND | DESCRIPTION |
|---|--|
| 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.

Develop Azure Functions using Visual Studio

11/2/2020 • 17 minutes to read • [Edit Online](#)

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](#).

Visual Studio provides the following benefits when you develop your functions:

- Edit, build, and run functions on your local development computer.
- Publish your Azure Functions project directly to Azure, and create Azure resources as needed.
- Use C# attributes to declare function bindings directly in the C# code.
- Develop and deploy pre-compiled C# functions. Pre-compiled functions provide a better cold-start performance than C# script-based functions.
- Code your functions in C# while having all of the benefits of Visual Studio development.

This article provides details about how to use Visual Studio to develop C# class library functions and publish them to Azure. Before you read this article, consider completing the [Functions quickstart for Visual Studio](#).

Unless otherwise noted, procedures and examples shown are for Visual Studio 2019.

Prerequisites

- Azure Functions Tools. To add Azure Function Tools, include the **Azure development** workload in your Visual Studio installation. Azure Functions Tools is available in the Azure development workload starting with Visual Studio 2017.
- 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 a [free account](#) before you begin.

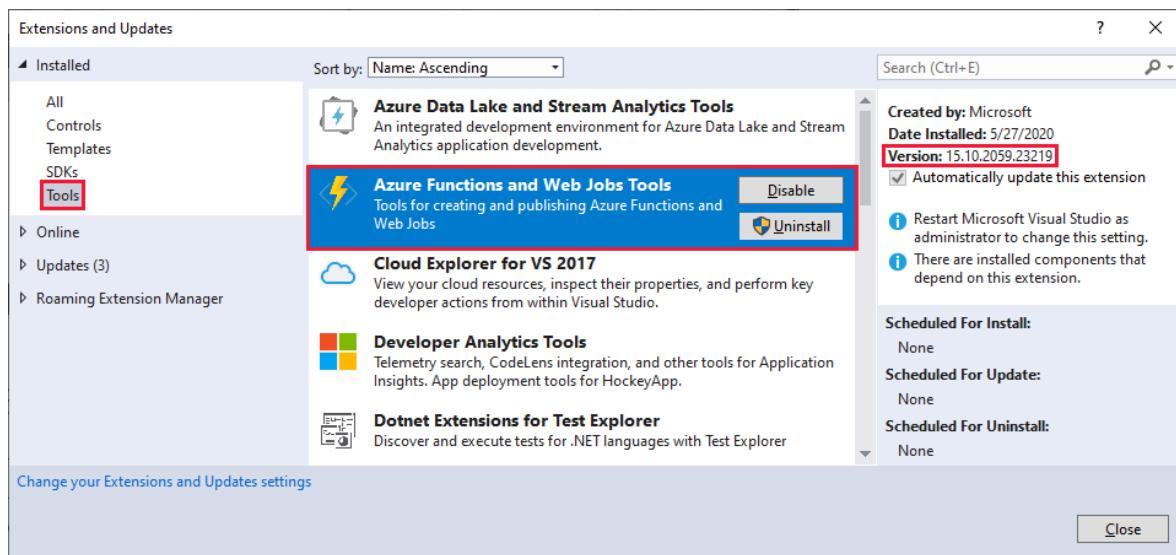
NOTE

In Visual Studio 2017, the Azure development workload installs Azure Functions Tools as a separate extension. When you update your Visual Studio 2017 installation, make sure that you're using the [most recent version](#) of the Azure Functions tools. The following sections show you how to check and (if needed) update your Azure Functions Tools extension in Visual Studio 2017.

Skip these sections if you're using Visual Studio 2019.

Check your tools version in Visual Studio 2017

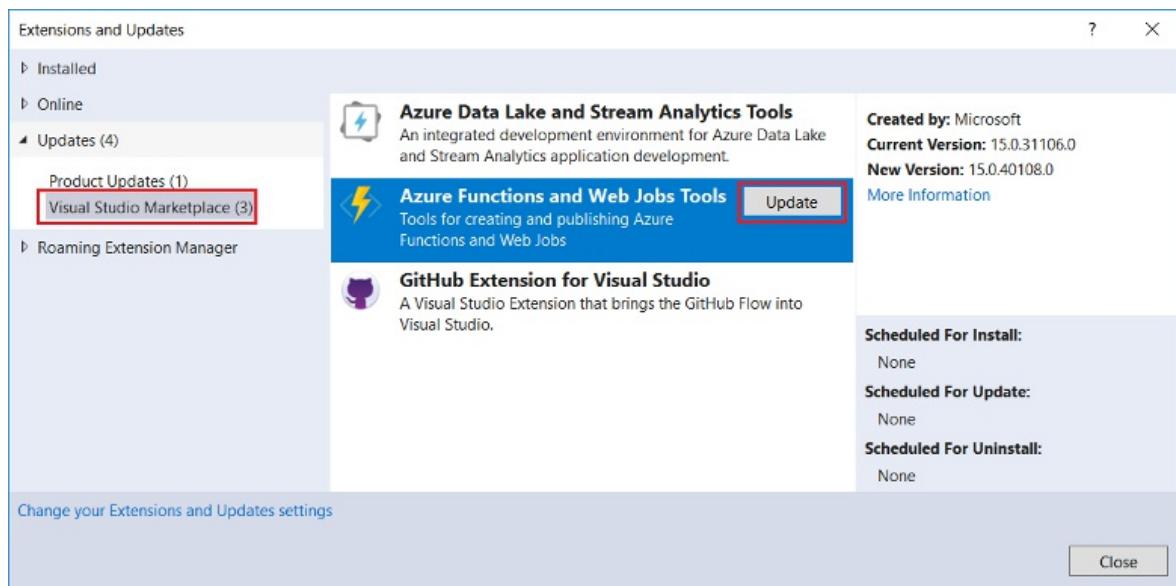
1. From the **Tools** menu, choose **Extensions and Updates**. Expand **Installed > Tools**, and then choose **Azure Functions and Web Jobs Tools**.



2. Note the installed **Version** and compare this version with the latest version listed in the [release notes](#).
3. If your version is older, update your tools in Visual Studio as shown in the following section.

Update your tools in Visual Studio 2017

1. In the Extensions and Updates dialog, expand **Updates > Visual Studio Marketplace**, choose **Azure Functions and Web Jobs Tools** and select **Update**.



2. After the tools update is downloaded, select **Close**, and then close Visual Studio to trigger the tools update with VSIX Installer.
3. In VSIX Installer, choose **Modify** to update the tools.
4. After the update is complete, choose **Close**, and then restart Visual Studio.

NOTE

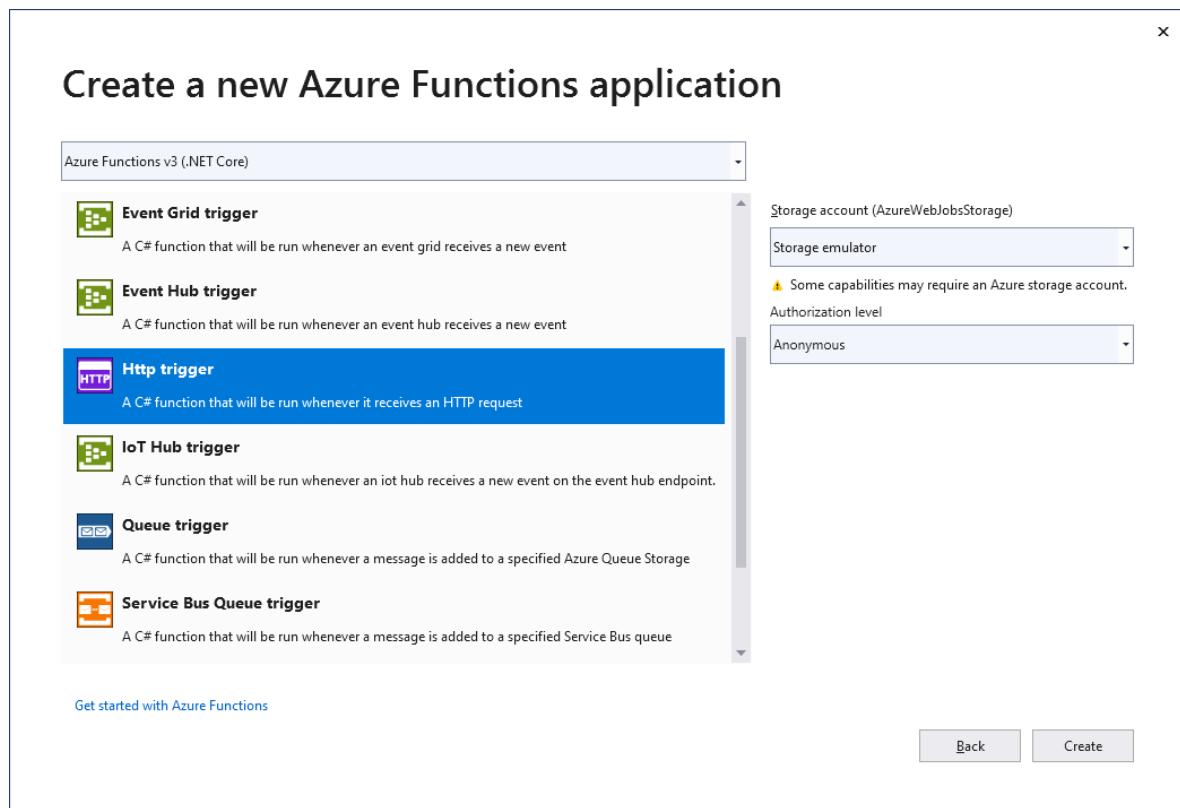
In Visual Studio 2019 and later, the Azure Functions tools extension is updated as part of Visual Studio.

Create an Azure Functions project

The Azure Functions project template in Visual Studio creates a 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:

| SETTING | VALUE | DESCRIPTION |
|--|---------------------------------------|--|
| Functions runtime | Azure Functions v3 (.NET Core) | This value creates a function project that uses the version 3.x runtime of Azure Functions, which supports .NET Core 3.x. Azure Functions 1.x supports the .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 an Azure Function 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.NET.Sdk.Functions` NuGet package, 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. Ensure the **Copy to Output Directory** setting for this file is set to **Copy if newer**.

For more information, see [Functions class library project](#).

Local settings file

The local.settings.json file stores app settings, connection strings, and settings used by local development tools. Settings in the local.settings.json file are used only when you're running projects locally. The local settings file has this structure:

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "<language worker>",
    "AzureWebJobsStorage": "<connection-string>",
    "AzureWebJobsDashboard": "<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:

| SETTING | DESCRIPTION |
|----------------------------|--|
| <code>IsEncrypted</code> | 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 <code>local.settings.json</code> file on your local computer when it contains secrets, such as service connection strings. The host automatically decrypts settings when it runs. Use the <code>func settings decrypt</code> command before trying to read locally encrypted settings. |
| <code>Values</code> | Array of application settings and connection strings used when a project is running locally. These key-value (string-string) pairs correspond to application settings in your function app in Azure, like <code>AzureWebJobsStorage</code> . 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. Values must be strings and not JSON objects or arrays. Setting names can't include a colon (<code>:</code>) or a double underline (<code>__</code>). Double underline characters are reserved by the runtime, and the colon is reserved to support dependency injection . |
| <code>Host</code> | 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. |
| <code>LocalHttpPort</code> | 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. |
| <code>CORS</code> | 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. |

| SETTING | DESCRIPTION |
|-------------------|---|
| CORScredentials | 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 Azure Storage Emulator installed locally and you set <code>AzureWebJobsStorage</code> to <code>UseDevelopmentStorage=true</code> , Core Tools uses the emulator. The emulator is useful during development, but you should test with an actual storage connection before deployment. |
| <code>AzureWebJobs.<FUNCTION_NAME>.Disabled</code> | <code>true</code> <code>false</code> | To disable a function when running locally, add <pre>"AzureWebJobs.<FUNCTION_NAME>.Disabled": "true"</pre> 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>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. |

| SETTING | VALUES | DESCRIPTION |
|----------------------------------|--------|--|
| FUNCTIONS_WORKER_RUNTIME_VERSION | ~7 | Indicates that PowerShell 7 be used when running locally. If not set, then PowerShell Core 6 is used. This setting is only used when running locally. When running in Azure, the PowerShell runtime version is determined by the <code>powershellVersion</code> site configuration setting, which can be set in the portal . |

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 [Azure Storage Emulator](#) for the `AzureWebJobsStorage` connection setting that's 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.

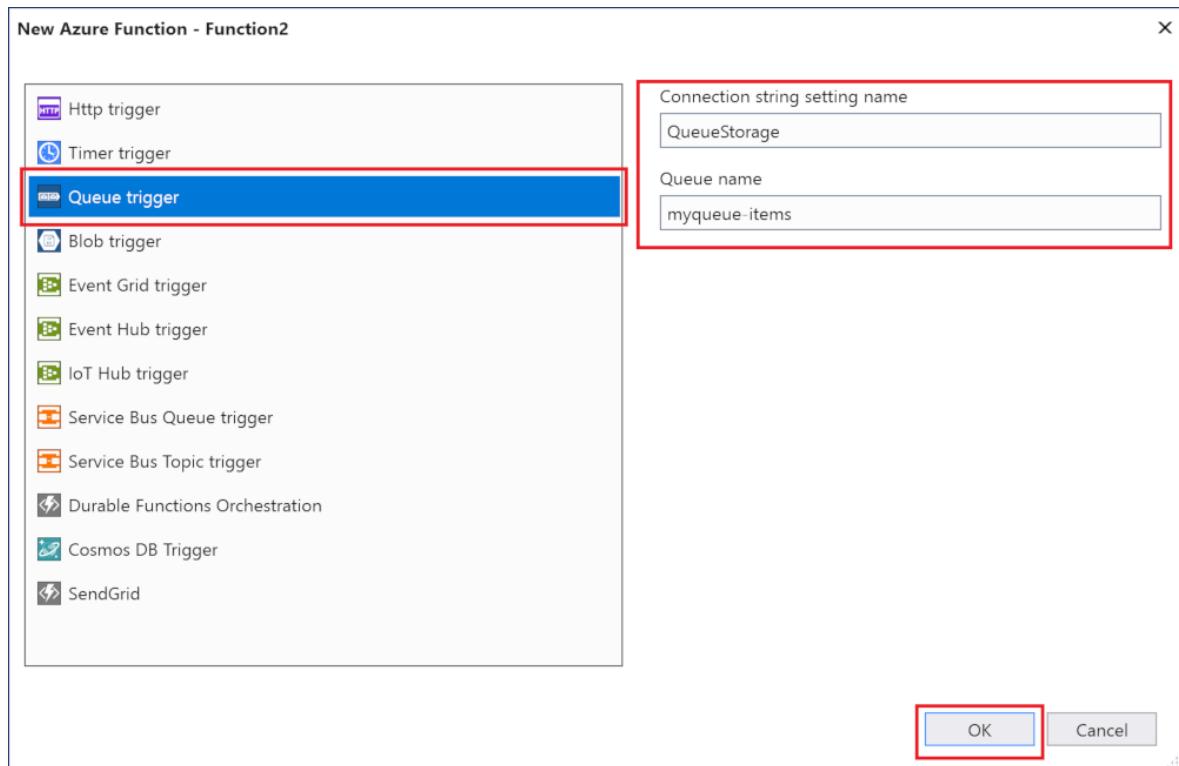
To set the storage account connection string:

1. In Visual Studio, select **View > Cloud Explorer**.
2. In **Cloud Explorer**, expand **Storage Accounts**, and then select your storage account. In the **Properties** tab, copy the **Primary Connection String** value.
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 Item**.
2. Select **Azure Function**, enter a **Name** for the class, and then select **Add**.
3. Choose your trigger, set the binding properties, and then select **OK**. The following example shows the settings for creating a Queue storage trigger function.



This trigger example uses a connection string with a key named `QueueStorage`. Define this connection string setting in the [local.settings.json file](#).

4. Examine the newly added class. You see a static `Run()` method that's attributed with the `FunctionName` attribute. This attribute indicates that the method is the entry point for the function.

For example, the following C# class represents a basic Queue storage trigger function:

```
using System;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace FunctionApp1
{
    public static class Function1
    {
        [FunctionName("QueueTriggerCSharp")]
        public static void Run([QueueTrigger("myqueue-items",
            Connection = "QueueStorage")]string myQueueItem, ILogger log)
        {
            log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
        }
    }
}
```

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've [configured the project for local development](#).
2. Add the appropriate NuGet extension package for the specific binding.

For more information, see [C# class library with Visual Studio](#). Find 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. 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](#).

4. 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.

```
public static class SimpleExampleWithOutput
{
    [FunctionName("CopyQueueMessage")]
    public static void Run(
        [QueueTrigger("myqueue-items-source", Connection = "AzureWebJobsStorage")] string myQueueItem,
        [Queue("myqueue-items-destination", Connection = "AzureWebJobsStorage")] out string
myQueueItemCopy,
        ILogger log)
    {
        log.LogInformation($"CopyQueueMessage function processed: {myQueueItem}");
        myQueueItemCopy = myQueueItem;
    }
}
```

The connection to Queue storage is obtained from the `AzureWebJobsStorage` setting. For more information, see the reference article for the specific binding.

This table shows the bindings that are supported in the major versions of the Azure Functions runtime:

| TYPE | 1.X | 2.X AND HIGHER ¹ | TRIGGER | INPUT | OUTPUT |
|-------------------|-----|-----------------------------|---------|-------|--------|
| Blob storage | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure Cosmos DB | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dapr ³ | | ✓ | ✓ | ✓ | ✓ |
| Event Grid | ✓ | ✓ | ✓ | | ✓ |
| Event Hubs | ✓ | ✓ | ✓ | | ✓ |
| HTTP & webhooks | ✓ | ✓ | ✓ | | ✓ |

| Type | 1.x | 2.x and higher | Trigger | Input | Output |
|-----------------------|-----|----------------|---------|-------|--------|
| IoT Hub | ✓ | ✓ | ✓ | | ✓ |
| Kafka ² | | ✓ | ✓ | | ✓ |
| Mobile Apps | ✓ | | | ✓ | ✓ |
| Notification Hubs | ✓ | | | | ✓ |
| Queue storage | ✓ | ✓ | ✓ | | ✓ |
| RabbitMQ ² | | ✓ | ✓ | | ✓ |
| SendGrid | ✓ | ✓ | | | ✓ |
| Service Bus | ✓ | ✓ | ✓ | | ✓ |
| SignalR | | ✓ | | ✓ | ✓ |
| Table storage | ✓ | ✓ | | ✓ | ✓ |
| Timer | ✓ | ✓ | ✓ | | |
| Twilio | ✓ | ✓ | | | ✓ |

¹ Starting with the version 2.x runtime, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

² Triggers aren't supported in the Consumption plan. Requires [runtime-driven triggers](#).

³ Supported only in Kubernetes, IoT Edge, and other self-hosted modes only.

Testing functions

Azure Functions Core Tools lets you run Azure Functions project on your local development computer. 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 test your function in Visual Studio:

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.

For more information, see [Strategies for testing your code in Azure Functions](#). When you run Visual Studio in debug mode, breakpoints are hit as expected.

Publish to Azure

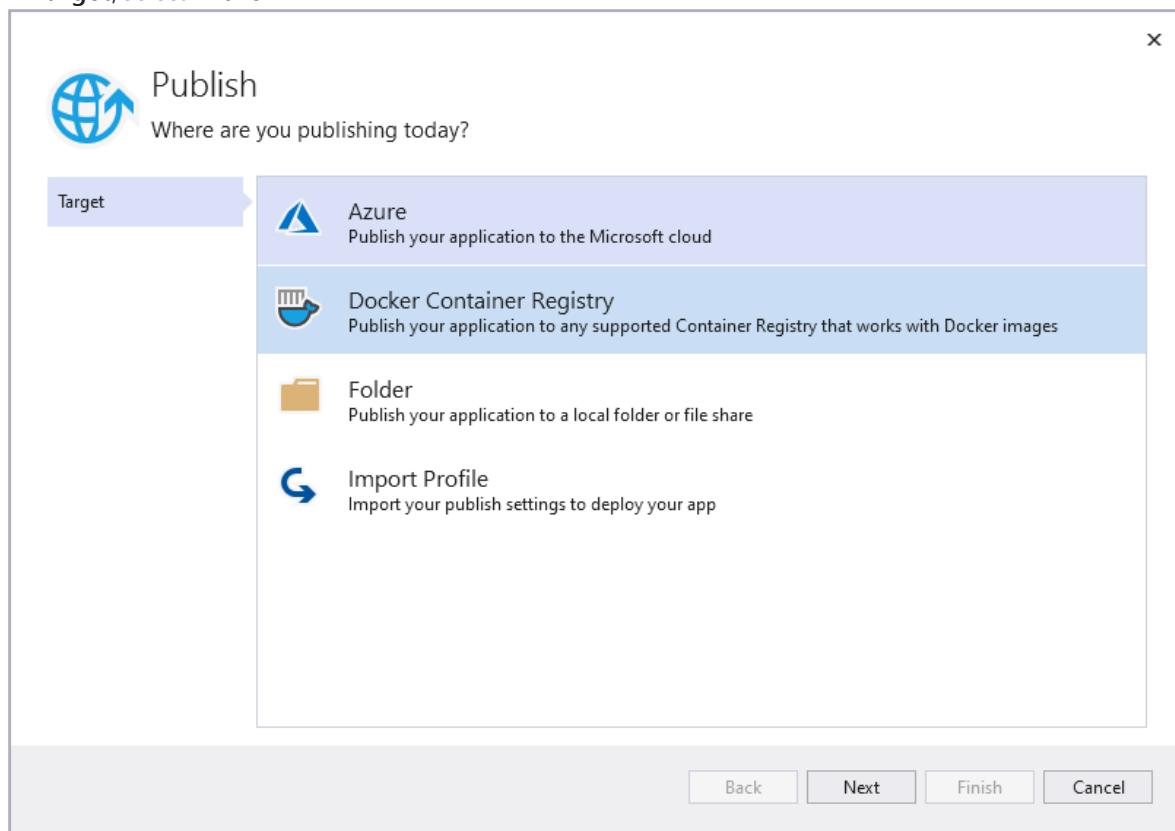
When you publish from Visual Studio, it uses one of two deployment methods:

- [Web Deploy](#): Packages and deploys Windows apps to any IIS server.
- [Zip Deploy with run-From-package enabled](#): Recommended for Azure Functions deployments.

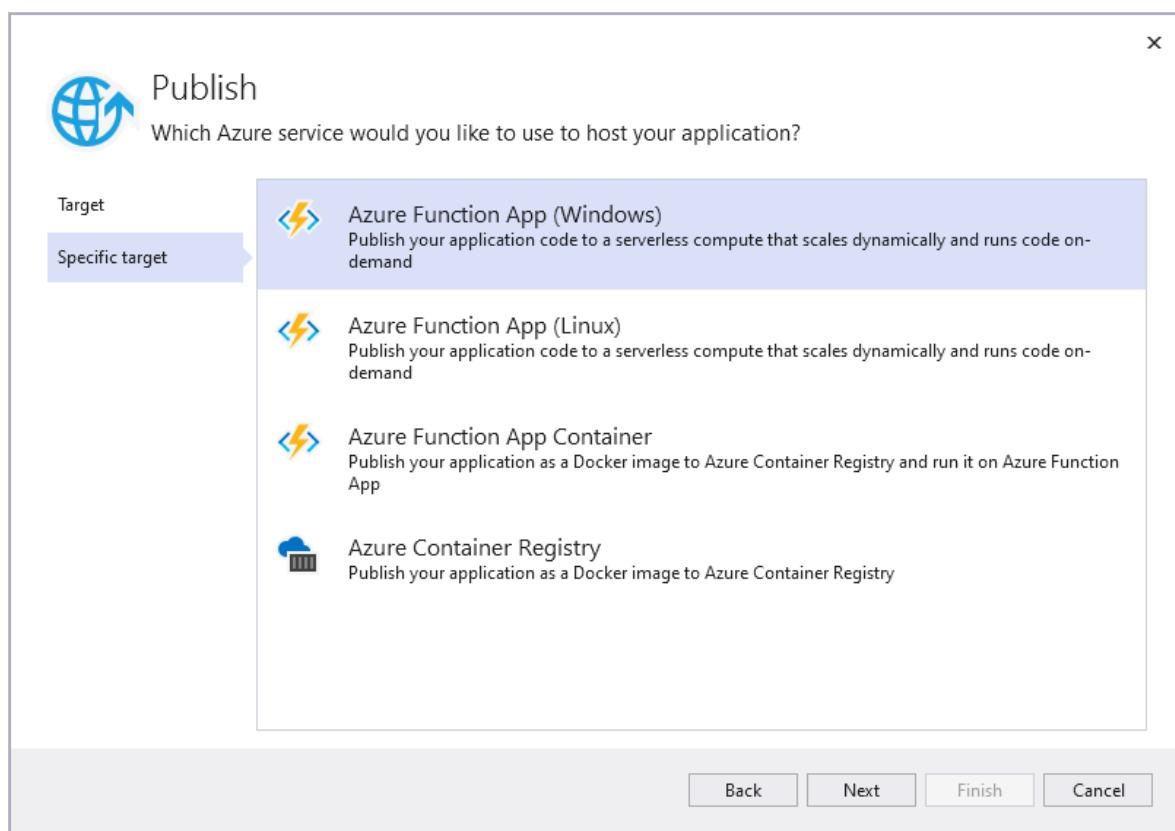
Use the following steps to publish your project to a function app in Azure.

1. In Solution Explorer, right-click the project and select **Publish**.

2. In **Target**, select **Azure**



3. In **Specific target**, select **Azure Function App (Windows)**



4. In **Function Instance**, select **Create a new Azure Function...** and then use the values specified in the following 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: <code>a-z</code> , <code>0-9</code> , and <code>-</code> . |
| 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 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 storage account. You can also choose an existing account that meets the storage account requirements . |

 Function App (Windows)
Create new

Name
FunctionApp220200930124040

Subscription
Visual Studio Enterprise

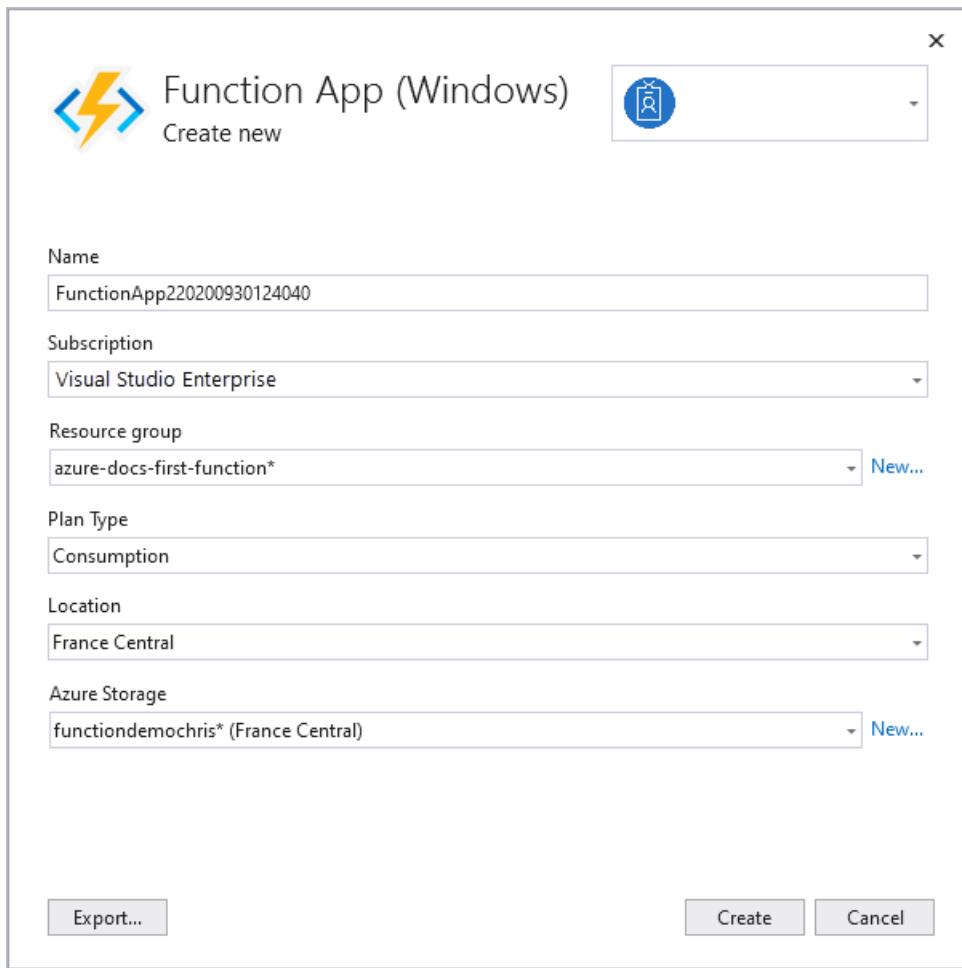
Resource group
azure-docs-first-function* [New...](#)

Plan Type
Consumption

Location
France Central

Azure Storage
functiondemochris* (France Central) [New...](#)

[Export...](#) [Create](#) [Cancel](#)



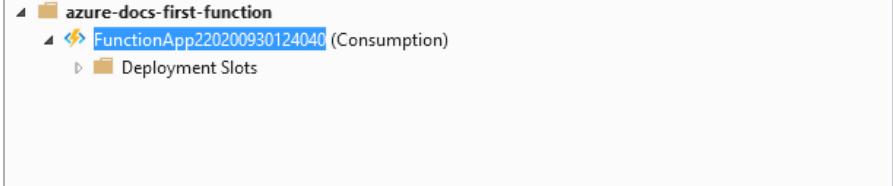
5. Select **Create** to create a function app and its related resources in Azure.
6. In the **Functions instance**, make sure that **Run from package file** is checked. Your function app is deployed using **Zip Deploy** with **Run-From-Package** mode enabled. This is the recommended deployment method for your functions project, since it results in better performance.

 Publish
Select existing or create a new Azure Function

Subscription
ca-chmaneu-demo-test

Target
Specific target
Functions instance

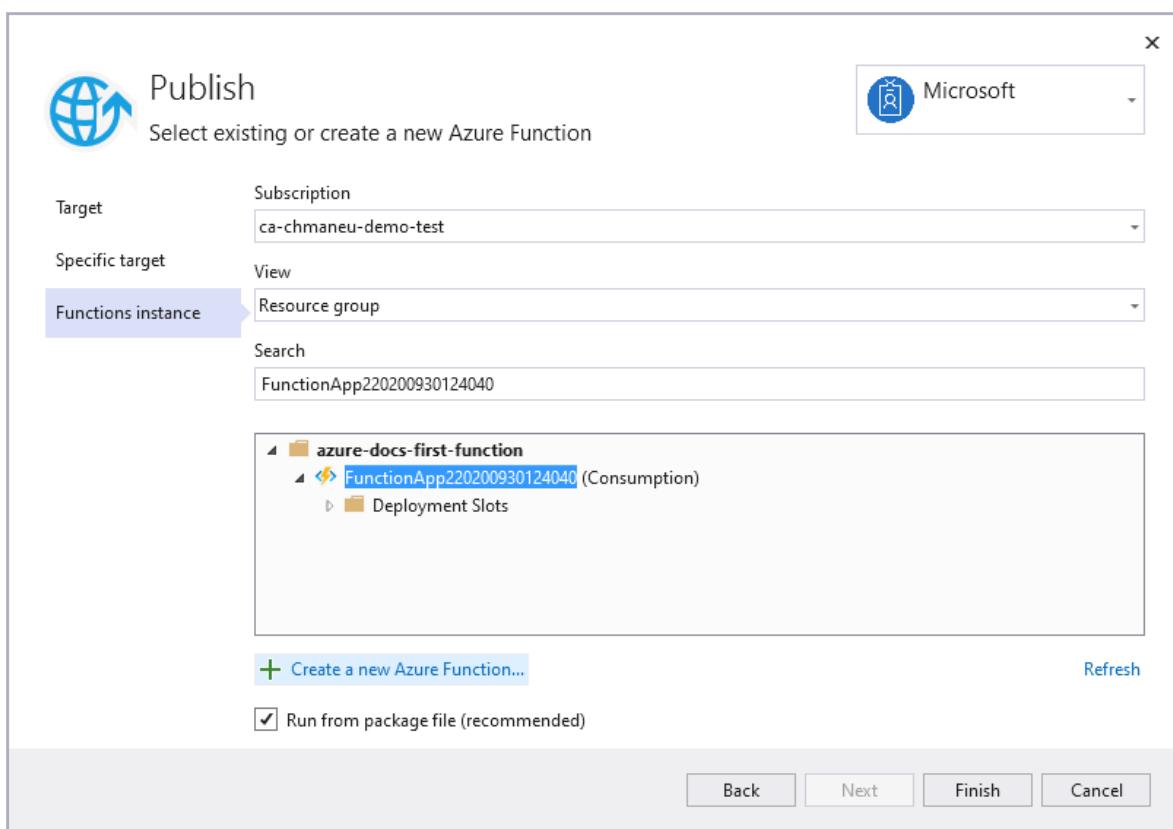
Resource group
Search
FunctionApp220200930124040



[+ Create a new Azure Function...](#) [Refresh](#)

Run from package file (recommended)

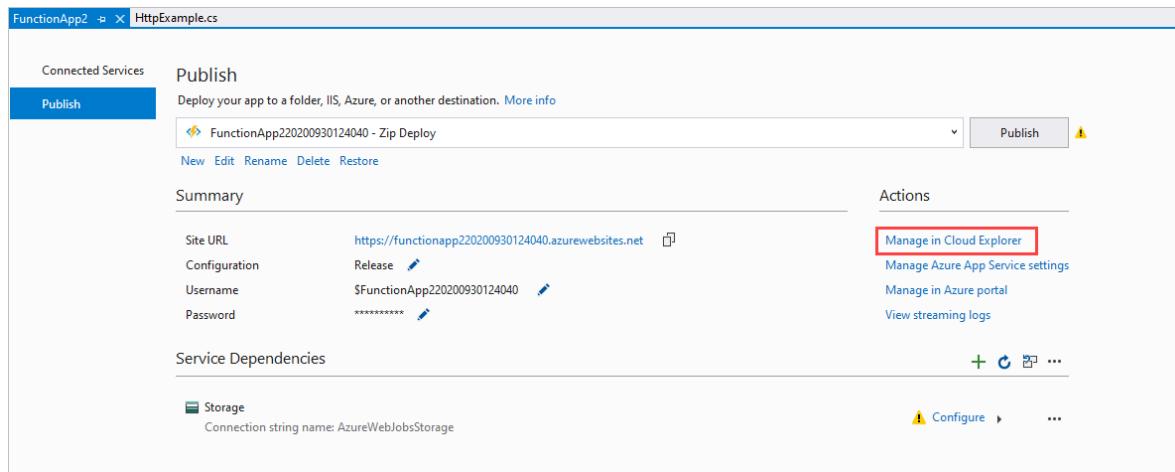
[Back](#) [Next](#) [Finish](#) [Cancel](#)



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, choose **Manage in Cloud Explorer**. This opens the new function app Azure resource in Cloud Explorer.

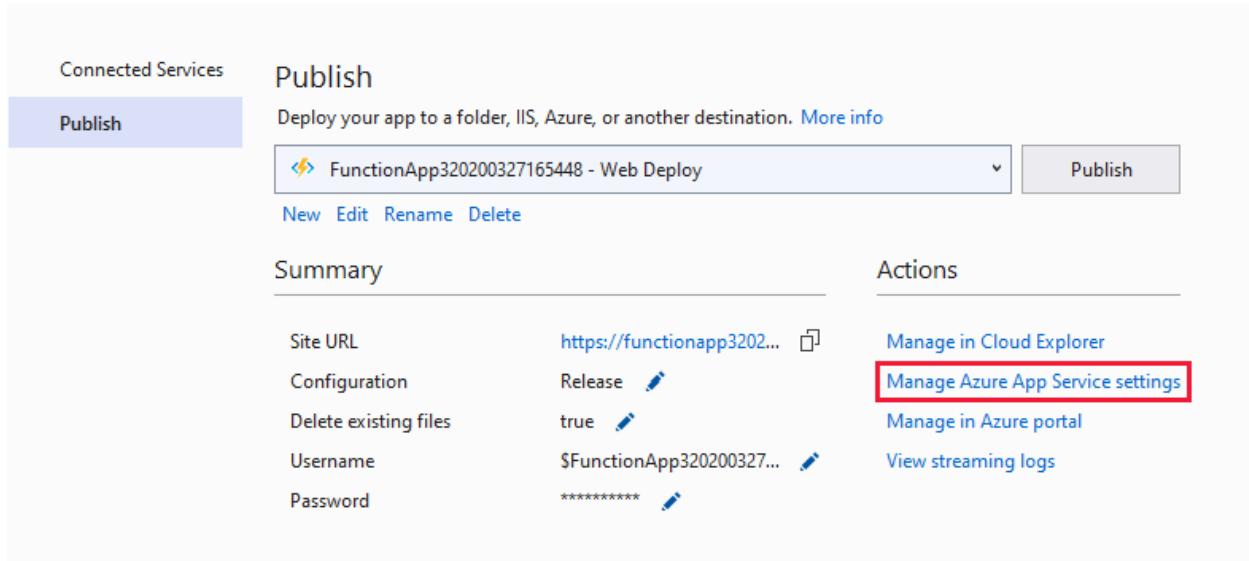


Cloud Explorer lets you use Visual Studio to view the contents of the site, start and stop the function app, and browse directly to function app resources on Azure and in the Azure portal.

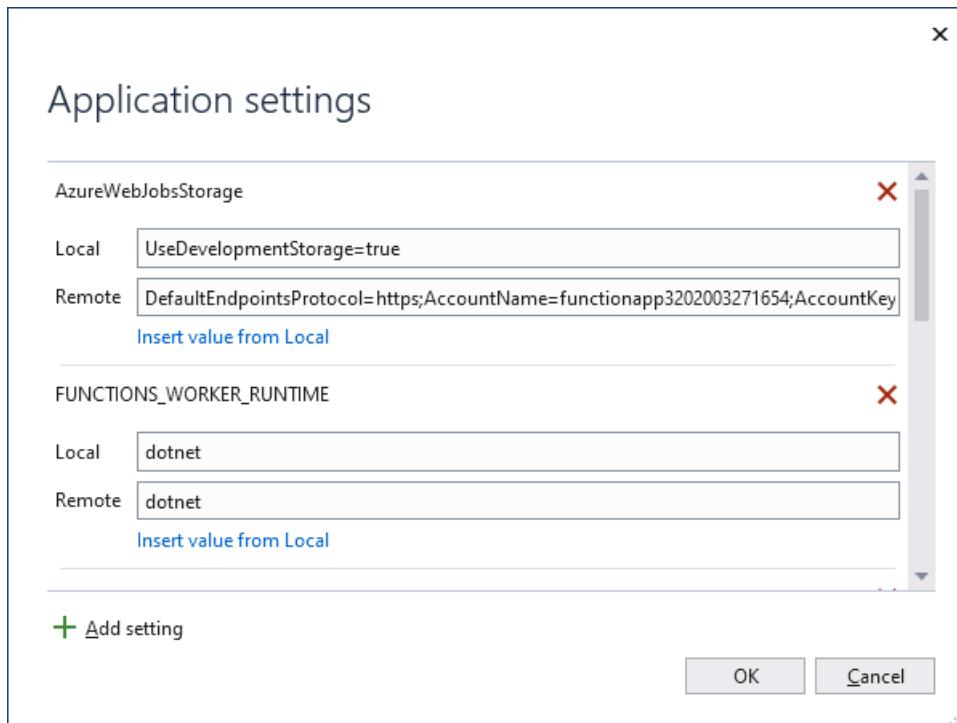
Function app settings

Because 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 select the **Manage Azure App Service settings** link that appears after you successfully publish your project.



Selecting this link displays the **Application settings** dialog for the function app, where you can add new application settings or modify existing ones.



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.

Monitoring functions

The recommended way to monitor the execution of your functions is by integrating your function app with Azure Application Insights. When you create a function app in the Azure portal, this integration is done for you by default. However, when you create your function app during Visual Studio publishing, the integration in your function app in Azure isn't done. To learn how to connect Application Insights to your function app, see [Enable Application Insights integration](#).

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](#).

For more information about developing functions as .NET class libraries, see [Azure Functions C# developer reference](#). This article also links to examples of how to use attributes to declare the various types of bindings supported by Azure Functions.

Work with Azure Functions Core Tools

12/4/2020 • 25 minutes to read • [Edit Online](#)

Azure Functions Core Tools lets you develop and test your functions on your local computer from the command prompt or terminal. Your local functions can connect to live Azure services, and you can debug your functions on your local computer using the full Functions runtime. You can even deploy a function app to your Azure subscription.

IMPORTANT

Do not mix local development with portal development in the same function app. When you create and publish functions from a local project, you should not try to maintain or modify project code in the portal.

Developing functions on your local computer and publishing them to Azure using Core Tools follows these basic steps:

- [Install the Core Tools and dependencies.](#)
- [Create a function app project from a language-specific template.](#)
- [Register trigger and binding extensions.](#)
- [Define Storage and other connections.](#)
- [Create a function from a trigger and language-specific template.](#)
- [Run the function locally.](#)
- [Publish the project to Azure.](#)

Core Tools versions

There are three versions of Azure Functions Core Tools. The version you use depends on your local development environment, [choice of language](#), and level of support required:

- **Version 3.x/2.x:** Supports either [version 3.x or 2.x of the Azure Functions runtime](#). These versions support .NET Core and .NET Framework, and use platform-specific package managers or npm for installation.
- **Version 1.x:** Supports version 1.x of the Azure Functions runtime. This version of the tools is only supported on Windows computers and is installed from an [npm package](#).

You can only install one version of Core Tools on a given computer. Unless otherwise noted, the examples in this article are for version 3.x.

Prerequisites

Azure Functions Core Tools currently depends on the Azure CLI for authenticating with your Azure account. This means that you must [install the Azure CLI locally](#) to be able to [publish to Azure](#) from Azure Functions Core Tools.

Install the Azure Functions Core Tools

[Azure Functions Core Tools](#) includes a version of the same runtime that powers Azure Functions runtime that you can run on your local development computer. It also provides commands to create functions, connect to Azure, and deploy function projects.

Version 3.x and 2.x

Version 3.x/2.x of the tools uses the Azure Functions runtime that is built on .NET Core. This version is supported on all platforms .NET Core supports, including , , and .

IMPORTANT

You can bypass the requirement for installing the .NET Core SDK by using [extension bundles](#).

- [Windows](#)
- [macOS](#)
- [Linux](#)

The following steps use a Windows installer (MSI) to install Core Tools v3.x. For more information about other package-based installers, which are required to install Core Tools v2.x, see the [Core Tools readme](#).

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

- [v3.x - Windows 64-bit](#) (Recommended. [Visual Studio Code debugging](#) requires 64-bit.)
- [v3.x - Windows 32-bit](#)

2. If you don't plan to use [extension bundles](#), install the [.NET Core 3.x SDK for Windows](#).

Create a local Functions project

A functions project directory contains the files `host.json` and `local.settings.json`, along with subfolders that contain the code for individual functions. This directory is the equivalent of a function app in Azure. To learn more about the Functions folder structure, see the [Azure Functions developers guide](#).

Version 3.x/2.x requires you to select a default language for your project when it is initialized. In version 3.x/2.x, all functions added use default language templates. In version 1.x, you specify the language each time you create a function.

In the terminal window or from a command prompt, run the following command to create the project and local Git repository:

```
func init MyFunctionProj
```

IMPORTANT

Java uses a Maven archetype to create the local Functions project, along with your first HTTP triggered function.

Use the following command to create your Java project:

```
mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure -DarchetypeArtifactId=azure-functions-archetype
```

. For an example using the Maven archetype, see the [Command line quickstart](#).

When you provide a project name, a new folder with that name is created and initialized. Otherwise, the current folder is initialized.

In version 3.x/2.x, when you run the command you must choose a runtime for your project.

```
Select a worker runtime:  
dotnet  
node  
python  
powershell
```

Use the up/down arrow keys to choose a language, then press Enter. If you plan to develop JavaScript or TypeScript functions, choose `node`, and then select the language. TypeScript has [some additional requirements](#).

The output looks like the following example for a JavaScript project:

```
Select a worker runtime: node
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing C:\myfunctions\myMyFunctionProj\.vscode\extensions.json
Initialized empty Git repository in C:/myfunctions/myMyFunctionProj/.git/
```

`func init` supports the following options, which are version 3.x/2.x-only, unless otherwise noted:

| 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 publish to a custom Linux container. |
| <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 publish an existing project to a custom Linux container. |
| <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>--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>javascript</code> , <code>node</code> (JavaScript), <code>powershell</code> , <code>python</code> , and <code>typescript</code> . For Java, use Maven . When not set, you're prompted to choose your runtime during initialization. |

IMPORTANT

By default, version 2.x and later versions of the Core Tools create function app projects for the .NET runtime as [C# class projects \(.csproj\)](#). These C# projects, which can be used with Visual Studio or Visual Studio Code, are compiled during testing and when publishing to Azure. If you instead want to create and work with the same C# script (.csx) files created in version 1.x and in the portal, you must include the `--csx` parameter when you create and deploy functions.

Register extensions

With the exception of HTTP and timer triggers, Functions bindings in runtime version 2.x and higher are implemented as extension packages. HTTP bindings and timer triggers don't require extensions.

To reduce incompatibilities between the various extension packages, Functions lets you reference an extension bundle in your `host.json` project file. If you choose not to use extension bundles, you also need to install .NET Core 2.x SDK locally and maintain an `extensions.csproj` with your functions project.

In version 2.x and beyond of the Azure Functions runtime, you have to explicitly register the extensions for the binding types used in your functions. You can choose to install binding extensions individually, or you can add an extension bundle reference to the `host.json` project file. Extension bundles removes the chance of having package compatibility issues when using multiple binding types. It is the recommended approach for registering binding extensions. Extension bundles also removes the requirement of installing the .NET Core 2.x SDK.

Use 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:

```
{
    "version": "2.0",
    "extensionBundle": {
        "id": "Microsoft.Azure.Functions.ExtensionBundle",
        "version": "[1.*, 2.0.0)"
    }
}
```

To learn more, see [Register Azure Functions binding extensions](#). You should add extension bundles to the `host.json` before you add bindings to the `function.json` file.

Explicitly install extensions

If you aren't able to use extension bundles, you can use Azure Functions Core Tools locally to install the specific extension packages required by your project.

IMPORTANT

You can't explicitly install extensions in a function app that is using extension bundles. Remove the `extensionBundle` section in `host.json` before explicitly installing extensions.

The following items describe some reasons 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.

NOTE

To manually install extensions by using Core Tools, you must have the [.NET Core 2.x SDK](#) installed. The .NET Core SDK is used by Azure Functions Core Tools to install extensions from NuGet. You don't need to know .NET to use Azure Functions extensions.

When you explicitly install extensions, a .NET project file named `extensions.csproj` is added to the root of your 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 the file.

There are several ways to use Core Tools to install the required extensions in your local project.

Install all extensions

Use the following command to automatically add all extension packages used by the bindings in your local project:

```
func extensions install
```

The command reads the `function.json` file to see which packages you need, installs them, and rebuilds the extensions project (`extensions.csproj`). It adds any new bindings at the current version but does not update existing bindings. Use the `--force` option to update existing bindings to the latest version when installing new ones.

If your function app uses bindings that Core Tools does not recognize, you must manually install the specific extension.

Install a specific extension

Use the following command to install a specific extension package at a specific version, in this case the Storage extension:

```
func extensions install --package Microsoft.Azure.WebJobs.Extensions.Storage --version 4.0.2
```

Local settings file

The `local.settings.json` file stores app settings, connection strings, and settings used by local development tools. Settings in the `local.settings.json` file are used only when you're running projects locally. The local settings file has this structure:

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "<language worker>",
    "AzureWebJobsStorage": "<connection-string>",
    "AzureWebJobsDashboard": "<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:

| SETTING | DESCRIPTION |
|----------------------------|--|
| <code>IsEncrypted</code> | 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 automatically decrypts settings when it runs. Use the <code>func settings decrypt</code> command before trying to read locally encrypted settings. |
| <code>Values</code> | Array of application settings and connection strings used when a project is running locally. These key-value (string-string) pairs correspond to application settings in your function app in Azure, like <code>AzureWebJobsStorage</code> . 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. Values must be strings and not JSON objects or arrays. Setting names can't include a colon (<code>:</code>) or a double underline (<code>__</code>). Double underline characters are reserved by the runtime, and the colon is reserved to support dependency injection . |
| <code>Host</code> | Settings in this section customize the Functions host process when you run projects locally. These settings are separate from the host.json settings, which also apply when you run projects in Azure. |
| <code>LocalHttpPort</code> | 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. |

| SETTING | DESCRIPTION |
|-------------------|---|
| 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. |
| CORScredentials | 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 |
|--|---|--|
| AzureWebJobsStorage | 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 Azure Storage Emulator installed locally and you set <code>AzureWebJobsStorage</code> to <code>UseDevelopmentStorage=true</code> , Core Tools uses the emulator. The emulator is useful during development, but you should test with an actual storage connection before deployment. |
| <code>AzureWebJobs.<FUNCTION_NAME>.Disabled</code> | <code>true</code> <code>false</code> | To disable a function when running locally, add <pre>"AzureWebJobs.<FUNCTION_NAME>.Disabled": "true"</pre> 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 |

| SETTING | VALUES | DESCRIPTION |
|----------------------------------|---|--|
| FUNCTIONS_WORKER_RUNTIME | <input type="button" value="dotnet"/> <input type="button" value="node"/> <input type="button" value="java"/> <input type="button" value="powershell"/> <input type="button" value="python"/> | 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. |
| FUNCTIONS_WORKER_RUNTIME_VERSION | ~7 | Indicates that PowerShell 7 be used when running locally. If not set, then PowerShell Core 6 is used. This setting is only used when running locally. When running in Azure, the PowerShell runtime version is determined by the <code>powershellVersion</code> site configuration setting, which can be set in the portal . |

By default, these settings are not migrated automatically when the project is published to Azure. Use the `--publish-local-settings` switch [when you publish](#) to make sure these settings are added to the function app in Azure. Note that values in **ConnectionStrings** are never published.

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 topics:

- [C# precompiled](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

When no valid storage connection string is set for [AzureWebJobsStorage](#) and the emulator isn't being used, the following error message is shown:

Missing value for AzureWebJobsStorage in local.settings.json. This is required for all triggers other than HTTP. You can run 'func azure functionapp fetch-app-settings <functionAppName>' or specify a connection string in local.settings.json.

Get your storage connection strings

Even when using the Microsoft Azure Storage Emulator for development, you may want to test with an actual storage connection. Assuming you have already [created a storage account](#), you can get a valid storage connection string in one of the following ways:

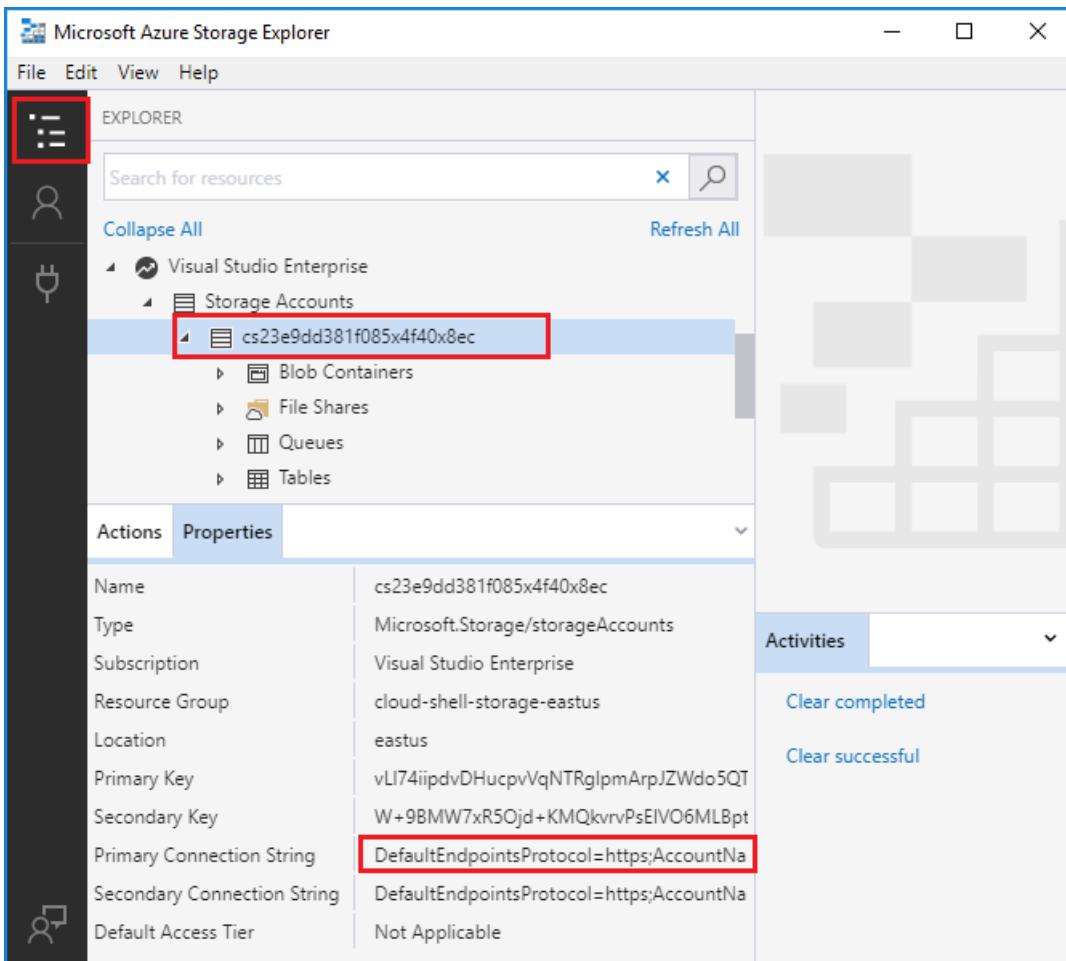
- From the [Azure portal](#), search for and select **Storage accounts**.

The screenshot shows the Microsoft Azure portal's search interface. In the top navigation bar, the search term 'storage accounts' is entered. Below the search bar, the results are displayed under the heading 'Services'. The first result, 'Storage accounts', is highlighted with a red box. Other listed services include Storage accounts (classic), Automation Accounts, Batch accounts, Genomics accounts, Integration accounts, SendGrid Accounts, Storage explorer, Synthetics Accounts, and Azure Maps Accounts.

Select your storage account, select **Access keys** in **Settings**, then copy one of the **Connection string** values.

This screenshot shows the 'Access keys' blade for a specific storage account named 'cs23e9dd381f085x4f40x8ec'. The blade includes sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, Events, Storage Explorer (preview), and Settings. Under the Settings section, the 'Access keys' option is highlighted with a red box. It displays two sets of access keys: 'key1' and 'key2', each with its corresponding key value and connection string.

- Use [Azure Storage Explorer](#) to connect to your Azure account. In the **Explorer**, expand your subscription, expand **Storage Accounts**, select your storage account, and copy the primary or secondary connection string.



- Use Core Tools from the project root to download the connection string from Azure with one of the following commands:
 - Download all settings from an existing function app:

```
func azure functionapp fetch-app-settings <FunctionAppName>
```

- Get the Connection string for a specific storage account:

```
func azure storage fetch-connection-string <StorageAccountName>
```

When you aren't already signed in to Azure, you're prompted to do so. These commands overwrite any existing settings in the local.settings.json file.

Create a function

To create a function, run the following command:

```
func new
```

In version 3.x/2.x, when you run `func new` you are prompted to choose a template in the default language of your function app, then you are also prompted to choose a name for your function. In version 1.x, you are also prompted to choose the language.

```
Select a language: Select a template:  
Blob trigger  
Cosmos DB trigger  
Event Grid trigger  
HTTP trigger  
Queue trigger  
SendGrid  
Service Bus Queue trigger  
Service Bus Topic trigger  
Timer trigger
```

Function code is generated in a subfolder with the provided function name, as you can see in the following queue trigger output:

```
Select a language: Select a template: Queue trigger  
Function name: [QueueTriggerJS] MyQueueTrigger  
Writing C:\myfunctions\myMyFunctionProj\MyQueueTrigger\index.js  
Writing C:\myfunctions\myMyFunctionProj\MyQueueTrigger\readme.md  
Writing C:\myfunctions\myMyFunctionProj\MyQueueTrigger\sample.dat  
Writing C:\myfunctions\myMyFunctionProj\MyQueueTrigger\function.json
```

You can also specify these options in the command using the following arguments:

| ARGUMENT | DESCRIPTION |
|---|---|
| <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> , <code>-l</code> | 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, do not use this option or choose a language that matches the worker runtime. |
| <code>--name</code> , <code>-n</code> | The function name. |
| <code>--template</code> , <code>-t</code> | Use the <code>func templates list</code> command to see the complete list of available templates for each supported language. |

For example, to create a JavaScript HTTP trigger in a single command, run:

```
func new --template "Http Trigger" --name MyHttpTrigger
```

To create a queue-triggered function in a single command, run:

```
func new --template "Queue Trigger" --name QueueTriggerJS
```

Run functions locally

To run a Functions project, run the Functions host. The host enables triggers for all functions in the project. The start command varies, depending on your project language.

- C#

- [Java](#)
- [JavaScript](#)
- [Python](#)
- [TypeScript](#)

```
func start --build
```

NOTE

Version 1.x of the Functions runtime requires the `host` command, as in the following example:

```
func host start
```

`func start` supports the following options:

| OPTION | DESCRIPTION |
|---------------------------------------|---|
| <code>--no-build</code> | Do no build current project before running. For dotnet projects only. Default is set to false. Not supported for version 1.x. |
| <code>--cors-credentials</code> | Allow cross-origin authenticated requests (i.e. cookies and the Authentication header) Not supported for version 1.x. |
| <code>--cors</code> | A comma-separated list of CORS origins, with no spaces. |
| <code>--language-worker</code> | Arguments to configure the language worker. For example, you may enable debugging for language worker by providing debug port and other required arguments . Not supported for version 1.x. |
| <code>--cert</code> | The path to a .pfx file that contains a private key. Only used with <code>--useHttps</code> . Not supported for version 1.x. |
| <code>--password</code> | Either the password or a file that contains the password for a .pfx file. Only used with <code>--cert</code> . Not supported for version 1.x. |
| <code>--port</code> , <code>-p</code> | The local port to listen on. Default value: 7071. |
| <code>--pause-on-error</code> | Pause for additional input before exiting the process. Used only when launching Core Tools from an integrated development environment (IDE). |

| OPTION | DESCRIPTION |
|--------------------------|--|
| --script-root , --prefix | Used to specify the path to the root of the function app that is to be run or deployed. This is used for compiled projects that generate project files into a subfolder. For example, when you build a C# class library project, the host.json, local.settings.json, and function.json files are generated in a <i>root</i> subfolder with a path like MyProject/bin/Debug/netstandard2.0. In this case, set the prefix as --script-root MyProject/bin/Debug/netstandard2.0. This is the root of the function app when running in Azure. |
| --timeout , -t | The timeout for the Functions host to start, in seconds. Default: 20 seconds. |
| --useHttps | Bind to https://localhost:{port} rather than to http://localhost:{port}. By default, this option creates a trusted certificate on your computer. |

When the Functions host starts, it outputs the URL of HTTP-triggered functions:

```
Found the following functions:
Host.Functions.MyHttpTrigger

Job host started
Http Function MyHttpTrigger: http://localhost:7071/api/MyHttpTrigger
```

IMPORTANT

When running locally, authorization isn't enforced for HTTP endpoints. This means that all local HTTP requests are handled as authLevel = "anonymous". For more information, see the [HTTP binding article](#).

Passing test data to a function

To test your functions locally, you [start the Functions host](#) and call endpoints on the local server using HTTP requests. The endpoint you call depends on the type of function.

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](#).

For more general information on testing functions, see [Strategies for testing your code in Azure Functions](#).

HTTP and webhook triggered functions

You call the following endpoint to locally run HTTP and webhook triggered functions:

```
http://localhost:{port}/api/{function_name}
```

Make sure to use the same server name and port that the Functions host is listening on. You see this in the output generated when starting the Function host. You can call this URL using any HTTP method supported

by the trigger.

The following 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
```

The following example is the same function called from a POST request passing `name` in the request body:

- [Bash](#)
- [Cmd](#)

```
curl --request POST http://localhost:7071/api/MyHttpTrigger --data '{"name":"Azure Rocks"}'
```

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.

Non-HTTP triggered functions

For all kinds of functions other than HTTP triggers and webhooks and Event Grid triggers, you can test your functions locally by calling an administration endpoint. Calling this endpoint with an HTTP POST request on the local server triggers the function.

To test Event Grid triggered functions locally, see [Local testing with viewer web app](#).

You can optionally pass test data to the execution in the body of the POST request. This functionality is similar to the **Test** tab in the Azure portal.

You call the following administrator endpoint to trigger non-HTTP functions:

```
http://localhost:{port}/admin/functions/{function_name}
```

To pass test data to the administrator endpoint of a function, you must supply the data in the body of a POST request message. The message body is required to have the following JSON format:

```
{
  "input": "<trigger_input>"
}
```

The `<trigger_input>` value contains data in a format expected by the function. The following cURL example is a POST to a `QueueTriggerJS` function. In this case, the input is a string that is equivalent to the message expected to be found in the queue.

- [Bash](#)
- [Cmd](#)

```
curl --request POST -H "Content-Type:application/json" --data '{"input":"sample queue data"}'
http://localhost:7071/admin/functions/QueueTrigger
```

Using the `func run` command (version 1.x only)

IMPORTANT

The `func run` command is only supported in version 1.x of the tools. For more information, see the topic [How to target Azure Functions runtime versions](#).

In version 1.x, you can also invoke a function directly by using `func run <FunctionName>` and provide input data for the function. This command is similar to running a function using the **Test** tab in the Azure portal.

`func run` supports the following options:

| OPTION | DESCRIPTION |
|--|--|
| <code>--content</code> , <code>-c</code> | Inline content. |
| <code>--debug</code> , <code>-d</code> | Attach a debugger to the host process before running the function. |
| <code>--timeout</code> , <code>-t</code> | Time to wait (in seconds) until the local Functions host is ready. |
| <code>--file</code> , <code>-f</code> | The file name to use as content. |
| <code>--no-interactive</code> | Does not prompt for input. Useful for automation scenarios. |

For example, to call an HTTP-triggered function and pass content body, run the following command:

```
func run MyHttpTrigger -c '{"name": "Azure"}'
```

Publish to Azure

The Azure Functions Core Tools supports two types of deployment: deploying function project files directly to your function app via [Zip Deploy](#) and [deploying a custom Docker container](#). You must have already [created a function app in your Azure subscription](#), to which you'll deploy your code. Projects that require compilation should be built so that the binaries can be deployed.

IMPORTANT

You must have the [Azure CLI](#) installed locally to be able to publish to Azure from Core Tools.

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.

Deploy project files

To publish your local code to a function app in Azure, use the `publish` command:

```
func azure functionapp publish <FunctionAppName>
```

IMPORTANT

Java uses Maven to publish your local project to Azure. Use the following command to publish to Azure:

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

Azure resources are created during initial deployment.

This command publishes to an existing function app in Azure. You'll get an error if you try to publish to a `<FunctionAppName>` that doesn't exist in your subscription. To learn how to create a function app from the command prompt or terminal window using the Azure CLI, see [Create a Function App for serverless execution](#). By default, this command uses [remote build](#) and deploys your app to run from the deployment package. To disable this recommended deployment mode, use the `--nozip` option.

IMPORTANT

When you create a function app in the Azure portal, it uses version 3.x of the Function runtime by default. To make the function app use version 1.x of the runtime, follow the instructions in [Run on version 1.x](#). You can't change the runtime version for a function app that has existing functions.

The following publish options apply for all versions:

| OPTION | DESCRIPTION |
|--|---|
| <code>--publish-local-settings -i</code> | Publish settings in local.settings.json to Azure, prompting to overwrite if the setting already exists. If you are using the Microsoft Azure Storage Emulator, first change the app setting to an actual storage connection . |
| <code>--overwrite-settings -y</code> | Suppress the prompt to overwrite app settings when <code>--publish-local-settings -i</code> is used. |

The following publish options are supported only for version 2.x and later versions:

| OPTION | DESCRIPTION |
|--|---|
| <code>--publish-settings-only</code> , <code>-o</code> | Only publish settings and skip the content. Default is prompt. |
| <code>--list-ignored-files</code> | Displays a list of files that are ignored during publishing, which is based on the .funcignore file. |
| <code>--list-included-files</code> | Displays a list of files that are published, which is based on the .funcignore file. |
| <code>--nozip</code> | Turns the default <code>Run-From-Package</code> mode off. |
| <code>--build-native-deps</code> | Skips generating .wheels folder when publishing Python function apps. |
| <code>--build</code> , <code>-b</code> | Performs build action when deploying to a Linux function app. Accepts: <code>remote</code> and <code>local</code> . |
| <code>--additional-packages</code> | List of packages to install when building native dependencies. For example: <code>python3-dev libevent-dev</code> . |

| OPTION | DESCRIPTION |
|---------------------|--|
| --force | Ignore pre-publishing verification in certain scenarios. |
| --csx | Publish a C# script (.csx) project. |
| --no-build | Project isn't built during publishing. For Python, <code>pip install</code> isn't performed. |
| --dotnet-cli-params | When publishing compiled C# (.csproj) functions, the core tools calls 'dotnet build --output bin/publish'. Any parameters passed to this will be appended to the command line. |

Deploy custom container

Azure Functions lets you deploy your function project in a [custom Docker container](#). For more information, see [Create a function on Linux using a custom image](#). Custom containers must have a Dockerfile. To create an app with a Dockerfile, use the --dockerfile option on `func init`.

```
func deploy
```

The following custom container deployment options are available:

| OPTION | DESCRIPTION |
|------------|--|
| --registry | The name of a Docker Registry the current user signed-in to. |
| --platform | Hosting platform for the function app. Valid options are <code>kubernetes</code> |
| --name | Function app name. |
| --max | Optionally, sets the maximum number of function app instances to deploy to. |
| --min | Optionally, sets the minimum number of function app instances to deploy to. |
| --config | Sets an optional deployment configuration file. |

Monitoring functions

The recommended way to monitor the execution of your functions is by integrating with Azure Application Insights. You can also stream execution logs to your local computer. To learn more, see [Monitor Azure Functions](#).

Application Insights integration

Application Insights integration should be enabled when you create your function app in Azure. If for some reason your function app isn't connected to an Application Insights instance, it's easy to do this integration in the Azure portal. To learn more, see [Enable Application Insights integration](#).

Enable streaming logs

You can view a stream of log files being generated by your functions in a command-line session on your local computer.

Built-in log streaming

Use the `logstream` option to start receiving streaming logs of a specific function app running in Azure, as in the following example:

```
func azure functionapp logstream <FunctionAppName>
```

NOTE

Built-in log streaming isn't yet enabled in Core Tools for function apps running on Linux in a Consumption plan. For these hosting plans, you instead need to use Live Metrics Stream to view the logs in near-real time.

Live Metrics Stream

You can view the [Live Metrics Stream](#) for your function app in a new browser window by including the `--browser` option, as in the following example:

```
func azure functionapp logstream <FunctionAppName> --browser
```

This type of streaming logs requires that Application Insights integration be enabled for your function app.

Next steps

Learn how to develop, test, and publish Azure Functions by using Azure Functions Core Tools [Microsoft learn module](#). 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

12/4/2020 • 5 minutes to read • [Edit Online](#)

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.

We recommend that you [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 and language:

| VISUAL STUDIO CODE | TERMINAL/COMMAND PROMPT | VISUAL STUDIO |
|---|---|-------------------------------------|
| <ul style="list-style-type: none">Get started with C#Get started with JavaGet started with JavaScriptGet started with PowerShellGet started with Python | <ul style="list-style-type: none">Get started with C#Get started with JavaGet started with JavaScriptGet started with PowerShellGet started with Python | Get started with C# |

If you don't have an [Azure subscription](#), create a [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.

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. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------|--------------------|---|
| Runtime stack | Preferred language | Choose a runtime that supports your favorite function programming language. Choose .NET Core for C# and F# functions. |
| 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. |

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 * | <input type="text" value="Visual Studio Enterprise"/> |
| Resource Group * | <input type="text" value="(New) myResourceGroup"/> Create new |

Instance Details

| | |
|---------------------|--|
| Function App name * | <input type="text" value="myfunctionapp"/> .azurewebsites.net |
| Publish * | Code Docker Container |
| Runtime stack * | <input type="text" value=".NET Core"/> |
| Version * | <input type="text" value="3.1"/> |
| Region * | <input type="text" value="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. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|--------------------------|--|
| Plan | 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. When you run in an App Service plan, you must manage the scaling of your function app . |

Function App X

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 * ▼
[Create new](#)

Operating system
The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux Windows

Plan
The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#) ⓘ

Plan type * ⓘ ▼

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. |

Function App

Basics Hosting Monitoring Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

Create new

Region

Central US

Review + create < Previous Next : Tags >

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.

Notifications

More events in the activity log → Dismiss all

Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource Pin to dashboard

a few seconds ago

Next, create a function in the new 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**.

The screenshot shows the Azure Functions blade for a function app named 'myfunctionapp'. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (selected), App keys, App files, Proxies, Deployment, and Deployment slots. At the top right, there's a search bar, a '+ Add' button, and a 'Filter by' dropdown. Below the sidebar, there's a 'Name' input field with 'No results.' message. The main area is titled 'New Function' with the sub-header 'Create a new function in this Function App. Start by selecting a template below.' It shows four templates: 'HTTP trigger' (selected and highlighted with a red box), 'Timer trigger', 'Azure Queue Storage trigger', and 'Azure Service Bus Queue trigger'. Each template has a small icon and a brief description.

3. In the **New Function** window, accept the default name for **New Function**, or enter a new name.
4. Choose **Anonymous** from the **Authorization level** drop-down list, and then select **Create Function**.
Azure creates the HTTP trigger function. Now, you can run the new function by sending an HTTP request.

Test the function

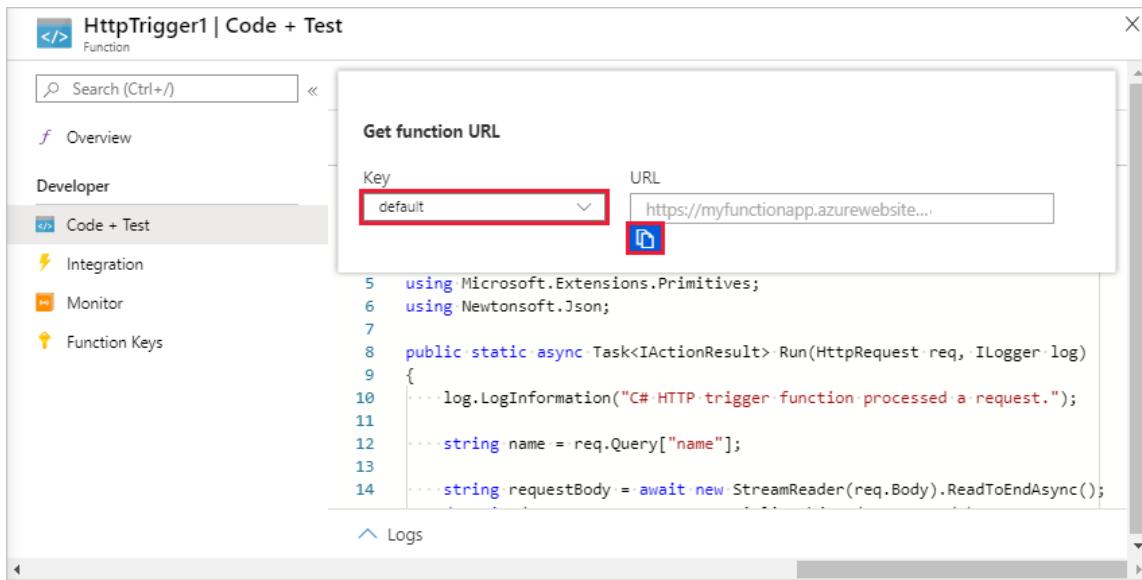
1. In your new HTTP trigger function, select **Code + Test** from the left menu, then select **Get function URL** from the top menu.

The screenshot shows the 'HttpTrigger1 | Code + Test' blade. On the left, there's a sidebar with links for Overview, Developer (selected and highlighted with a red box), Code + Test (selected), Integration, Monitor, and Function Keys. At the top right, there's a search bar, a 'Refresh' button, a 'Save' button, a 'Discard' button, a 'Test' button, and a 'Get function URL' button (which is also highlighted with a red box). The main area shows the function code in a code editor:

```
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(HttpContext req, ILogger log)
9 {
10     ....log.LogInformation("C# HTTP trigger function processed a request.");
11
12     string name = req.Query["name"];
13
14     string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
```

Below the code editor, there's a 'Logs' section.

2. In the **Get function URL** dialog box, select **default** from the drop-down list, and then select the **Copy to clipboard** icon.



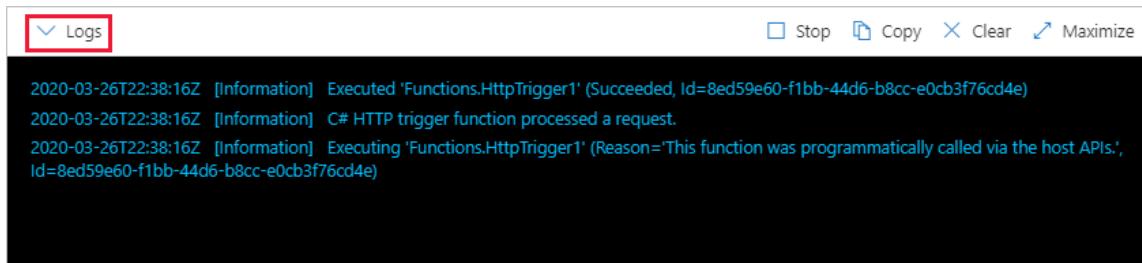
- Paste the function URL into your browser's address bar. Add the query string value `?name=<your_name>` to the end of this URL and press Enter to run the request.

The following example shows the response in the browser:



The request URL includes a key that is required, by default, to access your function over HTTP.

- 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.



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 have created in this quickstart, do not 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 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:

- 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's App Service overview for a resource named 'myfunctionapp'. The left sidebar lists various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions, Functions, App keys, App files, and Proxies. The main content area displays details about the app, including its status (Running), location (Central US), subscription (Visual Studio Enterprise), and runtime version (3.0.13139.0). A red box highlights the 'Resource group (change)' section, which shows 'myResourceGroup'. 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 quickstart.

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 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

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](#)

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

12/4/2020 • 7 minutes to read • [Edit Online](#)

In this article, you use command-line tools to create a C# class library-based 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](#).
- The [.NET Core SDK 3.1](#)
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - [Azure PowerShell](#) version 5.0 or later.

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
- [Azure PowerShell](#)
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
- Run `az --version` to check that the Azure CLI version is 2.4 or later.
- Run `az login` to sign in to Azure and verify an active subscription.
- Run `dotnet --list-sdks` to check that .NET Core SDK version 3.1.x is installed

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:

```
func init LocalFunctionProj --dotnet
```

2. Navigate into the project folder:

```
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).

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

`func new` creates a 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 is an [HttpRequest](#) that's decorated with the [HttpTriggerAttribute](#), which defines the trigger behavior.

```
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 LocalFunctionProj
{
    public static class HttpExample
    {
        [FunctionName("HttpExample")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "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;

            return name != null
                ? (ActionResult)new OkObjectResult($"Hello, {name}")
                : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
        }
    }
}
```

The return object is an [ActionResult](#) that returns a response message as either an [OkObjectResult](#) (200) or a [BadRequestObjectResult](#) (400). 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.
```

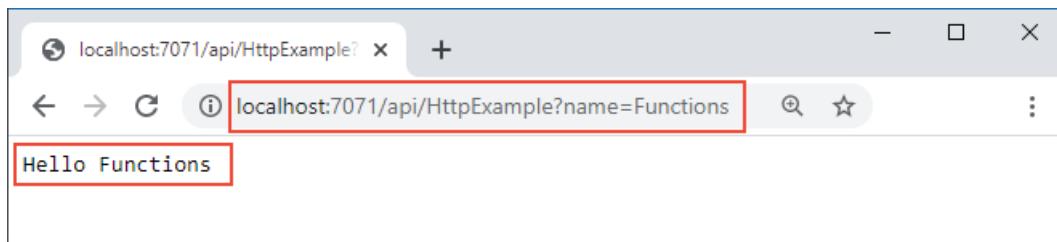
```
Http Functions:
```

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

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.
4. 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 maintains 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 you haven't done so already, sign in to Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

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

2. Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The `az group create` command creates a resource group. You generally create your resource group and resources in a region near you, using an available region returned from the `az account list-locations` command.

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

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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](#).

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location
wsteurope --runtime dotnet --functions-version 3 --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.

```
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.

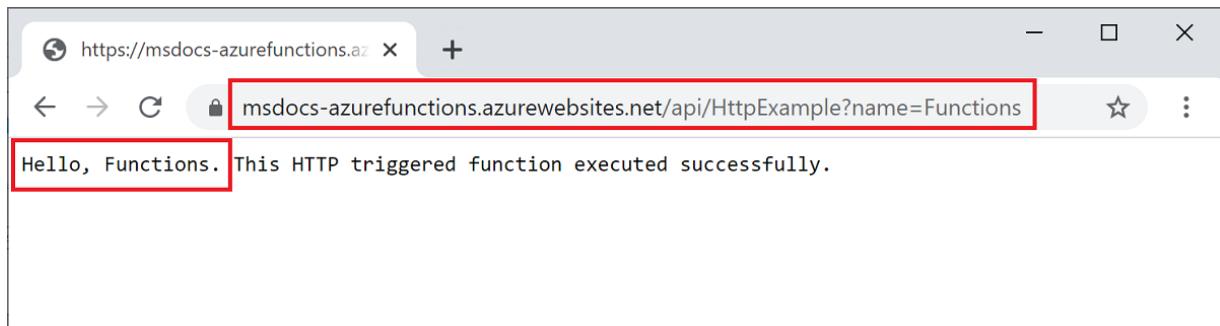
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
    HttpExample - [httpTrigger]
        Invoke url: https://msdocs-azurefunctions-qs.azurewebsites.net/api/httpexample
```

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](#)
- [curl](#)

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](#):

```
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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

Quickstart: Create your first function in Azure using Visual Studio

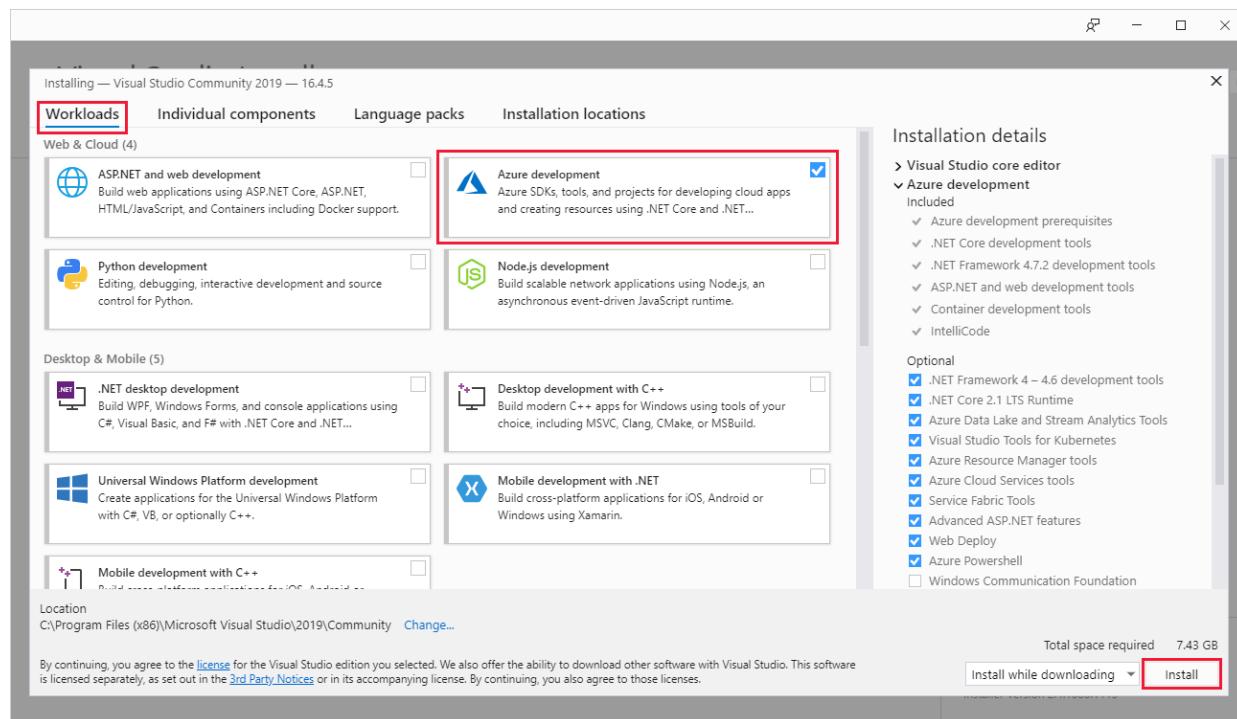
12/4/2020 • 7 minutes to read • [Edit Online](#)

In this article, you use Visual Studio to create a C# class library-based 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.

Prerequisites

To complete this tutorial, first install [Visual Studio 2019](#). Ensure you select the **Azure development** workload during installation. If you want to create an Azure Functions project by using Visual Studio 2017 instead, you must first install the [latest Azure Functions tools](#).



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

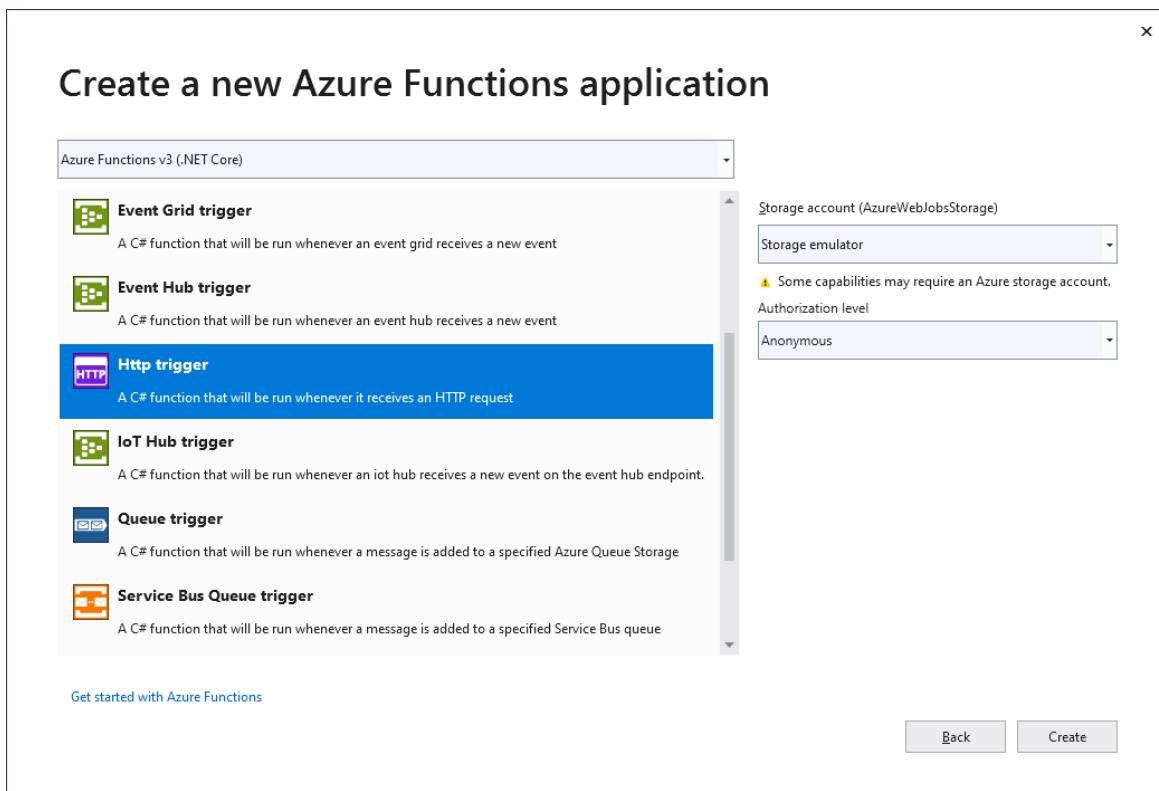
Create a function app project

The Azure Functions project template in Visual Studio creates a 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:

| SETTING | VALUE | DESCRIPTION |
|---------------------------------------|--------------------------------|--|
| Functions runtime | Azure Functions v3 (.NET Core) | This value creates a function project that uses the version 3.x runtime of Azure Functions, which supports .NET Core 3.x. Azure Functions 1.x supports the .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 an Azure Function 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.

- 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 `FunctionName` 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.

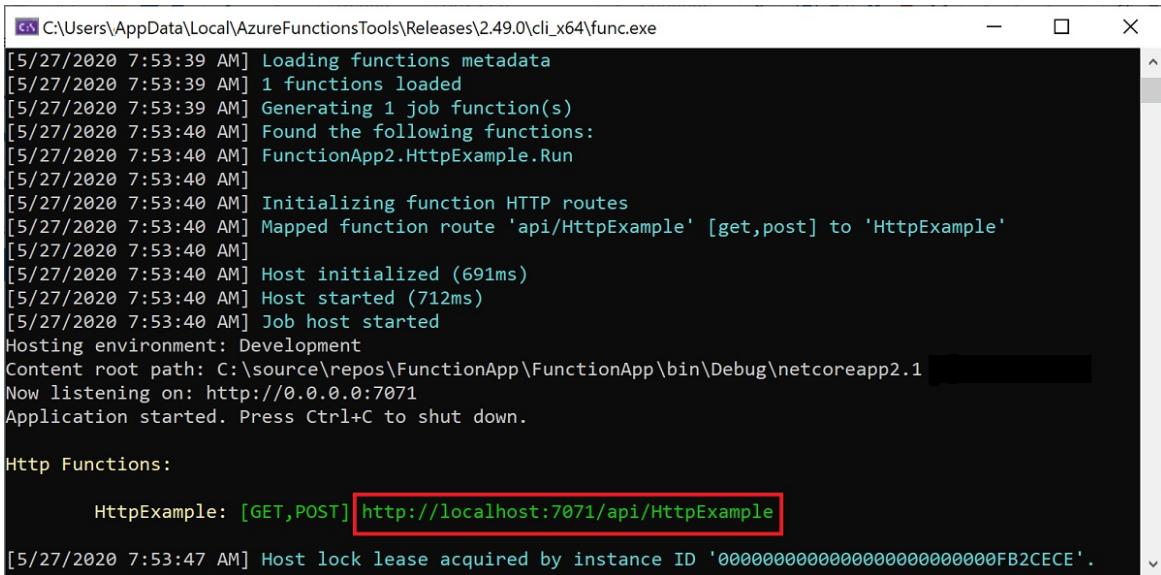
- In **File Explorer**, right-click the `Function1.cs` file and rename it to `HttpExample.cs`.
- In the code, rename the `Function1` class to 'HttpExample'.
- In the `HttpTrigger` method named `Run`, rename the `FunctionName` method attribute to `HttpExample`.

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.

- 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.
- Copy the URL of your function from the Azure Functions runtime output.

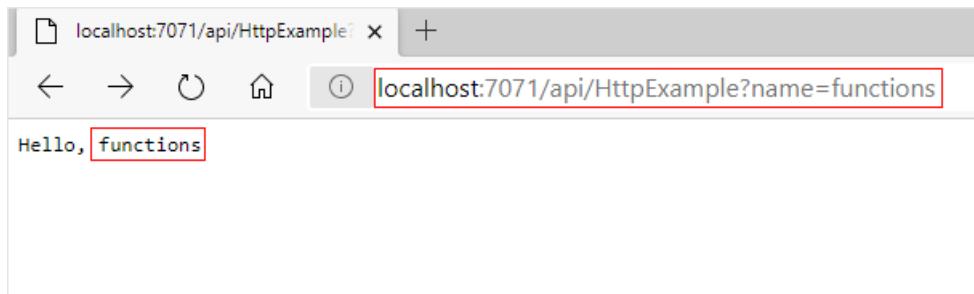


```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.49.0\cli_x64\func.exe
[5/27/2020 7:53:39 AM] Loading functions metadata
[5/27/2020 7:53:39 AM] 1 functions loaded
[5/27/2020 7:53:39 AM] Generating 1 job function(s)
[5/27/2020 7:53:40 AM] Found the following functions:
[5/27/2020 7:53:40 AM] 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'.
```

- Paste the URL for the HTTP request into your browser's address bar. Append the query string `?name=<YOUR_NAME>` to this URL and run the request. The following image shows the response in the browser to the local GET request returned by the function:



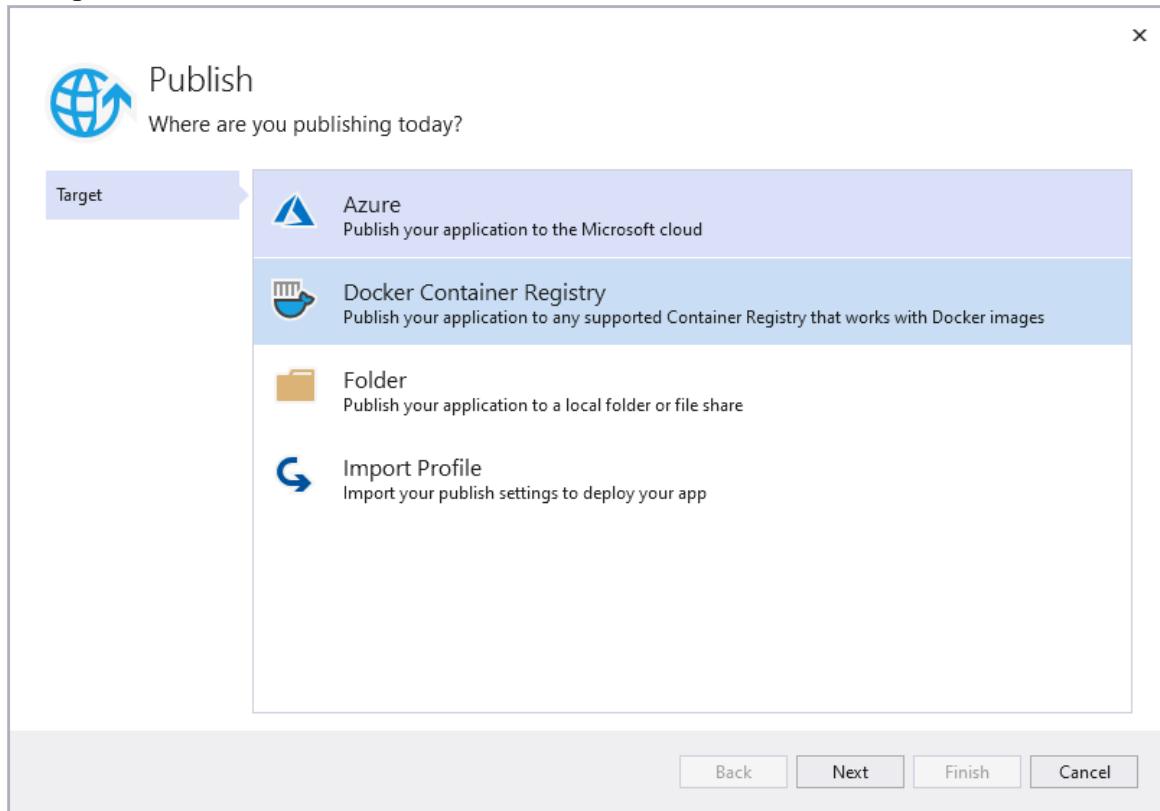
4. To stop debugging, press Shift+F5 in Visual Studio.

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

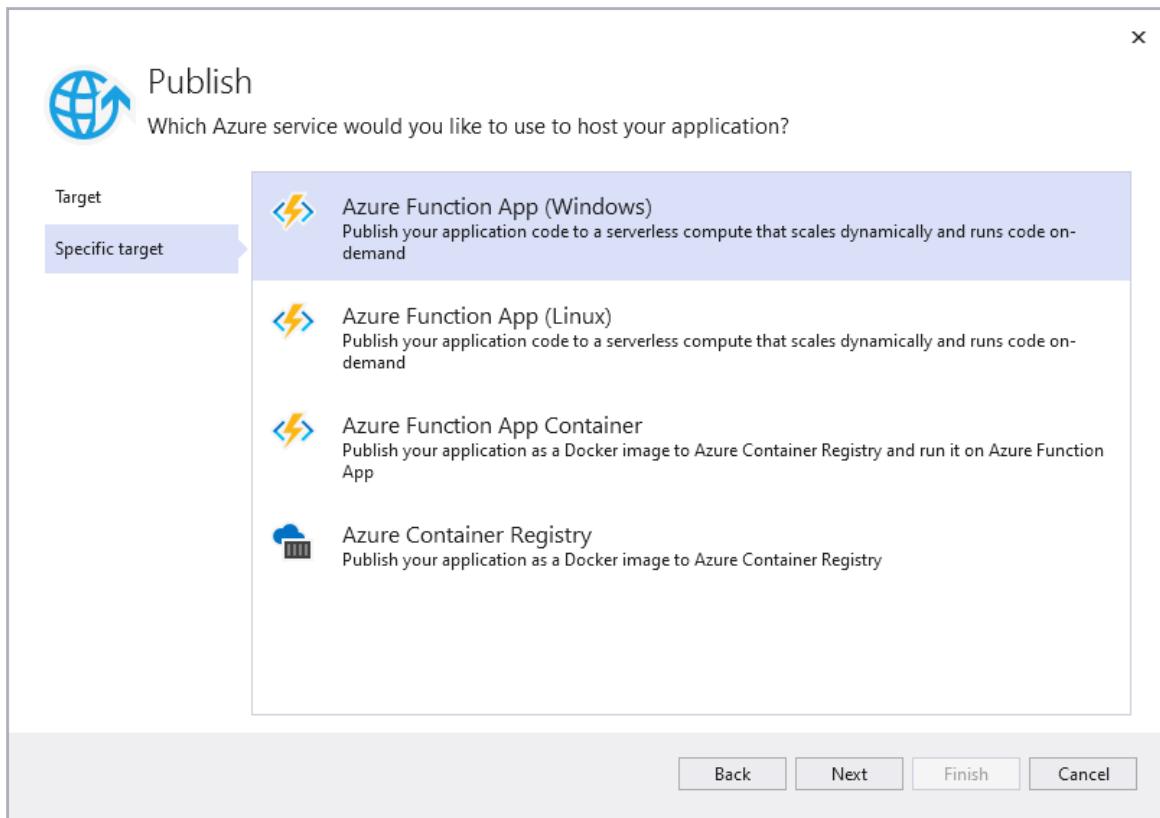
Publish the project to Azure

Before you can publish your project, you must have a function app in your Azure subscription. Visual Studio publishing creates a function app for you the first time you publish your project.

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



3. In **Specific target**, select **Azure Function App (Windows)**



4. In **Function Instance**, select **Create a new Azure Function...** and then use the values specified in the following 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: <code>a-z</code> , <code>0-9</code> , and <code>-</code> . |
| 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 app service | Choose a Location in a region near you or other services your functions access. |

| SETTING | VALUE | DESCRIPTION |
|---------------|---------------------------------|--|
| 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 an existing account that meets the storage account requirements . |

Function App (Windows)

Name: FunctionApp220200930124040

Subscription: Visual Studio Enterprise

Resource group: azure-docs-first-function* [New...](#)

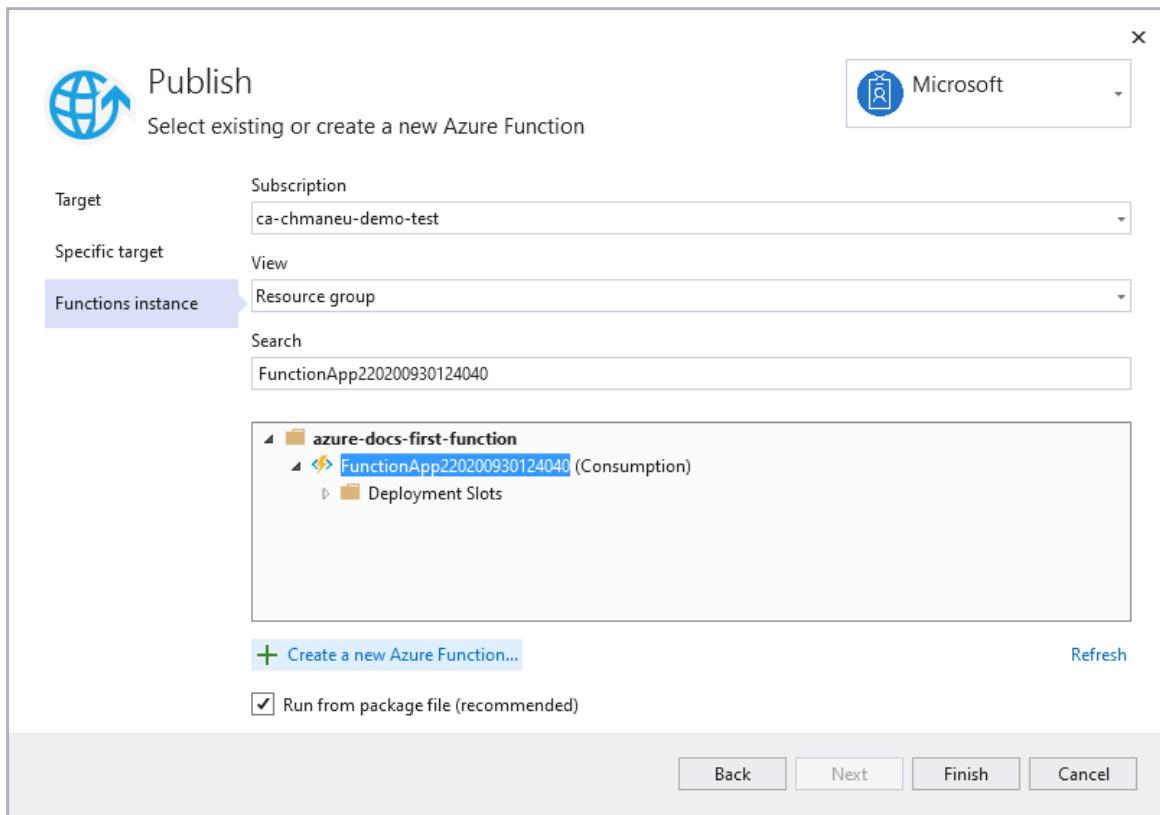
Plan Type: Consumption

Location: France Central

Azure Storage: functiondemochris* (France Central) [New...](#)

Buttons: Export..., Create, Cancel

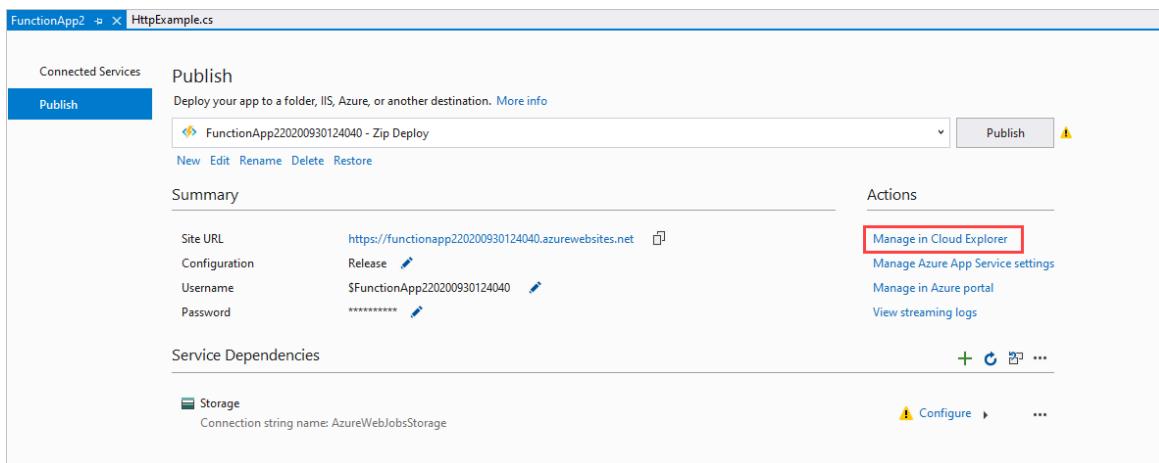
- Select **Create** to create a function app and its related resources in Azure.
- In the **Functions instance**, make sure that **Run from package file** is checked. Your function app is deployed using [Zip Deploy](#) with [Run-From-Package](#) mode enabled. This is the recommended deployment method for your functions project, since it results 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.

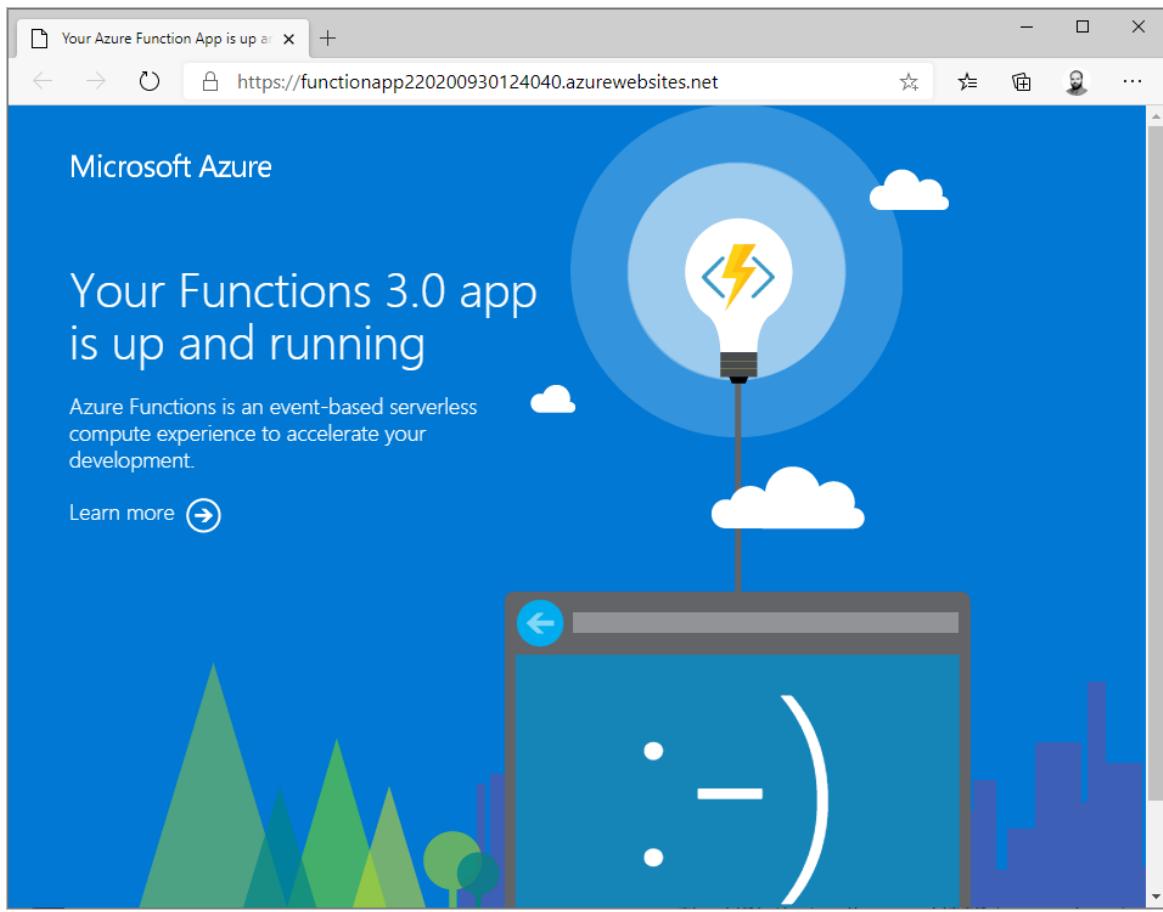
8. In the Publish tab, choose **Manage in Cloud Explorer**. This opens the new function app Azure resource in Cloud Explorer.



Cloud Explorer lets you use Visual Studio to view the contents of the site, start and stop the function app, and browse directly to function app resources on Azure and in the Azure portal.

Test your function in Azure

1. In Cloud Explorer, your new function app should be selected. If not, expand your subscription > **App Services**, and select your new function app.
2. Right-click the function app and choose **Open in Browser**. This opens the root of your function app in your default web browser and displays the page that indicates your function app is running.

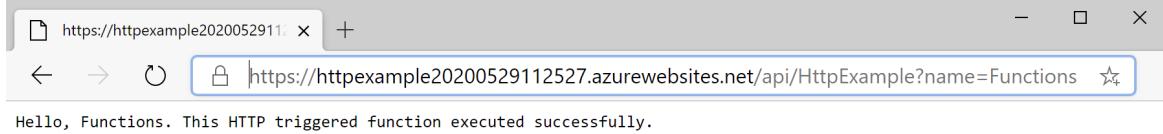


3. In the address bar in the browser, append the string `/api/HttpExample?name=Functions` to the base URL and run the request.

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

```
http://<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

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 have created in this quickstart, do not 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 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 the Cloud Explorer, expand your subscription > **App Services**, right-click your function app, and choose **Open in Portal**.

2. In 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 links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (selected), App keys, App files, and Proxies. The main area shows the URL as https://myfunctionapp.azurewebsites.net, Operating System as Windows, App Service Plan as ASP-myResourceGroup-a285 (Y1: 0), and Runtime version as 3.0.13139.0. The Resource group section is highlighted with a red box, showing 'myResourceGroup' with a status of Running and location Central US. The Metrics tab is selected at the bottom.

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.

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](#)

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

12/4/2020 • 6 minutes to read • [Edit Online](#)

In this article, you use Visual Studio Code to create a C# class library-based 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

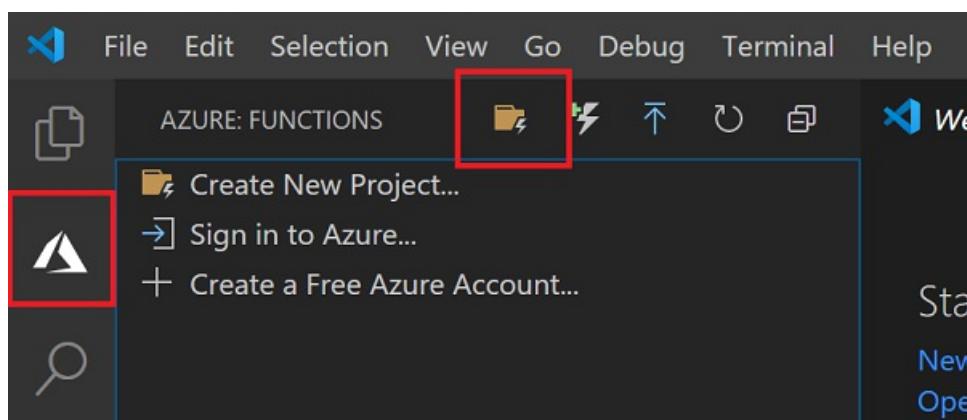
Before you get started, make sure you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- [Node.js](#), required by Windows for npm. Only [Active LTS and Maintenance LTS versions](#). Use the `node --version` command to check your version. Not required for local development on macOS and Linux.
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [C# extension](#) for Visual Studio Code.
- The [Azure Functions extension](#) for Visual Studio Code.

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. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select the **Create new project...** icon.



2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

- Provide the following information at the prompts:
 - Select a language for your function project: Choose `C#`.
 - Select a template for your project's first function: Choose `HTTP trigger`.
 - Provide a function name: Type `HttpExample`.
 - Provide a namespace: Type `My.Functions`.
 - Authorization level: Choose `Anonymous`, which enables anyone to call your function endpoint. To learn about authorization level, see [Authorization keys](#).
 - Select how you would like to open your project: Choose `Add to workspace`.
 - 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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
 2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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 2: Task - host start + □ ^ ×  
Application started. Press Ctrl+C to shut down.  
  
Http Functions:  
  
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample  
  
[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491  
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '0000000000000000000000000000FB2CECE'.  
[1/24/2020 8:47:18 AM] Debugger attached.
```

3. 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>

4. A response is returned, which looks like the following in a browser:



5. 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 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] }
```

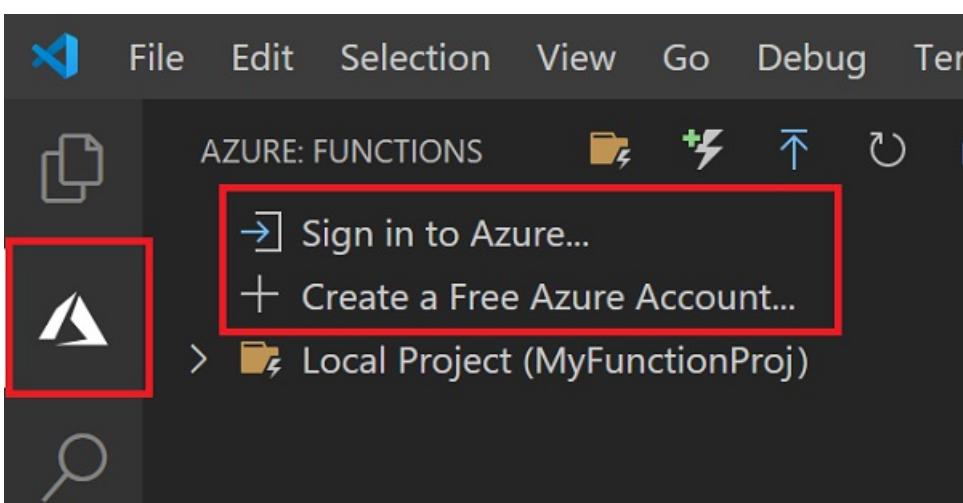
6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

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 publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure**.... If you don't already have one, you can [Create a free Azure account](#). Students can [create a free Azure account for Students](#).



If you're already signed in, go to the next section.

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

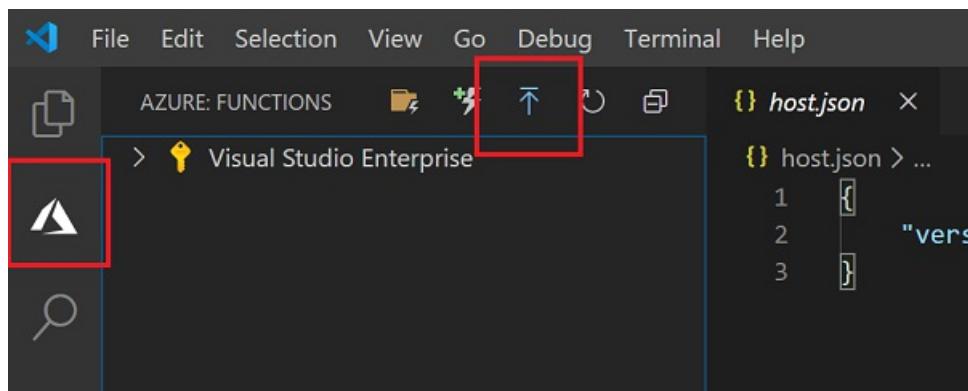
Publish the project to Azure

In this section, you create a function app and related resources in your Azure subscription and then deploy your code.

IMPORTANT

Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to function app...** button.



2. Provide the following information at the prompts:

- **Select folder:** Choose a folder from your workspace or browse to one that contains your function app. You won't see this if you already have a valid function app opened.
- **Select subscription:** Choose the subscription to use. You won't see this if you only have one subscription.
- **Select Function App in Azure:** Choose . (Don't choose the option, which isn't covered in this article.)
- **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.

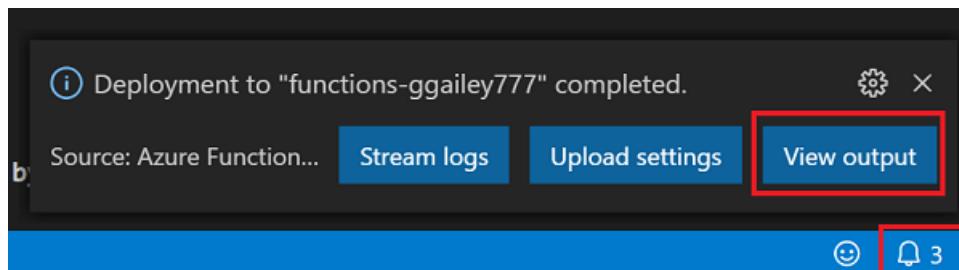
3. When completed, the following Azure resources are created in your subscription, using names based on your function app name:

- 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 consumption plan, which defines the underlying host for your serverless function app.
- 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 Application Insights instance connected to the function app, which tracks usage of your

serverless function.

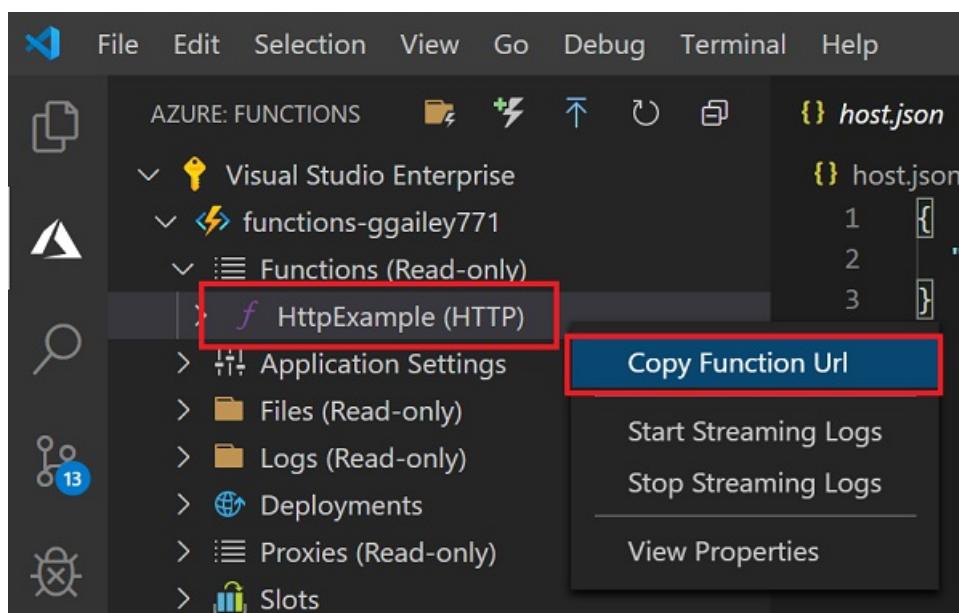
A notification is displayed after your function app is created and the deployment package is applied.

4. Select **View Output** in this notification 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 **Azure: Functions** area in the side bar, expand the new function app under your subscription. Expand **Functions**, right-click (Windows) or Ctrl + click (macOS) on **HttpExample**, and then choose **Copy function URL**.



2. Paste this URL for the HTTP request into your browser's address bar, add the `name` query string as `?name=Functions` to the end of this URL, and then execute the request. The URL that calls your HTTP-triggered function should be in the following format:

```
http://<FUNCTION_APP_NAME>.azurewebsites.net/api/httpexample?name=Functions
```

The following example shows the response in the browser to the remote GET request returned by the function:



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 Functions: 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**.

A screenshot of the Azure portal's Overview page for a function app named "myfunctionapp". The left sidebar has a "Functions" section with items like "Functions", "App keys", "App files", and "Proxies". The main content area shows the "Overview" tab selected. It displays the "Resource group (change)" section with "myResourceGroup" highlighted in blue. To the right, there are details: 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), and Runtime version (3.0.13139.0). At the bottom, there are links for Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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.

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. In the next article, you expand that function by adding an output binding. This binding writes the string from the HTTP request to a message in an Azure Queue Storage queue.

[Connect to an Azure Storage queue](#)

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

12/4/2020 • 3 minutes to read • [Edit Online](#)

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
- [Azure CLI](#)
- [Azure Functions Core Tools](#) version 2.6.666 or above
- [Gradle](#), version 4.10 and above

You also need an active Azure subscription. If you don't have an [Azure subscription](#), create a [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:

```
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.

```
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:

```
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:

```
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 [function 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.

```
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.

```
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 *appRegion*. 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.

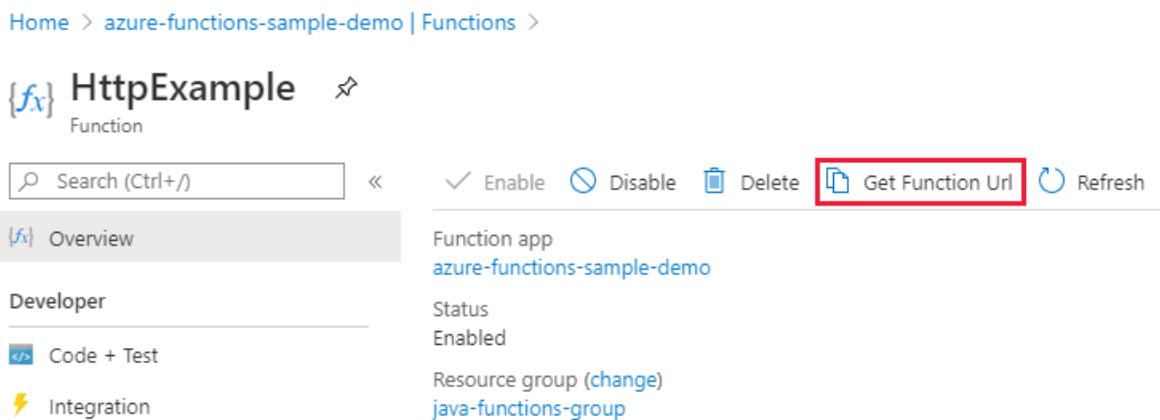
The authLevel for HTTP Trigger in sample project is `ANONYMOUS`, which will skip the authentication. However, if you use other authLevel like `FUNCTION` or `ADMIN`, you need to get the function key to call the function endpoint over HTTP. The easiest way to get the function key is from the [Azure portal](#).

[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.



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.

```
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](#)

Create your first function with Java and Eclipse

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 a [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](#)

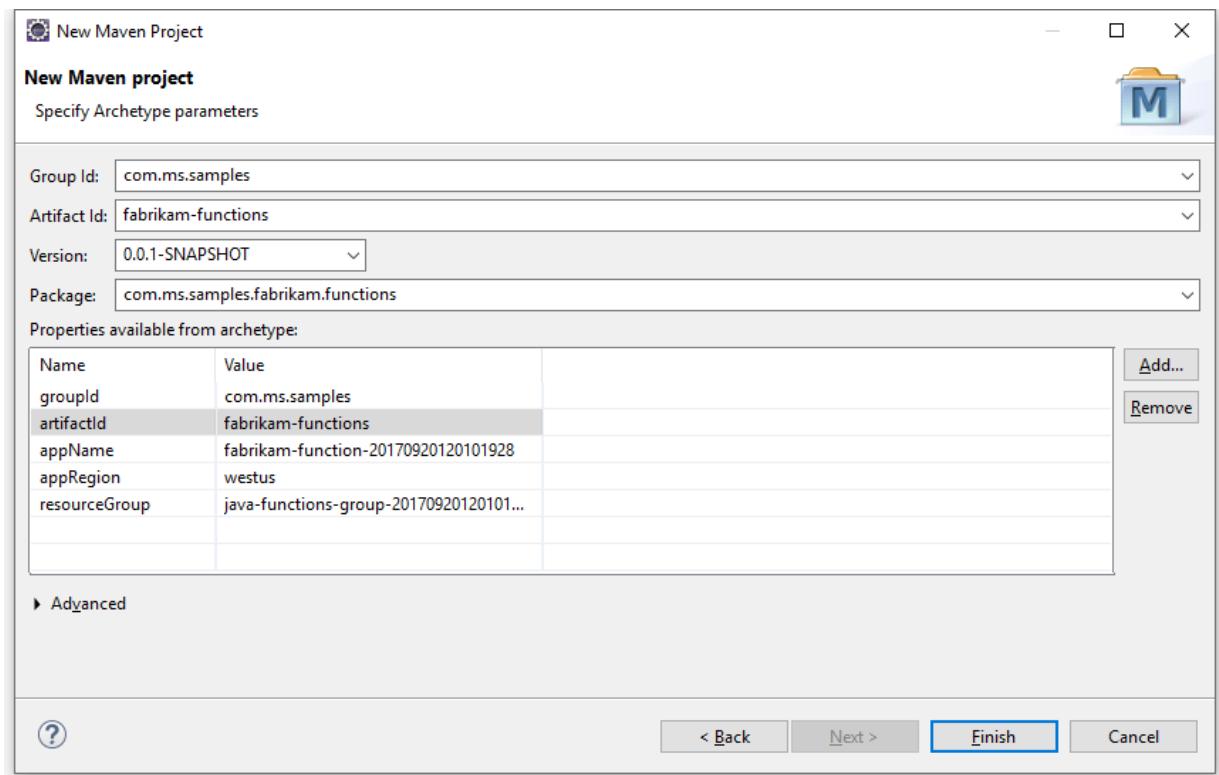
IMPORTANT

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.

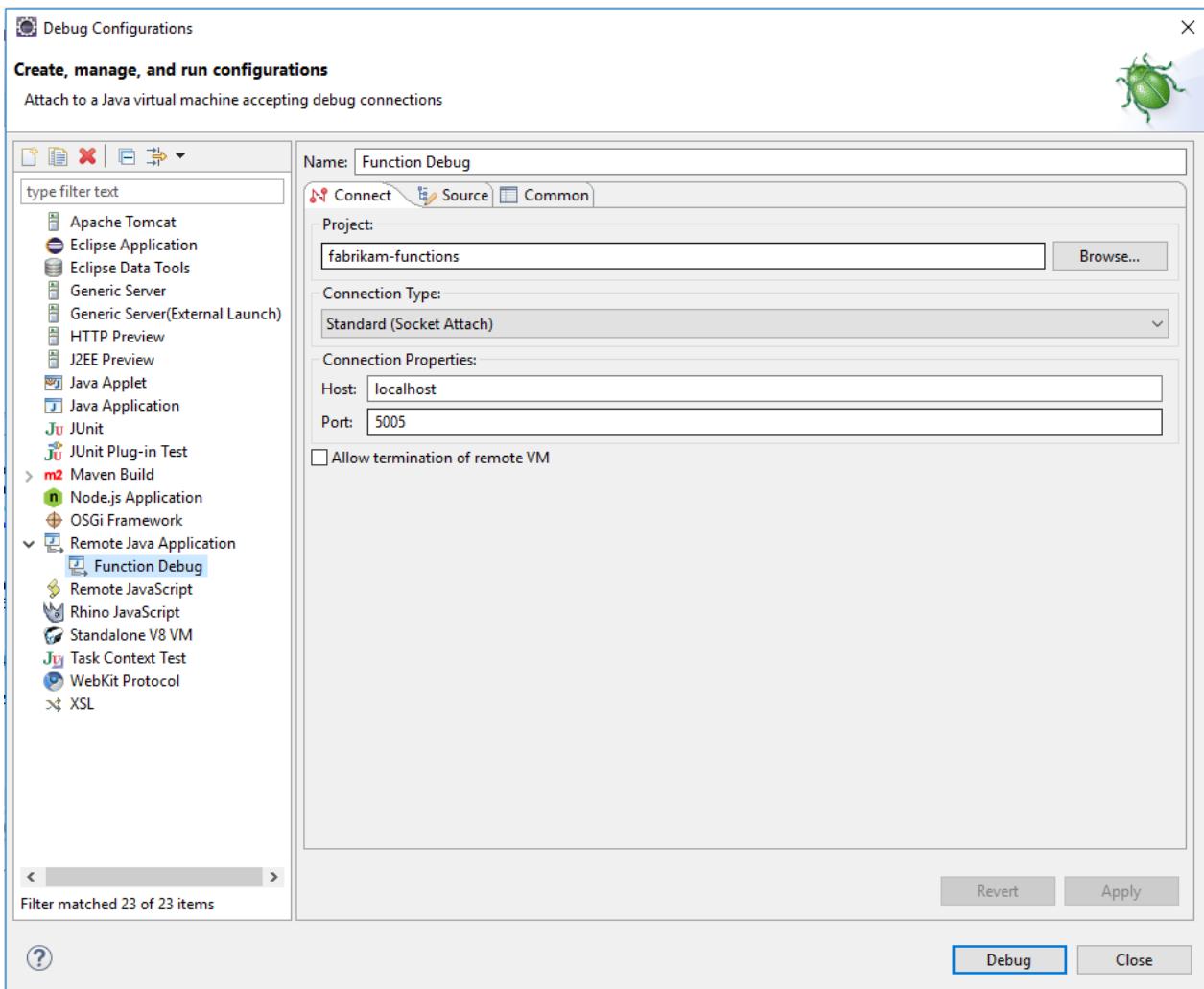
1. Right-click on the generated project, then choose **Run As** and **Maven build**.
2. In the **Edit Configuration** dialog, Enter `package` in the **Goals** and **Name** fields, then select **Run**. This will build and package the function code.
3. Once the build is complete, create another Run configuration 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.

```
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:

```
[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.

Create your first Azure function with Java and IntelliJ

11/2/2020 • 3 minutes to read • [Edit Online](#)

This article shows you:

- How to create a [serverless](#) function project with IntelliJ IDEA
- Steps for testing and debugging the function in the integrated development environment (IDE) on your own computer
- Instructions for deploying the function project to Azure Functions

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

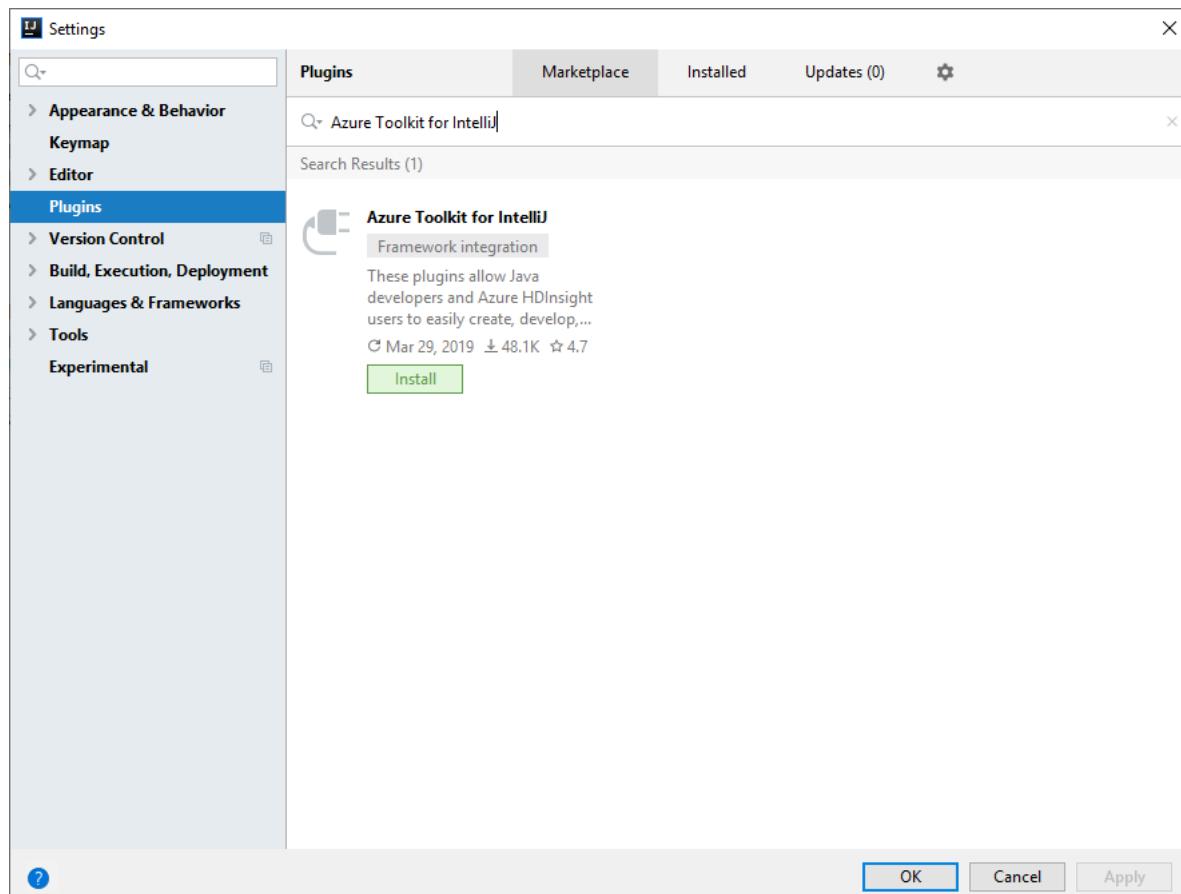
Set up your development environment

To develop a function with Java and IntelliJ, install the following software:

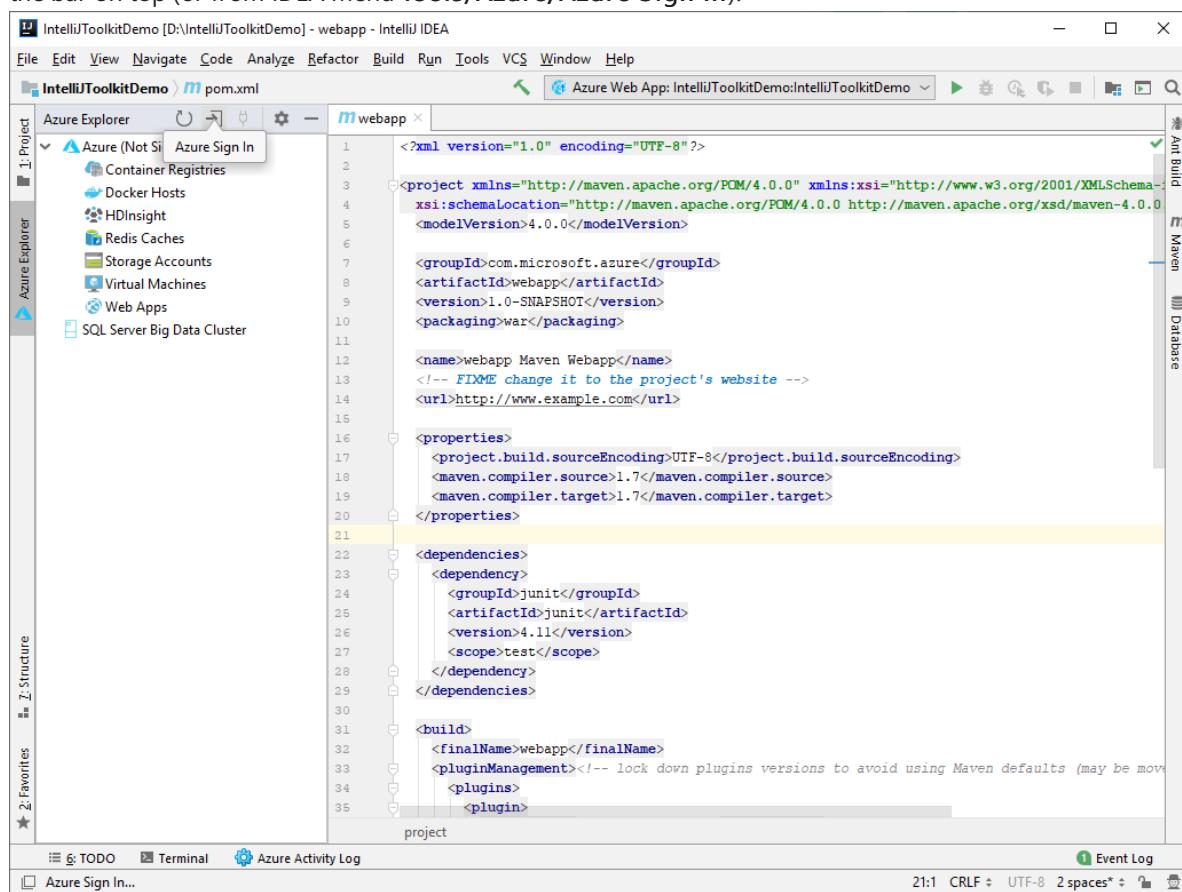
- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure supported Java Development Kit \(JDK\)](#) for Java 8
- An [IntelliJ IDEA](#) Ultimate Edition or Community Edition installed
- [Maven 3.5.0+](#)
- Latest [Function Core Tools](#)

Installation and Sign-in

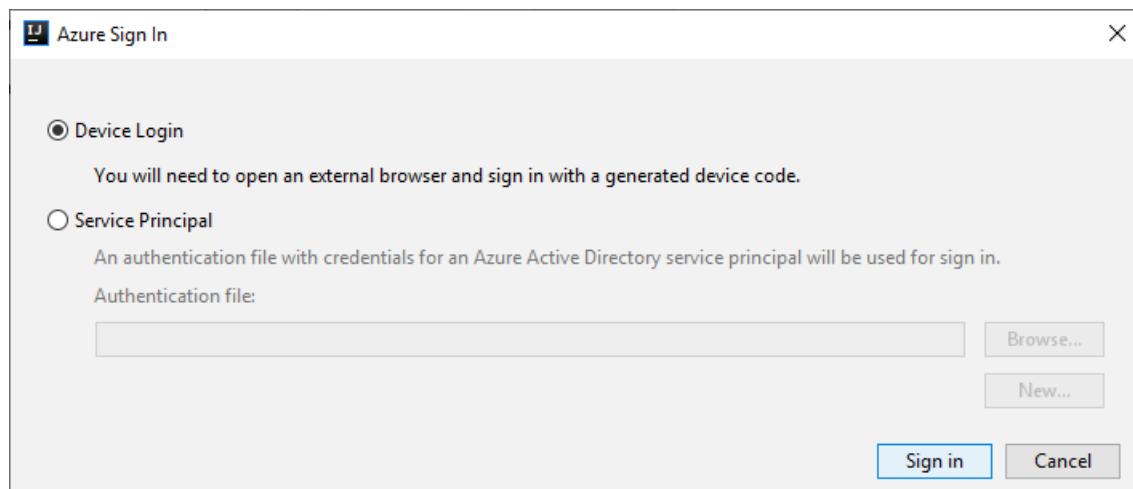
1. In IntelliJ IDEA's Settings/Preferences dialog (Ctrl+Alt+S), select Plugins. Then, find the **Azure Toolkit for IntelliJ** in the Marketplace and click **Install**. After installed, click **Restart** to activate the plugin.



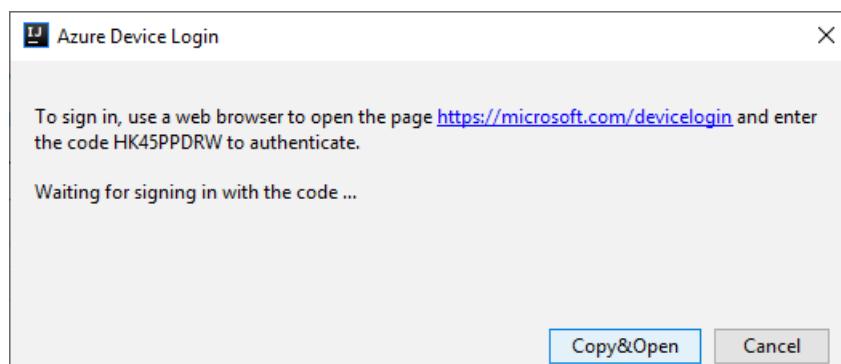
2. To sign in to your Azure account, open sidebar **Azure Explorer**, and then click the **Azure Sign In** icon in the bar on top (or from IDEA menu **Tools/Azure/Azure Sign in**).



3. In the Azure Sign In window, select **Device Login**, and then click **Sign in** ([other sign in options](#)).

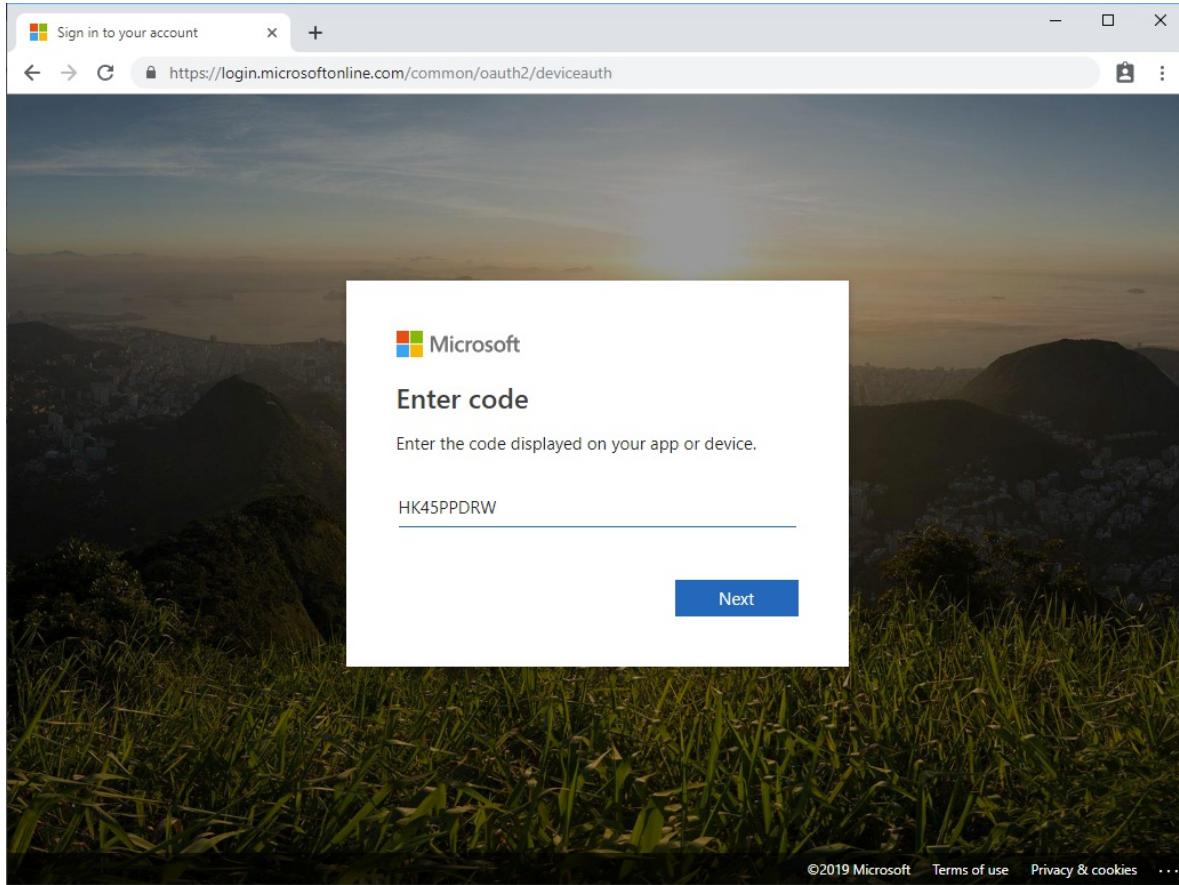


4. Click **Copy&Open** in Azure Device Login dialog .

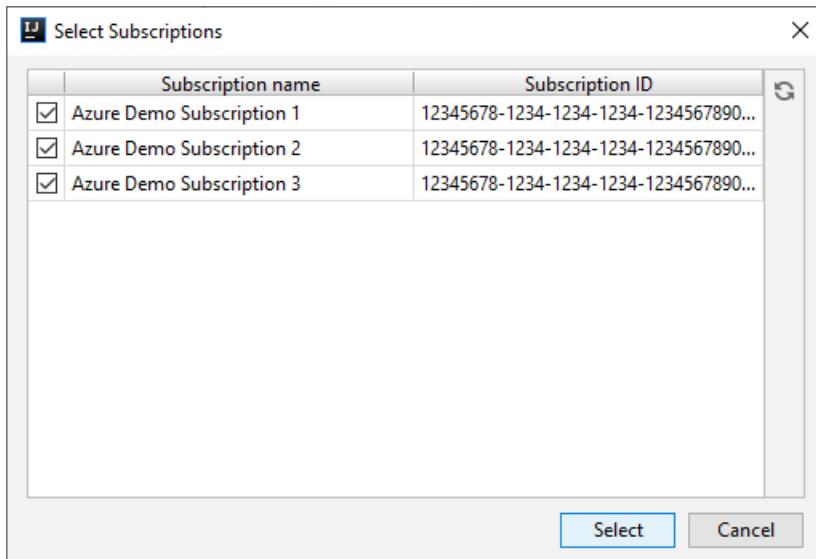


5. In the browser, paste your device code (which has been copied when you click **Copy&Open** in last step)

and then click **Next**.



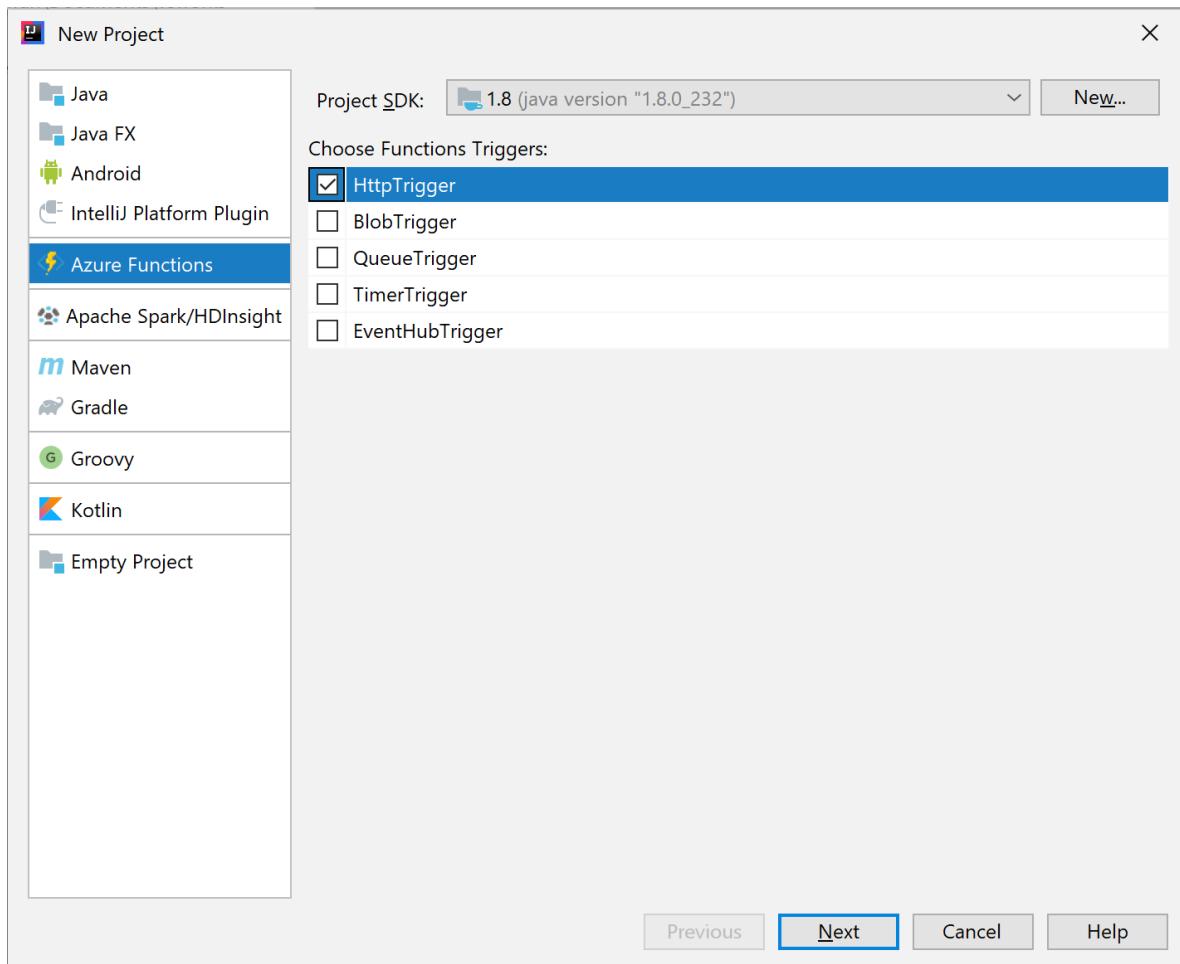
6. In the **Select Subscriptions** dialog box, select the subscriptions that you want to use, and then click **OK**.



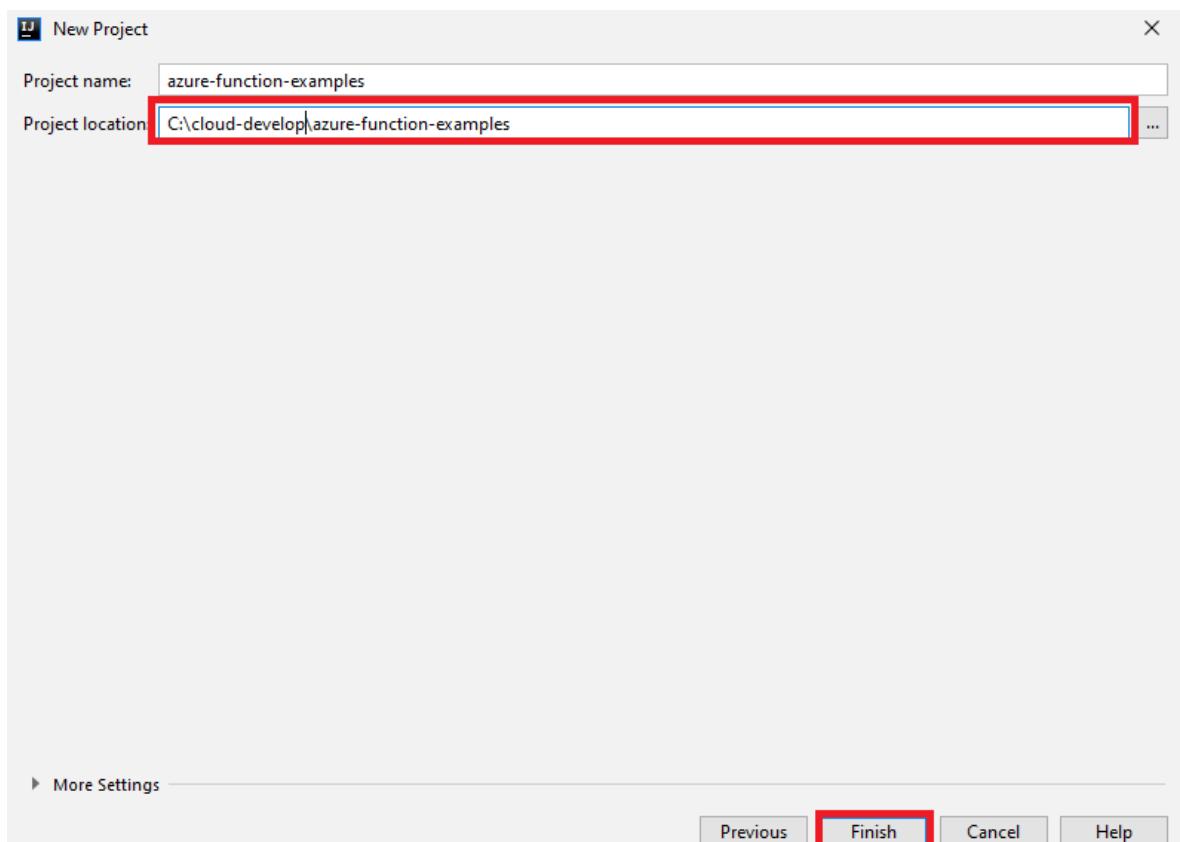
Create your local project

In this section, you use Azure Toolkit for IntelliJ to create a local Azure Functions project. Later in this article, you'll publish your function code to Azure.

1. Open IntelliJ Welcome dialog, select *Create New Project* to open a new Project wizard, select *Azure Functions*.



2. Select *Http Trigger*, then click *Next* and follow the wizard to go through all the configurations in the following pages; confirm your project location then click *Finish*; IntelliJ IDEA will then open your new project.



Run the Function App locally

1. Navigate to `src/main/java/org/example/functions/HttpTriggerFunction.java` to see the code generated.

Beside the line 17, you will notice that there is a green *Run* button, click it and select *Run 'azure-function-exam...*, you will see that your function app is running locally with a few logs.

The screenshot shows the IntelliJ IDEA interface. The top window displays the Java code for `HttpTriggerFunction.java`. The code defines a class `HttpTriggerFunction` with a single method `run` that handles both GET and POST requests. The bottom window, titled "Run: Run Functions - azure-function-examples", shows the application's startup logs and the endpoint information: "Now listening on: http://0.0.0.0:7071". The "Run" tab is selected in the bottom navigation bar.

```
package org.example.functions;
import ...;
public class HttpTriggerFunction {
    /**
     * This function Listens at endpoint "/api/HttpTrigger-Java". Two ways to invoke it using "curl" command in bash:
     * 1. curl -d "HTTP Body" {your host}/api/HttpTrigger-Java
     * 2. curl {your host}/api/HttpTrigger-Java?name=HTTP%20Query
     */
    @FunctionName("HttpTrigger-Java")
    public HttpResponseMessage run(
        @HttpTrigger(name = "req", methods = {HttpMethod.GET, HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<String> req,
        @OutputBinding(name = "res", dataType = " HttpResponseMessage") HttpResponseMessage res) {
        String query = req.getQueryParameters().get("name");
        String name = req.getBody().orElse(query);

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

[3/29/2020 1:53:50 PM] Job host started
Hosting environment: Production
Content root path: C:\Users\yucwan\IdeaProjects\azure-function-examples\target\azure-functions\azure-function-examples-1585477854732
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

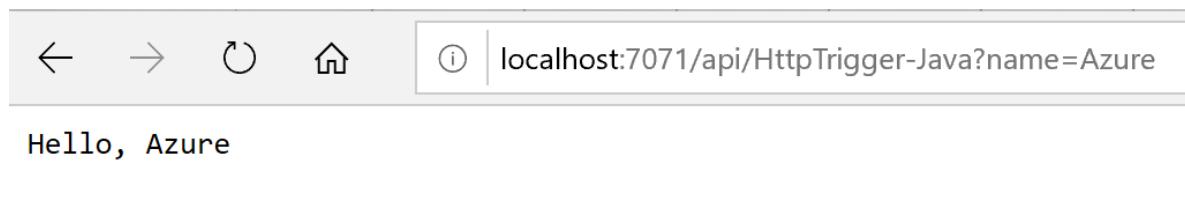
Http Functions:

HttpTrigger-Java: [GET,POST] http://localhost:7071/api/HttpTrigger-Java

[3/29/2020 1:53:55 PM] Host lock lease acquired by instance ID '000000000000000000000000000000001451810'.

2. You can try the function by accessing the printed endpoint from browser, like

<http://localhost:7071/api/HttpTrigger-Java?name=Azure>.



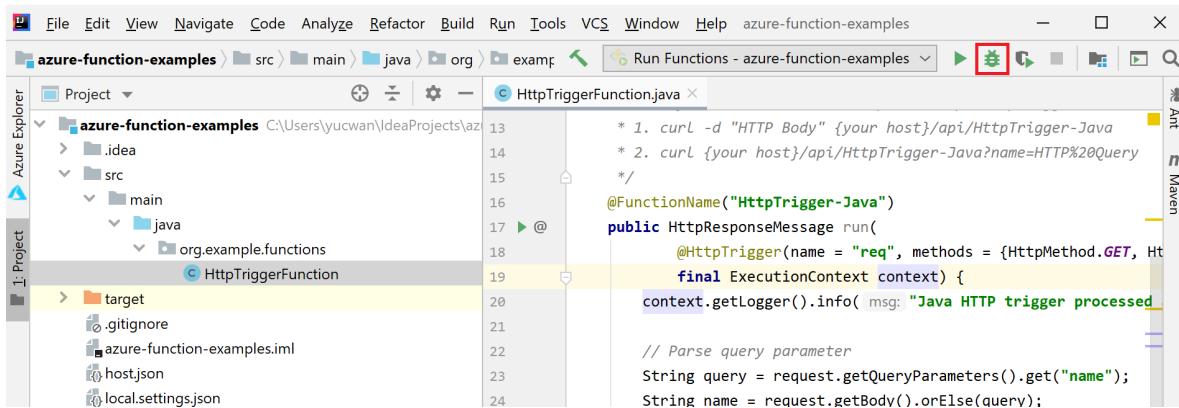
3. The log is also printed out in your IDEA, now stop the function by clicking the *stop* button.

The screenshot shows the IntelliJ IDEA interface with the "Run" tool window open. The logs show the function being triggered via the host API and then stopped. The "Run" tab is selected in the bottom navigation bar.

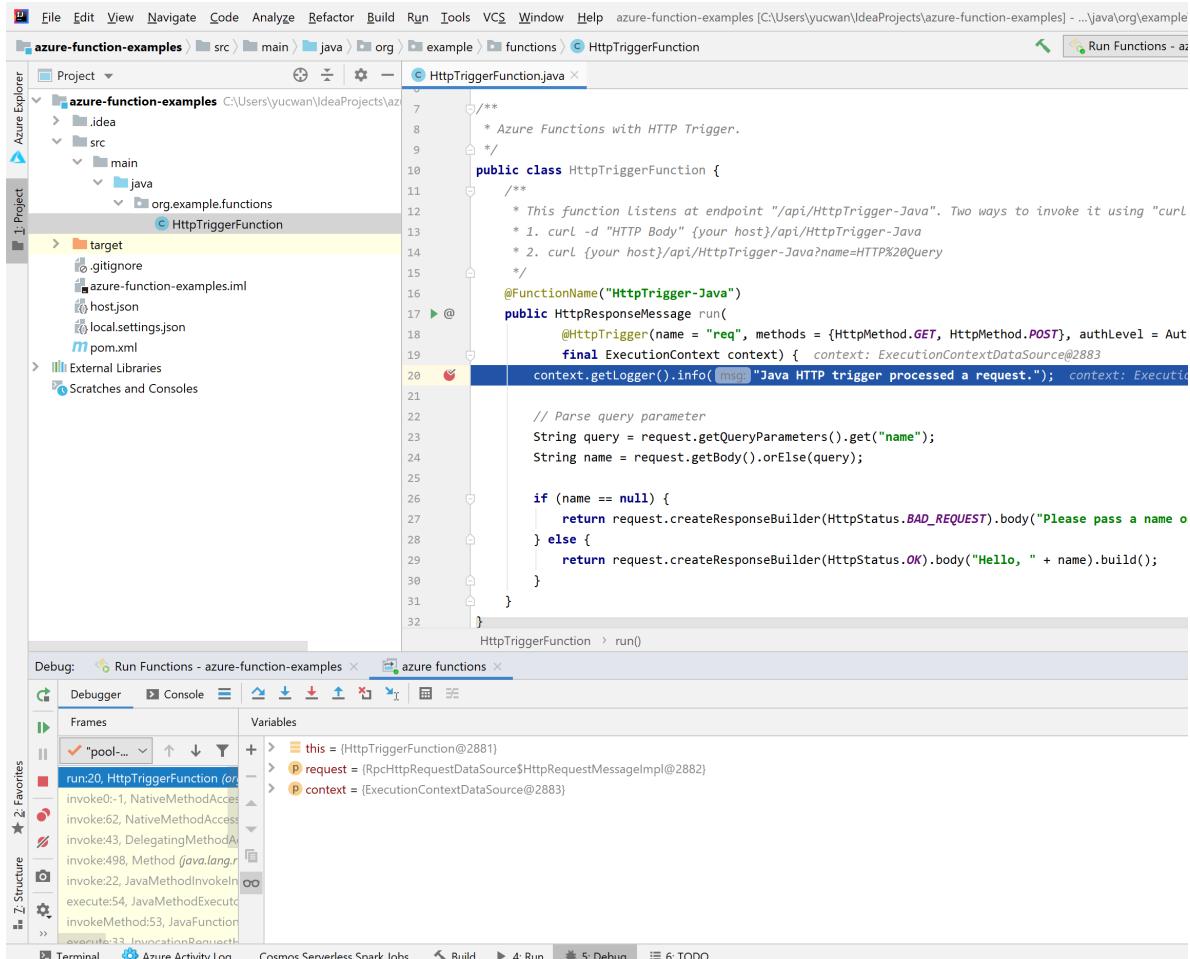
```
[3/29/2020 2:00:43 PM] "uri": "/api/HttpTrigger-Java"
[3/29/2020 2:00:43 PM]
[3/29/2020 2:00:43 PM] Executing 'Functions.HttpTrigger-Java' (Reason='This function was programmatically called via the host APIs.', Id=c65f30ca-f132-490e-8b54-1a674d285c4f)
[3/29/2020 2:00:43 PM] Java HTTP trigger processed a request.
[3/29/2020 2:00:43 PM] Function "HttpTrigger-Java" (Id: c65f30ca-f132-490e-8b54-1a674d285c4f) invoked by Java Worker
[3/29/2020 2:00:43 PM] Executed "Functions.HttpTrigger-Java" (Succeeded, Id=c65f30ca-f132-490e-8b54-1a674d285c4f)
[3/29/2020 2:00:43 PM] Executed HTTP request:
[3/29/2020 2:00:43 PM] "requestId": "1b4c0e067-a216-4451-9727-e94a619138a7",
[3/29/2020 2:00:43 PM] "method": "GET",
[3/29/2020 2:00:43 PM] "uri": "/api/HttpTrigger-Java",
[3/29/2020 2:00:43 PM] "identities": [
[3/29/2020 2:00:43 PM] {
[3/29/2020 2:00:43 PM]   "type": "WebJobsAuthLevel",
[3/29/2020 2:00:43 PM]   "level": "Admin"
[3/29/2020 2:00:43 PM] },
[3/29/2020 2:00:43 PM] ],
[3/29/2020 2:00:43 PM] "status": 200,
[3/29/2020 2:00:43 PM] "duration": 28
[3/29/2020 2:00:43 PM]
```

Debug the Function App locally

- Now let's try to debug your Function App locally, click the *Debug* button in the toolbar (if you don't see it, click *View* -> *Appearance* -> *Toolbar* to enable Toolbar).

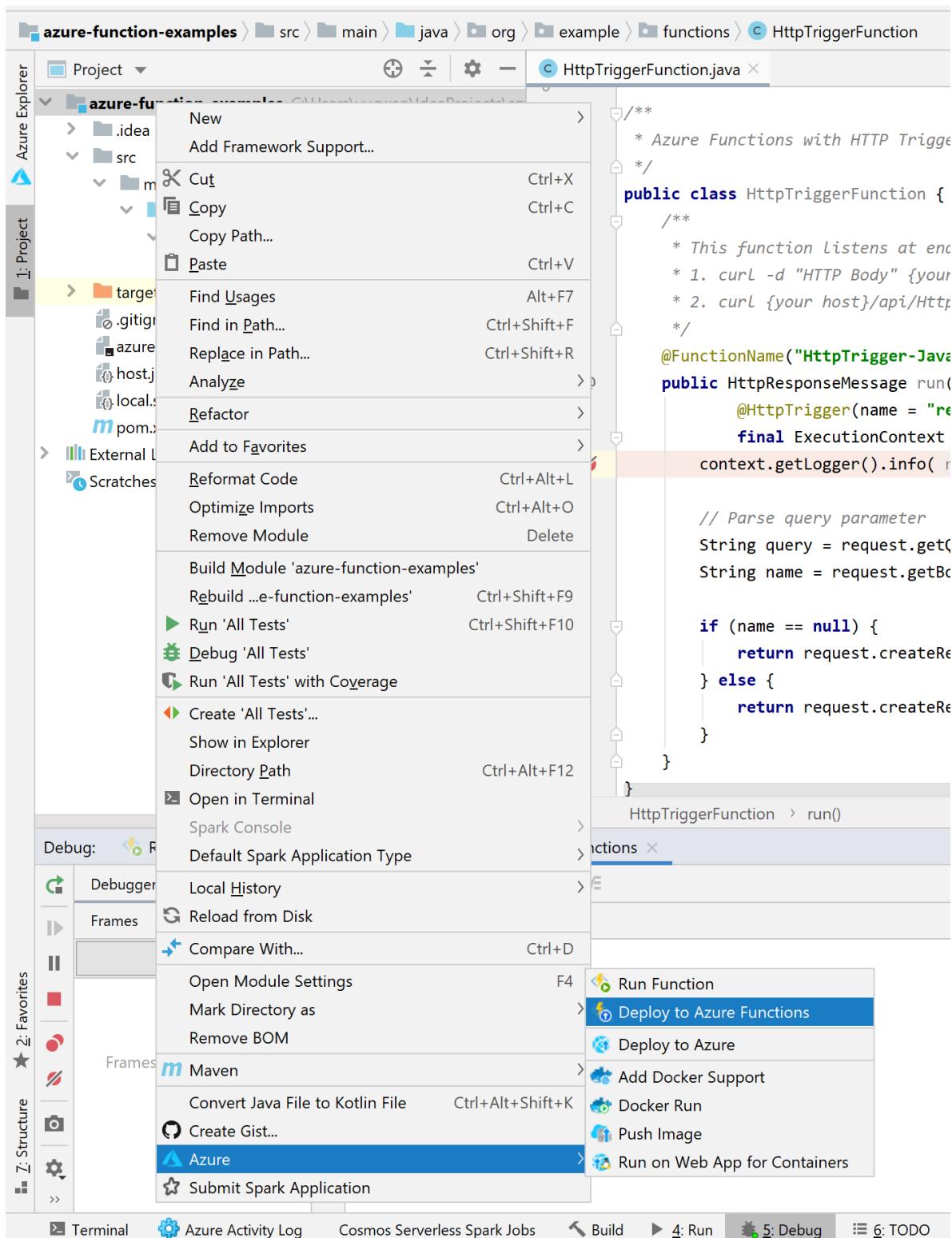


- Click on line 20 of the file `src/main/java/org/example/functions/HttpTriggerFunction.java` to add a breakpoint, access the endpoint `http://localhost:7071/api/HttpTrigger-Java?name=Azure` again, you will find the breakpoint is hit, you can try more debug features like *step*, *watch*, *evaluation*. Stop the debug session by click the stop button.

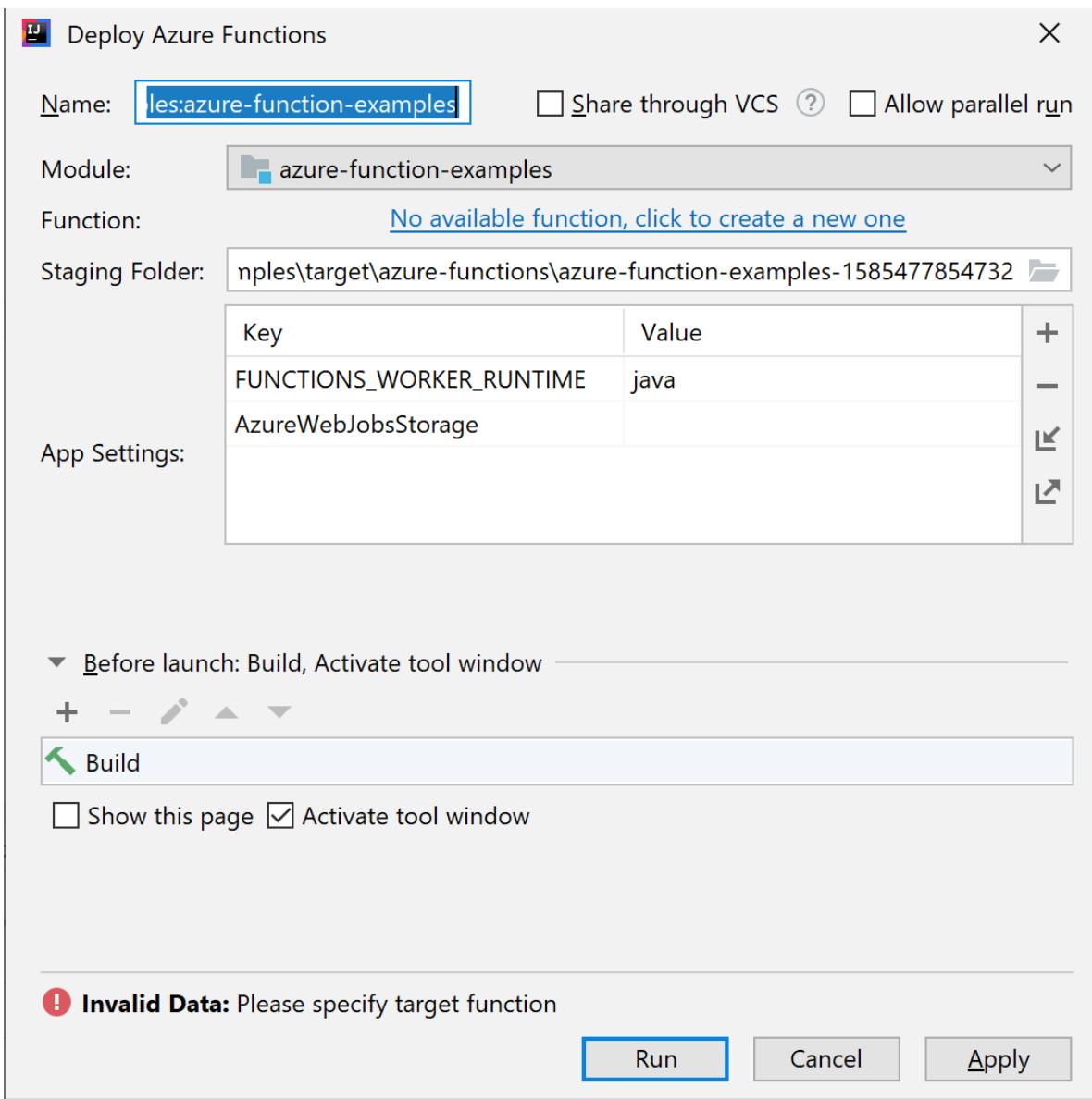


Deploy your Function App to Azure

- Right click your project in IntelliJ Project explorer, select *Azure* -> *Deploy to Azure Functions*



2. If you don't have any Function App yet, click *No available function*, click to create a new one.



3. Type in the Function app name and choose proper subscription/platform/resource group/App Service plan, you can also create resource group/App Service plan here. Then, keep app settings unchanged, click OK and wait some minutes for the new function to be created. After *Creating New Function App...* progress bar disappears.

Create Function App

Name: .azurewebsites.net

Subscription:

Platform: Windows Linux

Resource Group:

App Service Plan: [Create app service plan...](#)

Pricing Tier: N/A

AppSettings:

| Key | Value | Actions |
|-------------------------|-------|---|
| FUNCTIONS_WORKER_RUN... | java | + - ↶ ↷ |
| AzureWebJobsStorage | | + - ↶ ↷ |

OK Cancel

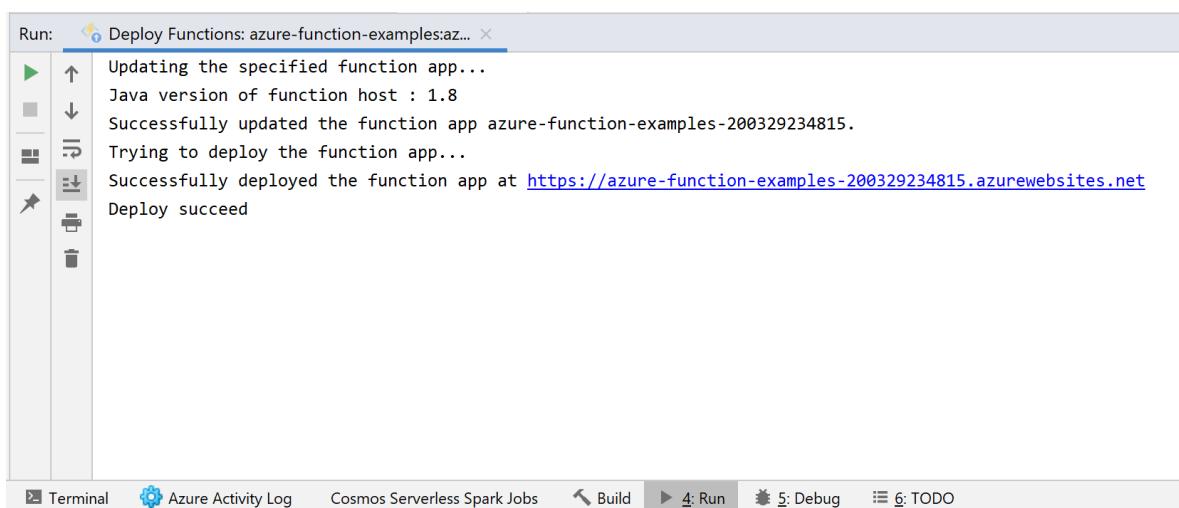
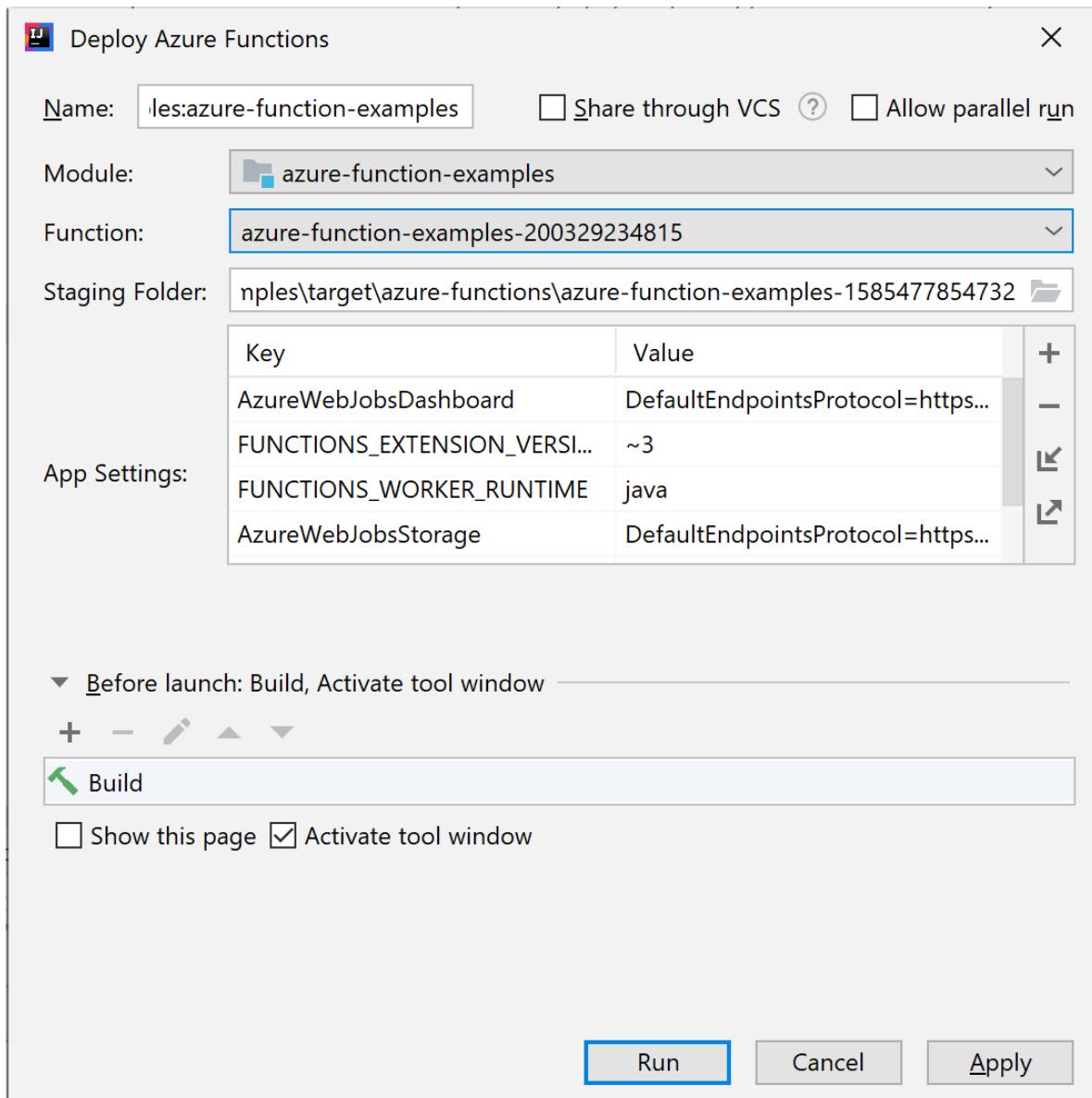
The screenshot shows the 'Create Function App' dialog box. At the top left is a 'Create Function App' icon and the title 'Create Function App'. At the top right is a close button ('X'). The main area contains several input fields and dropdowns:

- 'Name:' field containing 'azure-function-examples-200329230845' with '.azurewebsites.net' suffix.
- 'Subscription:' dropdown set to 'Visual Studio Enterprise'.
- 'Platform:' radio buttons for 'Windows' (selected) and 'Linux'.
- 'Resource Group:' dropdown set to 'AzureBackupRG_japaneast_1'.
- 'App Service Plan:' dropdown currently empty, with a blue button below it labeled 'Create app service plan...'.
- 'Pricing Tier:' label followed by 'N/A'.
- 'AppSettings:' section with a table:

| Key | Value | Actions |
|-------------------------|-------|---|
| FUNCTIONS_WORKER_RUN... | java | + - ↶ ↷ |
| AzureWebJobsStorage | | + - ↶ ↷ |

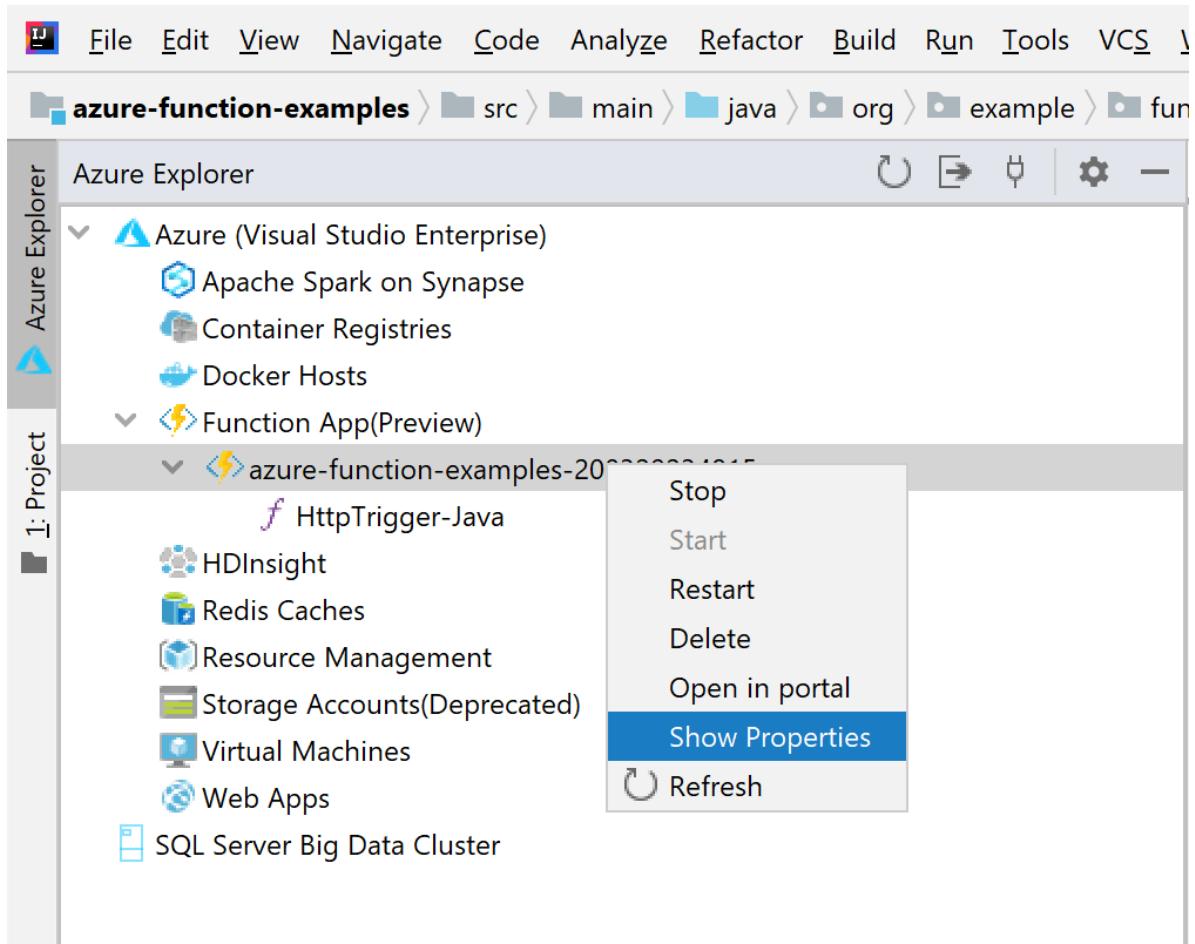
At the bottom are 'OK' and 'Cancel' buttons. The 'Create app service plan...' button is highlighted with a blue background.

4. Select the function app you want to deploy to, (the new function app you just created will be automatically selected). Click *Run* to deploy your functions.

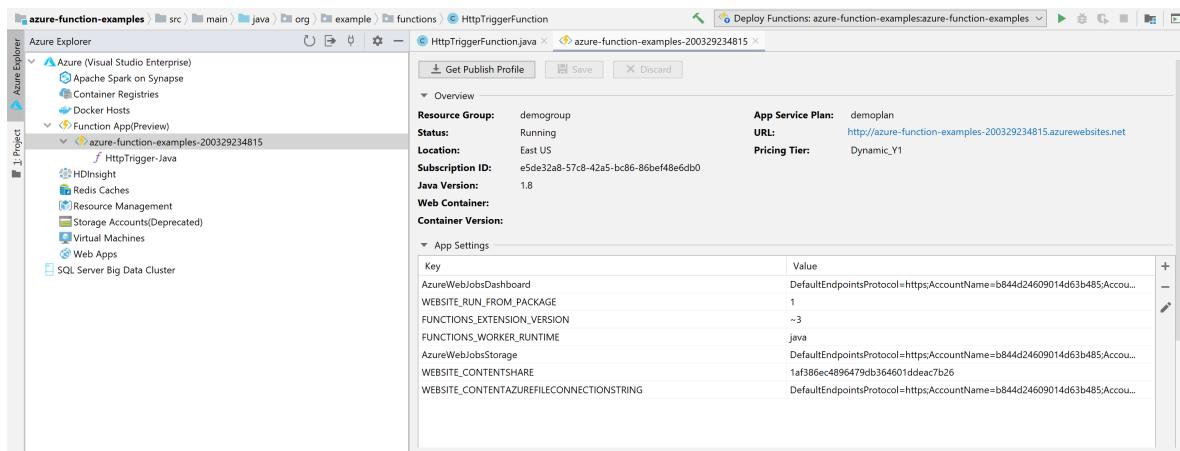


Manage Azure Functions from IDEA

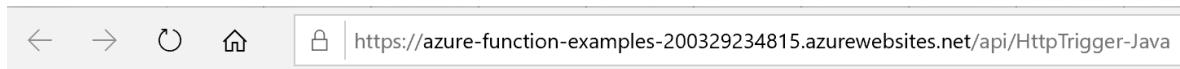
1. You can manage your functions with *Azure Explorer* in your IDEA, click on *Function App*, you will see all your functions here.



2. Click to select on one of your functions, and right click, select *Show Properties* to open the detail page.

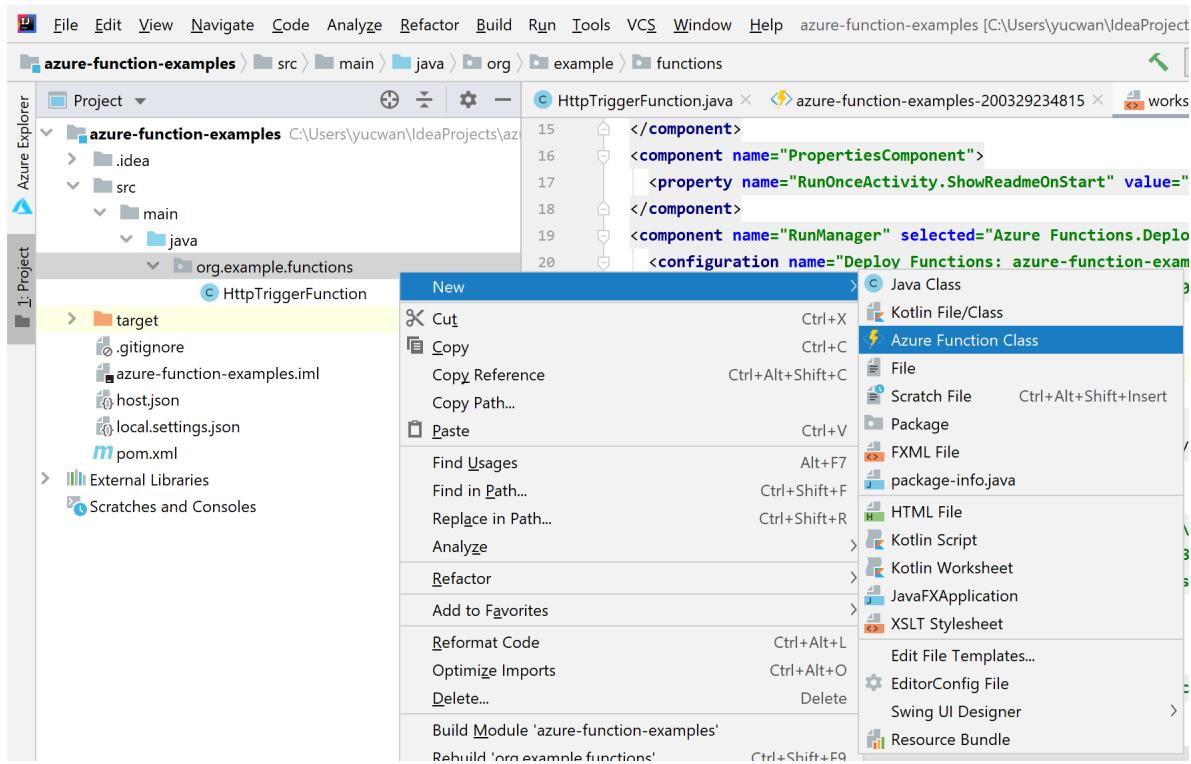


3. Right click on your Function *HttpTrigger-Java*, and select *Trigger Function*, you will see that the browser is opened with the trigger URL.

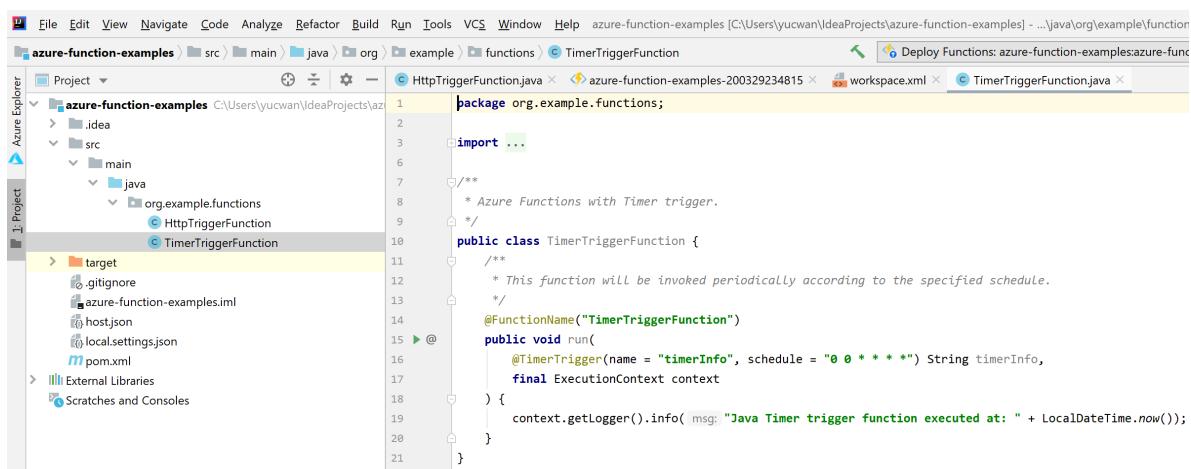
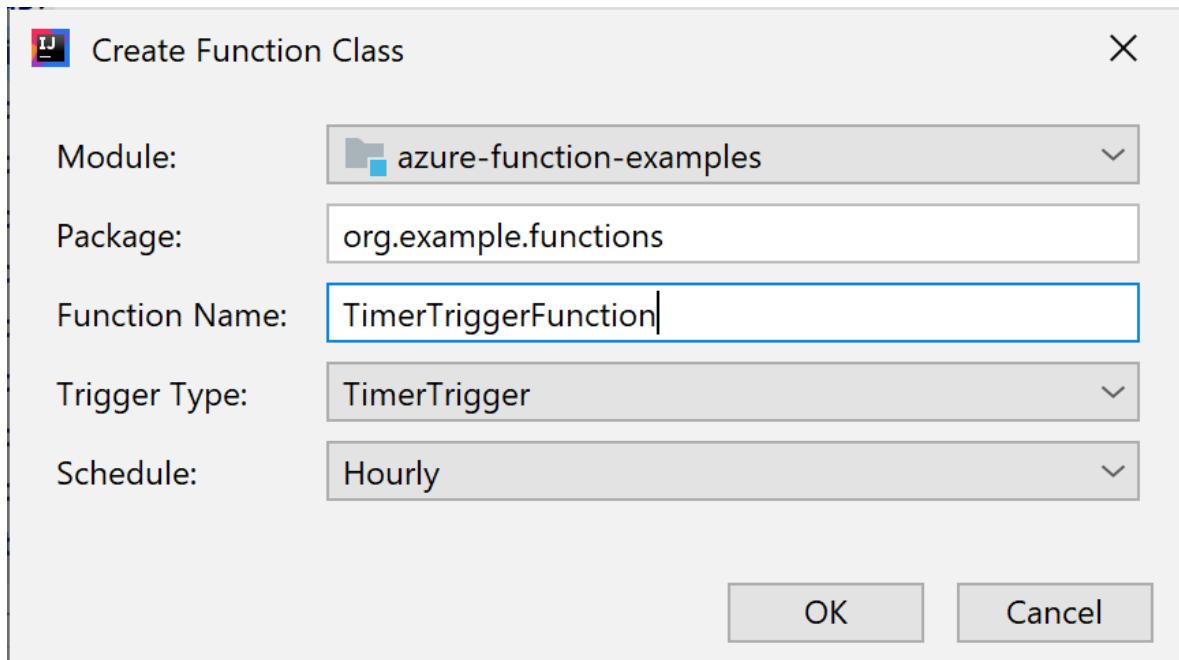


Add more Functions to the project

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

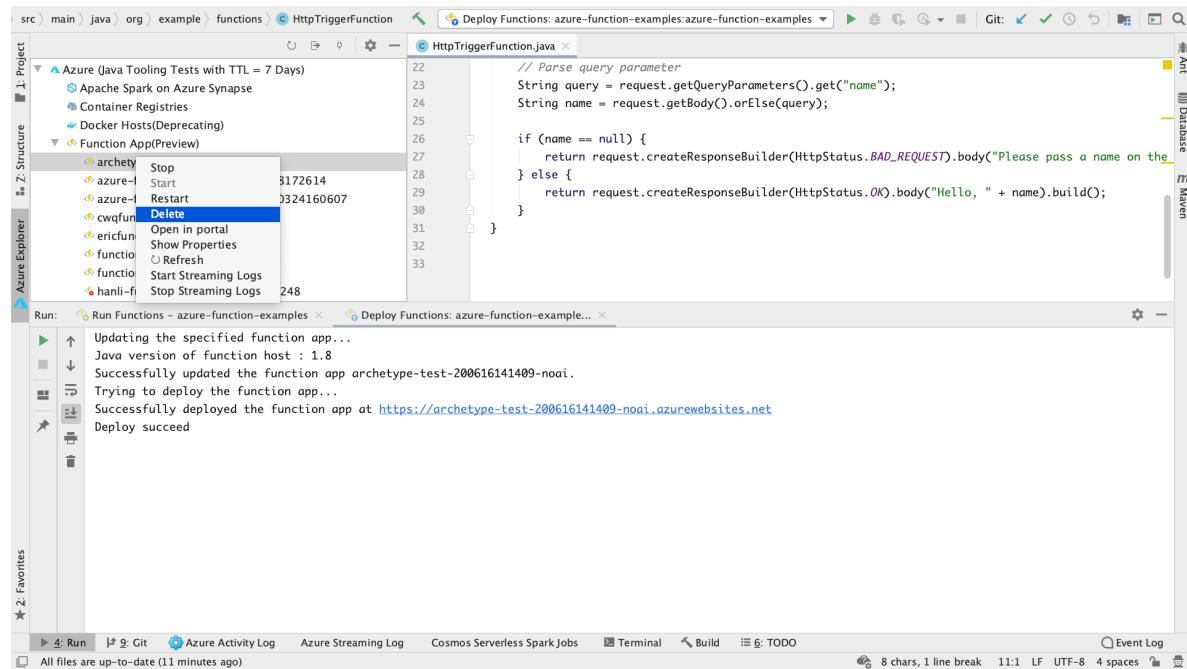


2. Fill in the class name *HttpTest* and select *HttpTrigger* in the create function class wizard, click *OK* to create, in this way, you can create new functions as you want.



Cleaning Up Functions

1. Deleting Azure Functions in Azure Explorer



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](#)

Quickstart: Create your first function with Kotlin and Maven

11/2/2020 • 5 minutes to read • [Edit Online](#)

This article guides you through using the Maven command-line tool to build and publish a Kotlin function project 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.

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

Prerequisites

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

- [Java Developer Kit](#), version 8
- [Apache Maven](#), version 3.0 or above
- [Azure CLI](#)
- [Azure Functions Core Tools](#) version 2.6.666 or above

IMPORTANT

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

Generate a new Functions project

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

- [bash](#)
- [PowerShell](#)
- [Cmd](#)

```
mvn archetype:generate \
-DarchetypeGroupId=com.microsoft.azure \
-DarchetypeArtifactId=azure-functions-kotlin-archetype
```

NOTE

If you're experiencing issues with running the command, take a look at what `maven-archetype-plugin` version is used. Because you are running the command in an empty directory with no `.pom` file, it might be attempting to use a plugin of the older version from `~/.m2/repository/org/apache/maven/plugins/maven-archetype-plugin` if you upgraded your Maven from an older version. If so, try deleting the `maven-archetype-plugin` directory and re-running the command.

Maven asks you for values needed to finish generating the project. For `groupId`, `artifactId`, and `version` values, see the [Maven naming conventions](#) reference. The `appName` value must be unique across Azure, so Maven generates an app name based on the previously entered `artifactId` as a default. The `packageName` value determines the Kotlin package for the generated function code.

The `com.fabrikam.functions` and `fabrikam-functions` identifiers below are used as an example and to make later

steps in this quickstart easier to read. You're encouraged to supply your own values to Maven in this step.

```
[INFO] Parameter: groupId, Value: com.fabrikam.function
[INFO] Parameter: artifactId, Value: fabrikam-function
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.fabrikam.function
[INFO] Parameter: packageInPathFormat, Value: com/fabrikam/function
[INFO] Parameter: appName, Value: fabrikam-function-20190524171507291
[INFO] Parameter: resourceGroup, Value: java-functions-group
[INFO] Parameter: package, Value: com.fabrikam.function
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.fabrikam.function
[INFO] Parameter: appRegion, Value: westus
[INFO] Parameter: artifactId, Value: fabrikam-function
```

Maven creates the project files in a new folder with a name of *artifactId*, in this example `fabrikam-functions`. The ready to run generated code in the project is a simple [HTTP triggered](#) function that echoes the body of the request:

```
class Function {

    /**
     * This function listens at endpoint "/api/HttpTrigger-Java". Two ways to invoke it using "curl" command in
     * bash:
     * 1. curl -d "HTTP Body" {your host}/api/HttpTrigger-Java&code={your function key}
     * 2. curl "{your host}/api/HttpTrigger-Java?name=HTTP%20Query&code={your function key}"
     * Function Key is not needed when running locally, it is used to invoke function deployed to Azure.
     * More details: https://aka.ms/functions_authorization_keys
     */
    @FunctionName("HttpTrigger-Java")
    fun run(
        @HttpTrigger(
            name = "req",
            methods = [HttpMethod.GET, HttpMethod.POST],
            authLevel = AuthorizationLevel.FUNCTION) request: HttpRequestMessage<Optional<String>>,
        context: ExecutionContext): HttpResponseMessage {
        context.logger.info("HTTP trigger processed a ${request.httpMethod.name} request.")

        val query = request.queryParameters["name"]
        val name = request.body.orElse(query)

        name?.let {
            return request
                .createResponseBuilder(HttpStatus.OK)
                .body("Hello, $name!")
                .build()
        }

        return request
            .createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Please pass a name on the query string or in the request body")
            .build()
    }
}
```

Run the function locally

Change directory to the newly created project folder and build and run the function with Maven:

```
cd fabrikam-function  
mvn clean package  
mvn azure-functions:run
```

NOTE

If you're experiencing this exception: `javax.xml.bind.JAXBException` with Java 9, see the workaround on [GitHub](#).

You see this output when the function is running locally on your system and ready to respond to HTTP requests:

```
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.  
  
Http Functions:  
  
HttpTrigger-Java: [GET,POST] http://localhost:7071/api/HttpTrigger-Java
```

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

```
curl -w '\n' -d LocalFunction http://localhost:7071/api/HttpTrigger-Java
```

```
Hello LocalFunction!
```

Use `ctrl-C` in the terminal to stop the function code.

Deploy the function to Azure

The deploy process to Azure Functions uses account credentials from the Azure CLI. [Sign in with the Azure CLI](#) before continuing.

```
az login
```

Deploy your code into a new Function app using the `azure-functions:deploy` Maven target.

NOTE

When you use Visual Studio Code to deploy your Function app, remember to choose a non-free subscription, or you will get an error. You can watch your subscription on the left side of the IDE.

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

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

```
[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] -----
```

Test the function app running on Azure using `curl`. You'll need to change the URL from the sample below to match the deployed URL for your own function app from the previous step.

NOTE

Make sure you set the **Access rights** to `Anonymous`. When you choose the default level of `Function`, you are required to present the [function key](#) in requests to access your function endpoint.

```
curl -w '\n' https://fabrikam-function-20170920120101928.azurewebsites.net/api/HttpTrigger-Java -d
AzureFunctions
```

```
Hello AzureFunctions!
```

Make changes and redeploy

Edit the `src/main.../Function.java` source file in the generated project to alter the text returned by your Function app. Change this line:

```
return request
    .createResponseBuilder(HttpStatus.OK)
    .body("Hello, $name!")
    .build()
```

To the following code:

```
return request
    .createResponseBuilder(HttpStatus.OK)
    .body("Hi, $name!")
    .build()
```

Save the changes and redeploy by running `azure-functions:deploy` from the terminal as before. The function app will be updated and this request:

```
curl -w '\n' -d AzureFunctionsTest https://fabrikam-functions-
20170920120101928.azurewebsites.net/api/HttpTrigger-Java
```

You see the updated output:

```
Hi, AzureFunctionsTest
```

Reference bindings

To work with [Functions triggers and bindings](#) other than HTTP trigger and Timer trigger, you need to install binding extensions. While not required by this article, you'll need to know how to do enable extensions when working with other binding types.

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:

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[1.*, 2.0.0)"  
    }  
}
```

Next steps

You've created a Kotlin function app with a simple HTTP trigger and deployed it to Azure Functions.

- Review the [Java Functions developer guide](#) for more information on developing Java and Kotlin functions.
- Add additional functions with different triggers to your project using the `azure-functions:add` Maven target.
- Write and debug functions locally with [Visual Studio Code](#), [IntelliJ](#), and [Eclipse](#).
- Debug functions deployed in Azure with Visual Studio Code. See the Visual Studio Code [serverless Java applications](#) documentation for instructions.

Quickstart: Create your first HTTP triggered function with Kotlin and IntelliJ

11/2/2020 • 3 minutes to read • [Edit Online](#)

This article shows you how to create a [serverless](#) function project with IntelliJ IDEA and Apache Maven. It also shows how to locally debug your function code in the integrated development environment (IDE) and then deploy the function project to Azure.

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

Set up your development environment

To develop a function with Kotlin and IntelliJ, install the following software:

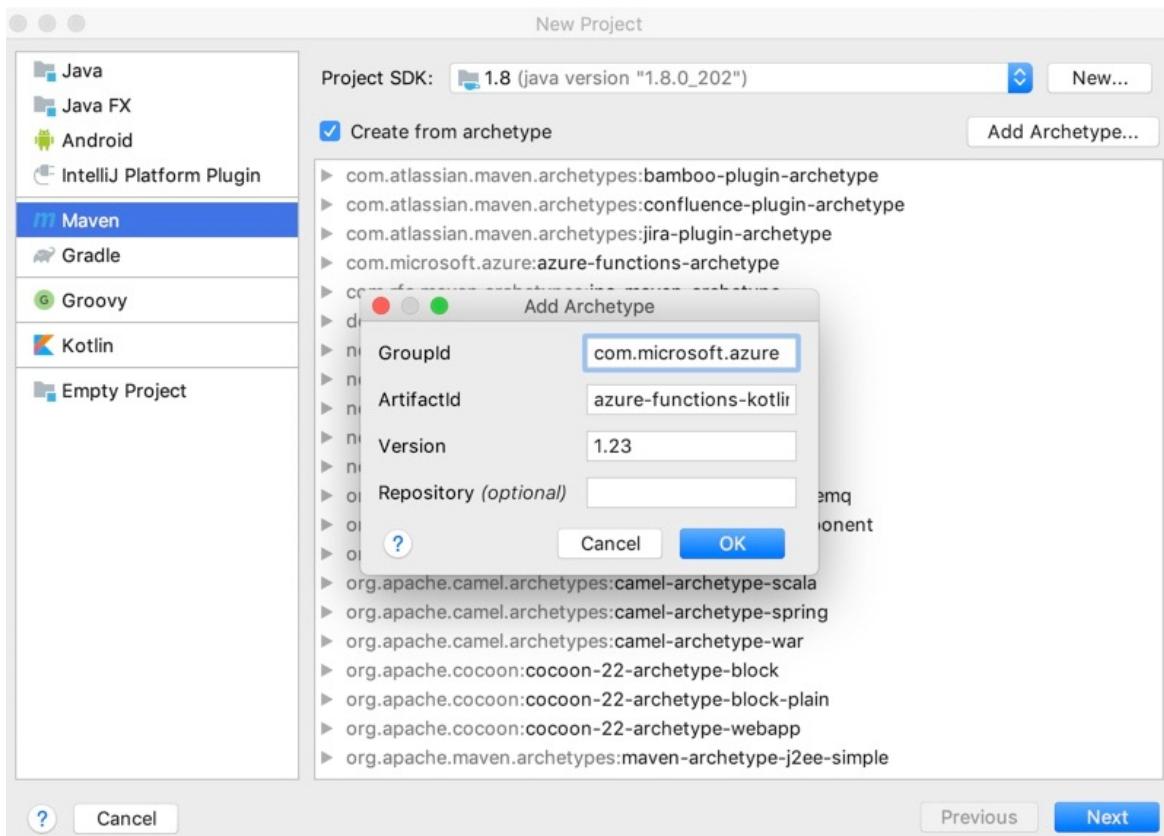
- [Java Developer Kit](#) (JDK), version 8
- [Apache Maven](#), version 3.0 or higher
- [IntelliJ IDEA](#), Community or Ultimate versions with Maven
- [Azure CLI](#)
- [Version 2.x](#) of the Azure Functions Core Tools. It provides a local development environment for writing, running, and debugging 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 Functions 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**.

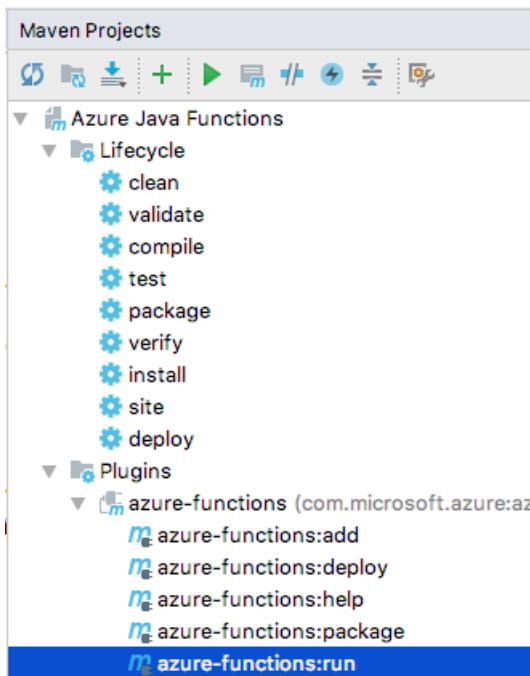
Maven creates the project files in a new folder with the same name as the *ArtifactId* value. The project's generated code is a simple [HTTP-triggered](#) function that echoes the body of the triggering HTTP request.

Run functions locally in the IDE

NOTE

To run and debug functions 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.

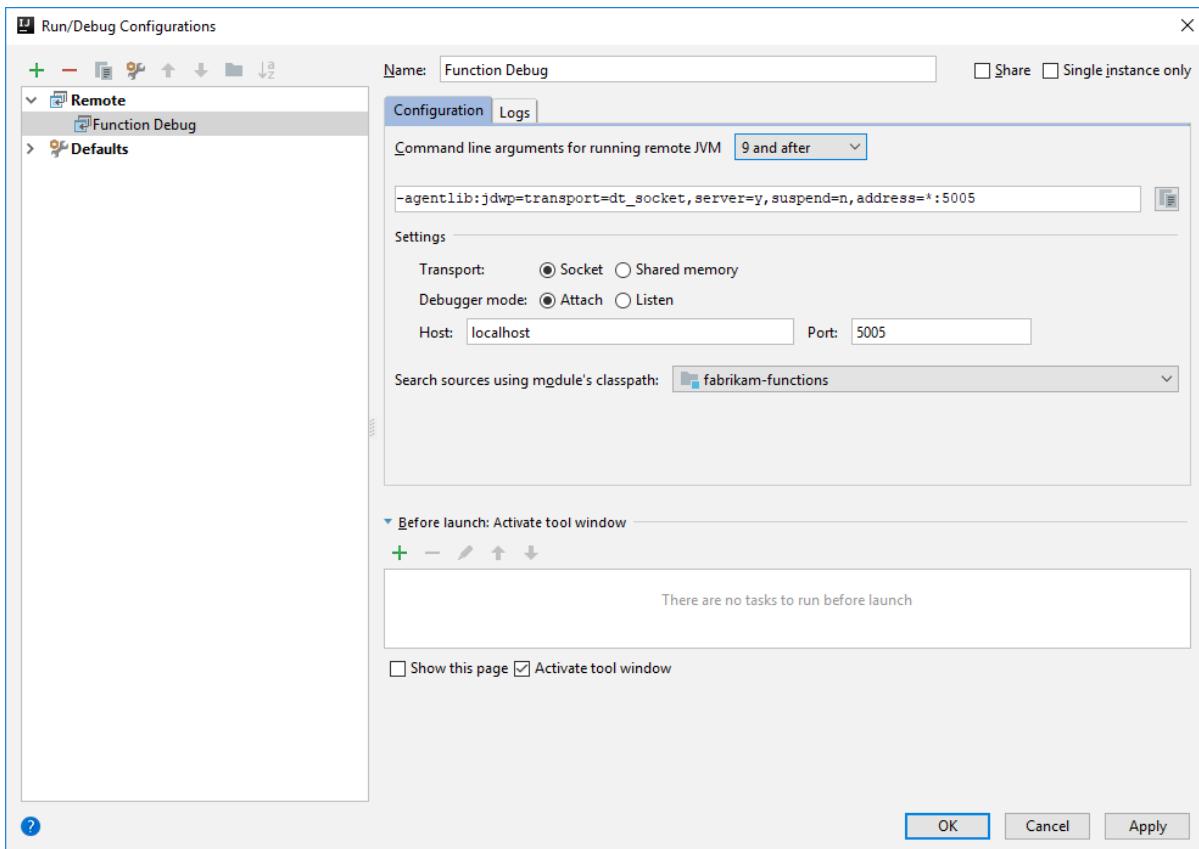
Debug the function in IntelliJ

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 function to Azure

- Before you can deploy your function to Azure, you must [log in by using the Azure CLI](#).

```
az login
```

- Deploy your code into a new function 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 function in the Azure CLI output after the function has been successfully deployed.

```
[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

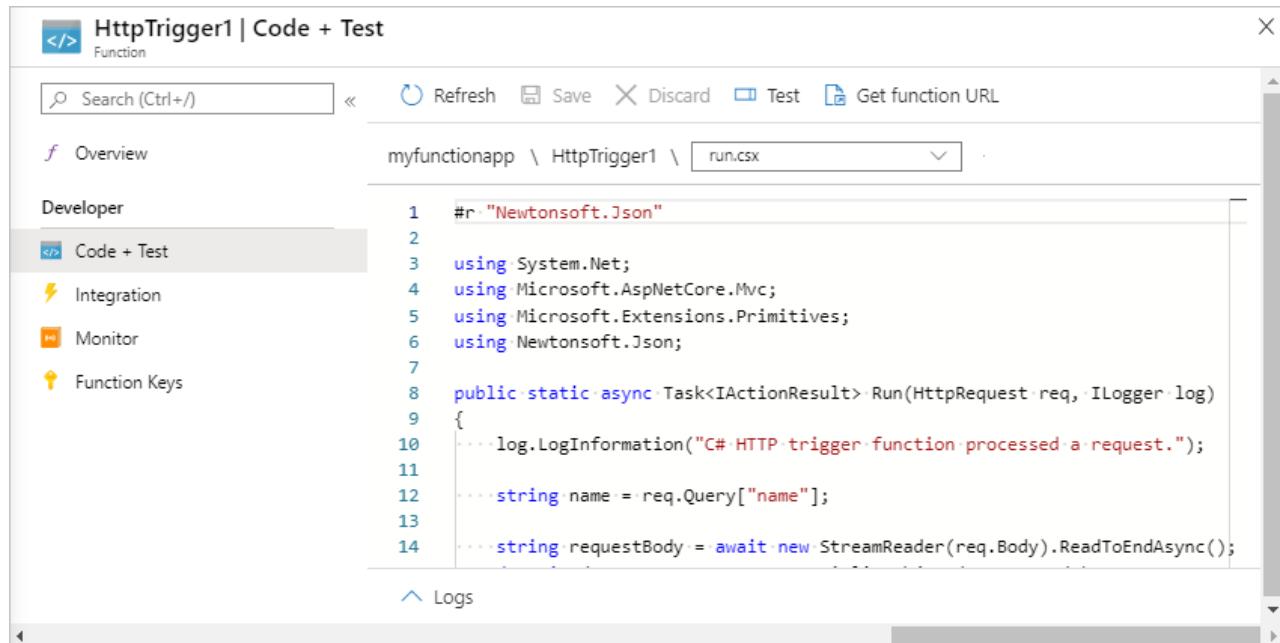
Now that you have deployed your first Kotlin function to Azure, review the [Java Functions developer guide](#) for more information on developing Java and Kotlin functions.

- Add additional functions with different triggers to your project by using the `azure-functions:add` Maven target.

Create a function app on Linux in an Azure App Service plan

12/4/2020 • 5 minutes to read • [Edit Online](#)

Azure Functions lets you host your functions on Linux in a default Azure App Service container. This article walks you through how to use the [Azure portal](#) to create a Linux-hosted function app that runs in an [App Service plan](#). You can also [bring your own custom container](#).



The screenshot shows the Azure portal's "Code + Test" interface for an "HttpTrigger1" function. The left sidebar has tabs for Overview, Developer, Code + Test (which is selected), Integration, Monitor, and Function Keys. The main area shows the file path "myfunctionapp \ HttpTrigger1 \ run.csx" and the C# code for the function:

```
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(HttpContext req, ILogger log)
9 {
10     log.LogInformation("C# HTTP trigger function processed a request.");
11     string name = req.Query["name"];
12     string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
```

If you don't have an [Azure subscription](#), create a [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 a function app

You must have a function app to host the execution of your functions on Linux. The function app provides an environment for execution of your function code. It lets you group functions as a logical unit for easier management, deployment, scaling, and sharing of resources. In this article, you create an App Service plan when you create your 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**.

The screenshot shows the Azure portal's 'New' blade. On the left, there's a sidebar with various service icons and links like Home, Dashboard, All services, Favorites, etc. A red box highlights the 'Create a resource' button at the top. The main area has two tabs: 'Azure Marketplace' (selected) and 'Featured'. Under 'Featured', there are several categories: Get started, Recently created, AI + Machine Learning, Analytics, Blockchain, Compute (highlighted with a red box), Containers, Databases, Developer Tools, DevOps, Identity, Integration, Internet of Things, Media, Mixed Reality, IT & Management Tools, Networking, and Software as a Service (SaaS). Each category has a small icon and a 'Learn more' link. The 'Compute' category is expanded, showing options like Virtual machine, SQL Server 2017 Enterprise Windows Server 2016, Reserved VM Instances, Kubernetes Service, Service Fabric Cluster, Web App for Containers, and Function App (which is also highlighted with a red box).

- 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 (default) | Option to publish code files or a Docker container. |
| Runtime stack | Preferred language | Choose a runtime that supports your favorite function programming language. Choose .NET Core for C# and F# functions. |
| 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. |

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 * ⓘ Visual Studio Enterprise

Resource Group * ⓘ (New) myresourcegroup [Create new](#)

Instance Details

Function App name * myfunctionapplinux .azurewebsites.net

Publish * [Code](#) Docker Container

Runtime stack * .NET Core

Version * 3.1

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 | Linux | An operating system is pre-selected for you based on your runtime stack selection, but you can change the setting if necessary. |
| Plan | 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. When you run in an App Service plan, you must manage the scaling of your 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 * ▼
[Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux Windows

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#) ⓘ

Plan type * ▼

Review + create < Previous Next : Monitoring >

5. Select **Next : Monitoring**. On the Monitoring page, enter the following settings.

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|--------------------------------------|-----------------|---|
| Application Insights | Yes (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. |

Function App

Basics Hosting **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights * No Yes

Application Insights *

Region

Review + create

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.

The screenshot shows the Azure Notifications panel. At the top, there are icons for search, refresh, notifications, help, and user profile. The notifications area has a title "Notifications" and a close button. It contains a message: "More events in the activity log → Deployment succeeded". Below the message, it says "Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful." At the bottom, there are two buttons: "Go to resource" (highlighted with a red box) and "Pin to dashboard". A timestamp "a few seconds ago" is shown at the bottom right.

Even after your function app is available, it may take a few minutes to be fully initialized.

Next, you create a function in the new function app.

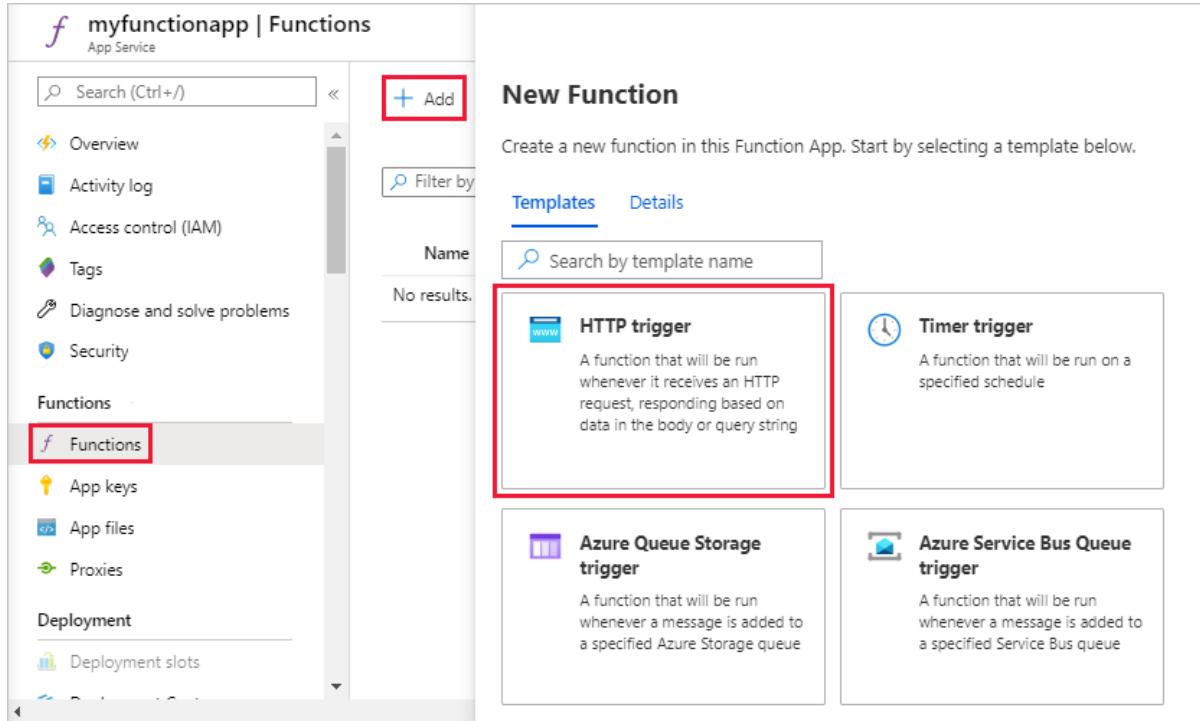
Create an HTTP trigger function

This section shows you how to create a function in your new function app in the portal.

NOTE

The portal development experience can be useful for trying out Azure Functions. For most scenarios, consider developing your functions locally and publishing the project to your function app using either [Visual Studio Code](#) or the [Azure Functions Core Tools](#).

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**.



3. In the **New Function** window, accept the default name for **New Function**, or enter a new name.
4. Choose **Anonymous** from the **Authorization level** drop-down list, and then select **Create Function**.

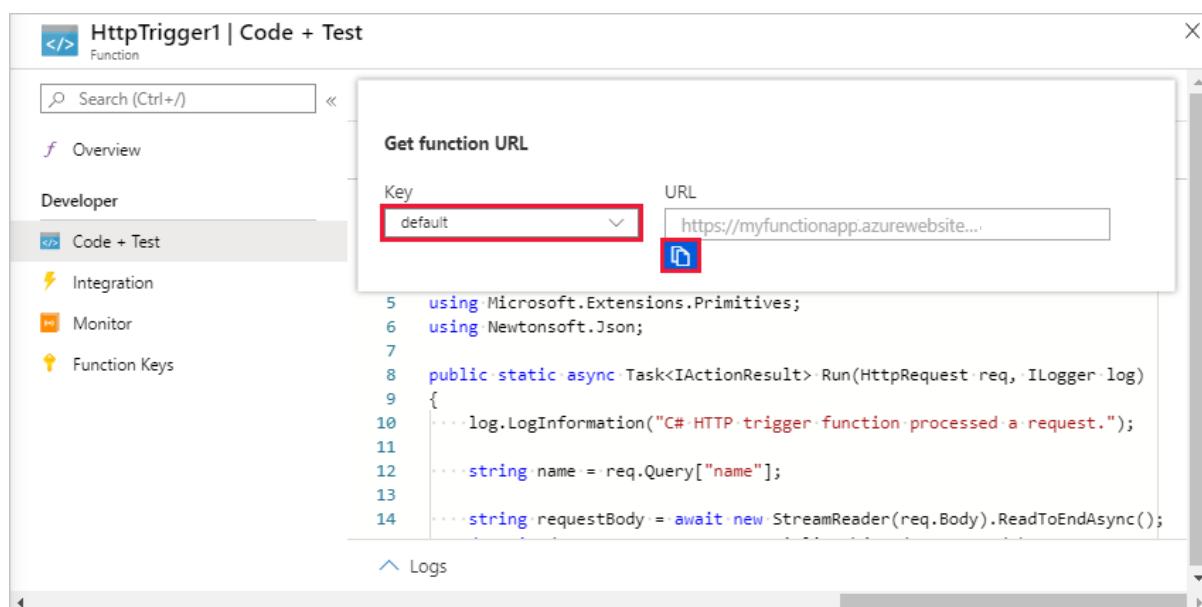
Azure creates the HTTP trigger function. Now, you can run the new function by sending an HTTP request.

Test the function

1. In your new HTTP trigger function, select **Code + Test** from the left menu, then select **Get function URL** from the top menu.

```
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(HttpContext req, ILogger log)
9  {
10    ....log.LogInformation("C# HTTP trigger function processed a request.");
11
12    string name = req.Query["name"];
13
14    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
```

2. In the Get function URL dialog box, select default from the drop-down list, and then select the Copy to clipboard icon.



3. Paste the function URL into your browser's address bar. Add the query string value ?name=<your_name> to the end of this URL and press Enter to run the request.

The following example shows the response in the browser:



The request URL includes a key that is required, by default, to access your function over HTTP.

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.

```

2020-03-26T22:38:16Z [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=8ed59e60-f1bb-44d6-b8cc-e0cb3f76cd4e)
2020-03-26T22:38:16Z [Information] C# HTTP trigger function processed a request.
2020-03-26T22:38:16Z [Information] Executing 'Functions.HttpTrigger1' (Reason=This function was programmatically called via the host APIs., Id=8ed59e60-f1bb-44d6-b8cc-e0cb3f76cd4e)

```

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 have created in this quickstart, do not 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 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 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**.

| 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 |
| Metrics Features (8) Notifications (0) Quickstart | |

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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 have created a function app with a simple HTTP trigger function.

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](#)

For more information, see [Azure Functions HTTP bindings](#).

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

12/4/2020 • 9 minutes to read • [Edit Online](#)

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.

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](#).
- The [Azure Functions Core Tools](#) version 3.x.
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - [Azure PowerShell](#) version 5.0 or later.
- [Python 3.8 \(64-bit\)](#), [Python 3.7 \(64-bit\)](#), [Python 3.6 \(64-bit\)](#), which are all supported by version 3.x of Azure Functions.

Prerequisite check

Verify your prerequisites, which depend on whether you are using Azure CLI or Azure PowerShell for creating Azure resources:

- [Azure CLI](#)
- [Azure PowerShell](#)
- In a terminal or command window, run `func --version` to check that the Azure Functions Core Tools are version 3.x.
- Run `az --version` to check that the Azure CLI version is 2.4 or later.
- Run `az login` to sign in to Azure and verify an active subscription.
- Run `python --version` (Linux/macOS) or `py --version` (Windows) to check your Python version reports 3.8.x, 3.7.x or 3.6.x.

Create and activate a virtual environment

In a suitable folder, run the following commands to create and activate a virtual environment named `.venv`. Be sure to use Python 3.8, 3.7 or 3.6, which are supported by Azure Functions.

- [bash](#)
- [PowerShell](#)
- [Cmd](#)

```
python -m venv .venv
```

```
source .venv/bin/activate
```

If Python didn't install the venv package on your Linux distribution, run the following command:

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment.

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:

```
func init LocalFunctionProj --python
```

2. Navigate into the project folder:

```
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).

```
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.

`__init__.py`

`__init__.py` contains a `main()` Python function that's triggered according to the configuration in `function.json`.

```

import logging

import azure.functions as func


def main(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
        )

```

For an HTTP trigger, the function receives request data in the variable `req` as defined in `function.json`. `req` is an instance of the [azure.functions.HttpRequest class](#). The return object, defined as `$return` in `function.json`, is an instance of [azure.functions.HttpResponse class](#). To learn more, see [Azure Functions HTTP triggers and bindings](#).

function.json

`function.json` is a configuration file that defines the input and output `bindings` for the function, including the trigger type.

You can change `scriptFile` to invoke a different Python file if desired.

```
{
    "scriptFile": "__init__.py",
    "bindings": [
        {
            "authLevel": "function",
            "type": "httpTrigger",
            "direction": "in",
            "name": "req",
            "methods": [
                "get",
                "post"
            ]
        },
        {
            "type": "http",
            "direction": "out",
            "name": "$return"
        }
    ]
}
```

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:

```
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  
...
```

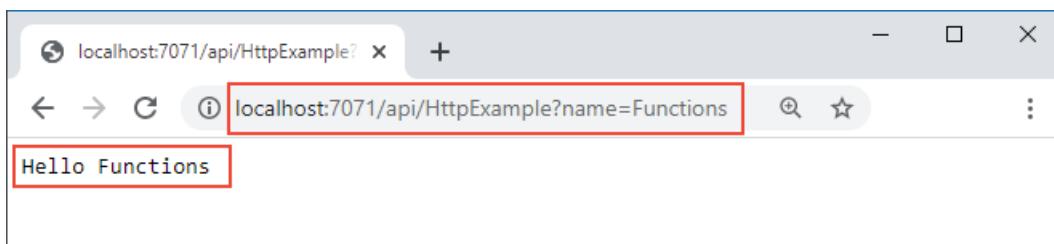
NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. 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 maintains 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 you haven't done so already, sign in to Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az login
```

The [az login](#) command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in the `westeurope` region.

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az group create --name AzureFunctionsQuickstart-rg --location westeurope
```

The [az group create](#) command creates a resource group. You generally create your resource group and resources in a region near you, using an available region 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 PowerShell](#)

```
az storage account create --name <STORAGE_NAME> --location westeurope --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 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 storage account incurs only a few cents (USD) for this quickstart.

4. Create the function app in Azure:

- [Azure CLI](#)
- [Azure PowerShell](#)

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location  
westeurope --runtime python --runtime-version 3.8 --functions-version 3 --name <APP_NAME> --  
storage-account <STORAGE_NAME> --os-type linux
```

The [az functionapp create](#) command creates the function app in Azure. If you are using Python 3.7 or 3.6, change `--runtime-version` to `3.7` or `3.6`, respectively.

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.

```
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.

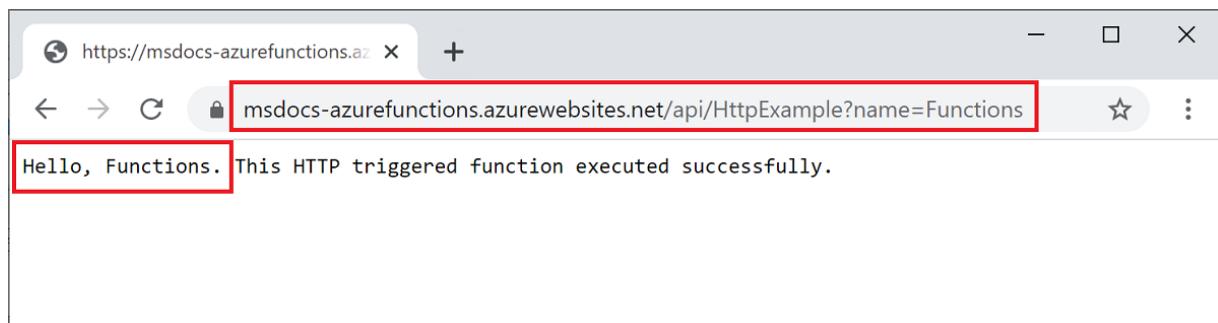
...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
HttpExample - [httpTrigger]
Invoke url: https://msdocs-azurefunctions-qs.azurewebsites.net/api/httpexample
```

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](#)
- [curl](#)

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](#) in Application Insights in the Azure portal:

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

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 PowerShell](#)

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

Next steps

[Connect to an Azure Storage queue](#)

[Having issues? Let us know.](#)

Create a function using Azure for Students Starter

12/4/2020 • 6 minutes to read • [Edit Online](#)

In this tutorial, we'll create a "hello world" HTTP function in an Azure for Students Starter subscription. We'll also walk through what's available in Azure Functions in this subscription type.

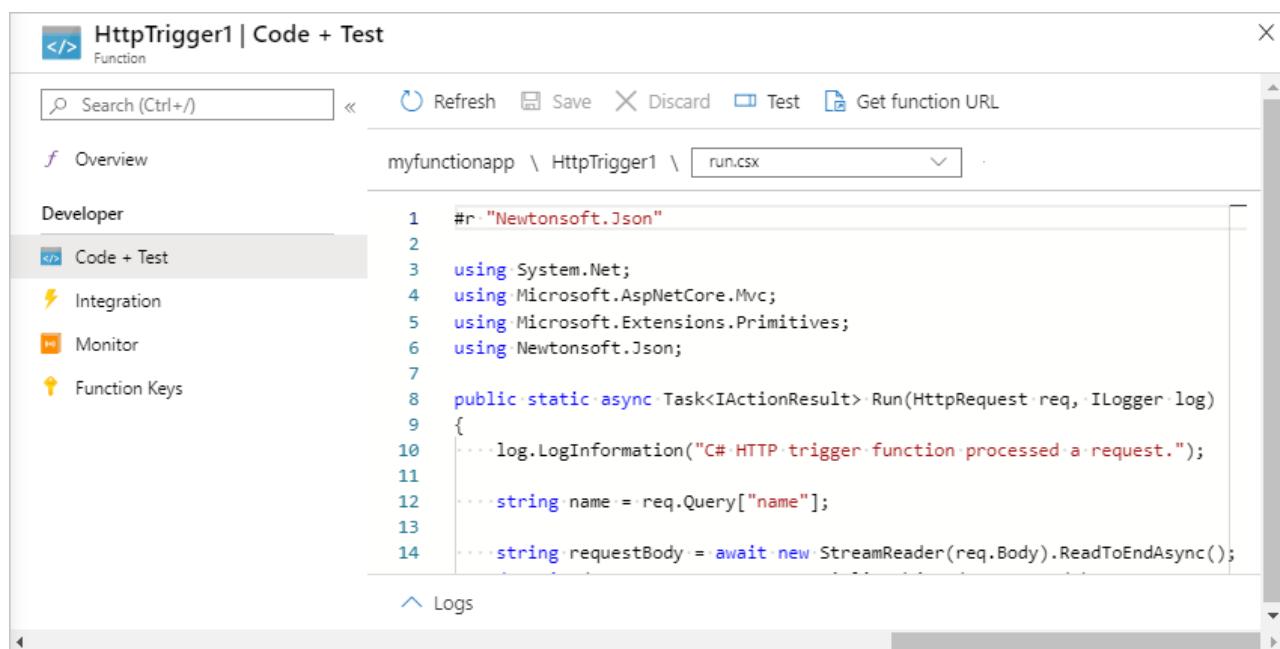
Microsoft *Azure for Students Starter* gets you started with the Azure products you need to develop in the cloud at no cost to you. [Learn more about this offer here.](#)

Azure Functions lets you execute your code in a **serverless** environment without having to first create a VM or publish a web application. [Learn more about Functions here.](#)

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

Create a function

In this article, learn how to use Azure Functions to create an "hello world" HTTP trigger function in the Azure portal.



The screenshot shows the Azure portal interface for creating an Azure Function. The title bar says "HttpTrigger1 | Code + Test". The left sidebar has tabs for Overview, Code + Test (which is selected), Integration, Monitor, and Function Keys. The main area shows the function's path: myfunctionapp \ HttpTrigger1 \ run.csx. The code editor contains the following C# code:

```
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(HttpContext req, ILogger log)
9 {
10     log.LogInformation("C# HTTP trigger function processed a request.");
11
12     string name = req.Query["name"];
13
14     string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
```

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.

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. Choose .NET Core for C# and F# functions. |
| 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. |

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 * [Visual Studio Enterprise](#)
 Resource Group * [\(New\) myResourceGroup](#) [Create new](#)

Instance Details

Function App name * [myfunctionapp](#) [.azurewebsites.net](#)

Publish * [Code](#) [Docker Container](#)

Runtime stack * [.NET Core](#)

Version * [3.1](#)

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 | 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. When you run in an App Service plan, you must manage the scaling of your function app . |

Function App X

[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 * ▼
[Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux Windows

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#) ↗

Plan type * ○ ▼

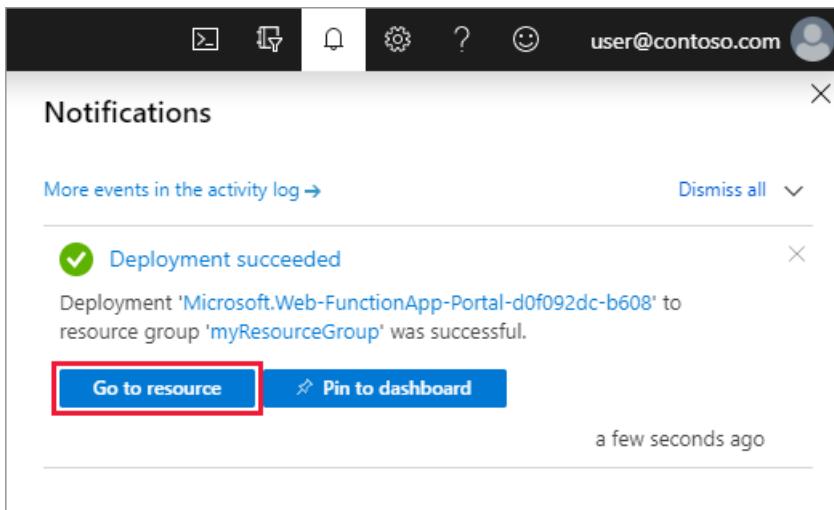
Review + create
 < Previous
 Next : Monitoring >

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. |

The screenshot shows the Azure portal's "Function App" creation interface. The "Monitoring" tab is selected. Under "Application Insights", "Enable Application Insights" is set to "Yes". A dropdown menu shows "(New) myfunctionapp (Central US)" with a "Create new" option. The "Region" is set to "Central US". At the bottom, the "Review + create" button is highlighted with a red box, while "< Previous" and "Next : Tags >" are standard grey buttons.

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.



Next, you create a function in the new 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**.

A screenshot of the Azure portal's 'New Function' creation window. On the left, the sidebar shows 'myfunctionapp | Functions' with sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (selected), App keys, App files, Proxies, Deployment, and Deployment slots. The main area is titled 'New Function' with the sub-instruction 'Create a new function in this Function App. Start by selecting a template below.' It features two tabs: 'Templates' (selected) and 'Details'. Under 'Templates', there is a search bar 'Search by template name'. Four templates are listed:

- HTTP trigger** (selected, highlighted with a red box): 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.

3. In the **New Function** window, accept the default name for **New Function**, or enter a new name.
4. Choose **Anonymous** from the **Authorization level** drop-down list, and then select **Create Function**.

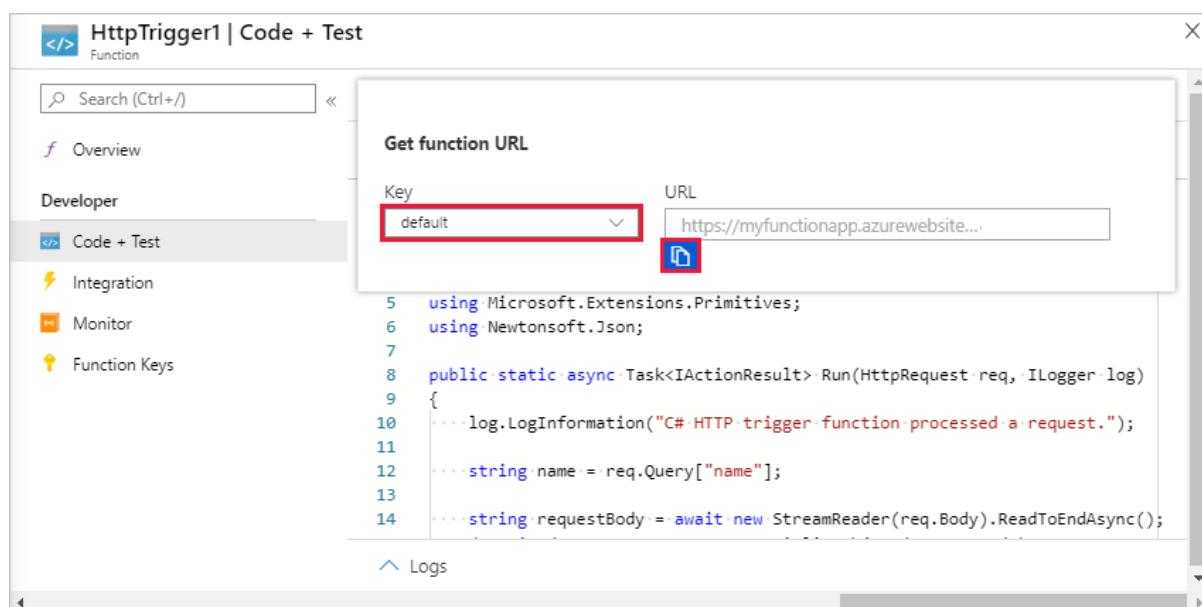
Azure creates the HTTP trigger function. Now, you can run the new function by sending an HTTP request.

Test the function

1. In your new HTTP trigger function, select **Code + Test** from the left menu, then select **Get function URL** from the top menu.

```
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(HttpContext req, ILogger log)
9  {
10    ....log.LogInformation("C# HTTP trigger function processed a request.");
11
12    string name = req.Query["name"];
13
14    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
```

2. In the Get function URL dialog box, select default from the drop-down list, and then select the Copy to clipboard icon.



3. Paste the function URL into your browser's address bar. Add the query string value ?name=<your_name> to the end of this URL and press Enter to run the request.

The following example shows the response in the browser:



The request URL includes a key that is required, by default, to access your function over HTTP.

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.

Logs

Stop Copy Clear Maximize

2020-03-26T22:38:16Z [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=8ed59e60-f1bb-44d6-b8cc-e0cb3f76cd4e)
 2020-03-26T22:38:16Z [Information] C# HTTP trigger function processed a request.
 2020-03-26T22:38:16Z [Information] Executing 'Functions.HttpTrigger1' (Reason=This function was programmatically called via the host APIs., Id=8ed59e60-f1bb-44d6-b8cc-e0cb3f76cd4e)

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 have created in this quickstart, do not 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 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 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**.

myfunctionapp

Search (Ctrl+ /) Overview Activity log Access control (IAM) Tags Diagnose and solve problems Security Functions Functions App keys App files Proxies Metrics 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

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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.

Supported features in Azure for Students Starter

In Azure for Students Starter, you have access to most of the features of the Azure Functions runtime, with several key limitations listed below:

- The HTTP trigger is the only trigger type supported.
 - All input and all output bindings are supported! [See the full list here.](#)

- Languages Supported:
 - C# (.NET Core 2)
 - JavaScript (Node.js 8 & 10)
 - F# (.NET Core 2)
 - [See languages supported in higher plans here](#)
- Windows is the only supported operating system.
- Scale is restricted to [one free tier instance](#) running for up to 60 minutes each day. You'll serverlessly scale from 0 to 1 instance automatically as HTTP traffic is received, but no further.
- Only [version 2.x and later](#) of the Functions runtime is supported.
- All developer tooling is supported for editing and publishing functions. This includes VS Code, Visual Studio, the Azure CLI, and the Azure portal. If you'd like to use anything other than the portal, you'll need to first create an app in the portal, and then choose that app as a deployment target in your preferred tool.

Next steps

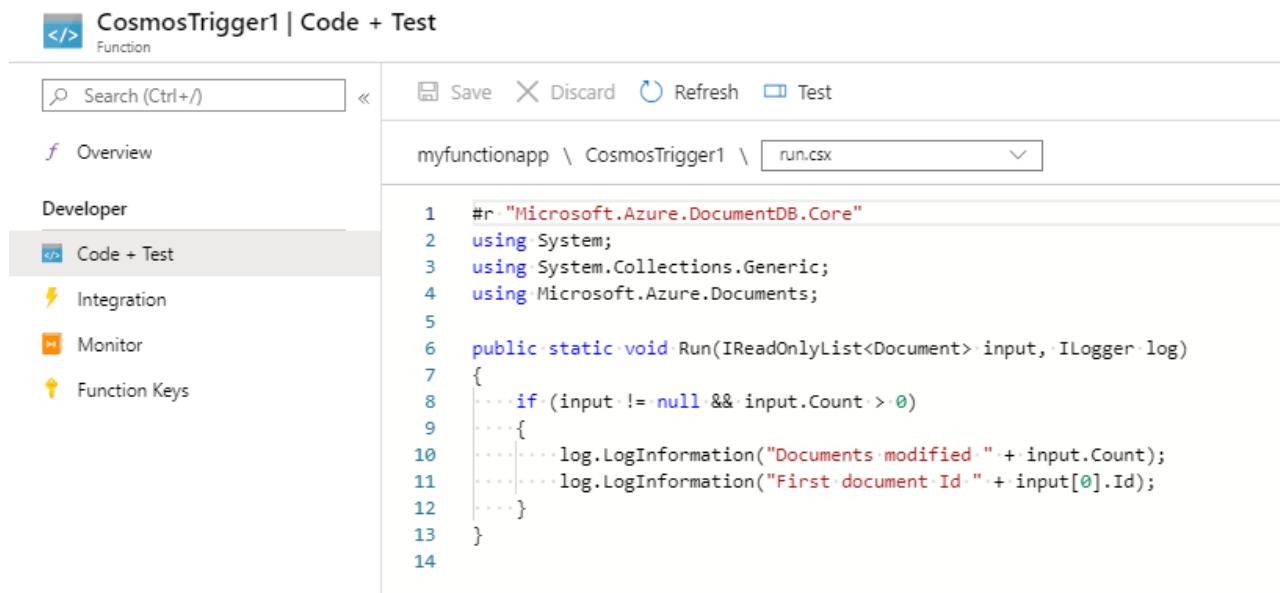
You've now finished creating a function app with a simple HTTP trigger function. Next, you can explore local tooling, more languages, monitoring, and integrations.

- [Create your first function using Visual Studio](#)
- [Create your first function using Visual Studio Code](#)
- [Azure Functions JavaScript developer guide](#)
- [Use Azure Functions to connect to an Azure SQL Database](#)
- [Learn more about Azure Functions HTTP bindings.](#)
- [Monitor your Azure Functions](#)

Create a function triggered by Azure Cosmos DB

12/4/2020 • 9 minutes to read • [Edit Online](#)

Learn how to create a function 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](#).



The screenshot shows the Azure portal interface for a function named "CosmosTrigger1". The left sidebar has "Developer" selected, with "Code + Test" highlighted. The main area shows the file "run.csx" with 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
```

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 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's API for MongoDB](#), [Cassandra API](#), [Gremlin API](#), and [Table API](#).

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. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

| Setting | Value | Description |
|--------------------------|--------------------------------------|---|
| Subscription | Subscription name | Select the Azure subscription that you want to use for this Azure Cosmos 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 | <p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.</p> |
| API | The type of account to create | <p>Select Core (SQL) to create a document database and query by using SQL syntax.</p> <p>The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.</p> |
| 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 Free Tier Discount | Apply or Do not apply | With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier . |
| 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. |

| SETTING | VALUE | DESCRIPTION |
|--------------|------------------------------|---|
| Account Type | Production or Non-Production | Select Production if the account will be used for a production workload. Select Non-Production if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime. |

NOTE

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 do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

Microsoft Azure (Preview) Search resources, services, and docs (G+ /)

Home > New > Create Azure Cosmos DB Account

[Basics](#) [Networking](#) [Backup Policy](#) [Encryption](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. [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 * SubscriptionName

Resource Group * (New) my-resource-group [Create new](#)

Instance Details

Account Name * Enter account name

API * Core (SQL)

Notebooks (Preview) Off

Location * (US) West US

Capacity mode Provisioned throughput [Learn more about capacity mode](#)

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$/month discount per account.

Apply Free Tier Discount Apply Do Not Apply

Account Type Production Non-Production

Geo-Redundancy Enable Disable

Multi-region Writes Enable Disable

[Review + create](#) [Previous](#) [Next: Networking](#)

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.
6. 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**.

Dashboard > Microsoft.Azure.CosmosDB-20190321000000 - Overview

Microsoft.Azure.CosmosDB-20190321000000 - Overview

Deployment

Overview Your deployment is complete [Go to resource](#)

Deployment name: Microsoft.Azure.CosmosDB-20190321000000
Subscription: Contoso Subscription
Resource group: myResourceGroup

DEPLOYMENT DETAILS [\(Download\)](#)
Start time: 3/21/2019, 5:00:03 PM
Duration: 5 minutes 38 seconds
Correlation ID: 8e0be948-0c60-4da0-0000-000000000000

| RESOURCE | TYPE | STATUS | OPERATION DETAILS |
|------------------|-------------------------------------|--------|-----------------------------------|
| mysqlapicosmosdb | Microsoft.DocumentDb/databaseAcc... | OK | Operation details |

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the Azure portal interface for a Cosmos DB account named 'mysqlapicosmosdb'. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, Replicate data globally, Default consistency, and Firewall and virtual networks. The main content area displays a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then instructs the user to connect to it using a sample app, with a 'Choose a platform' section showing options for .NET, .NET Core, Xamarin, Java, Node.js, and Python. Step 1, 'Add a collection', guides the user to create an 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput. Step 2, 'Download and run your .NET app', provides a link to download a sample .NET app. A note at the bottom states: 'Once collection is created, download a sample .NET app connected to it, extract, build and run.'

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. 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. Choose .NET Core for C# and F# functions. |
| 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. |

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 * | Visual Studio Enterprise |
| Resource Group * | (New) myResourceGroup |
| | Create new |

Instance Details

| | |
|---------------------|---|
| Function App name * | myfunctionapp |
| | .azurewebsites.net |
| Publish * | Code Docker Container |
| Runtime stack * | .NET Core |
| Version * | 3.1 |
| 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 | 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. When you run in an App Service plan, you must manage the scaling of your function app . |

Function App X

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 * ▼
[Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux **Windows**

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * ▼

[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. |

Function App

Basics Hosting Monitoring **Tags** Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

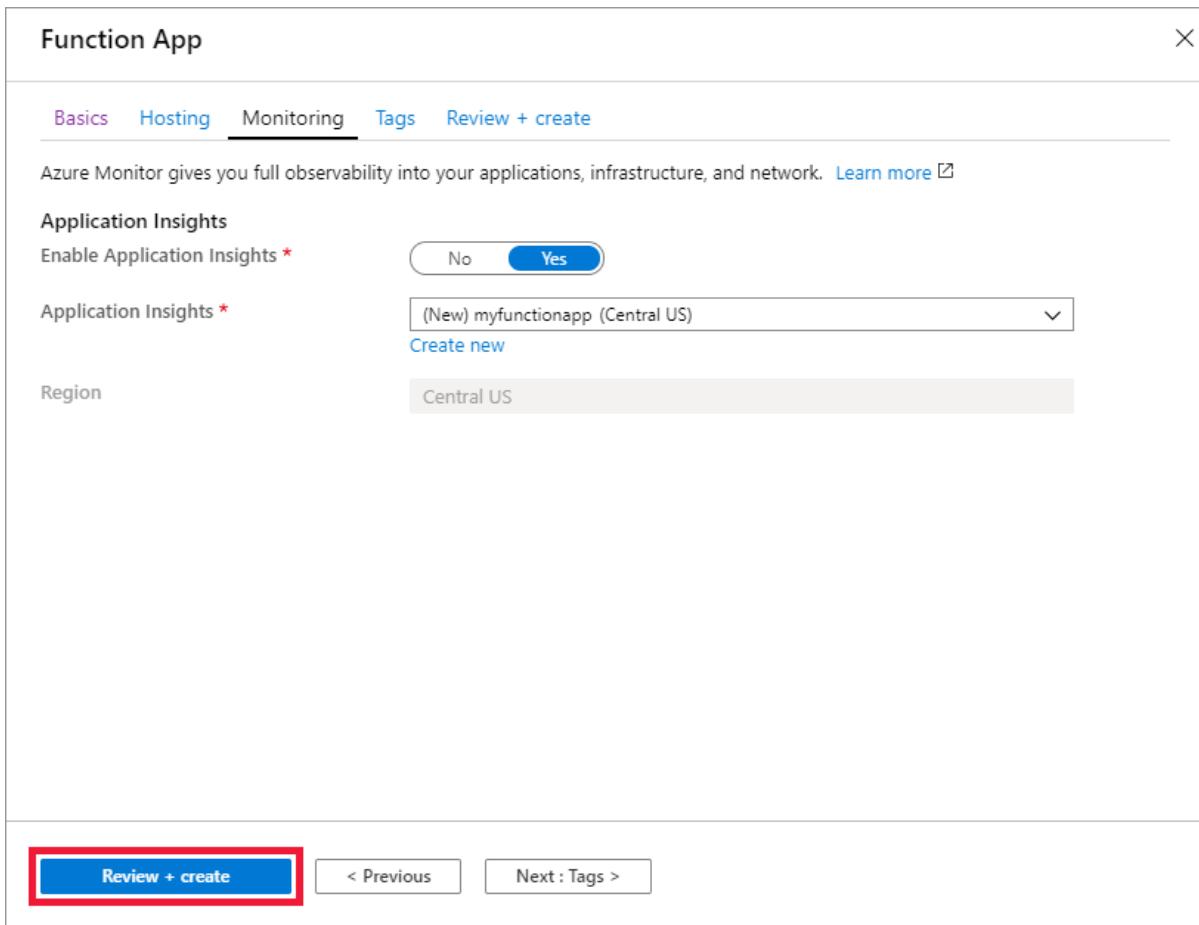
Create new

Region

Central US

Review + create

< Previous Next : Tags >



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.

Notifications

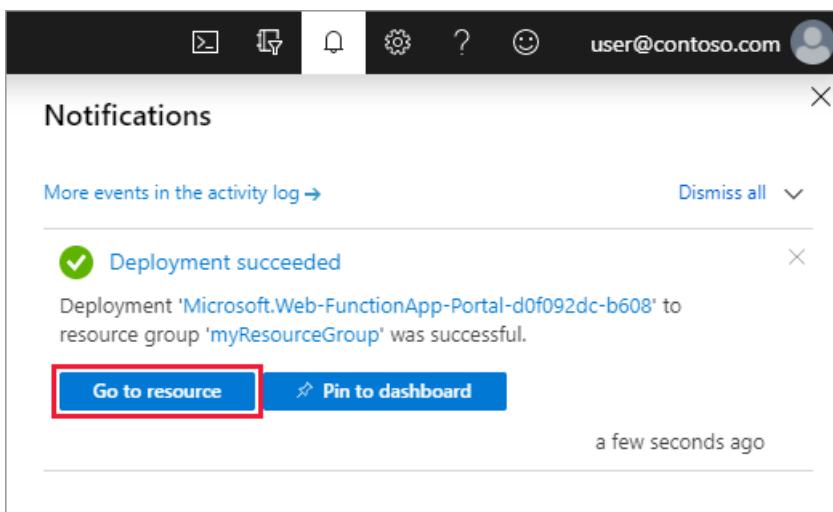
More events in the activity log → Dismiss all

Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource Pin to dashboard

a few seconds ago



Next, you create a function in the new function app.

Create Azure Cosmos DB trigger

1. In your function app, select **Functions** from the left menu, and then select **Add** from the top menu.

2. On the **New Function** page, enter `cosmos` in the search field and then choose the **Azure Cosmos DB trigger** template.

The screenshot shows the Azure Functions blade for the 'myfunctionapp' App Service. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (selected), App keys, App files, and Proxies. A red box highlights the 'Functions' link. On the right, the 'New Function' blade is open. It has a search bar at the top with 'cosmos' typed in. Below it, there are two tabs: 'Templates' (selected) and 'Details'. Under 'Templates', a search result for 'Azure Cosmos DB trigger' is shown, also highlighted with a red box. The description below the icon says: 'A function that will be run whenever documents change in a document collection'.

3. Configure the new trigger with the settings as specified in the following table:

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---|-----------------------------|--|
| New Function | Accept the default name | The name of the function. |
| 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. |

New Function

Create a new function in this function app. Start by selecting a template below.

Templates Details

New Function*

CosmosTrigger1

Cosmos DB account connection* ⓘ

dmab-cosmosdb_DOCUMENTDB (new) ▾

New

Database name* ⓘ

Tasks

Collection name* ⓘ

Items

Collection name for leases* ⓘ

leases

Create lease collection if it does not ex...* ⓘ

Yes

Create Function

4. Select Create Function.

Azure creates the Cosmos DB trigger function.

5. To display the template-based function code, select Code + Test.

CosmosTrigger1 | Code + Test

Function

Search (Ctrl+ /)

Save Discard Refresh Test

myfunctionapp \ CosmosTrigger1 \ run.csx

Developer

Code + Test

Integration

Monitor

Function Keys

```
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
```

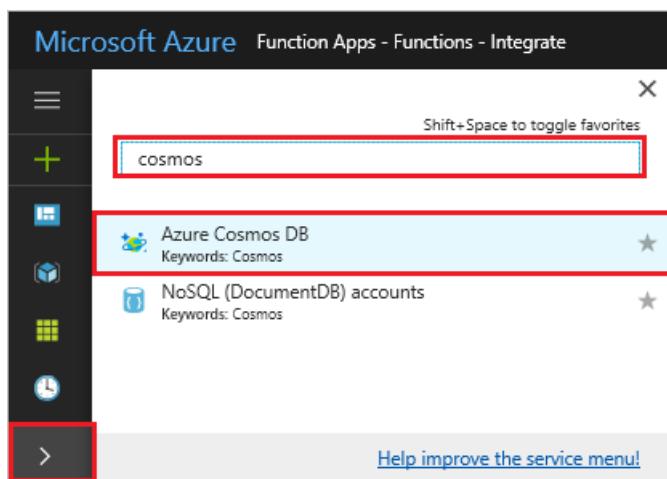
This function template writes the number of documents and the first document ID to the logs.

Next, you connect to your Azure Cosmos DB account and create the `Items` container in the `Tasks` database.

Create the Items container

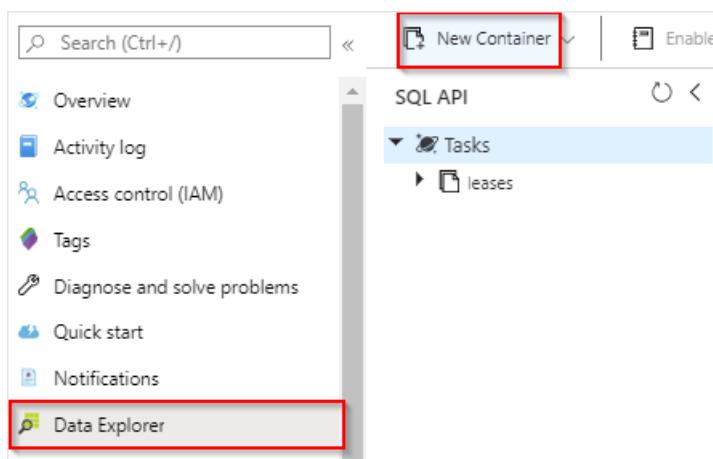
1. Open a second instance of the [Azure portal](#) in a new tab in the browser.
2. On the left side of the portal, expand the icon bar, type `cosmos` in the search field, and select **Azure Cosmos**

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.

| |
|---|
| * Database id ⓘ |
| <input type="radio"/> Create new <input checked="" type="radio"/> Use existing |
| Tasks |
| * Container id ⓘ |
| Items |
| * Partition key ⓘ |
| /category |
| <input type="checkbox"/> My partition key is larger than 100 bytes |
| * Throughput (400 - 100,000 RU/s) ⓘ |
| <input type="radio"/> Autopilot (preview) <input checked="" type="radio"/> Manual |
| 400 |

Unique keys ⓘ

+ Add unique key

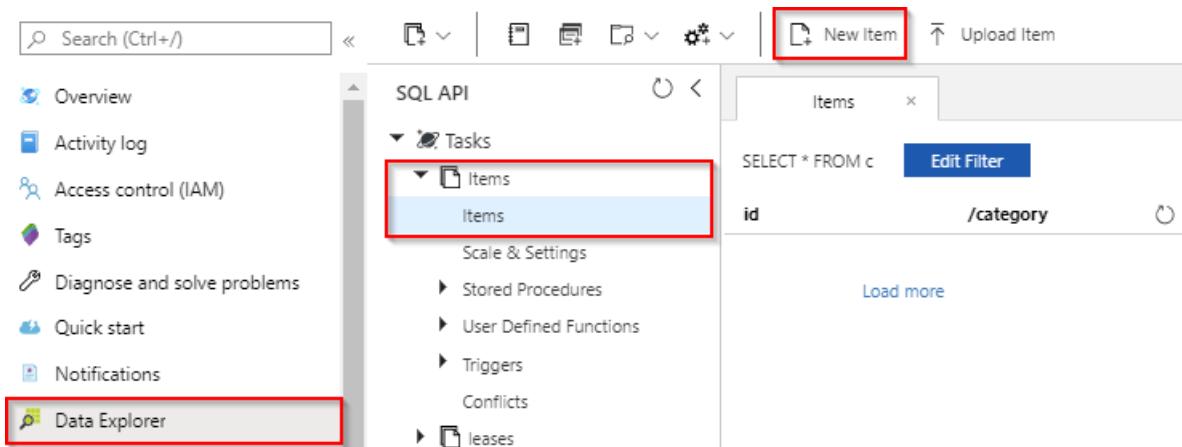
| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------|-----------------|--|
| Database ID | Tasks | The name for your new database. This must match the name defined in your function binding. |
| Container ID | Items | The name for the new container. This must match the name defined in your function 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

- Expand the new **Items** container in Data Explorer, choose **Items**, then select **New Item**.



- Replace the contents of the new item with the following content, then choose **Save**.

```
{
  "id": "task1",
  "category": "general",
  "description": "some task"
}
```

- 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.

```

Logs
Pause Clear Copy logs Expand ▾

2017-09-15T20:38:39 Welcome, you are now connected to log-streaming service.
2017-09-15T20:39:39 No new trace in the past 1 min(s).
2017-09-15T20:40:00.298 Function started (Id=8f09badd-419b-42a5-91cf-a200314ed250)
2017-09-15T20:40:00.316 Documents modified 1
2017-09-15T20:40:00.316 First document Id task1
2017-09-15T20:40:00.316 Function completed (Success, Id=8f09badd-419b-42a5-91cf-a200314ed250, Duration=31ms)

< >

```

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 have created in this quickstart, do not 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 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 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**.

| myfunctionapp | | App Service | |
|--|--|---|---|
| <input type="text"/> Search (Ctrl+/ Overview Resource group (change) myResourceGroup | | Browse Refresh Stop Restart Swap Get publish profile Reset publish profile Delete | |
| Activity log Access control (IAM) Tags Diagnose and solve problems Security | | Status Running Location Central US Subscription Visual Studio Enterprise Subscription ID 11111111-1111-1111-1111-111111111111 Tags Click here to add tags | |
| Functions <ul style="list-style-type: none"> Functions App keys App files Proxies | | 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 | |
| | | Metrics | Features (8) Notifications (0) Quickstart |

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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 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

5/12/2020 • 5 minutes to read • [Edit Online](#)

Learn how to create a function triggered when files are uploaded to or updated in a Blob storage container.

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. 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. Choose .NET Core for C# and F# functions. |
| 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. |

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 * | Visual Studio Enterprise |
| Resource Group * | (New) myResourceGroup |
| | Create new |

Instance Details

| | |
|---------------------|---|
| Function App name * | myfunctionapp |
| | .azurewebsites.net |
| Publish * | Code Docker Container |
| Runtime stack * | .NET Core |
| Version * | 3.1 |
| 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 | 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. When you run in an App Service plan, you must manage the scaling of your function app . |

Function App X

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 * ▼
[Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux Windows

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * ▼

[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. |

Function App

Basics Hosting Monitoring **Tags** Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

Create new

Region

Central US

Review + create < Previous Next : Tags >

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.

✉️ 📈 🔔 ⚙️ ? 😊 user@contoso.com

Notifications

More events in the activity log → Dismiss all

Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource ⚡ Pin to dashboard

a few seconds ago

You've successfully created your new function app.

myFunctionApp-dma

Resource group (change) : myResourceGroup-dma

Status : Running

Location : Central US

Subscription (change) : APEX C+L - Aquent Vendor Subscriptions

Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdbaf52c

Tags (change) : Click here to add tags

URL : https://myfunctionapp-dma.azurewebsites.net

Operating System : Windows

App Service Plan : ASP-myResourceGroupdma-82be (Y1: 0)

Properties : See More

Runtime version : 3.0.13139.0

Metrics Features (9) Notifications (0) Quickstart

Memory working set

160MB
140MB
120MB
100MB
80MB
60MB
40MB
20MB
0B

8:45 PM 9 PM 9:15 PM 9:30 PM UTC-07:00

Memory working set (Sum)
myfunctionapp-dma
47.8 MB

Function Execution Count

1
0.8
0.6
0.4
0.2
0

8:45 PM 9 PM 9:15 PM 9:30 PM UTC-07:00

Function Execution Count (Sum)
myfunctionapp-dma
1

Next, you create a function in the new function app.

Create an Azure Blob storage triggered function

1. Select **Functions**, and then select **+ Add** to add a new function.

myFunctionApp-dma | Functions

Search (Ctrl+)/

+ Add Develop Locally Refresh Enable Disable Delete

Filter by name...

| Name ↑ | Trigger ↑↓ | Status ↑↓ |
|--------|------------|-----------|
| | | |

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Security

Functions (preview)

Functions

App keys

App files

Proxies

Deployment

2. Choose the **Azure Blob Storage trigger template**.

3. Use the settings as specified in the table below the image.

New Function

Create a new function in this Function App. Start by selecting a template below.

Templates **Details**

New Function *

BlobTrigger|

Path * ⓘ

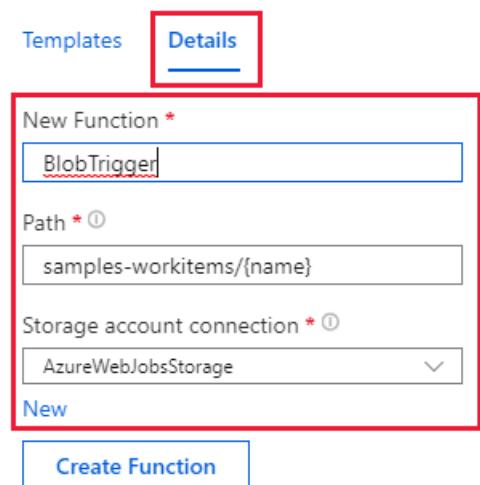
samples-workitems/{name}

Storage account connection * ⓘ

AzureWebJobsStorage

New

Create Function



| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------------------|-----------------------------|--|
| 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. |

4. Select **Create Function** to create your function.

New Function

Create a new function in this Function App. Start by selecting a template below.

Templates **Details**

New Function *

BlobTrigger|

Path * ⓘ

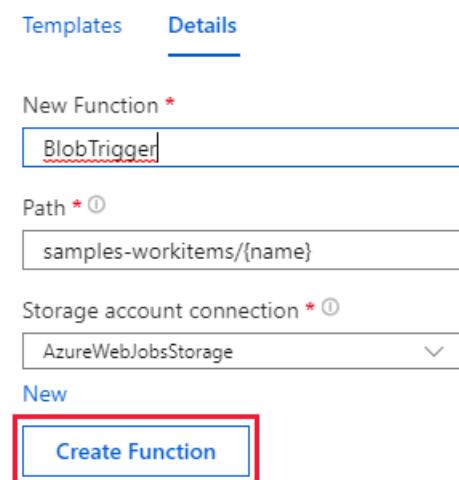
samples-workitems/{name}

Storage account connection * ⓘ

AzureWebJobsStorage

New

Create Function



Next, create the **samples-workitems** container.

Create the container

1. In your function, on the **Overview** page, select your resource group.

myFunctionApp-dma

Resource group (change) : **myResourceGroup-dma**

Status : Running

Location : Central US

Subscription (change) : APEX C+L - Aquent Vendor Subscriptions

Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdabf52c

Tags (change) : Click here to add tags

Metrics Features (9) Notifications (0) Quickstart

Memory working set

Function Execution Count

2. Find and select your resource group's storage account.

Home > myFunctionApps > myFunctionApps > myResourceGroup

myResourceGroup

Subscription (change) : APEX C+L - Aquent Vendor Subscriptions Deployments : 16 Succeeded

Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdabf52c

Tags (change) : Click here to add tags

Filter by name... Type == all Location == all Add filter

Showing 1 to 49 of 49 records. Show hidden types No grouping

| Name | Type | Location |
|--|-------------------|------------|
| myvmpublic2248 | Network interface | East US |
| myVmPublic_c_disk1_c465e5fc5e44cc9a28725e331cc8da3 | Disk | East US |
| myvmpublic356 | Network interface | East US |
| myVmPublic_disk1_ff3f9091009b40fc9ac6f5fd938c5d1 | Disk | East US |
| storageaccountmyres73c | Storage account | Central US |

3. Choose **Containers**, and then choose **+ Container**.

storageaccountmyres85c8 | Containers

Container Change access level Refresh Delete

Search containers by prefix

| Name | Last modified | Public access level | Lease state |
|-----------------------|-----------------------|---------------------|-------------|
| azure-webjobs-hosts | 3/30/2020, 5:47:34 PM | Private | Available |
| azure-webjobs-secrets | 3/30/2020, 5:47:30 PM | Private | Available |
| samples-workitems | 3/30/2020, 5:48:12 PM | Private | Available |

4. In the **Name** field, type `samples-workitems`, and then select **Create**.

New container

Name *

Public access level ⓘ

Private (no anonymous access) ▾

Advanced

Create **Discard**

Now that you have a blob container, you can test the function by uploading a file to the container.

Test the function

- Back in the Azure portal, browse to your function expand the **Logs** at the bottom of the page and make sure that log streaming isn't paused.

BlobTrigger1 | Code + Test

Function

Search (Ctrl+ /) Refresh Save Discard Test

myFunctionApp-dma \ BlobTrigger1 \ run.csx

```
1  public static void Run(Stream myBlob, string name, ILogger log)
2  {
3      log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} bytes");
4  }
5
```

Logs

- In a separate browser window, go to your resource group in the Azure portal, and select the storage account.
- Select **Containers**, and then select the **samples-workitems** container.

Home > myFunctionApp-dma > myFunctionApp-dma > myResourceGroup-dma > storageaccountmyres85c8 | Containers

storageaccountmyres85c8 | Containers

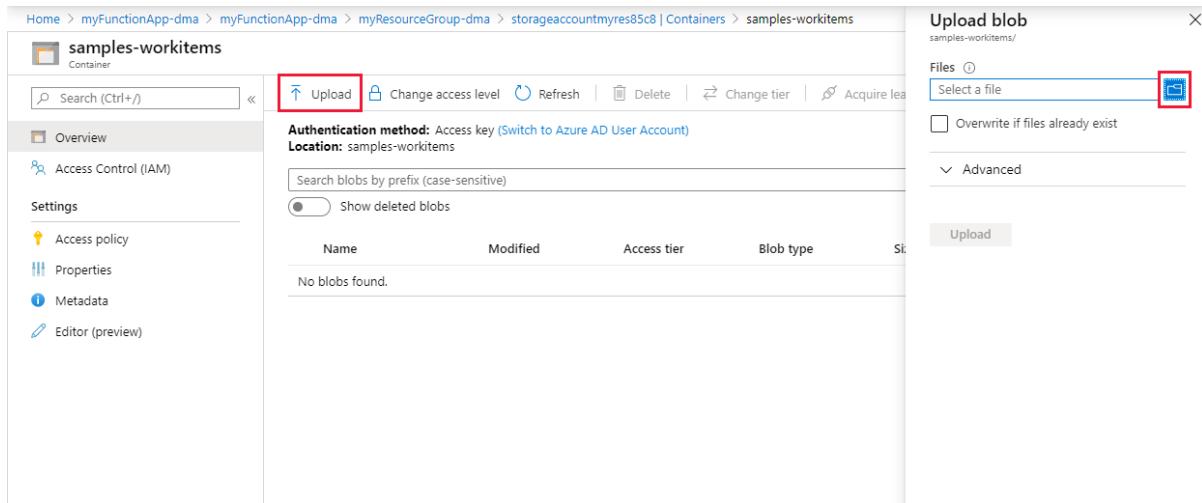
Storage account

Search (Ctrl+ /) Container Change access level Refresh Delete

| Name | Last modified | Public access level | Lease state |
|--------------------------|-----------------------|---------------------|-------------|
| azure-webjobs-hosts | 3/30/2020, 5:47:34 PM | Private | Available |
| azure-webjobs-secrets | 3/30/2020, 5:47:30 PM | Private | Available |
| samples-workitems | 3/30/2020, 5:48:12 PM | Private | Available |

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Data transfer Storage Explorer (preview) Settings Access keys Geo-replication CORS Configuration Encryption

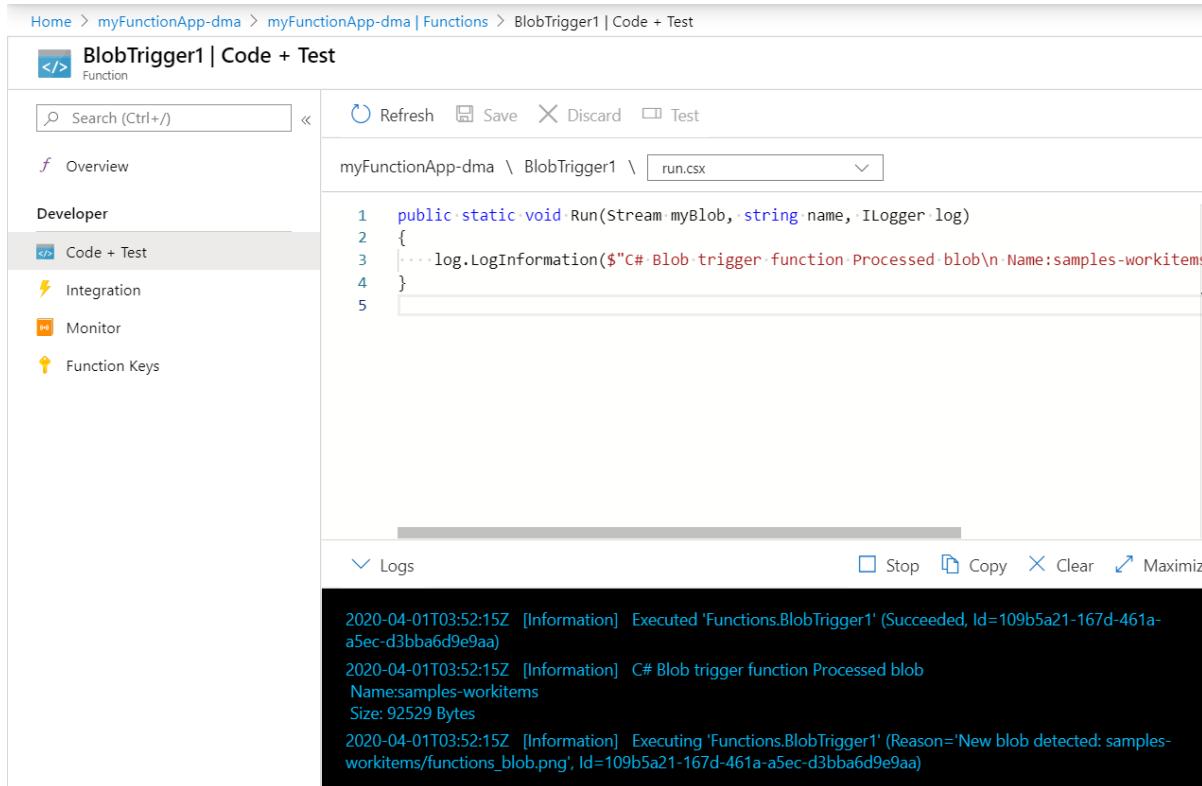
4. Select **Upload**, and then select the folder icon to choose a file to upload.



The screenshot shows the Azure Storage Explorer interface. On the left, there's a sidebar with options like Overview, Access Control (IAM), Settings, and Editor (preview). The main area shows a container named 'samples-workitems'. At the top, there are buttons for Upload, Change access level, Refresh, Delete, Change tier, and Acquire lease. Below these are sections for Authentication method (Access key) and Location (samples-workitems). There's also a search bar for blobs by prefix and a toggle for Show deleted blobs. A table lists blobs with columns Name, Modified, Access tier, Blob type, and Size. A note says 'No blobs found.' On the right, a modal window titled 'Upload blob' is open, showing a 'Files' section with a 'Select a file' input field containing a file icon, and an 'Advanced' section with a checkbox for Overwrite if files already exist. An 'Upload' button is at the bottom of the modal.

5. Browse to a file on your local computer, such as an image file, choose the file. Select **Open** and then **Upload**.

6. Go back to your function logs and verify that the blob has been read.



The screenshot shows the Azure Functions blade for a function named 'BlobTrigger1'. The left sidebar has tabs for Overview, Developer, Code + Test (which is selected), Integration, Monitor, and Function Keys. The main area shows the function's code in a CSX editor. The code defines a static void Run method that logs information about the processed blob. Below the code editor is a 'Logs' panel showing the execution history:

```
2020-04-01T03:52:15Z [Information] Executed 'Functions.BlobTrigger1' (Succeeded, Id=109b5a21-167d-461a-a5ec-d3bba6d9e9aa)
2020-04-01T03:52:15Z [Information] C# Blob trigger function Processed blob
Name:samples-workitems
Size: 92529 Bytes
2020-04-01T03:52:15Z [Information] Executing 'Functions.BlobTrigger1' (Reason='New blob detected: samples-workitems/functions_blob.png', Id=109b5a21-167d-461a-a5ec-d3bba6d9e9aa)
```

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 running your function app in an App Service plan.

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 have created in this quickstart, do not 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 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 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 links for "Functions", "App keys", "App files", and "Proxies". The main content area is titled "Overview" and displays the following details:

| Setting | Value |
|------------------|---|
| Status | Running |
| Location | Central US |
| Subscription | Visual Studio Enterprise |
| Subscription ID | 1111111-1111-1111-1111-111111111111 |
| 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 |

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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 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](#)

Create a function triggered by Azure Queue storage

11/2/2020 • 5 minutes to read • [Edit Online](#)

Learn how to create a function that is triggered when messages are submitted to an Azure Storage queue.

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. 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. Choose .NET Core for C# and F# functions. |
| 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. |

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 * | Visual Studio Enterprise |
| Resource Group * | (New) myResourceGroup |
| | Create new |

Instance Details

| | |
|---------------------|---|
| Function App name * | myfunctionapp |
| | .azurewebsites.net |
| Publish * | Code Docker Container |
| Runtime stack * | .NET Core |
| Version * | 3.1 |
| 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 | 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. When you run in an App Service plan, you must manage the scaling of your function app . |

Function App X

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 * ▼
[Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux **Windows**

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * ▼

[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. |

Function App X

Basics Hosting **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights * Yes No

Application Insights * ▼
[Create new](#)

Region

[Review + create](#) [< Previous](#) [Next : Tags >](#)

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.

X

Notifications

[More events in the activity log →](#) [Dismiss all](#) ▾

✓ Deployment succeeded X

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to
resource group 'myResourceGroup' was successful.

[Go to resource](#) [Pin to dashboard](#)

a few seconds ago

myFunctionApp-dma

Resource group (change) : myResourceGroup-dma

Status : Running

Location : Central US

Subscription (change) : APEX C+L - Aquent Vendor Subscriptions

Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdabf52c

Tags (change) : Click here to add tags

URL : https://myfunctionapp-dma.azurewebsites.net

Operating System : Windows

App Service Plan : ASP-myResourceGroupdma-82be (Y1: 0)

Properties : See More

Runtime version : 3.0.13139.0

Metrics Features (9) Notifications (0) Quickstart

Memory working set

160MB
140MB
120MB
100MB
80MB
60MB
40MB
20MB
0B

8:45 PM 9 PM 9:15 PM 9:30 PM UTC-07:00

Memory working set (Sum)
47.8 MB

Function Execution Count

1
0.8
0.6
0.4
0.2
0

8:45 PM 9 PM 9:15 PM 9:30 PM UTC-07:00

Function Execution Count (Sum)
1

Next, you create a function in the new function app.

Create a Queue triggered function

1. Select **Functions**, and then select **+ Add** to add a new function.

myFunctionApp-dma | Functions

+ Add Develop Locally Refresh | Enable Disable Delete

Filter by name...

| Name ↑ | Trigger ↑ | Status ↑ |
|--------|-----------|----------|
| | | |

Functions (preview)

Functions (highlighted with a red box)

App keys
App files
Proxies

Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Security

Deployment

2. Choose the **Azure Queue Storage trigger template**.

3. Use the settings as specified in the table below the image.

New Function

Create a new function in this Function App. Start by selecting a template below.

Templates **Details**

New Function *

QueueTrigger1

Queue name * ⓘ

myqueue-items

Storage account connection * ⓘ

AzureWebJobsStorage

New

Create Function

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------------------|-----------------------------|--|
| 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. |

4. Select **Create Function** to create your function.

New Function

Create a new function in this Function App. Start by selecting a template below.

Templates **Details**

New Function *

QueueTrigger1

Queue name * ⓘ

myqueue-items

Storage account connection * ⓘ

AzureWebJobsStorage

New

Create Function

Next, you connect to your Azure storage account and create the **myqueue-items** storage queue.

Create the queue

1. In your function, on the Overview page, select your resource group.

myFunctionApp-dma

Overview

Resource group (change) : myResourceGroup-dma

Status : Running

Location : Central US

Subscription (change) : APEX C+L - Aquent Vendor Subscriptions

Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdabf52c

Tags (change) : Click here to add tags

URL : https://myfunctionapp-dma.azurewebsites.net

Operating System : Windows

App Service Plan : ASP-myResourceGroupdma-82be (Y1: 0)

Runtime version : 3.0.13139.0

Properties : See More

Metrics Features (0) Notifications (0) Quickstart

Memory working set Function Execution Count

100MB 90 80 70

2. Find and select your resource group's storage account.

Home > myFunctionApps > myFunctionApps > myResourceGroup

myResourceGroup

Overview

Subscription (change) : APEX C+L - Aquent Vendor Subscriptions Deployments : 16 Succeeded

Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdabf52c

Tags (change) : Click here to add tags

Add Edit columns Delete resource group Refresh Move Export to CSV Assign tags Delete ...

Filter by name... Type == all Location == all Add filter

No grouping

| Name | Type | Location |
|---|-------------------|------------|
| myvmpublic2248 | Network interface | East US |
| myVmPublic_1_c465e5bf5e44cc9a28725e331cc8da3 | Disk | East US |
| myvmpublic356 | Network interface | East US |
| myVmPublic_disk1_ff3f9091009b40fc9ac6f5f3d938c5d1 | Disk | East US |
| storageaccountmyres73c | Storage account | Central US |

3. Choose Queues, and then choose + Queue.

Home > myFunctionApp-dma > myFunctionApp-dma > myResourceGroup-dma > storageaccountmyres85c8 | Queues

storageaccountmyres85c8 | Queues

+ Queue Refresh Delete

Authentication method: Access key (Switch to Azure AD User Account)

Search queues by prefix

| Queue | Url |
|---|--|
| azure-webjobs-blobtrigger-myfunctionapp-dma | https://storageaccountmyres85c8.queue.core.windows.net/... |

File service

Table service

Queue service

Monitoring

Insights (preview)

Alerts

Metrics

Workbooks

Queues

4. In the Name field, type `myqueue-items`, and then select Create.

The screenshot shows the 'Queues' blade in the Azure Storage account 'storageaccountmyres85c8'. On the left, a navigation menu lists 'File service', 'Table service', 'Queue service', and 'Monitoring' sections. Under 'Queue service', 'Queues' is selected. A search bar at the top has 'Search (Ctrl+ /)' placeholder text. Below it are buttons for '+ Queue', 'Refresh', and 'Delete'. A modal dialog box titled 'Add queue' is open, containing a 'Queue name *' input field with 'myqueue-items' typed in. At the bottom of the dialog are 'OK' and 'Cancel' buttons, with 'OK' being highlighted with a red box.

Now that you have a storage queue, you can test the function by adding a message to the queue.

Test the function

1. Back in the Azure portal, browse to your function expand the **Logs** at the bottom of the page and make sure that log streaming isn't paused.

The screenshot shows the 'Code + Test' blade for the function 'QueueTrigger1'. The left sidebar includes 'Overview', 'Developer' (with 'Code + Test' selected), 'Integration', 'Monitor', and 'Function Keys'. 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
```

At the bottom of the blade, there is a 'Logs' section indicated by a red box around the '^' icon.

2. In a separate browser window, go to your resource group in the Azure portal, and select the storage account.
3. Select **Queues**, and then select the **myqueue-items** container.

The screenshot shows the Azure Storage Queues blade for the 'storageaccountmyres85c8' storage account. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, and Storage Explorer (preview). Under Settings, there are sections for Access keys, Geo-replication, CORS, Configuration, and Encryption. The main area displays a list of queues. One queue, 'myqueue-items', is highlighted with a red box. Its URL is shown as <https://storageaccountmyres85c8.queue.core.windows.net/myqueue-items>.

4. Select **Add message**, and type "Hello World!" in **Message text**. Select **OK**.

The screenshot shows the 'myqueue-items' queue details page. The left sidebar has sections for Overview, Access Control (IAM), Settings (with Access policy and Metadata), and a 'Queue' section. The main area features a 'Add message to queue' dialog. It contains a 'Message text' input field (which is empty and highlighted with a red box) and an 'Expires in:' dropdown set to 7 days. A checkbox for 'Encode the message body in Base64' is checked. At the bottom are 'OK' and 'Cancel' buttons.

5. Wait for a few seconds, then go back to your function logs and verify that the new message has been read from the queue.

The screenshot shows the 'QueueTrigger1 | Code + Test' blade. The left sidebar includes Overview, Developer (with Code + Test selected), Integration, Monitor, and Function Keys. The main area shows the 'run.csx' code for the function:1 using System;
2
3 public static void Run(string myQueueItem, ILogger log)
4 {
5 log.LogInformation(\$"C# Queue trigger function processed: {myQueueItem}");
6 }
7Below the code is a 'Logs' section with two entries:2020-04-02T05:05:06Z [Information] Trigger Details: MessageId: 80c38072-c353-499f-8af0-c85ab8e72ec0, DequeueCount: 1, InsertionTime: 4/2/2020 5:04:56 AM +00:00
2020-04-02T05:05:06Z [Information] Executing 'Functions.QueueTrigger1' (Reason='New queue message detected on 'myqueue-items', Id=57d8de7e-4d1e-4ca8-bd93-157cf0f4c49d')

6. 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

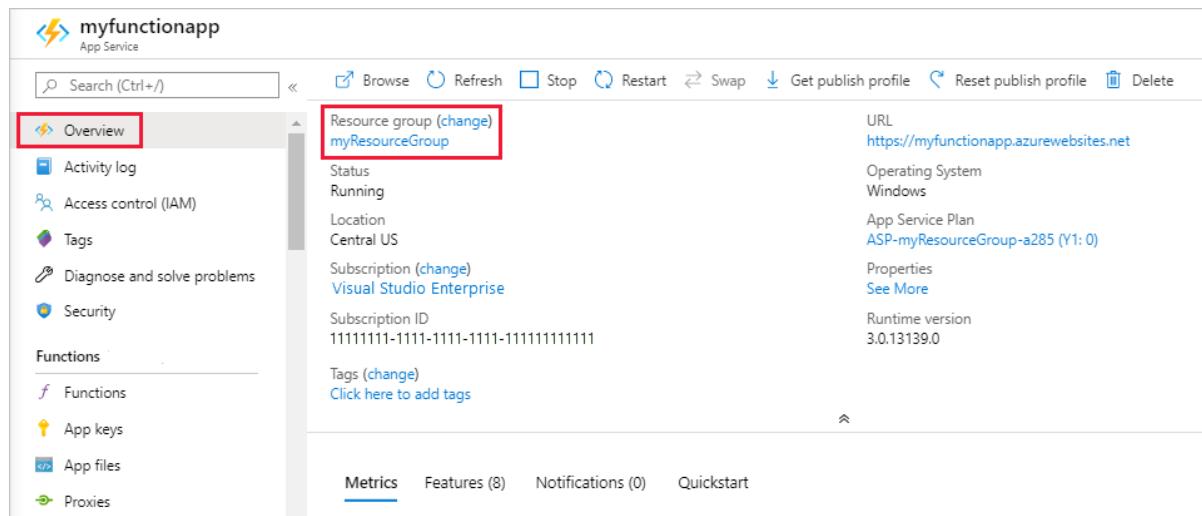
Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you have created in this quickstart, do not clean up the resources.

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 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 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 "Resource group" section with a red box around it, containing the text "Resource group (change) myResourceGroup". The main content area displays various details about the app, such as Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), and App Service Plan (ASP-myResourceGroup-a285 (Y1: 0)). At the bottom, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart, with "Metrics" being the active tab.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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 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 a 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](#)

Create a function in Azure that is triggered by a timer

11/2/2020 • 4 minutes to read • [Edit Online](#)

Learn how to use Azure Functions to create a [serverless](#) function that runs based on a schedule that you define.

Prerequisites

To complete this tutorial:

- If you don't have an Azure subscription, 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. 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. Choose .NET Core for C# and F# functions. |
| 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. |

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 * [Visual Studio Enterprise](#)
 Resource Group * [\(New\) myResourceGroup](#) [Create new](#)

Instance Details

Function App name * [myfunctionapp](#) .azurewebsites.net

Publish * [Code](#) [Docker Container](#)

Runtime stack * [.NET Core](#)

Version * [3.1](#)

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 | 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. When you run in an App Service plan, you must manage the scaling of your 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 * [Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * [Linux](#) [Windows](#)

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * Consumption (Serverless)

[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. |

Function App

Basics Hosting Monitoring **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

Create new

Region

Central US

Review + create

< Previous Next : Tags >

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.

Notifications

More events in the activity log → Dismiss all

✓ Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource Pin to dashboard

a few seconds ago

Your new function app is ready to use. Next, you'll create a function in the new function app.

Create a timer triggered function

1. In your function app, select **Functions**, and then select **+ Add**

2. Select the Timer 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

 **Azure Event Hub trigger**
A function that will be run

 **Azure Cosmos DB trigger**
A function that will be run

3. Configure the new trigger with the settings as specified in the table below the image, and then select **Create Function**.

X

New Function

Create a new function in this function app. Start by selecting a template below.

Templates Details

New Function*

TimerTrigger1

Schedule* ⓘ

0 */1 * * * *

Create Function

| 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.

The screenshot shows the Azure Functions portal interface. At the top, the navigation bar includes 'Home > Function App > myFunctionApp-dma > myFunctionApp-dma | Functions > TimerTrigger1 | Code + Test'. The main title is 'TimerTrigger1 | Code + Test'. On the left, a sidebar under 'Developer' has tabs for 'Overview', 'Code + Test' (which is highlighted with a red box), 'Integration', 'Monitor', and 'Function Keys'. The main area shows the file 'run.csx' with the following code:

```
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
```

2. Verify execution by viewing the information written to the logs.

The screenshot shows the 'Logs' section of the Azure Functions portal. The title bar includes 'Logs', 'Log Level', 'Stop', 'Copy', 'Clear', and 'Maximize'. The main area displays the message 'Connected!' followed by a log entry: '2020-04-17T04:00:00Z [Information] Executing 'Functions.TimerTrigger1' (Reason='Timer fired at 2020-04-17T04:00:00.0159786+00:00', Id=5f8a4f2a-4839-96e7-4b7003866e98)'. The background of the logs area is black.

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 input and output bindings for your function and also set the schedule.
2. Select **Timer (myTimer)**.

Home > Function App > myFunctionApp-dma > myFunctionApp-dma | Functions > TimerTrigger1 | Integration

TimerTrigger1 | Integration

Function

Search (Ctrl+ /) Refresh

Overview

Developer

Code + Test

Integration

Monitor

Function Keys

Integration

Edit the trigger and choose from a selection of inputs and outputs for your function, including Azure Blob Storage, Cosmos DB and others.

Trigger

Timer (myTimer)

Inputs

No inputs defined

+ Add input

Function

TimerTrigger1

Outputs

No outputs defined

+ Add output

The screenshot shows the Azure Function app integration page. The 'Integration' tab is selected in the sidebar. The main area shows the trigger configuration, which includes a 'Trigger' section with 'Timer (myTimer)' selected, and 'Inputs', 'Function', and 'Outputs' sections. The 'Function' section shows 'TimerTrigger1'. The left sidebar has tabs for 'Overview', 'Developer', 'Code + Test', 'Integration' (which is selected and highlighted with a red box), 'Monitor', and 'Function Keys'.

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 * * *

The screenshot shows the 'Edit Trigger' dialog. It has buttons for 'Save' (highlighted with a red box), 'Discard', and 'Delete'. Below are fields for 'Binding Type' (set to 'Timer'), 'Timestamp parameter name' (set to 'myTimer'), and 'Schedule' (set to '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 have created in this quickstart, do not 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 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 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 has a red box around the 'Resource group (change)' section, which displays the resource group name 'myResourceGroup' and various details like status, location, subscription, and tags.

| 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 |

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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 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](#)

Add bindings to an existing function in Azure Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

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. Using Visual Studio Code can make it easier to add bindings to a function.

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](#).

Manually add bindings based on examples

When adding a binding to an existing function, you'll need update both the function code and the function.json configuration file, if used by your language. Both .NET class library and Java functions use attributes instead of function.json, so you'll need to update that instead.

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.

- [C#](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

| SERVICE | EXAMPLES | SAMPLES |
|-----------------|--|----------------------|
| Blob storage | Trigger Input Output | Link |
| Azure Cosmos DB | Trigger Input Output | Link |
| Event Grid | Trigger Output | Link |
| Event Hubs | Trigger Output | |

| SERVICE | EXAMPLES | SAMPLES |
|---------------|--|----------------------|
| IoT Hub | Trigger Output | |
| HTTP | Trigger | Link |
| Queue storage | Trigger Output | Link |
| SendGrid | Output | |
| Service Bus | Trigger Output | Link |
| SignalR | Trigger Input Output | |
| Table storage | Input Output | |
| Timer | Trigger | Link |
| Twilio | Output | Link |

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

11/2/2020 • 6 minutes to read • [Edit Online](#)

Azure Cosmos DB is a great way to store unstructured and JSON data. Combined with Azure Functions, 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. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

| SETTING | VALUE | DESCRIPTION |
|----------------|---------------------|---|
| Subscription | Subscription name | Select the Azure subscription that you want to use for this Azure Cosmos account. |
| Resource Group | Resource group name | Select a resource group, or select Create new , then enter a unique name for the new resource group. |

| Setting | Value | Description |
|--------------------------|--------------------------------------|---|
| Account Name | A unique name | <p>Enter a name to identify your Azure Cosmos account. Because <code>documents.azure.com</code> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.</p> |
| API | The type of account to create | <p>Select Core (SQL) to create a document database and query by using SQL syntax.</p> <p>The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.</p> |
| 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 Free Tier Discount | Apply or Do not apply | With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier . |
| 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. |
| Account Type | Production or Non-Production | Select Production if the account will be used for a production workload. Select Non-Production if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime. |

NOTE

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 do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

Microsoft Azure (Preview) Search resources, services, and docs (G+)

Home > New >

Create Azure Cosmos DB Account

Basics Networking Backup Policy Encryption Tags Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. [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 * SubscriptionName

Resource Group * (New) my-resource-group Create new

Instance Details

Account Name * Enter account name

API * Core (SQL)

Notebooks (Preview) On Off

Location * (US) West US

Capacity mode Provisioned throughput Serverless (Preview) [Learn more about capacity mode](#)

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$[redacted]/month discount per account.

Apply Free Tier Discount

Account Type Production Non-Production

Geo-Redundancy Enable Disable

Multi-region Writes Enable Disable

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.
6. 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 for a function app named "Microsoft.Azure.CosmosDB-20190321000000". The left sidebar has "Deployment" selected. The main area displays a green checkmark indicating "Your deployment is complete". Below it, a "Go to resource" button is available. Deployment details show a start time of 3/21/2019, 5:00:03 PM, a duration of 5 minutes 38 seconds, and a correlation ID of 8e0be948-0c60-4da0-0000-000000000000. A table lists one resource: "mysqlapicosmosdb" of type Microsoft.DocumentDb/databaseAcc... with status OK, and a "Operation details" link.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the "Quick start" page for the Azure Cosmos DB account "mysqlapicosmosdb". The left sidebar has "Quick start" selected. The main area congratulates the user on account creation and provides instructions to connect using a sample app. It offers to "Choose a platform" with options for .NET, .NET Core, Xamarin, Java, Node.js, and Python. Step 1, "Add a collection", explains that data is stored in collections and provides a "Create 'Items' collection" button. Step 2, "Download and run your .NET app", instructs users to download a sample .NET app connected to the collection. A "Download" button is provided.

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

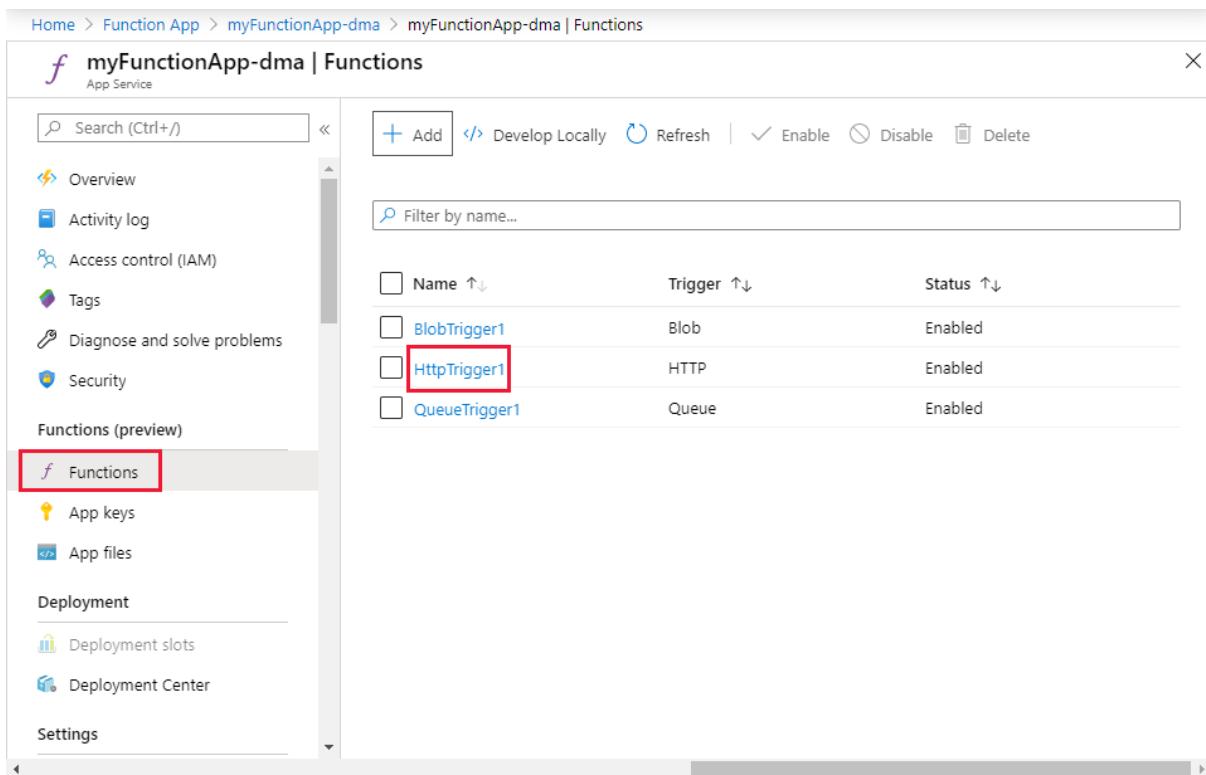
Search (Ctrl+ /) Filter by name...

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Security

Functions (preview)

| Name ↑↓ | Trigger ↑↓ | Status ↑↓ |
|---------------|------------|-----------|
| BlobTrigger1 | Blob | Enabled |
| HttpTrigger1 | HTTP | Enabled |
| QueueTrigger1 | Queue | Enabled |

f Functions App keys App files Deployment Deployment slots Deployment Center Settings



3. Select Integration and + Add output.

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

Function

Trigger

HTTP (req)

Inputs

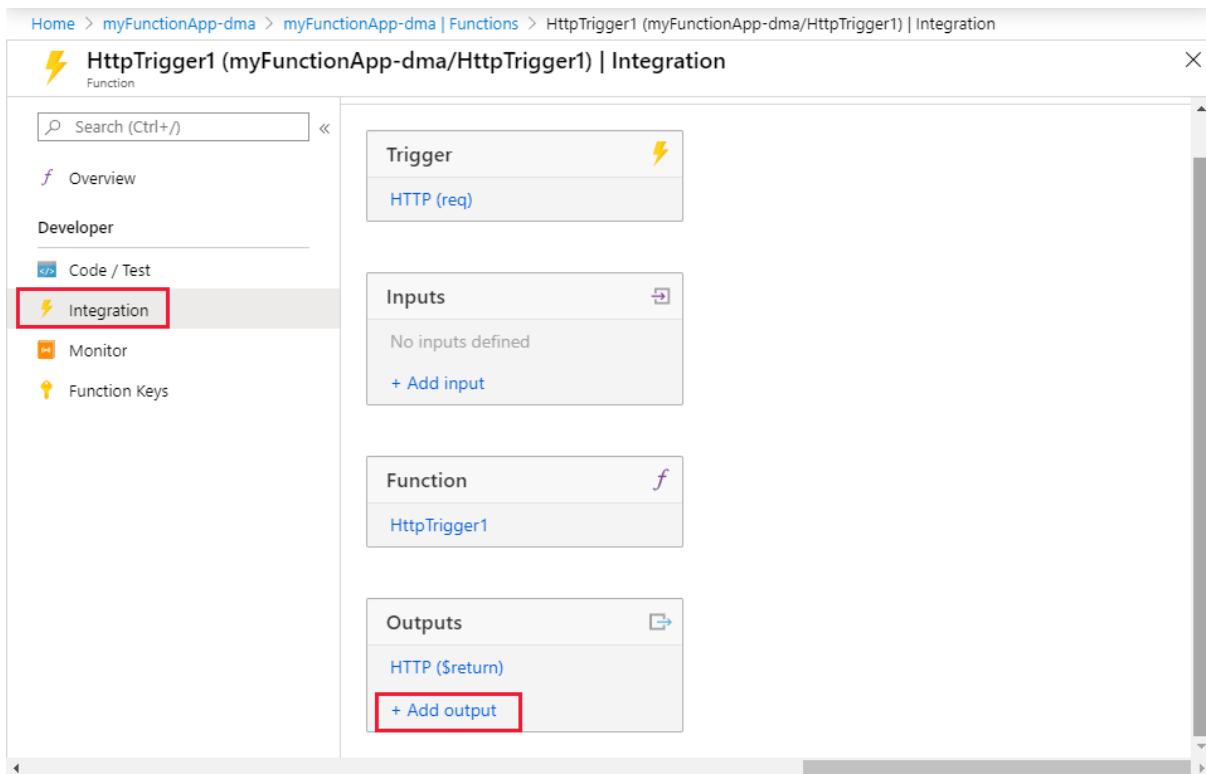
No inputs defined + Add input

Function

HttpTrigger1

Outputs

HTTP (\$return) + Add output



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 database and collection* ⓘ

No

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 Cosmos DB object in code. |
| Database name | taskDatabase | Name of database to save documents. |
| Collection name | taskCollection | Name of the database collection. |
| If true, creates the Cosmos DB database and collection | Yes | The collection doesn't already exist, so create it. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|------------------------------|-----------------|--|
| 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#](#)
- [JavaScript](#)

Replace the existing C# function with the following code:

```
#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

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 | Code + Test

HttpTrigger1 | Code + Test

Function

Save Discard Refresh **Test** ...

myFunctionApp- \ HttpTrigger1 \ run.csx

Input Output

Provide parameters to test the HTTP request. Results can be found in the Output tab.

HTTP method: POST

Key: master (Host key)

Query:

| Name | Value |
|---------|--------------|
| name | Maria Anders |
| duedate | 03/17/2017 |
| task | shopping |

+ Add parameter

Headers:

+ Add header

Run Close

Connected!

2. Select Run and verify that a 200 status is returned.

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 | Code + Test

HttpTrigger1 | Code + Test

Function

Save Discard Refresh **Test** Get function URL

myFunctionApp-dma \ HttpTrigger1 \ run.csx

Input Output

HTTP response code: 200 OK

HTTP response content:

```
2020-04-14T05:10:00Z [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=ce811ba9-dfb4-42d1-a0cb-a7caed0efc5)
2020-04-14T05:09:59Z [Information] Executing 'Functions.HttpTrigger1' (Reason='This function was programmatically called via the host APIs.', Id=ce811ba9-dfb4-42d1-a0cb-a7caed0efc5)
```

Logs Log Level Stop Copy Clear Maximize

Connected!

3. In the Azure portal, search for and select Azure Cosmos DB.

Azure Cosmos DB

Services

- 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.

Home > Resource groups > portal-testing > glengatest-portal | Data Explorer

glengatest-portal | Data Explorer

Items

SELECT * FROM c

| id | /p... |
|---------|---|
| 32be... | "name": "Marie", "duedate": "4/20/20", "task": "shopping", "id": "32bede0a-2b0c-4f1e-8fc2-eae87f36b5a3", "_rid": "27zbApHu48BAAAAAAAA==", "_self": "dbs/27zbApHu48/colls/27zbApHu48/docs/27zbAP", "_etag": "\1b0114c8-0000-0500-0000-5e95426b0000\"", "_attachments": "attachments/", "_ts": 1586848171 |
| ec24... | |
| 9c03... | |
| 7d43... | |
| 566e... | |
| 2c2f... | |
| 0435... | |
| 6fe4... | |

You've successfully added a binding to your HTTP trigger to store unstructured data in an Azure Cosmos DB.

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 a Cosmos DB database, see [Azure Functions 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.

Connect Azure Functions to Azure Storage using command line tools

12/4/2020 • 16 minutes to read • [Edit Online](#)

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.

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.

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.

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.

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.

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 use by the function app. 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 that connection write to a Storage queue in the same account when running the function locally.

- From the root of the project, run the following command, replacing `<app_name>` with the name of your function app from the previous quickstart. This command will overwrite any existing values in the file.

```
func azure functionapp fetch-app-settings <app_name>
```

- Open `local.settings.json` 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 `local.settings.json` 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

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage --version 3.0.4
```

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 let 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:

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "res"
  }
]
```

```
"scriptFile": "__init__.py",
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  }
]
```

```

"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "Response"
  }
]

```

Each binding has at least a type, a direction, and a name. In the example above, the first binding is of type `httpTrigger` with the direction `in`. For the `in` direction, `name` specifies the name of an input parameter that's sent to the function when invoked by the trigger.

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:

```

{
  "authLevel": "function",
  "type": "httpTrigger",
  "direction": "in",
  "name": "req",
  "methods": [
    "get",
    "post"
  ]
},
{
  "type": "http",
  "direction": "out",
  "name": "res"
},
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
]
}

```

The second binding in the collection is of type `http` with the direction `out`, in which case the special `name` of `$return` indicates that this binding uses the function's return value rather than providing an input parameter.

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:

```

"bindings": [
  {
    "authLevel": "anonymous",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  },
  {
    "type": "queue",
    "direction": "out",
    "name": "msg",
    "queueName": "outqueue",
    "connection": "AzureWebJobsStorage"
  }
]

```

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:

```

{
  "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"
  }
]
}

```

In this case, `msg` is given to the function as an output argument. For a `queue` type, you must also specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from *local.settings.json*) in `connection`.

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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.

Update `HttpExample_init_.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```

import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )

```

The `msg` parameter is an instance of the `azure.functions.InputStream class`. Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```

module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```

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
})

```

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```

if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}

```

At this point, your function should look as follows:

```

[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

```
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:

```

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.

```
@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:

```
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:

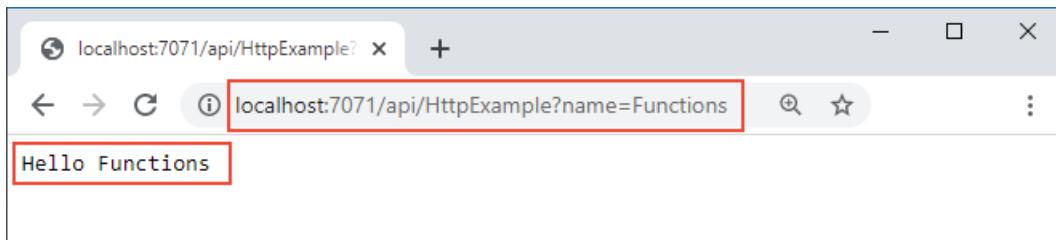
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
...
```

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. When you're done, use `Ctrl+C` and choose `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:

```
{  
    "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`, pasting 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](#)
 - [PowerShell](#)
 - [Azure CLI](#)

```
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 should include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
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 first name you used when testing the function earlier. The command reads and removes the first message from the queue.

- [bash](#)
- [PowerShell](#)
- [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>
```

In the local project folder, use the following Maven command to republish your project:

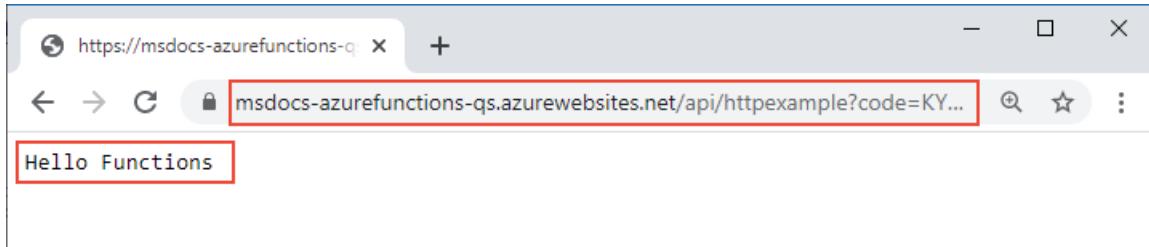
```
mvn azure-functions:deploy
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

- [Browser](#)
- [curl](#)

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.



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.

```
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](#)

- [Examples of complete Function projects in JavaScript.](#)
- [Azure Functions JavaScript developer guide](#)
- [Examples of complete Function projects in TypeScript.](#)
- [Azure Functions TypeScript developer guide](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)
- [Examples of complete Function projects in PowerShell.](#)
- [Azure Functions PowerShell developer guide](#)

Add messages to an Azure Storage queue using Functions

11/2/2020 • 5 minutes to read • [Edit Online](#)

In Azure Functions, input and output bindings provide a declarative way to make data from external services available to your code. In this quickstart, you use an output binding to create a message in a queue when a function is triggered by an HTTP request. You use Azure storage container to view the queue messages that your function creates.

Prerequisites

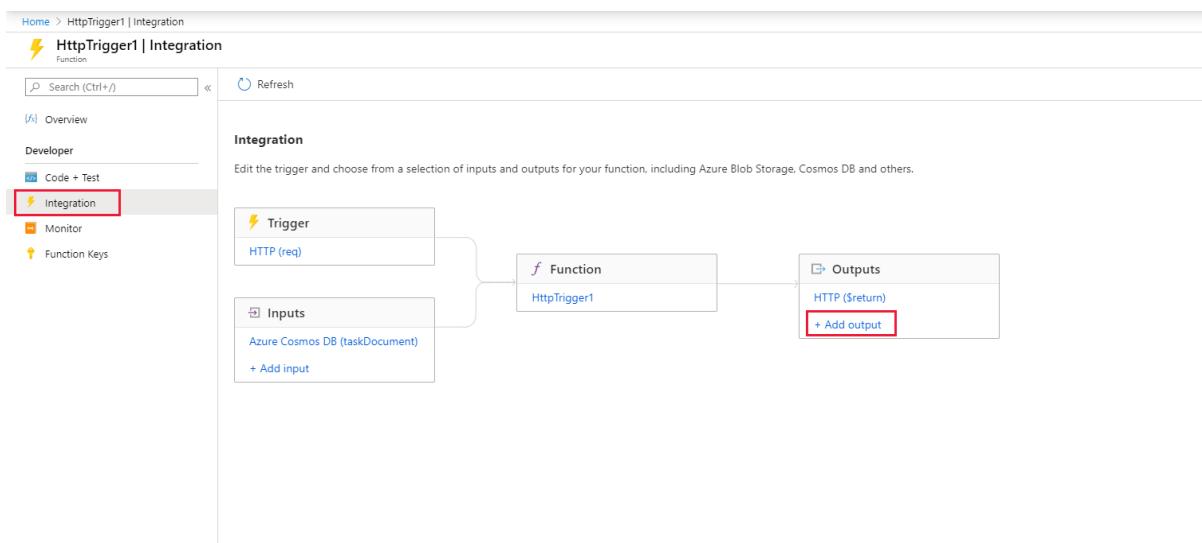
To complete this quickstart:

- An Azure subscription. If you don't have one, create a [free account](#) before you begin.
- Follow the directions in [Create your first function from the Azure portal](#) and don't do the **Clean up resources** step. That quickstart creates the function app and function that you use here.

Add an output binding

In this section, you use the portal UI to add a queue storage output binding to the function you created earlier. This binding makes it possible to write minimal code to create a message in a queue. You don't have to write code for tasks such 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, open the function app page for the function app that you created in [Create your first function from the Azure portal](#). To do open the page, search for and select **Function App**. Then, select your function app.
2. Select your function app, and then select the function that you created in that earlier quickstart.
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

X

Start by selecting the type of output binding you want to add.

Binding Type

Azure Queue Storage

Azure Queue Storage details

Message parameter name* ⓘ

outputQueueItem

Queue name* ⓘ

outqueue

Storage account connection* ⓘ

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 storage account connection already being 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 that is passed to the HTTP trigger in the query string. For example, if the query string includes `name=Azure`, the queue message will be *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 depending on your function language:

- C#
- JavaScript

Add an **outputQueueItem** parameter to the method signature as shown in the following example.

```
public static async Task<IActionResult> Run(HttpContext req,
    ICollector<string> outputQueueItem, ILogger log)
{
    ...
}
```

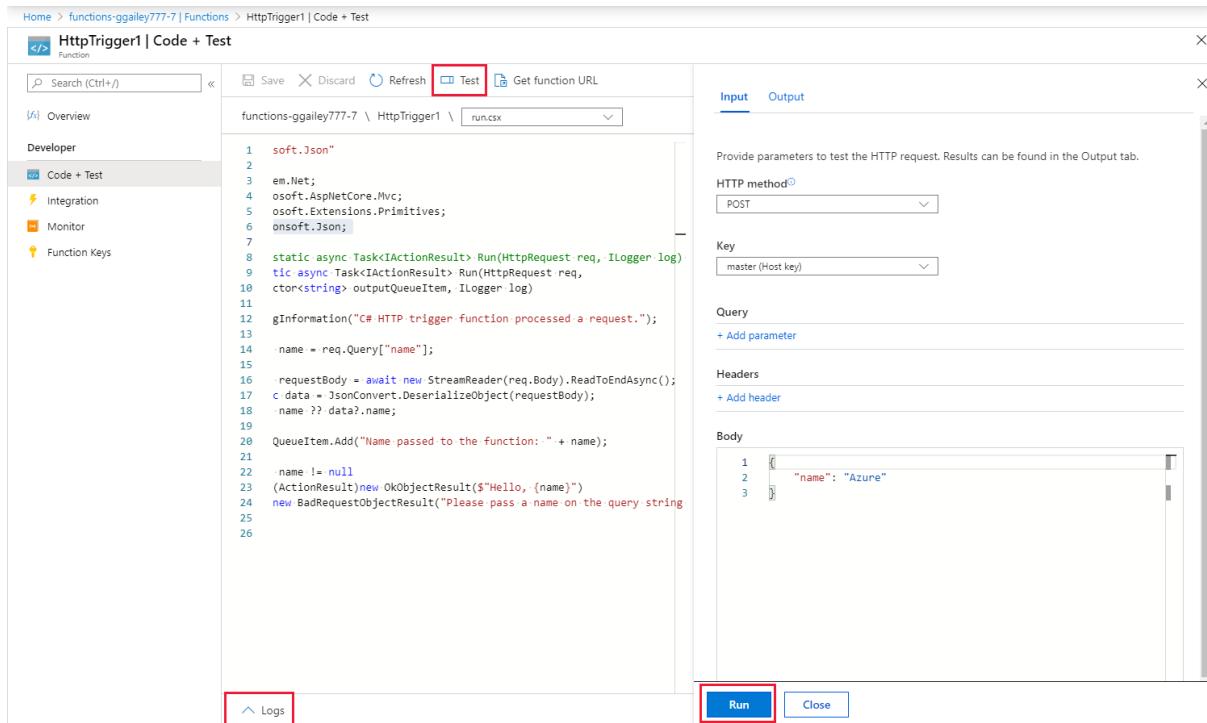
In the body of the function just before the `return` statement, add code that uses the parameter to create a queue message.

```
outputQueueItem.Add("Name passed to the function: " + name);
```

3. Select **Save** to save changes.

Test the function

1. After the code changes are saved, select **Test**.
2. Confirm that your test matches the image below and select **Run**.



Notice that the **Request body** contains the `name` value *Azure*. This value appears in the queue message that is created when the function is invoked.

As an alternative to selecting **Run** here, you can call the function by entering a URL in a browser and specifying the `name` value in the query string. The browser method is shown in the [previous quickstart](#).

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'll use storage account to verify that the queue and a message in it were created.

Find the storage account connected to AzureWebJobsStorage

1. Go to your function app and select Configuration.
2. Under Application settings, select AzureWebJobsStorage.

The screenshot shows the Azure portal's Configuration blade for a function app named 'functions-ggailey777-7'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events, Functions (selected), App keys, App files, Proxies, Deployment (selected), Deployment slots, Deployment Center, Settings (selected), Configuration (highlighted with a red box), Authentication / Authorizati..., Application Insights, Identity, and Backups. The main content area shows the 'Application settings' tab selected. A table lists application settings with columns for Name, Value, Source, and Deployment slot. The 'AzureWebJobsStorage' setting is highlighted with a red box. Other settings listed include APPINSIGHTS_INSTRUMENTATIONKEY, APPLICATIONINSIGHTS_CONNECTION_STRIN, FUNCTIONS_EXTENSION_VERSION, FUNCTIONS_WORKER_RUNTIME, storageaccountmyres81bd_STORAGE, storageaccountmyres8409_STORAGE, WEBSITE_CONTENTAZUREFILECONNECTION:, and WEBSITE_CONTENTSHARE.

| Name | Value | Source | Deployment slot |
|--------------------------------------|-------------------------------------|------------|-----------------|
| APPINSIGHTS_INSTRUMENTATIONKEY | (Hidden value. Click to show value) | App Config | |
| APPLICATIONINSIGHTS_CONNECTION_STRIN | (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 | |
| storageaccountmyres81bd_STORAGE | (Hidden value. Click to show value) | App Config | |
| storageaccountmyres8409_STORAGE | (Hidden value. Click to show value) | App Config | |
| WEBSITE_CONTENTAZUREFILECONNECTION: | (Hidden value. Click to show value) | App Config | |
| WEBSITE_CONTENTSHARE | (Hidden value. Click to show value) | App Config | |

3. Locate and make note of the account name.

The screenshot shows the 'Add/Edit application setting' dialog. The 'Name' field is filled with 'AzureWebJobsStorage'. The 'Value' field contains the value 'DefaultEndpointsProtocol=https;AccountName=storageaccountmyresbcb9;AccountKey=fLt9Wtr+z/30JESevURlqh4aiO8...'. A checkbox for 'Deployment slot setting' is present but unchecked. The 'Value' field is highlighted with a red box.

Examine the output queue

1. In the resource group for your function app, select the storage account that you're using for this quickstart.
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, and you'll see a new message appear 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 have created in this quickstart, do not 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 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 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 "Resource groups" section with "myResourceGroup" selected. The main content area shows details for "myResourceGroup", including its status as "Running", location as "Central US", and app service plan as "ASP-myResourceGroup-a285 (Y1: 0)". The URL listed is <https://myfunctionapp.azurewebsites.net>. The "Metrics" tab is selected at the bottom.

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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 added an output binding to an existing function. For more information about binding to Queue storage, see [Azure Functions Storage queue 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.

Connect Azure Functions to Azure Storage using Visual Studio Code

12/4/2020 • 16 minutes to read • [Edit Online](#)

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 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 start this article, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool 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](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- 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 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 the F1 key to open the command palette, then search for and run the command

- Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

IMPORTANT

Because it contains secrets, the local.settings.json file never gets published, and is excluded from source control.

- 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 usage is enabled in the host.json file at the root of the project, which appears as follows:

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage
```

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 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.json file. 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:

| 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. |

| PROMPT | VALUE | DESCRIPTION |
|---|---------------------|--|
| The name used to identify this binding in your code | msg | Name that identifies the binding parameter referenced in your code. |
| The queue to which the message will be sent | outqueue | 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" | AzureWebJobsStorage | 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:

```
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
```

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};
```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```

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
})

```

Update `HttpExample__init__.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```

import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )

```

The `msg` parameter is an instance of the [azure.functions.InputStream class](#). Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```
if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}
```

At this point, your function should look as follows:

```
[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}
```

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.

```
// Write the name to the message queue.
```

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:

```

@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.

```

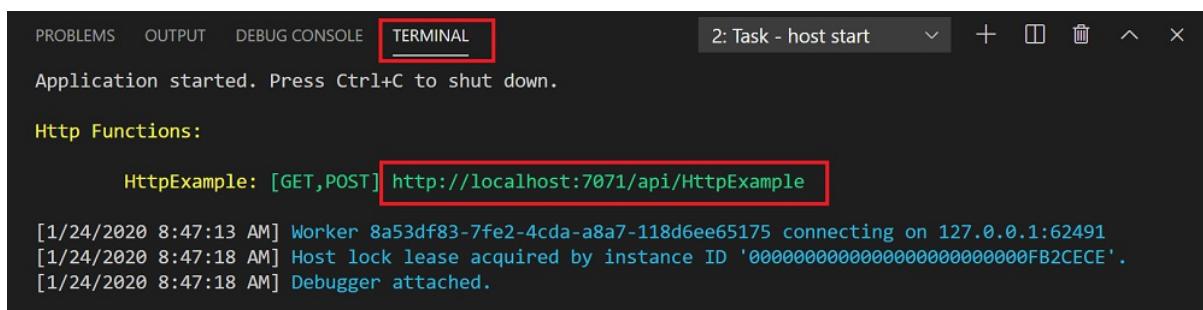
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);

```

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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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
2: Task - host start
Application started. Press Ctrl+C to shut down.

Http Functions:

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

[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '0000000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.

```

3. 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
```

4. A response is returned, which looks like the following in a browser:



5. 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 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] }
```

6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

Run the function locally

Azure Functions Core Tools integrates with Visual Studio Code to let you run and debug an Azure Functions project locally. For details on how to debug in Visual Studio Code, see [Debug PowerShell Azure Functions locally](#).

1. Press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The title bar says "TERMINAL". The terminal window displays the following text:

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

Http Functions:

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

[4/20/2019 6:19:06 AM] System Log: {
[4/20/2019 6:19:06 AM]   Log-Message: The enforced concurrency level (pool size limit) is '1'.
[4/20/2019 6:19:06 AM] }
[4/20/2019 6:19:06 AM] System Log: {
```

Below the terminal window, the status bar shows "unctions (MyFunctionProj)" and "Ln 10, Col 28 (27 selected)".

- Append the query string `?name=<yourname>` to this URL, and then use `Invoke-RestMethod` in a second PowerShell command prompt to execute the request, as follows:

```
PS > Invoke-RestMethod -Method Get -Uri http://localhost:7071/api/HttpTrigger?name=PowerShell
Hello PowerShell
```

You can also execute the GET request from a browser from the following URL:

```
http://localhost:7071/api/HttpExample?name=PowerShell
```

When you call the `HttpTrigger` endpoint without passing a `name` parameter either as a query parameter or in the body, the function returns a `BadRequest` error. When you review the code in `run.ps1`, you see that this error occurs by design.

- Information about the request is shown in `Terminal` panel.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The title bar says "TERMINAL". The terminal window displays the following detailed log output:

```
[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 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] }
```

- When done, press **Ctrl + C** to stop Core Tools.

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

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.

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.

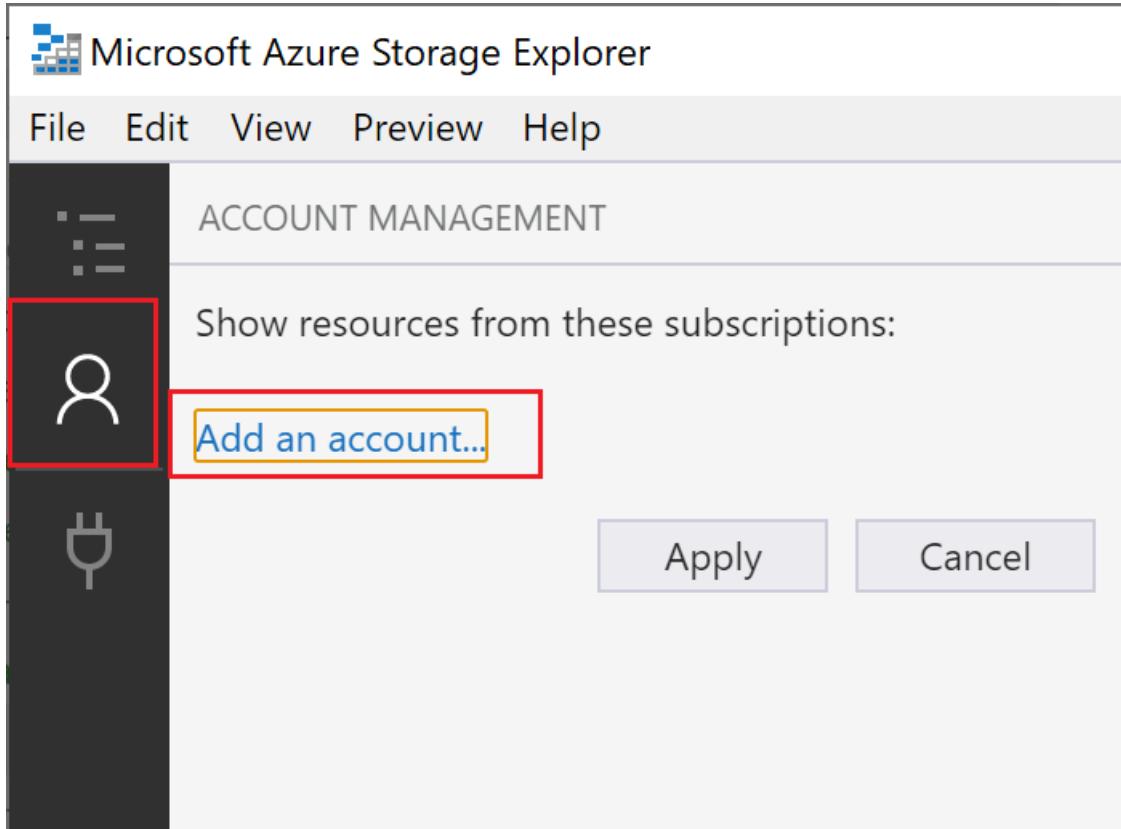
Browse to the location of your test code under `src/test/java`, open the `Function.java` project file, and replace the line of code under `//Invoke` with the following code.

```
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);
```

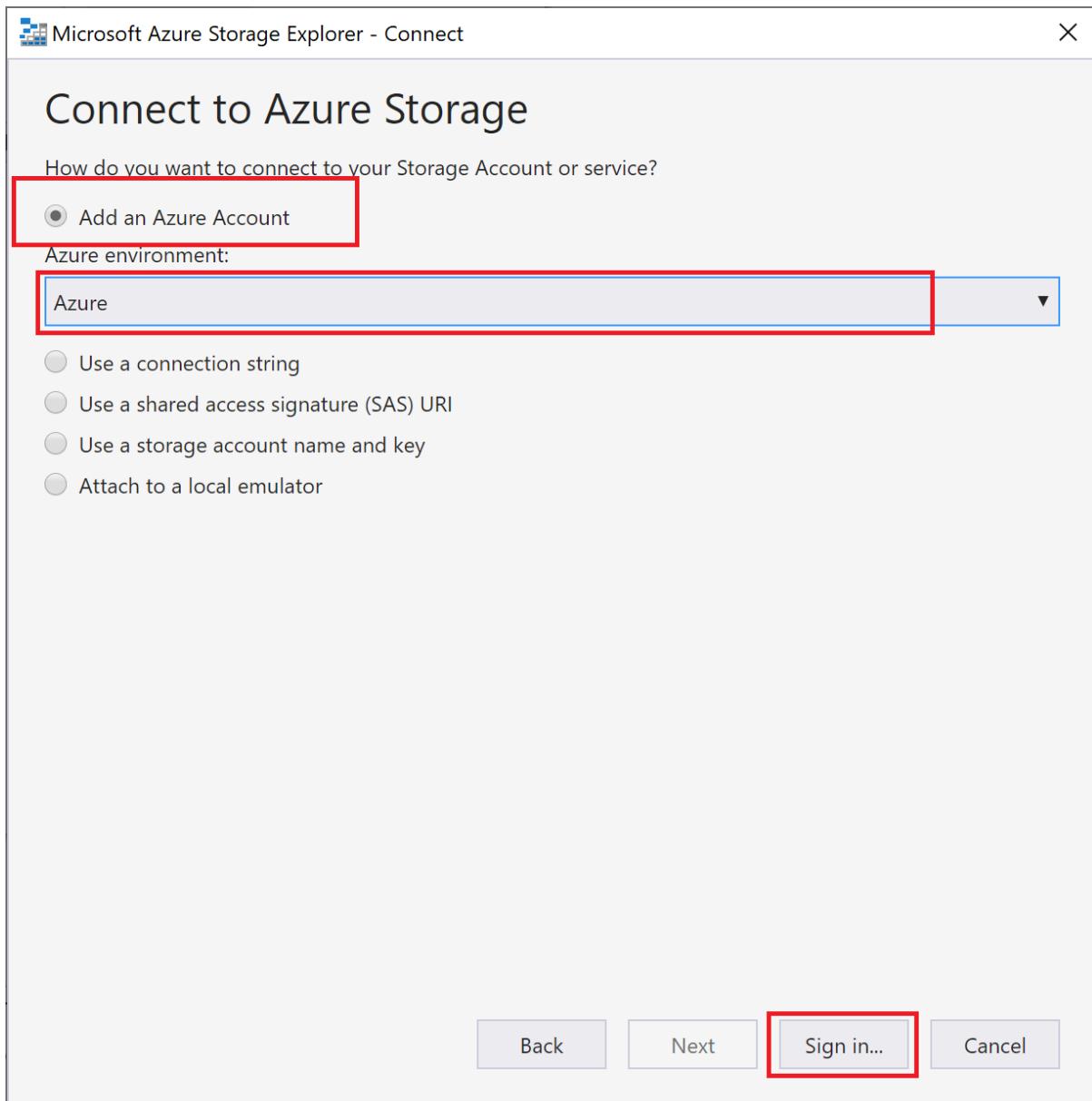
Connect Storage Explorer to your account

Skip this section if you have 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 select **Sign in....**

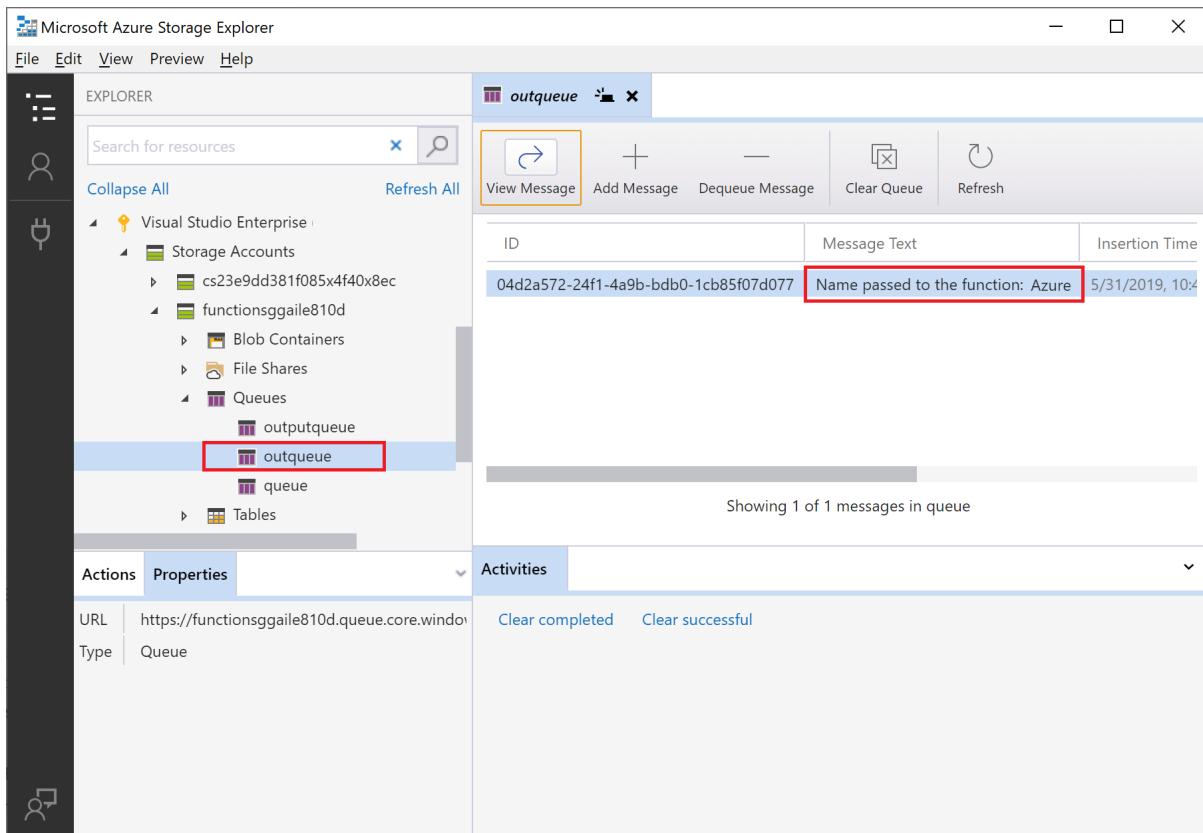


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account.

Examine the output queue

1. In Visual Studio Code, press the F1 key 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 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'll see a new message appear 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 deployment completes, you can again use cURL or a browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL, as in the following example:

```
curl https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....&name=<yourname>
```

- 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

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:

- In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Open in portal`.
- 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'. On the left, there's a sidebar with links like 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 area has a title bar with 'myfunctionapp' and 'App Service'. Below the title bar are buttons for Browse, Refresh, Stop, Restart, Swap, Get publish profile, Reset publish profile, and Delete. The 'Overview' tab is selected. A red box highlights the 'Resource group (change)' field, which contains 'myResourceGroup'. To the right of this field are details: 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), and Runtime version (3.0.13139.0). At the bottom of the main area are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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](#)
- [Examples of complete Function projects in JavaScript.](#)
- [Azure Functions JavaScript developer guide](#)
- [Examples of complete Function projects in Java.](#)
- [Azure Functions Java developer guide](#)
- [Examples of complete Function projects in TypeScript.](#)
- [Azure Functions TypeScript developer guide](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)
- [Examples of complete Function projects in PowerShell.](#)
- [Azure Functions PowerShell developer guide](#)

Connect functions to Azure Storage using Visual Studio

12/4/2020 • 6 minutes to read • [Edit Online](#)

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

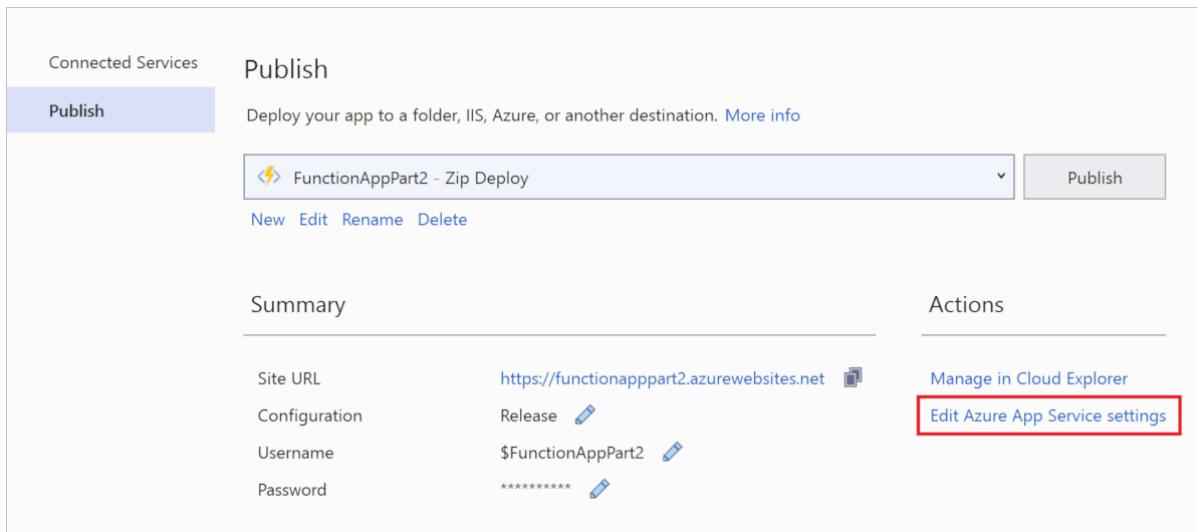
Before you start this article, you must:

- Complete [part 1 of the Visual Studio quickstart](#).
- 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. Under **Actions**, select **Edit 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:

```
Install-Package Microsoft.Azure.WebJobs.Extensions.Storage -Version 3.0.6
```

Now, you can add the storage 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. 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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The Run method definition should now look like the following:

```
[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)
```

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 `msg` output binding object to create a queue message. Add this code before the method returns.

```

if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}

```

At this point, your function should look as follows:

```

[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)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

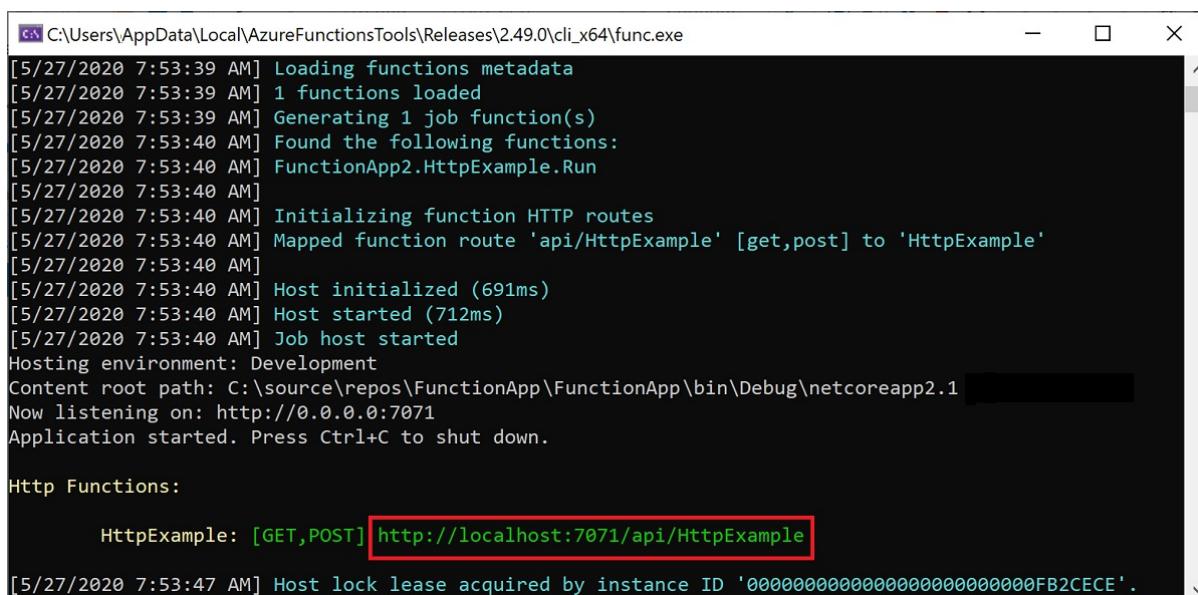
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

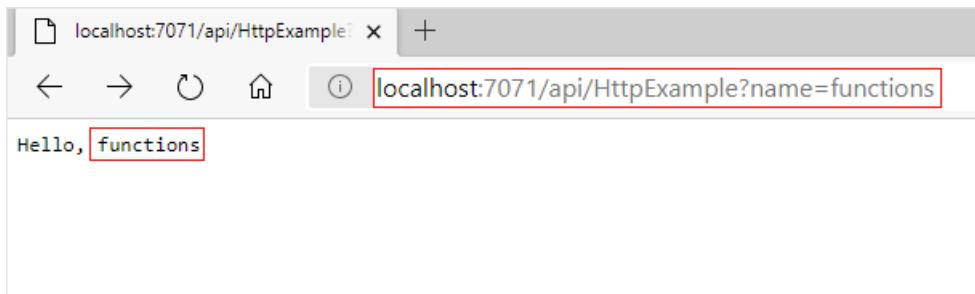
Http Functions:

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

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

```

3. Paste the URL for the HTTP request into your browser's address bar. Append the query string `?name=<YOUR_NAME>` to this URL 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.

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 Cloud Explorer to verify that the queue was created along with the new message.

Examine the output queue

1. In Visual Studio from the **View** menu, select **Cloud Explorer**.
2. In **Cloud Explorer**, expand your Azure subscription and **Storage Accounts**, then expand the storage account used by your function. If you can't remember the storage account name, check the `AzureWebJobsStorage` connection string setting in the `local.settings.json` file.
3. Expand the **Queues** node, and then double-click the queue named **outqueue** to view the contents of the queue in Visual Studio.

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*.

| ID | Message Text | Insertion Time |
|--------------------------------------|------------------------------------|---------------------|
| 04d2a572-24f1-4a9b-bdb0-1cb85f07d077 | Name passed to the function: Azure | 5/31/2019, 10:42:23 |

4. Run the function again, send another request, and you'll see a new message appear in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

1. In **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
2. After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourusername>` to the URL.
3. 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 have created in this quickstart, do not 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 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 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 "Resource group (change)" section with a red box around it, containing the text "myResourceGroup". The main content area displays various details about the function app, such as Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), App Service Plan (ASP-myResourceGroup-a285), 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 quickstart.

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 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. 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

12/4/2020 • 6 minutes to read • [Edit Online](#)

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:

```
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:

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[1.*, 2.0.0)"  
    }  
}
```

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:

```
@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:

```
@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.

```
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:

```

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.

```

@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](#)
- [Gradle](#)

```

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:

```
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:

```
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:

```
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:

```
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](#)
- [Gradle](#)

```
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:

```
curl -w "\n" https://fabrikam-functions-20190929094703749.azurewebsites.net/api/HttpTrigger-Java?  
code=zYRohsTwBlZ68YF.... --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:

```
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](#)

Strategies for testing your code in Azure Functions

12/4/2020 • 7 minutes to read • [Edit Online](#)

This article demonstrates how to create automated tests for Azure Functions.

Testing all code is recommended, however you may get the best results by wrapping up a Function's logic and creating tests outside the Function. Abstracting logic away limits a Function's lines of code and allows the Function to be solely responsible for calling other classes or modules. This article, however, demonstrates how to create automated tests against an HTTP and timer-triggered functions.

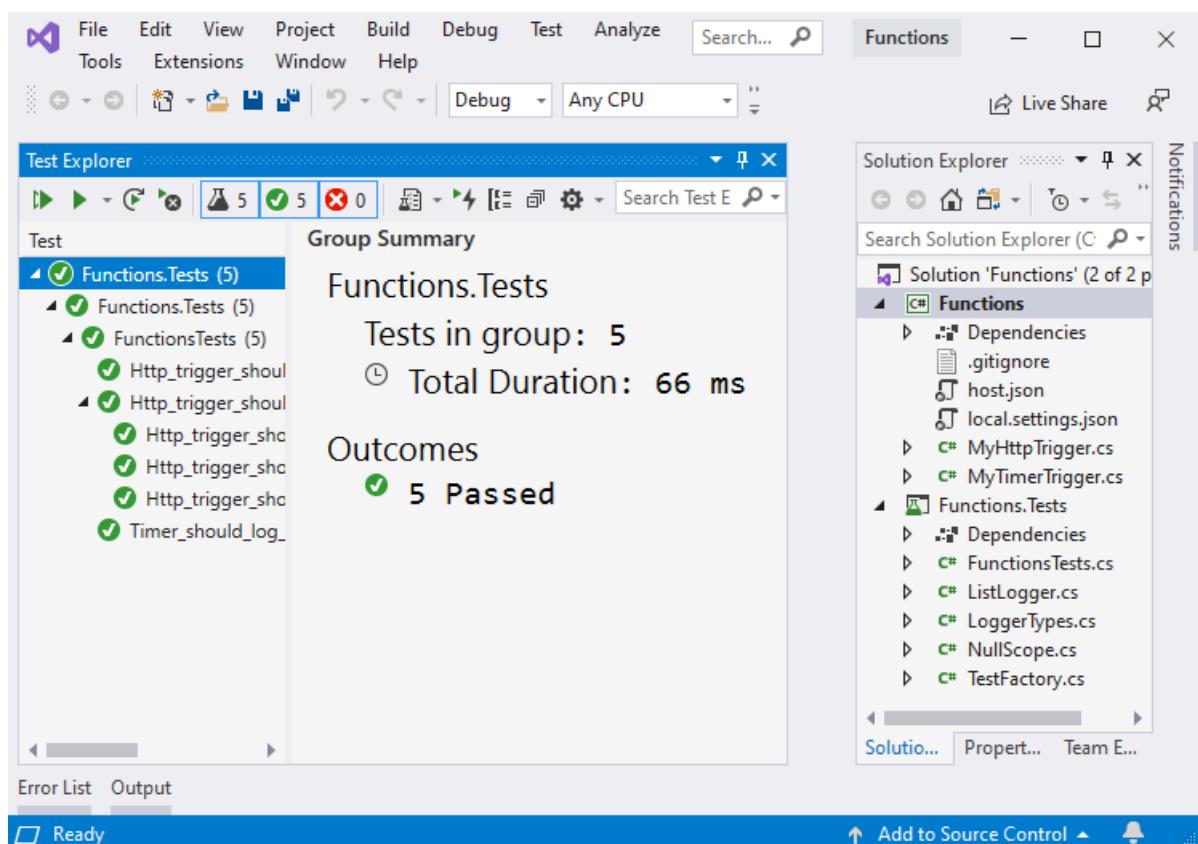
The content that follows is split into two different sections meant to target different languages and environments. You can learn to build tests in:

- [C# in Visual Studio with xUnit](#)
- [JavaScript in VS Code with Jest](#)

The sample repository is available on [GitHub](#).

C# in Visual Studio

The following example describes how to create a C# Function app in Visual Studio and run and tests with [xUnit](#).



Setup

To set up your environment, create a Function and test app. The following steps help you create the apps and functions required to support the tests:

1. [Create a new Functions app and name it Functions](#)
2. [Create an HTTP function from the template and name it MyHttpTrigger.](#)
3. [Create a timer function from the template and name it MyTimerTrigger.](#)

4. Create an xUnit Test app in the solution and name it `Functions.Tests`.
5. Use NuGet to add a reference from the test app to `Microsoft.AspNetCore.Mvc`
6. Reference the `Functions` app from `Functions.Tests` app.

Create test classes

Now that the projects are created, you can create the classes used to run the automated tests.

Each function takes an instance of `ILogger` to handle message logging. Some tests either don't log messages or have no concern for how logging is implemented. Other tests need to evaluate messages logged to determine whether a test is passing.

You'll create a new class named `ListLogger` which holds an internal list of messages to evaluate during a testing. To implement the required `ILogger` interface, the class needs a scope. The following class mocks a scope for the test cases to pass to the `ListLogger` class.

Create a new class in `Functions.Tests` project named `NullScope.cs` and enter the following code:

```
using System;

namespace Functions.Tests
{
    public class NullScope : IDisposable
    {
        public static NullScope Instance { get; } = new NullScope();

        private NullScope() { }

        public void Dispose() { }
    }
}
```

Next, create a new class in `Functions.Tests` project named `ListLogger.cs` and enter the following code:

```

using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Text;

namespace Functions.Tests
{
    public class ListLogger : ILogger
    {
        public IList<string> Logs;

        public IDisposable BeginScope<TState>(TState state) => NullScope.Instance;

        public bool IsEnabled(LogLevel logLevel) => false;

        public ListLogger()
        {
            this.Logs = new List<string>();
        }

        public void Log<TState>(LogLevel logLevel,
                               EventId eventId,
                               TState state,
                               Exception exception,
                               Func<TState, Exception, string> formatter)
        {
            string message = formatter(state, exception);
            this.Logs.Add(message);
        }
    }
}

```

The `ListLogger` class implements the following members as contracted by the `ILogger` interface:

- **BeginScope**: Scopes add context to your logging. In this case, the test just points to the static instance on the `NullScope` class to allow the test to function.
- **IsEnabled**: A default value of `false` is provided.
- **Log**: This method uses the provided `formatter` function to format the message and then adds the resulting text to the `Logs` collection.

The `Logs` collection is an instance of `List<string>` and is initialized in the constructor.

Next, create a new file in *Functions.Tests* project named `LoggerTypes.cs` and enter the following code:

```

namespace Functions.Tests
{
    public enum LoggerTypes
    {
        Null,
        List
    }
}

```

This enumeration specifies the type of logger used by the tests.

Now create a new class in *Functions.Tests* project named `TestFactory.cs` and enter the following code:

```

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Primitives;
using System.Collections.Generic;

namespace Functions.Tests
{
    public class TestFactory
    {
        public static IEnumerable<object[]> Data()
        {
            return new List<object[]>
            {
                new object[] { "name", "Bill" },
                new object[] { "name", "Paul" },
                new object[] { "name", "Steve" }

            };
        }

        private static Dictionary<string, StringValues> CreateDictionary(string key, string value)
        {
            var qs = new Dictionary<string, StringValues>
            {
                { key, value }
            };
            return qs;
        }

        public static HttpRequest CreateHttpRequest(string queryStringKey, string queryStringValue)
        {
            var context = new DefaultHttpContext();
            var request = context.Request;
            request.Query = new QueryCollection(CreateDictionary(queryStringKey, queryStringValue));
            return request;
        }

        public static ILogger CreateLogger(LoggerTypes type = LoggerTypes.Null)
        {
            ILogger logger;

            if (type == LoggerTypes.List)
            {
                logger = new ListLogger();
            }
            else
            {
                logger = NullLoggerFactory.Instance.CreateLogger("Null Logger");
            }

            return logger;
        }
    }
}

```

The `TestFactory` class implements the following members:

- **Data:** This property returns an `IEnumerable` collection of sample data. The key value pairs represent values that are passed into a query string.
- **CreateDictionary:** This method accepts a key/value pair as arguments and returns a new `Dictionary` used to create `QueryCollection` to represent query string values.

- **CreateHttpRequest**: This method creates an HTTP request initialized with the given query string parameters.
- **CreateLogger**: Based on the logger type, this method returns a logger class used for testing. The `ListLogger` keeps track of logged messages available for evaluation in tests.

Finally, create a new class in `Functions.Tests` project named `FunctionsTests.cs` and enter the following code:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Xunit;

namespace Functions.Tests
{
    public class FunctionsTests
    {
        private readonly ILogger logger = TestFactory.CreateLogger();

        [Fact]
        public async void Http_trigger_should_return_known_string()
        {
            var request = TestFactory.CreateHttpRequest("name", "Bill");
            var response = (OkObjectResult)await MyHttpTrigger.Run(request, logger);
            Assert.Equal("Hello, Bill. This HTTP triggered function executed successfully.", response.Value);
        }

        [Theory]
        [MemberData(nameof(TestFactory.Data), MemberType = typeof(TestFactory))]
        public async void Http_trigger_should_return_string_from_member_data(string queryStringKey,
            string queryStringValue)
        {
            var request = TestFactory.CreateHttpRequest(queryStringKey, queryStringValue);
            var response = (OkObjectResult)await MyHttpTrigger.Run(request, logger);
            Assert.Equal($"Hello, {queryStringValue}. This HTTP triggered function executed successfully.",
                response.Value);
        }

        [Fact]
        public void Timer_should_log_message()
        {
            var logger = (ListLogger)TestFactory.CreateLogger(LoggerTypes.List);
            MyTimerTrigger.Run(null, logger);
            var msg = logger.Logs[0];
            Assert.Contains("C# Timer trigger function executed at", msg);
        }
    }
}
```

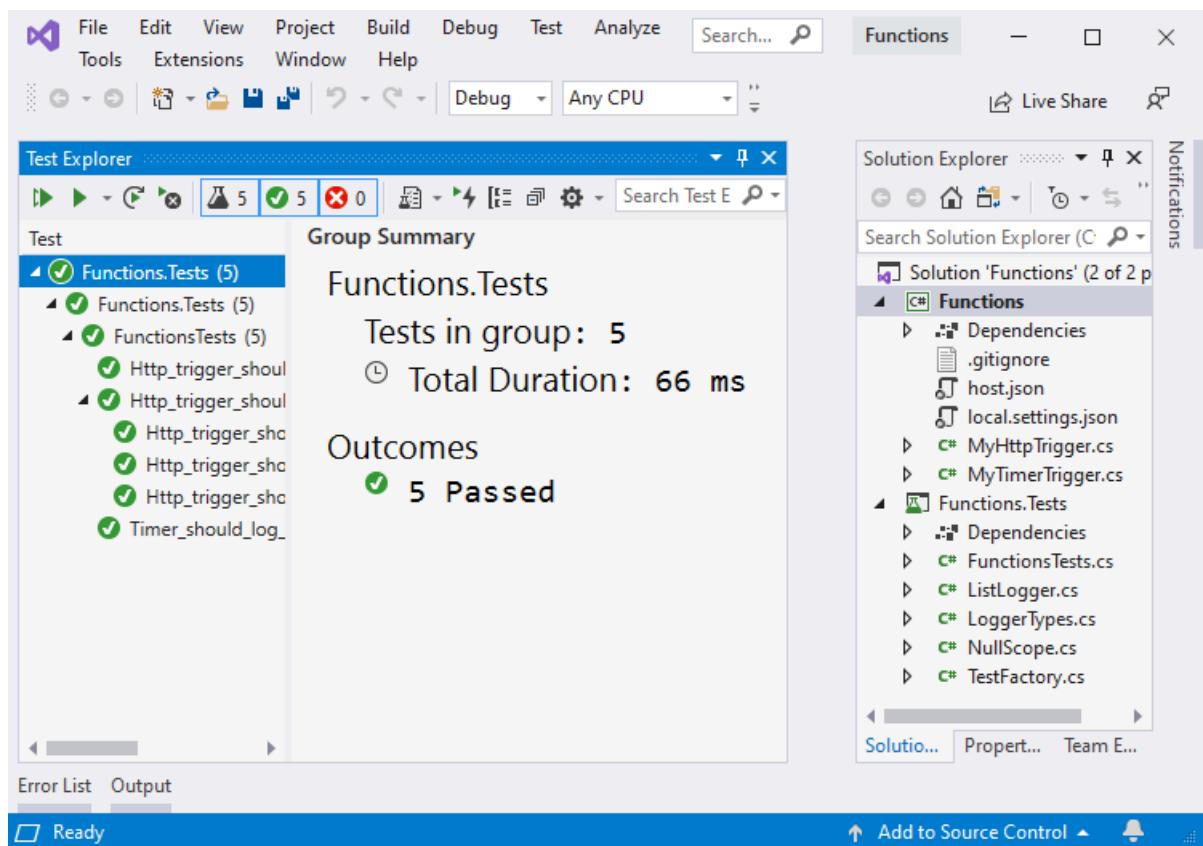
The members implemented in this class are:

- **Http_trigger_should_return_known_string**: This test creates a request with the query string values of `name=Bill` to an HTTP function and checks that the expected response is returned.
- **Http_trigger_should_return_string_from_member_data**: This test uses xUnit attributes to provide sample data to the HTTP function.
- **Timer_should_log_message**: This test creates an instance of `ListLogger` and passes it to a timer functions. Once the function is run, then the log is checked to ensure the expected message is present.

If you want to access application settings in your tests, you can [inject](#) an `IConfiguration` instance with mocked environment variable values into your function.

Run tests

To run the tests, navigate to the **Test Explorer** and click **Run all**.

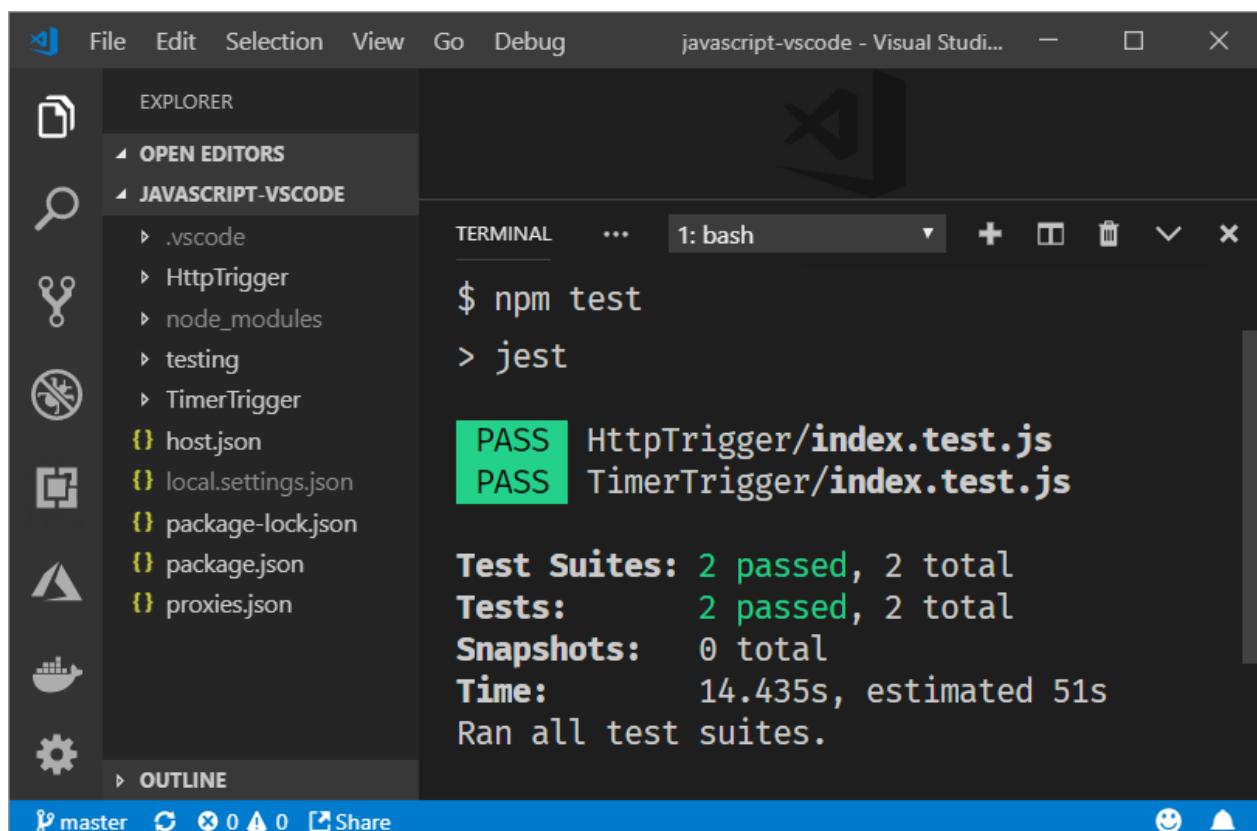


Debug tests

To debug the tests, set a breakpoint on a test, navigate to the **Test Explorer** and click **Run > Debug Last Run**.

JavaScript in VS Code

The following example describes how to create a JavaScript Function app in VS Code and run and tests with [Jest](#). This procedure uses the [VS Code Functions extension](#) to create Azure Functions.



Setup

To set up your environment, initialize a new Node.js app in an empty folder by running `npm init`.

```
npm init -y
```

Next, install Jest by running the following command:

```
npm i jest
```

Now update `package.json` to replace the existing test command with the following command:

```
"scripts": {  
  "test": "jest"  
}
```

Create test modules

With the project initialized, you can create the modules used to run the automated tests. Begin by creating a new folder named `testing` to hold the support modules.

In the `testing` folder add a new file, name it `defaultContext.js`, and add the following code:

```
module.exports = {  
  log: jest.fn()  
};
```

This module mocks the `log` function to represent the default execution context.

Next, add a new file, name it `defaultTimer.js`, and add the following code:

```
module.exports = {  
  IsPastDue: false  
};
```

This module implements the `IsPastDue` property to stand in as a fake timer instance. Timer configurations like NCrontab expressions are not required here as the test harness is simply calling the function directly to test the outcome.

Next, use the VS Code Functions extension to [create a new JavaScript HTTP Function](#) and name it `HttpTrigger`. Once the function is created, add a new file in the same folder named `index.test.js`, and add the following code:

```
const httpFunction = require('./index');  
const context = require('../testing/defaultContext')  
  
test('Http trigger should return known text', async () => {  
  
  const request = {  
    query: { name: 'Bill' }  
  };  
  
  await httpFunction(context, request);  
  
  expect(context.log.mock.calls.length).toBe(1);  
  expect(context.res.body).toEqual('Hello Bill');  
});
```

The HTTP function from the template returns a string of "Hello" concatenated with the name provided in the query string. This test creates a fake instance of a request and passes it to the HTTP function. The test checks that the `log` method is called once and the returned text equals "Hello Bill".

Next, use the VS Code Functions extension to create a new JavaScript Timer Function and name it `TimerTrigger`. Once the function is created, add a new file in the same folder named `index.test.js`, and add the following code:

```
const timerFunction = require('./index');
const context = require('../testing/defaultContext');
const timer = require('../testing/defaultTimer');

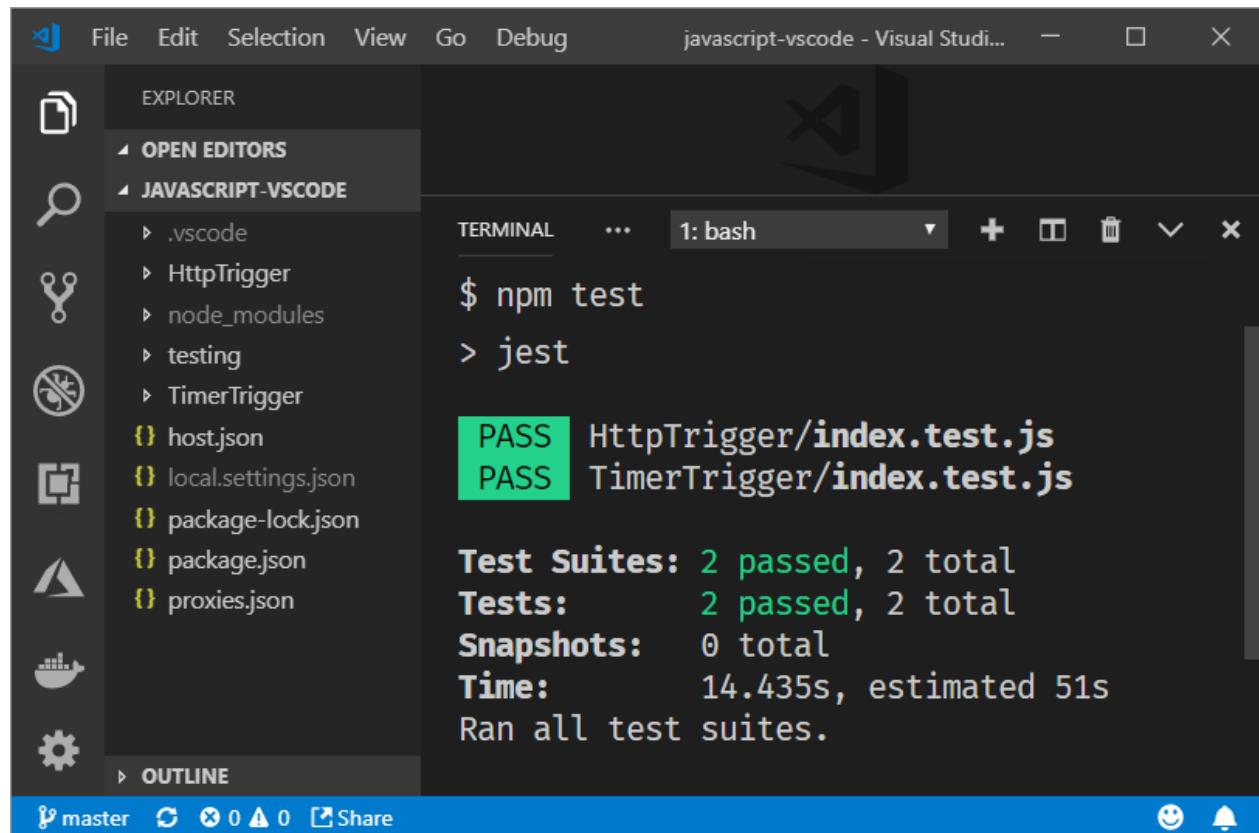
test('Timer trigger should log message', () => {
    timerFunction(context, timer);
    expect(context.log.mock.calls.length).toBe(1);
});
```

The timer function from the template logs a message at the end of the body of the function. This test ensures the `log` function is called once.

Run tests

To run the tests, press **CTRL + ~** to open the command window, and run `npm test`:

```
npm test
```



Debug tests

To debug your tests, add the following configuration to your `launch.json` file:

```
{  
  "type": "node",  
  "request": "launch",  
  "name": "Jest Tests",  
  "disableOptimisticBPs": true,  
  "program": "${workspaceRoot}/node_modules/jest/bin/jest.js",  
  "args": [  
    "-i"  
  ],  
  "internalConsoleOptions": "openOnSessionStart"  
}
```

Next, set a breakpoint in your test and press F5.

Next steps

Now that you've learned how to write automated tests for your functions, continue with these resources:

- [Manually run a non HTTP-triggered function](#)
- [Azure Functions error handling](#)
- [Azure Function Event Grid Trigger Local Debugging](#)

Debug PowerShell Azure Functions locally

12/4/2020 • 7 minutes to read • [Edit Online](#)

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:

```
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:

```
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.

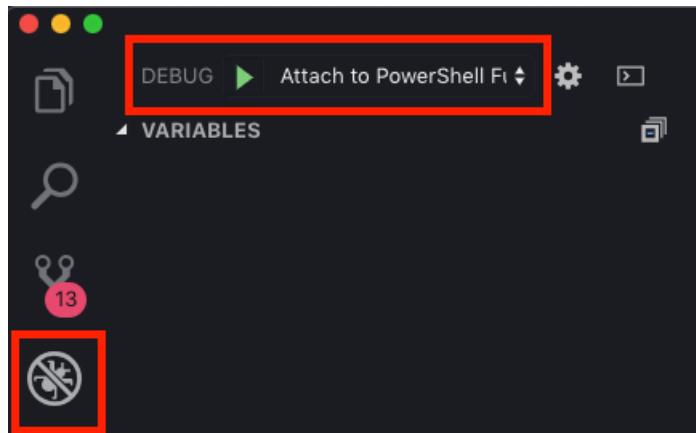
1. Press F1 to display the command pallet, then search for `Session`.
2. Choose **PowerShell: Show Session Menu**.
3. If your **Current session** isn't **PowerShell Core 6**, choose **Switch to: PowerShell Core 6**.

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:

```
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:

```
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:

```
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:

```
Get-PsHostProcessInfo
```

This cmdlet returns a table that looks like the following 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:

```
# 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](#).

Azure Function Event Grid Trigger Local Debugging

12/4/2020 • 2 minutes to read • [Edit Online](#)

This article demonstrates how to debug a local function that handles an Azure Event Grid event raised by a storage account.

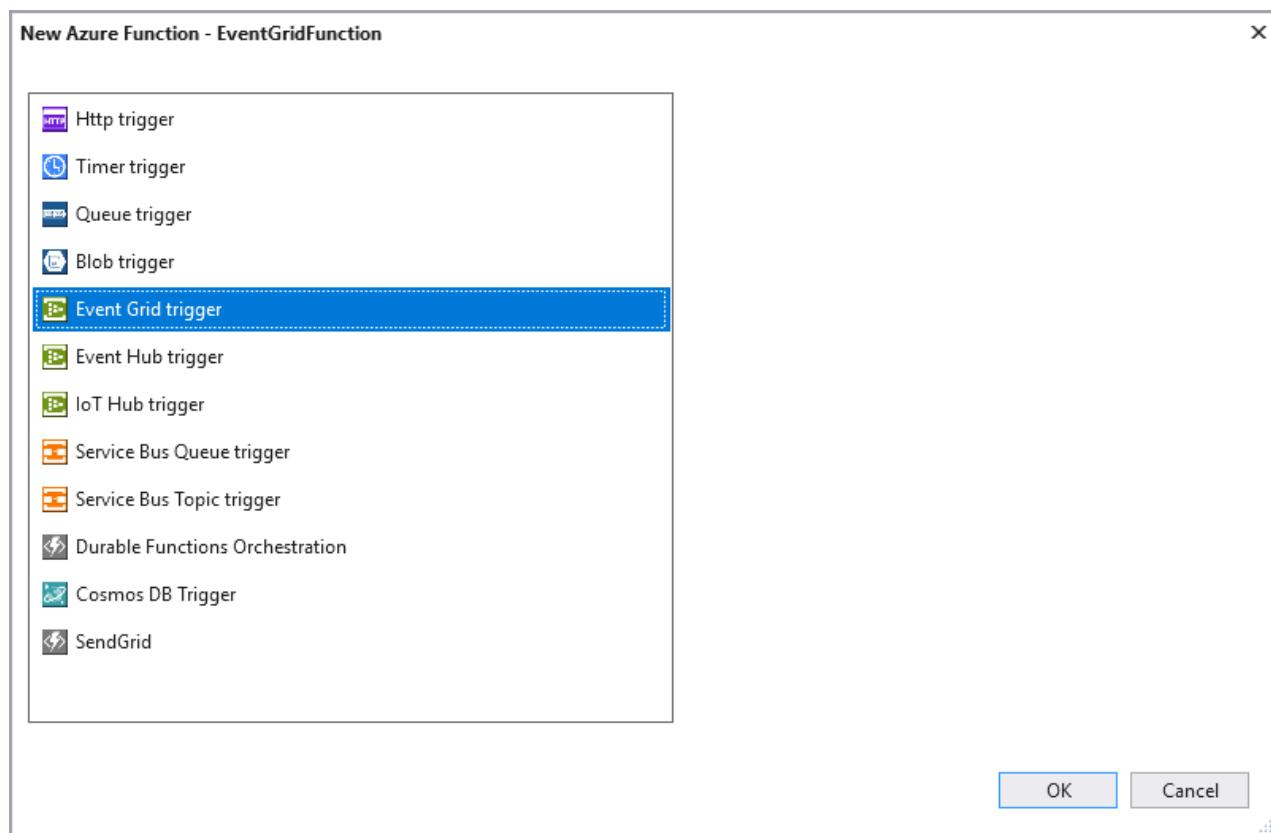
Prerequisites

- Create or use an existing function app
- Create or use an existing storage account
- Download [ngrok](#) to allow Azure to call your local function

Create a new function

Open your function app in Visual Studio and, right-click on the project name in the Solution Explorer and click **Add** > **New Azure Function**.

In the *New Azure Function* window, select **Event Grid trigger** and click **OK**.



Once the function is created, open the code file and copy the URL commented out at the top of the file. This location is used when configuring the Event Grid trigger.

```

// Default URI for triggering event grid function in the local environment.
// http://localhost:7071/runtime/webhooks/EventGrid?functionName={functionname}

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Azure.EventGrid.Models;
using Microsoft.Azure.WebJobs.Extensions.EventGrid;
using Microsoft.Extensions.Logging;

namespace LocalDebugDemo
{
    public static class EventGridFunction
    {
        [FunctionName("EventGridFunction")]
        public static void Run([EventGridTrigger]EventGridEvent eventGridEvent,
        ILogger log)
        {
            log.LogInformation(eventGridEvent.Data.ToString());
        }
    }
}

```

Then, set a breakpoint on the line that begins with `log.LogInformation`.

```

9
10  namespace LocalDebugDemo
11  {
12      public static class EventGridFunction
13      {
14          [FunctionName("EventGridFunction")]
15          public static void Run([EventGridTrigger]EventGridEvent eventGridEvent, ILogger log)
16          {
17              ● log.LogInformation(eventGridEvent.Data.ToString());
18          }
19      }
20  }
21

```

Next, press F5 to start a debugging session.

Allow Azure to call your local function

To break into a function being debugged on your machine, you must enable a way for Azure to communicate with your local function from the cloud.

The `ngrok` utility provides a way for Azure to call the function running on your machine. Start `ngrok` using the following command:

```
ngrok http -host-header=localhost 7071
```

As the utility is set up, the command window should look similar to the following screenshot:

```
ɔ Select Command Prompt - ngrok http -host-header=localhost 7071
ngrok by @inconshreveable                                     (Ctrl+C to quit)

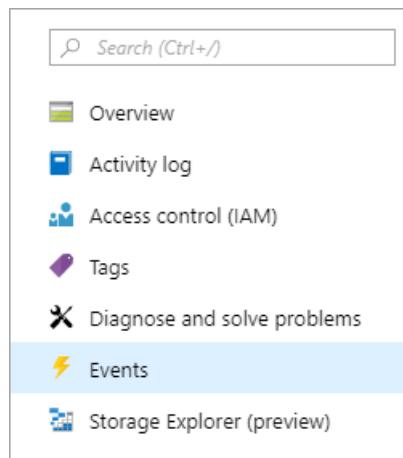
Session Status          online
Account                  (Plan: Free)
Version                 2.2.8
Region                  United States (us)
Web Interface           http://127.0.0.1:4040
Forwarding              http://9e14d0df.ngrok.io -> localhost:7071
Forwarding              https://9e14d0df.ngrok.io -> localhost:7071

Connections             ttl     opn      rt1      rt5      p50      p90
                        0       0       0.00    0.00    0.00    0.00
```

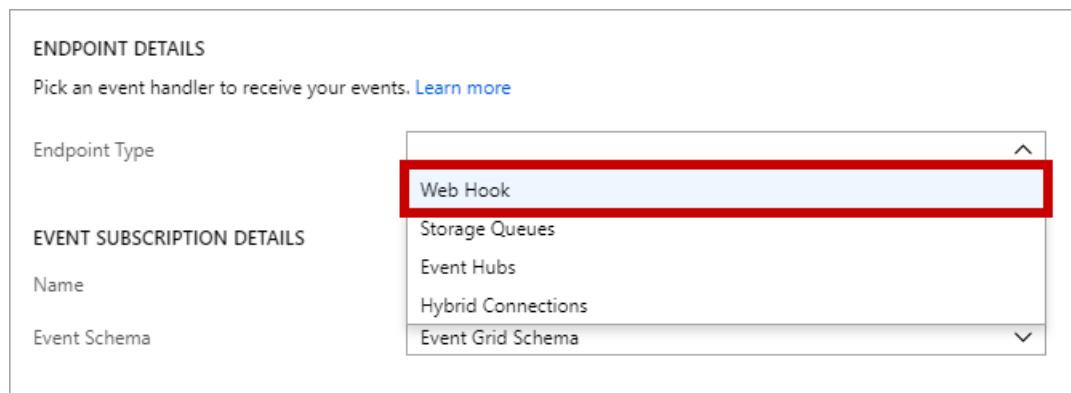
Copy the HTTPS URL generated when *ngrok* is run. This value is used when configuring the event grid event endpoint.

Add a storage event

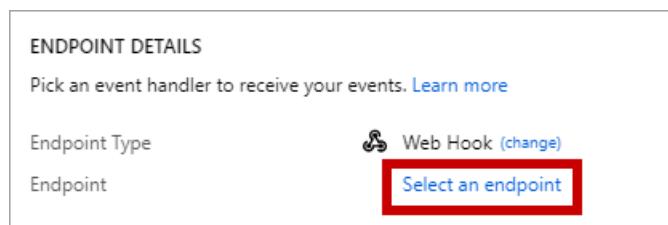
Open the Azure portal and navigate to a storage account and click on the **Events** option.



In the *Events* window, click on the **Event Subscription** button. In the *Event Subscription* window, click on the **Endpoint Type** dropdown and select **Web Hook**.

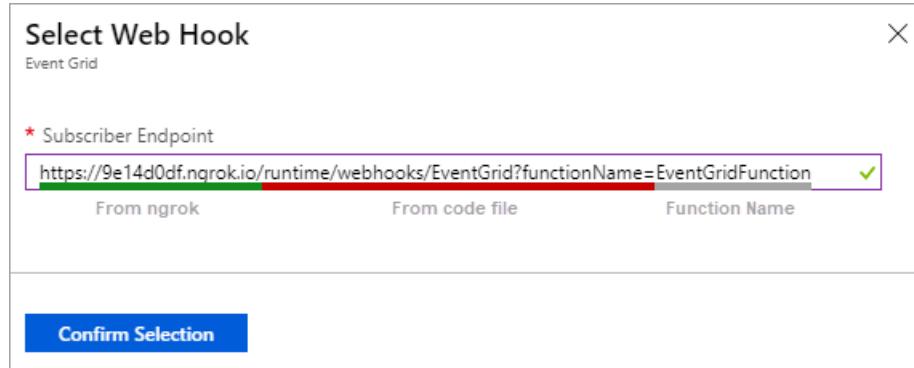


Once the endpoint type is configured, click on **Select an endpoint** to configure the endpoint value.



The *Subscriber Endpoint* value is made up from three different values. The prefix is the HTTPS URL generated by *ngrok*. The remainder of the URL comes from the URL found in the function code file, with the function name added at the end. Starting with the URL from the function code file, the *ngrok* URL replaces `http://localhost:7071` and the function name replaces `{functionname}`.

The following screenshot shows how the final URL should look:



Once you've entered the appropriate value, click **Confirm Selection**.

IMPORTANT

Every time you start *ngrok*, the HTTPS URL is regenerated and the value changes. Therefore you must create a new Event Subscription each time you expose your function to Azure via *ngrok*.

Upload a file

Now you can upload a file to your storage account to trigger an Event Grid event for your local function to handle.

Open [Storage Explorer](#) and connect to the your storage account.

- Expand **Blob Containers**
- Right-click and select **Create Blob Container**.
- Name the container **test**
- Select the *test* container
- Click the **Upload** button
- Click **Upload Files**
- Select a file and upload it to the blob container

Debug the function

Once the Event Grid recognizes a new file is uploaded to the storage container, the break point is hit in your local function.

```
public static class EventGridFunction
{
    [FunctionName("EventGridFunction")]
    public static void Run([EventGridTrigger]EventGridEvent eventGridEvent, ILogger log)
    {
        log.LogInformation(eventGridEvent.Data.ToString());
    }
}
```

Clean up resources

To clean up the resources created in this article, delete the **test** container in your storage account.

Next steps

- [Automate resizing uploaded images using Event Grid](#)
- [Event Grid trigger for Azure Functions](#)

Use dependency injection in .NET Azure Functions

11/2/2020 • 6 minutes to read • [Edit Online](#)

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.

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

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.

To register the method, add the `FunctionsStartup` assembly attribute that specifies the type name used during startup.

```
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 additional 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 setup in the `Configure` method. As a result, Functions-specific types like `BindingContext` and `ExecutionContext` aren't available during setup or as injectable types.

Use injected dependencies

Constructor injection is used to make your dependencies available in a function. The use of constructor injection requires that you do not 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.

```
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(HttpClient httpClient, IMyService service)
        {
            this._client = httpClient;
            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 request of the service.
- **Scoped:** The scoped service lifetime matches a function execution lifetime. Scoped services are created once

per 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 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 additional filters and categories.

The following example demonstrates how to add an `ILogger<HttpTrigger>` with logs that are exposed to the host.

```
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.

```
{
    "version": "2.0",
    "logging": {
        "applicationInsights": {
            "samplingSettings": {
                "isEnabled": true,
                "excludedTypes": "Request"
            }
        },
        "logLevel": {
            "MyNamespace.HttpTrigger": "Information"
        }
    }
}
```

Function app provided services

The function host registers many services. The following services are safe to take as a dependency in your application:

| SERVICE TYPE | LIFETIME | DESCRIPTION |
|---|------------------------|---|
| <code>Microsoft.Extensions.Configuration.IConfiguration</code> | <code>Singleton</code> | Runtime configuration |
| <code>Microsoft.Azure.WebJobs.Host.Executors.IJobHostEnvironment</code> | <code>Singleton</code> | 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.

Consider the following class that includes a property named consistent with an app setting:

```
public class MyOptions
{
    public string MyCustomSetting { get; set; }
}
```

And a `local.settings.json` file that might structure the custom setting as follows:

```
{  
    "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:

```
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.

```
using System;  
using Microsoft.Extensions.Options;  
  
public class HttpTrigger  
{  
    private readonly MyOptions _settings;  
  
    public HttpTrigger(IOptions<MyOptions> options)  
    {  
        _settings = options.Value;  
    }  
}
```

Refer to [Options pattern in ASP.NET Core](#) for more details regarding working with options.

Customizing configuration sources

NOTE

Configuration source customization is available beginning in Azure Functions host versions 2.0.14192.0 and 3.0.14191.0.

To specify additional configuration sources, override the `ConfigureAppConfiguration` method in your function app's `StartUp` class.

The following sample adds configuration values from a base and an optional environment-specific app settings files.

```

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();
        }
    }
}

```

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` are not automatically copied to the function app's output folder. Update your `.csproj` file to match the following sample to ensure the files are copied.

```

<None Update="appsettings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</None>
<None Update="appsettings.Development.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <CopyToPublishDirectory>Never</CopyToPublishDirectory>
</None>

```

IMPORTANT

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.

Next steps

For more information, see the following resources:

- [How to monitor your function app](#)
- [Best practices for functions](#)

Manage connections in Azure Functions

11/2/2020 • 5 minutes to read • [Edit Online](#)

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, 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.

Connection limit

The number of available connections is limited partly because a function app 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.

HttpClient example (C#)

Here's an example of C# function code that creates a static [HttpClient](#) instance:

```
// 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.

HTTP agent examples (JavaScript)

Because it provides better connection management options, you should use the native [http.agent](#) class instead of non-native methods, such as the [node-fetch](#) module. Connection parameters are configured through options on the [http.agent](#) class. For detailed options available with the HTTP agent, see [new Agent\(\[options\]\)](#).

The global [http.globalAgent](#) class used by [http.request\(\)](#) has all of these values set to their respective defaults. The recommended way to configure connection limits in Functions is to set a maximum number globally. The following example sets the maximum number of sockets for the function app:

```
http.globalAgent.maxSockets = 200;
```

The following example creates a new HTTP request with a custom HTTP agent only for that request:

```
var http = require('http');
var httpAgent = new http.Agent();
httpAgent.maxSockets = 200;
options.agent = httpAgent;
http.request(options, onResponseCallback);
```

DocumentClient code example (C#)

[DocumentClient](#) 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:

```

#r "Microsoft.Azure.Documents.Client"
using Microsoft.Azure.Documents.Client;

private static Lazy<DocumentClient> lazyClient = new Lazy<DocumentClient>(InitializeDocumentClient);
private static DocumentClient documentClient => lazyClient.Value;

private static DocumentClient InitializeDocumentClient()
{
    // Perform any initialization here
    var uri = new Uri("example");
    var authKey = "authKey";

    return new DocumentClient(uri, authKey);
}

public static async Task Run(string input)
{
    Uri collectionUri = UriFactory.CreateDocumentCollectionUri("database", "collection");
    object document = new { Data = "example" };
    await documentClient.UpsertDocumentAsync(collectionUri, document);

    // Rest of function
}

```

If you are working with functions v3.x, you need a reference to Microsoft.Azure.DocumentDB.Core. Add a reference in the code:

```
#r "Microsoft.Azure.DocumentDB.Core"
```

Also, create a file named "function.proj" for your trigger and add the below content :

```

<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>netcoreapp3.0</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="Microsoft.Azure.DocumentDB.Core" Version="2.12.0" />
    </ItemGroup>
</Project>

```

CosmosClient code example (JavaScript)

[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:

```
const cosmos = require('@azure/cosmos');
const endpoint = process.env.COSMOS_API_URL;
const key = process.env.COSMOS_API_KEY;
const { CosmosClient } = cosmos;

const client = new CosmosClient({ endpoint, key });
// All function invocations also reference the same database and container.
const container = client.database("MyDatabaseName").container("MyContainerName");

module.exports = async function (context) {
    const { resources: itemArray } = await container.items.readAll().fetchAll();
    context.log(itemArray);
}
```

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

12/4/2020 • 7 minutes to read • [Edit Online](#)

Handling errors in Azure Functions is important to avoid lost data, missed events, and to monitor the health of your application.

This article describes general strategies for error handling along with links to binding-specific errors.

Handling errors

Errors raised in an Azure Functions can come from any of the following origins:

- Use of built-in Azure Functions [triggers and bindings](#)
- Calls to APIs of underlying Azure services
- Calls to REST endpoints
- Calls to client libraries, packages, or third-party APIs

Following good error handling practices is important to avoid loss of data or missed messages. Recommended error handling practices include the following actions:

- [Enable Application Insights](#)
- [Use structured error handling](#)
- [Design for idempotency](#)
- [Implement retry policies](#) (where appropriate)

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 try/catch block. In the catch block, you can capture and log errors.

Retry policies (preview)

A retry policy can be defined on any function for any trigger type in your function app. The retry policy re-executes a function until either successful execution or until the maximum number of retries occur. Retry policies can be defined for all functions in an app or for individual functions. By default, a function app won't retry messages (aside from the [specific triggers that have a retry policy on the trigger source](#)). A retry policy is evaluated whenever an execution results in an uncaught exception. As a best practice, you should catch all exceptions in your code and rethrow any errors that should result in a retry. Event Hubs and Azure Cosmos DB checkpoints won't be written until the retry policy for the execution has completed, meaning progressing on that partition is paused until the current batch has completed.

Retry policy options

The following options are available for defining a retry policy.

Max Retry Count is the maximum number of times an execution is retried before eventual failure. A value of `-1` means to retry indefinitely. The current retry count is stored in memory of the instance. It's possible that an instance has a failure between retry attempts. When an instance fails during a retry policy, the retry count is lost. When there are instance failures, triggers like Event Hubs, Azure Cosmos DB, and Queue storage are able to resume processing and retry the batch on a new instance, with the retry count reset to zero. Other triggers, like HTTP and timer, don't resume on a new instance. This means that the max retry count is a best effort, and in some rare cases an execution could be retried more than the maximum, or for triggers like HTTP and timer be retried less than the maximum.

Retry Strategy controls how retries behave. The following are two supported retry options:

| OPTION | DESCRIPTION |
|---------------------------------|---|
| <code>fixedDelay</code> | A specified amount of time is allowed to elapse between each retry, |
| <code>exponentialBackoff</code> | The first retry waits for the minimum delay. On subsequent retries, time is added exponentially to the initial duration for each retry, until the maximum delay is reached. Exponential back-off adds some small randomization to delays to stagger retries in high-throughput scenarios. |

App level configuration

A retry policy can be defined for all functions in an app [using the `host.json` file](#).

Function level configuration

A retry policy can be defined for a specific function. Function-specific configuration takes priority over app-level configuration.

Fixed delay retry

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)
- [PowerShell](#)

Retries require NuGet package [Microsoft.Azure.WebJobs >= 3.0.23](#)

```
[FunctionName("EventHubTrigger")]
[FixedDelayRetry(5, "00:00:10")]
public static async Task Run([EventHubTrigger("myHub", Connection = "EventHubConnection")] EventData[]
events, ILogger log)
{
// ...
}
```

Exponential backoff retry

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)
- [PowerShell](#)

Retries require NuGet package [Microsoft.Azure.WebJobs >= 3.0.23](#)

```
[FunctionName("EventHubTrigger")]
[ExponentialBackoffRetry(5, "00:00:04", "00:15:00")]
public static async Task Run([EventHubTrigger("myHub", Connection = "EventHubConnection")] EventData[]
events, ILogger log)
{
// ...
}
```

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| strategy | n/a | Required. The retry strategy to use. Valid values are <code>fixedDelay</code> or <code>exponentialBackoff</code> . |
| maxRetryCount | n/a | Required. The maximum number of retries allowed per function execution. <code>-1</code> means to retry indefinitely. |
| delayInterval | n/a | The delay that will be used between retries when using <code>fixedDelay</code> strategy. |
| minimumInterval | n/a | The minimum retry delay when using <code>exponentialBackoff</code> strategy. |
| maximumInterval | n/a | The maximum retry delay when using <code>exponentialBackoff</code> strategy. |

Retry limitations during preview

- For .NET projects, you may need to manually pull in a version of [Microsoft.Azure.WebJobs](#) >= 3.0.23.
- In the consumption plan, the app may be scaled down to zero while retrying the final messages in a queue.
- In the consumption plan, the app may be scaled down while performing retries. For best results, choose a retry interval <= 00:01:00 and <= 5 retries.

Using retry support on top of trigger resilience

The function app retry policy is independent of any retries or resiliency that the trigger provides. The function retry policy will only layer on top of a trigger resilient retry. For example, if using Azure Service Bus, by default queues have a message delivery count of 10. The default delivery count means after 10 attempted deliveries of a queue message, Service Bus will dead-letter the message. You can define a retry policy for a function that has a Service Bus trigger, but the retries will layer on top of the Service Bus delivery attempts.

For instance, if you used the default Service Bus delivery count of 10, and defined a function retry policy of 5. The message would first dequeue, incrementing the service bus delivery account to 1. If every execution failed, after five attempts to trigger the same message, that message would be marked as abandoned. Service Bus would immediately requeue the message, it would trigger the function and increment the delivery count to 2. Finally, after 50 eventual attempts (10 service bus deliveries * five function retries per delivery), the message would be abandoned and trigger a dead-letter on service bus.

WARNING

It is not recommended to set the delivery count for a trigger like Service Bus Queues to 1, meaning the message would be dead-lettered immediately after a single function retry cycle. This is because triggers provide resiliency with retries, while the function retry policy is best effort and may result in less than the desired total number of retries.

Triggers with additional resiliency or retries

The following triggers support retries at the trigger source:

- [Azure Blob storage](#)
- [Azure Queue storage](#)
- [Azure Service Bus \(queue/topic\)](#)

By default, most triggers retry requests up to five times. After the fifth retry, both the Azure Queue storage will write a message to a [poison queue](#). The default Service Bus queue and topic policy will write a message to a [dead-letter queue](#) after 10 attempts.

Binding error codes

When integrating with Azure services, errors may originate from the APIs of the underlying services. Information relating to binding-specific errors is available in the **Exceptions and return codes** section of the following articles:

- [Azure Cosmos DB](#)
- [Blob storage](#)
- [Event Hubs](#)
- [IoT Hubs](#)
- [Notification Hubs](#)
- [Queue storage](#)
- [Service Bus](#)
- [Table storage](#)

Manually run a non HTTP-triggered function

11/2/2020 • 2 minutes to read • [Edit Online](#)

This article demonstrates how to manually run a non HTTP-triggered function via specially formatted HTTP request.

In some contexts, you may 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 [another resource's action](#).

[Postman](#) is used in the following example, but you may use [cURL](#), [Fiddler](#) or any other like tool to send HTTP requests.

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.
- **Folder path:** To access non HTTP-triggered functions via an HTTP request, you have to send the request through the folders `admin/functions`.
- **Function name:** The name of the function you want to run.

You use this request location in Postman along with the function's master key in the request to Azure to run the function.

NOTE

When running locally, the function's master key is not required. You can directly [call the function](#) omitting the `x-functions-key` header.

Get the function's master key

1. Navigate to your function app in the [Azure portal](#), select **App Keys**, and then the `_master` key.

myFunctionApp-dma | App keys

Function App

Search (Ctrl+ /) Refresh

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Security
- Events (preview)

Functions

- Functions
- App keys**
- App files
- Proxies

Host keys (all functions)

Use Host keys with your clients to access all your HTTP functions in the app. _master key grants admin access to Functions Runtime APIs.

+ New host key Show values

Filter host keys

| Name | Value | Renew |
|---------|-----------------------------------|-----------------------|
| _master | Hidden value. Click to show value | Renew |
| default | Hidden value. Click to show value | Renew |

System keys

System keys are automatically managed by the Function runtime. System Keys provide granular

2. In the **Edit key** section, copy the key value to your clipboard, and then select **OK**.

Edit key

Renew and save key value

Name: [Copy to clipboard](#)

Value:

[OK](#) [Cancel](#)

3. After copying the `_master` key, select **Code + Test**, and then select **Logs**. You'll see messages from the function logged here when you manually run the function from Postman.

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 shows the code editor with the file 'functions-ggailey777-7 \ QueueTrigger1 \ run.csx'. The code is:

```
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 red box around the 'Logs' button.

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.

Call the function

Open Postman and follow these steps:

1. Enter the **request location** in the URL text box.
2. Ensure the HTTP method is set to **POST**.
3. Select the **Headers** tab.
4. Type **x-functions-key** as the first key and paste the master key (from the clipboard) as the value.
5. Type **Content-Type** as the second key and type **application/json** as the value.

The screenshot shows the Postman application window. At the top, the URL is set to `https://myfunctionsdemos.azurewebsites.net/admin/functions/QueueTrigger`. The 'Headers' tab is selected, showing two entries: `x-functions-key` and `Content-Type`, both of which are checked. Below the headers, there is a note: 'Hit the Send button to get a response.' and a large blue 'Send' button.

6. Select the **Body** tab.

7. Type `{ "input": "test" }` as the body for the request.

The screenshot shows the Postman application window with the 'Body' tab selected. The body content is set to `{ "input": "test" }`. The 'raw' option is selected under the body type dropdown. The rest of the interface is identical to the previous screenshot, including the 'Headers' tab with the same two entries.

8. Select **Send**.

The screenshot shows the Postman application interface. At the top, there's a toolbar with 'File', 'Edit', 'View', and 'Help' options, and buttons for 'New', 'Import', 'Runner', and a plus sign. Below the toolbar, the URL 'https://myfunctionsdemos.azurewebsites.net/admin/functions/QueueTrigger' is entered. The method dropdown shows 'POST'. To the right of the URL, there's a 'Send' button highlighted with a red box, and a 'Save' button. Above the URL, there are buttons for '+', '...', 'No Environment', and settings. The main area has tabs for 'Params', 'Authorization', 'Headers (2)', 'Body', 'Pre-request Script', 'Tests', 'Cookies', and 'Code'. Under 'Headers (2)', there are two entries: 'x-functions-key' with value '/0TXT/nXdlUKKwdTyabatbTYVBQzoQj6C3K...' and 'Content-Type' with value 'application/json'. Below the headers, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. The 'Test Results' tab is active, showing 'Status: 202 Accepted', 'Time: 237 ms', and 'Size: 188 B'. A red box highlights the 'Status: 202 Accepted' text. At the bottom of the interface, there are icons for file operations and help.

Postman then reports a status of 202 Accepted.

9. Next, return to your function in the Azure portal. Review the logs and you'll see messages coming from the manual call to the function.

Home > functions-ggailey777-7 | Functions > QueueTrigger1 | Code + Test

QueueTrigger1 | Code + Test

Function

Search (Ctrl+ /) <>

Save Discard Refresh Test

functions-ggailey777-7 \ QueueTrigger1 \ run.csx

Overview Developer Code + Test Integration Monitor Function Keys

```
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
```

Logs Log Level Stop Copy Clear Maximize

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)

Next steps

- [Strategies for testing your code in Azure Functions](#)
- [Azure Function Event Grid Trigger Local Debugging](#)

Continuous deployment for Azure Functions

5/12/2020 • 2 minutes to read • [Edit Online](#)

You can use Azure Functions to deploy your code continuously by using [source control integration](#). Source control integration enables a workflow in which a code update triggers deployment to Azure. If you're new to Azure Functions, get started by reviewing the [Azure Functions overview](#).

Continuous deployment is a good option for projects where you integrate multiple and frequent contributions. When you use continuous deployment, you maintain a single source of truth for your code, which allows teams to easily collaborate. You can configure continuous deployment in Azure Functions from the following source code locations:

- [Azure Repos](#)
- [GitHub](#)
- [Bitbucket](#)

The unit of deployment for functions in Azure is the function app. All functions in a function app are deployed at the same time. After you enable continuous deployment, access to function code in the Azure portal is configured as *read-only* because the source of truth is set to be elsewhere.

Requirements for continuous deployment

For continuous deployment to succeed, your directory structure must be compatible with the basic folder structure that Azure Functions expects.

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file and one or more subfolders. Each subfolder contains the code for a separate function. The folder structure is shown in the following representation:

```
FunctionApp
| - host.json
| - MyFirstFunction
| | - function.json
| | - ...
| - MySecondFunction
| | - function.json
| | - ...
| - SharedCode
| - bin
```

In version 2.x and higher of the Functions runtime, all functions in the function app must share the same language stack.

The [host.json](#) file contains runtime-specific configurations and is in the root folder of the function app. A *bin* folder contains packages and other library files that the function app requires. See the language-specific requirements for a function app project:

- [C# class library \(.csproj\)](#)
- [C# script \(.csx\)](#)
- [F# script](#)
- [Java](#)
- [JavaScript](#)

- Python

NOTE

Continuous deployment is not yet supported for Linux apps running on a Consumption plan.

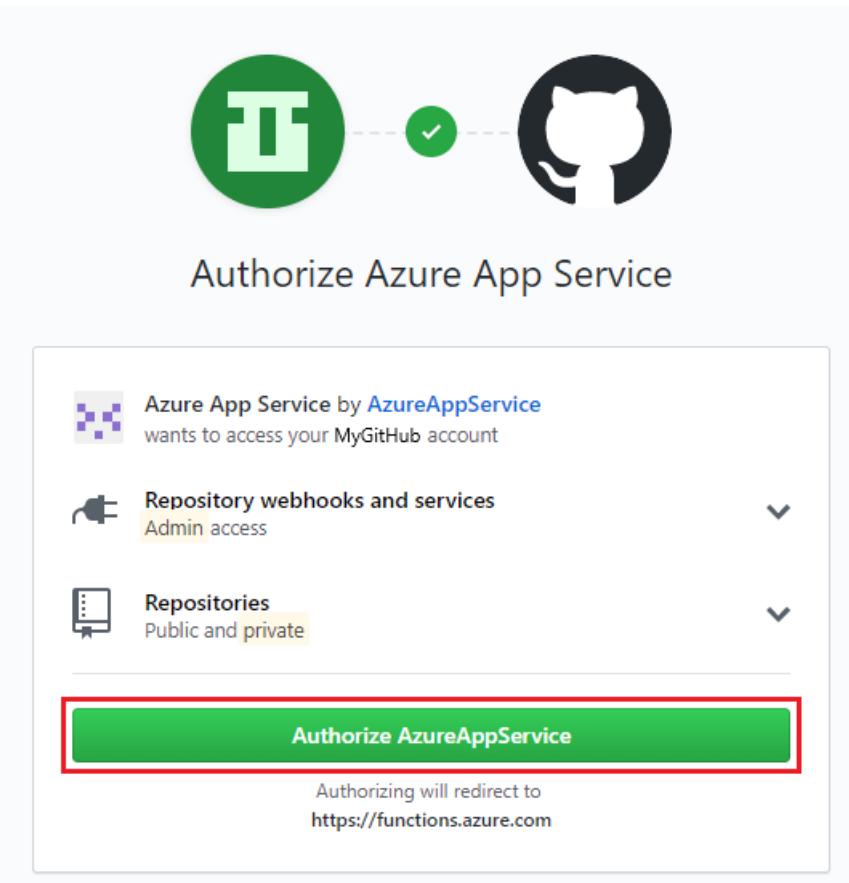
Set up continuous deployment

To configure continuous deployment for an existing function app, complete these steps. The steps demonstrate integration with a GitHub repository, but similar steps apply for Azure Repos or other source code repositories.

1. In your function app in the [Azure portal](#), select **Deployment Center**, select **GitHub**, and then select **Authorize**. If you've already authorized GitHub, select **Continue** and skip the next step.

The screenshot shows the Azure Function App Deployment Center interface. On the left, there's a navigation sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (Functions, App keys, App files), Deployment (Deployment slots, Deployment Center - highlighted with a red box), Settings (Configuration, Authentication / Authorization, Application Insights, Identity, Backups). The main area is titled "Deployment Center" with a gear icon. It says "App Service Deployment Center enables you to choose the location of your code as well as options for build and deployment to the cloud. Learn more". Below this is a flow diagram with four numbered circles: 1 SOURCE CONTROL, 2 BUILD PROVIDER, 3 CONFIGURE, 4 SUMMARY. A section titled "Continuous Deployment (CI / CD)" contains four cards: "Azure Repos" (Configure continuous integration with an Azure Repo, part of Azure DevOps Services (formerly known as VSTS)), "GitHub" (Configure continuous integration with a GitHub repo, with a note "Not Authorized" and an "Authorize" button highlighted with a red box), "Bitbucket", and "Local Git".

2. In GitHub, select **Authorize AzureAppService**.



Enter your GitHub password and then select **Continue**.

3. Select one of the following build providers:

- **App Service build service**: Best when you don't need a build or if you need a generic build.
- **Azure Pipelines (Preview)**: Best when you need more control over the build. This provider currently is in preview.

Select **Continue**.

4. Configure information specific to the source control option you specified. For GitHub, you must enter or select values for **Organization**, **Repository**, and **Branch**. The values are based on the location of your code. Then, select **Continue**.



Code

| | |
|--------------|-----------------------------------|
| Organization | MyGitHub |
| Repository | functions-typescript-intermediate |
| Branch | master |



If you can't find an organization or repository, you might need to enable additional permissions on GitHub.



[Back](#) [Continue](#)

5. Review all details, and then select **Finish** to complete your deployment configuration.

When the process is finished, 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.

NOTE

After you configure continuous integration, you can no longer edit your source files in the Functions portal.

Next steps

[Best practices for Azure Functions](#)

Azure Functions deployment slots

12/4/2020 • 7 minutes to read • [Edit Online](#)

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 under the Apps Service plan may have multiple slots, while under the Consumption plan only one slot is allowed.

The following reflect how functions are affected by swapping slots:

- Traffic redirection is seamless; no requests are dropped because of a swap.
- If a function is running during a swap, execution continues and the next triggers are routed to the swapped app instance.

NOTE

Slots are currently not available for the Linux Consumption plan.

Why use slots?

There are a number of advantages to using deployment slots. The following scenarios describe common uses for slots:

- **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.

Swap operations

During a swap, one slot is considered the source and the other 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's previously warmed up in the source slot.

4. **Repeat operation:** Now that the source slot has the pre-swap 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
- Non-public certificates and TLS/SSL settings
- Scale settings
- WebJobs schedulers
- IP restrictions
- Always On
- Diagnostic settings
- Cross-origin resource sharing (CORS)

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
- WebJobs content
- Hybrid connections *
- Virtual network integration *
- Service endpoints *
- Azure Content Delivery Network *

Features marked with an asterisk (*) are planned to be unswapped.

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 left sidebar has a tree view with 'slot-dma (myfunctionapp-dma/slot-dma)' selected under 'App Service (Slot)'. Under 'Deployment', 'Deployment slots' is selected and highlighted with a red box. 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 is a table showing two slots:

| NAME | STATUS | APP SERVICE PLAN |
|--|---------|-----------------------------|
| myfunctionapp-dma PRODUCTION | 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_CONTENTAZUREFILECONNECTION: | 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

- Once setting section disappears, select **Save** to keep the changes

Home > myFunctionApp-dma > myFunctionApp-dma | Deployment slots > slot-dma (myfunctionapp-dma/slot-dma)

slot-dma (myfunctionapp-dma/slot-dma) 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 | ✓ |

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

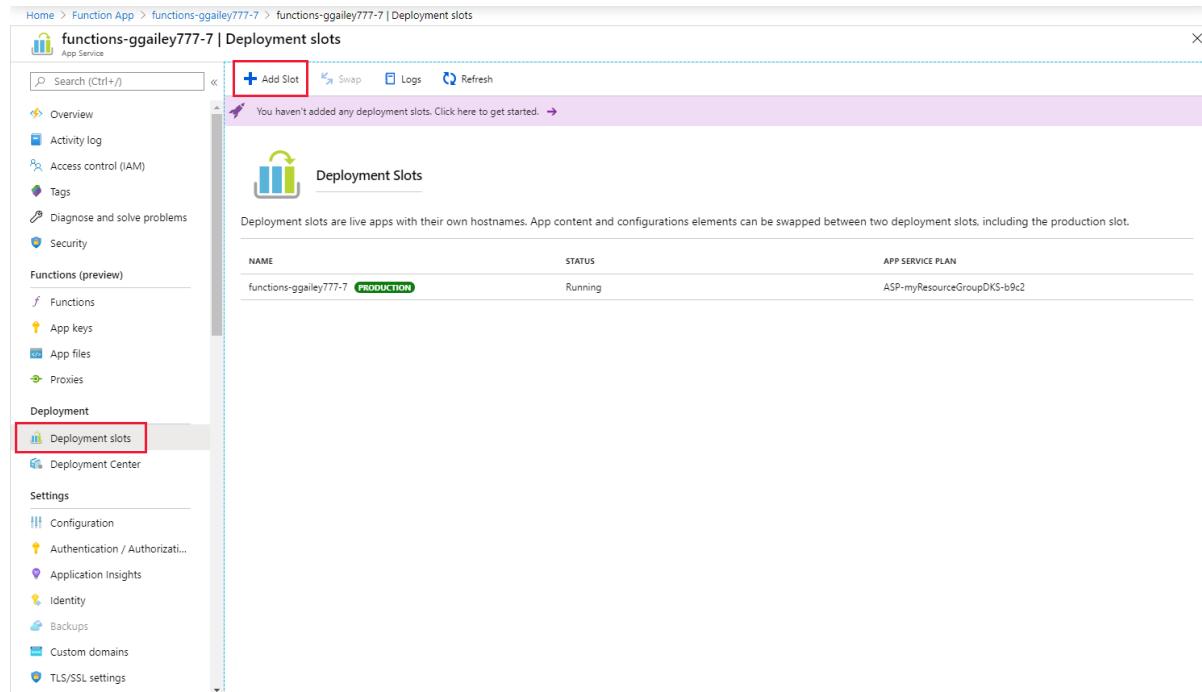
All slots scale to the same number of workers as the production slot.

- 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.

Add a slot

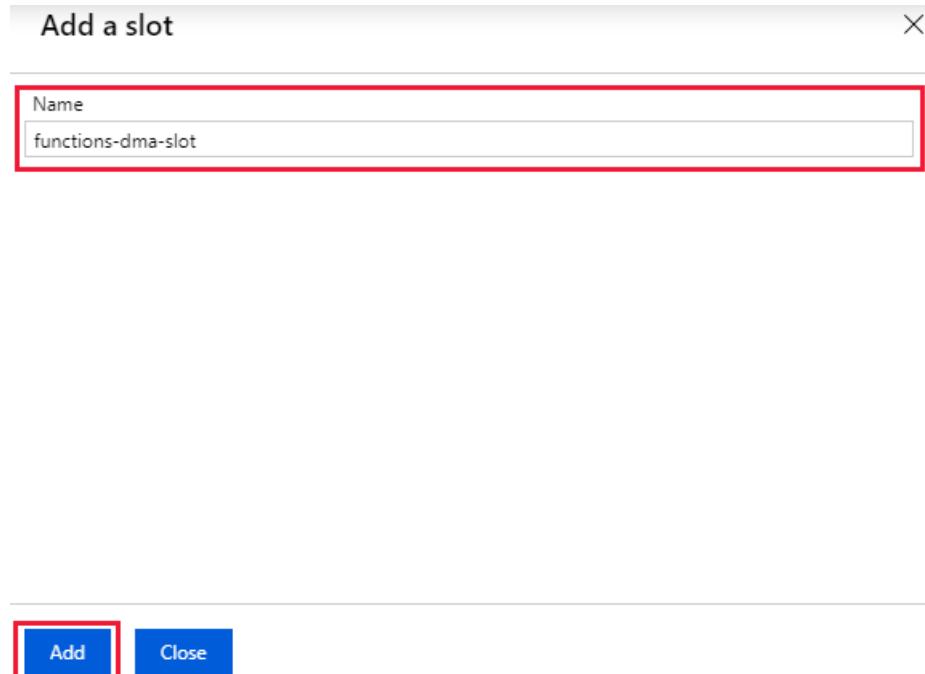
You can add a slot via the [CLI](#) or through the portal. The following steps demonstrate how to create a new slot in the portal:

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 is open, showing options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Functions (preview), Deployment slots (which is selected and highlighted with a red box), Deployment Center, Settings, Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, and TLS/SSL settings. At the top right, there are buttons for '+ Add Slot', 'Swap', 'Logs', and 'Refresh'. A message at the top says 'You haven't added any deployment slots. Click here to get started.' Below the message, there's a section titled 'Deployment Slots' with a table showing one slot: 'NAME: functions-ggailey777-7 STATUS: PRODUCTION APP SERVICE PLAN: ASP-myResourceGroupDKS-b9c2'. The '+ Add Slot' button is highlighted with a red box.

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' which is highlighted with a red box. At the bottom, there are two buttons: 'Add' (highlighted with a red box) and 'Close'.

Swap slots

You can swap slots via the [CLI](#) or through the portal. The following steps demonstrate how to swap slots in the portal:

1. Navigate to the function app.
2. Select **Deployment slots**, and then select **Swap**.

Home > myFunctionApp-dma > myFunctionApp-dma | Deployment slots

myFunctionApp-dma | Deployment slots

Deployment Slots

You have reached the slots quota limit (2) for the current plan.

| NAME | STATUS | APP SERVICE PLAN |
|----------------------------|------------|-----------------------------|
| myfunctionapp-dma | PRODUCTION | ASP-myResourceGroupdma-82be |
| myfunctionapp-dma-slot-dma | Running | ASP-myResourceGroupdma-82be |

Navigation:

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Security
- Functions (preview)**
 - Functions
 - App keys
 - App files
 - Proxies
- Deployment**
 - Deployment slots**
 - Deployment Center
- Settings**
 - Configuration
 - Authentication / Authorizati...
 - Application Insights
 - Identity
 - Backups
 - Custom domains
 - TLS/SSL settings
 - Networking

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 |

The operation may take a moment while the swap operation is executing.

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 pre-swapped state, do another swap to reverse the swap.

Remove a slot

You can remove a slot via the [CLI](#) or through the portal. The following steps demonstrate how to remove a slot in the portal:

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. On the left, a sidebar lists various settings like Functions, App files, and Deployment. The 'Deployment slots' option is selected and highlighted with a red box. The main content area is titled 'slot-dma (myfunctionapp-dma/slot-dma) | Deployment slots'. It displays a message about reaching the slots quota limit. Below this is a section titled 'Deployment Slots' with a brief description. A table lists the deployment slots:

| NAME | STATUS | APP SERVICE PLAN |
|----------------------------|------------|-----------------------------|
| myfunctionapp-dma | PRODUCTION | ASP-myResourceGroupdma-82be |
| myfunctionapp-dma-slot-dma | Running | ASP-myResourceGroupdma-82be |

2. Select **Delete**.

Home > Function App > myFunctionApp-dma > myFunctionApp-dma | Deployment slots > slot-dma (myfunctionapp-dma/slot-dma)

slot-dma (myfunctionapp/dma) App Service (Slot)

Search (Ctrl+ /) < 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

URL
<https://myfunctionapp-slot.azurewebsites.net>

Operating System
Windows

App Service Plan
ASP-myResourceGroupdma-82be (Y1: 0)

Properties
See More

Runtime version
3.0.13139.0

Tags (change)
Click here to add tags

Metrics Features (9) Notifications (0) Quickstart

Memory working set

| |
|------|
| 100B |
| 90B |
| 80B |
| 70B |
| 60B |
| 50B |

Function Execution Count

| |
|-----|
| 100 |
| 90 |
| 80 |
| 70 |
| 60 |
| 50 |

Functions (preview)

Functions

App keys

App files

Proxies

Deployment

Deployment slots

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" X

 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 delete 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](#)

Automate slot management

Using the [Azure CLI](#), you can automate the following actions for a slot:

- [create](#)
- [delete](#)
- [list](#)
- [swap](#)
- [auto-swap](#)

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.

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 has a navigation tree with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (preview), Deployment (with Deployment slots highlighted), Deployment Center, Settings, Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, and TLS/SSL settings. The main content area is titled 'slot-dma (myfunctionapp-dma/slot-dma) | Deployment slots'. It includes a search bar, 'Add Slot', 'Swap', 'Logs', and 'Refresh' buttons. A message bar at the top says 'You have reached the slots quota limit (2) for the current plan.' Below is a 'Deployment Slots' section with a sub-section 'Deployment Slots'. The table lists two slots:

| NAME | STATUS | APP SERVICE PLAN |
|--|---------|-----------------------------|
| myfunctionapp-dma PRODUCTION | 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.

The screenshot shows the Azure portal interface for changing an app service plan. On the left, there's a sidebar with various options like 'Scale up (App Service plan)', 'WebJobs', 'Push', etc. The 'Change App Service plan' option is selected and highlighted with a red box. The main area is titled 'Change App Service plan' and contains a summary of the current plan details (App Service plan: ASP-deletemenow-ab7c) and the destination plan details (App Service plan: (New) ASP-glengatest-premium-stage-a402). It also shows resource group (delete-me-now), region (Central US), and pricing tier (ElasticPremium (EP1)). At the bottom right is a large blue 'OK' button.

4. Select OK.

Limitations

Azure Functions deployment slots have the following limitations:

- The number of slots available to an app depends on the plan. The Consumption plan is only allowed one deployment slot. Additional slots are available for apps running under the App Service plan.
- Swapping a slot resets keys for apps that have an `AzureWebJobsSecretStorageType` app setting equal to `files`.
- Slots aren't available for the Linux Consumption plan.

Support levels

There are two levels of support for deployment slots:

- **General availability (GA):** Fully supported and approved for production use.
- **Preview:** Not yet supported, but is expected to reach GA status in the future.

| OS/HOSTING PLAN | LEVEL OF SUPPORT |
|---------------------|----------------------|
| Windows Consumption | General availability |
| Windows Premium | General availability |
| Windows Dedicated | General availability |
| Linux Consumption | Unsupported |
| Linux Premium | General availability |
| Linux Dedicated | General availability |

Next steps

- [Deployment technologies in Azure Functions](#)

Continuous delivery by using Azure DevOps

11/2/2020 • 4 minutes to read • [Edit Online](#)

You can automatically deploy your function to an Azure Functions app by using [Azure Pipelines](#).

You have two options for defining your pipeline:

- **YAML file:** A YAML file describes the pipeline. The file might have a build steps section and a release section.
The YAML file must be in the same repo as the app.
- **Template:** Templates are ready-made tasks that build or deploy your app.

YAML-based pipeline

To create a YAML-based pipeline, first build your app, and then deploy the app.

Build your app

How you build your app in Azure Pipelines depends on your app's programming language. Each language has specific build steps that create a deployment artifact. A deployment artifact is used to deploy your function app in Azure.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

You can use the following sample to create a YAML file to build a .NET app:

```
pool:  
  vmImage: 'VS2017-Win2016'  
steps:  
- script: |  
    dotnet restore  
    dotnet build --configuration Release  
- task: DotNetCoreCLI@2  
  inputs:  
    command: publish  
    arguments: '--configuration Release --output publish_output'  
    projects: '*.*proj'  
    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 must include one of the following YAML samples in your YAML file, depending on the hosting OS.

Windows function app

You can use the following snippet to deploy a Windows function app:

```
steps:
- task: AzureFunctionApp@1
  inputs:
    azureSubscription: '<Azure service connection>'
    appType: functionApp
    appName: '<Name of function app>'
    #Uncomment the next lines to deploy to a deployment slot
    #deployToSlotOrASE: true
    #resourceGroupName: '<Resource Group Name>'
    #slotName: '<Slot name>'
```

Linux function app

You can use the following snippet to deploy a Linux function app:

```
steps:
- task: AzureFunctionApp@1
  inputs:
    azureSubscription: '<Azure service connection>'
    appType: functionAppLinux
    appName: '<Name of function app>'
    #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>'
```

Template-based pipeline

Templates in Azure DevOps are predefined groups of tasks that build or deploy an app.

Build your app

How you build your app in Azure Pipelines depends on your app's programming language. Each language has specific build steps that create a deployment artifact. A deployment artifact is used to update your function app in Azure.

To use built-in build templates, when you create a new build pipeline, select **Use the classic editor** to create a pipeline by using designer templates.

Connect Select Configure Review

New pipeline

Where is your code?

-  Azure Repos Git YAML
Free private Git repositories, pull requests, and code search
-  Bitbucket Cloud YAML
Hosted by Atlassian
-  GitHub YAML
Home to the world's largest community of developers
-  GitHub Enterprise Server YAML
The self-hosted version of GitHub Enterprise
-  Other Git
Any Internet-facing Git repository
-  Subversion
Centralized version control by Apache

[Use the classic editor](#) to create a pipeline without YAML.

After you configure the source of your code, search for Azure Functions build templates. Select the template that matches your app language.

Select a template

Or start with an [Empty job](#)

Azure Functions X

Configuration as code

 [YAML](#)
Looking for a better experience to configure your pipelines using YAML files? Try the new YAML pipeline creation experience. [Learn more](#)

Others

-  [Azure Functions for .NET](#)
Build and package a .NET based Azure Functions application to be deployed on Azure Functions.
-  [Azure Functions for Node.js](#)
Build and package a Node.js based Azure Functions application to be deployed on Azure Functions.
-  [Azure Functions for Python](#)
Build and package a Python based Azure Functions application to be deployed on Azure Functions.

In some cases, build artifacts have a specific folder structure. You might need to select the **Prepend root folder name to archive paths** check box.

The screenshot shows the Azure DevOps Pipeline editor. On the left, there's a list of tasks: 'Get sources', 'Agent job 1', 'Build extensions', 'Use Node version 10.14.1', 'Install Application Dependencies', 'Run 'build' script', 'Remove extraneous packages', 'Archive files', and 'Publish Artifact: drop'. The 'Archive files' task is selected and highlighted with a red box. On the right, the task configuration pane is open for 'Archive files'. It includes fields for 'Root folder or file to archive', 'Archive type' (set to 'zip'), 'Archive file to create' (set to '\$(Build.ArtifactStagingDirectory)/\$(Build.BuildId).zip'), and 'Control Options' and 'Output Variables' sections.

JavaScript apps

If your JavaScript app has a dependency on Windows native modules, you must update the agent pool version to Hosted VS2017.

The screenshot shows the Azure DevOps Pipeline editor. The 'Pipeline' section is highlighted with a red box. Below it, the pipeline tasks are listed: 'Get sources', 'Agent job 1', 'Build extensions', 'Use Node version 10.14.1', 'Install Application Dependencies', 'Run 'build' script', 'Remove extraneous packages', 'Archive files', and 'Publish Artifact: drop'. To the right, a 'Agent pool' dropdown menu is open, showing options like 'Hosted', 'Hosted macOS', 'Hosted macOS High Sierra', 'Hosted Ubuntu 1604', 'Hosted VS2017' (which is highlighted with a red box), 'Hosted Windows 2019 with VS2019', 'Hosted Windows Container', and 'Private'. There is also a 'Default' option at the bottom.

Deploy your app

When you create a new release pipeline, search for the Azure Functions release template.

The screenshot shows the 'Add tasks' search interface. At the top, there are 'Add tasks' and 'Refresh' buttons. A search bar contains the text 'Azure Functions'. Below the search bar, a list of templates is shown, with 'Azure Functions' highlighted with a red box. The description for 'Azure Functions' is: 'Update a function app with .NET, Python, JavaScript, PowerShell, Java based web applications'.

Deploying to a deployment slot is not supported in the release template.

Create a build pipeline by using the 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 master 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 Action

12/4/2020 • 8 minutes to read • [Edit Online](#)

Use [GitHub Actions](#) to define a workflow to automatically build and deploy code to your Azure function app.

In GitHub Actions, a [workflow](#) is an automated process that you define in your GitHub repository. This process tells GitHub how to build and deploy your functions app project on GitHub.

A workflow is defined by a YAML (.yml) file in the `/.github/workflows/` path in your repository. This definition contains the various steps and parameters that make up the workflow.

For an Azure Functions workflow, the file has three sections:

| SECTION | TASKS |
|----------------|--|
| Authentication | Download a publish profile. Create a GitHub secret. |
| Build | Set up the environment. Build the function app. |
| Deploy | Deploy the function app. |

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 a GitHub repository.
 - [Quickstart: Create a function in Azure using Visual Studio Code](#)

Generate deployment credentials

The recommended way to authenticate with Azure Functions for GitHub Actions is by using a publish profile. You can also authenticate with a service principal. To learn more, see [this GitHub Actions repository](#).

After saving your publish profile credential as a [GitHub secret](#), you'll use this secret within your workflow to authenticate with Azure.

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](#).

myfunctionapp
App Service

Search (Ctrl+ /) | Get publish profile | Reset publish profile | Delete

Overview

- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Security

Functions

Click here to add tags

Resource group (change)
myResourceGroup

Status
Running

Location
Central US

Subscription (change)
Subscription

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

2. Save and copy the contents of the file.

Add the GitHub secret

1. In [GitHub](#), browse to your repository, select **Settings** > **Secrets** > **Add a new secret**.

Unwatch 1 | Star 0 | Fork 1 | Settings

Code Issues 0 Pull requests 0 ZenHub Actions Projects 0 Wiki Security Insights

Secrets

Secrets are environment variables that are **encrypted** and only exposed to selected actions.

Add a new secret

Name: AZURE_CREDENTIALS

Value:

```
{
  "clientId": "REDACTED",
  "clientSecret": "REDACTED",
  "subscriptionId": "REDACTED",
  "tenantId": "REDACTED",
  "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
  "resourceManagerEndpointUrl": "https://management.azure.com/",
  "activeDirectoryGraphResourceId": "https://graph.windows.net/",
  "sqlManagementEndpointUrl": "https://management.core.windows.net:8443/",
  "galleryEndpointUrl": "https://gallery.azure.com/",
  "managementEndpointUrl": "https://management.core.windows.net/"
}
```

Add secret

2. Add a new secret using `AZURE_FUNCTIONAPP_PUBLISH_PROFILE` for **Name**, the content of the publishing profile file for **Value**, and then select **Add secret**.

GitHub can now authenticate to your function app in Azure.

Create the environment

Setting up the environment is done using a language-specific publish setup action.

- [.NET](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

.NET (including ASP.NET) uses the `actions/setup-dotnet` action.

The following example shows the part of the workflow that sets up the environment:

```
- name: Setup DotNet 2.2.402 Environment
uses: actions/setup-dotnet@v1
with:
  dotnet-version: 2.2.402
```

Build the function app

This depends on the language and for languages supported by Azure Functions, this section should be the standard build steps of each language.

The following example shows the part of the workflow that builds the function app, which is language-specific:

- [.NET](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

```
env:
  AZURE_FUNCTIONAPP_PACKAGE_PATH: '.' # set this to the path to your web app project, defaults to the
repository root

- name: 'Resolve Project Dependencies Using Dotnet'
  shell: bash
  run: |
    pushd './${{ env.AZURE_FUNCTIONAPP_PACKAGE_PATH }}'
    dotnet build --configuration Release --output ./output
    popd
```

Deploy the function app

Use the `Azure/functions-action` action to deploy your code to a function app. This action has three parameters:

| PARAMETER | EXPLANATION |
|------------------------------|--|
| <code>app-name</code> | (Mandatory) The name of your function app. |
| <code>slot-name</code> | (Optional) The name of the deployment slot you want to deploy to. The slot must already be defined in your function app. |
| <code>publish-profile</code> | (Optional) The name of the GitHub secret for your publish profile. |

The following example uses version 1 of the `functions-action` and a `publish profile` for authentication

- [.NET](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Set up a .NET Linux workflow that uses a publish profile.

```
name: Deploy DotNet project to Azure function app with a Linux environment

on:
  [push]

env:
  AZURE_FUNCTIONAPP_NAME: your-app-name # set this to your application's name
  AZURE_FUNCTIONAPP_PACKAGE_PATH: '.'    # set this to the path to your web app project, defaults to the
repository root
  DOTNET_VERSION: '2.2.402'           # set this to the dotnet version to use

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: 'Checkout GitHub Action'
        uses: actions/checkout@master

      - name: Setup DotNet ${{ env.DOTNET_VERSION }} Environment
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: ${{ env.DOTNET_VERSION }}

      - name: 'Resolve Project Dependencies Using Dotnet'
        shell: bash
        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 }}
```

Set up a .NET Windows workflow that uses a publish profile.

```
name: Deploy DotNet project to Azure function app with a Windows environment

on:
  [push]

env:
  AZURE_FUNCTIONAPP_NAME: your-app-name # set this to your application's name
  AZURE_FUNCTIONAPP_PACKAGE_PATH: '.'    # set this to the path to your web app project, defaults to the
repository root
  DOTNET_VERSION: '2.2.402'           # set this to the dotnet version to use

jobs:
  build-and-deploy:
    runs-on: windows-latest
    steps:
      - name: 'Checkout GitHub Action'
        uses: actions/checkout@master

      - name: Setup DotNet ${{ env.DOTNET_VERSION }} Environment
        uses: actions/setup-dotnet@v1
        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 }}'
```

Next steps

[Learn more about Azure and GitHub integration](#)

Zip deployment for Azure Functions

11/2/2020 • 6 minutes to read • [Edit Online](#)

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.

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 may 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](#).

Deployment .zip file requirements

The .zip file that you use for push deployment must contain all of the files needed to run your function.

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.

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file and one or more subfolders. Each subfolder contains the code for a separate function. The folder structure is shown in the following representation:

```
FunctionApp
| - host.json
| - MyFirstFunction
| | - function.json
| | - ...
| - MySecondFunction
| | - function.json
| | - ...
| - SharedCode
| - bin
```

In version 2.x and higher of the Functions runtime, all functions in the function app must share the same language stack.

The [host.json](#) file contains runtime-specific configurations and is in the root folder of the function app. A `bin` folder contains packages and other library files that the function app requires. See the language-specific requirements for a function app project:

- C# class library (.csproj)
- C# script (.csx)
- F# script
- Java
- JavaScript
- Python

A function app includes all of the files and folders in the `wwwroot` directory. A .zip file deployment includes the contents of the `wwwroot` directory, but not the directory itself. When deploying a C# class library project, you must include the compiled library files and dependencies in a `bin` subfolder in your .zip package.

Download your function app files

When you are developing on a local computer, it's easy to create a .zip file of the function app project folder on your development computer.

However, you might have created your functions by using the editor in the Azure portal. You can download an existing function app project 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, and a red-highlighted 'Download app content' button. Below these are sections for Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggailey777), and URL (https://functions-ggailey777.azurewebsites.net). Further down, there are sections for Subscription ID, Location (South Central US), and App Service plan / pricing tier (SouthCentralUSPlan (Consumption)). At the bottom, there's a 'Configured features' section with links to 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:

```
https://<function_app>.scm.azurewebsites.net/api/zip/site/wwwroot/
```

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.

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.

```
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.

```
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.

```
curl -u <deployment_user> https://<app_name>.scm.azurewebsites.net/api/deployments
```

With PowerShell

The following example uses [Publish-AzWebapp](#) upload the .zip file. Replace the placeholders `<group-name>`, `<app-name>`, and `<zip-file-path>`.

```
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.

```
$username = "<deployment-user>"  
$password = "<deployment-password>"  
$apiUrl = "https://<app-name>.scm.azurewebsites.net/api/deployments"  
$base64AuthInfo = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes("{0}:{1}" -f $username, $password))  
$userAgent = "powershell/1.0"  
Invoke-RestMethod -Uri $apiUrl -Headers @{Authorization=("Basic {0}" -f $base64AuthInfo)} -UserAgent $userAgent -Method GET
```

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.

Next steps

[Continuous deployment for Azure Functions](#)

Run your Azure Functions from a package file

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 `d:\home\site\wwwroot` 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 from a package file:

- Reduces the risk of file copy locking issues.
- Can be deployed to a production app (with restart).
- You can be certain of the files that are running in your app.
- Improves the performance of [Azure Resource Manager deployments](#).
- May reduce cold-start times, particularly for JavaScript functions with large npm package trees.

For more information, see [this announcement](#).

Enabling functions to run from a package

To enable your function app to run from a package, you just add a `WEBSITE_RUN_FROM_PACKAGE` setting to your function app settings. The `WEBSITE_RUN_FROM_PACKAGE` setting can have one of the following values:

| VALUE | DESCRIPTION |
|--------------------------|--|
| <code>1</code> | Recommended for function apps running on Windows. Run from a package file in the <code>d:\home\data\SitePackages</code> folder of your function app. If not deploying with zip deploy , this option requires the folder to also have a file named <code>packagename.txt</code> . This file contains only the name of the package file in folder, without any whitespace. |
| <code><URL></code> | Location of a specific package file you want to run. When using Blob storage, you should use a private container with a Shared Access Signature (SAS) to enable the Functions runtime to access to the package. You can use the Azure Storage Explorer to upload package files to your Blob storage account. When you specify a URL, you must also sync triggers after you publish an updated package. |

Caution

When running a function app on Windows, the external URL option yields worse cold-start performance. When deploying your function app to Windows, you should set `WEBSITE_RUN_FROM_PACKAGE` to `1` and publish with zip deployment.

The following shows a function app configured to run from a .zip file hosted in Azure Blob storage:

Application settings

| APP SETTING NAME | VALUE |
|-----------------------------------|---|
| WEBSITE_RUN_FROM_PACKAGE | https://myblobstorage.blob.core.windows.net/content/MyFunction... |
| + Add new setting | |

NOTE

Currently, only .zip package files are supported.

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 `d:\home\data\SitePackages` folder. With the `WEBSITE_RUN_FROM_PACKAGE` app setting value of `1`, the zip deployment APIs copy your package to the `d:\home\data\SitePackages` folder instead of extracting the files to `d:\home\site\wwwroot`. It also creates the `packagename.txt` file. After a restart, 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. Before a restart, all existing function executions are allowed to complete or time out. To learn more, see [Deployment behaviors](#).

Adding 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](#).

Troubleshooting

- Run From Package makes `wwwroot` read-only, so you will receive an error when writing files to this directory.
- Tar and gzip formats are not supported.
- This feature does not compose with local cache.
- For improved cold-start performance, use the local Zip option (`WEBSITE_RUN_FROM_PACKAGE=1`).
- Run From Package is incompatible with deployment customization option (`SCM_DO_BUILD_DURING_DEPLOYMENT=true`), the build step will be ignored during deployment.

Next steps

[Continuous deployment for Azure Functions](#)

Azure Functions on Kubernetes with KEDA

11/2/2020 • 3 minutes to read • [Edit Online](#)

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.

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 0 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 using [Azure Functions Core Tools](#).

Installing with 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](#). If it doesn't already have one, you can add a Dockerfile by running the following command at the root of your Functions project:

NOTE

The Core Tools automatically create the Dockerfile for Azure Functions written in .NET, Node, Python, or PowerShell. For function apps written in Java, the Dockerfile must be created manually. Use the Azure Functions [image list](#) to find the correct image to base the Azure Function.

```
func init --docker-only
```

To build an image and deploy your functions to Kubernetes, run the following command:

NOTE

The Core Tools will leverage the docker CLI to build and publish the image. Be sure to have docker installed already and connected to your account with `docker login`.

```
func kubernetes deploy --name <name-of-function-deployment> --registry <container-registry-username>
```

Replace `<name-of-function-deployment>` with the name of your function app.

The deploy command executes a series of actions:

1. The Dockerfile created earlier is used to build a local image for the function app.
2. The local image is tagged and pushed to the container registry where the user is logged in.
3. 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 above flow works for private registries as well. If you are 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.

```
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

Steps to uninstall KEDA are documented [on the KEDA site](#).

Supported triggers in KEDA

KEDA has support for the following Azure Function triggers:

- [Azure Storage Queues](#)
- [Azure Service Bus Queues](#)
- [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 leverage the KEDA prometheus trigger to [scale HTTP Azure Functions from 1 to \$n\$ instances](#).

Next Steps

For more information, see the following resources:

- [Create a function using a custom image](#)
- [Code and test Azure Functions locally](#)
- [How the Azure Function Consumption plan works](#)

Automate resource deployment for your function app in Azure Functions

11/2/2020 • 10 minutes to read • [Edit Online](#)

You can use an Azure Resource Manager template to deploy a function app. This article outlines the required resources and parameters for doing so. You might need to deploy additional resources, depending on the [triggers and bindings](#) in your function app.

For more information about creating templates, see [Authoring Azure Resource Manager templates](#).

For sample templates, see:

- [Function app on Consumption plan](#)
- [Function app on Azure App Service plan](#)

Required resources

An Azure Functions deployment typically consists of these resources:

| RESOURCE | REQUIREMENT | SYNTAX AND PROPERTIES REFERENCE |
|---|-----------------------|---|
| A function app | Required | Microsoft.Web/sites |
| An Azure Storage account | Required | Microsoft.Storage/storageAccounts |
| An Application Insights component | Optional | Microsoft.Insights/components |
| A hosting plan | Optional ¹ | Microsoft.Web/serverfarms |

¹A hosting plan is only required when you choose to run your function app on a [Premium plan](#) or on an [App Service plan](#).

TIP

While not required, it is strongly recommended that you [configure Application Insights](#) for your app.

Storage account

An Azure storage account is required for a function app. You need a general purpose account that supports blobs, tables, queues, and files. For more information, see [Azure Functions storage account requirements](#).

```
{  
  "type": "Microsoft.Storage/storageAccounts",  
  "name": "[variables('storageAccountName')]",  
  "apiVersion": "2019-06-01",  
  "location": "[resourceGroup().location]",  
  "kind": "StorageV2",  
  "sku": {  
    "name": "[parameters('storageAccountType')]"  
  }  
}
```

In addition, the property `AzureWebJobsStorage` must be specified as an app setting in the site configuration. If the function app doesn't use Application Insights for monitoring, it should also specify `AzureWebJobsDashboard` as an app setting.

The Azure Functions runtime uses the `AzureWebJobsStorage` connection string to create internal queues. When Application Insights is not enabled, the runtime uses the `AzureWebJobsDashboard` connection string to log to Azure Table storage and power the **Monitor** tab in the portal.

These properties are specified in the `appSettings` collection in the `siteConfig` object:

```
"appSettings": [
  {
    "name": "AzureWebJobsStorage",
    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'),
';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
  },
  {
    "name": "AzureWebJobsDashboard",
    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'),
';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
  }
]
```

Application Insights

Application Insights is recommended for monitoring your function apps. The Application Insights resource is defined with the type **Microsoft.Insights/components** and the kind **web**:

```
{
  "apiVersion": "2015-05-01",
  "name": "[variables('appInsightsName')]",
  "type": "Microsoft.Insights/components",
  "kind": "web",
  "location": "[resourceGroup().location]",
  "tags": {
    "[concat('hidden-link:', resourceGroup().id, '/providers/Microsoft.Web/sites/',
variables('functionAppName'))]": "Resource"
  },
  "properties": {
    "Application_Type": "web",
    "ApplicationId": "[variables('appInsightsName')]"
  }
},
```

In addition, the instrumentation key needs to be provided to the function app using the `APPINSIGHTS_INSTRUMENTATIONKEY` application setting. This property is specified in the `appSettings` collection in the `siteConfig` object:

```
"appSettings": [
  {
    "name": "APPINSIGHTS_INSTRUMENTATIONKEY",
    "value": "[reference(resourceId('microsoft.insights/components/', variables('appInsightsName')),
'2015-05-01').InstrumentationKey]"
  }
]
```

Hosting plan

The definition of the hosting plan varies, and can be one of the following:

- [Consumption plan](#) (default)

- Premium plan
- App Service plan

Function app

The function app resource is defined by using a resource of type **Microsoft.Web/sites** and kind **functionapp**:

```
{
  "apiVersion": "2015-08-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[resourceGroup().location]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
    "[resourceId('Microsoft.Insights/components', variables('appInsightsName'))]"
  ]
}
```

IMPORTANT

If you are explicitly defining a hosting plan, an additional item would be needed in the dependsOn array:

```
"[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]"
```

A function app must include these application settings:

| SETTING NAME | DESCRIPTION | EXAMPLE VALUES |
|------------------------------|--|---|
| AzureWebJobsStorage | A connection string to a storage account that the Functions runtime uses for internal queueing | See Storage account |
| FUNCTIONS_EXTENSION_VERSION | The version of the Azure Functions runtime | ~2 |
| FUNCTIONS_WORKER_RUNTIME | The language stack to be used for functions in this app | dotnet , node , java , python , or powershell |
| WEBSITE_NODE_DEFAULT_VERSION | Only needed if using the node language stack, specifies the version to use | 10.14.1 |

These properties are specified in the `appSettings` collection in the `siteConfig` property:

```

"properties": {
    "siteConfig": {
        "appSettings": [
            {
                "name": "AzureWebJobsStorage",
                "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-
01').keys[0].value)]"
            },
            {
                "name": "FUNCTIONS_WORKER_RUNTIME",
                "value": "node"
            },
            {
                "name": "WEBSITE_NODE_DEFAULT_VERSION",
                "value": "10.14.1"
            },
            {
                "name": "FUNCTIONS_EXTENSION_VERSION",
                "value": "~2"
            }
        ]
    }
}

```

Deploy on Consumption plan

The Consumption plan automatically allocates compute power when your code is running, scales out as necessary to handle load, and then scales in when code is not running. You don't have to pay for idle VMs, and you don't have to reserve capacity in advance. To learn more, see [Azure Functions scale and hosting](#).

For a sample Azure Resource Manager template, see [Function app on Consumption plan](#).

Create a Consumption plan

A Consumption plan does not need to be defined. One will automatically be created or selected on a per-region basis when you create the function app resource itself.

The Consumption plan is a special type of "serverfarm" resource. For Windows, you can specify it by using the `Dynamic` value for the `computeMode` and `sku` properties:

```

{
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2016-09-01",
    "name": "[variables('hostingPlanName')]",
    "location": "[resourceGroup().location]",
    "properties": {
        "name": "[variables('hostingPlanName')]",
        "computeMode": "Dynamic"
    },
    "sku": {
        "name": "Y1",
        "tier": "Dynamic",
        "size": "Y1",
        "family": "Y",
        "capacity": 0
    }
}

```

NOTE

The Consumption plan cannot be explicitly defined for Linux. It will be created automatically.

If you do explicitly define your Consumption plan, you will need to set the `serverFarmId` property on the app so that it points to the resource ID of the plan. You should ensure that the function app has a `dependson` setting for the plan as well.

Create a function app

Windows

On Windows, a Consumption plan requires two additional settings in the site configuration:

`WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` and `WEBSITE_CONTENTSHARE`. These properties configure the storage account and file path where the function app code and configuration are stored.

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[resourceGroup().location]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
  ],
  "properties": {
    "siteConfig": {
      "appSettings": [
        {
          "name": "AzureWebJobsStorage",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "WEBSITE_CONTENTSHARE",
          "value": "[toLower(variables('functionAppName'))]"
        },
        {
          "name": "FUNCTIONS_WORKER_RUNTIME",
          "value": "node"
        },
        {
          "name": "WEBSITE_NODE_DEFAULT_VERSION",
          "value": "10.14.1"
        },
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        }
      ]
    }
  }
}
```

Linux

On Linux, the function app must have its `kind` set to `functionapp,linux`, and it must have the `reserved` property

set to `true`:

```
{  
    "apiVersion": "2016-03-01",  
    "type": "Microsoft.Web/sites",  
    "name": "[variables('functionAppName')]",  
    "location": "[resourceGroup().location]",  
    "kind": "functionapp,linux",  
    "dependsOn": [  
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"  
    ],  
    "properties": {  
        "siteConfig": {  
            "appSettings": [  
                {  
                    "name": "AzureWebJobsStorage",  
                    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',  
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountName'), '2019-06-  
01').keys[0].value)]"  
                },  
                {  
                    "name": "FUNCTIONS_WORKER_RUNTIME",  
                    "value": "node"  
                },  
                {  
                    "name": "WEBSITE_NODE_DEFAULT_VERSION",  
                    "value": "10.14.1"  
                },  
                {  
                    "name": "FUNCTIONS_EXTENSION_VERSION",  
                    "value": "~2"  
                }  
            ]  
        },  
        "reserved": true  
    }  
}
```

Deploy on Premium plan

The Premium plan offers the same scaling as the Consumption plan but includes dedicated resources and additional capabilities. To learn more, see [Azure Functions Premium Plan](#).

Create a Premium plan

A Premium plan is a special type of "serverfarm" resource. You can specify it by using either `EP1`, `EP2`, or `EP3` for the `Name` property value in the `sku` [description object](#).

```
{  
  "type": "Microsoft.Web/serverfarms",  
  "apiVersion": "2018-02-01",  
  "name": "[parameters('hostingPlanName')]",  
  "location": "[resourceGroup().location]",  
  "properties": {  
    "name": "[parameters('hostingPlanName')]",  
    "workerSize": "[parameters('workerSize')]",  
    "workerSizeId": "[parameters('workerSizeId')]",  
    "numberOfWorkers": "[parameters('numberOfWorkers')]",  
    "hostingEnvironment": "[parameters('hostingEnvironment')]",  
    "maximumElasticWorkerCount": "20"  
  },  
  "sku": {  
    "Tier": "ElasticPremium",  
    "Name": "EP1"  
  }  
}
```

Create a function app

A function app on a Premium plan must have the `serverFarmId` property set to the resource ID of the plan created earlier. In addition, a Premium plan requires two additional settings in the site configuration:

`WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` and `WEBSITE_CONTENTSHARE`. These properties configure the storage account and file path where the function app code and configuration are stored.

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[resourceGroup().location]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
  ],
  "properties": {
    "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "siteConfig": {
      "appSettings": [
        {
          "name": "AzureWebJobsStorage",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "WEBSITE_CONTENTSHARE",
          "value": "[toLower(variables('functionAppName'))]"
        },
        {
          "name": "FUNCTIONS_WORKER_RUNTIME",
          "value": "node"
        },
        {
          "name": "WEBSITE_NODE_DEFAULT_VERSION",
          "value": "10.14.1"
        },
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        }
      ]
    }
  }
}
```

Deploy on App Service plan

In the App Service plan, your function app runs on dedicated VMs on Basic, Standard, and Premium SKUs, similar to web apps. For details about how the App Service plan works, see the [Azure App Service plans in-depth overview](#).

For a sample Azure Resource Manager template, see [Function app on Azure App Service plan](#).

Create an App Service plan

An App Service plan is defined by a "serverfarm" resource.

```
{  
  "type": "Microsoft.Web/serverfarms",  
  "apiVersion": "2018-02-01",  
  "name": "[variables('hostingPlanName')]",  
  "location": "[resourceGroup().location]",  
  "sku": {  
    "name": "S1",  
    "tier": "Standard",  
    "size": "S1",  
    "family": "S",  
    "capacity": 1  
  }  
}
```

To run your app on Linux, you must also set the `kind` to `Linux`:

```
{  
  "type": "Microsoft.Web/serverfarms",  
  "apiVersion": "2018-02-01",  
  "name": "[variables('hostingPlanName')]",  
  "location": "[resourceGroup().location]",  
  "kind": "Linux",  
  "sku": {  
    "name": "S1",  
    "tier": "Standard",  
    "size": "S1",  
    "family": "S",  
    "capacity": 1  
  }  
}
```

Create a function app

A function app on an App Service plan must have the `serverFarmId` property set to the resource ID of the plan created earlier.

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[resourceGroup().location]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
  ],
  "properties": {
    "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "siteConfig": {
      "appSettings": [
        {
          "name": "AzureWebJobsStorage",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "FUNCTIONS_WORKER_RUNTIME",
          "value": "node"
        },
        {
          "name": "WEBSITE_NODE_DEFAULT_VERSION",
          "value": "10.14.1"
        },
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        }
      ]
    }
  }
}
```

Linux apps should also include a `linuxFxVersion` property under `siteConfig`. If you are just deploying code, the value for this is determined by your desired runtime stack:

| STACK | EXAMPLE VALUE |
|------------|---|
| Python | DOCKER microsoft/azure-functions-python3.6:2.0 |
| JavaScript | DOCKER microsoft/azure-functions-node8:2.0 |
| .NET | DOCKER microsoft/azure-functions-dotnet-core2.0:2.0 |

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[resourceGroup().location]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
  ],
  "properties": {
    "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "siteConfig": {
      "appSettings": [
        {
          "name": "AzureWebJobsStorage",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "FUNCTIONS_WORKER_RUNTIME",
          "value": "node"
        },
        {
          "name": "WEBSITE_NODE_DEFAULT_VERSION",
          "value": "10.14.1"
        },
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        }
      ],
      "linuxFxVersion": "DOCKER|microsoft/azure-functions-node8:2.0"
    }
  }
}
```

If you are [deploying a custom container image](#), you must specify it with `linuxFxVersion` and include configuration that allows your image to be pulled, as in [Web App for Containers](#). Also, set `WEBSITES_ENABLE_APP_SERVICE_STORAGE` to `false`, since your app content is provided in the container itself:

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[resourceGroup().location]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
  ],
  "properties": {
    "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "siteConfig": {
      "appSettings": [
        {
          "name": "AzureWebJobsStorage",
          "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2019-06-01').keys[0].value)]"
        },
        {
          "name": "FUNCTIONS_WORKER_RUNTIME",
          "value": "node"
        },
        {
          "name": "WEBSITE_NODE_DEFAULT_VERSION",
          "value": "10.14.1"
        },
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        },
        {
          "name": "DOCKER_REGISTRY_SERVER_URL",
          "value": "[parameters('dockerRegistryUrl')]"
        },
        {
          "name": "DOCKER_REGISTRY_SERVER_USERNAME",
          "value": "[parameters('dockerRegistryUsername')]"
        },
        {
          "name": "DOCKER_REGISTRY_SERVER_PASSWORD",
          "value": "[parameters('dockerRegistryPassword')]"
        },
        {
          "name": "WEBSITES_ENABLE_APP_SERVICE_STORAGE",
          "value": "false"
        }
      ],
      "linuxFxVersion": "DOCKER|myacr.azurecr.io/myimage:mytag"
    }
  }
}
```

Customizing a deployment

A function app has many child resources that you can use in your deployment, including app settings and source control options. You also might choose to remove the **sourcecontrols** child resource, and use a different [deployment option](#) instead.

IMPORTANT

To successfully deploy your application by using Azure Resource Manager, it's important to understand how resources are deployed in Azure. In the following example, 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. For example, when you are using functions with [Logic Apps](#), your functions are a dependency of another resource.

```
{
  "apiVersion": "2015-08-01",
  "name": "[parameters('appName')]",
  "type": "Microsoft.Web/sites",
  "kind": "functionapp",
  "location": "[parameters('location')]",
  "dependsOn": [
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
    "[resourceId('Microsoft.Web/serverfarms', parameters('appName'))]"
  ],
  "properties": {
    "serverFarmId": "[variables('appServicePlanName')]",
    "siteConfig": {
      "alwaysOn": true,
      "appSettings": [
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        },
        {
          "name": "Project",
          "value": "src"
        }
      ]
    }
  },
  "resources": [
    {
      "apiVersion": "2015-08-01",
      "name": "appsettings",
      "type": "config",
      "dependsOn": [
        "[resourceId('Microsoft.Web/Sites', parameters('appName'))]",
        "[resourceId('Microsoft.Web/Sites/sourcecontrols', parameters('appName'), 'web')]",
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
      ],
      "properties": {
        "AzureWebJobsStorage": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'), '2019-06-01').keys[0].value)]",
        "AzureWebJobsDashboard": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'), '2019-06-01').keys[0].value)]",
        "FUNCTIONS_EXTENSION_VERSION": "~2",
        "FUNCTIONS_WORKER_RUNTIME": "dotnet",
        "Project": "src"
      }
    },
    {
      "apiVersion": "2015-08-01",
      "name": "web",
      "type": "sourcecontrols",
      "dependsOn": [
        "[resourceId('Microsoft.Web/sites/', parameters('appName'))]"
      ],
      "properties": {
        "RepoUrl": "[parameters('sourceCodeRepositoryURL')]",
        "branch": "[parameters('sourceCodeBranch')]",
        "IsManualIntegration": "[parameters('sourceCodeManualIntegration')]"
      }
    }
  ]
}
```

TIP

This template uses the [Project](#) app settings value, which sets the base directory in which the Functions deployment engine (Kudu) looks for deployable code. In our repository, our functions are in a subfolder of the `src` folder. So, in the preceding example, we set the app settings value to `src`. If your functions are in the root of your repository, or if you are not deploying from source control, you can remove this app settings value.

Deploy your template

You can use any of the following ways to deploy your template:

- [PowerShell](#)
- [Azure CLI](#)
- [Azure portal](#)
- [REST API](#)

Deploy to Azure button

Replace `<url-encoded-path-to-azuredeploy-json>` with a [URL-encoded](#) version of the raw path of your `azuredeploy.json` file in GitHub.

Here is an example that uses markdown:

```
[![Deploy to Azure](https://azuredploy.net/deploybutton.png)]  
(https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-to-azuredploy-json>)
```

Here is an example that uses 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 template that create a function app with its required resources. To run locally, you must have [Azure PowerShell](#) installed. Run [Connect-AzAccount](#) to sign in.

```
# 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'  
  
# Create the parameters for the file, which for this template is the function app name.  
$TemplateParams = @{"appName" = "<function-app-name>"}  
  
# Deploy the template  
New-AzResourceGroupDeployment -ResourceGroupName "MyResourceGroup" -TemplateFile template.json -  
TemplateParameterObject $TemplateParams -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. Replace `<function-app-name>` with a unique name for your function app.

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](#)

Install the Azure Functions Runtime preview 2

11/2/2020 • 4 minutes to read • [Edit Online](#)

IMPORTANT

The Azure Functions Runtime preview 2 supports only version 1.x of the Azure Functions runtime. This preview feature is not being updated to support version 2.x and higher of the runtime, and no future updates are planned. If you need to host the Azure Functions runtime outside of Azure, consider using [Azure Functions on Kubernetes with KEDA](#)

If you would like to install the Azure Functions Runtime preview 2, follow these steps:

1. Ensure your machine passes the minimum requirements.
2. Download the [Azure Functions Runtime Preview Installer](#).
3. Uninstall the Azure Functions Runtime preview 1.
4. Install the Azure Functions Runtime preview 2.
5. Complete the configuration of the Azure Functions Runtime preview 2.
6. Create your first function in Azure Functions Runtime Preview

Prerequisites

Before you install the Azure Functions Runtime preview, you must have the following resources available:

1. A machine running Microsoft Windows Server 2016 or Microsoft Windows 10 Creators Update (Professional or Enterprise Edition).
2. A SQL Server instance running within your network. Minimum edition required is SQL Server Express.

Uninstall Previous Version

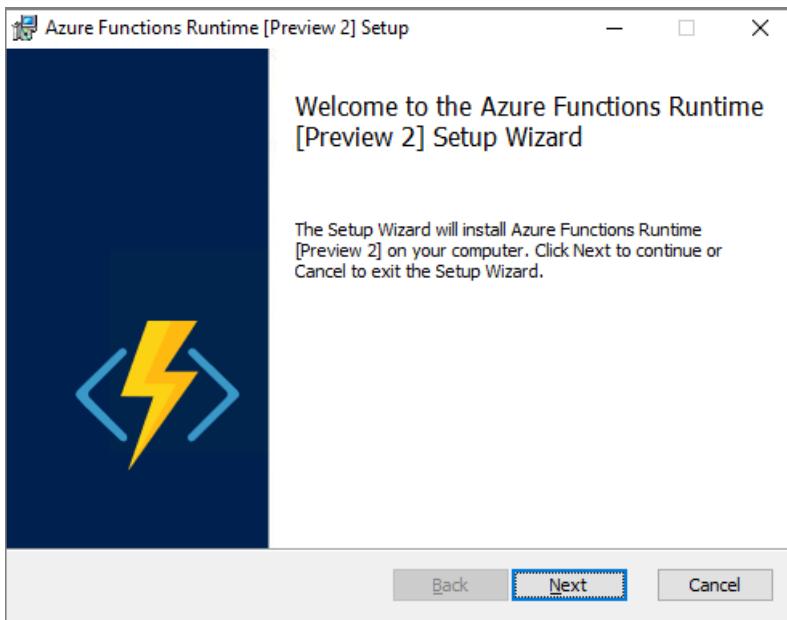
If you have previously installed the Azure Functions Runtime preview, you must uninstall before installing the latest release. Uninstall the Azure Functions Runtime preview by removing the program in Add/Remove Programs in Windows.

Install the Azure Functions Runtime Preview

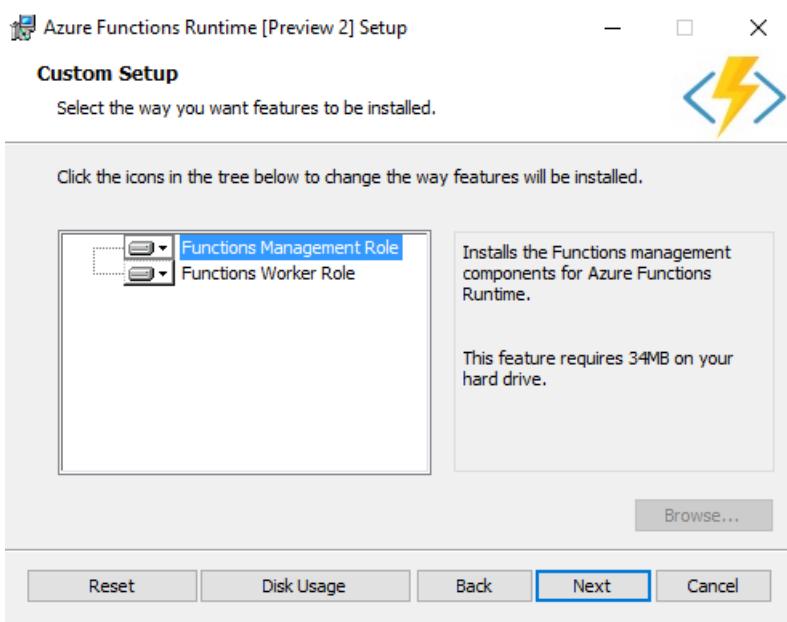
The Azure Functions Runtime Preview Installer guides you through the installation of the Azure Functions Runtime preview Management and Worker Roles. It is possible to install the Management and Worker role on the same machine. However, as you add more function apps, you must deploy more worker roles on additional machines to be able to scale your functions onto multiple workers.

Install the Management and Worker Role on the same machine

1. Run the Azure Functions Runtime Preview Installer.



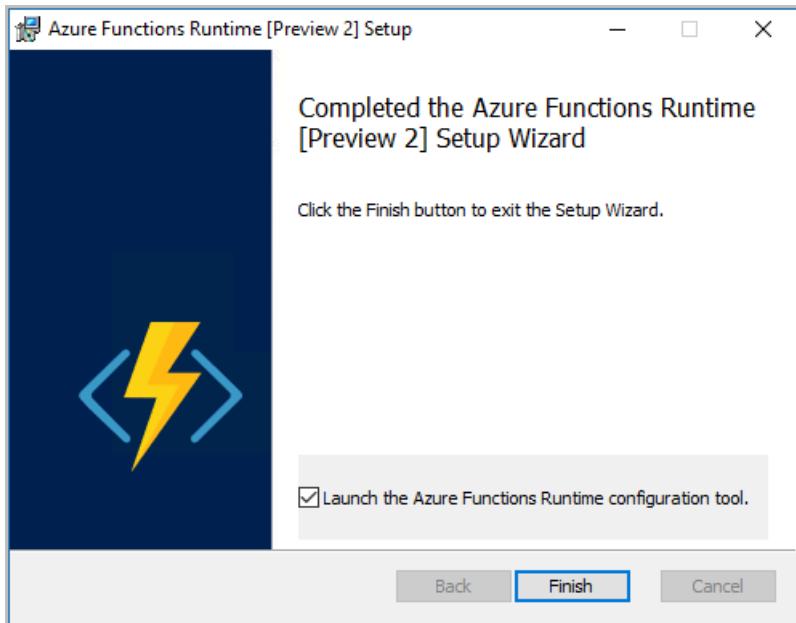
2. Click **Next**.
3. Once you have read the terms of the EULA, check the **box** to accept the terms and click **Next** to advance.
4. Select the roles you want to install on this machine **Functions Management Role** and/or **Functions Worker Role** and click **Next**.



NOTE

You can install the **Functions Worker Role** on many other machines. To do so, follow these instructions, and only select **Functions Worker Role** in the installer.

5. Click **Next** to have the **Azure Functions Runtime Setup Wizard** begin the installation process on your machine.
6. Once complete, the setup wizard launches the **Azure Functions Runtime** configuration tool.



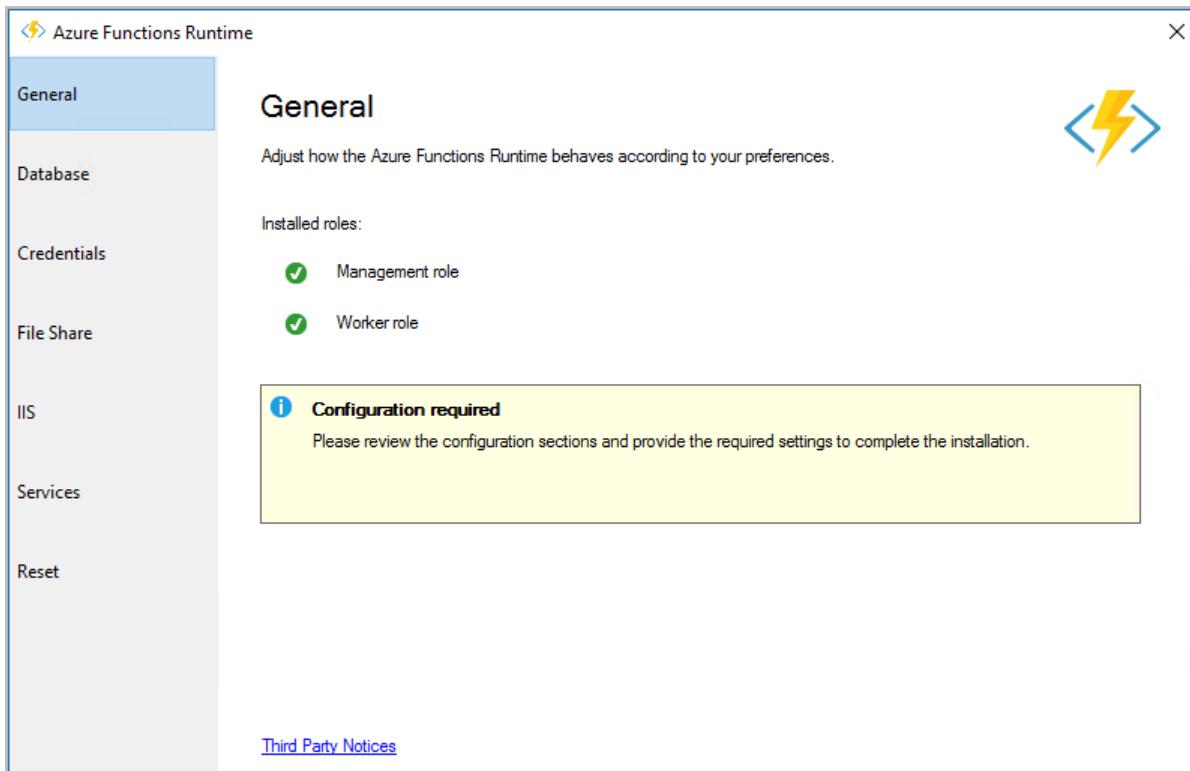
NOTE

If you are installing on **Windows 10** and the **Container** feature has not been previously enabled, the **Azure Functions Runtime Setup** prompts you to reboot your machine to complete the install.

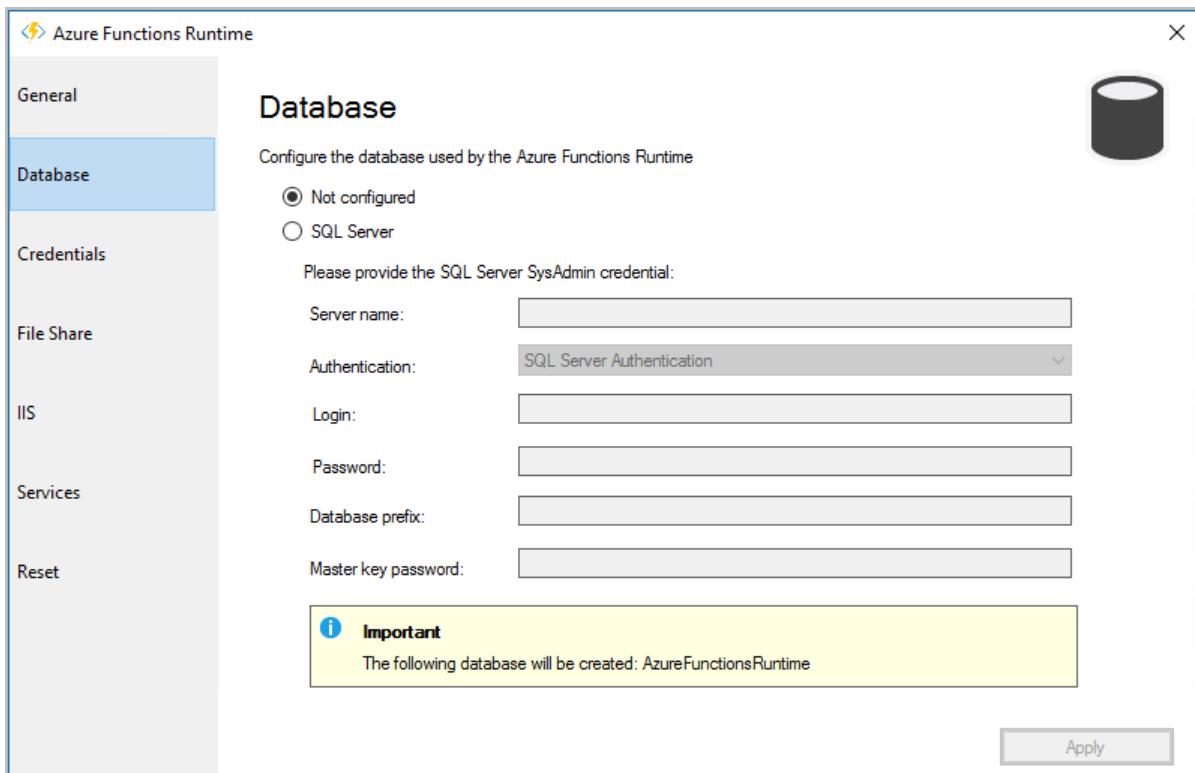
Configure the Azure Functions Runtime

To complete the Azure Functions Runtime installation, you must complete the configuration.

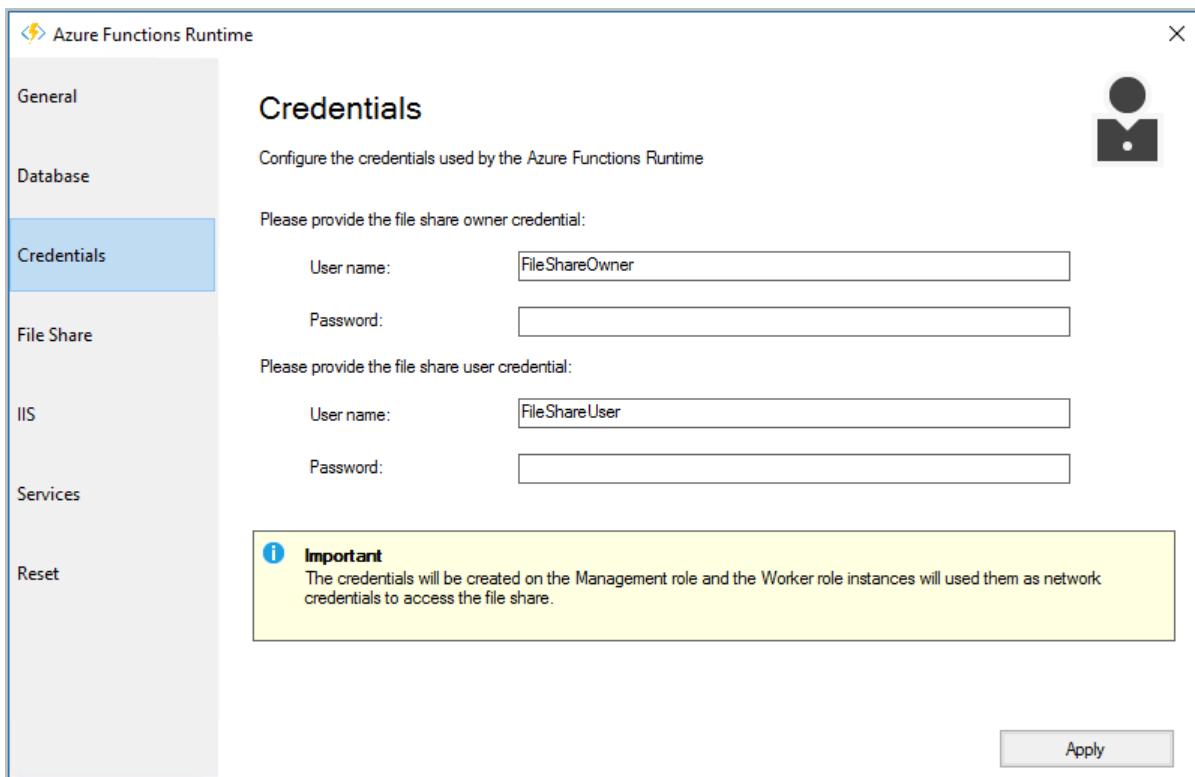
1. The **Azure Functions Runtime** configuration tool shows which roles are installed on your machine.



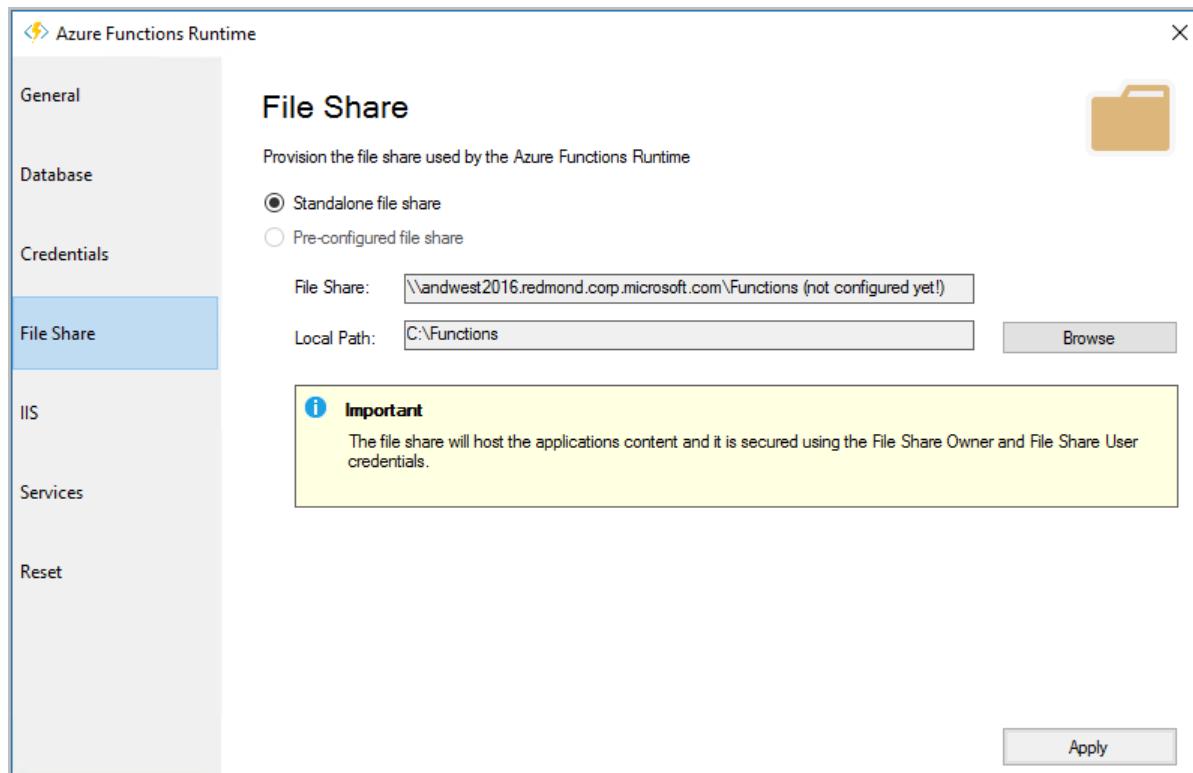
2. Click the **Database** tab, enter the connection details for your SQL Server instance, including specifying a **Database master key**, and click **Apply**. Connectivity to a SQL Server instance is required in order for the Azure Functions Runtime to create a database to support the Runtime.



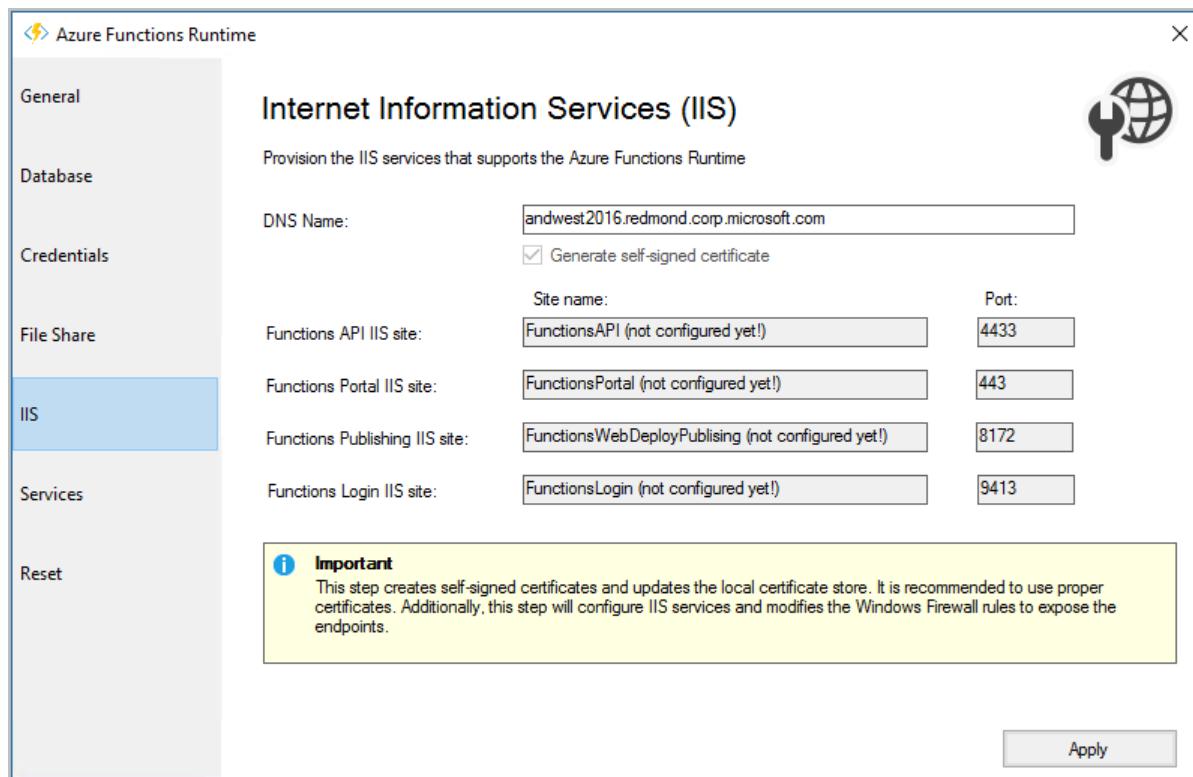
3. Click the **Credentials** tab. Here, you must create two new credentials for use with a file share for hosting all your function apps. Specify **User name** and **Password** combinations for the **file share owner** and for the **file share user**, then click **Apply**.



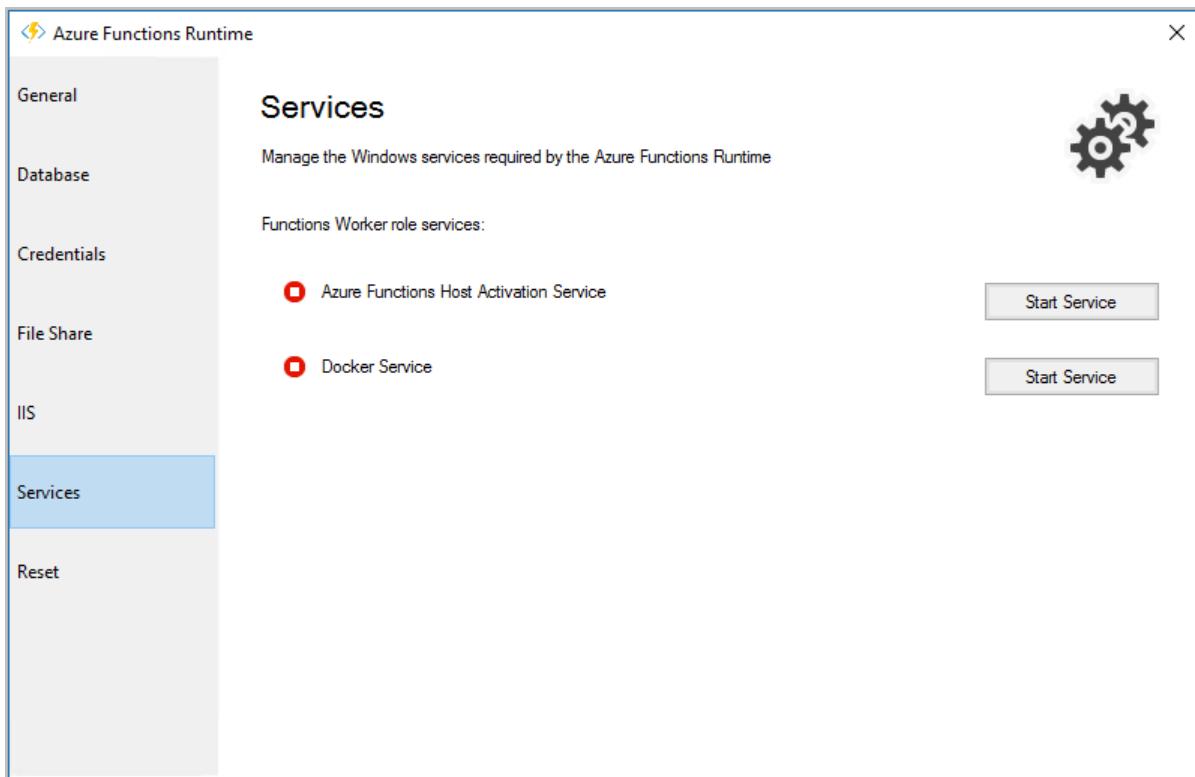
4. Click the **File Share** tab. Here you must specify the details of the file share location. The file share can be created for you or you can use an existing File Share and click **Apply**. If you select a new File Share location, you must specify a directory for use by the Azure Functions Runtime.



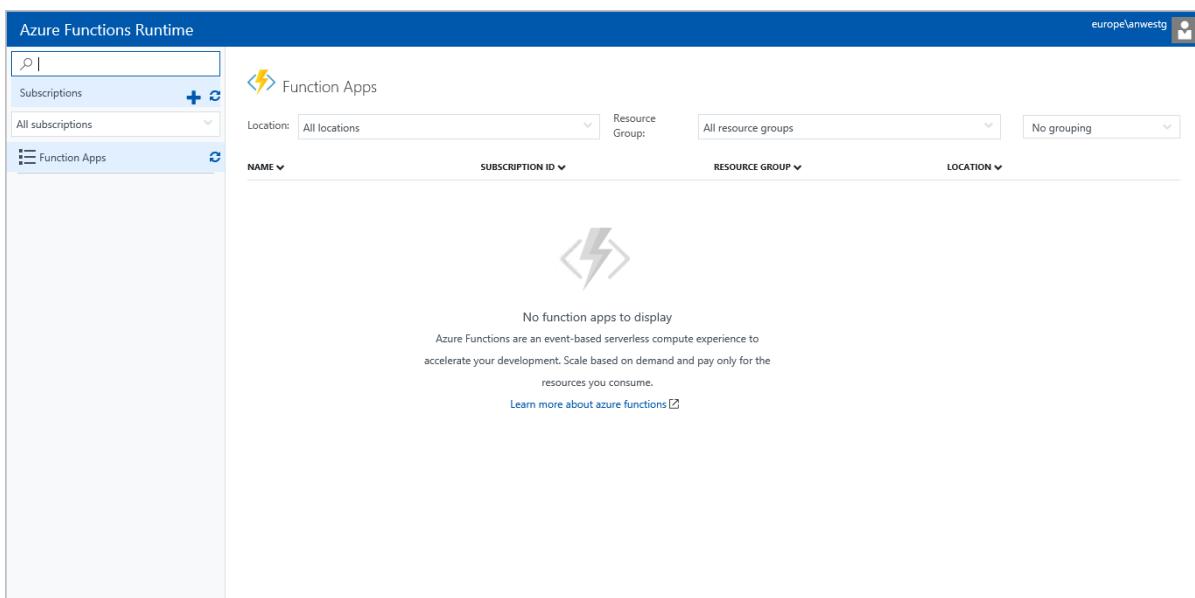
5. Click the **IIS** tab. This tab shows the details of the websites in IIS that the Azure Functions Runtime configuration tool creates. You may specify a custom DNS name here for the Azure Functions Runtime preview portal. Click **Apply** to complete.



6. Click the **Services** tab. This tab shows the status of the services in your Azure Functions Runtime configuration tool. If the **Azure Functions Host Activation Service** is not running after initial configuration, click **Start Service**.



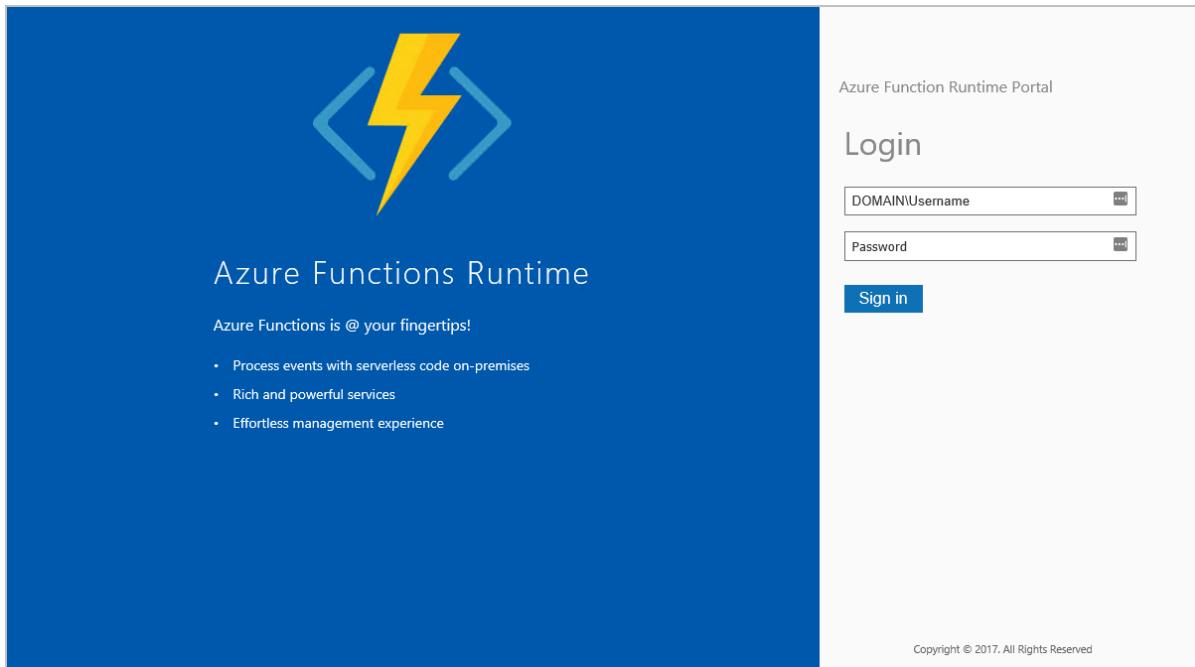
7. Browse to the Azure Functions Runtime Portal as <https://<machinename>.<domain>/>.



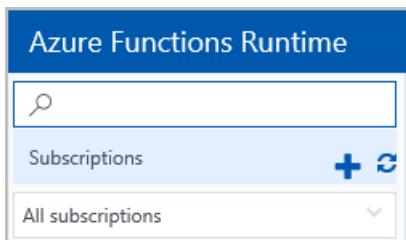
Create your first function in Azure Functions Runtime preview

To create your first function in Azure Functions Runtime preview

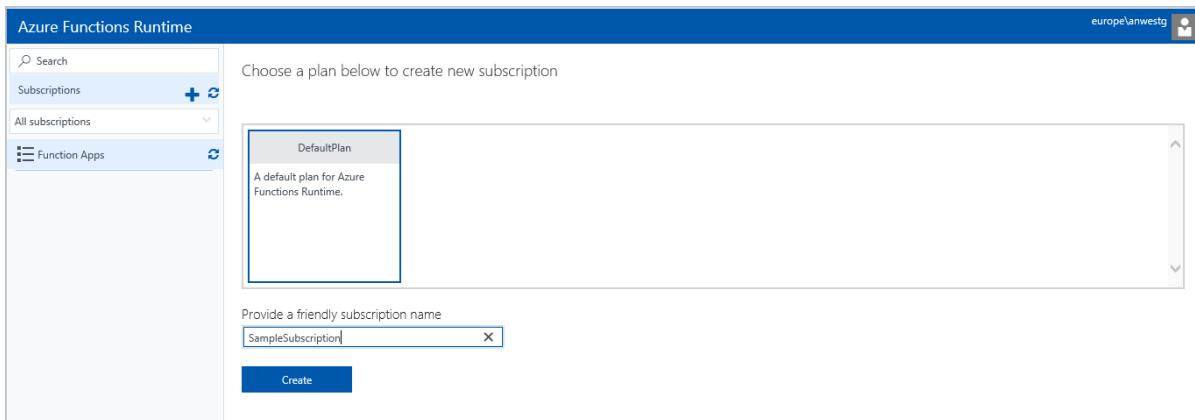
1. Browse to the Azure Functions Runtime Portal as <https://<machinename>.<domain>/> for example <https://mycomputer.mydomain.com>.
2. You are prompted to **Log in**, if deployed in a domain use your domain account username and password, otherwise use your local account username and password to log in to the portal.



3. To create function apps, you must create a Subscription. In the top left-hand corner of the portal, click the + option next to the subscriptions.



4. Choose **DefaultPlan**, enter a name for your Subscription, and click **Create**.



5. All of your function apps are listed in the left-hand pane of the portal. To create a new Function App, select the heading **Function Apps** and click the + option.
6. Enter a name for your function app, select the correct Subscription, choose which version of the Azure Functions runtime you wish to program against and click **Create**

New function app

Creating a Function App will automatically provision a new container capable of hosting and running your code. [Learn more](#)

| | |
|---------------|--|
| Name | <input type="text" value="Enter a name"/> |
| Subscription | SampleSubscription(2af55c88-65b2-4195-953c-4c10d97...) |
| Runtime image | v1(.net framework/server core) v2(.net core/nano server) |

[Create](#)

7. Your new function app is listed in the left-hand pane of the portal. Select Functions and then click **New Function** at the top of the center pane in the portal.

Azure Functions Runtime

Choose a template below or [go to the quickstart](#)

| | |
|--|---|
| Timer trigger A function that will be run on a specified schedule C# JavaScript PowerShell | Queue trigger A function that will be run whenever a message is added to a specified Azure Storage queue C# JavaScript PowerShell |
| Service Bus Queue trigger ServiceBusQueueTrigger_description C# JavaScript | Blob trigger A function that will be run whenever a blob is added to a specified container C# JavaScript |

8. Select the Timer Trigger function, in the right-hand flyout name your function and change the Schedule to `*/5 * * * *` (this cron expression causes your timer function to execute every five seconds), and click **Create**

Timer trigger

New Function

Language: [C#](#)

Name:

Timer trigger

Schedule [i](#)

[Create](#) [Cancel](#)

9. Your function has now been created. You can view the execution log of your Function app by expanding the **log** pane at the bottom of the portal.

Azure Functions Runtime

europe\anwestg 

run.csx 

Subscriptions 

All subscriptions 

Function Apps

TestFunction

Functions

TimerTriggerCSharp1

Integrate 

Manage 

Logs 

Pause  Clear  Copy logs  Expand 

```
1 using System;
2
3 public static void Run(TimerInfo myTimer, TraceWriter log)
4 {
5     log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
6 }
7
```

11/29/2017 10:25:05 AM [ANWESTG1] C# Timer trigger function executed at: 11/29/2017 10:25:04 AM
11/29/2017 10:25:05 AM [ANWESTG1] Function completed (Success, Id=f88202c8-c195-4f85-823d-a7deddfccb02, Duration=2ms)
11/29/2017 10:25:10 AM [ANWESTG1] Function started (Id=ab749622-5663-4fef-93ee-243e644283fe)
11/29/2017 10:25:10 AM [ANWESTG1] C# Timer trigger function executed at: 11/29/2017 10:25:10 AM
11/29/2017 10:25:10 AM [ANWESTG1] Function completed (Success, Id=ab749622-5663-4fef-93ee-243e644283fe, Duration=12ms)
11/29/2017 10:25:15 AM [ANWESTG1] Function started (Id=97840bfc-0024-4837-bf98-28d897a7800c)
11/29/2017 10:25:15 AM [ANWESTG1] C# Timer trigger function executed at: 11/29/2017 10:25:15 AM
11/29/2017 10:25:15 AM [ANWESTG1] Function completed (Success, Id=97840bfc-0024-4837-bf98-28d897a7800c, Duration=2ms)

Manage your function app

11/2/2020 • 4 minutes to read • [Edit Online](#)

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.

This article describes how to configure and manage your function apps.

TIP

Many configuration options can also be managed by using the [Azure CLI](#).

Get started in the Azure portal

1. To begin, go to the [Azure portal](#) and sign in to your Azure account. In the search bar at the top of the portal, enter the name of your function app and select it from the list.
2. Under **Settings** in the left pane, select **Configuration**.

| Name | Value | Source |
|---------------------------------------|--|------------|
| APPINSIGHTS_INSTRUMENTATIONKEY | Hidden value. Click show values button above to view | App Config |
| APPLICATIONINSIGHTS_CONNECTION_STRING | Hidden value. Click show values button above to view | App Config |
| AzureWebJobs.HttpTrigger1.Disabled | Hidden value. Click show values button above to view | App Config |
| AzureWebJobsStorage | Hidden value. Click show values button above to view | App Config |
| FUNCTIONS_EXTENSION_VERSION | Hidden value. Click show values button above to view | App Config |

You can navigate to everything you need to manage your function app from the overview page, in particular the [Application settings](#) and [Platform features](#).

Application settings

The **Application settings** tab maintains settings that are used by your function app. These settings are stored encrypted, and you must select **Show values** to see the values in the portal. You can also access application settings by using the Azure CLI.

Portal

To add a setting in the portal, select **New application setting** and add the new key-value pair.

The screenshot shows the Azure Portal interface for managing application settings and connection strings. The 'Application settings' section is highlighted with a red box around the 'New application setting' button. The 'Connection strings' section is also shown below it.

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... |
|--|--|---------------|
| APPINSIGHTS_INSTRUMENTATIONKEY | Hidden value. Click show values button abc | |
| AzureWebJobsStorage | Hidden value. Click show values button abc | |
| FUNCTIONS_EXTENSION_VERSION | Hidden value. Click show values button abc | |
| FUNCTIONS_WORKER_RUNTIME | Hidden value. Click show values button abc | |
| WEBSITE_CONTENTAZUREFILECONNECTIONSTRING | Hidden value. Click show values button abc | |
| WEBSITE_CONTENTSHARE | Hidden value. Click show values button abc | |
| WEBSITE_NODE_DEFAULT_VERSION | Hidden value. Click show values button abc | |
| WEBSITE_TIME_ZONE | Hidden value. Click show values button abc | |

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) | | | |
| (no connection strings to display) | | | |

Azure CLI

The `az functionapp config appsettings list` command returns the existing application settings, as in the following example:

```
az functionapp config appsettings list --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME>
```

The `az functionapp config appsettings set` command adds or updates an application setting. The following example creates a setting with a key named `CUSTOM_FUNCTION_APP_SETTING` and a value of `12345`:

```
az functionapp config appsettings set --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--settings CUSTOM_FUNCTION_APP_SETTING=12345
```

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 topics:

- [C# precompiled](#)
- [C# script \(.csx\)](#)
- [F# script \(.fsx\)](#)
- [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. To learn more, see [Local settings file](#).

Platform features

Function apps run in, and are maintained by, the Azure App Service platform. As such, your function apps have access to most of the features of Azure's core web hosting platform. The left pane is where you access the many features of the App Service platform that you can use in your function apps.

NOTE

Not all App Service features are available when a function app runs on the Consumption hosting plan.

The rest of this article focuses on the following App Service features in the Azure portal that are useful for Functions:

- [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 screenshot shows the Azure App Service Editor interface. On the left is a file explorer sidebar titled 'EXPLORE' showing project files like 'index.js', 'function.json', and various trigger files. The main area is a code editor with the following content:

```
index.js HttpTriggerJS1
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};
```

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 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.

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 portal. To learn more, see [Code and test Azure Functions locally](#).

Console

The screenshot shows the in-portal console window. It features a dark theme with a light-colored sidebar containing a file tree icon and a command history icon. The main area displays a command-line interface with the following output:

```
Console
connectedfunctions

D:\home\site\wwwroot
> |
```

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.

When developing locally, we recommend using the [Azure Functions Core Tools](#) and the [Azure CLI](#).

Advanced tools (Kudu)

The screenshot shows the Kudu interface, which consists of two main sections. The top section is a file browser with a header bar containing links for 'Kudu', 'Environment', 'Debug console', 'Process explorer', 'Tools', and 'Site extensions'. Below the header is a breadcrumb navigation bar with the path '/'. There are three items listed: '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). The bottom section is a 'Kudu Remote Execution Console' window. It displays a Windows command prompt with the following text:
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>

A small 'Use old console' link is visible in the top right corner of the console window.

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, like <https://<myfunctionapp>.scm.azurewebsites.net/>

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](#).

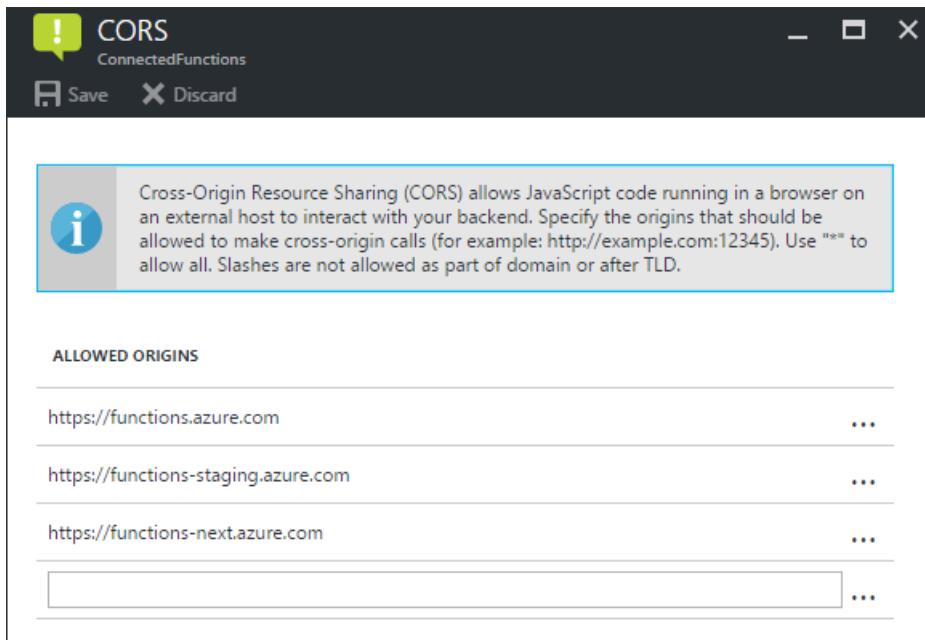
Cross-origin resource sharing

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.

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.



When the wildcard (`*`) is used, all other domains are ignored.

Use the `az functionapp cors add` command to add a domain to the allowed origins list. The following example adds the contoso.com domain:

```
az functionapp cors add --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--allowed-origins https://contoso.com
```

Use the `az functionapp cors show` command to list the current allowed origins.

Authentication

The screenshot shows two overlapping windows. The left window is titled 'Authentication / Authorization' and contains sections for 'App Service Authentication' (set to 'On'), 'Action to take when request is not authenticated' (set to 'Log in with Azure Active Directory'), and 'Authentication Providers' (listing Azure Active Directory, Facebook, Google, and Twitter, all set to 'Not Configured'). The right window is titled 'Microsoft Account Authentication Settings' and lists various OAuth scopes with their descriptions, such as wl.basic, wl.offline_access, wl.signin, etc. Both windows have 'Save' and 'Discard' buttons at the top.

When functions use an HTTP trigger, you can require calls to first be authenticated. App Service supports Azure Active Directory authentication and sign-in with social providers, such as Facebook, Microsoft, and Twitter. For details on configuring specific authentication providers, see [Azure App Service authentication overview](#).

Next steps

- [Configure Azure App Service Settings](#)
- [Continuous deployment for Azure Functions](#)

How to target Azure Functions runtime versions

11/2/2020 • 4 minutes to read • [Edit Online](#)

A function app runs on a specific version of the Azure Functions runtime. There are three major versions: [1.x](#), [2.x](#), and [3.x](#). By default, function apps are created in version 3.x of the runtime. This article explains how to configure a function app in Azure to run on the version you choose. For information about how to configure a local development environment for a specific version, see [Code and test Azure Functions locally](#).

Automatic and manual version updates

Azure Functions lets you target a specific version of the runtime by using the `FUNCTIONS_EXTENSION_VERSION` application setting in a function app. The function app is kept on the specified major version until you explicitly choose to move to a new version. If you specify only the major version, the function app is automatically updated to new minor versions of the runtime when they become available. New minor versions shouldn't introduce breaking changes.

If you specify a minor version (for example, "2.0.12345"), the function app is pinned to that specific version until you explicitly change it. Older minor versions are regularly removed from the production environment. After this occurs, your function app runs on the latest version instead of the version set in `FUNCTIONS_EXTENSION_VERSION`. Because of this, you should quickly resolve any issues with your function app that require a specific minor version, so that you can instead target the major version. Minor version removals are announced in [App Service announcements](#).

NOTE

If you pin to a specific major version of Azure Functions, and then try to publish to Azure using Visual Studio, a dialog window will pop up prompting you to update to the latest version or cancel the publish. To avoid this, add the `<DisableFunctionExtensionVersionUpdate>true</DisableFunctionExtensionVersionUpdate>` property in your `.csproj` file.

When a new version is publicly available, a prompt in the portal gives you the chance to move up to that version. After moving to a new version, you can always use the `FUNCTIONS_EXTENSION_VERSION` application setting to move back to a previous version.

The following table shows the `FUNCTIONS_EXTENSION_VERSION` values for each major version to enable automatic updates:

| MAJOR VERSION | FUNCTIONS_EXTENSION_VERSION | VALUE |
|---------------|-----------------------------|-----------------|
| 3.x | | <code>~3</code> |
| 2.x | | <code>~2</code> |
| 1.x | | <code>~1</code> |

A change to the runtime version causes a function app to restart.

View and update the current runtime version

You can change the runtime version used by your function app. Because of the potential of breaking changes, you

can only change the runtime version before you have created any functions in your function app.

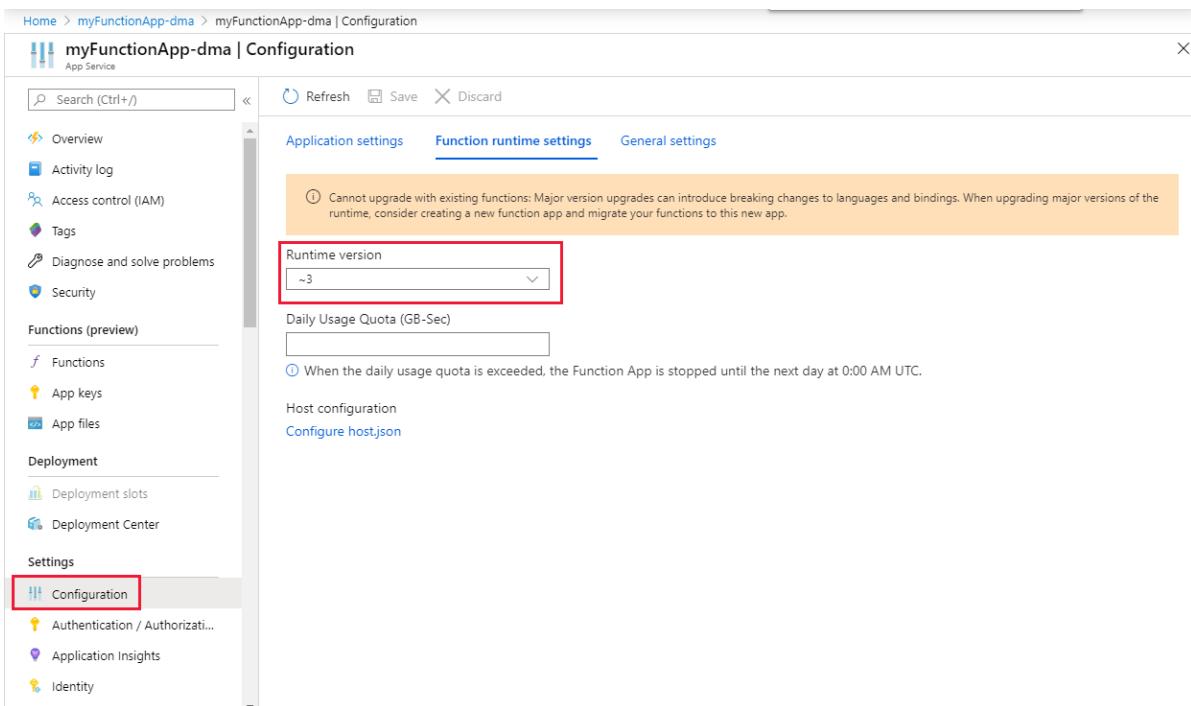
IMPORTANT

Although the runtime version is determined by the `FUNCTIONS_EXTENSION_VERSION` setting, you should make this change in the Azure portal and not by changing the setting directly. This is because the portal validates your changes and makes other related changes as needed.

- [Portal](#)
- [Azure CLI](#)
- [PowerShell](#)

Use the following procedure to view and update the runtime version currently used by a function app.

1. In the [Azure portal](#), browse to your function app.
2. Under **Settings**, choose **Configuration**. In the **Function runtime settings** tab, locate the **Runtime version**. Note the specific runtime version. In the example below, the version is set to `~3`.



3. To pin your function app to the version 1.x runtime, choose `~1` under **Runtime version**. This switch is disabled when you have functions in your app.
4. When you change the runtime version, go back to the **Overview** tab and choose **Restart** to restart the app. The function app restarts running on the version 1.x runtime, and the version 1.x templates are used when you create functions.

Home > myFunctionApp-dma > myFunctionApp-dma

myFunctionApp-dma App Service

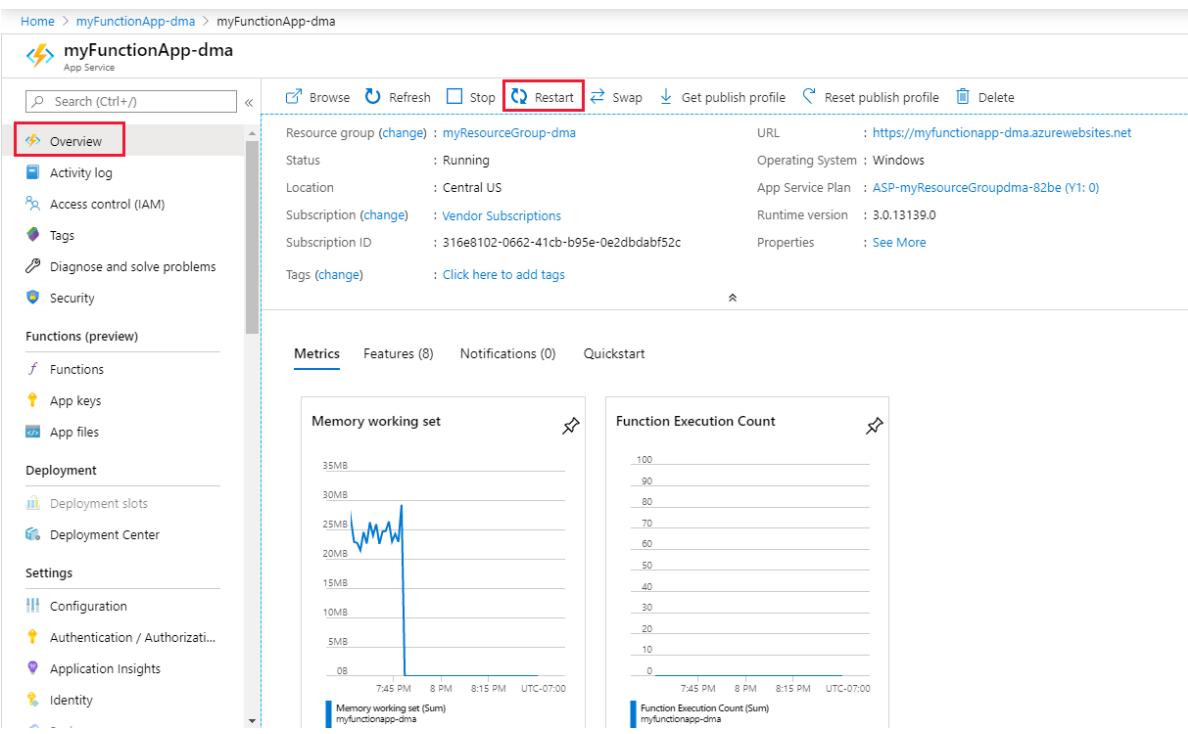
Search (Ctrl+ /) Overview Activity log Access control (IAM) Tags Diagnose and solve problems Security Functions (preview) Functions App keys App files Deployment Deployment slots Deployment Center Settings Configuration Authentication / Authorizati... Application Insights Identity

Browse Refresh Stop Restart Swap Get publish profile Reset publish profile Delete

Resource group (change) : myResourceGroup-dma Status : Running Location : Central US Subscription (change) : Vendor Subscriptions Subscription ID : 316e8102-0662-41cb-b95e-0e2dbdabf52c Tags (change) : Click here to add tags URL : https://myfunctionapp-dma.azurewebsites.net Operating System : Windows App Service Plan : ASP-myResourceGroupdma-82be (Y1: 0) Runtime version : 3.0.13139.0 Properties : See More

Metrics Features (8) Notifications (0) Quickstart

Memory working set Function Execution Count



NOTE

Using the Azure portal, you can't change the runtime version for a function app that already contains functions.

The function app restarts after the change is made to the application setting.

Next steps

[Target the 2.0 runtime in your local development environment](#)

[See Release notes for runtime versions](#)

How to disable functions in Azure Functions

11/2/2020 • 3 minutes to read • [Edit Online](#)

This article explains how to disable a function in Azure Functions. To *disable* a function means to make the runtime ignore the automatic trigger that's defined for the function. This lets you prevent a specific function from running without stopping the entire function app.

The recommended way to disable a function is with an app setting in the format

`AzureWebJobs.<FUNCTION_NAME>.Disabled` set to `true`. You can create and modify this application setting in a number of ways, including by using the [Azure CLI](#) and from your function's **Manage** tab in the [Azure portal](#).

NOTE

When you disable an HTTP triggered function by using the methods described in this article, the endpoint may still be accessible when running on your local computer.

Use the Azure CLI

In the Azure CLI, you use the `az functionapp config appsettings set` command to create and modify the app setting. The following command disables a function named `QueueTrigger` by creating an app setting named `AzureWebJobs.QueueTrigger.Disabled` set it to `true`.

```
az functionapp config appsettings set --name <myFunctionApp> \
--resource-group <myResourceGroup> \
--settings AzureWebJobs.QueueTrigger.Disabled=true
```

To re-enable the function, rerun the same command with a value of `false`.

```
az functionapp config appsettings set --name <myFunctionApp> \
--resource-group <myResourceGroup> \
--settings AzureWebJobs.QueueTrigger.Disabled=false
```

Use the Portal

You can also 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 it's disabled.

The screenshot shows the Azure Functions portal interface. At the top, there's a header with the function name 'HttpTrigger1 (myfunctionapp/HttpTrigger1)'. Below the header is a toolbar with buttons for 'Enable' (marked with a checkmark), 'Disable' (marked with a circle), 'Delete', 'Get Function Url', and 'Refresh'. A search bar is also present. On the left, there's a sidebar with links: 'Overview' (which is selected and highlighted with a red box), 'Developer', 'Code / Test', 'Integration', 'Monitor', and 'Function Keys'. The main content area displays basic information about the function app: 'Function app myfunctionapp', 'Status Enabled', and 'Application Insights myfunctionapp'.

NOTE

The portal-integrated testing functionality ignores the `Disabled` setting. This means that a disabled function still runs when started from the **Test** window in the portal.

local.settings.json

Functions can be disabled in the same way when running locally. To disable a function named `HttpExample`, add an entry to the `Values` collection in the `local.settings.json` file, as follows:

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "FUNCTIONS_WORKER_RUNTIME": "python",  
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
        "AzureWebJobs.HttpExample.Disabled": "true"  
    }  
}
```

Other methods

While the application setting method is recommended for all languages and all runtime versions, there are several other ways to disable functions. These methods, which vary by language and runtime version, are maintained for backward compatibility.

C# class libraries

In a class library function, you can also use the `Disable` attribute to prevent the function from being triggered. You can use the attribute without a constructor parameter, as shown in the following example:

```

public static class QueueFunctions
{
    [Disable]
    [FunctionName("QueueTrigger")]
    public static void QueueTrigger(
        [QueueTrigger("myqueue-items")] string myQueueItem,
        TraceWriter log)
    {
        log.Info($"C# function processed: {myQueueItem}");
    }
}

```

The attribute without a constructor parameter requires that you recompile and redeploy the project to change the function's disabled state. A more flexible way to use the attribute is to include a constructor parameter that refers to a Boolean app setting, as shown in the following example:

```

public static class QueueFunctions
{
    [Disable("MY_TIMER_DISABLED")]
    [FunctionName("QueueTrigger")]
    public static void QueueTrigger(
        [QueueTrigger("myqueue-items")] string myQueueItem,
        TraceWriter log)
    {
        log.Info($"C# function processed: {myQueueItem}");
    }
}

```

This method lets you enable and disable the function by changing the app setting, without recompiling or redeploying. Changing an app setting causes the function app to restart, so the disabled state change is recognized immediately.

IMPORTANT

The `Disabled` attribute is the only way to disable a class library function. The generated `function.json` file for a class library function is not meant to be edited directly. If you edit that file, whatever you do to the `disabled` property will have no effect.

The same goes for the **Function state** switch on the **Manage** tab, since it works by changing the `function.json` file.

Also, note that the portal may indicate the function is disabled when it isn't.

Functions 1.x - scripting languages

In version 1.x, you can also use the `disabled` property of the `function.json` file to tell the runtime not to trigger a function. This method only works for scripting languages such as C# script and JavaScript. The `disabled` property can be set to `true` or to the name of an app setting:

```
{  
  "bindings": [  
    {  
      "type": "queueTrigger",  
      "direction": "in",  
      "name": "myQueueItem",  
      "queueName": "myqueue-items",  
      "connection": "MyStorageConnectionAppSetting"  
    }  
  ],  
  "disabled": true  
}
```

or

```
"bindings": [  
  ...  
,  
  "disabled": "IS_DISABLED"
```

In the second example, the function is disabled when there is an app setting that is named IS_DISABLED and is set to `true` or 1.

IMPORTANT

The portal now uses application settings to disable v1.x functions. When an application setting conflicts with the function.json file, an error can occur. You should remove the `disabled` property from the function.json file to prevent errors.

Next steps

This article is about disabling automatic triggers. For more information about triggers, see [Triggers and bindings](#).

Azure Functions geo-disaster recovery

11/2/2020 • 3 minutes to read • [Edit Online](#)

When entire Azure regions or datacenters experience downtime, it is critical for compute to continue processing in a different region. This article will explain some of the strategies that you can use to deploy functions to allow for disaster recovery.

Basic concepts

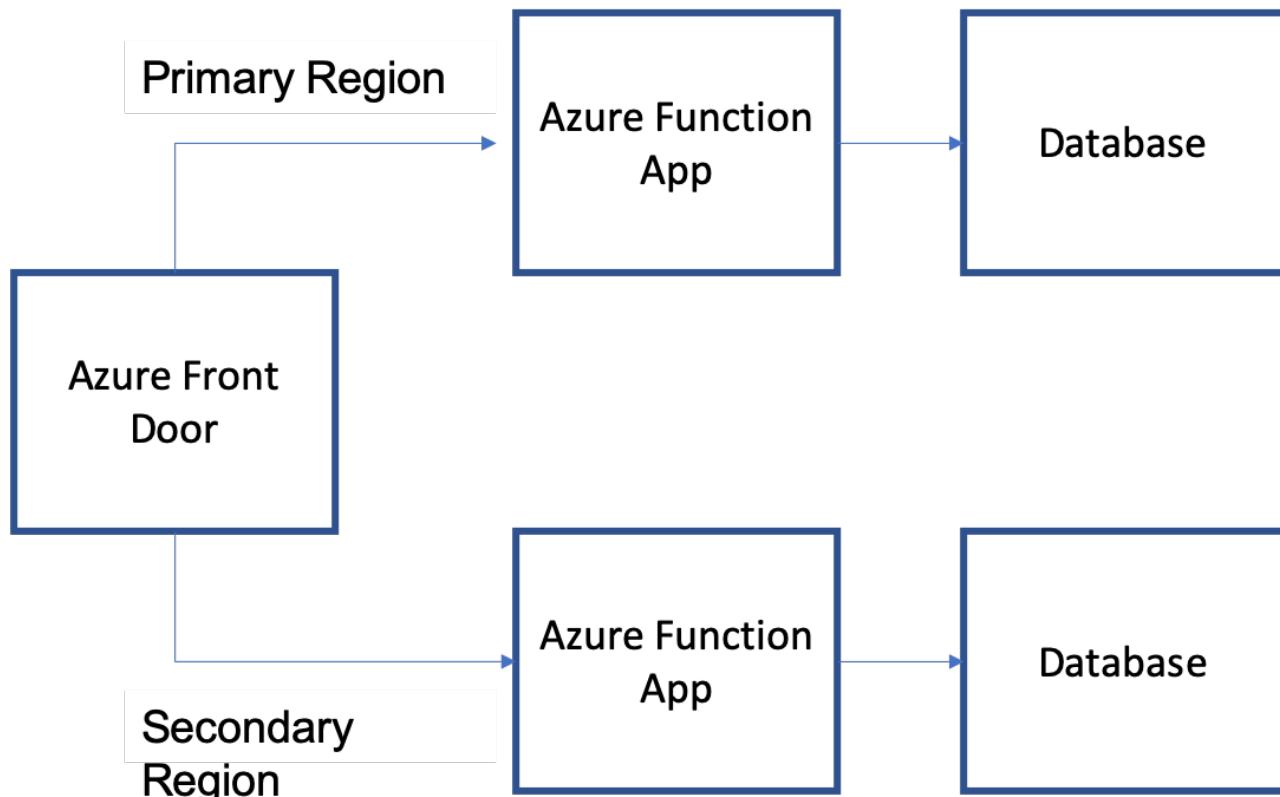
Azure Functions run in a specific region. To get higher availability, you can deploy the same functions to multiple regions. When in multiple regions you can have your functions running in the *active/active* pattern or *active/passive* pattern.

- Active/active. Both regions are active and receiving events (duplicate or rotationally). Active/active is recommended for HTTPS functions in combination with Azure Front Door.
- Active/passive. One region is active and receiving events, while a secondary is idle. When failover is required, the secondary region is activated and takes over processing. This is recommended for non-HTTP functions like Service Bus and Event Hubs.

Read how to [run apps in multiple regions](#) for more information on multi-region deployments.

Active/active for HTTPS functions

To achieve active/active deployments of functions, it requires some component that can coordinate the events between both regions. For HTTPS functions, this coordination is accomplished using [Azure Front Door](#). Azure Front Door can route and round-robin HTTPS requests between multiple regional functions. It also periodically checks the health of each endpoint. If a regional function stops responding to health checks, Azure Front Door will take it out of rotation and only forward traffic to healthy functions.



Active/active for non-HTTPS functions

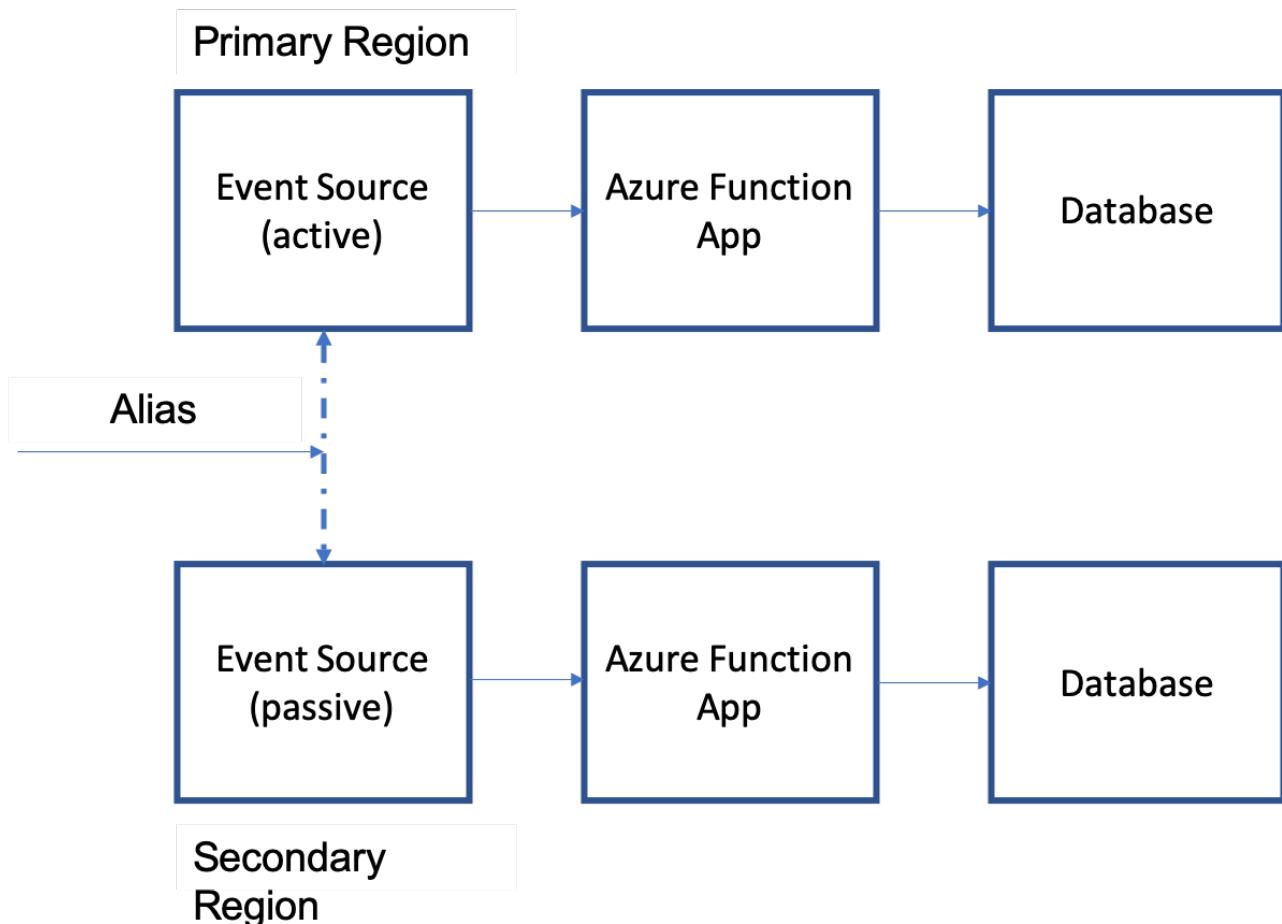
You can still achieve active/active deployments for non-HTTPS functions. However, you need to consider how the two regions will interact or coordinate with one another. If you deployed the same function app into two regions, each triggering on the same Service Bus queue, they would act as competing consumers on de-queueing that queue. While this means each message is only being processed by one of the instances, it also means there is still a single point of failure on the single Service Bus. If you deploy two Service Bus queues (one in a primary region, one in a secondary region), and the two function apps pointed to their region queue, the challenge now comes in 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 an active/active pattern, it creates other challenges around duplication of compute and when or how data is consolidated. For these reasons, it is recommended for non-HTTPS triggers to use the active/passive pattern.

Active/passive for non-HTTPS functions

Active/passive provides a way for only a single function to process each message, but provides a mechanism to fail over to a secondary region in case of a disaster. Azure Functions works alongside [Azure Service Bus geo-recovery](#) and [Azure Event Hubs geo-recovery](#).

Using Azure Event Hubs triggers as an example, the active/passive pattern would involve the following:

- Azure Event Hub deployed to both a primary and secondary region.
- Geo-disaster enabled to pair the primary and secondary Event Hub. 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 deployed to both a primary and secondary region.
- The function apps are triggering 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 will route to the primary event hub. The primary function app

is listening exclusively to the primary event hub. The secondary function app will be passive and idle. As soon as failover is initiated, publishers sending to the shared alias will now route to the secondary event hub. The secondary function app will now become active and start 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

- [Create Azure Front Door](#)
- [Event Hubs failover considerations](#)

How to configure monitoring for Azure Functions

12/4/2020 • 11 minutes to read • [Edit Online](#)

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. 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. To learn more about Application Insights costs, see [Manage usage and costs for Application Insights](#).

Later in this article, you learn how to configure and customize the data that your functions send to Application Insights. For a function app, logging is configured in the [host.json](#) file.

NOTE

You can use specially configured application settings to represent specific settings in a host.json file for a specific environment. This lets you effectively change host.json settings without having to republish the host.json file in your project. To learn more, see [Override host.json values](#).

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. The following chart describes the main categories of logs that the runtime creates.

- [v2.x+](#)
- [v1.x](#)

| CATEGORY | TABLE | DESCRIPTION |
|------------------------------------|-------------------------------|--|
| Function.<YOUR_FUNCTION_NAME> | dependencies | Dependency data is automatically collected for some services. For successful runs, these logs are at the Information level. To learn more, see Dependencies . Exceptions are logged at the Error level. The runtime also creates Warning level logs, such as when queue messages are sent to the poison queue . |
| Function.<YOUR_FUNCTION_NAME> | customMetrics customEvents | C# and JavaScript SDKs let you collect custom metrics and log custom events. To learn more, see Custom telemetry data . |
| Function.<YOUR_FUNCTION_NAME> | traces | Includes function started and completed logs for specific function runs. For successful runs, these logs are at the Information level. Exceptions are logged at the Error level. The runtime also creates Warning level logs, such as when queue messages are sent to the poison queue . |
| Function.<YOUR_FUNCTION_NAME>.User | traces | User-generated logs, which can be any log level. To learn more about writing to logs from your functions, see Writing to logs . |
| Host.Aggregator | customMetrics | 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 Information level. If you filter at Warning or above, you won't see any of this data. |
| Host.Results | requests | These runtime-generated logs indicate success or failure of a function. All of these logs are written at Information level. If you filter at Warning or above, you won't see any of this data. |

| CATEGORY | TABLE | DESCRIPTION |
|-----------|--------|---|
| Microsoft | traces | Fully-qualified log category that reflects a .NET runtime component invoked by the host. |
| Worker | traces | Logs generated by the language worker process for non-.NET languages. Language worker logs may also be logged in a Microsoft.* category, such as Microsoft.Azure.WebJobs.Script.Workers.Rpc.RpcFunction. These logs are written at Information level. |

NOTE

For .NET class library functions, these categories assume you are using `ILogger` and not `ILogger<T>`. To learn more, 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:

| LOGLEVEL | CODE | DESCRIPTION |
|-------------|------|--|
| Trace | 0 | Logs that contain the most detailed messages. These messages may 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.

For each category, you indicate the minimum log level to send. The host.json settings vary depending on the [Functions runtime version](#).

The example below defines logging based on the following rules:

- For logs of `Host.Results` or `Function`, only log events at `Error` or a higher level.
 - For logs of `Host.Aggregator`, log all generated metrics (`Trace`).
 - For all other logs, including user logs, log only `Information` level and higher events.
- v2.x+**
- v1.x**

```
{
  "logging": {
    "fileLoggingMode": "always",
    "logLevel": {
      "default": "Information",
      "Host.Results": "Error",
      "Function": "Error",
      "Host.Aggregator": "Trace"
    }
  }
}
```

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+](#)
- [v1.x](#)

```
{
  "logging": {
    "fileLoggingMode": "always",
    "logLevel": {
      "default": "Information",
      "Host": "Error",
      "Function": "Error",
      "Host.Aggregator": "Information"
    }
  }
}
```

You can use a log level setting of `None` prevent any logs from being written for a category.

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. Here's an example:

```
{
  "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+](#)
- [v1.x](#)

```
{
  "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. This makes sure that *all* function executions (requests) and exceptions are logged while other types of telemetry remain subject to sampling.

To learn more, see [Sampling in Application Insights](#).

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, you add an application setting named `SCALE_CONTROLLER_LOGGING_ENABLED` to your function app settings. The value of this setting must be of the format `<DESTINATION>:<VERBOSITY>`, based on the following:

| | |
|----------------------------------|---|
| <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> , make sure 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, as well as 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:

```
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`:

```
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:

```
az functionapp config appsettings delete --name <FUNCTION_APP_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--setting-names SCALE_CONTROLLER_LOGGING_ENABLED
```

Enable Application Insights integration

For a function app to send data to Application Insights, it needs to know the instrumentation key of an Application Insights resource. The key must be in an app setting named `APPINSIGHTS_INSTRUMENTATIONKEY`.

When you create your function app [in the Azure portal](#), from the command line by using [Azure Functions Core Tools](#), or by using [Visual Studio Code](#), Application Insights integration is enabled by default. The Application Insights resource has the same name as your function app, and it's created either in the same region or in the nearest region.

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 choose a different **Location** in an [Azure geography](#) where you want to store your data.

| Function App Create <p>* App name functionapp0921 .azurewebsites.net</p> <p>* Subscription Visual Studio Enterprise</p> <p>* Resource Group ? <input checked="" type="radio"/> Create new <input type="radio"/> Use existing functionapp0921</p> <p>* OS <input checked="" type="radio"/> Windows <input type="radio"/> Linux</p> <p>* Hosting Plan ? Consumption Plan</p> <p>* Location West Europe</p> <p>* Runtime Stack .NET</p> <p>* Storage ? <input checked="" type="radio"/> Create new <input type="radio"/> Use existing functionapp09218f22</p> <p>Application Insights ></p> | <h3>functionapp0921 - Application Insights</h3> <p>App Service</p> <p>Application Insights site extensions</p> <p>Collect application monitoring data using Application Insights site extension</p> <p>Enable Disable ?</p> <p>Link to an Application Insights resource</p> <p>Your app will be connected to an auto-created Application Insights resource: functionapp0921</p> <p>Instrumentation key will be added to App Settings. This will overwrite any instrumentation key value in web app configuration files.</p> <p>Change your resource</p> <p><input checked="" type="radio"/> Create new resource * New resource name functionapp0921 * Location West Europe</p> <p><input type="radio"/> Select existing resource <input type="checkbox"/> Search to find more resources</p> <p>Top 5 relevant resources - Relevance is determined by resource group, location, or in alphabetical order.</p> <table border="1"> <thead> <tr> <th>Name</th> <th>Resource Group</th> <th>Location</th> </tr> </thead> <tbody> <tr> <td>functions-ggailey777-vs</td> <td>functions-ggailey777-vs</td> <td>West Europe</td> </tr> </tbody> </table> <p>Apply</p> | Name | Resource Group | Location | functions-ggailey777-vs | functions-ggailey777-vs | West Europe |
|---|---|-------------|----------------|----------|-------------------------|-------------------------|-------------|
| Name | Resource Group | Location | | | | | |
| functions-ggailey777-vs | functions-ggailey777-vs | West Europe | | | | | |

When you choose **Create**, an Application Insights resource is created with your function app, which has the `APPINSIGHTS_INSTRUMENTATIONKEY` set in application settings. Everything is ready to go.

Add to an existing function app

If an Application Insights resources wasn't created with your function app, use the following steps to create the resource. You can then add the instrumentation key from that resource as an [application setting](#) in your function app.

1. In the [Azure portal](#), search for and select **function app**, and then choose 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.

functions-ggailey7
App Service

Search (Ctrl+ /) <> Browse Refresh Start Restart Swap Get publish profile

Overview

Application Insights is not configured. Configure Application Insights to capture function logs.

Resource group (change) : myResourceGroup

Status : Stopped

Location : Central US

Subscription (change) : Subscription

Subscription ID : 11111111-1111-1111-1111-111111111111

Tags (change) : Click here to add tags

Metrics Features (8) Notifications (0) Quickstart

3. Expand **Change your resource** and create an Application Insights resource by using the settings specified in the following 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 one that's close to that region. |

Application Insights

Collect application monitoring data using Application Insights

[Enable](#) [Disable](#) ⓘ [Feedback](#)

Link to an Application Insights resource

^ Change your resource

Create new resource

| | |
|----------------------|-------------|
| New resource name * | Location * |
| functions-ggailey777 | West Europe |

Select existing resource

[Search to find more resources](#)

Top 5 relevant resources - Relevance is determined by resource group, location, or in alphabetical order.

| Name | Resource Group | Location |
|-----------------------|----------------|----------|
| Apply | | |

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, select **Configuration** under **Settings**, and then select **Application settings**. If you see a setting named `APPINSIGHTS_INSTRUMENTATIONKEY`, Application Insights integration is enabled for your function app running in Azure. If for some reason this setting doesn't exist, add it using your Application Insights instrumentation key as the value.

NOTE

Early versions of Functions used built-in monitoring, which is no longer recommended. When enabling Application Insights integration for such a function app, you must also [disable built-in logging](#).

Disable built-in logging

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 built-in logging is used 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 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, make sure no existing functions in the same function app use the setting for Azure Storage triggers or bindings.

Next steps

To learn more about monitoring, see:

- [Monitor Azure Functions](#)
- [Analyze Azure Functions telemetry data in Application Insights](#)
- [Application Insights](#)

Analyze Azure Functions telemetry in Application Insights

11/2/2020 • 5 minutes to read • [Edit Online](#)

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 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.

The screenshot shows the Azure Functions Monitor tab for the 'HttpTrigger1' function in the 'myfunctionapp' function app. The left sidebar includes tabs for Overview, Developer, Code / Test, Integration, and Monitor, with 'Monitor' selected and highlighted with a red box. The top navigation bar includes a search bar, a Refresh button (also highlighted with a red box), and a 'Live app metrics' link. The main area displays summary metrics: Application Insights Instance (myfunctionapp), Success count in last 30 days (16), Error count in last 30 days (0), and a note that the query returned 16 items with a 'Run in Application Insights' link. Below this is a table of function invocation logs:

| 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.

The screenshot shows the Azure portal interface for Application Insights. On the left, there's a summary card for 'FunctionMonitoringExamples' with a success count of 6 over the last 30 days. Below it is a table of log entries from February 4, 2020, showing various successful requests. On the right, the 'Invocation Details' panel displays a list of log entries for the same period, with a red box highlighting the 'Run in Application Insights' button.

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 'Analytics' blade in the Application Insights portal. It displays a query results table for 'functions-ggailey777'. The table has columns: timestamp [UTC], id, operation_Name, success, resultCode, duration, and operation_Id. The data shows completed requests for 'HttpTriggerCSharp1' over the last 30 days, with a total of 20 records. The table includes sorting and filtering options.

| 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 | rFjoQUPKPePU=.7fc2b67c... | HttpTriggerCSharp1 | True | 0 | 2.1411 | rFjoQUPKPePU=... |
| 2019-04-08T07:15:05.407 | uclluRmA950=.7fc2b675... | HttpTriggerCSharp1 | True | 0 | 10.5045 | uclluRmA950=... |
| 2019-04-08T07:05:19.357 | IYzrQGz0TOY=.7fc2b558... | HttpTriggerCSharp1 | True | 0 | 8.0953 | IYzrQGz0TOY=... |
| 2019-04-08T07:04:46.803 | 49UmDBycab0=.7fc2b5... | HttpTriggerCSharp1 | True | 0 | 8.1741 | 49UmDBycab0=... |
| 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 | 9ozxDVdljYc=.5220de23... | HttpTriggerCSharp1 | True | 0 | 2.0653 | 9ozxDVdljYc=... |
| 2019-04-08T06:53:19.047 | 85i2g+dRHt4=.5220dd... | HttpTriggerCSharp1 | True | 0 | 1.8892 | 85i2g+dRHt4=... |
| 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 'myfunctionapp | Application Insights' blade in the Azure portal. On the left, there's a sidebar with 'Functions', 'Deployment', 'Settings', and 'Identity' sections. Under 'Settings', the 'Application Insights' option is highlighted with a red box. At the top right, there's a message: 'Application Insights is linked through Instrumentation Key in app settings' with a checkmark icon. Below it is a blue button labeled 'Turn on Application Insights'.

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 screenshot shows the 'functions-ggailey777' Application Insights dashboard. The left sidebar includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Investigate' (with sub-options like Application map, Smart Detection, Live Metrics Stream, Metrics, Search, Availability, Failures, Performance, Servers, Browser, Troubleshooting guides (pre...), Workbooks), 'Usage' (with sub-options like Users, Sessions, Events, Funnels), and 'Analytics'. The main area has a header with 'Application Dashboard', 'Getting started', 'Search', 'Analytics', 'Monitor resource group', and 'More'. It features four charts: 'Failed requests' (a bar chart showing 6 failed requests at 10:45 AM), 'Server response time' (a line chart showing an average of 198.58 ms), 'Server requests' (a line chart showing 37 server requests between 10:45 AM and 11 AM), and 'Availability' (a line chart showing 100% availability).

The following areas of Application Insights can be helpful when evaluating the behavior, performance, and errors in

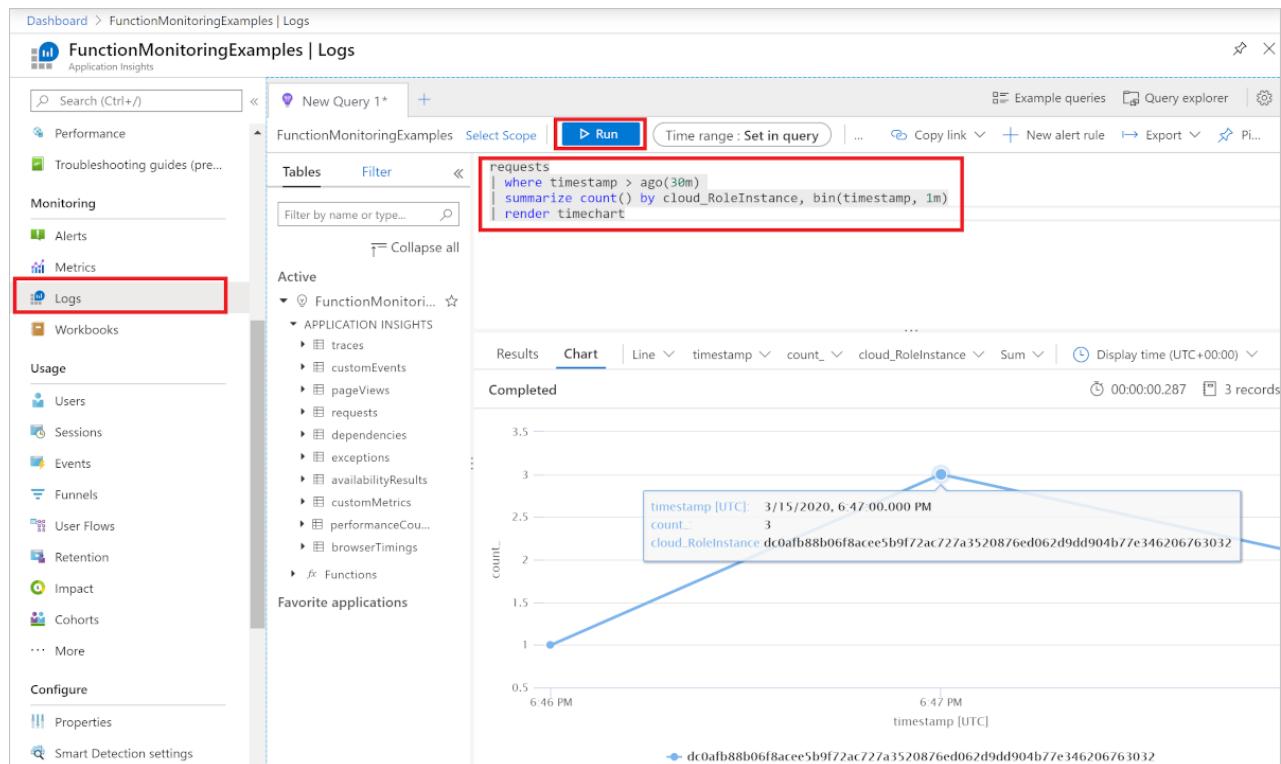
your functions:

| 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.



Here's a query example that shows the distribution of requests per worker over the last 30 minutes.

```
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 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. |
| 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`.

```
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.

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 |

| TIMESTAMP [UTC] | NAME | VALUE |
|---------------------------|---------------|-------------|
| 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

12/4/2020 • 3 minutes to read • [Edit Online](#)

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 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.

Portal

You can view both types of log streams in the portal.

Built-in log streaming

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 Azure portal interface with the 'Platform features' tab selected. The 'Log streaming' link under the 'Monitoring' category is highlighted with a red box.

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** (highlighted with a red box)
- Process explorer
- Metrics

API

- API definition
- CORS

App Service plan

- App Service plan
- Scale up
- Scale out
- Quotas

Resource management

- Diagnose and solve problems
- Activity log
- Access control (IAM)
- Tags
- Locks
- Automation script

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**.

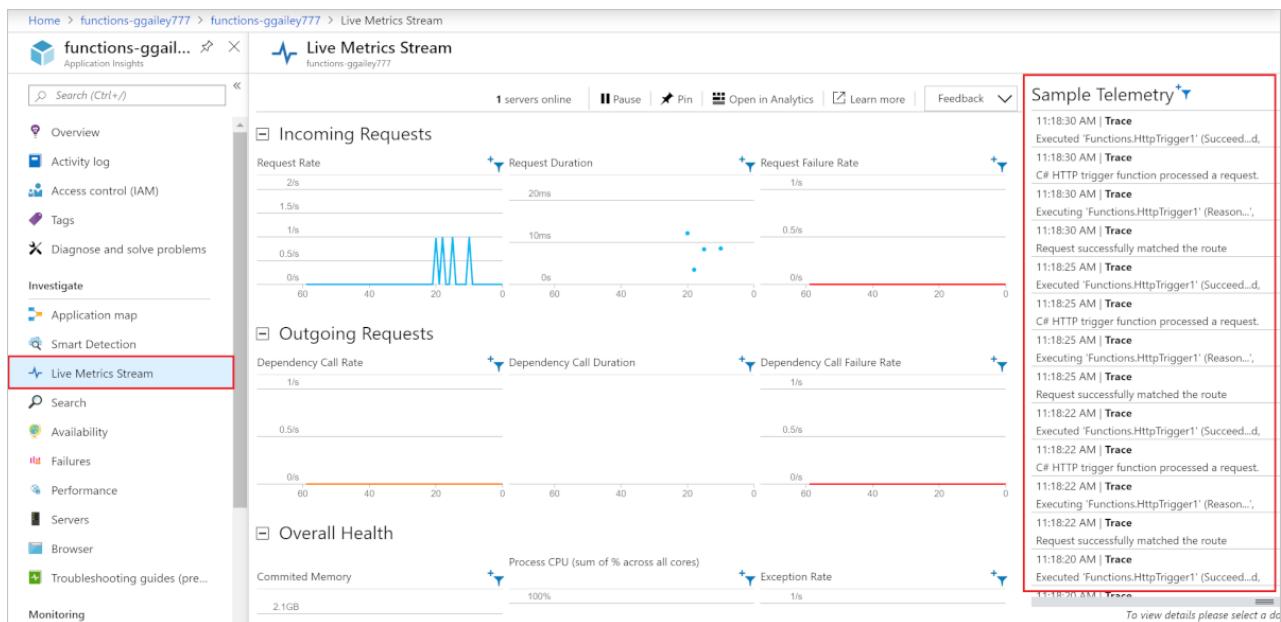
The screenshot shows the Azure Functions Log streaming interface. At the top, there are tabs for 'Overview', 'Platform features', and 'Log streaming'. Below the tabs are buttons for 'Reconnect', 'Copy logs', 'Pause', and 'Clear'. A blue bar highlights the 'Application logs' tab. The main area displays a log stream with the following entries:

```
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'
```

Live Metrics Stream

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**.



Visual Studio Code

To turn on the streaming logs for your function app in Azure:

1. Select F1 to open the command palette, and then search for and run the command **Azure Functions: Start Streaming Logs**.
2. Select your function app in Azure, and then select **Yes** to enable application logging for the function app.
3. Trigger your functions in Azure. Notice that log data is displayed in the Output window in Visual Studio Code.
4. When you're done, remember to run the command **Azure Functions: Stop Streaming Logs** to disable logging for the function app.

Core Tools

Built-in log streaming

Use the `logstream` option to start receiving streaming logs of a specific function app running in Azure, as in the following example:

```
func azure functionapp logstream <FunctionAppName>
```

NOTE

Built-in log streaming isn't yet enabled in Core Tools for function apps running on Linux in a Consumption plan. For these hosting plans, you instead need to use Live Metrics Stream to view the logs in near-real time.

Live Metrics Stream

You can view the **Live Metrics Stream** for your function app in a new browser window by including the `--browser` option, as in the following example:

```
func azure functionapp logstream <FunctionAppName> --browser
```

Azure CLI

You can enable streaming logs by using the [Azure CLI](#). Use the following commands to sign in, choose your subscription, and stream log files:

```
az login  
az account list  
az account set --subscription <subscriptionNameOrId>  
az webapp log tail --resource-group <RESOURCE_GROUP_NAME> --name <FUNCTION_APP_NAME>
```

Azure PowerShell

You can enable streaming logs by using [Azure PowerShell](#). For PowerShell, use the [Set-AzWebApp](#) command to enable logging on the function app, as shown in the following snippet:

```
# Enable Logs  
Set-AzWebApp -RequestTracingEnabled $True -HttpLoggingEnabled $True -DetailedErrorLoggingEnabled $True -  
ResourceGroupName $ResourceGroupName -Name $AppName
```

For more information, see the [complete code example](#).

Next steps

- [Monitor Azure Functions](#)
- [Analyze Azure Functions telemetry in Application Insights](#)

Monitoring Azure Functions with Azure Monitor Logs

12/4/2020 • 2 minutes to read • [Edit Online](#)

Azure Functions offers an integration with [Azure Monitor Logs](#) to monitor functions. This article shows you how to configure Azure Functions to send system-generated and user-generated logs to Azure Monitor Logs.

Azure Monitor Logs gives you the ability to consolidate logs from different resources in the same workspace, where it can be analyzed with [queries](#) to quickly retrieve, consolidate, and analyze collected data. You can create and test queries using [Log Analytics](#) in the Azure portal and then either directly analyze the data using these tools or save queries for use with [visualizations](#) or [alert rules](#).

Azure Monitor uses a version of the [Kusto query language](#) used by Azure Data Explorer that is suitable for simple log queries but also includes advanced functionality such as aggregations, joins, and smart analytics. You can quickly learn the query language using [multiple lessons](#).

NOTE

Integration with Azure Monitor Logs is currently in public preview for v2 and v3 function apps running on Windows Consumption, Premium, and Dedicated hosting plans.

Setting up

- From the **Monitoring** section of your function app in the [Azure portal](#), select **Diagnostic settings**, and then select **Add diagnostic setting**.

| Name | Storage account | Event hub | Log Analytics workspace | Edit setting |
|--------------------------------|-----------------|-----------|-------------------------|--------------|
| No diagnostic settings defined | | | | |

- In the **Diagnostics settings** page, under **Category details and log**, choose **FunctionAppLogs**.

The **FunctionAppLogs** table contains the desired logs.

- Under **Destination details**, choose **Send to Log Analytics** and then select your **Log Analytics workspace**.
- Enter a **Diagnostic settings name**, and then select **Save**.

Diagnostics settings

X

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

User-generated logs

To generate custom logs, use the logging statement specific to your language. Here are sample code snippets:

- [C#](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

```
log.LogInformation("My app logs here.");
```

Querying the logs

To query the generated logs:

1. From your function app, select **Diagnostic settings**.
2. From the **Diagnostic settings** list, select the Log Analytics workspace that you configured to send the function logs to.
3. From the **Log Analytics workspace** page, select **Logs**.

Azure Functions writes all logs to the **FunctionAppLogs** table under **LogManagement**.

The screenshot shows the Azure Log Analytics workspace interface. On the left, there's a navigation sidebar 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), and Logs (which 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, there's a 'New Query 1' button, a 'Run' button, and a time range selector set to 'Last 24 hours'. A message bar says 'Type your query here or click one of the example queries to start'. To the right of the search bar are buttons for 'Example queries', 'Query explorer', and 'Prett...'. The central part of the screen shows a 'Queries History' section with a placeholder message: 'You have no recent queries. Start querying using example queries or type your own queries in the query editor.' On the far right, there's a 'Learn more' section with links to 'Getting started', 'Online course', 'Language reference', 'Community', and 'What's new'.

Here are some sample queries:

All logs

```
FunctionAppLogs  
| order by TimeGenerated desc
```

Specific function logs

```
FunctionAppLogs  
| where FunctionName == "<Function name>"
```

Exceptions

```
FunctionAppLogs  
| where ExceptionDetails != ""  
| order by TimeGenerated asc
```

Next steps

- Review the [Azure Functions overview](#).
- Learn more about [Azure Monitor Logs](#).
- Learn more about the [query language](#).

Add a TLS/SSL certificate in Azure App Service

12/4/2020 • 16 minutes to read • [Edit Online](#)

Azure App Service provides a highly scalable, self-patching web hosting service. This article shows you how to create, upload, or import a private certificate or a public certificate into App Service.

Once the certificate is added to your App Service app or [function app](#), you can [secure a custom DNS name with it](#) or [use it in your application code](#).

The following table lists the options you have for adding certificates in App Service:

| OPTION | DESCRIPTION |
|---|--|
| Create a free App Service Managed Certificate (Preview) | A private certificate that's easy to use if you just need to secure your www custom domain or any non-naked domain in App Service. |
| Purchase 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 . |
| Upload a public certificate | Public certificates are not used to secure custom domains, but you can load them into your code if you need them to access remote resources. |

Prerequisites

To follow this how-to guide:

- [Create an App Service app](#).
- Free certificate only: map a subdomain (for example, [www.contoso.com](#)) to App Service with a [CNAME record](#).

Private certificate requirements

NOTE

Azure Web Apps does **not** support AES256 and all pfx files should be encrypted with TripleDES.

The [free App Service Managed Certificate](#) or 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](#)
- Contains private key at least 2048 bits long

- Contains all intermediate certificates in the certificate chain

To secure a custom domain in a TLS binding, the certificate has 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 can work with App Service but are not covered by this article. Work with your certificate authority on the exact steps to create ECC certificates.

Prepare your web app

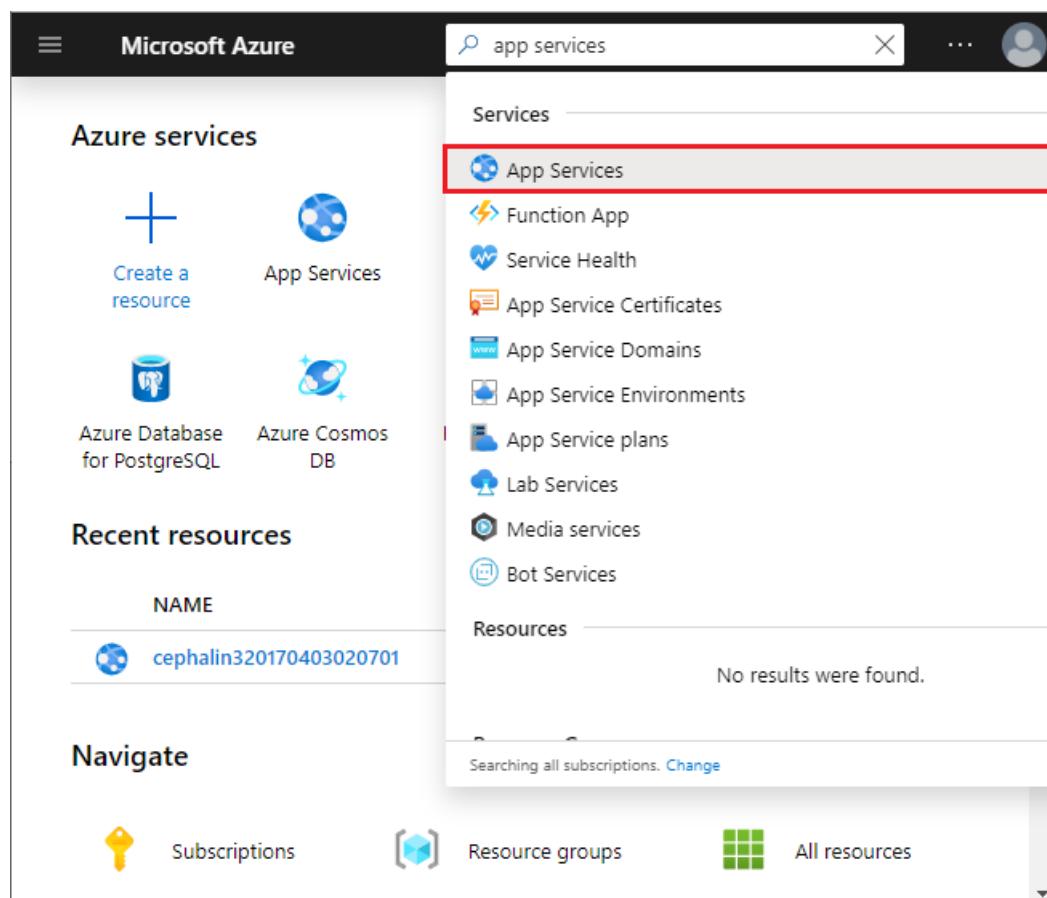
To create custom security bindings or enable client certificates for your App Service app, your [App Service plan](#) must be in the **Basic**, **Standard**, **Premium**, or **Isolated** tier. In this step, you make sure that your web app is in the supported pricing tier.

Sign in to Azure

Open the [Azure portal](#).

Navigate to your web app

Search for and select **App Services**.



The screenshot shows the Microsoft Azure portal interface. At the top, there's a navigation bar with a menu icon, the text "Microsoft Azure", a search bar containing "app services", and a user profile icon. Below the search bar, the main content area is titled "Azure services". On the left, there's a sidebar with icons for "Create a resource" (plus sign), "App Services" (blue globe), "Azure Database for PostgreSQL" (blue cylinder), and "Azure Cosmos DB" (blue globe with arrows). To the right of the sidebar, under the heading "Services", a list of items is shown, each with an icon and text: "App Services" (highlighted with a red box), "Function App", "Service Health", "App Service Certificates", "App Service Domains", "App Service Environments", "App Service plans", "Lab Services", "Media services", and "Bot Services". Below this list, under "Resources", it says "No results were found.". At the bottom of the page, there's a "Navigate" section with links for "Subscriptions", "Resource groups", and "All resources".

On the App Services page, select the name of your web app.

Home > App Services

App Services

Microsoft

Add Edit columns Refresh Assign tags Start Restart More

Subscriptions: All 2 selected – Don't see a subscription? Open Directory + Subscription settings

Filter by name All subsc... All resou... All locati... All tags No group...

6 items

| Name | Status | App Type | App Service |
|----------------------------|---------|----------|-------------|
| cephalin320170403020701 | Running | Web App | test-sku |
| denniseastusbot | Running | Web App | z76-z763 |
| myFirstAzureWebApp20190... | Running | Web App | ServicePl |
| WebApplicationASPDotNET... | Running | Web App | ServicePl |

You have landed on the management page of your web app.

Check the pricing tier

In the left-hand navigation of your web app page, scroll to the **Settings** section and select **Scale up (App Service plan)**.

cephalin320170403020701 App Service

Search (Ctrl+/)

Deployment Center

Settings

- Configuration
- Container settings
- Authentication / Authorization
- Application Insights
- Identity
- Backups
- Custom domains
- TLS/SSL settings
- Networking

Scale up (App Service plan)

Check to make sure that your web app is not in the F1 or D1 tier. Your web app's current tier is highlighted by a dark blue box.



Dev / Test

For less demanding workloads



Production

For most production workloads



Isolated

Advanced networking and scale

Recommended pricing tiers

F1

Shared infrastructure

1 GB memory

60 minutes/day compute

Free

D1

Shared infrastructure

1 GB memory

240 minutes/day compute

<price>/Month (Estimated)

B1

100 total ACU

1.75 GB memory

A-Series compute equivalent

<price>/Month (Estimated)

▼ See additional options

Included hardware

Every instance of your App Service plan will include the following hardware configuration:

Azure Compute Units (ACU)

Dedicated compute resources used to run applications deployed in the App...

Memory

Memory available to run applications

Custom SSL is not supported in the F1 or D1 tier. If you need to scale up, follow the steps in the next section.

Otherwise, close the Scale up page and skip the [Scale up your App Service plan](#) section.

Scale up your App Service plan

Select any of the non-free tiers (B1, B2, B3, or any tier in the Production category). For additional options, click [See additional options](#).

Click **Apply**.



Dev / Test

For less demanding workloads



Production

For most production workloads



Isolated

Advanced networking and scale

Recommended pricing tiers

F1

Shared infrastructure
1 GB memory
60 minutes/day compute
Free

D1

Shared infrastructure
1 GB memory
240 minutes/day compute
<price>/Month (Estimated)

B1

100 total ACU
1.75 GB memory
A-Series compute equivalent
<price>/Month (Estimated)

[▼ See additional options](#)

Included features

Every app hosted on this App Service plan will have access to these features:

Custom domains / SSL



Configure and purchase custom domains with SNI SSL bindings

Manual scale



Up to 3 instances. Subject to availability.

Included hardware

Every instance of your App Service plan will include the following hardware configuration:

Azure Compute Units (ACU)



Dedicated compute resources used to run applications deployed in the App...

Memory



Memory per instance available to run applications deployed and running in...

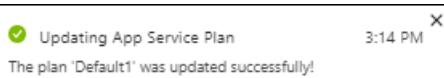
Storage



10 GB disk storage shared by all apps deployed in the App Service plan.

[Apply](#)

When you see the following notification, the scale operation is complete.



Create a free certificate (Preview)

The free App Service Managed Certificate is a turn-key solution for securing your custom DNS name in App Service. It's a fully functional TLS/SSL certificate that's managed by App Service and renewed automatically. The free certificate comes with the following limitations:

- Does not support wildcard certificates.

- Does not support naked domains.
- Is not exportable.
- Is not supported on App Service Environment (ASE)
- Does not support A records. For example, automatic renewal doesn't work with A records.

NOTE

The free certificate is issued by DigiCert. For some top-level domains, you must explicitly allow DigiCert as a certificate issuer by creating a [CAA domain record](#) with the value: `0 issue digicert.com`.

To create a free App Service Managed Certificate:

In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.

From the left navigation of your app, select **TLS/SSL settings > Private Key Certificates (.pfx) > Create App Service Managed Certificate**.

The screenshot shows the Azure portal interface for managing certificates. On the left, there's a sidebar with various settings like Quickstart, Deployment slots, and TLS/SSL settings (which is highlighted with a red box). The main content area has tabs for Bindings, Private Key Certificates (.pfx) (which is also highlighted with a red box), and Public Key Certificates (.cer). Below these tabs, there's a section for 'Private Key Certificate' with options to Import App Service Certificate, Upload Certificate, Import Key Vault Certificate, or Create App Service Managed Certificate (this last option is highlighted with a red box). At the bottom, there's a table for 'Private Key Certificates' with columns for Status Filter (All, Healthy, Warning, Expired), Health St..., Hostname, Expiration, and Thumbprint. A note states: "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. Uploaded certificates are not available for manual download from the Azure Management Portal, they can only be used by your app hosted on App Service after the required App Settings are set properly or used for TLS/SSL. [Learn more](#)".

Any non-naked domain that's properly mapped to your app with a CNAME record is listed in the dialog. Select the custom domain to create a free certificate for and select **Create**. You can create only one certificate for each supported custom domain.

When the operation completes, you see the certificate in the **Private Key Certificates** list.

| Private Key Certificates | | | |
|--|-----------------|------------|---------------------------------|
| Status Filter | | | |
| All | Healthy | Warning | Expired |
| Health Status | Hostname | Expiration | Thumbprint |
| ✓ Healthy | www.contoso.com | 4/11/2020 | 6A3BCCA5CC4B0158F0A097CE9F39... |
| ... | | | |

IMPORTANT

To secure a custom domain with this certificate, you still need to create a certificate binding. Follow the steps in [Create binding](#).

Import an App Service Certificate

If you purchase an App Service Certificate from Azure, Azure manages the following tasks:

- Takes care of the purchase process from GoDaddy.
- Performs domain verification of the certificate.
- Maintains the certificate in [Azure Key Vault](#).
- Manages certificate renewal (see [Renew certificate](#)).
- Synchronize the certificate automatically with the imported copies in App Service apps.

To purchase an App Service certificate, go to [Start certificate order](#).

If you already have a working App Service certificate, you can:

- [Import the certificate into App Service](#).
- [Manage the certificate](#), such as renew, rekey, and export it.

NOTE

App Service Certificates are not supported in Azure National Clouds at this time.

Start certificate order

Start an App Service certificate order in the [App Service Certificate create page](#).

The screenshot shows two side-by-side windows. The left window is titled 'App Service Certificate' and contains fields for 'Name' (ContosoCert), 'Naked Domain Host Name' (contosocertificate.com), 'Subscription' (dropdown), 'Resource Group' (Create new selected), and 'Certificate SKU' (Standard). The right window is titled 'Certificate Pricing' and shows two columns: 'S1 Standard' and 'W1 Wild Card'. Both columns list '1 Year', 'X.509 v3', 'RSA-SHA256', 'Auto Renew', and 'Improves SEO'. The 'S1 Standard' row has a price of \$69.99 USD/YEAR (ESTIMATED) and the 'W1 Wild Card' row has a price of \$299.99 USD/YEAR (ESTIMATED). A note at the bottom of the left window states: 'Create certificate operation may take 1-10 minutes to complete. Once created, App Service Certificates can only be used by other App Services within the same subscription.'

Use the following table to help you configure the certificate. When finished, click **Create**.

| SETTING | DESCRIPTION |
|---------|---|
| Name | A friendly name for your App Service certificate. |

| SETTING | DESCRIPTION |
|------------------------|--|
| Naked Domain Host Name | Specify the root domain here. The issued certificate secures <i>both</i> the root domain and the <code>www</code> subdomain. In the issued certificate, the Common Name field contains the root domain, and the Subject Alternative Name field contains the <code>www</code> domain. To secure any subdomain only, specify the fully qualified domain name of the subdomain here (for example, <code>mysubdomain.contoso.com</code>). |
| Subscription | The subscription that will contain the certificate. |
| Resource group | The resource group that will contain the certificate. You can use a new resource group or select the same resource group as your App Service app, for example. |
| Certificate SKU | Determines the type of certificate to create, whether a standard certificate or a wildcard certificate . |
| Legal Terms | Click to confirm that you agree with the legal terms. The certificates are obtained from GoDaddy. |

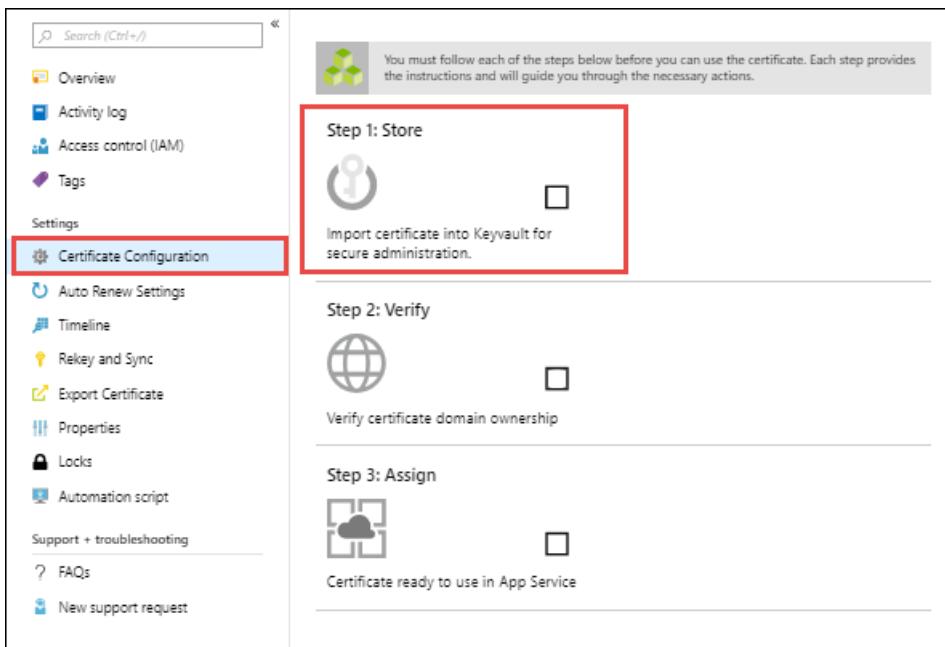
NOTE

App Service Certificates purchased from Azure are issued by GoDaddy. For some top-level domains, you must explicitly allow GoDaddy as a certificate issuer by creating a [CAA domain record](#) with the value: `0 issue godaddy.com`

Store in Azure Key Vault

Once the certificate purchase process is complete, there are few more steps you need to complete before you can start using this certificate.

Select the certificate in the [App Service Certificates](#) page, then click **Certificate Configuration > Step 1: Store**.



[Key Vault](#) is an Azure service that helps safeguard cryptographic keys and secrets used by cloud applications and services. It's the storage of choice for App Service certificates.

In the **Key Vault Status** page, click **Key Vault Repository** to create a new vault or choose an existing vault. If you

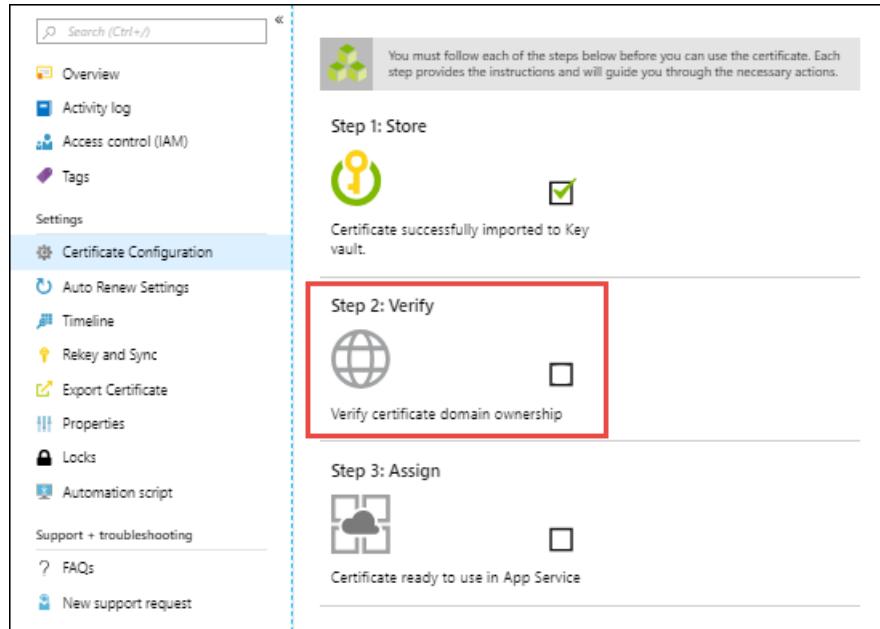
choose to create a new vault, use the following table to help you configure the vault and click Create. Create the new Key Vault inside the same subscription and resource group as your App Service app.

| SETTING | DESCRIPTION |
|------------------------|--|
| Name | A unique name that consists for alphanumeric characters and dashes. |
| Resource group | As a recommendation, select the same resource group as your App Service certificate. |
| Location | Select the same location as your App Service app. |
| Pricing tier | For information, see Azure Key Vault pricing details . |
| Access policies | Defines the applications and the allowed access to the vault resources. You can configure it later, following the steps at Assign a Key Vault access policy . |
| Virtual Network Access | Restrict vault access to certain Azure virtual networks. You can configure it later, following the steps at Configure Azure Key Vault Firewalls and Virtual Networks |

Once you've selected the vault, close the **Key Vault Repository** page. The **Step 1: Store** option should show a green check mark for success. Keep the page open for the next step.

Verify domain ownership

From the same **Certificate Configuration** page you used in the last step, click **Step 2: Verify**.



Select **App Service Verification**. Since you already mapped the domain to your web app (see [Prerequisites](#)), it's already verified. Just click **Verify** to finish this step. Click the Refresh button until the message **Certificate is Domain Verified** appears.

NOTE

Four types of domain verification methods are supported:

- **App Service** - The most convenient option when the domain is already mapped to an App Service app in the same subscription. It takes advantage of the fact that the App Service app has already verified the domain ownership.
- **Domain** - Verify an [App Service domain that you purchased from Azure](#). Azure automatically adds the verification TXT record for you and completes the process.
- **Mail** - Verify the domain by sending an email to the domain administrator. Instructions are provided when you select the option.
- **Manual** - Verify the domain using either an HTML page (**Standard** certificate only) or a DNS TXT record. Instructions are provided when you select the option.

Import certificate into App Service

In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.

From the left navigation of your app, select **TLS/SSL settings > Private Key Certificates (.pfx) > Import App Service Certificate**.

The screenshot shows the Azure portal interface for managing certificates. On the left, there's a sidebar with various service icons like Quickstart, Deployment slots, Deployment Center, Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, and TLS/SSL settings. The 'TLS/SSL settings' item is highlighted with a red box. The main content area has a header with 'Search (Ctrl+)', 'Refresh', 'Delete bindings', 'Buy Certificate', and 'FAQs'. Below the header, there are tabs for 'Bindings', 'Private Key Certificates (.pfx)' (which is selected and highlighted with a red box), and 'Public Key Certificates (.cer)'. A sub-header 'Private Key Certificate' is shown with a 'PFX' icon. A descriptive text block explains that private key certificates (.pfx) can be used for TLS/SSL bindings and loaded to the certificate store. It includes a 'Learn more' link and notes about manual download from the management portal. Below this are four action buttons: 'Import App Service Certificate' (with a plus sign icon), 'Upload Certificate' (with a plus sign icon), 'Import Key Vault Certificate' (with a plus sign icon), and 'Create App Service Managed Certificate' (with a plus sign icon). At the bottom, there's a 'Private Key Certificates' section with a status filter ('All', 'Healthy', 'Warning', 'Expired') and columns for 'Health St...', 'Hostname', 'Expiration', and 'Thumbprint'. A message states 'No private key certificates available for app.'

Select the certificate that you just purchased and select **OK**.

When the operation completes, you see the certificate in the **Private Key Certificates** list.

The screenshot shows the 'Private Key Certificates' list. At the top, there's a 'Status Filter' with 'All' selected. Below it is a table with columns for 'Health Status', 'Hostname', 'Expiration', and 'Thumbprint'. One row is visible, showing a green checkmark in the 'Health Status' column, the hostname 'contoso.com, www.contoso.com' in the 'Hostname' column, the date '10/10/2020' in the 'Expiration' column, and a long thumbprint value in the 'Thumbprint' column. There are also three dots at the end of the row.

IMPORTANT

To secure a custom domain with this certificate, you still need to create a certificate binding. Follow the steps in [Create binding](#).

Import a certificate from Key Vault

If you use Azure Key Vault to manage your certificates, you can import a PKCS12 certificate from Key Vault into App Service as long as it [satisfies the requirements](#).

Authorize App Service to read from the vault

By default, the App Service resource provider doesn't have access to the Key Vault. In order to use a Key Vault for a certificate deployment, you need to [authorize the resource provider read access to the KeyVault](#).

`abfa0a7c-a6b6-4736-8310-5855508787cd` is the resource provider service principal name for App Service, and it's the same for all Azure subscriptions. For Azure Government cloud environment, use

`6a02c803-daf3-4136-b4c3-5a6f318b4714` instead as the resource provider service principal name.

Import a certificate from your vault to your app

In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.

From the left navigation of your app, select **TLS/SSL settings > Private Key Certificates (.pfx) > Import Key Vault Certificate**.

The screenshot shows the Azure portal interface for managing certificates. On the left, there's a sidebar with various options like Quickstart, Deployment slots, Deployment Center, Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, and TLS/SSL settings. The 'TLS/SSL settings' option is highlighted with a red box. On the right, the main content area has a header with Refresh, Delete bindings, Buy Certificate, and FAQs buttons. Below the header, there are tabs for Bindings, Private Key Certificates (.pfx), and Public Key Certificates (.cer). The 'Private Key Certificates (.pfx)' tab is selected and highlighted with a red box. Underneath, there's a sub-header 'Private Key Certificate' with a PFX icon. A descriptive text explains that private key certificates (.pfx) can be used for TLS/SSL bindings and loaded to the certificate store. It includes links for 'Learn more' and 'Upload'. Below this, there are four buttons: 'Import App Service Certificate', 'Upload Certificate', 'Import Key Vault Certificate' (which is also highlighted with a red box), and 'Create App Service Managed Certificate'. At the bottom, there's a section titled 'Private Key Certificates' with a status filter showing 'All' (selected), Healthy, Warning, and Expired. There are columns for Health St..., Hostname, Expiration, and Thumbprint. A message at the bottom states 'No private key certificates available for app.'

Use the following table to help you select the certificate.

| SETTING | DESCRIPTION |
|--------------|---|
| Subscription | The subscription that the Key Vault belongs to. |
| Key Vault | The vault with the certificate you want to import. |
| Certificate | Select from the list of PKCS12 certificates in the vault. All PKCS12 certificates in the vault are listed with their thumbprints, but not all are supported in App Service. |

When the operation completes, you see the certificate in the **Private Key Certificates** list. If the import fails with an error, the certificate doesn't meet the [requirements for App Service](#).

The screenshot shows the 'Private Key Certificates' list. At the top, there's a 'Status Filter' with 'All' (selected), Healthy, Warning, and Expired buttons. Below that, there are columns for Health Status, Hostname, Expiration, and Thumbprint. A single row is shown, indicating a healthy certificate for contoso.com, www.contoso.com, with an expiration date of 10/10/2020 and a thumbprint of 6A3BCCA5CC4B0158F0A097CE9F39... . There are three dots at the end of the row.

NOTE

If you update your certificate in Key Vault with a new certificate, App Service automatically syncs your certificate within 48 hours.

IMPORTANT

To secure a custom domain with this certificate, you still need to create a certificate binding. Follow the steps in [Create binding](#).

Upload a private certificate

Once you obtain a certificate from your certificate provider, follow the steps in this section to make it ready for App Service.

Merge intermediate certificates

If your certificate authority gives you multiple certificates in the certificate chain, you need to merge the certificates in order.

To do this, open each certificate you received in a text editor.

Create a file for the merged certificate, called *mergedcertificate.crt*. In a text editor, copy the content of each certificate into this file. The order of your certificates should follow the order in the certificate chain, beginning with your certificate and ending with the root certificate. It looks like the following 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 certificate to PFX

Export your merged TLS/SSL certificate with the private key that your certificate request was generated with.

If you generated your certificate request using OpenSSL, then you have created a private key file. To export your certificate to PFX, run the following command. Replace the placeholders *<private-key-file>* and *<merged-certificate-file>* with the paths to your private key and your merged certificate file.

```
openssl pkcs12 -export -out myserver.pfx -inkey <private-key-file> -in <merged-certificate-file>
```

When prompted, define an export password. You'll use this password when uploading your TLS/SSL certificate to App Service later.

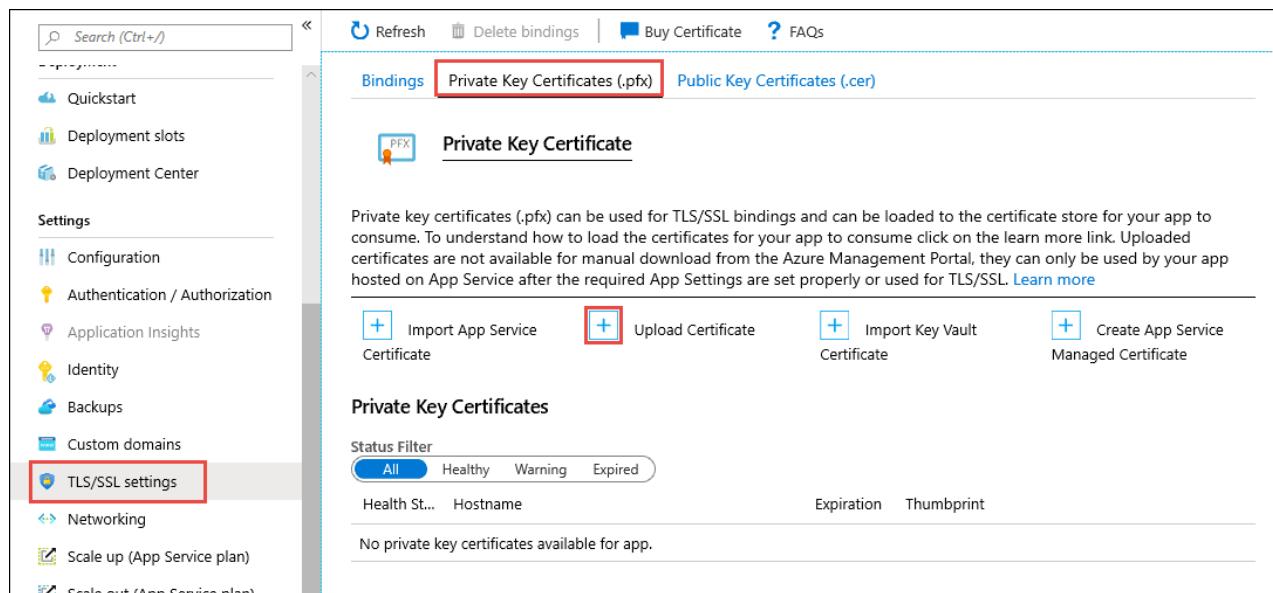
If you used IIS or *Certreq.exe* to generate your certificate request, install the certificate to your local machine, and then [export the certificate to PFX](#).

Upload certificate to App Service

You're now ready upload the certificate to App Service.

In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.

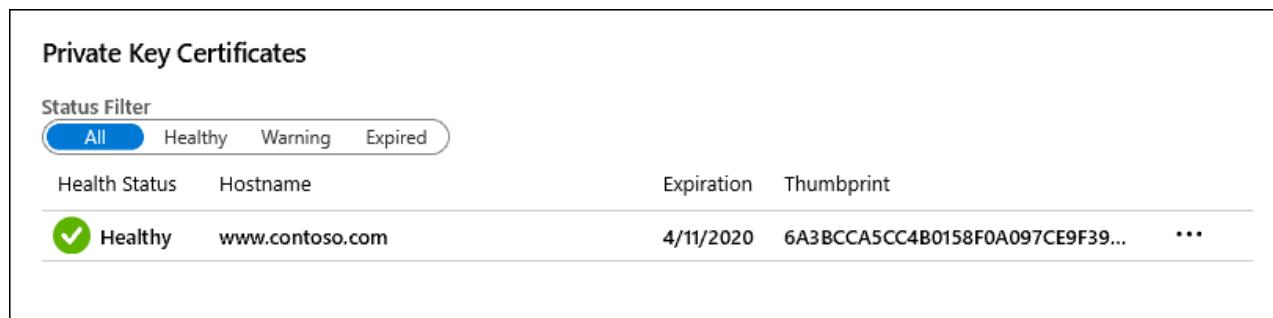
From the left navigation of your app, select **TLS/SSL settings > Private Key Certificates (.pfx) > Upload Certificate**.



The screenshot shows the Azure portal interface for managing certificates. On the left, there's a sidebar with various settings like Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, and TLS/SSL settings. The TLS/SSL settings option is highlighted with a red box. In the main content area, there are tabs for Bindings, Private Key Certificates (.pfx), and Public Key Certificates (.cer). The Private Key Certificates (.pfx) tab is also highlighted with a red box. Below it, there's a sub-section titled 'Private Key Certificate' with a 'PFX' icon. A note says: '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. Uploaded certificates are not available for manual download from the Azure Management Portal, they can only be used by your app hosted on App Service after the required App Settings are set properly or used for TLS/SSL.' There are four buttons below: 'Import App Service Certificate', 'Upload Certificate' (which is highlighted with a red box), 'Import Key Vault Certificate', and 'Create App Service Managed Certificate'. At the bottom, there's a table header for 'Private Key Certificates' with columns for Status Filter (All, Healthy, Warning, Expired), Health Status, Hostname, Expiration, and Thumbprint. A message says 'No private key certificates available for app.'

In **PFX Certificate File**, select your PFX file. In **Certificate password**, type the password that you created when you exported the PFX file. When finished, click **Upload**.

When the operation completes, you see the certificate in the **Private Key Certificates** list.



| Private Key Certificates | | | | |
|--------------------------|----------|-----------------|---|-----|
| Status Filter | | | | |
| All | Healthy | Warning | Expired | |
| Health Status | Hostname | Expiration | Thumbprint | |
| | Healthy | www.contoso.com | 4/11/2020 6A3BCCA5CC4B0158F0A097CE9F39... | ... |

IMPORTANT

To secure a custom domain with this certificate, you still need to create a certificate binding. Follow the steps in [Create binding](#).

Upload a public certificate

Public certificates are supported in the **.cer** format.

In the [Azure portal](#), from the left menu, select **App Services > <app-name>**.

From the left navigation of your app, click **TLS/SSL settings > Public Certificates (.cer) > Upload Public Key Certificate**.

In **Name**, type a name for the certificate. In **CER Certificate file**, select your CER file.

Click **Upload**.

Add Public Key Certificate (.cer)

 **Upload Public Key Certificate**

Upload a public key certificate (.cer) to be consumed in your app runtime. Note: Public key certificates cannot be used to configure TLS/SSL Bindings. [Learn more](#)

* Name
Contoso ✓

* CER Certificate file
"somecert.cer" 

Upload

Once the certificate is uploaded, copy the certificate thumbprint and see [Make the certificate accessible](#).

Manage App Service certificates

This section shows you how to manage an App Service certificate you purchased in [Import an App Service certificate](#).

- [Rekey certificate](#)
- [Renew certificate](#)
- [Export certificate](#)
- [Delete certificate](#)

Rekey certificate

If you think your certificate's private key is compromised, you can rekey your certificate. Select the certificate in the [App Service Certificates](#) page, then select **Rekey and Sync** from the left navigation.

Click **Rekey** to start the process. This process can take 1-10 minutes to complete.

The screenshot shows the Azure portal interface for managing certificates. On the left, there's a navigation menu with items like Overview, Activity log, Access control (IAM), Tags, Settings, Certificate Configuration, Auto Renew Settings, Timeline, Rekey and Sync (which is highlighted with a red box), Export Certificate, Properties, Locks, Automation script, Support + troubleshooting, FAQs, and New support request. At the top, there are buttons for Refresh, Rekey (which is highlighted with a red box), and Sync. The main content area is titled "Rekey Certificate" and contains a warning message: "Rekeying your certificate will roll the certificate with a new certificate issued from the certificate authority. While Rekeying your certificate will go through Pending Issuance state and once the certificate is ready make sure you sync your resources using this certificate to prevent disruption to service." Below this, there's a note: "Certificate rekey operations are free and rekey does not incur additional charges." A section titled "Linked Private Certificate" shows a table with columns STATUS, LINKED PRIVATE CERTIFI..., RESOURCE GROUP, and THUMBPRINT. One row is shown with a green checkmark under STATUS and the text "myResourceGro...".

Rekeying your certificate rolls the certificate with a new certificate issued from the certificate authority.

Once the rekey operation is complete, click **Sync**. The sync operation automatically updates the hostname bindings for the certificate in App Service without causing any downtime to your apps.

NOTE

If you don't click **Sync**, App Service automatically syncs your certificate within 48 hours.

Renew certificate

To turn on automatic renewal of your certificate at any time, select the certificate in the [App Service Certificates](#) page, then click **Auto Renew Settings** in the left navigation. By default, App Service Certificates have a one-year validity period.

Select **On** and click **Save**. Certificates can start automatically renewing 60 days before expiration if you have automatic renewal turned on.

The screenshot shows the Azure portal interface for managing certificates. On the left, there's a navigation menu with items like Overview, Activity log, Access control (IAM), Tags, Settings, Certificate Configuration, Auto Renew Settings (which is highlighted with a red box), Timeline, Rekey and Sync, Export Certificate, Properties, Locks, Automation script, Support + troubleshooting, FAQs, and Help. At the top, there are buttons for Refresh, Save (which is highlighted with a red box), Discard, Manual Renew, and Sync. The main content area is titled "Manual renewal not allowed at this time" and contains a warning message: "App Service Certificate is not eligible for manual renewal right now. Manual renewal will become available 60 days before expiry to prevent accidental renewal. Use Manual Renew ONLY if you really need to get a new certificate issued with extended expiry. For rolling keys use the Rekey feature. Turn off the auto renew feature to opt out of automatic auto renewal." Below this, there's a section titled "Linked Private Certificate" with a table showing a single row where the "Auto Renew App Service Certificate" switch is set to "On".

To manually renew the certificate instead, click **Manual Renew**. You can request to manually renew your certificate 60 days before expiration.

Once the renew operation is complete, click **Sync**. The sync operation automatically updates the hostname

bindings for the certificate in App Service without causing any downtime to your apps.

NOTE

If you don't click **Sync**, App Service automatically syncs your certificate within 48 hours.

Export certificate

Because an App Service Certificate is a [Key Vault secret](#), you can export a PFX copy of it and use it for other Azure services or outside of Azure.

To export the App Service Certificate as a PFX file, run the following commands in the [Cloud Shell](#). You can also run it locally if you [installed Azure CLI](#). Replace the placeholders with the names you used when you [created the App Service certificate](#).

```
secretname=$(az resource show \
    --resource-group <group-name> \
    --resource-type "Microsoft.CertificateRegistration/certificateOrders" \
    --name <app-service-cert-name> \
    --query "properties.certificates.<app-service-cert-name>.keyVaultSecretName" \
    --output tsv)

az keyvault secret download \
    --file appservicecertificate.pfx \
    --vault-name <key-vault-name> \
    --name $secretname \
    --encoding base64
```

The downloaded *appservicecertificate.pfx* file is a raw PKCS12 file that contains both the public and private certificates. In each prompt, use an empty string for the import password and the PEM pass phrase.

Delete certificate

Deletion of an App Service certificate is final and irreversible. Deletion of a App Service Certificate resource results in the certificate being revoked. Any binding in App Service with this certificate becomes invalid. To prevent accidental deletion, Azure puts a lock on the certificate. To delete an App Service certificate, you must first remove the delete lock on the certificate.

Select the certificate in the [App Service Certificates](#) page, then select **Locks** in the left navigation.

Find the lock on your certificate with the lock type **Delete**. To the right of it, select **Delete**.

| Lock name | Lock type | Scope | Notes |
|-----------|-----------|-------|--|
| contoso | Delete | | Deleting this App Service certi... Edit Delete |

Now you can delete the App Service certificate. From the left navigation, select **Overview > Delete**. In the confirmation dialog, type the certificate name and select **OK**.

Automate with scripts

Azure CLI

```

#!/bin/bash

fqdn=<replace-with-www.{yourdomain}>
pfxPath=<replace-with-path-to-your-.PFX-file>
pfxPassword=<replace-with-your=.PFX-password>
resourceGroup=myResourceGroup
webappname=mywebapp$RANDOM

# Create a resource group.
az group create --location westeurope --name $resourceGroup

# Create an App Service plan in Basic tier (minimum required by custom domains).
az appservice plan create --name $webappname --resource-group $resourceGroup --sku B1

# Create a web app.
az webapp create --name $webappname --resource-group $resourceGroup \
--plan $webappname

echo "Configure a CNAME record that maps $fqdn to $webappname.azurewebsites.net"
read -p "Press [Enter] key when ready ..."

# Before continuing, go to your DNS configuration UI for your custom domain and follow the
# instructions at https://aka.ms/appservicecustomdns to configure a CNAME record for the
# hostname "www" and point it to your web app's default domain name.

# Map your prepared custom domain name to the web app.
az webapp config hostname add --webapp-name $webappname --resource-group $resourceGroup \
--hostname $fqdn

# Upload the SSL certificate and get the thumbprint.
thumbprint=$(az webapp config ssl upload --certificate-file $pfxPath \
--certificate-password $pfxPassword --name $webappname --resource-group $resourceGroup \
--query thumbprint --output tsv)

# Binds the uploaded SSL certificate to the web app.
az webapp config ssl bind --certificate-thumbprint $thumbprint --ssl-type SNI \
--name $webappname --resource-group $resourceGroup

echo "You can now browse to https://$fqdn"

```

PowerShell

```

$fqdn=<Replace with your custom domain name>
$pfxPath=<Replace with path to your .PFX file>
$pfxPassword=<Replace with your .PFX password>
$webappname="mywebapp$(Get-Random)"
$location="West Europe"

# Create a resource group.
New-AzResourceGroup -Name $webappname -Location $location

# Create an App Service plan in Free tier.
New-AzAppServicePlan -Name $webappname -Location $location ` 
-ResourceGroupName $webappname -Tier Free

# Create a web app.
New-AzWebApp -Name $webappname -Location $location -AppServicePlan $webappname ` 
-ResourceGroupName $webappname

Write-Host "Configure a CNAME record that maps $fqdn to $webappname.azurewebsites.net"
Read-Host "Press [Enter] key when ready ..."

# Before continuing, go to your DNS configuration UI for your custom domain and follow the
# instructions at https://aka.ms/appservicecustomdns to configure a CNAME record for the
# hostname "www" and point it to your web app's default domain name.

# Upgrade App Service plan to Basic tier (minimum required by custom SSL certificates)
Set-AzAppServicePlan -Name $webappname -ResourceGroupName $webappname ` 
-Tier Basic

# Add a custom domain name to the web app.
Set-AzWebApp -Name $webappname -ResourceGroupName $webappname ` 
-HostNames @($fqdn, "$webappname.azurewebsites.net")

# Upload and bind the SSL certificate to the web app.
New-AzWebAppSSLBinding -WebAppName $webappname -ResourceGroupName $webappname -Name $fqdn ` 
-CertificateFilePath $pfxPath -CertificatePassword $pfxPassword -SslState SniEnabled

```

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](#)

Configure your App Service or Azure Functions app to use Azure AD login

12/4/2020 • 8 minutes to read • [Edit Online](#)

This article shows you how to configure Azure App Service or Azure Functions to use Azure Active Directory (Azure AD) as an authentication provider.

NOTE

The express settings flow sets up an AAD V1 application registration. If you wish to use [Azure Active Directory v2.0](#) (including [MSAL](#)), please follow the [advanced configuration instructions](#).

Follow these best practices when setting up your app and authentication:

- Give each App Service app its own permissions and consent.
- Configure each App Service app with its own registration.
- Avoid permission sharing between environments by using separate app registrations for separate deployment slots. When testing new code, this practice can help prevent issues from affecting the production app.

NOTE

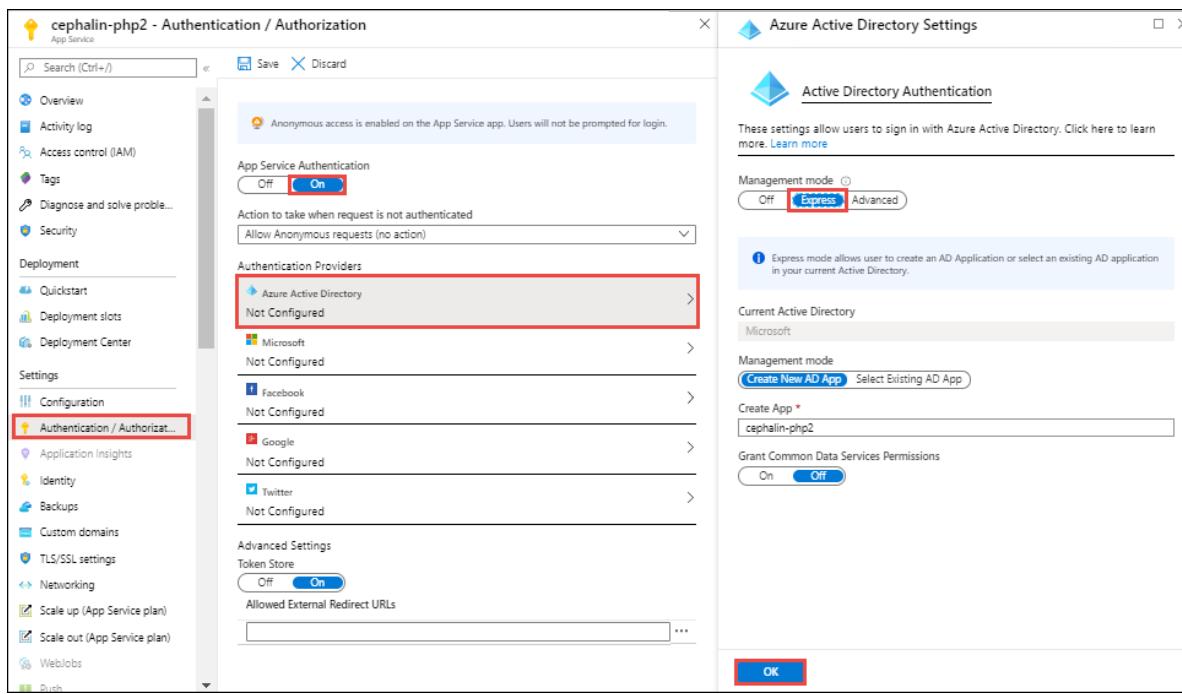
This feature is currently not available on Linux Consumption plan for Azure Functions

Configure with express settings

NOTE

The **Express** option is not available for government clouds.

1. In the [Azure portal](#), search for and select **App Services**, and then select your app.
2. From the left navigation, select **Authentication / Authorization > On**.
3. Select **Azure Active Directory > Express**.
If you want to choose an existing app registration instead:
 - a. Choose **Select Existing AD app**, then click **Azure AD App**.
 - b. Choose an existing app registration and click **OK**.
4. Select **OK** to register the App Service app in Azure Active Directory. A new app registration is created.



5. (Optional) By default, App Service provides authentication but doesn't restrict authorized access to your site content and APIs. You must authorize users in your app code. To restrict app access only to users authenticated by Azure Active Directory, set **Action to take when request is not authenticated** to **Log in with Azure Active Directory**. When you set this functionality, your app requires all requests to be authenticated. It also redirects all unauthenticated to Azure Active Directory for authentication.

Caution

Restricting access in this way applies to all calls to your app, which might not be desirable for apps that have a publicly available home page, as in many single-page applications. For such applications, **Allow anonymous requests (no action)** might be preferred, with the app manually starting login itself. For more information, see [Authentication flow](#).

6. Select **Save**.

Configure with advanced settings

You can configure app settings manually if you want to use an app registration from a different Azure AD tenant. To complete this custom configuration:

1. Create a registration in Azure AD.
2. Provide some of the registration details to App Service.

Create an app registration in Azure AD for your App Service app

You'll need the following information when you configure your App Service app:

- Client ID
- Tenant ID
- Client secret (optional)
- Application ID URI

Perform the following steps:

1. Sign in to the [Azure portal](#), search for and select **App Services**, and then select your app. Note your app's URL. You'll use it to configure your Azure Active Directory app registration.
2. Select **Azure Active Directory > App registrations > New registration**.
3. In the **Register an application** page, enter a **Name** for your app registration.

4. In **Redirect URI**, select **Web** and type <app-url>/.auth/login/aad/callback . For example,
https://contoso.azurewebsites.net/.auth/login/aad/callback .
5. Select **Create**.
6. After the app registration is created, copy the **Application (client) ID** and the **Directory (tenant) ID** for later.
7. Select **Authentication**. Under **Implicit grant**, enable **ID tokens** to allow OpenID Connect user sign-ins from App Service.
8. (Optional) Select **Branding**. In **Home page URL**, enter the URL of your App Service app and select **Save**.
9. Select **Expose an API > Set**. For single-tenant app, paste in the URL of your App Service app and select **Save** and for multi-tenant app, paste in the URL which is based on one of tenant verified domains and then select **Save**.

NOTE

This value is the **Application ID URI** of the app registration. If your web app requires access to an API in the cloud, you need the **Application ID URI** of the web app when you configure the cloud App Service resource. You can use this, for example, if you want the cloud service to explicitly grant access to the web app.

10. Select **Add a scope**.
 - a. In **Scope name**, enter *user_impersonation*.
 - b. In the text boxes, enter the consent scope name and description you want users to see on the consent page. For example, enter *Access my app*.
 - c. Select **Add scope**.
11. (Optional) To create a client secret, select **Certificates & secrets > New client secret > Add**. Copy the client secret value shown in the page. It won't be shown again.
12. (Optional) To add multiple **Reply URLs**, select **Authentication**.

Enable Azure Active Directory in your App Service app

1. In the [Azure portal](#), search for and select **App Services**, and then select your app.
2. In the left pane, under **Settings**, select **Authentication / Authorization > On**.
3. (Optional) By default, App Service authentication allows unauthenticated access to your app. To enforce user authentication, set **Action to take when request is not authenticated** to **Log in with Azure Active Directory**.
4. Under **Authentication Providers**, select **Azure Active Directory**.
5. In **Management mode**, select **Advanced** and configure App Service authentication according to the following table:

| FIELD | DESCRIPTION |
|-----------|---|
| Client ID | Use the Application (client) ID of the app registration. |

| FIELD | DESCRIPTION |
|--------------------------|---|
| 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 the Directory (tenant) ID in which the app registration was created. This value is used to redirect users to the correct Azure AD tenant, as well as to download the appropriate metadata to determine the appropriate token signing keys and token issuer claim value for example. For applications that use Azure AD v1 and for Azure Functions apps, omit /v2.0 in the URL. |
| Client Secret (Optional) | Use the client secret you generated in the app registration. |
| Allowed Token Audiences | If this is a cloud or server app and you want to allow authentication tokens from a web app, add the Application ID URI of the web app here. The configured Client ID is <i>always</i> implicitly considered to be an allowed audience. |

6. Select **OK**, and then select **Save**.

You're now ready to use Azure Active Directory for authentication in your App Service app.

Configure a native client application

You can register native clients to allow authentication to Web API's hosted in your app using a client library such as the [Active Directory Authentication Library](#).

1. In the [Azure portal](#), select **Active Directory > App registrations > New registration**.
2. In the **Register an application** page, enter a **Name** for your app registration.
3. 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`.

NOTE

For a Microsoft Store application, use the [package SID](#) as the URI instead.

4. Select **Create**.
5. After the app registration is created, copy the value of **Application (client) ID**.
6. Select **API permissions > Add a permission > My APIs**.
7. 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 [Create an app registration in Azure AD for your App Service app](#).
8. Under **Delegated permissions**, select **user_impersonation**, and then select **Add permissions**.

You have now configured a native client application that can access your App Service app on behalf of a user.

Configure a daemon client application for service-to-service calls

Your application can acquire a token to call a Web API hosted in your App Service or Function app on behalf of 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. In the [Azure portal](#), select **Active Directory > App registrations > New registration**.
2. In the **Register an application** page, enter a **Name** for your daemon app registration.
3. For a daemon application, you don't need a Redirect URI so you can keep that empty.
4. Select **Create**.
5. After the app registration is created, copy the value of **Application (client) ID**.
6. Select **Certificates & secrets > New client secret > Add**. Copy the client secret value shown in the page. It won't be shown again.

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 / Authorization 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 Azure AD 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 additional 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 click **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 is not performed by App Service Authentication / Authorization). 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.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)

Configure your App Service or Azure Functions app to use Facebook login

4/1/2020 • 2 minutes to read • [Edit Online](#)

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

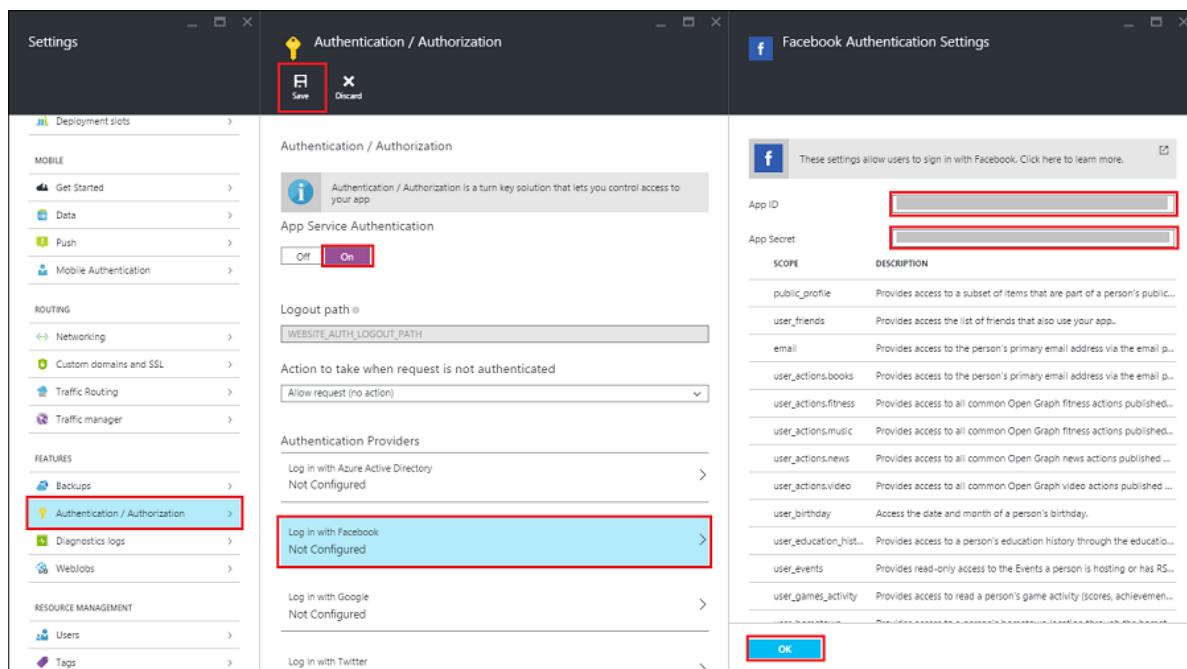
The app secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

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 Service app.
2. Select **Settings > Authentication / Authorization**, and make sure that **App Service Authentication** is **On**.
3. Select **Facebook**, and then paste in the App ID and App Secret values that you obtained previously. Enable any scopes needed by your application.
4. Select **OK**.



By default, App Service provides authentication, but it doesn't restrict authorized access to your site content and APIs. You need to authorize users in your app code.

5. (Optional) To restrict access only to users authenticated by Facebook, set **Action to take when request is not authenticated** to **Facebook**. When you set this functionality, your app requires all requests to be authenticated. It also redirects all unauthenticated requests to Facebook for authentication.

Caution

Restricting access in this way applies to all calls to your app, which might not be desirable for apps that have a publicly available home page, as in many single-page applications. For such applications, **Allow anonymous requests (no action)** might be preferred so that the app manually starts authentication itself. For more information, see [Authentication flow](#).

6. Select **Save**.

You're now ready to use Facebook for authentication in your app.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

Configure your App Service or Azure Functions app to use Google login

4/1/2020 • 2 minutes to read • [Edit Online](#)

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 [Google Sign-In for server-side apps](#) 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. In the [Azure portal](#), go to your App Service app.
2. Select **Settings > Authentication / Authorization**, and make sure that **App Service Authentication** is **On**.
3. Select **Google**, then paste in the App ID and App Secret values that you obtained previously. Enable any scopes needed by your application.
4. Select **OK**.

App Service provides authentication but doesn't restrict authorized access to your site content and APIs. For more information, see [Authorize or deny users](#).

5. (Optional) To restrict site access only to users authenticated by Google, set **Action to take when request is not authenticated to Google**. When you set this functionality, your app requires that all requests be authenticated. It also redirects all unauthenticated requests to Google for authentication.

Caution

Restricting access in this way applies to all calls to your app, which might not be desirable for apps that have a publicly available home page, as in many single-page applications. For such applications, **Allow anonymous requests (no action)** might be preferred so that the app manually starts authentication itself. For more information, see [Authentication flow](#).

6. Select **Save**.

You are now ready to use Google for authentication in your app.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

Configure your App Service or Azure Functions app to use Microsoft Account login

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows you how to configure Azure App Service or Azure Functions to use AAD to support personal Microsoft account logins.

NOTE

Both personal Microsoft accounts and organizational accounts use the AAD identity provider. At this time, it is not possible to configure this identity provider to support both types of log-ins.

Register your app with Microsoft Account

1. Go to [App registrations](#) in the Azure portal. If needed, sign in with your Microsoft account.
2. Select **New registration**, then enter an application name.
3. Under **Supported account types**, select **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**
4. In **Redirect URIs**, select **Web**, and then enter `https://<app-domain-name>/.auth/login/aad/callback`. Replace `<app-domain-name>` with the domain name of your app. For example, `https://contoso.azurewebsites.net/.auth/login/aad/callback`. Be sure to use the HTTPS scheme in the URL.
5. Select **Register**.
6. Copy the **Application (Client) ID**. You'll need it later.
7. From the left pane, select **Certificates & secrets > New client secret**. Enter a description, select the validity duration, and select **Add**.
8. Copy the value that appears on the **Certificates & secrets** page. After you leave the page, it won't be displayed again.

IMPORTANT

The client secret value (password) is an important security credential. Do not share the password with anyone or distribute it within a client application.

Add Microsoft Account information to your App Service application

1. Go to your application in the [Azure portal](#).
2. Select **Settings > Authentication / Authorization**, and make sure that **App Service Authentication** is **On**.
3. Under **Authentication Providers**, select **Azure Active Directory**. Select **Advanced** under **Management mode**. Paste in the Application (client) ID and client secret that you obtained earlier. Use `https://login.microsoftonline.com/9188040d-6c67-4c5b-b112-36a304b66dad/v2.0` for the **Issuer Url** field.

4. Select **OK**.

App Service provides authentication, but doesn't restrict authorized access to your site content and APIs. You must authorize users in your app code.

5. (Optional) To restrict access to Microsoft account users, set **Action to take when request is not authenticated** to **Log in with Azure Active Directory**. When you set this functionality, your app requires all requests to be authenticated. It also redirects all unauthenticated requests to use AAD for authentication. Note that because you have configured your **Issuer Url** to use the Microsoft Account tenant, only personal accounts will successfully authenticate.

Caution

Restricting access in this way applies to all calls to your app, which might not be desirable for apps that have a publicly available home page, as in many single-page applications. For such applications, **Allow anonymous requests (no action)** might be preferred so that the app manually starts authentication itself. For more information, see [Authentication flow](#).

6. Select **Save**.

You are now ready to use Microsoft Account for authentication in your app.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

Configure your App Service or Azure Functions app to use Twitter login

4/1/2020 • 2 minutes to read • [Edit Online](#)

This article shows how to configure Azure App Service or Azure Functions to use Twitter as an authentication provider.

To complete the procedure in this article, you need a Twitter account that has a verified email address and phone number. To create a new Twitter account, go to [twitter.com](#).

Register your application with Twitter

1. Sign in to the [Azure portal](#) and go to your application. Copy your **URL**. You'll use it to configure your Twitter app.
2. Go to the [Twitter Developers](#) website, sign in with your Twitter 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/twitter/callback`. For example,
`https://contoso.azurewebsites.net/.auth/login/twitter/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:

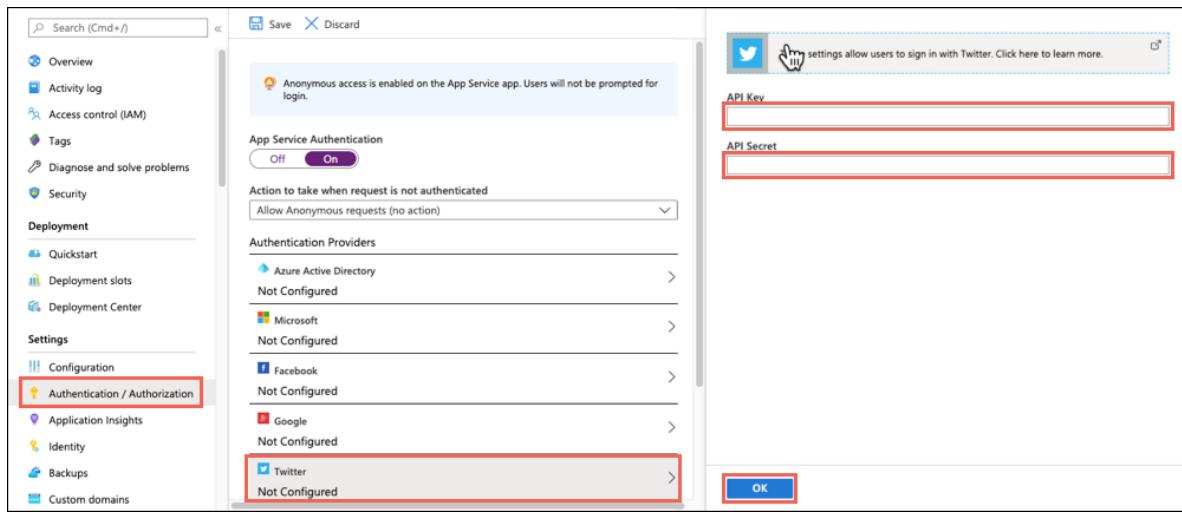
- API key
- API secret key

NOTE

The API secret key is an important security credential. Do not share this secret with anyone or distribute it with your app.

Add Twitter information to your application

1. Go to your application in the [Azure portal](#).
2. Select **Settings > Authentication / Authorization**, and make sure that **App Service Authentication** is **On**.
3. Select **Twitter**.
4. Paste in the `API key` and `API secret key` values that you obtained previously.
5. Select **OK**.



By default, App Service provides authentication but doesn't restrict authorized access to your site content and APIs. You must authorize users in your app code.

6. (Optional) To restrict access to your site to only users authenticated by Twitter, set **Action to take when request is not authenticated** to **Twitter**. When you set this functionality, your app requires all requests to be authenticated. It also redirects all unauthenticated requests to Twitter for authentication.

Caution

Restricting access in this way applies to all calls to your app, which might not be desirable for apps that have a publicly available home page, as in many single-page applications. For such applications, **Allow anonymous requests (no action)** might be preferred so that the app manually starts authentication itself. For more information, see [Authentication flow](#).

7. Select **Save**.

You are now ready to use Twitter for authentication in your app.

Next steps

- [App Service Authentication / Authorization overview](#).
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

Configure your App Service or Azure Functions app to login using an OpenID Connect provider (Preview)

11/2/2020 • 4 minutes to read • [Edit Online](#)

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 do not 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 name in the configuration, and only one can serve as the default redirect target.

Caution

Enabling an OpenID Connect provider 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.

Register your application with the identity provider

Your provider will require you to register the details of your application with it. Please see the instructions relevant to that provider. You will need to collect a **client ID** and **client secret** for your application.

IMPORTANT

The app secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

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 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](#))

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.

Additionally, you will 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 are unable to use a configuration metadata document, you will 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

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.

- Within the `identityProviders` object, add an `openIdConnectProviders` object if one does not already exist.
- Within the `openIdConnectProviders` object, add a key for your new provider. This is a friendly name used to reference the provider in the rest of the configuration. For example, if you wanted to require all requests to be authenticated with this provider, you would set `globalValidation.unauthenticatedClientAction` to "RedirectToLoginPage" and set `redirectToProvider` to this same friendly name.
- Assign an object to that key with a `registration` object within it, and optionally a `login` object:

```
"myCustomIDP" : {
  "registration" : {},
  "login": {
    "nameClaimType": "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name",
    "scope": [],
    "loginParameterNames": []
  }
}
```

- Within the registration object, set the `clientId` to the client ID you collected, set `clientCredential.secretSettingName` to the name of the application setting where you stored the client secret, and create a `openIdConnectConfiguration` object:

```
"registration": {
  "clientId": "bd96cf8a-3f2b-4806-b180-d3c5fd11a2be",
  "clientCredential": {
    "secretSettingName": "IDP_CLIENT_SECRET"
  },
  "openIdConnectConfiguration" : {}
}
```

- Within the `openIdConnectConfiguration` object, provide the OpenID Connect metadata you gathered earlier. There are two options for this, based on which information you collected:

- Set the `wellKnownOpenIdConfiguration` property to the configuration metadata URL you gathered earlier.
- Alternatively, set the four individual values gathered as follows:
 - Set `issuer` to the issuer URL
 - Set `authorizationEndpoint` to the authorization Endpoint
 - Set `tokenEndpoint` to the token endpoint
 - Set `certificationUri` to the URL of the JSON Web Key Set document

These two options are mutually exclusive.

Once this configuration has been set, you are ready to use your OpenID Connect provider for authentication in your app.

An example configuration might look like the following (using Sign in with Apple as an example, where the `APPLE_GENERATED_CLIENT_SECRET` setting points to a generated JWT as per [Apple documentation](#)):

```
{  
    "platform": {  
        "enabled": true  
    },  
    "globalValidation": {  
        "redirectToProvider": "apple",  
        "unauthenticatedClientAction": "RedirectToLoginPage"  
    },  
    "identityProviders": {  
        "openIdConnectProviders": {  
            "apple": {  
                "registration": {  
                    "clientId": "com.contoso.example.client",  
                    "clientCredential": {  
                        "secretSettingName": "APPLE_GENERATED_CLIENT_SECRET"  
                    },  
                    "openIdConnectConfiguration": {  
                        "wellKnownOpenIdConfiguration": "https://appleid.apple.com/.well-known/openid-  
configuration"  
                    }  
                },  
                "login": {  
                    "nameClaimType": "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name",  
                    "scope": [],  
                    "loginParameterNames": []  
                }  
            }  
        }  
    },  
    "login": {  
        "tokenStore": {  
            "enabled": true  
        }  
    }  
}
```

Next steps

- [App Service Authentication / Authorization overview.](#)
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

Configure your App Service or Azure Functions app to sign in using a Sign in with Apple provider (Preview)

12/4/2020 • 6 minutes to read • [Edit Online](#)

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 <input checked="" type="radio"/> iOS, tvOS, watchOS <input type="radio"/> macOS | App ID Prefix E88K2FF6LU (Team ID) |
| Description easy auth test sign in with apple You cannot use special characters such as @, &, *, ;, " | Bundle ID <input checked="" type="radio"/> Explicit <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

 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 | |
| Devices | |
| Profiles | |
| 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 easy auth test sign in with apple You cannot use special characters such as @, &, *, ; | Identifier com.microsoft.easyauthtest.client We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*). |
|--|---|

| | | |
|-------------------------------------|--------------------|---------------------------|
| ENABLED | NAME | Configure |
| <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. Under 'Primary App ID', the value 'Easy Auth test ID (SGGM6D27TK.com.microms)' is selected. In the 'Domains' section, the 'Web Domain' is listed as 'easyauth.net'. Under 'Return URLs', the URL 'https://easyauth.net/auth/login/apple/callback' is entered and an 'Add' button is visible. At the bottom right, there are 'Cancel' and 'Save' buttons, with 'Save' highlighted.

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

- On the Keys tab in the Apple Developer portal, choose **Create a key** or select the (+) button.
- On the **Register a New Key** page give the key a name, check the box next to **Sign in with Apple** and select **Configure**.
- On the **Configure Key** page, link the key to the primary app ID you created previously and select **Save**.
- Finish creating the key by confirming the information and selecting **Continue** and then reviewing the information and selecting **Register**.
- 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:

```
{  
  "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:

```
eyJhbGciOiJFUzI1NiIsImtpZCI6I1VSS0VZSUQwMDEifQ.eyJzdWIiOiJjb20ueW91cmNvbXBhbnkuYXBwMSIsIm5iZiI6MTU2MDIwMzIwNywiZXhwIjoxNTYwMjg5NjA3LCJpc3MiOJBQkMxMjNERUZHIIwiYXVkJoiHR0cHM6Ly9hcHBsZWlkRpdzNg1sgecolPNTmDkaz08-BHAZCsdeeTN1gFEzBytIpMKFvEQbtGKam51ec1UK7S9o0va4EK4jV4VimgDrr-LGWW03TaAxAvy3_ZokohFFkVG
```

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 [PKCS#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 jwt.io. 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.

```
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:

```
"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):

```
{
  "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.](#)
- [Advanced usage of authentication and authorization in Azure App Service](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

Advanced usage of authentication and authorization in Azure App Service

12/4/2020 • 17 minutes to read • [Edit Online](#)

This article shows you how to customize the built-in [authentication and authorization in App Service](#), and to manage identity from your application.

To get started quickly, see one of the following tutorials:

- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service](#)
- [How to configure your app to use Azure Active Directory login](#)
- [How to configure your app to use Facebook login](#)
- [How to configure your app to use Google login](#)
- [How to configure your app to use Microsoft Account login](#)
- [How to configure your app to use Twitter login](#)
- [How to configure your app to login using an OpenID Connect provider \(Preview\)](#)
- [How to configure your app to login using an Sign in with Apple \(Preview\)](#)

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:

```
<a href="/.auth/login/aad">Log in with Azure AD</a>
<a href="/.auth/login/microsoftaccount">Log in with Microsoft Account</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_url` 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:

```
<a href="/.auth/login/<provider>?post_login_redirect_url=/Home/Index">Log in</a>
```

Validate tokens from providers

In a client-directed sign-in, the application signs in the user to the provider manually and then submits the

authentication token to App Service for validation (see [Authentication flow](#)). 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:

| Provider Value | Required in Request Body | Comments |
|------------------|--|--|
| aad | {"access_token": "<access_token>"} | |
| microsoftaccount | {"access_token": "<token>"} | The <code>expires_in</code> property is optional. When requesting the token from Live services, always request the <code>wl.basic</code> scope. |
| google | {"id_token": "<id_token>"} | The <code>authorization_code</code> property is optional. When specified, it can also optionally be accompanied by the <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>"} | |

If the provider token is validated successfully, the API returns with an `authenticationToken` in the response body, which is your session token.

```
{
  "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 Azure Active Directory and Google, performs a server-side sign-out on the identity provider.

Here's a simple sign-out link in a webpage:

```
<a href="/.auth/logout">Sign out</a>
```

By default, a successful sign-out redirects the client to the URL `/.auth/logout/done`. 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](#):

```
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](#):

```
az webapp config appsettings set --name <app_name> --resource-group <group_name> --settings WEBSITE_AUTH_PRESERVE_URL_FRAGMENT="true"
```

Access user claims

App Service passes user claims to your application by using special headers. External requests aren't allowed to set these headers, so they are present only if set by App Service. Some example headers include:

- `X-MS-CLIENT-PRINCIPAL-NAME`

- X-MS-CLIENT-PRINCIPAL-ID

Code that is written in any language or framework can get the information that it needs from these headers. For ASP.NET 4.6 apps, the **ClaimsPrincipal** is automatically set with the appropriate values. ASP.NET Core, however, doesn't provide an authentication middleware that integrates with App Service user claims. For a workaround, see [MaximeRouiller.Azure.AppService.EasyAuth](#).

If the **token store** is enabled for your app, you can also obtain additional details on the authenticated user by calling `/auth/me`. The Mobile Apps server SDKs provide helper methods to work with this data. For more information, see [How to use the Azure Mobile Apps Node.js SDK](#), and [Work with the .NET backend server SDK for Azure Mobile Apps](#).

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 |
| Microsoft Account | X-MS-TOKEN-MICROSOFTACCOUNT-ACCESS-TOKEN X-MS-TOKEN-MICROSOFTACCOUNT-EXPIRES-ON X-MS-TOKEN-MICROSOFTACCOUNT-AUTHENTICATION-TOKEN X-MS-TOKEN-MICROSOFTACCOUNT-REFRESH-TOKEN |
| Twitter | X-MS-TOKEN-TWITTER-ACCESS-TOKEN X-MS-TOKEN-TWITTER-ACCESS-TOKEN-SECRET |

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 identity provider 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. If using the Mobile Apps SDK, you can add the parameter to one of the `LogicAsync` overloads (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 Account:** When [configuring Microsoft Account Authentication Settings](#), select the `wl.offline_access` scope.
- **Azure Active Directory:** 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 > authsettings**.
 3. Click **Edit**.
 4. Modify the following property. Replace `<app_id>` with the Azure Active Directory application ID of the service you want to access.

```
"additionalLoginParams": ["response_type=code id_token", "resource=<app_id>"]
```

5. Click **Put**.

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.

```
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](#).

```
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.

Limit the domain of sign-in accounts

Both Microsoft Account and Azure Active Directory lets you sign in from multiple domains. For example, Microsoft Account allows *outlook.com*, *live.com*, and *hotmail.com* accounts. Azure AD allows any number of custom domains for the sign-in accounts. However, you may want to accelerate your users straight to your own branded Azure AD sign-in page (such as `contoso.com`). To suggest the domain name of the sign-in accounts, follow these steps.

In <https://resources.azure.com>, navigate to `subscriptions > <subscription_name> resourceGroups > <resource_group_name> > providers > Microsoft.Web > sites > <app_name> > config > authsettings`.

Click **Edit**, modify the following property, and then click **Put**. Be sure to replace `<domain_name>` with the domain you want.

```
"additionalLoginParams": ["domain_hint=<domain_name>"]
```

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 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:

- For [Azure App Service](#), you can [manage enterprise-level access directly in Azure AD](#). 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](#).

Configure using a file (preview)

Your auth settings can optionally be configured via a file that is provided by your deployment. This may be required by certain preview capabilities of App Service Authentication / Authorization.

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

Caution

During preview, enabling file-based configuration 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.

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/authsettings](#). To modify this, you can use an [Azure Resource Manager template](#) or a tool like [Azure Resource Explorer](#). Within the `authsettings` collection, you will need to set three properties (and may remove others):

- a. Set `enabled` to "true"
- b. Set `isAuthFromFile` to "true"
- c. Set `authFilePath` to the name of the file (for example, "auth.json")

NOTE

The format for `authFilePath` 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 setting `isAuthFromFile` back to "false".

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:

```
{
  "platform": {
    "enabled": <true|false>
  },
  "globalValidation": {
    "unauthenticatedClientAction": "RedirectLoginPage|AllowAnonymous|Return401|Return403",
    "redirectToProvider": "<default provider alias>",
    "excludedPaths": [
      "/path1",
      "/path2"
    ]
  },
  "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 file system token store
(default)>"
      },
      "azureBlobStorage": {
        "sasUrlSettingName": "<app setting name containing the sas url for the Azure Blob Storage if
opting to use that for a token store>"
      }
    },
    "preserveUrlFragmentsForLogins": <true|false>,
    "allowedExternalRedirectUrls": [
      ...
    ]
  }
}
```

```
"https://uri1.azurewebsites.net/",
"https://uri2.azurewebsites.net/",
"url_scheme_of_your_app://easyauth.callback"
],
"cookieExpiration": {
    "convention": "FixedTime|IdentityDerived",
    "timeToExpiration": "<timespan>"
},
"nonce": {
    "validateNonce": <true|false>,
    "nonceExpirationInterval": "<timespan>"
}
},
"identityProviders": {
    "azureActiveDirectory": {
        "enabled": <true|false>,
        "registration": {
            "openIdIssuer": "<issuer url>",
            "clientId": "<app id>",
            "clientSecretSettingName": "APP_SETTING_CONTAINING_AAD_SECRET",
        },
        "login": {
            "loginParameters": [
                "paramName1=value1",
                "paramName2=value2"
            ]
        },
        "validation": {
            "allowedAudiences": [
                "audience1",
                "audience2"
            ]
        }
    },
    "facebook": {
        "enabled": <true|false>,
        "registration": {
            "appId": "<app id>",
            "appSecretSettingName": "APP_SETTING_CONTAINING_FACEBOOK_SECRET"
        },
        "graphApiVersion": "v3.3",
        "login": {
            "scopes": [
                "public_profile",
                "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"
            ],
            "///
        }
    },
    //...
}
}

```

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.

```
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 will see the `runtimeVersion` field in the CLI output. It will resemble the following example output, which has been truncated for clarity:

```
{
  "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:

```
{
  "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.

```
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. You can find the release notes on the different runtime versions [here] (<https://github.com/Azure/app-service-announcements>) 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](#)

Set up Azure App Service access restrictions

12/4/2020 • 6 minutes to read • [Edit Online](#)

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.

The access-restriction capability works with all Azure App Service-hosted workloads. The workloads can include web apps, API apps, Linux apps, Linux container apps, and functions.

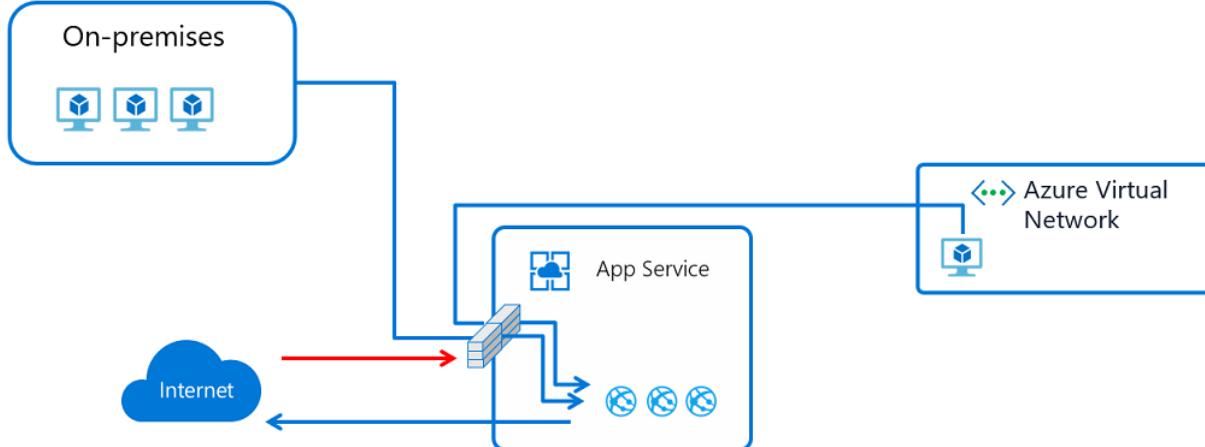
When a request is made to your app, the FROM address is evaluated against the IP address rules in your access-restriction list. If the FROM address is in a subnet that's configured with service endpoints to Microsoft.Web, the source subnet is compared against the virtual network rules in your access-restriction 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 is enabled by [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](#).



Add or edit access-restriction rules in the portal

To add an access-restriction rule to your app, do the following:

1. Sign in to the Azure portal.
2. On the left pane, select **Networking**.
3. On the **Networking** pane, under **Access Restrictions**, select **Configure Access Restrictions**.

VNet Integration
Securely access resources available in or through your Azure VNet.
[Learn More](#)

Hybrid connections
Securely access applications in private networks
[Learn More](#)

Azure CDN
Secure, reliable content delivery with broad global reach and rich feature set
[Learn More](#)

Access Restrictions
Define and manage rules that control access to your application.
[Learn More](#)

- On the Access Restrictions page, review the list of access-restriction rules that are defined for your app.

| PRIORITY | NAME | SOURCE | ENDPOINT STATUS | ACTION |
|------------|-----------------|---------------------------------------|-----------------|--|
| 100 | IP example rule | 122.133.144.0/24 | Enabled | ✓ Allow |
| 150 | deny example | 122.133.144.32/28 | Enabled | ✗ Deny |
| 200 | test rule | networking-demos-vnet/simple-se-su... | Enabled | ✓ Allow |
| 2147483647 | Deny all | Any | | ✗ Deny |

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, the app is accessible from anywhere.

Add an access-restriction rule

To add an access-restriction rule to your app, on the **Access Restrictions** pane, select **Add rule**. After you add a rule, it becomes effective immediately.

Rules are enforced in priority order, starting from the lowest number in the **Priority** column. An implicit *deny all* is in effect after you add even a single rule.

On the **Add IP Restriction** pane, when you create a rule, do the following:

- Under **Action**, select either **Allow** or **Deny**.

Add IP Restriction

The screenshot shows the 'Add IP Restriction' configuration page. It includes fields for Name (with placeholder 'Enter name for the IpAddress rule'), Action (Allow or Deny), Priority (Ex. 300), Description, Type (Virtual Network), Subscription (Purple Demo Subscription), Virtual Network (networking-demos-vnet), and Subnet (app-gateway-subnet). A large 'Add rule' button is at the bottom.

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
app-gateway-subnet

Add rule

2. Optionally, enter a name and description of the rule.
3. In the **Type** drop-down list, select the type of rule.
4. In the **Priority** box, enter a priority value.
5. In the **Subscription**, **Virtual Network**, and **Subnet** drop-down lists, select what you want to restrict access to.

Set an IP address-based rule

Follow the procedure as outlined in the preceding section, but with the following variation:

- For step 3, in the **Type** drop-down list, select **IPv4** or **IPv6**.

Specify the IP address 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](#).

Use service endpoints

By using service endpoints, you can restrict access to selected Azure virtual network subnets. 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'll be 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 will be configured for service endpoints in anticipation of having them enabled later on the subnet.

Add IP Restriction

Name i
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
vnet-integration-subnet

 Selected subnet 'networking-demos-vnet/vnet-integration-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.

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 currently supported for web apps that use IP Secure Sockets Layer (SSL) virtual IP (VIP).
- There is a limit of 512 rows of IP or service-endpoint restrictions. If you require more than 512 rows of restrictions, we suggest that you consider installing a standalone security product, such as Azure Front Door, Azure App Gateway, or a WAF.

Manage access-restriction rules

You can edit or delete an existing access-restriction rule.

Edit a rule

1. To begin editing an existing access-restriction rule, on the **Access Restrictions** page, double-click the rule you want to edit.
2. On the **Edit IP Restriction** pane, make your changes, and then select **Update rule**. Edits are effective immediately, including changes in priority ordering.

Edit IP Restriction

Name

* Priority

Action
 Allow Deny

Description

* Subscription

* Virtual Network

* Subnet

NOTE

When you edit a rule, you can't switch between an IP address rule and a virtual network rule.

Edit IP Restriction

Name

* Priority

Action
 Allow Deny

Description

* Subscription

* Virtual Network

* Subnet

Delete a rule

To delete a rule, on the **Access Restrictions** page, select the ellipsis (...) next to the rule you want to delete, and then select **Remove**.

| PRIORITY | NAME | SOURCE | ENDPOINT STATUS | ACTION |
|------------|-----------------|--|-----------------|--------------------|
| 100 | IP example rule | 122.133.144.0/24 | | Allow |
| 150 | deny example | 122.133.144.32/28 | | Deny |
| 200 | test rule | networking-demos-vnet/simple-se-sub... | Enabled | Allow |
| 2147483647 | Deny all | Any | | Deny |

Block a single IP address

When you add your first IP restriction rule, the service adds an explicit *Deny all* rule with a priority of 2147483647. In practice, the explicit *Deny all* rule is the final rule to be executed, and it blocks access to any IP address that's not explicitly allowed by an *Allow* rule.

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 an explicit *Allow All* rule.

| PRIORITY | NAME | SOURCE | ENDPOINT STATUS | ACTION |
|------------|-----------|----------------|-----------------|--------------------|
| 200 | Block Me | 131.107.147/32 | | Deny |
| 300 | Allow All | 0.0.0.0/0 | | Allow |
| 2147483647 | Deny all | Any | | Deny |

Restrict access to an SCM site

In addition to being able to control access to your app, you can restrict access to the SCM site that's 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 **Same restrictions as <app name>** check box, everything is blanked out. If you clear the check box, your SCM site settings are reapplied.

The screenshot shows the 'Access Restrictions' page for a web application named 'vnet-integration-app'. At the top, there are two tabs: 'vnet-integration-app.azurewebsites.net' (selected) and 'vnet-integration-app.scm.azurewebsites.net'. Below the tabs, there's a note: 'Same restrictions as vnet-integration-app.azurewebsites.net'. A large blue button labeled '+ Add rule' is visible. A table lists one rule:

| PRIORITY | NAME | SOURCE | ENDPOINT STATUS | ACTION |
|----------|-----------|--------|-----------------|--------|
| 1 | Allow all | Any | Green checkmark | Allow |

Manage access-restriction rules programmatically

You can add access restrictions programmatically by doing either of the following:

- Use the [Azure CLI](#). For example:

```
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
```

- Use [Azure PowerShell](#). For example:

```
Add-AzWebAppAccessRestrictionRule -ResourceGroupName "ResourceGroup" -WebAppName "AppName"
-Name "Ip example rule" -Priority 100 -Action Allow -IpAddress 122.133.144.0/24
```

You can also set values manually by doing either of the following:

- Use an [Azure REST API](#) PUT operation on the app configuration in Azure Resource Manager. The location for this information in Azure Resource Manager is:

`management.azure.com/subscriptions/subscription ID/resourceGroups/resource
groups/providers/Microsoft.Web/sites/web app name/config/web?api-version=2018-02-01`

- Use an ARM template. As an example, you can use [resources.azure.com](#) and edit the ipSecurityRestrictions block to add the required JSON.

The JSON syntax for the earlier example is:

```
{
  "properties": {
    "ipSecurityRestrictions": [
      {
        "ipAddress": "122.133.144.0/24",
        "action": "Allow",
        "priority": 100,
        "name": "IP example rule"
      }
    ]
  }
}
```

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](#)

How to use managed identities for App Service and Azure Functions

12/4/2020 • 16 minutes to read • [Edit Online](#)

This topic 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

Managed identities for App Service and Azure Functions won't behave as expected if your app is migrated across subscriptions/tenants. The app needs to obtain a new identity, which is done by disabling and re-enabling the feature. See [Removing an identity](#) below. Downstream resources also need to have access policies updated to use the new identity.

A managed identity from Azure Active Directory (Azure AD) allows your app to easily access other Azure AD-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 Azure AD, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to your application and is deleted if your 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.

Add a system-assigned identity

Creating an app with a system-assigned identity requires an additional property to be set on the application.

Using the Azure portal

To set up a managed identity in the portal, you will first create an application as normal and then enable the feature.

1. Create an app in the portal as you normally would. Navigate to it in the portal.
2. If using a function app, navigate to **Platform features**. For other app types, 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 portal interface for managing the identity of an App Service. The left sidebar lists various settings like Authentication / Authorization, Application Insights, Backups, Custom domains, TLS/SSL settings, Networking, and Scale up (App Service plan). The 'Identity' section is highlighted with a red box. The main content area shows the 'System assigned' tab selected, with a note explaining its function. A status switch is shown as 'On', also highlighted with a red box. At the bottom, there are 'Save', 'Discard', 'Refresh', and 'Got feedback?' buttons.

NOTE

To find the managed identity for your web app or slot app in the Azure portal, under **Enterprise applications**, look in the **User settings** section. Usually, the slot name is similar to <app name>/slots/<slot name> .

Using the Azure CLI

To set up a managed identity using the Azure CLI, you will need to use the `az webapp identity assign` command against an existing application. You have three options for running the examples in this section:

- Use [Azure Cloud Shell](#) from the Azure portal.
- Use the embedded Azure Cloud Shell via the "Try It" button, located in the top-right corner of each code block below.
- [Install the latest version of Azure CLI](#) (2.0.31 or later) if you prefer to use a local CLI console.

The following steps will walk you through creating a web app and assigning it an identity using the CLI:

1. If you're using the Azure CLI in a local console, first sign in to Azure using `az login`. Use an account that's associated with the Azure subscription under which you would like to deploy the application:

```
az login
```

2. Create a web application using the CLI. For more examples of how to use the CLI with App Service, see [App Service CLI samples](#):

```
az group create --name myResourceGroup --location westus
az appservice plan create --name myPlan --resource-group myResourceGroup --sku S1
az webapp create --name myApp --resource-group myResourceGroup --plan myPlan
```

3. Run the `identity assign` command to create the identity for this application:

```
az webapp identity assign --name myApp --resource-group myResourceGroup
```

Using Azure PowerShell

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

The following steps will walk you through creating an app and assigning it an identity using Azure PowerShell. The instructions for creating a web app and a function app are different.

Using Azure PowerShell for a web app

1. If needed, install the Azure PowerShell using the instructions found in the [Azure PowerShell guide](#), and then run `Login-AzAccount` to create a connection with Azure.
2. Create a web application using Azure PowerShell. For more examples of how to use Azure PowerShell with App Service, see [App Service PowerShell samples](#):

```
# Create a resource group.  
New-AzResourceGroup -Name $resourceGroupName -Location $location  
  
# Create an App Service plan in Free tier.  
New-AzAppServicePlan -Name $webappName -Location $location -ResourceGroupName $resourceGroupName -Tier Free  
  
# Create a web app.  
New-AzWebApp -Name $webappName -Location $location -AppServicePlan $webappName -ResourceGroupName $resourceGroupName
```

3. Run the `Set-AzWebApp -AssignIdentity` command to create the identity for this application:

```
Set-AzWebApp -AssignIdentity $true -Name $webappName -ResourceGroupName $resourceGroupName
```

Using Azure PowerShell for a function app

1. If needed, install the Azure PowerShell using the instructions found in the [Azure PowerShell guide](#), and then run `Login-AzAccount` to create a connection with Azure.
2. Create a function app using Azure PowerShell. For more examples of how to use Azure PowerShell with Azure Functions, see the [Az.Functions reference](#):

```
# Create a resource group.  
New-AzResourceGroup -Name $resourceGroupName -Location $location  
  
# Create a storage account.  
New-AzStorageAccount -Name $storageAccountName -ResourceGroupName $resourceGroupName -SkuName $sku  
  
# Create a function app with a system-assigned identity.  
New-AzFunctionApp -Name $functionAppName -ResourceGroupName $resourceGroupName -Location $location -  
StorageAccountName $storageAccountName -Runtime $runtime -IdentityType SystemAssigned
```

You can also update an existing function app using `Update-AzFunctionApp` instead.

Using an Azure Resource Manager template

An Azure Resource Manager template can be used to automate deployment of your Azure resources. To learn more about deploying to App Service and Functions, see [Automating resource deployment in App Service](#) and [Automating resource deployment in Azure Functions](#).

Any resource of type `Microsoft.Web/sites` can be created with an identity by including the following property in the resource definition:

```
"identity": {  
    "type": "SystemAssigned"  
}
```

NOTE

An application can have both system-assigned and user-assigned identities at the same time. In this case, the `type` property would be `SystemAssigned,UserAssigned`

Adding the system-assigned type tells Azure to create and manage the identity for your application.

For example, a web app might look like the following:

```
{  
    "apiVersion": "2016-08-01",  
    "type": "Microsoft.Web/sites",  
    "name": "[variables('appName')]",  
    "location": "[resourceGroup().location]",  
    "identity": {  
        "type": "SystemAssigned"  
    },  
    "properties": {  
        "name": "[variables('appName')]",  
        "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",  
        "hostingEnvironment": "",  
        "clientAffinityEnabled": false,  
        "alwaysOn": true  
    },  
    "dependsOn": [  
        "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]"  
    ]  
}
```

When the site is created, it has the following additional properties:

```
"identity": {  
    "type": "SystemAssigned",  
    "tenantId": "<TENANTID>",  
    "principalId": "<PRINCIPALID>"  
}
```

The `tenantId` property identifies what Azure AD tenant the identity belongs to. The `principalId` is a unique identifier for the application's new identity. Within Azure AD, the service principal has the same name that you gave to your App Service or Azure Functions instance.

If you need to reference these properties in a later stage in the template, you can do so via the [reference\(\)](#) template function with the `'Full'` flag, as in this example:

```
{  
    "tenantId": "[reference(resourceId('Microsoft.Web/sites', variables('appName')), '2018-02-01',  
    'Full').identity.tenantId]",  
    "objectId": "[reference(resourceId('Microsoft.Web/sites', variables('appName')), '2018-02-01',  
    'Full').identity.principalId]"  
}
```

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.

Using the Azure portal

First, you'll need to create a user-assigned identity resource.

1. Create a user-assigned managed identity resource according to [these instructions](#).
2. Create an app in the portal as you normally would. Navigate to it in the portal.
3. If using a function app, navigate to **Platform features**. For other app types, scroll down to the **Settings** group in the left navigation.
4. Select **Identity**.
5. Within the **User assigned** tab, click **Add**.
6. Search for the identity you created earlier and select it. Click **Add**.

The screenshot shows two windows side-by-side. On the left is the 'userassigned-windows - Identity' blade under 'App Services'. It has tabs for 'System assigned' and 'User assigned'. A red box highlights the 'Identity' link in the left sidebar. On the right is the 'Add user assigned managed identity' dialog. It has a 'Subscription' dropdown set to 'APEX C+L - Aquent Vendor Subscriptions'. A red box highlights the search bar where 'userassig' is typed. Below it, a list shows 'userassignedmanagedidentity' with a red box highlighting it. At the bottom is a large red box around the 'Add' button.

Using Azure PowerShell

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

The following steps will walk you through creating an app and assigning it an identity using Azure PowerShell.

NOTE

The current version of the Azure PowerShell commandlets for Azure App Service do not support user-assigned identities. The below instructions are for Azure Functions.

1. If needed, install the Azure PowerShell using the instructions found in the [Azure PowerShell guide](#), and then run `Login-AzAccount` to create a connection with Azure.
2. Create a function app using Azure PowerShell. For more examples of how to use Azure PowerShell with Azure Functions, see the [Az.Functions reference](#). The below script also makes use of `New-AzUserAssignedIdentity` which must be installed separately as per [Create, list or delete a user-assigned managed identity using Azure PowerShell](#).

```

# Create a resource group.
New-AzResourceGroup -Name $resourceGroupName -Location $location

# Create a storage account.
New-AzStorageAccount -Name $storageAccountName -ResourceGroupName $resourceGroupName -SkuName $sku

# Create a user-assigned identity. This requires installation of the "Az.ManagedServiceIdentity" module.
$userAssignedIdentity = New-AzUserAssignedIdentity -Name $userAssignedIdentityName -ResourceGroupName
$resourceGroupName

# Create a function app with a user-assigned identity.
New-AzFunctionApp -Name $functionAppName -ResourceGroupName $resourceGroupName -Location $location -
StorageAccountName $storageAccountName -Runtime $runtime -IdentityType UserAssigned -IdentityId
$userAssignedIdentity.Id

```

You can also update an existing function app using `Update-AzFunctionApp` instead.

Using an Azure Resource Manager template

An Azure Resource Manager template can be used to automate deployment of your Azure resources. To learn more about deploying to App Service and Functions, see [Automating resource deployment in App Service](#) and [Automating resource deployment in Azure Functions](#).

Any resource of type `Microsoft.Web/sites` can be created with an identity by including the following block in the resource definition, replacing `<RESOURCEID>` with the resource ID of the desired identity:

```

"identity": {
    "type": "UserAssigned",
    "userAssignedIdentities": {
        "<RESOURCEID>": {}
    }
}

```

NOTE

An application can have both system-assigned and user-assigned identities at the same time. In this case, the `type` property would be `SystemAssigned,UserAssigned`

Adding the user-assigned type tells Azure to use the user-assigned identity specified for your application.

For example, a web app might look like the following:

```
{
  "apiVersion": "2016-08-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('appName')]",
  "location": "[resourceGroup().location]",
  "identity": {
    "type": "UserAssigned",
    "userAssignedIdentities": {
      "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', variables('identityName'))]": {}
    }
  },
  "properties": {
    "name": "[variables('appName')]",
    "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "hostingEnvironment": "",
    "clientAffinityEnabled": false,
    "alwaysOn": true
  },
  "dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
    "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', variables('identityName'))]"
  ]
}
```

When the site is created, it has the following additional properties:

```
"identity": {
  "type": "UserAssigned",
  "userAssignedIdentities": {
    "<RESOURCEID>": {
      "principalId": "<PRINCIPALID>",
      "clientId": "<CLIENTID>"
    }
  }
}
```

The principalId is a unique identifier for the identity that's used for Azure AD administration. The clientId is a unique identifier for the application's new identity that's used for specifying which identity to use during runtime calls.

Obtain tokens for Azure resources

An app can use its managed identity to get tokens to access other resources protected by Azure AD, such as Azure Key Vault. These tokens represent the application accessing the resource, and not any specific user of the application.

You may need to configure the target resource to allow access from your application. For example, if you request a token to access Key Vault, you need to make sure you have added an access policy that includes your application's identity. Otherwise, your calls to Key Vault will be rejected, even if they include the token. To learn more about which resources support Azure Active Directory tokens, see [Azure services that support Azure AD authentication](#).

IMPORTANT

The back-end services for managed identities maintain a cache per resource URI for around 8 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.

There is a simple REST protocol for obtaining a token in App Service and Azure Functions. This can be used for all applications and languages. For .NET and Java, the Azure SDK provides an abstraction over this protocol and facilitates a local development experience.

Using the REST protocol

NOTE

An older version of this protocol, using the "2017-09-01" API version, used the `secret` header instead of `X-IDENTITY-HEADER` and only accepted the `clientid` property for user-assigned. It also returned the `expires_on` in a timestamp format. `MSI_ENDPOINT` can be used as an alias for `IDENTITY_ENDPOINT`, and `MSI_SECRET` can be used as an alias for `IDENTITY_HEADER`. This version of the protocol is currently required for Linux Consumption hosting plans.

An app with a managed identity has two environment variables defined:

- `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:

| PARAMETER NAME | IN | DESCRIPTION |
|-------------------|--------|--|
| resource | Query | The Azure AD resource URI of the resource for which a token should be obtained. This could be one of the Azure services that support Azure AD authentication or any other resource URI. |
| api-version | Query | The version of the token API to be used. Please use "2019-08-01" or later (unless using Linux Consumption, which currently only offers "2017-09-01" - see note above). |
| X-IDENTITY-HEADER | Header | The value of the <code>IDENTITY_HEADER</code> 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. |

| PARAMETER NAME | IN | DESCRIPTION |
|----------------|-------|---|
| 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 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.

A successful 200 OK response includes a JSON body with the following properties:

| PROPERTY NAME | DESCRIPTION |
|---------------|--|
| access_token | The requested access token. The calling web service can use this token to authenticate to the receiving web service. |
| client_id | The client ID of the identity that was used. |
| expires_on | The timespan when the access token expires. The date is represented as the number of seconds from "1970-01-01T0:0Z UTC" (corresponds to the token's <code>exp</code> claim). |
| not_before | The timespan when the access token takes effect, and can be accepted. The date is represented as the number of seconds from "1970-01-01T0:0Z UTC" (corresponds to the token's <code>nbf</code> claim). |
| resource | The resource the access token was requested for, which matches the <code>resource</code> query string parameter of the request. |
| token_type | Indicates the token type value. The only type that Azure AD supports is Bearer. For more information about bearer tokens, see The OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC 6750) . |

This response is the same as the [response for the Azure AD service-to-service access token request](#).

REST protocol examples

An example request might look like the following:

```
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/1.1 200 OK
Content-Type: application/json

{
    "access_token": "eyJ0eXAi...",
    "expires_on": "1586984735",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer",
    "client_id": "5E29463D-71DA-4FE0-8E69-999B57DB23B0"
}
```

Code examples

- [.NET](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

TIP

For .NET languages, you can also use [Microsoft.Azure.Services.AppAuthentication](#) instead of crafting this request yourself.

```
private readonly HttpClient _client;
// ...
public async Task<HttpResponseMessage> GetToken(string resource) {
    var request = new HttpRequestMessage(HttpMethod.Get,
        String.Format("{0}/?resource={1}&api-version=2019-08-01",
        Environment.GetEnvironmentVariable("IDENTITY_ENDPOINT"), resource));
    request.Headers.Add("X-IDENTITY-HEADER", Environment.GetEnvironmentVariable("IDENTITY_HEADER"));
    return await _client.SendAsync(request);
}
```

Using the Microsoft.Azure.Services.AppAuthentication library for .NET

For .NET applications and functions, the simplest way to work with a managed identity is through the [Microsoft.Azure.Services.AppAuthentication](#) package. This library will also allow you to test your code locally on your development machine, using your user account from Visual Studio, the [Azure CLI](#), or Active Directory Integrated Authentication. When hosted in the cloud, it will default to using a system-assigned identity, but you can customize this behavior using a connection string environment variable which references the client ID of a user-assigned identity. For more on development options with this library, see the [Microsoft.Azure.Services.AppAuthentication reference](#). This section shows you how to get started with the library in your code.

1. Add references to the [Microsoft.Azure.Services.AppAuthentication](#) and any other necessary NuGet packages to your application. The below example also uses [Microsoft.Azure.KeyVault](#).

2. Add the following code to your application, modifying to target the correct resource. This example shows two ways to work with Azure Key Vault:

```
using Microsoft.Azure.Services.AppAuthentication;
using Microsoft.Azure.KeyVault;
// ...
var azureServiceTokenProvider = new AzureServiceTokenProvider();
string accessToken = await azureServiceTokenProvider.GetAccessTokenAsync("https://vault.azure.net");
// OR
var kv = new KeyVaultClient(new
KeyVaultClient.AuthenticationCallback(azureServiceTokenProvider.KeyVaultTokenCallback));
```

If you want to use a user-assigned managed identity, you can set the `AzureServicesAuthConnectionString` application setting to `RunAs=App;AppId=<clientId-guid>`. Replace `<clientId-guid>` with the client ID of the identity you want to use. You can define multiple such connection strings by using custom application settings and passing their values into the `AzureServiceTokenProvider` constructor.

```
var identityConnectionString1 = Environment.GetEnvironmentVariable("UA1_ConnectionString");
var azureServiceTokenProvider1 = new AzureServiceTokenProvider(identityConnectionString1);

var identityConnectionString2 = Environment.GetEnvironmentVariable("UA2_ConnectionString");
var azureServiceTokenProvider2 = new AzureServiceTokenProvider(identityConnectionString2);
```

To learn more about configuring `AzureServiceTokenProvider` and the operations it exposes, see the [Microsoft.Azure.Services.AppAuthentication reference](#) and the [App Service and KeyVault with MSI .NET sample](#).

Using the Azure SDK for Java

For Java applications and functions, the simplest way to work with a managed identity is through the [Azure SDK for Java](#). This section shows you how to get started with the library in your code.

1. Add a reference to the [Azure SDK library](#). For Maven projects, you might add this snippet to the `dependencies` section of the project's POM file:

```
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure</artifactId>
    <version>1.23.0</version>
</dependency>
```

2. Use the `AppServiceMSICredentials` object for authentication. This example shows how this mechanism may be used for working with Azure Key Vault:

```
import com.microsoft.azure.AzureEnvironment;
import com.microsoft.azure.management.Azure;
import com.microsoft.azure.management.keyvault.Vault
//...
Azure azure = Azure.authenticate(new AppServiceMSICredentials(AzureEnvironment.AZURE))
    .withSubscription(subscriptionId);
Vault myKeyVault = azure.vaults().getByResourceGroup(resourceGroup, keyvaultName);
```

Remove an identity

A system-assigned identity can be removed by disabling the feature using the portal, PowerShell, or CLI in the same way that it was created. User-assigned identities can be removed individually. To remove all identities, set the identity type to "None".

Removing a system-assigned identity in this way will also delete it from Azure AD. System-assigned identities are also automatically removed from Azure AD when the app resource is deleted.

To remove all identities in an [ARM template](#):

```
"identity": {  
    "type": "None"  
}
```

To remove all identities in Azure PowerShell (Azure Functions only):

```
# Update an existing function app to have IdentityType "None".  
Update-AzFunctionApp -Name $functionAppName -ResourceGroupName $resourceGroupName -IdentityType None
```

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.

Next steps

[Access SQL Database securely using a managed identity](#)

Use Key Vault references for App Service and Azure Functions

11/2/2020 • 4 minutes to read • [Edit Online](#)

This topic shows you how to work with secrets from Azure Key Vault in your App Service or 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.

Granting your app access to Key Vault

In order to read secrets from 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 [system-assigned managed identity](#) for your application.

NOTE

Key Vault references currently only support system-assigned managed identities. User-assigned identities cannot be used.

3. Create an [access policy in Key Vault](#) for the application identity you created earlier. 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.

IMPORTANT

Key Vault references are not presently able to resolve secrets stored in a key vault with [network restrictions](#) unless the app is hosted within an [App Service Environment](#).

Reference syntax

A Key Vault reference is of the form `@Microsoft.KeyVault({referenceString})`, where `{referenceString}` is replaced by one of the following options:

| REFERENCE STRING | DESCRIPTION |
|--|--|
| <code>SecretUri=secretUri</code> | The SecretUri should be the full data-plane URI of a secret in Key Vault, including a version, e.g., https://myvault.vault.azure.net/secrets/mysecret/ec96f02080254f109c51a1f14cdb1931 |
| <code>VaultName=vaultName;SecretName=secretName;SecretVersion=secretVersion</code> | The VaultName should be the name of your Key Vault resource. The SecretName should be the name of the target secret. The SecretVersion should be the version of the secret to use. |

NOTE

Versions are currently required. When rotating secrets, you will need to update the version in your application configuration. For example, a complete reference would look like the following:

```
@Microsoft.KeyVault(SecretUri=https://myvault.vault.azure.net/secrets/mysecret/ec96f02080254f109c51a1f14cdb1931)
```

Alternatively:

```
@Microsoft.KeyVault(VaultName=myvault;SecretName=mysecret;SecretVersion=ec96f02080254f109c51a1f14cdb1931)
```

Source Application Settings from Key Vault

Key Vault references can be used as values for [Application Settings](#), allowing you to keep secrets in Key Vault instead of the site config. Application Settings are securely encrypted at rest, but if you need secret management capabilities, they should go into Key Vault.

To use a Key Vault reference for an application setting, 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 application settings using Key Vault references should be marked as slot settings, as you should have separate vaults for each environment.

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 will 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.

An example pseudo-template for a function app might look like the following:

```
{
  //...
  "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"
      },
      //...
    }
  ]
}
```

```

    {
        "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('storageConnectionStringResourceId')).secretUriWithVersion, ')')]",
            "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING": "[concat('@Microsoft.KeyVault(SecretUri=',
reference(variables('storageConnectionStringResourceId')).secretUriWithVersion, ')')]",
            "APPINSIGHTS_INSTRUMENTATIONKEY": "[concat('@Microsoft.KeyVault(SecretUri=',
reference(variables('appInsightsKeyResourceId')).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')]"
        ],
        "properties": {
            //...
        }
    }
},
{
    "type": "Microsoft.KeyVault/vaults",
    "name": "[variables('keyVaultName')]",
    //...
    "dependsOn": [
        "[resourceId('Microsoft.Web/sites', variables('functionAppName'))]"
    ],
    "properties": {
        //...
        "accessPolicies": [
            {
                "tenantId": "[reference(concat('Microsoft.Web/sites/', variables('functionAppName'), '/providers/Microsoft.ManagedIdentity/Identities/default'), '2015-08-31-PREVIEW').tenantId]",
                "objectId": "[reference(concat('Microsoft.Web/sites/', variables('functionAppName'), '/providers/Microsoft.ManagedIdentity/Identities/default'), '2015-08-31-PREVIEW').principalId]",
                "permissions": {
                    "secrets": [ "get" ]
                }
            }
        ]
    }
},
{
    "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'), '2015-05-01-'
    }
}

```

```

variables['storageAccountName'], accountKey, secrets['variables['storageAccountResourceID'], 2015-05-01-preview'].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')), '2015-05-01').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.

Troubleshooting Key Vault References

If a reference is not resolved properly, the reference value will be used instead. This means that for application settings, an environment variable would be created whose value has the `@Microsoft.KeyVault(...)` syntax. This may cause the application to throw errors, as it was expecting a secret of a certain structure.

Most commonly, this is 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. Below the setting configuration, you should see status information, including any errors. The absence of these implies that the reference syntax is invalid.

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. Find **Key Vault Application Settings Diagnostics** and click **More info**.

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. Click on **Key Vault Application Settings Diagnostics**.

Encrypt your application data at rest using customer-managed keys

11/2/2020 • 4 minutes to read • [Edit Online](#)

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.

```
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:

```
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:

```
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:

```
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:

```
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

11/2/2020 • 6 minutes to read • [Edit Online](#)

Azure Cosmos DB is a great way to store unstructured and JSON data. Combined with Azure Functions, 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. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

| SETTING | VALUE | DESCRIPTION |
|----------------|---------------------|---|
| Subscription | Subscription name | Select the Azure subscription that you want to use for this Azure Cosmos account. |
| Resource Group | Resource group name | Select a resource group, or select Create new , then enter a unique name for the new resource group. |

| Setting | Value | Description |
|--------------------------|--------------------------------------|---|
| Account Name | A unique name | <p>Enter a name to identify your Azure Cosmos account. Because <code>documents.azure.com</code> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.</p> |
| API | The type of account to create | <p>Select Core (SQL) to create a document database and query by using SQL syntax.</p> <p>The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.</p> |
| 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 Free Tier Discount | Apply or Do not apply | With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier . |
| 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. |
| Account Type | Production or Non-Production | Select Production if the account will be used for a production workload. Select Non-Production if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime. |

NOTE

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 do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

Microsoft Azure (Preview) Search resources, services, and docs (G+)

Home > New > Create Azure Cosmos DB Account

Basics Networking Backup Policy Encryption Tags Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. 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 * SubscriptionName

Resource Group * (New) my-resource-group [Create new](#)

Instance Details

Account Name * Enter account name

API * Core (SQL)

Notebooks (Preview) [On](#) [Off](#)

Location * (US) West US

Capacity mode [Provisioned throughput](#) [Serverless \(Preview\)](#) [Learn more about capacity mode](#)

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$ /month discount per account.

Apply Free Tier Discount [Apply](#) [Do Not Apply](#)

Account Type [Production](#) [Non-Production](#)

Geo-Redundancy [Enable](#) [Disable](#)

Multi-region Writes [Enable](#) [Disable](#)

[Review + create](#) [Previous](#) [Next: Networking](#)

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.
6. 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 for a function app named "Microsoft.Azure.CosmosDB-20190321000000". The deployment is marked as complete. Key details include:

- Deployment name: Microsoft.Azure.CosmosDB-20190321000000
- Subscription: Contoso Subscription
- Resource group: myResourceGroup

DEPLOYMENT DETAILS (Download):
Start time: 3/21/2019, 5:00:03 PM
Duration: 5 minutes 38 seconds
Correlation ID: 8e0be948-0c60-4da0-0000-000000000000

| RESOURCE | TYPE | STATUS | OPERATION DETAILS |
|------------------|-------------------------------------|--------|-----------------------------------|
| mysqlapicosmosdb | Microsoft.DocumentDb/databaseAcc... | OK | Operation details |

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the "Quick start" page for the Azure Cosmos DB account "mysqlapicosmosdb". It includes:

- A message: "Congratulations! Your Azure Cosmos DB account was created."
- Instructions: "Now, let's connect to it using a sample app:"
- A "Choose a platform" section with links for .NET, .NET Core, Xamarin, Java, Node.js, and Python.
- Step 1: "Add a collection" - A note: "In Azure Cosmos DB, data is stored in collections." with a "Create 'Items' collection" button.
- Step 2: "Download and run your .NET app" - A note: "Once collection is created, download a sample .NET app connected to it, extract, build and run." with a "Download" button.

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

Search (Ctrl+ /) <> Add Develop Locally Refresh Enable Disable Delete

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Security

Functions (preview)

f Functions (highlighted with red box)

App keys App files

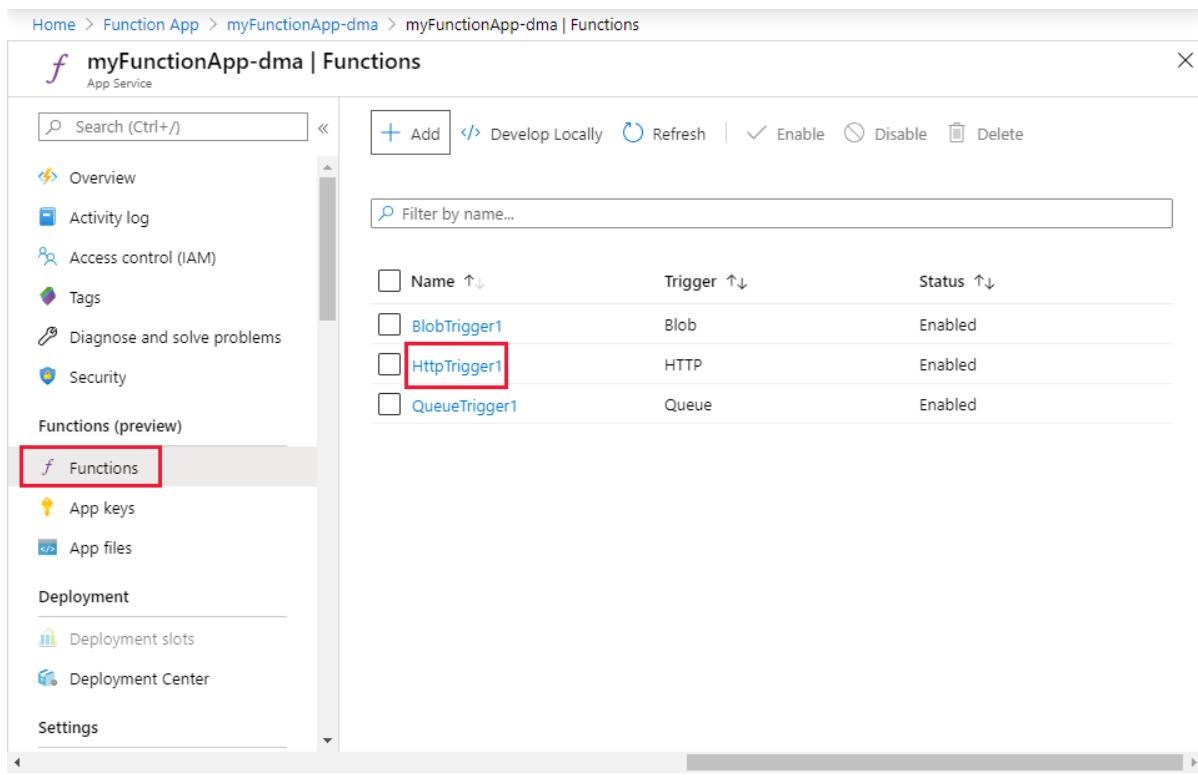
Deployment

Deployment slots Deployment Center

Settings

Filter by name...

| Name ↑↓ | Trigger ↑↓ | Status ↑↓ |
|--|------------|-----------|
| BlobTrigger1 | Blob | Enabled |
| HttpTrigger1 (highlighted with red box) | HTTP | Enabled |
| QueueTrigger1 | Queue | Enabled |



3. Select **Integration** and **+ Add output**.

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

HttpTrigger1 (myFunctionApp-dma/HttpTrigger1) | Integration

Function

Search (Ctrl+ /) <>

Overview Developer

Code / Test (highlighted with red box)

Integration (highlighted with red box)

Monitor Function Keys

Trigger

HTTP (req)

Inputs

No inputs defined + Add input

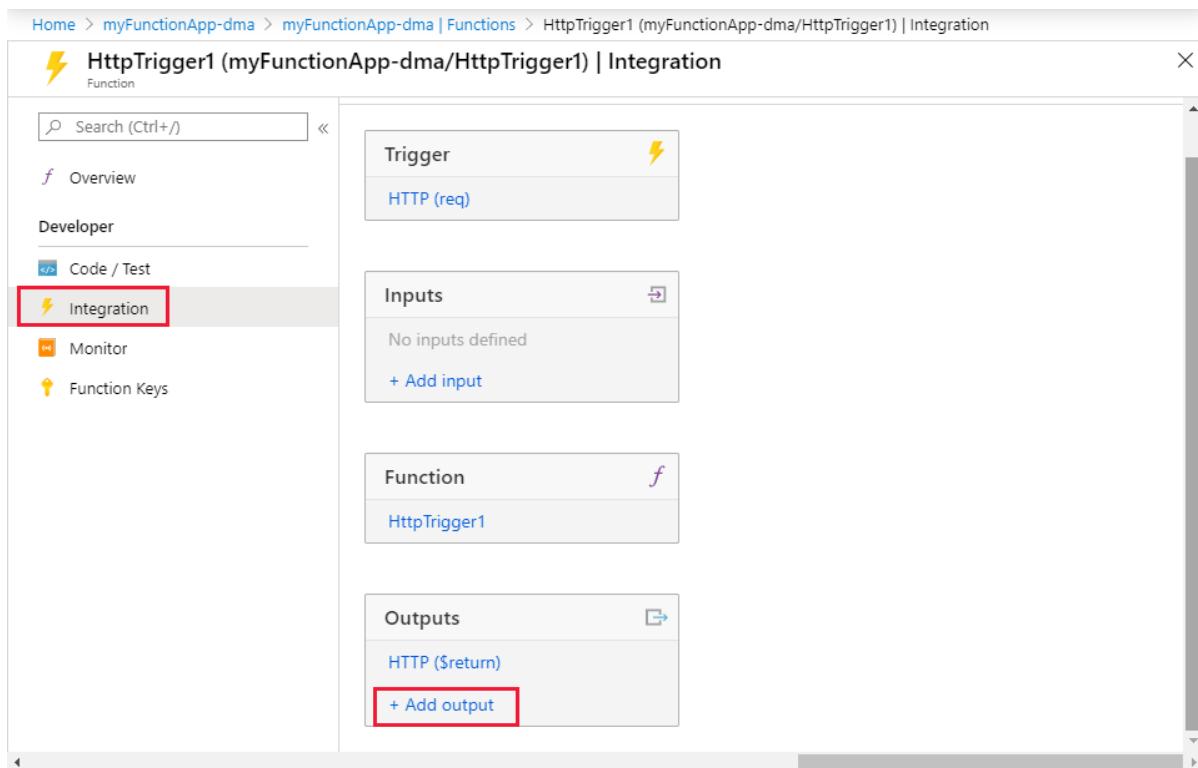
Function

HttpTrigger1

Outputs

HTTP (\$return)

+ Add output (highlighted with 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...* ⓘ

No

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 Cosmos DB object in code. |
| Database name | taskDatabase | Name of database to save documents. |
| Collection name | taskCollection | Name of the database collection. |
| If true, creates the Cosmos DB database and collection | Yes | The collection doesn't already exist, so create it. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|------------------------------|-----------------|--|
| 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#](#)
- [JavaScript](#)

Replace the existing C# function with the following code:

```
#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

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 | Code + Test

HttpTrigger1 | Code + Test

Function

Save Discard Refresh ...

myFunctionApp-dma \ HttpTrigger1 \ run.csx

Developer

Code + Test (selected)

Integration

Monitor

Function Keys

```

3  using Microsoft.AspNetCore.Mvc;
4  using Microsoft.AspNetCore.Http;
5  using Microsoft.Extensions.Logging;
6
7  public static IActionResult Run(HttpContext r)
8  {
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     }
}

```

Log Level Stop Copy Clear

Connected!

Input Output

Provide parameters to test the HTTP request. Results can be found in the Output tab.

HTTP method: POST

Key: master (Host key)

Query

| Name | Value |
|---------|--------------|
| name | Maria Anders |
| dueDate | 03/17/2017 |
| task | shopping |

+ Add parameter

Headers

+ Add header

2. Select Run and verify that a 200 status is returned.

Home > myFunctionApp-dma > myFunctionApp-dma | Functions > HttpTrigger1 | Code + Test

HttpTrigger1 | Code + Test

Function

Save Discard Refresh Get function URL

myFunctionApp-dma \ HttpTrigger1 \ run.csx

Developer

Code + Test (selected)

Integration

Monitor

Function Keys

```

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 }

```

Logs

Connected!

2020-04-14T05:10:00Z [Information] Executed 'Functions.HttpTrigger1' (Succeeded, Id=ce811ba9-dfba-42d1-a0cb-a7caeed0efc5)

2020-04-14T05:09:59Z [Information] Executing 'Functions.HttpTrigger1' (Reason='This function was programmatically called via the host APIs.', Id=ce811ba9-dfba-42d1-a0cb-a7caeed0efc5)

Input Output

HTTP response code: 200 OK

HTTP response content:

3. In the Azure portal, search for and select Azure Cosmos DB.

Azure Cosmos DB

Services

- 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.

Home > Resource groups > portal-testing > glengatest-portal | Data Explorer

glengatest-portal | Data Explorer

Items

```

SELECT * FROM c
Edit Filter
id / p...
32be...
ec24...
9c03...
7d43...
566e...
2c2f...
0435...
6fe4...
Load more
1
2
3
4
5
6
7
8
9
10
11
{
  "name": "Marie",
  "duedate": "4/20/20",
  "task": "shopping",
  "id": "32bededa-2b0c-4f1e-8fc2-eae87f36b5a3",
  "_rid": "27zbAPhKu48BAAAAAAAA==",
  "_self": "dbs/27zbAPhKu48/colls/27zbAPhKu48/docs/27zbAPhKu48/_id",
  "_etag": "\\"1b011ac8-0000-0500-0000-5e95426b0000\\",
  "_attachments": "attachments/",
  "_ts": 1566840171
}

```

You've successfully added a binding to your HTTP trigger to store unstructured data in an Azure Cosmos DB.

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 a Cosmos DB database, see [Azure Functions 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.

Connect Azure Functions to Azure Storage using command line tools

12/4/2020 • 16 minutes to read • [Edit Online](#)

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.

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.

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.

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.

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.

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 use by the function app. 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 that connection 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, replacing `<app_name>` with the name of your function app from the previous quickstart. This command will overwrite any existing values in the file.

```
func azure functionapp fetch-app-settings <app_name>
```

2. Open `local.settings.json` 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 `local.settings.json` 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

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage --version 3.0.4
```

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 let 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:

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "res"
  }
]
```

```
"scriptFile": "__init__.py",
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  }
]
```

```
"bindings": [
  {
    "authLevel": "function",
    "type": "httpTrigger",
    "direction": "in",
    "name": "Request",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "Response"
  }
]
```

Each binding has at least a type, a direction, and a name. In the example above, the first binding is of type `httpTrigger` with the direction `in`. For the `in` direction, `name` specifies the name of an input parameter that's sent to the function when invoked by the trigger.

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:

```
{
  "authLevel": "function",
  "type": "httpTrigger",
  "direction": "in",
  "name": "req",
  "methods": [
    "get",
    "post"
  ],
},
{
  "type": "http",
  "direction": "out",
  "name": "res"
},
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
]
```

The second binding in the collection is of type `http` with the direction `out`, in which case the special `name` of `$return` indicates that this binding uses the function's return value rather than providing an input parameter.

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:

```
"bindings": [
  {
    "authLevel": "anonymous",
    "type": "httpTrigger",
    "direction": "in",
    "name": "req",
    "methods": [
      "get",
      "post"
    ]
  },
  {
    "type": "http",
    "direction": "out",
    "name": "$return"
  },
  {
    "type": "queue",
    "direction": "out",
    "name": "msg",
    "queueName": "outqueue",
    "connection": "AzureWebJobsStorage"
  }
]
```

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:

```
{
  "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"
}
]
```

In this case, `msg` is given to the function as an output argument. For a `queue` type, you must also specify the name of the queue in `queueName` and provide the *name* of the Azure Storage connection (from `local.settings.json`) in `connection`.

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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.

Update `HttpExample_init_.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```
import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )
```

The `msg` parameter is an instance of the `azure.functions.InputStream class`. Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};
```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```
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
})
```

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```
if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}
```

At this point, your function should look as follows:

```

[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

```
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:

```

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.

```
@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:

```
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:

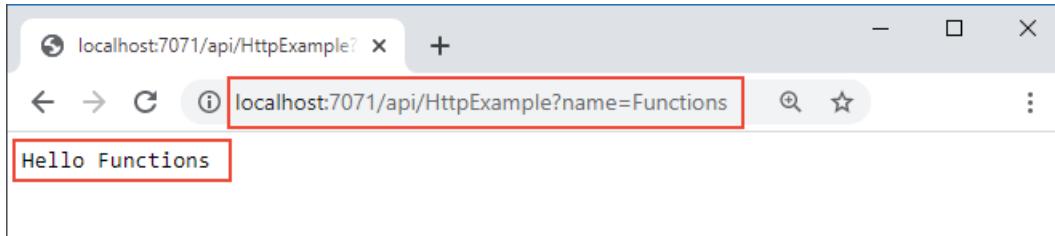
    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
...
```

NOTE

If `HttpExample` doesn't appear as shown below, 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`:



3. The terminal in which you started your project also shows log output as you make requests.

4. When you're done, use `Ctrl+C` and choose `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:

```
{  
    "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`, pasting 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](#)
- [PowerShell](#)
- [Azure CLI](#)

```
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 should include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

```
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 first name you used when testing the function earlier. The command reads and removes the first message from the queue.

- [bash](#)
- [PowerShell](#)
- [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>
```

In the local project folder, use the following Maven command to republish your project:

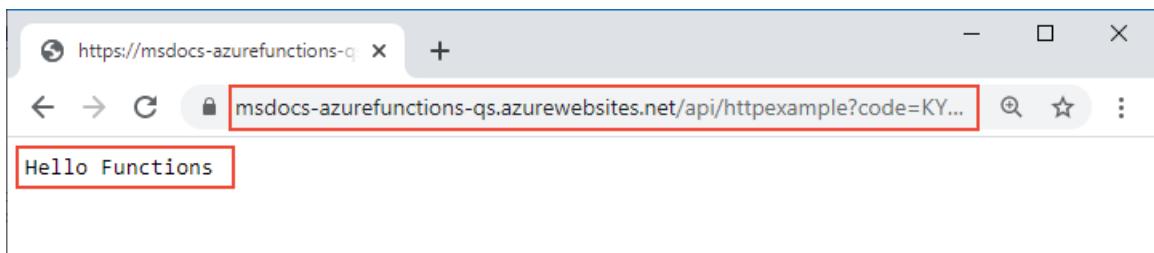
```
mvn azure-functions:deploy
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

- [Browser](#)
- [curl](#)

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.



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.

```
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
- Examples of complete Function projects in JavaScript.
- Azure Functions JavaScript developer guide
- Examples of complete Function projects in TypeScript.
- Azure Functions TypeScript developer guide
- Examples of complete Function projects in Python.
- Azure Functions Python developer guide
- Examples of complete Function projects in PowerShell.
- Azure Functions PowerShell developer guide

Add messages to an Azure Storage queue using Functions

11/2/2020 • 5 minutes to read • [Edit Online](#)

In Azure Functions, input and output bindings provide a declarative way to make data from external services available to your code. In this quickstart, you use an output binding to create a message in a queue when a function is triggered by an HTTP request. You use Azure storage container to view the queue messages that your function creates.

Prerequisites

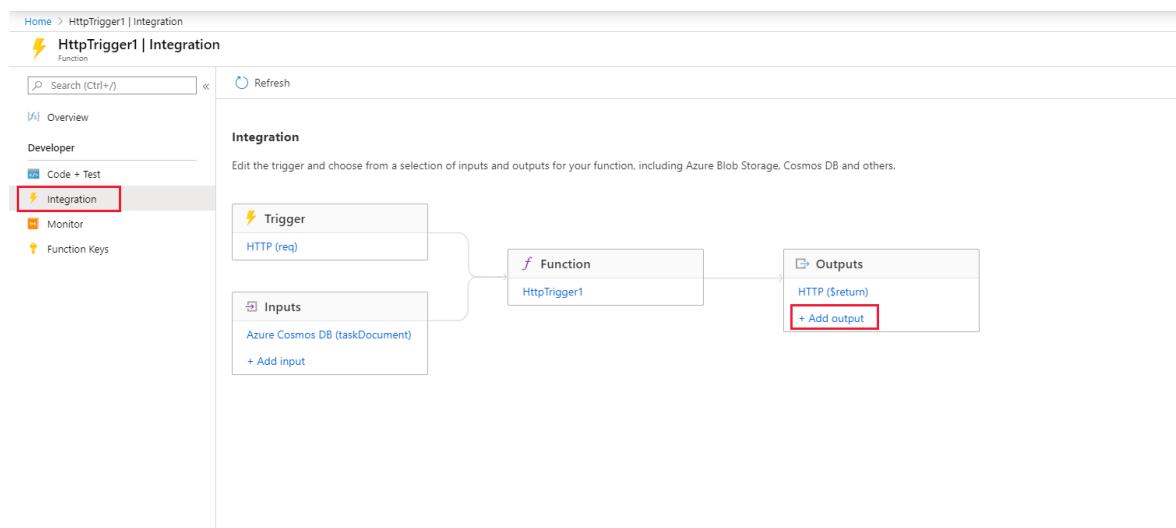
To complete this quickstart:

- An Azure subscription. If you don't have one, create a [free account](#) before you begin.
- Follow the directions in [Create your first function from the Azure portal](#) and don't do the **Clean up resources** step. That quickstart creates the function app and function that you use here.

Add an output binding

In this section, you use the portal UI to add a queue storage output binding to the function you created earlier. This binding makes it possible to write minimal code to create a message in a queue. You don't have to write code for tasks such 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, open the function app page for the function app that you created in [Create your first function from the Azure portal](#). To do open the page, search for and select **Function App**. Then, select your function app.
2. Select your function app, and then select the function that you created in that earlier quickstart.
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

X

Start by selecting the type of output binding you want to add.

Binding Type

Azure Queue Storage

Azure Queue Storage details

Message parameter name* ⓘ

outputQueueItem

Queue name* ⓘ

outqueue

Storage account connection* ⓘ

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 storage account connection already being 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 that is passed to the HTTP trigger in the query string. For example, if the query string includes `name=Azure`, the queue message will be *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 depending on your function language:

- C#
- JavaScript

Add an **outputQueueItem** parameter to the method signature as shown in the following example.

```
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.

```
outputQueueItem.Add("Name passed to the function: " + name);
```

3. Select **Save** to save changes.

Test the function

1. After the code changes are saved, select **Test**.
2. Confirm that your test matches the image below and select **Run**.

The screenshot shows the Azure Functions portal interface. The left sidebar has 'Overview', 'Developer', and 'Code + Test' selected. The main area shows the code for 'HttpTrigger1' in 'run.csx'. The 'Test' tab is highlighted. In the 'Input' tab of the test configuration, there is a JSON object with a single key-value pair: "name": "Azure". The 'Run' button at the bottom right of the test configuration panel is also highlighted with a red box.

Notice that the **Request body** contains the `name` value *Azure*. This value appears in the queue message that is created when the function is invoked.

As an alternative to selecting **Run** here, you can call the function by entering a URL in a browser and specifying the `name` value in the query string. The browser method is shown in the [previous quickstart](#).

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'll use storage account to verify that the queue and a message in it were created.

Find the storage account connected to AzureWebJobsStorage

1. Go to your function app and select **Configuration**.
2. Under **Application settings**, select **AzureWebJobsStorage**.

The screenshot shows the Azure Functions configuration interface for a function named 'HttpTrigger1'. The left sidebar lists various settings like Overview, Activity log, and Functions. The 'Configuration' section is selected and highlighted with a red box. The main area displays application settings in a table format. One row, 'AzureWebJobsStorage', is also highlighted with a red box.

| Name | Value | Source | Deployment slot |
|---------------------------------------|--|------------|-----------------|
| APPINSIGHTS_INSTRUMENTATIONKEY | <small>Hidden value. Click to show value</small> | App Config | |
| APPLICATIONINSIGHTS_CONNECTION_STRING | <small>Hidden value. Click to show value</small> | App Config | |
| AzureWebJobsStorage | <small>Hidden value. Click to show value</small> | App Config | |
| FUNCTIONS_EXTENSION_VERSION | <small>Hidden value. Click to show value</small> | App Config | |
| FUNCTIONS_WORKER_RUNTIME | <small>Hidden value. Click to show value</small> | App Config | |
| storageaccountmyres81bd_STORAGE | <small>Hidden value. Click to show value</small> | App Config | |
| storageaccountmyres8409_STORAGE | <small>Hidden value. Click to show value</small> | App Config | |
| WEBSITE_CONTENTAZUREFILECONNECTION: | <small>Hidden value. Click to show value</small> | App Config | |
| WEBSITE_CONTENTSHARE | <small>Hidden value. Click to show value</small> | App Config | |

3. Locate and make note of the account name.

The screenshot shows the 'Add/Edit application setting' dialog. The 'Name' field is set to 'AzureWebJobsStorage'. The 'Value' field contains a connection string: 'DefaultEndpointsProtocol=https;AccountName=storageaccountmyresbcb9;AccountKey=/fLt9Wtr+z/30JESevURlqh4aiO8...'. A red box highlights the 'Value' field. There is a checkbox for 'Deployment slot setting' which is unchecked.

Examine the output queue

1. In the resource group for your function app, select the storage account that you're using for this quickstart.
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, and you'll see a new message appear in the queue.

Clean up resources

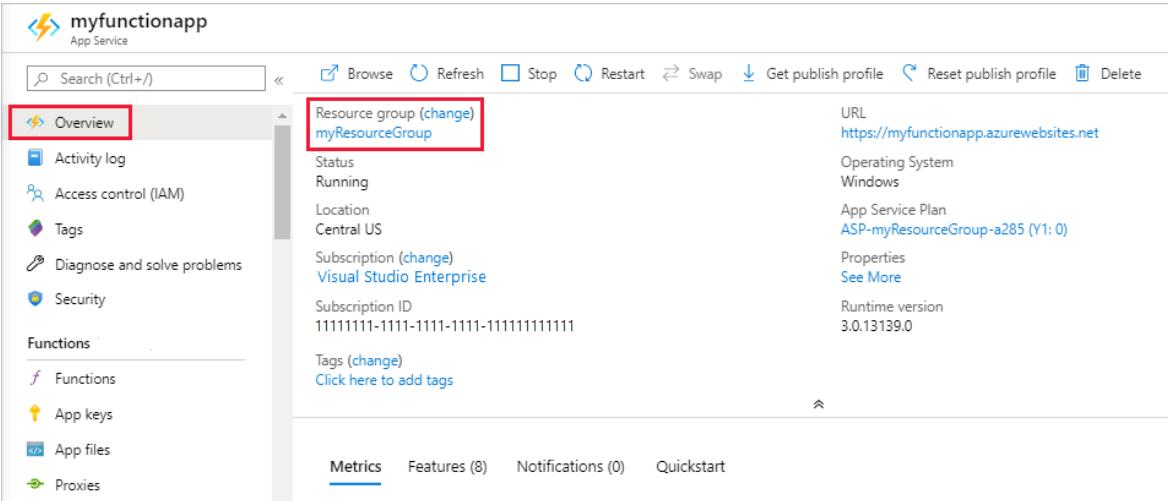
Other quickstarts in this collection build upon this quickstart. If you plan to work with subsequent quickstarts, tutorials, or with any of the services you have created in this quickstart, do not clean up the resources.

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 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 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 "Resource group" section with a red box around it, showing "myResourceGroup". The main content area has a "Resource group (change)" section with a red box around it, showing "myResourceGroup". The "Overview" tab is also highlighted with a red box. The right side of the screen displays various details about the app, such as Status (Running), Location (Central US), Subscription (Visual Studio Enterprise), and URL (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 quickstart.

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 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 added an output binding to an existing function. For more information about binding to Queue storage, see [Azure Functions Storage queue 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.

Connect Azure Functions to Azure Storage using Visual Studio Code

12/4/2020 • 16 minutes to read • [Edit Online](#)

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 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 start this article, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool 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](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).
- 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 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 the F1 key to open the command palette, then search for and run the command

- Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

IMPORTANT

Because it contains secrets, the local.settings.json file never gets published, and is excluded from source control.

- 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 usage is enabled in the host.json file at the root of the project, which appears as follows:

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

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 Storage extension package to your project.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.Storage
```

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 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.json file. 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:

| PROMPT | VALUE | DESCRIPTION |
|--------------------------|-------|-----------------------------------|
| Select binding direction | out | The binding is an output binding. |

| PROMPT | VALUE | DESCRIPTION |
|---|---------------------|--|
| Select binding with direction... | Azure Queue Storage | The binding is an Azure Storage queue binding. |
| The name used to identify this binding in your code | msg | Name that identifies the binding parameter referenced in your code. |
| The queue to which the message will be sent | outqueue | 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" | AzureWebJobsStorage | 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:

```
{
  "type": "queue",
  "direction": "out",
  "name": "msg",
  "queueName": "outqueue",
  "connection": "AzureWebJobsStorage"
}
```

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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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:

```
@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:

```
@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.

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this code before the `context.res` statement.

```
context.bindings.msg = (req.query.name || req.body.name);
```

At this point, your function should look as follows:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        // Add a message to the Storage queue,
        // which is the name passed to the function.
        context.bindings.msg = (req.query.name || req.body.name);
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};
```

Add code that uses the `msg` output binding object on `context.bindings` to create a queue message. Add this

code before the `context.res` statement.

```
context.bindings.msg = name;
```

At this point, your function should look as follows:

```
import { AzureFunction, Context, HttpRequest } from "@azure/functions"

const httpTrigger: AzureFunction = async function (context: Context, req: HttpRequest): Promise<void> {
    context.log('HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));

    if (name) {
        // Add a message to the storage queue,
        // which is the name passed to the function.
        context.bindings.msg = name;
        // Send a "hello" response.
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
};

export default httpTrigger;
```

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.

```
$outputMsg = $name
Push-OutputBinding -name msg -Value $outputMsg
```

At this point, your function should look as follows:

```

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
})

```

Update `HttpExample__init__.py` to match the following code, adding the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement.

```

import logging

import azure.functions as func


def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> str:

    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}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )

```

The `msg` parameter is an instance of the `azure.functions.InputStream class`. Its `set` method writes a string message to the queue, in this case the name passed to the function in the URL query string.

Add code that uses the `msg` output binding object to create a queue message. Add this code before the method returns.

```
if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}
```

At this point, your function should look as follows:

```
[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}
```

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.

```
// Write the name to the message queue.
```

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:

```

@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.

```

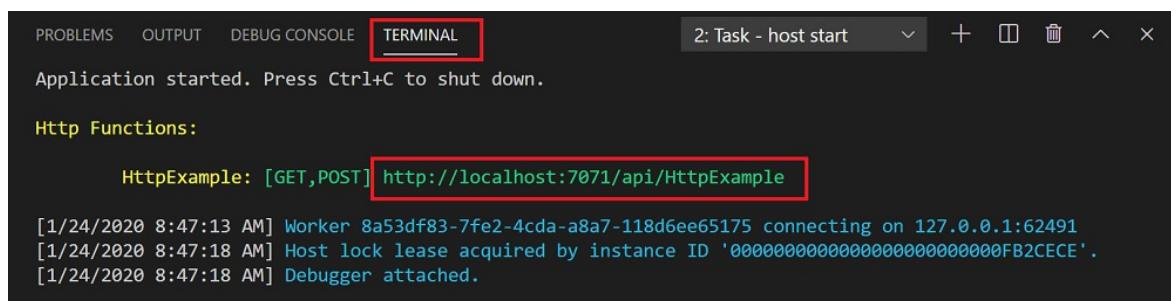
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);

```

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 call your function, press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. If you haven't already installed Azure Functions Core Tools, select **Install** at the prompt. When the Core Tools are installed, 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
2: Task - host start
Application started. Press Ctrl+C to shut down.

Http Functions:

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

[1/24/2020 8:47:13 AM] Worker 8a53df83-7fe2-4cda-a8a7-118d6ee65175 connecting on 127.0.0.1:62491
[1/24/2020 8:47:18 AM] Host lock lease acquired by instance ID '00000000000000000000000000FB2CECE'.
[1/24/2020 8:47:18 AM] Debugger attached.

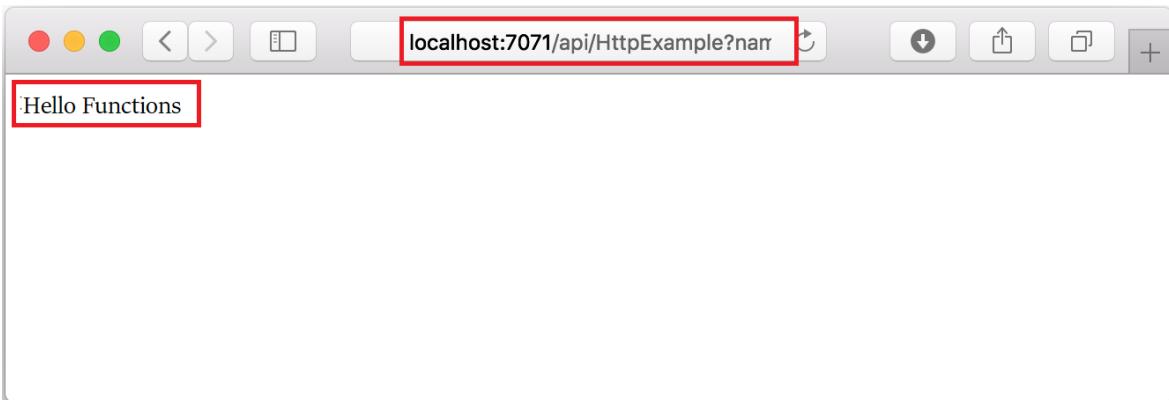
```

3. 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

4. A response is returned, which looks like the following in a browser:



5. 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 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] }
```

6. Press **Ctrl + C** to stop Core Tools and disconnect the debugger.

Run the function locally

Azure Functions Core Tools integrates with Visual Studio Code to let you run and debug an Azure Functions project locally. For details on how to debug in Visual Studio Code, see [Debug PowerShell Azure Functions locally](#).

1. Press **F5** to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.
2. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 4: Task - host start + □ ×
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

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

[4/20/2019 6:19:06 AM] System Log: {
[4/20/2019 6:19:06 AM]   Log-Message: The enforced concurrency level (pool size limit) is '1'.
[4/20/2019 6:19:06 AM] }
[4/20/2019 6:19:06 AM] System Log: {

functions (MyFunctionProj) 🔥 Ln 10, Col 28 (27 selected) Spaces: 4 UTF-8 LF PowerShell 📦 6.2 😊 🚙 1
```

- Append the query string `?name=<yourname>` to this URL, and then use `Invoke-RestMethod` in a second PowerShell command prompt to execute the request, as follows:

```
PS > Invoke-RestMethod -Method Get -Uri http://localhost:7071/api/HttpTrigger?name=PowerShell
Hello PowerShell
```

You can also execute the GET request from a browser from the following URL:

```
http://localhost:7071/api/HttpExample?name=PowerShell
```

When you call the `HttpTrigger` endpoint without passing a `name` parameter either as a query parameter or in the body, the function returns a `BadRequest` error. When you review the code in `run.ps1`, you see that this error occurs by design.

- Information about the request is shown in `Terminal` panel.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Task - host start + □ ×
[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 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] }
```

- When done, press `Ctrl + C` to stop Core Tools.

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

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.

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.

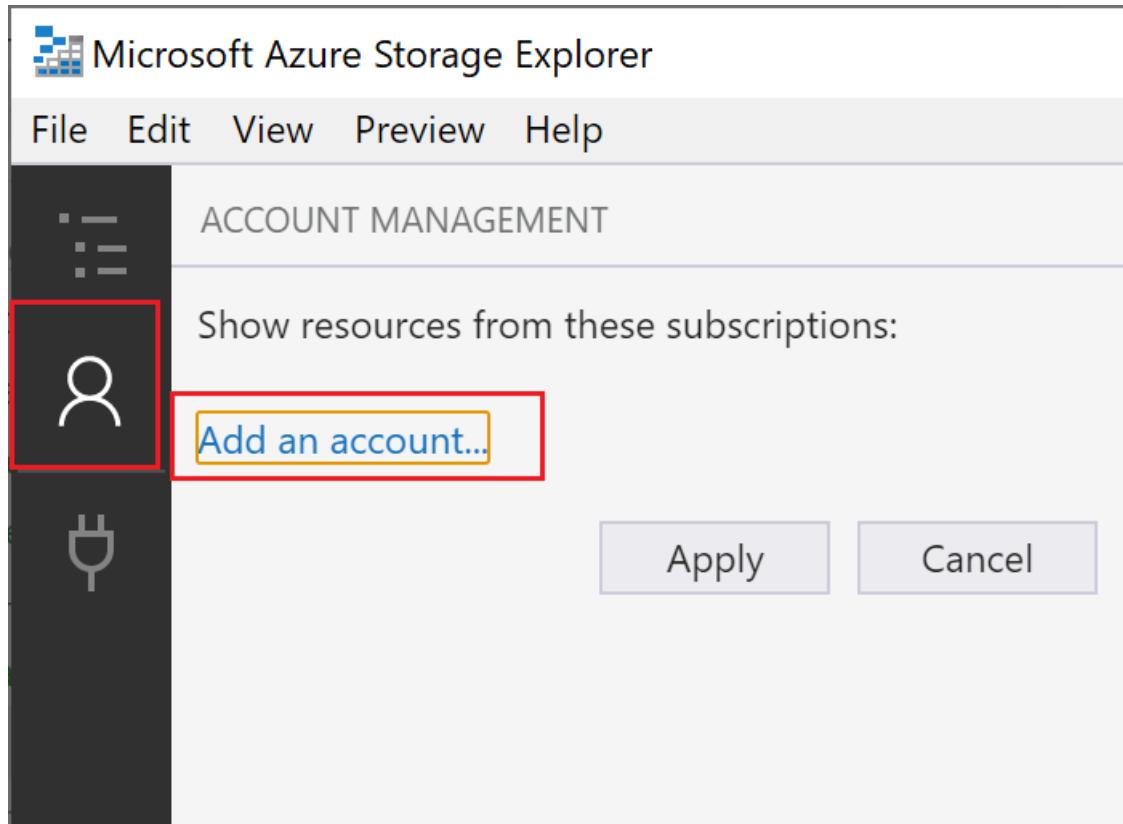
Browse to the location of your test code under `src/test/java`, open the `Function.java` project file, and replace the line of code under `//Invoke` with the following code.

```
@SuppressWarnings("unchecked")
final OutputBinding<String> msg = (OutputBinding<String>)mock(OutputBinding.class);
final HttpResponseMessage ret = new Function().run(req, msg, context);
```

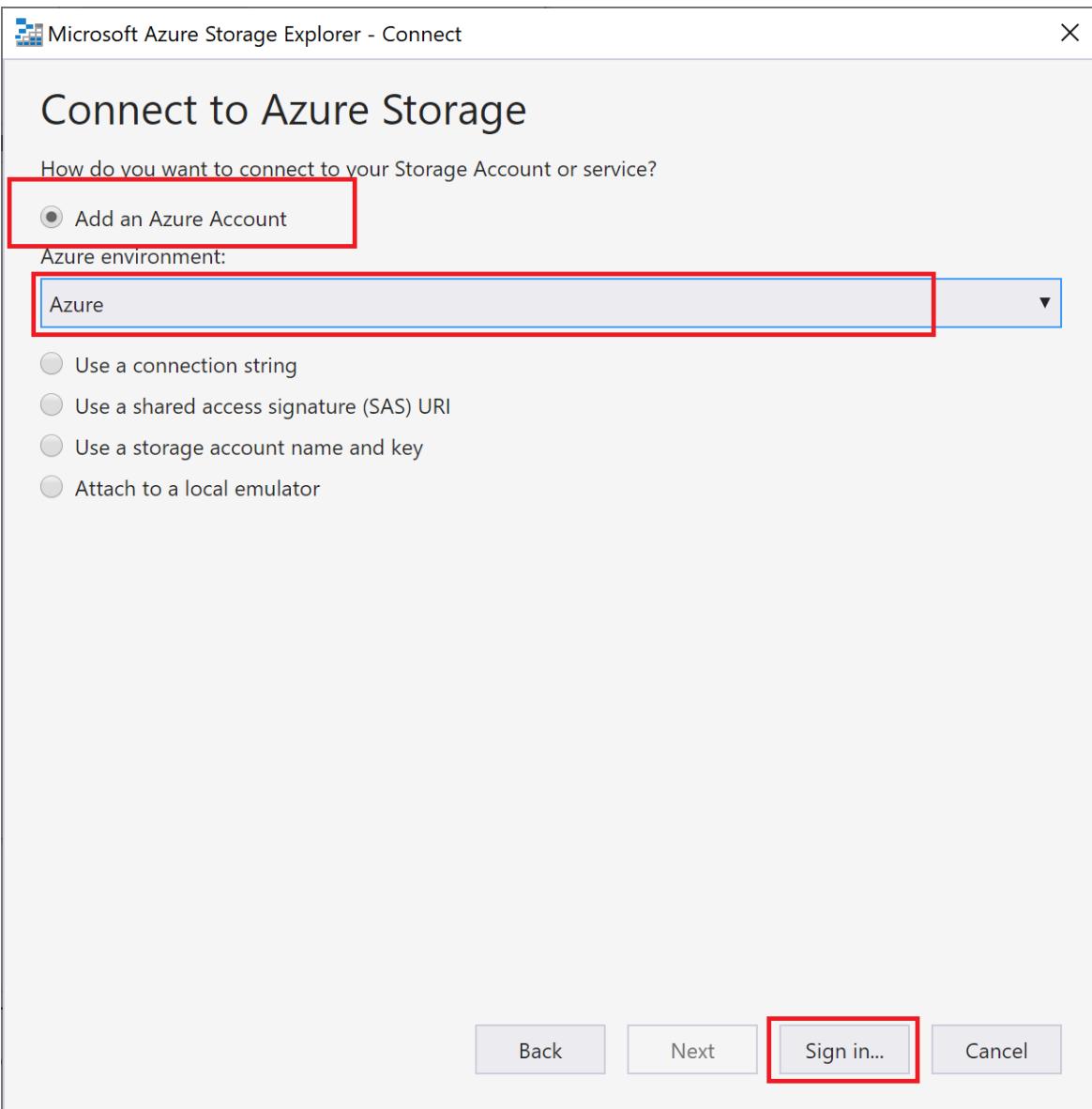
Connect Storage Explorer to your account

Skip this section if you have 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 select **Sign in....**

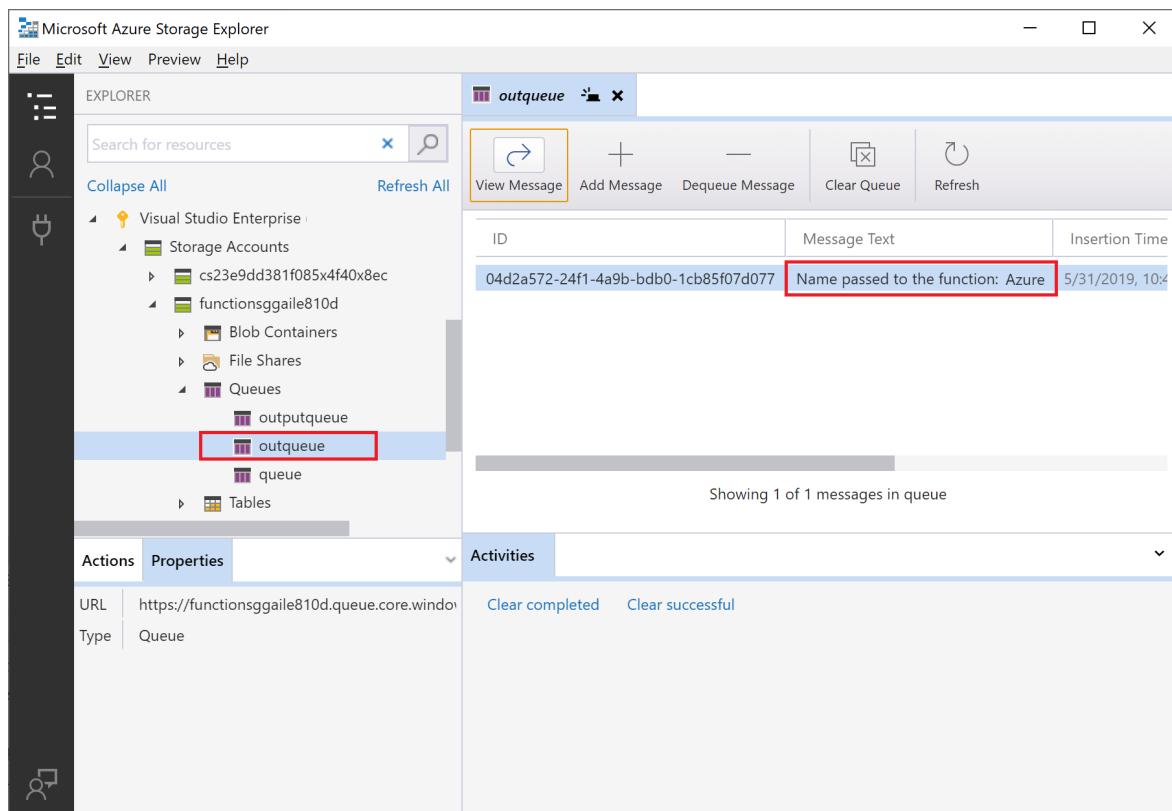


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account.

Examine the output queue

1. In Visual Studio Code, press the F1 key 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 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'll see a new message appear 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 deployment completes, you can again use cURL or a browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL, as in the following example:

```
curl https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....&name=<yourname>
```

- Again [view the message in the storage queue](#) to verify that the output binding again generates a new message in the queue.

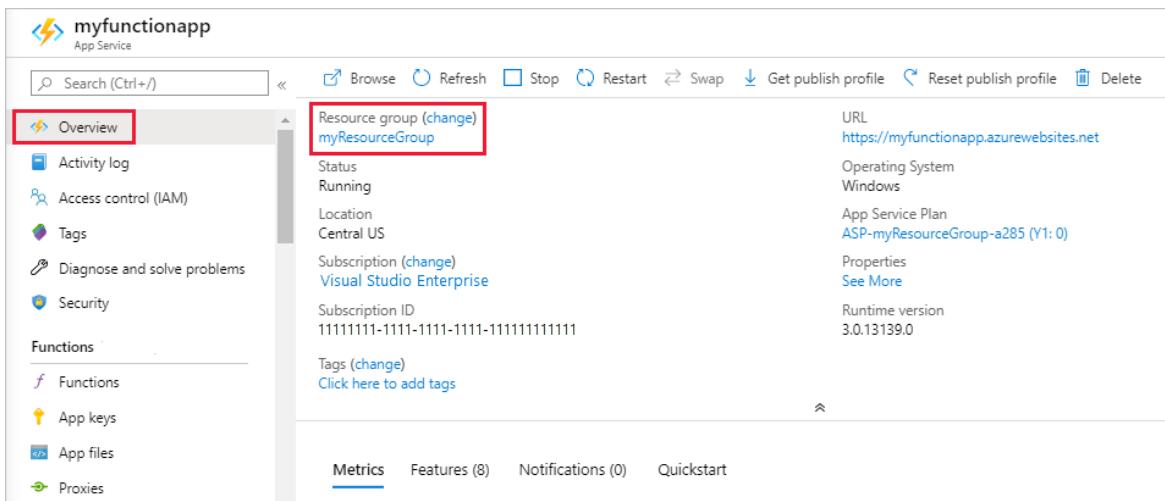
Clean up resources

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:

- In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Open in portal`.
- 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 'Resource group (change)' link under the 'myResourceGroup' section, which is highlighted with a red box. The main content area displays various details about the app service, including its status (Running), location (Central US), subscription (Visual Studio Enterprise), and runtime version (3.0.13139.0). At the bottom, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

4. In the **Resource group** page, review the list of included resources, and verify that they are 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](#)
- [Examples of complete Function projects in JavaScript](#).
- [Azure Functions JavaScript developer guide](#)
- [Examples of complete Function projects in Java](#).
- [Azure Functions Java developer guide](#)
- [Examples of complete Function projects in TypeScript](#).
- [Azure Functions TypeScript developer guide](#)
- [Examples of complete Function projects in Python](#).
- [Azure Functions Python developer guide](#)
- [Examples of complete Function projects in PowerShell](#).
- [Azure Functions PowerShell developer guide](#)

Connect functions to Azure Storage using Visual Studio

12/4/2020 • 6 minutes to read • [Edit Online](#)

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

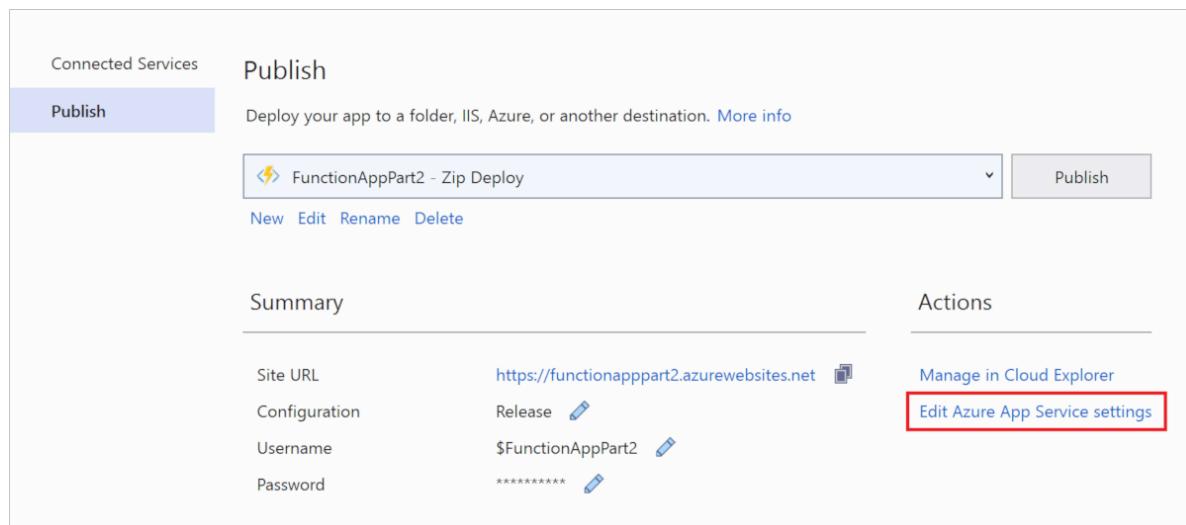
Before you start this article, you must:

- Complete [part 1 of the Visual Studio quickstart](#).
- 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. Under **Actions**, select **Edit 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:

```
Install-Package Microsoft.Azure.WebJobs.Extensions.Storage -Version 3.0.6
```

Now, you can add the storage 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. 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 parameter to the `Run` method definition:

```
[Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] ICollector<string> msg,
```

The `msg` parameter is an `ICollector<T>` type, which represents a collection of messages that are written 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 `StorageAccountAttribute`. This attribute indicates the setting that contains the Storage account connection string and can be applied at the class, method, or parameter level. In this case, you could omit `StorageAccountAttribute` because you are already using the default storage account.

The `Run` method definition should now look like the following:

```
[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)
```

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 `msg` output binding object to create a queue message. Add this code before the method returns.

```

if (!string.IsNullOrEmpty(name))
{
    // Add a message to the output collection.
    msg.Add(string.Format("Name passed to the function: {0}", name));
}

```

At this point, your function should look as follows:

```

[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)
{
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(string.Format("Name passed to the function: {0}", name));
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}

```

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.

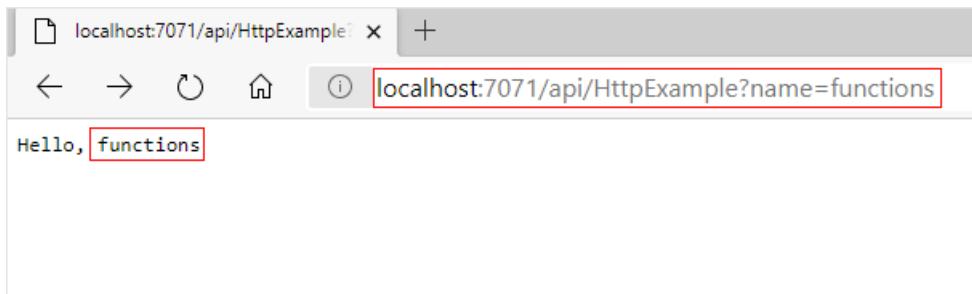
Http Functions:

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

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

```

3. Paste the URL for the HTTP request into your browser's address bar. Append the query string `?name=<YOUR_NAME>` to this URL and run the request. The following image shows the response in the browser to the local GET request returned by the function:



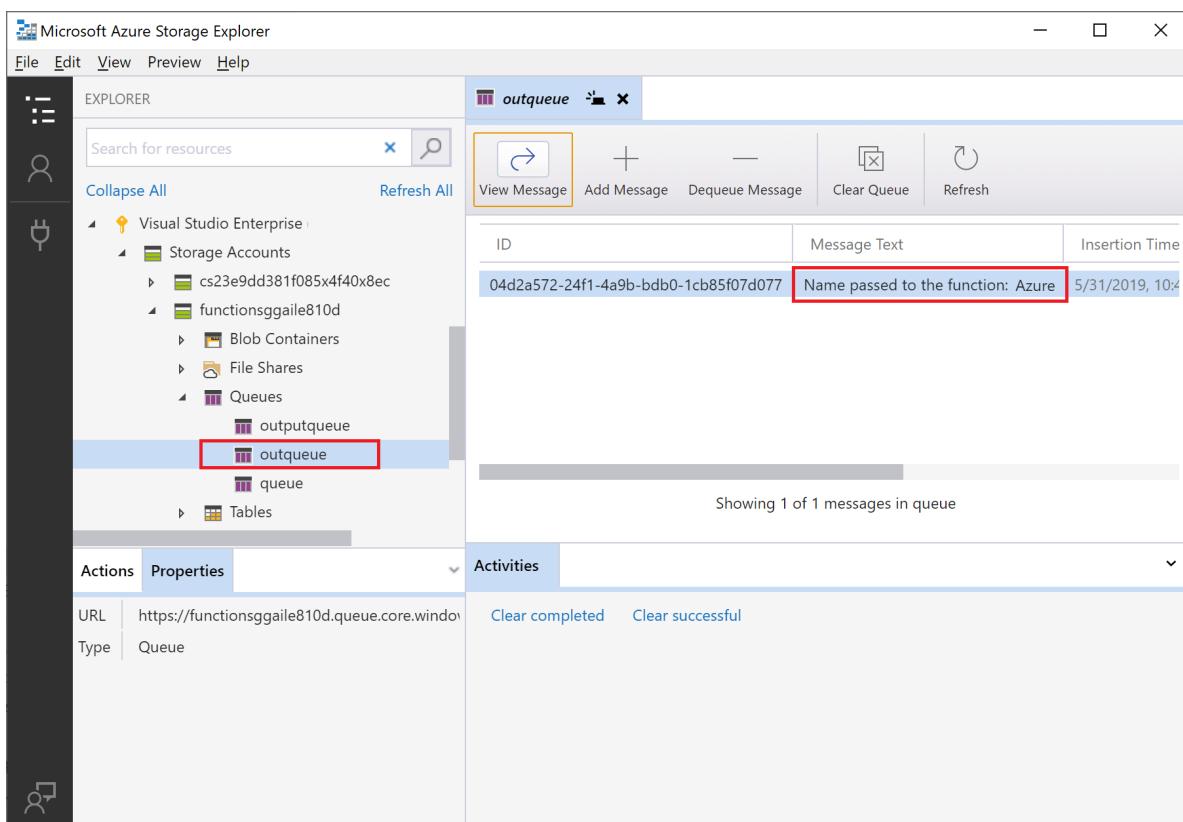
4. To stop debugging, press Shift+F5 in Visual Studio.

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 Cloud Explorer to verify that the queue was created along with the new message.

Examine the output queue

1. In Visual Studio from the **View** menu, select **Cloud Explorer**.
2. In **Cloud Explorer**, expand your Azure subscription and **Storage Accounts**, then expand the storage account used by your function. If you can't remember the storage account name, check the `AzureWebJobsStorage` connection string setting in the `/local.settings.json` file.
3. Expand the **Queues** node, and then double-click the queue named **outqueue** to view the contents of the queue in Visual Studio.

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*.



4. Run the function again, send another request, and you'll see a new message appear in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

1. In **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
2. After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourusername>` to the URL.
3. 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 have created in this quickstart, do not 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 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 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**.

| Resource Group Details | Value |
|------------------------|--------------------------------------|
| Status | Running |
| Location | Central US |
| Subscription | Visual Studio Enterprise |
| Subscription ID | 11111111-1111-1111-1111-111111111111 |
| Tags | Click here to add tags |
| Metrics | |
| Features (0) | |
| Notifications (0) | |
| Quickstart | |

To get to that page from the dashboard, select **Resource groups**, and then select the resource group that you used for this quickstart.

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 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. 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](#)

Use Azure Functions to connect to an Azure SQL Database

11/2/2020 • 4 minutes to read • [Edit Online](#)

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](#).

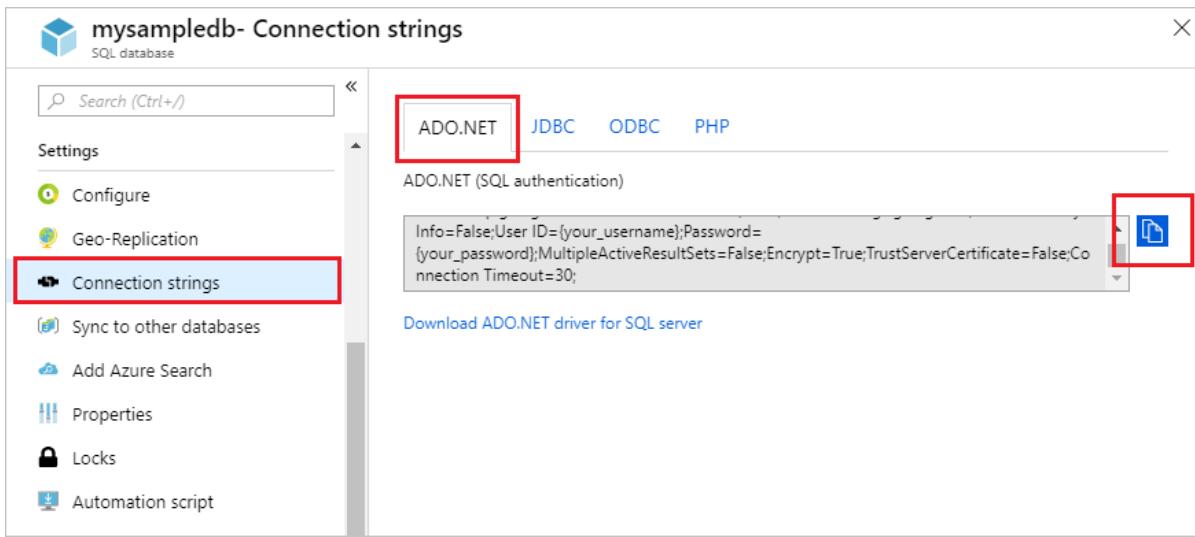
Prerequisites

- 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 > Edit Azure App Service settings**. Select **Add setting**, in **New app setting name**, type `sqldb_connection`, and select **OK**.

Publish

Deploy your app to a folder, IIS, Azure, or another destination. [More info](#)

⚡ SqlConnectionArticle20200224120443 - Zip Deploy ▼ Publish

New Edit Rename Delete

Summary

Site URL <https://sqlconnectionarticle20200224120443.azurewebsites.net> Manage in Cloud Explorer

Configuration Release Edit Azure App Service settings

Actions

Application Settings

FUNCTIONS_WORKER_RUNTIME

| Setting | Local | Remote |
|------------------------------|-------|--|
| New App Setting Name | do | sqldb_connection ③ |
| WEBSITE_CONTENTSHARE | do | do |
| WEBSITE_CONFIG | do | do |
| WEBSITE_DNSRECORDS | do | do |
| WEBSITE_LOADBALANCER | do | do |
| WEBSITE_OPENMETRICS | do | do |
| WEBSITE_PUBLISHKEY | do | do |
| WEBSITE_RUMCONFIG | do | do |
| WEBSITE_SCALINGPOLICIES | do | do |
| WEBSITE_SITESCALEINTEGRATION | do | do |
| WEBSITE_SKINS | do | do |
| WEBSITE_VMSIZE | do | do |

WEBSITE_CONTENTSHARE

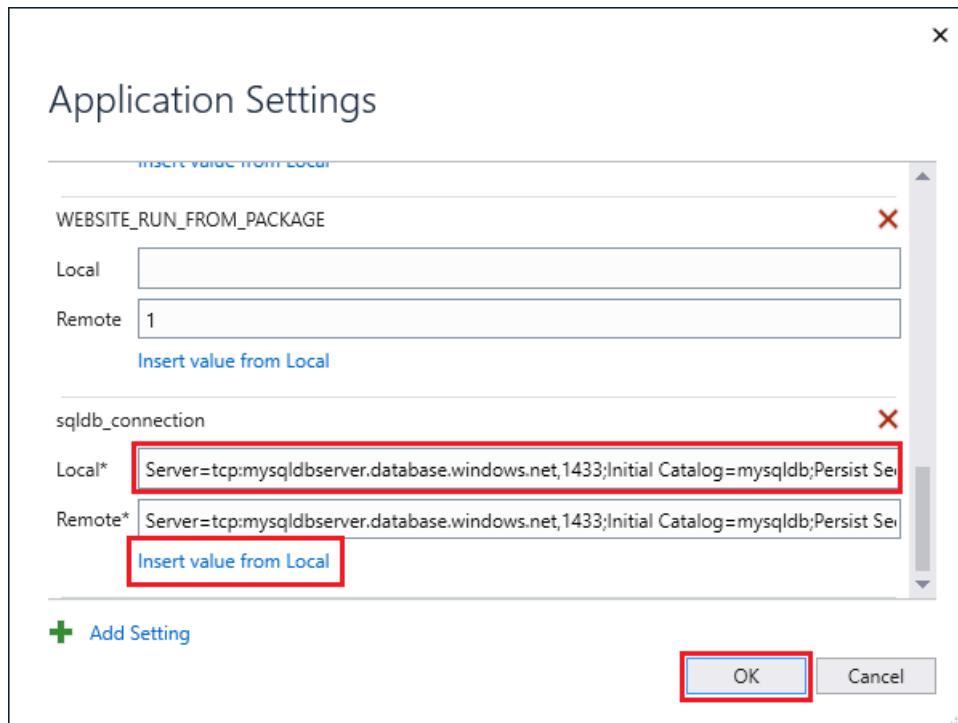
② Add Setting

OK Cancel

④ OK Cancel

The screenshot shows the 'Application Settings' dialog in the Azure portal. A new setting named 'sqldb_connection' is being added under the 'FUNCTIONS_WORKER_RUNTIME' section. The 'Local' field contains 'do' and the 'Remote' field contains 'sqldb_connection'. Step 3 is highlighted with a red box around the 'sqldb_connection' input field. Step 4 is highlighted with a red box around the 'OK' button at the bottom right of the dialog.

2. 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 2019.
2. In Solution Explorer, right-click the function app project and choose **Manage NuGet Packages**.
3. On the **Browse** tab, search for `System.Data.SqlClient` and, when found, select it.
4. In the `System.Data.SqlClient` page, select version `4.5.1` 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 **OK**. 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:

```
using System.Data.SqlClient;
using System.Threading.Tasks;
```

5. Replace the existing `Run` function with the following code:

```
[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 '0000000000000000000000006C79E40E'.
[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.

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 Functions with an Azure virtual network

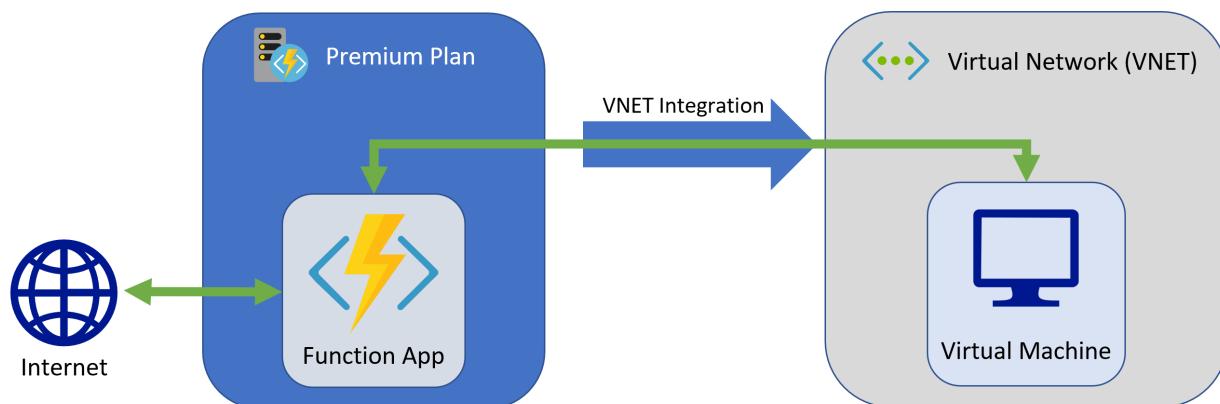
11/2/2020 • 9 minutes to read • [Edit Online](#)

This tutorial shows you how to use Azure Functions to connect to resources in an Azure virtual network. You'll create a function that has access to both the internet and to a VM running WordPress in virtual network.

- Create a function app in the Premium plan
- Deploy a WordPress site to VM in a virtual network
- Connect the function app to the virtual network
- Create a function proxy to access WordPress resources
- Request a WordPress file from inside the virtual network

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.

Create a function app in a Premium plan

First, you create a function app in the [Premium plan](#). This plan provides serverless scale while supporting virtual network integration.

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. Choose .NET for C# and F# functions. |
| Region | Preferred region | Choose a region near you or near other services your functions access. |

Function App X

[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 * ⓘ | Visual Studio Enterprise |
| Resource Group * ⓘ | (New) myResourceGroup |
| | Create new |

Instance Details

| | |
|---------------------|--|
| Function App name * | myfunctionapp |
| | .azurewebsites.net |
| Publish * | <input checked="" type="radio"/> Code <input type="radio"/> Docker Container |
| Runtime stack * | .NET Core |
| Region * | Central US |

[Review + create](#) [< Previous](#) [Next : Hosting >](#)

4. Select **Next: Hosting**. On the **Hosting** page, enter the following settings:

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|-----------------|-------------|
|---------|-----------------|-------------|

| 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. |
| Plan | Premium | <p>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.</p> <p>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.</p> |

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 * [Create new](#)

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * [Linux](#) [Windows](#)

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * ⓘ [Change](#)

[Cannot find your App service plan? Try a different location in Basics tab.](#)

Windows Plan (Central US) * ⓘ [Create new](#)

Sku and size * **Elastic Premium EP1**
210 total ACU, 3.5 GB memory
[Change size](#)

[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, you can change the New resource name or choose a different Location in an Azure geography to store your data. |

Function App

Basics Hosting **Monitoring** Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

(New) myfunctionapp (Central US)

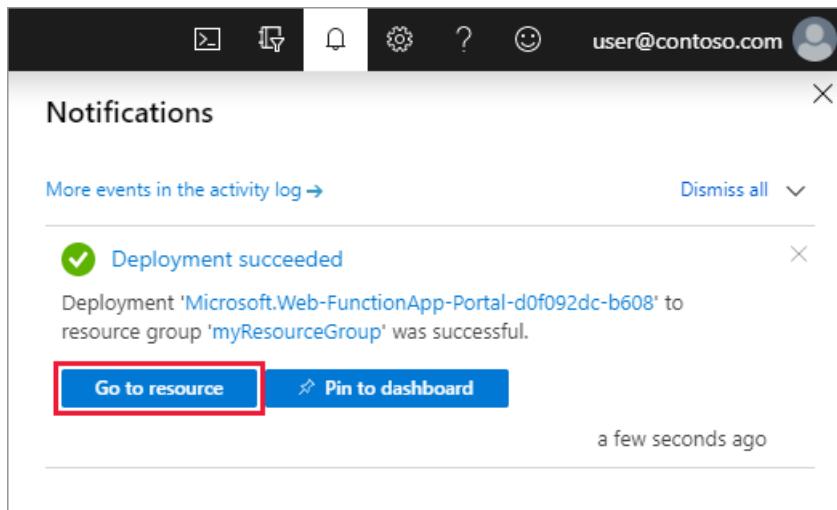
Create new

Region

Central US

Review + create < Previous Next : Tags >

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.



You can pin the function app to the dashboard by selecting the pin icon in the upper right-hand corner. Pinning makes it easier to return to this function app after you create your VM.

Create a VM inside a virtual network

Next, create a preconfigured VM that runs WordPress inside a virtual network ([WordPress LEMP7 Max Performance](#) by Jetware). A WordPress VM is used because of its low cost and convenience. This same scenario works with any resource in a virtual network, such as REST APIs, App Service Environments, and other Azure services.

1. In the portal, choose **+ Create a resource** on the left navigation pane, in the search field type `WordPress LEMP7 Max Performance`, and press Enter.
2. Choose **Wordpress LEMP Max Performance** in the search results. Select a software plan of **Wordpress LEMP Max Performance for CentOS** as the **Software Plan** and select **Create**.
3. 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 Enterprise [▼](#)

 * Resource group [?](#)

myResourceGroup [▼](#)

[Create new](#)

INSTANCE DETAILS

* Virtual machine name [?](#)

VNET-WordPress [▼](#) [✓](#)

* Region [?](#)

(Europe) West Europe [▼](#)

Availability options [?](#)

No infrastructure redundancy required [▼](#)

* Image [?](#)

Wordpress LEMP7 Max Performance on CentOS [▼](#)

[Browse all images](#)

* Size [?](#)

Standard B1s

1 vcpu, 1 GB memory

[Change size](#)

ADMINISTRATOR ACCOUNT

Authentication type [?](#)

Password SSH public key

* Username [?](#)

myusername [▼](#) [✓](#)

* Password [?](#)

..... [▼](#) [✓](#)

* Confirm password [?](#)

..... [▼](#) [✓](#)

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------------|----------------------|---|
| Subscription | Your subscription | The subscription under which your resources are created. |
| Resource group | myResourceGroup | Choose myResourceGroup, or the resource group you created with your function app. Using the same resource group for the function app, WordPress VM, and hosting plan makes it easier to clean up resources when you're done with this tutorial. |
| Virtual machine name | VNET-Wordpress | The VM name needs to be unique in the resource group |
| Region | (Europe) West Europe | Choose a region near you or near the functions that access the VM. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------------|-----------------|---|
| Size | B1s | Choose Change size and then select the B1s standard image, which has 1 vCPU and 1 GB of memory. |
| Authentication type | Password | To use password authentication, you must also specify a Username , a secure Password , and then Confirm password . For this tutorial, you won't need to sign in to the VM unless you need to troubleshoot. |

4. Choose the **Networking** tab and under Configure virtual networks select **Create new**.

5. 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

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).

| ADDRESS RANGE | ADDRESSES | OVERLAP |
|---------------|---|---------|
| 10.10.0.0/16 | 10.10.0.0 - 10.10.255.255 (65536 addresses) | None |
| | (0 Addresses) | None |

SUBNETS

The subnet's address range in CIDR notation. It must be contained by the address space of the virtual network.

| SUBNET NAME | ADDRESS RANGE | ADDRESSES |
|--------------|---------------|---|
| Tutorial-Net | 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-Net | Name of the subnet. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|------------------------|-----------------|--|
| 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 WordPress site. A /24 subnet provides 254 host addresses. |

6. Select **OK** to create the virtual network.
7. Back in the **Networking** tab, choose **None** for **Public IP**.
8. Choose the **Management** tab, then in **Diagnostics storage account**, choose the Storage account you created with your function app.
9. Select **Review + create**. After validation completes, select **Create**. The VM create process takes a few minutes. The created VM can only access the virtual network.
10. After the VM is created, choose **Go to resource** to view the page for your new VM, then choose **Networking** under **Settings**.
11. Verify that there's no **Public IP**. Make a note the **Private IP**, which you use to connect to the VM from your function app.

| PRIORITY | NAME | PORT | PROTOCOL | SOURCE |
|----------|------------------|------|----------|--------|
| 1010 | HTTP | 80 | TCP | Any |
| 1020 | HTTPS | 443 | TCP | Any |
| 1030 | HTTP_web_control | 1999 | TCP | Any |

You now have a WordPress site deployed entirely within your virtual network. This site isn't accessible from the public internet.

Connect your function app to the virtual network

With a WordPress site running in a VM in a virtual network, you can now connect your function app to that virtual network.

1. In your new function app, select **Networking** in the left menu.
2. Under **VNet Integration**, select **Click here to configure**.

myfunctionapp | Networking

App Service

Search (Ctrl+ /)

Settings

- Configuration
- Authentication / Authorizati...
- Application Insights
- Identity
- Backups
- Custom domains
- TLS/SSL settings
- Networking**

VNet Integration

Securely access resources available in or through your Azure VNet.
[Learn More](#)

Click here to configure

Hybrid connections

Securely access applications in private networks
[Learn More](#)

Configure your hybrid connection endpoints

Azure Front Door with Web Application Firewall

Scalable and secure entry point for accelerated delivery of your web applications
[Learn More](#)

Configure Azure Front Door with WAF for your app

Azure CDN

Secure, reliable content delivery with broad global reach and rich feature set
[Learn More](#)

3. On the VNET Integration page, select Add VNet.

VNet Integration

myfunctionapp

Disconnect Refresh

VNet Configuration

Securely access resources available in or through your Azure VNet. [Learn more](#)

Add VNet

VNet Details

| | |
|-----------|----------------|
| VNet NAME | Not Configured |
| LOCATION | Not Configured |

VNet Address Space

| | |
|----------------|-------------|
| Start Address | End Address |
| Not Configured | |

4. In Network Feature Status, use the settings in the table below the image:

Network Feature Status

X

vnet-function1

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. It doesn't matter that your functions use a different subnet than your VM. The virtual network automatically routes traffic between the two subnets. |
| Subnet name | Function-Net | Name of the new subnet. |
| Virtual network address block | 10.10.0.0/16 | Choose the same address block used by the WordPress site. You should only have one address block defined. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------|-----------------|---|
| Address range | 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. |

- 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 where the WordPress site is running. Next, you use [Azure Functions Proxies](#) to return a file from the WordPress site.

Create a proxy to access VM resources

With VNet Integration enabled, you can create a proxy in your function app to forward requests to the VM running in the virtual network.

- In your function app, select **Proxies** from the left menu, and then select **Add**. Use the proxy settings in the table below the image:

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|----------------|--|---|
| Name | Plant | The name can be any value. It's used to identify the proxy. |
| Route Template | /plant | Route that maps to a VM resource. |
| Backend URL | http://<YOUR_VM_IP>/wp-content/themes/twentyseventeen/assets/images/header.jpg | Replace <YOUR_VM_IP> with the IP address of your WordPress VM that you created earlier. This mapping returns a single file from the site. |

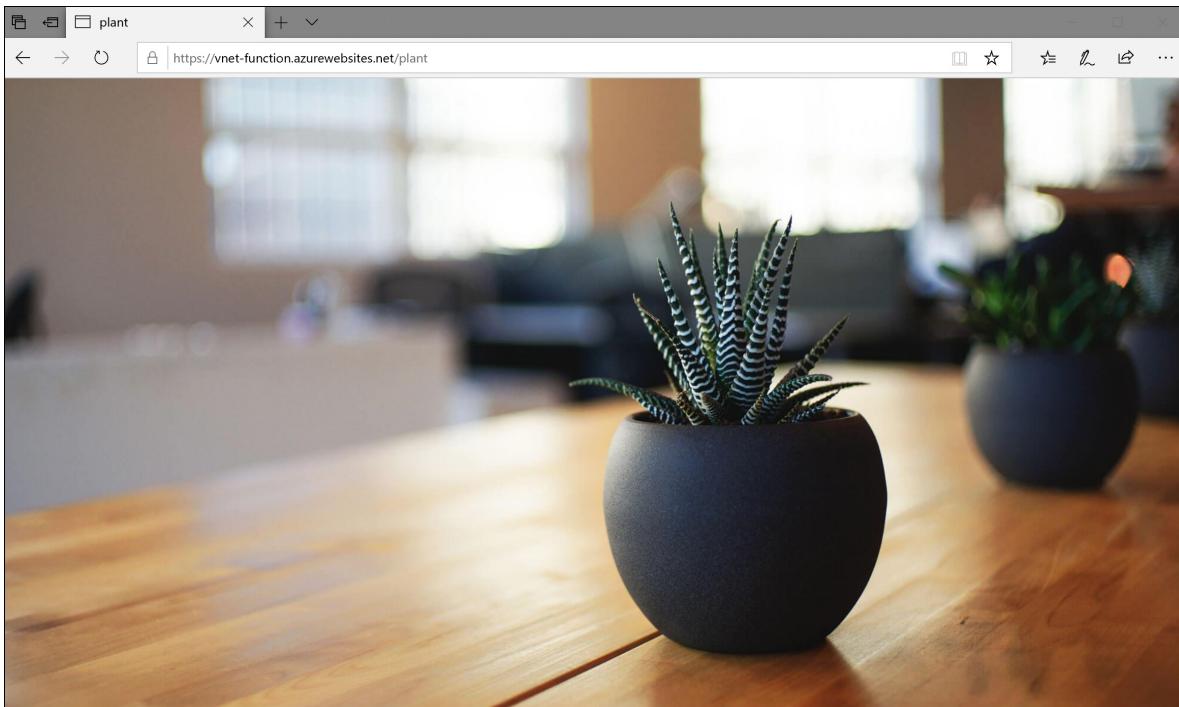
- Select **Create** to add the proxy to your function app.

Try it out

- In your browser, try to access the URL you used as the **Backend URL**. As expected, the request times out. A timeout occurs because your WordPress site is connected only to your virtual network and not the

internet.

2. Copy the **Proxy URL** value from your new proxy and paste it into the address bar of your browser. The returned image is from the WordPress site running inside your virtual network.



Your function app is connected to both the internet and your virtual network. The proxy is receiving a request over the public internet, and then acting as a simple HTTP proxy to forward that request to the connected virtual network. The proxy then relays the response back to you publicly over the internet.

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, the WordPress site serves as an API that is called by using a proxy in the function app. This scenario makes a good tutorial because it's easy to set up and visualize. You could use any other API deployed within a virtual network. You could also have created a function with code that calls APIs deployed within the virtual network. A more realistic scenario is a function that uses data client APIs to call a SQL Server instance deployed in the virtual network.

Functions running in a Premium plan share the same underlying App Service infrastructure as web apps on PremiumV2 plans. All the documentation for [web apps in Azure App Service](#) applies to your Premium plan functions.

[Learn more about the networking options in Functions](#)

Create an OpenAPI definition for a serverless API using Azure API Management

11/2/2020 • 7 minutes to read • [Edit Online](#)

REST APIs are often described using an OpenAPI definition. This definition contains information about what operations are available in an API and how the request and response data for the API should be structured.

In this tutorial, you create a function that determines whether an emergency repair on a wind turbine is cost-effective. You then create an OpenAPI definition for the function app using [Azure API Management](#) so that the function can be called from other apps and services.

In this tutorial, you learn how to:

- Create a function in Azure
- Generate an OpenAPI definition using Azure API Management
- Test the definition by calling the function
- Download the OpenAPI definition

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.

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. Choose .NET Core for C# and F# functions. |
| Version | Version number | Choose the version of your installed runtime. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|------------------|--|
| Region | Preferred region | Choose a region near you or near other services your functions access. |

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 * | <input type="text" value="Visual Studio Enterprise"/> |
| Resource Group * | <input type="text" value="(New) myResourceGroup"/> Create new |

Instance Details

| | |
|---------------------|---|
| Function App name * | <input type="text" value="myfunctionapp"/> .azurewebsites.net |
| Publish * | Code Docker Container |
| Runtime stack * | <input type="text" value=".NET Core"/> |
| Version * | <input type="text" value="3.1"/> |
| Region * | <input type="text" value="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. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------|--------------------------|---|
| Plan | 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. When you run in an App Service plan, you must manage the scaling of your function app. |

Function App X

[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 * ▼
[Create new](#)

Operating system
The Operating System has been recommended for you based on 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 * Consumption (Serverless) ▼

[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. |

Function App

Basics Hosting Monitoring Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights *

No Yes

Application Insights *

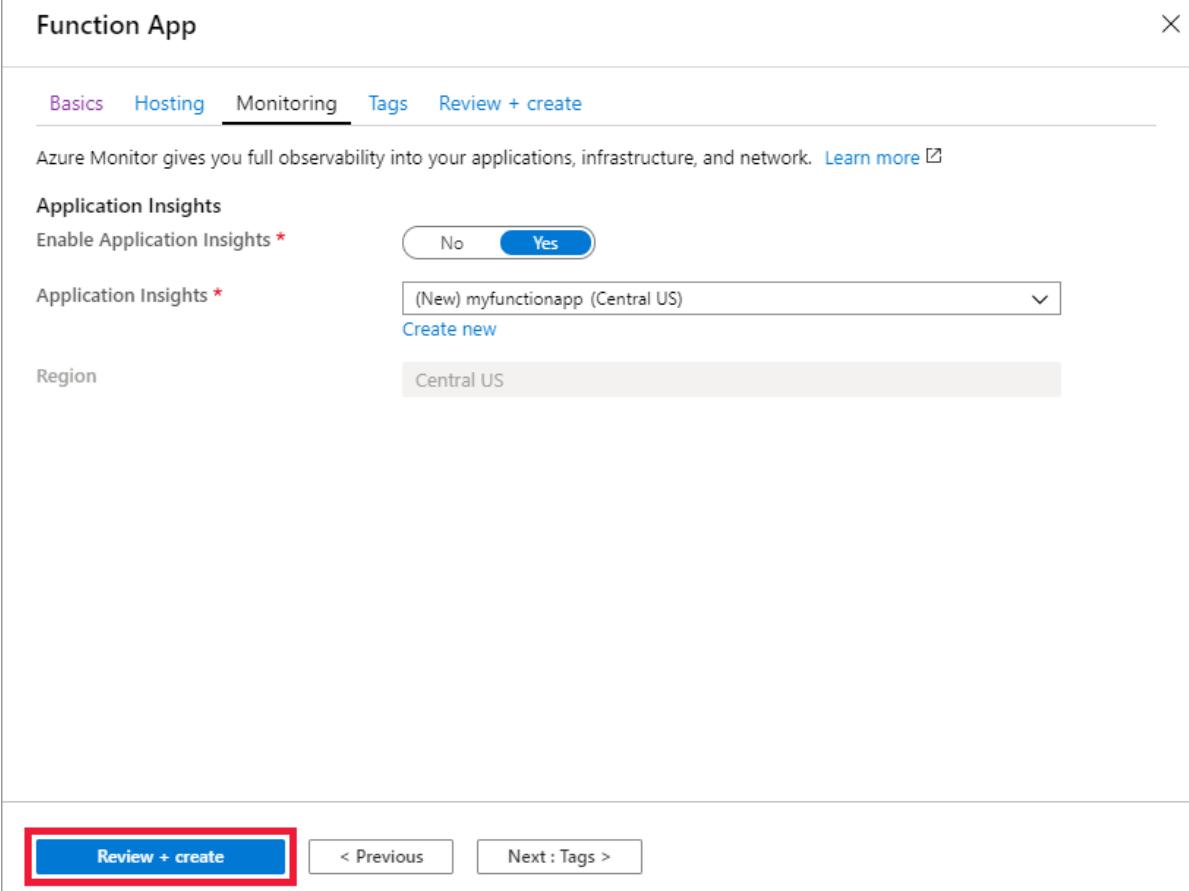
(New) myfunctionapp (Central US)

Create new

Region

Central US

Review + create < Previous Next : Tags >



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.

Notifications

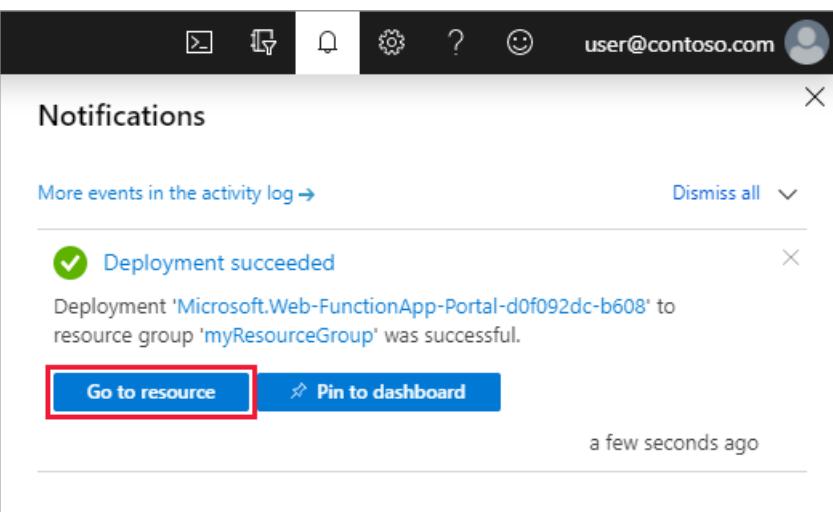
More events in the activity log → Dismiss all

Deployment succeeded

Deployment 'Microsoft.Web-FunctionApp-Portal-d0f092dc-b608' to resource group 'myResourceGroup' was successful.

Go to resource Pin to dashboard

a few seconds ago



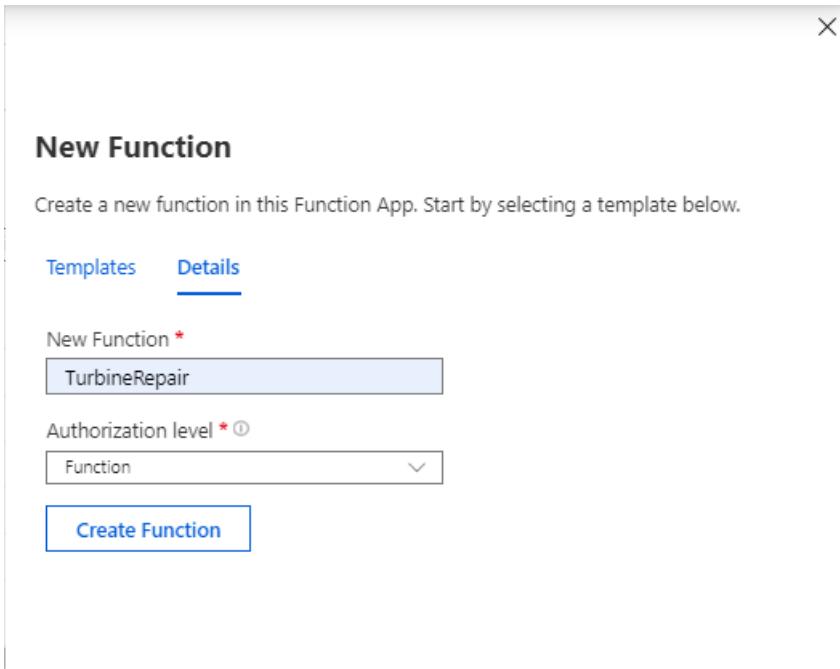
Create the function

This tutorial uses an HTTP triggered function that takes two parameters:

- The estimated time to make a turbine repair, in hours.
- The capacity of the turbine, in kilowatts.

The function then calculates how much a repair will cost, and how much revenue the turbine could make in a 24-hour period. To create the HTTP triggered function in the [Azure portal](#):

1. From the left menu of your functions app, select **Functions**, and then select **Add** from the top menu.
2. In the **New Function** window, select **Http trigger**.
3. For **New Function**, enter `TurbineRepair`.
4. Choose **Function** from the **Authorization level** drop-down list, and then select **Create Function**.



5. Select **Code + Test**, and then select `run.csx` from the drop-down list. Replace the contents of the `run.csx` C# script file with the following code, then choose **Save**:

```

#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

const double revenuePerkW = 0.12;
const double technicianCost = 250;
const double turbineCost = 100;

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    // Get query strings if they exist
    int tempVal;
    int? hours = Int32.TryParse(req.Query["hours"], out tempVal) ? tempVal : (int?)null;
    int? capacity = Int32.TryParse(req.Query["capacity"], out tempVal) ? tempVal : (int?)null;

    // Get request body
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);

    // Use request body if a query was not sent
    capacity = capacity ?? data?.capacity;
    hours = 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 on the query string or 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 (ActionResult)new OkObjectResult(new{
        message = repairTurbine,
        revenueOpportunity = "$"+ revenueOpportunity,
        costToFix = "$"+ costToFix
    });
}

```

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.

- To test the function, select **Test**, select the **Input** tab, enter the following input for the **Body**, and then select **Run**:

```
{
"hours": "6",
"capacity": "2500"
}
```

The screenshot shows the Azure Functions Test interface. On the left, the 'Code + Test' tab is selected. The code editor contains C# code for a function named 'TurbineRepair'. In the 'Input' tab, there is a JSON object with 'hours' and 'capacity' fields. The 'Run' button at the bottom is highlighted with a red box.

The following output is returned in the **Output** tab:

```
{"message": "Yes", "revenueOpportunity": "$7200", "costToFix": "$1600"}
```

Now you have a function that determines the cost-effectiveness of emergency repairs. Next, you generate an OpenAPI definition for the function app.

Generate the OpenAPI definition

To generate the OpenAPI definition:

1. Select the function app, choose **API Management** from the left menu, and then select **Create new** under **API Management**.

The screenshot shows the 'myfunctionapp | API Management' page. The left sidebar has 'API Management' selected. The main area shows the 'Create new' configuration page for API Management. A red box highlights the 'Create new' button in the 'API Management' dropdown.

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. |
| 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. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|---------------------|-----------------|--|
| Location | West US | Choose the West US location. |
| 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 □ X

* Name
 ✓
.azure-api.net

* Subscription
 ▼

* Resource group
 ▼
[Create new](#)

* Location
 ▼

* Organization name i
 ✓

* Administrator email i
 ✓

Pricing tier ([View full pricing details](#))
 ▼

Create
[Automation options](#)

3. Choose **Create** 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, and then select **Link API**.
5. The **Import Azure Functions** opens with the **TurbineRepair** function highlighted. Choose **Select** to continue.

Import Azure Functions

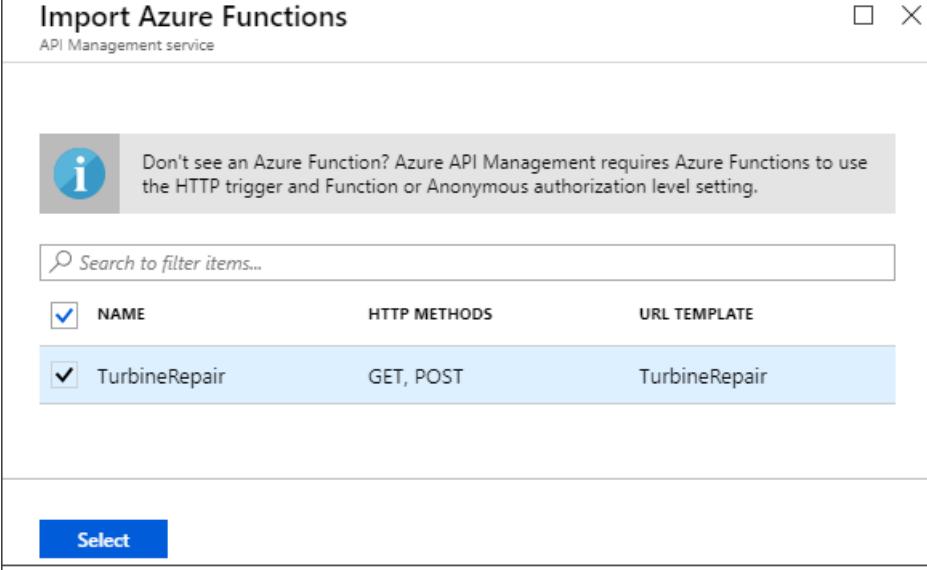
API Management service

Don't see an Azure Function? Azure API Management requires Azure Functions to use the HTTP trigger and Function or Anonymous authorization level setting.

Search to filter items...

| <input checked="" type="checkbox"/> NAME | HTTP METHODS | URL TEMPLATE |
|---|--------------|---------------|
| <input checked="" type="checkbox"/> TURBINEREPAIR | GET, POST | TURBINEREPAIR |

Select



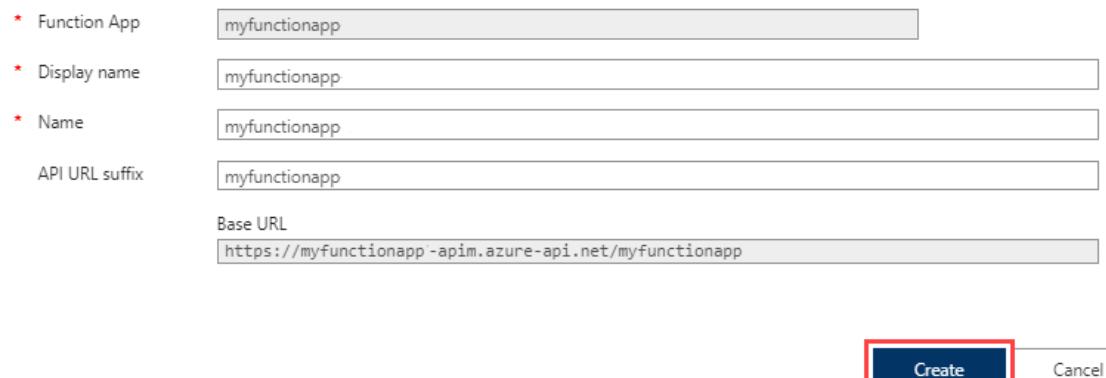
6. In the **Create from Function App** page, accept the defaults, and then select **Create**.

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.

Test the API

Before you use the OpenAPI definition, you should verify that the API works.

1. On your function app page, select **API Management**, select the **Test** tab, and then select **POST TURBINEREPAIR**.
2. Enter the following code in the **Request body**:

```
{  
  "hours": "6",  
  "capacity": "2500"  
}
```

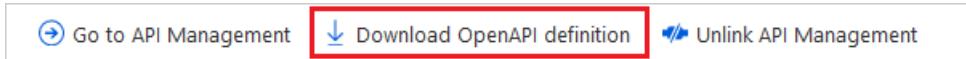
3. Select **Send**, and then view the **HTTP response**.

The screenshot shows the Azure portal interface for API Management. The left sidebar has a tree view with 'App Service plan', 'Development Tools', 'API' (which is expanded to show 'API Management', 'API definition', 'CORS', 'Monitoring', 'Metrics', 'Diagnostics logs', 'Log stream', and 'Process explorer'), and 'Support + troubleshooting'. The main area shows a 'REVISION 1' for 'TurbineRepair' with a 'Test' tab selected. It displays a 'POST' request to 'TurbineRepair' with a JSON body: { "hours": "6", "capacity": "2500"}. There are sections for 'Query parameters', 'Headers', and 'Request body'. At the bottom is a 'Send' button.

Download the OpenAPI definition

If your API works as expected, you can download the OpenAPI definition.

1. Select **Download OpenAPI definition** at the top of the page.



2. Save the downloaded JSON file, and then open it. Review the definition.

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

You have used API Management integration to generate an OpenAPI definition of your functions. You can now edit the definition in API Management in the portal. You can also [learn more about API Management](#).

[Edit the OpenAPI definition in API Management](#)

Exporting an Azure-hosted API to PowerApps and Microsoft Flow

11/2/2020 • 7 minutes to read • [Edit Online](#)

PowerApps is a service for building and using custom business apps that connect to your data and work across platforms. Power Automate makes it easy to automate workflows and business processes between your favorite apps and services. Both PowerApps and Microsoft Flow come with a variety of built-in connectors to data sources such as Office 365, Dynamics 365, Salesforce, and more. In some cases, app and flow builders also want to connect to data sources and APIs built by their organization.

Similarly, developers that want to expose their APIs more broadly within an organization can make their APIs available to app and flow builders. This article shows you how to export an API built with [Azure Functions](#) or [Azure App Service](#). The exported API becomes a *custom connector*, which is used in PowerApps and Microsoft Flow just like a built-in connector.

IMPORTANT

The API definition functionality shown in this article is only supported for [version 1.x of the Azure Functions runtime](#) and App Services apps. Version 2.x of Functions integrates with API Management to create and maintain OpenAPI definitions. To learn more, see [Create an OpenAPI definition for a function with Azure API Management](#).

Create and export an API definition

Before exporting an API, you must describe the API using an OpenAPI definition (formerly known as a [Swagger](#) file). This definition contains information about what operations are available in an API and how the request and response data for the API should be structured. PowerApps and Microsoft Flow can create custom connectors for any OpenAPI 2.0 definition. Azure Functions and Azure App Service have built-in support for creating, hosting, and managing OpenAPI definitions. For more information, see [Host a RESTful API with CORS in Azure App Service](#).

NOTE

You can also build custom connectors in the PowerApps and Microsoft Flow UI, without using an OpenAPI definition. For more information, see [Register and use a custom connector \(PowerApps\)](#) and [Register and use a custom connector \(Microsoft Flow\)](#).

To export the API definition, follow these steps:

1. In the [Azure portal](#), navigate to your function app or an App Service application.

From the left menu, under **API**, select **API definition**.

The screenshot shows the Azure portal interface for an 'App Service' named 'myfunctionapp'. On the left, there's a navigation sidebar with sections like 'Development Tools' (Console, Advanced Tools, App Service Editor, Resource explorer, Extensions), 'API' (API Management, API definition, CORS), and 'CORS'. The 'API definition' item is highlighted with a red box. The main content area is titled 'API definition' and contains a brief description of what it does. It has a 'URL of API definition' input field, an 'Export to PowerApps + Microsoft Flow' button, and 'Save' and 'Discard' buttons.

2. The **Export to PowerApps + Microsoft Flow** button should be available (if not, you must first create an OpenAPI definition). Select this button to begin the export process.

Export to PowerApps + Microsoft Flow

3. Select the **Export Mode**:

Express lets you create the custom connector from within the Azure portal. It requires that you are signed into PowerApps or Microsoft Flow and have permission to create connectors in the target environment. This approach is recommended if these two requirements can be met. If using this mode, follow the [Use express export](#) instructions below.

Manual lets you export the API definition, which you then import using the PowerApps or Microsoft Flow portals. This approach is recommended if the Azure user and the user with permission to create connectors are different people, or if the connector needs to be created in another Azure tenant. If using this mode, follow the [Use manual export](#) instructions below.

This screenshot shows the 'PowerApps + Microsoft Flow' app settings page. It features icons for PowerApps and Microsoft Flow, a description of the app as a custom API, and a note about using it for PowerApps and Microsoft Flow. At the bottom, there's a 'Export Mode' section with two buttons: 'Express' (which is selected and highlighted with a red box) and 'Manual'.

NOTE

The custom connector uses a *copy* of the API definition, so PowerApps and Microsoft Flow will not immediately know if you make changes to the application and its API definition. If you do make changes, repeat the export steps for the new version.

Use express export

To complete the export in **Express** mode, follow these steps:

1. Make sure you're signed in to the PowerApps or Microsoft Flow tenant to which you want to export.
2. Use the settings as specified in the table.

| SETTING | DESCRIPTION |
|---------|-------------|
|---------|-------------|

| SETTING | DESCRIPTION |
|---------------------------------------|--|
| Environment | Select the environment that the custom connector should be saved to. For more information, see Environments overview . |
| Custom API Name | Enter a name, which PowerApps and Microsoft Flow builders will see in their connector list. |
| Prepare security configuration | If necessary, provide the security configuration details needed to grant users access to your API. This example shows an API key. For more information, see Specify authentication type below. |

The screenshot shows the 'PowerApps + Microsoft Flow' app interface with three main steps:

- 1 Configure Custom API**: You have permission to create custom APIs in some environments and can create a custom API immediately. Environment: Microsoft (new default). * Custom API Name: Turbine Repair.
- 2 Prepare security configuration**: Your metadata has security definitions. PowerApps and Microsoft Flow will ask for related configuration settings during import. Provide the security configuration details for the schema below.

| DEFINITION NAME | TYPE |
|-----------------|--------|
| apikeyQuery | apiKey |

 Select security scheme: API Key. * API Key Name: API Key (contact meganb@contoso.com).
- 3 Export to PowerApps + Microsoft Flow**: Click OK to create a new custom API using the above parameters. By default, the custom API will be shared only with you. Visit the PowerApps or Microsoft Flow portal to share with the other members of the organization.

OK

3. Click **OK**. The custom connector is now built and added to the environment you specified.

Use manual export

To complete the export in **Manual** mode, follow these steps:

1. Click **Download** and save the file, or click the copy button and save the URL. You will use the download file or the URL during import.

The screenshot shows the 'PowerApps + Microsoft Flow' app interface. At the top, there are icons for PowerApps and Microsoft Flow. Below them, the title 'PowerApps + Microsoft Flow' is displayed. A sub-header says 'Use this app as a custom API for PowerApps and Microsoft Flow. [Learn more](#)'. There are three buttons for 'Export Mode': 'Express' (selected), 'Manual', and 'Advanced'. Step 1, 'Prepare your metadata', is highlighted with a large number '1'. A note below it says: 'Your API definition is available as an OpenAPI (Swagger) document. Download a copy or make note of the link below. You will use this in step 3.' An 'API definition location' field contains a URL: 'https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=' followed by a redacted code segment. A blue 'Download' button is at the bottom of the section.

2. If your API definition includes any security definitions, these definitions are called out in step #2. During import, PowerApps and Microsoft Flow detects these definitions and prompts for security information. Gather the credentials related to each definition for use in the next section. For more information, see [Specify authentication type](#) below.

The screenshot shows the 'PowerApps + Microsoft Flow' app interface. Step 2, 'Prepare security configuration', is highlighted with a large number '2'. A note says: 'Your metadata has security definitions. PowerApps and Microsoft Flow will ask for related configuration settings during import. [Learn more](#)'. Below is a table with two rows:

| DEFINITION NAME | TYPE |
|-----------------|--------|
| apikeyQuery | apiKey |

This example shows the API key security definition that was included in the OpenAPI definition.

Now that you've exported the API definition, you import it to create a custom connector in PowerApps and Microsoft Flow. Custom connectors are shared between the two services, so you only need to import the definition once.

To import the API definition into PowerApps and Microsoft Flow, follow these steps:

1. Go to [powerapps.com](#) or [flow.microsoft.com](#).
2. In the upper right corner, click the gear icon, then click **Custom connectors**.

The screenshot shows a menu for creating a custom connector. At the top is a red box around the '+ Create custom connector' button. Below it are four options: 'Create from blank', 'Import an OpenAPI file' (which is also highlighted with a red box), 'Import an OpenAPI from URL', and 'Import a Postman collection'.

3. Click **Create custom connector**, then click **Import an OpenAPI definition**.
4. Enter a name for the custom connector, then navigate to the OpenAPI definition that you exported, and click

Continue.

Create custom connector

Custom connector title

Turbine Repair

Upload an OpenAPI file

ApiDef.json

Continue Cancel

5. On the **General** tab, review the information that comes from the OpenAPI definition.
6. On the **Security** tab, if you are prompted to provide authentication details, enter the values appropriate for the authentication type. Click **Continue**.

Authentication type

Choose what authentication is implemented by your API *

API Key

API Key

Users will be required to provide the API Key when creating a connection

| Parameter label | Parameter name | Parameter location |
|---------------------|----------------|--------------------|
| API Key (contact n) | code | Query |

This example shows the required fields for API key authentication. The fields differ depending on the authentication type.

7. On the **Definitions** tab, all the operations defined in your OpenAPI file are auto-populated. If all your required operations are defined, you can go to the next step. If not, you can add and modify operations here.

Actions (1)

Actions determine the operations that users can perform. Actions can be used to read, create, update or delete resources in the underlying connector.

1 CalculateCosts ...

+ New action

References (0)

References are reusable parameters used by both actions and triggers.

General

* Summary Learn more

Calculates costs

* Description Learn more

Determines if a technician should be sent for repair

* Operation ID

This is the unique string used to identify the operation.

CalculateCosts

Visibility Learn more

none advanced internal important

This example has one operation, named `CalculateCosts`. The metadata, like **Description**, all comes from the OpenAPI file.

8. Click **Create connector** at the top of the page.

You can now connect to the custom connector in PowerApps and Microsoft Flow. For more information on creating

connectors in the PowerApps and Microsoft Flow portals, see [Register your custom connector \(PowerApps\)](#) and [Register your custom connector \(Microsoft Flow\)](#).

Specify authentication type

PowerApps and Microsoft Flow support a collection of identity providers that provide authentication for custom connectors. If your API requires authentication, ensure that it is captured as a *security definition* in your OpenAPI document, like the following example:

```
"securityDefinitions": {  
    "AAD": {  
        "type": "oauth2",  
        "flow": "accessCode",  
        "authorizationUrl": "https://login.windows.net/common/oauth2/authorize",  
        "scopes": {}  
    }  
}
```

During export, you provide configuration values that allow PowerApps and Microsoft Flow to authenticate users.

This section covers the authentication types that are supported in Express mode: API key, Azure Active Directory, and Generic OAuth 2.0. PowerApps and Microsoft Flow also support Basic Authentication, and OAuth 2.0 for specific services like Dropbox, Facebook, and SalesForce.

API key

When using an API key, the users of your connector are prompted to provide the key when they create a connection. You specify an API key name to help them understand which key is needed. In the earlier example, we use the name `API Key (contact meganb@contoso.com)` so people know where to get information about the API key. For Azure Functions, the key is typically one of the host keys, covering several functions within the function app.

Azure Active Directory (Azure AD)

When using Azure AD, you need two Azure AD application registrations: one for the API itself, and one for the custom connector:

- To configure registration for the API, use the [App Service Authentication/Authorization](#) feature.
- To configure registration for the connector, follow the steps in [Adding an Azure AD application](#). The registration must have delegated access to your API and a reply URL of
`https://msmanaged-na.consent.azure-apim.net/redirect`.

For more information, see the Azure AD registration examples for [PowerApps](#) and [Microsoft Flow](#). These examples use Azure Resource Manager as the API; substitute your API if you follow the steps.

The following configuration values are required:

- **Client ID** - the client ID of your connector Azure AD registration
- **Client secret** - the client secret of your connector Azure AD registration
- **Login URL** - the base URL for Azure AD. In Azure, typically `https://login.windows.net`.
- **Tenant ID** - the ID of the tenant to be used for the login. This ID should be "common" or the ID of the tenant in which the connector is created.
- **Resource URL** - the resource URL of the Azure AD registration for your API

IMPORTANT

If someone else will import the API definition into PowerApps and Microsoft Flow as part of the manual flow, you must provide them with the client ID and client secret of the *connector registration*, as well as the resource URL of your API. Make sure that these secrets are managed securely. **Do not share the security credentials of the API itself.**

Generic OAuth 2.0

When using generic OAuth 2.0, you can integrate with any OAuth 2.0 provider. Doing so allows you to work with custom providers that are not natively supported.

The following configuration values are required:

- **Client ID** - the OAuth 2.0 client ID
- **Client secret** - the OAuth 2.0 client secret
- **Authorization URL** - the OAuth 2.0 authorization URL
- **Token URL** - the OAuth 2.0 token URL
- **Refresh URL** - the OAuth 2.0 refresh URL

How to use managed identities for App Service and Azure Functions

12/4/2020 • 16 minutes to read • [Edit Online](#)

This topic 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

Managed identities for App Service and Azure Functions won't behave as expected if your app is migrated across subscriptions/tenants. The app needs to obtain a new identity, which is done by disabling and re-enabling the feature. See [Removing an identity](#) below. Downstream resources also need to have access policies updated to use the new identity.

A managed identity from Azure Active Directory (Azure AD) allows your app to easily access other Azure AD-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 Azure AD, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to your application and is deleted if your 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.

Add a system-assigned identity

Creating an app with a system-assigned identity requires an additional property to be set on the application.

Using the Azure portal

To set up a managed identity in the portal, you will first create an application as normal and then enable the feature.

1. Create an app in the portal as you normally would. Navigate to it in the portal.
2. If using a function app, navigate to **Platform features**. For other app types, 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 portal interface for managing an App Service. The left sidebar lists various settings like Authentication / Authorization, Application Insights, and Identity (which is highlighted with a red box). The main content area shows the 'System assigned' tab selected for managed identities. It includes a description of what a system-assigned identity is, a 'Save' button, and a 'Status' section with an 'Off' button that has been changed to 'On' (also highlighted with a red box).

NOTE

To find the managed identity for your web app or slot app in the Azure portal, under **Enterprise applications**, look in the **User settings** section. Usually, the slot name is similar to <app name>/slots/<slot name>.

Using the Azure CLI

To set up a managed identity using the Azure CLI, you will need to use the `az webapp identity assign` command against an existing application. You have three options for running the examples in this section:

- Use [Azure Cloud Shell](#) from the Azure portal.
- Use the embedded Azure Cloud Shell via the "Try It" button, located in the top-right corner of each code block below.
- [Install the latest version of Azure CLI](#) (2.0.31 or later) if you prefer to use a local CLI console.

The following steps will walk you through creating a web app and assigning it an identity using the CLI:

1. If you're using the Azure CLI in a local console, first sign in to Azure using [az login](#). Use an account that's associated with the Azure subscription under which you would like to deploy the application:

```
az login
```

2. Create a web application using the CLI. For more examples of how to use the CLI with App Service, see [App Service CLI samples](#):

```
az group create --name myResourceGroup --location westus
az appservice plan create --name myPlan --resource-group myResourceGroup --sku S1
az webapp create --name myApp --resource-group myResourceGroup --plan myPlan
```

3. Run the `identity assign` command to create the identity for this application:

```
az webapp identity assign --name myApp --resource-group myResourceGroup
```

Using Azure PowerShell

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

The following steps will walk you through creating an app and assigning it an identity using Azure PowerShell. The instructions for creating a web app and a function app are different.

Using Azure PowerShell for a web app

1. If needed, install the Azure PowerShell using the instructions found in the [Azure PowerShell guide](#), and then run `Login-AzAccount` to create a connection with Azure.
2. Create a web application using Azure PowerShell. For more examples of how to use Azure PowerShell with App Service, see [App Service PowerShell samples](#):

```
# Create a resource group.  
New-AzResourceGroup -Name $resourceGroupName -Location $location  
  
# Create an App Service plan in Free tier.  
New-AzAppServicePlan -Name $webappName -Location $location -ResourceGroupName $resourceGroupName -Tier Free  
  
# Create a web app.  
New-AzWebApp -Name $webappName -Location $location -AppServicePlan $webappName -ResourceGroupName $resourceGroupName
```

3. Run the `Set-AzWebApp -AssignIdentity` command to create the identity for this application:

```
Set-AzWebApp -AssignIdentity $true -Name $webappName -ResourceGroupName $resourceGroupName
```

Using Azure PowerShell for a function app

1. If needed, install the Azure PowerShell using the instructions found in the [Azure PowerShell guide](#), and then run `Login-AzAccount` to create a connection with Azure.
2. Create a function app using Azure PowerShell. For more examples of how to use Azure PowerShell with Azure Functions, see the [Az.Functions reference](#):

```
# Create a resource group.  
New-AzResourceGroup -Name $resourceGroupName -Location $location  
  
# Create a storage account.  
New-AzStorageAccount -Name $storageAccountName -ResourceGroupName $resourceGroupName -SkuName $sku  
  
# Create a function app with a system-assigned identity.  
New-AzFunctionApp -Name $functionAppName -ResourceGroupName $resourceGroupName -Location $location -  
StorageAccountName $storageAccountName -Runtime $runtime -IdentityType SystemAssigned
```

You can also update an existing function app using `Update-AzFunctionApp` instead.

Using an Azure Resource Manager template

An Azure Resource Manager template can be used to automate deployment of your Azure resources. To learn more about deploying to App Service and Functions, see [Automating resource deployment in App Service](#) and [Automating resource deployment in Azure Functions](#).

Any resource of type `Microsoft.Web/sites` can be created with an identity by including the following property in the resource definition:

```
"identity": {  
    "type": "SystemAssigned"  
}
```

NOTE

An application can have both system-assigned and user-assigned identities at the same time. In this case, the `type` property would be `SystemAssigned,UserAssigned`

Adding the system-assigned type tells Azure to create and manage the identity for your application.

For example, a web app might look like the following:

```
{  
    "apiVersion": "2016-08-01",  
    "type": "Microsoft.Web/sites",  
    "name": "[variables('appName')]",  
    "location": "[resourceGroup().location]",  
    "identity": {  
        "type": "SystemAssigned"  
    },  
    "properties": {  
        "name": "[variables('appName')]",  
        "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",  
        "hostingEnvironment": "",  
        "clientAffinityEnabled": false,  
        "alwaysOn": true  
    },  
    "dependsOn": [  
        "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]"  
    ]  
}
```

When the site is created, it has the following additional properties:

```
"identity": {  
    "type": "SystemAssigned",  
    "tenantId": "<TENANTID>",  
    "principalId": "<PRINCIPALID>"  
}
```

The `tenantId` property identifies what Azure AD tenant the identity belongs to. The `principalId` is a unique identifier for the application's new identity. Within Azure AD, the service principal has the same name that you gave to your App Service or Azure Functions instance.

If you need to reference these properties in a later stage in the template, you can do so via the `reference()` [template function](#) with the `'Full'` flag, as in this example:

```
{  
    "tenantId": "[reference(resourceId('Microsoft.Web/sites', variables('appName')), '2018-02-01',  
    'Full').identity.tenantId]",  
    "objectId": "[reference(resourceId('Microsoft.Web/sites', variables('appName')), '2018-02-01',  
    'Full').identity.principalId]",  
}
```

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.

Using the Azure portal

First, you'll need to create a user-assigned identity resource.

1. Create a user-assigned managed identity resource according to [these instructions](#).
2. Create an app in the portal as you normally would. Navigate to it in the portal.
3. If using a function app, navigate to **Platform features**. For other app types, scroll down to the **Settings** group in the left navigation.
4. Select **Identity**.
5. Within the **User assigned** tab, click **Add**.
6. Search for the identity you created earlier and select it. Click **Add**.

The screenshot shows the Azure portal interface for managing identities. On the left, there's a sidebar with various settings like Configuration, Authentication / Authorization, Application Insights, and Identity (which is highlighted with a red box). The main area shows a table for user-assigned identities, with a red box around the '+ Add' button. On the right, a modal window titled 'Add user assigned managed identity' is open. It has a subscription dropdown set to 'APEX C+L - Aquent Vendor Subscriptions'. A search bar contains 'userassig...'. Below it, a list shows a single result: 'userassignedmanagedidentity' under 'Resource Group: appRG'. At the bottom of the modal, there's another red box around the 'Add' button.

Using Azure PowerShell

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

The following steps will walk you through creating an app and assigning it an identity using Azure PowerShell.

NOTE

The current version of the Azure PowerShell commandlets for Azure App Service do not support user-assigned identities. The below instructions are for Azure Functions.

1. If needed, install the Azure PowerShell using the instructions found in the [Azure PowerShell guide](#), and then run `Login-AzAccount` to create a connection with Azure.
2. Create a function app using Azure PowerShell. For more examples of how to use Azure PowerShell with Azure Functions, see the [Az.Functions reference](#). The below script also makes use of `New-AzUserAssignedIdentity` which must be installed separately as per [Create, list or delete a user-assigned managed identity using Azure PowerShell](#).

```
# Create a resource group.  
New-AzResourceGroup -Name $resourceGroupName -Location $location  
  
# Create a storage account.  
New-AzStorageAccount -Name $storageAccountName -ResourceGroupName $resourceGroupName -SkuName $sku  
  
# Create a user-assigned identity. This requires installation of the "Az.ManagedServiceIdentity"  
# module.  
$userAssignedIdentity = New-AzUserAssignedIdentity -Name $userAssignedIdentityName -ResourceGroupName  
$resourceGroupName  
  
# Create a function app with a user-assigned identity.  
New-AzFunctionApp -Name $functionAppName -ResourceGroupName $resourceGroupName -Location $location -  
StorageAccountName $storageAccountName -Runtime $runtime -IdentityType UserAssigned -IdentityId  
$userAssignedIdentity.Id
```

You can also update an existing function app using `Update-AzFunctionApp` instead.

Using an Azure Resource Manager template

An Azure Resource Manager template can be used to automate deployment of your Azure resources. To learn more about deploying to App Service and Functions, see [Automating resource deployment in App Service](#) and [Automating resource deployment in Azure Functions](#).

Any resource of type `Microsoft.Web/sites` can be created with an identity by including the following block in the resource definition, replacing `<RESOURCEID>` with the resource ID of the desired identity:

```
"identity": {  
    "type": "UserAssigned",  
    "userAssignedIdentities": {  
        "<RESOURCEID>": {}  
    }  
}
```

NOTE

An application can have both system-assigned and user-assigned identities at the same time. In this case, the `type` property would be `SystemAssigned,UserAssigned`

Adding the user-assigned type tells Azure to use the user-assigned identity specified for your application.

For example, a web app might look like the following:

```
{
    "apiVersion": "2016-08-01",
    "type": "Microsoft.Web/sites",
    "name": "[variables('appName')]",
    "location": "[resourceGroup().location]",
    "identity": {
        "type": "UserAssigned",
        "userAssignedIdentities": {
            "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', variables('identityName'))]": {}
        }
    },
    "properties": {
        "name": "[variables('appName')]",
        "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
        "hostingEnvironment": "",
        "clientAffinityEnabled": false,
        "alwaysOn": true
    },
    "dependsOn": [
        "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
        "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', variables('identityName'))]"
    ]
}
```

When the site is created, it has the following additional properties:

```
"identity": {
    "type": "UserAssigned",
    "userAssignedIdentities": {
        "<RESOURCEID>": {
            "principalId": "<PRINCIPALID>",
            "clientId": "<CLIENTID>"
        }
    }
}
```

The principalId is a unique identifier for the identity that's used for Azure AD administration. The clientId is a unique identifier for the application's new identity that's used for specifying which identity to use during runtime calls.

Obtain tokens for Azure resources

An app can use its managed identity to get tokens to access other resources protected by Azure AD, such as Azure Key Vault. These tokens represent the application accessing the resource, and not any specific user of the application.

You may need to configure the target resource to allow access from your application. For example, if you request a token to access Key Vault, you need to make sure you have added an access policy that includes your application's identity. Otherwise, your calls to Key Vault will be rejected, even if they include the token. To learn more about which resources support Azure Active Directory tokens, see [Azure services that support Azure AD authentication](#).

IMPORTANT

The back-end services for managed identities maintain a cache per resource URI for around 8 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.

There is a simple REST protocol for obtaining a token in App Service and Azure Functions. This can be used for all applications and languages. For .NET and Java, the Azure SDK provides an abstraction over this protocol and

facilitates a local development experience.

Using the REST protocol

NOTE

An older version of this protocol, using the "2017-09-01" API version, used the `secret` header instead of `X-IDENTITY-HEADER` and only accepted the `clientid` property for user-assigned. It also returned the `expires_on` in a timestamp format. `MSI_ENDPOINT` can be used as an alias for `IDENTITY_ENDPOINT`, and `MSI_SECRET` can be used as an alias for `IDENTITY_HEADER`. This version of the protocol is currently required for Linux Consumption hosting plans.

An app with a managed identity has two environment variables defined:

- `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:

| PARAMETER NAME | IN | DESCRIPTION |
|-------------------|--------|--|
| resource | Query | The Azure AD resource URI of the resource for which a token should be obtained. This could be one of the Azure services that support Azure AD authentication or any other resource URI. |
| api-version | Query | The version of the token API to be used. Please use "2019-08-01" or later (unless using Linux Consumption, which currently only offers "2017-09-01" - see note above). |
| X-IDENTITY-HEADER | Header | The value of the <code>IDENTITY_HEADER</code> 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. |

| PARAMETER NAME | IN | DESCRIPTION |
|----------------|-------|---|
| 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 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.

A successful 200 OK response includes a JSON body with the following properties:

| PROPERTY NAME | DESCRIPTION |
|---------------|--|
| access_token | The requested access token. The calling web service can use this token to authenticate to the receiving web service. |
| client_id | The client ID of the identity that was used. |
| expires_on | The timespan when the access token expires. The date is represented as the number of seconds from "1970-01-01T0:0:0Z UTC" (corresponds to the token's <code>exp</code> claim). |
| not_before | The timespan when the access token takes effect, and can be accepted. The date is represented as the number of seconds from "1970-01-01T0:0:0Z UTC" (corresponds to the token's <code>nbf</code> claim). |
| resource | The resource the access token was requested for, which matches the <code>resource</code> query string parameter of the request. |
| token_type | Indicates the token type value. The only type that Azure AD supports is Bearer. For more information about bearer tokens, see The OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC 6750) . |

This response is the same as the [response for the Azure AD service-to-service access token request](#).

REST protocol examples

An example request might look like the following:

```
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/1.1 200 OK
Content-Type: application/json

{
    "access_token": "eyJ0eXAi...",
    "expires_on": "1586984735",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer",
    "client_id": "5E29463D-71DA-4FE0-8E69-999B57DB23B0"
}
```

Code examples

- [.NET](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

TIP

For .NET languages, you can also use [Microsoft.Azure.Services.AppAuthentication](#) instead of crafting this request yourself.

```
private readonly HttpClient _client;
// ...
public async Task<HttpResponseMessage> GetToken(string resource) {
    var request = new HttpRequestMessage(HttpMethod.Get,
        String.Format("{0}/?resource={1}&api-version=2019-08-01",
        Environment.GetEnvironmentVariable("IDENTITY_ENDPOINT"), resource));
    request.Headers.Add("X-IDENTITY-HEADER", Environment.GetEnvironmentVariable("IDENTITY_HEADER"));
    return await _client.SendAsync(request);
}
```

Using the Microsoft.Azure.Services.AppAuthentication library for .NET

For .NET applications and functions, the simplest way to work with a managed identity is through the [Microsoft.Azure.Services.AppAuthentication](#) package. This library will also allow you to test your code locally on your development machine, using your user account from Visual Studio, the [Azure CLI](#), or Active Directory Integrated Authentication. When hosted in the cloud, it will default to using a system-assigned identity, but you can customize this behavior using a connection string environment variable which references the client ID of a user-assigned identity. For more on development options with this library, see the [Microsoft.Azure.Services.AppAuthentication reference](#). This section shows you how to get started with the library in your code.

1. Add references to the [Microsoft.Azure.Services.AppAuthentication](#) and any other necessary NuGet packages to your application. The below example also uses [Microsoft.Azure.KeyVault](#).

2. Add the following code to your application, modifying to target the correct resource. This example shows two ways to work with Azure Key Vault:

```
using Microsoft.Azure.Services.AppAuthentication;
using Microsoft.Azure.KeyVault;
// ...
var azureServiceTokenProvider = new AzureServiceTokenProvider();
string accessToken = await azureServiceTokenProvider.GetAccessTokenAsync("https://vault.azure.net");
// OR
var kv = new KeyVaultClient(new
KeyVaultClient.AuthenticationCallback(azureServiceTokenProvider.KeyVaultTokenCallback));
```

If you want to use a user-assigned managed identity, you can set the `AzureServicesAuthConnectionString` application setting to `RunAs=App;AppId=<clientId-guid>`. Replace `<clientId-guid>` with the client ID of the identity you want to use. You can define multiple such connection strings by using custom application settings and passing their values into the `AzureServiceTokenProvider` constructor.

```
var identityConnectionString1 = Environment.GetEnvironmentVariable("UA1_ConnectionString");
var azureServiceTokenProvider1 = new AzureServiceTokenProvider(identityConnectionString1);

var identityConnectionString2 = Environment.GetEnvironmentVariable("UA2_ConnectionString");
var azureServiceTokenProvider2 = new AzureServiceTokenProvider(identityConnectionString2);
```

To learn more about configuring `AzureServiceTokenProvider` and the operations it exposes, see the [Microsoft.Azure.Services.AppAuthentication reference](#) and the [App Service and KeyVault with MSI .NET sample](#).

Using the Azure SDK for Java

For Java applications and functions, the simplest way to work with a managed identity is through the [Azure SDK for Java](#). This section shows you how to get started with the library in your code.

1. Add a reference to the [Azure SDK library](#). For Maven projects, you might add this snippet to the `dependencies` section of the project's POM file:

```
<dependency>
<groupId>com.microsoft.azure</groupId>
<artifactId>azure</artifactId>
<version>1.23.0</version>
</dependency>
```

2. Use the `AppServiceMSICredentials` object for authentication. This example shows how this mechanism may be used for working with Azure Key Vault:

```
import com.microsoft.azure.AzureEnvironment;
import com.microsoft.azure.management.Azure;
import com.microsoft.azure.management.keyvault.Vault
//...
Azure azure = Azure.authenticate(new AppServiceMSICredentials(AzureEnvironment.AZURE))
    .withSubscription(subscriptionId);
Vault myKeyVault = azure.vaults().getByResourceGroup(resourceGroup, keyvaultName);
```

Remove an identity

A system-assigned identity can be removed by disabling the feature using the portal, PowerShell, or CLI in the same way that it was created. User-assigned identities can be removed individually. To remove all identities, set the identity type to "None".

Removing a system-assigned identity in this way will also delete it from Azure AD. System-assigned identities are also automatically removed from Azure AD when the app resource is deleted.

To remove all identities in an [ARM template](#):

```
"identity": {  
    "type": "None"  
}
```

To remove all identities in Azure PowerShell (Azure Functions only):

```
# Update an existing function app to have IdentityType "None".  
Update-AzFunctionApp -Name $functionAppName -ResourceGroupName $resourceGroupName -IdentityType None
```

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.

Next steps

[Access SQL Database securely using a managed identity](#)

Customize an HTTP endpoint in Azure Functions

11/2/2020 • 6 minutes to read • [Edit Online](#)

In this article, you learn how Azure Functions allows you to build highly scalable APIs. Azure Functions comes with a collection of built-in HTTP triggers and bindings, which make it easy to author an endpoint in a variety of languages, including Node.js, C#, and more. In this article, you'll customize an HTTP trigger to handle specific actions in your API design. You'll also prepare for growing your API by integrating it with Azure Functions Proxies and setting up mock APIs. These tasks are accomplished on top of the Functions serverless compute environment, so you don't have to worry about scaling resources - you can just focus on your API logic.

Prerequisites

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.

The resulting function will be used for the rest of this article.

Sign in to Azure

Sign in to the [Azure portal](#) with your Azure account.

Customize your HTTP function

By default, your HTTP trigger function is configured to accept any HTTP method. You can also use the default URL, `http://<yourapp>.azurewebsites.net/api/<funcname>?code=<functionkey>`. In this section, you modify the function to respond only to GET requests with `/api/hello`.

1. Navigate to your function in the Azure portal. Select **Integration** in the left menu, and then select **HTTP (req)** under **Trigger**.

The screenshot shows the Azure portal interface for editing an Azure Function named 'HttpTrigger1'. The left sidebar has 'Integration' selected under 'Developer' tools. The main area shows the 'Edit Trigger' configuration. The 'Trigger' section has 'HTTP (req)' selected. The 'Binding Type' is set to 'HTTP'. The 'Request parameter name' is 'req'. The 'Route template' is '/hello'. The 'Authorization level' is 'Anonymous'. The 'Selected HTTP methods' dropdown is set to 'GET'. The 'Save' button is highlighted with a red box.

2. Use the HTTP trigger settings as specified in the following 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 |

You didn't include the `/api` base path prefix in the route template, because it's handled by a global setting.

3. Select Save.

For more information about customizing HTTP functions, see [Azure Functions HTTP bindings](#).

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 it now uses the `/api/hello` path.
3. Copy the URL into 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 it's working. You should see the response, "Hello John."
6. You can also try calling the endpoint with another HTTP method to confirm that the function isn't executed. To do so, use a REST client, such as cURL, Postman, or Fiddler.

Proxies overview

In the next section, you'll surface your API through a proxy. Azure Functions Proxies allows you to forward requests to other resources. You define an HTTP endpoint just like with 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 easy for clients to consume, which 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 proxies, see [Working with Azure Functions Proxies](#).

Create your first proxy

In this section, you create a new proxy, which serves as a frontend to your overall API.

Setting up the frontend environment

Repeat the steps to [Create a function app](#) to create a new function app in which you will create your proxy. This new app's URL serves as the frontend for our API, and the function app you were previously editing serves as a backend.

1. Navigate to your new frontend function app in the portal.
2. Select **Platform Features** and choose **Application Settings**.
3. Scroll down to **Application settings**, where key/value pairs are stored, and create a new setting with the key `HELLO_HOST`. 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 testing your HTTP function. You'll reference this setting in the configuration later.

NOTE

App settings are recommended 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.

4. Select **Save**.

Creating a proxy on the frontend

1. Navigate back to your front-end function app in the portal.
2. In the left-hand menu, select **Proxies**, and then select **Add**.
3. On the **New Proxy** page, use the settings in the following table, and then select **Create**.

| 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 | <code>https://%HELLO_HOST%/api/hello</code> | Specifies the endpoint to which the request should be proxied |

New proxy

Name

Route template

Allowed HTTP methods

▼

Backend URL

+ Request override

+ Response override

Create

Azure Functions Proxies doesn't provide the `/api` base path prefix, which must be included in the route template. The `%HELLO_HOST%` syntax references the app setting you created earlier. The resolved URL will point 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'll use a proxy to create a mock API for your solution. This proxy allows client development to progress, without needing the backend fully implemented. Later in development, you can create a new function app, which supports this logic and redirect your proxy to it.

To create this mock API, we'll create a new proxy, this time using the [App Service Editor](#). To get started, navigate to your function app in the portal. Select **Platform features**, and under **Development Tools** find **App Service Editor**. The App Service Editor opens in a new tab.

Select `proxies.json` in the left navigation. 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. To learn more about this file, see [Proxies advanced configuration](#).

If you've followed along so far, your `proxies.json` should look like as follows:

```
{
    "$schema": "http://json.schemastore.org/proxies",
    "proxies": {
        "HelloProxy": {
            "matchCondition": {
                "route": "/api/remotehello"
            },
            "backendUri": "https://%HELLO_HOST%/api/hello"
        }
    }
}
```

Next, you'll add your mock API. Replace your `proxies.json` file with the following code:

```
{
    "$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`, without the `backendUri` property. Instead of calling another resource, it modifies the default response from Proxies using a response override. Request and response overrides can also be used in conjunction with a backend URL. This technique is particularly useful when proxying to a legacy system, where you might need to modify headers, query parameters, and so on. To learn more about request and response overrides, see [Modifying requests and responses in Proxies](#).

Test your mock API by calling the `<YourProxyApp>.azurewebsites.net/api/users/{username}` endpoint using a browser or your favorite REST client. Be sure to replace `{username}` with a string value representing a username.

Next steps

In this article, you learned how to build and customize an API on Azure Functions. You also learned how to bring multiple APIs, including mocks, 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.

The following references may be helpful as you develop your API further:

- [Azure Functions HTTP bindings](#)
- [Working with Azure Functions Proxies](#)
- [Documenting an Azure Functions API \(preview\)](#)

Managing hybrid environments with PowerShell in Azure Functions and App Service Hybrid Connections

5/12/2020 • 7 minutes to read • [Edit Online](#)

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.

```
# For configuration of WinRM, see
# https://docs.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 |
|---------|-----------------|-------------|
|---------|-----------------|-------------|

| 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

[X](#)

[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](#) [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 *

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * Not finding your plan? Try a different location in Basics tab.

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. |

Home > New > Function App

Function App

Basics Hosting Monitoring Tags Review + create

Azure Monitor gives you full observability into your applications, infrastructure, and network. [Learn more](#)

Application Insights

Enable Application Insights * No Yes

Application Insights * [Create new](#)

Region

Review + create [< Previous](#) [Next : Tags >](#)

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 managing a function app named "myPowerShellFunctionApp-dma". The left sidebar is titled "myPowerShellFunctionApp-dma | Networking" and contains several sections: Deployment Center, Settings (Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, TLS/SSL settings), Networking (selected), Scale up (App Service plan), Scale out (App Service plan), WebJobs, Push, MySQL In App, Properties, Locks, Export template, App Service plan (App Service plan, Quotas, Change App Service plan), Development Tools (Console), and Network Feature Status. The "Networking" section is currently active. On the right, there are four main sections: "VNet Integration" (Securely access resources available in or through your Azure VNet), "Hybrid connections" (Securely access applications in private networks, with a link to "Configure your hybrid connection endpoints" which is highlighted with a red box), "Azure Front Door with Web Application Firewall" (Scalable and secure entry point for accelerated delivery of your web applications), and "Azure CDN" (Secure, reliable content delivery with broad global reach and rich feature set).

3. Select **Add hybrid connection**.

The screenshot shows the "Hybrid connections" blade for a function app named "ContosoPowerShellHybrid". At the top, it displays the app service plan ("ServicePlanb818da2f-b9c1 (Standard)"), the number of connections used (0), and the connection quota (25). Below this, there is a "Download connection manager" button and an "Add hybrid connection" button, which is highlighted with a red box. A table at the bottom lists hybrid connections, showing columns for Name, Status, Endpoint, and Namespace. The table currently displays "No results".

4. Enter information about the hybrid connection as shown right after the following screenshot. You have the

option of making the **Endpoint Host** setting match the host name of the on-premises server to make it easier to remember the server later when you're running remote commands. The port matches the default Windows remote management service port that was defined on the server earlier.

| SETTING | SUGGESTED VALUE |
|------------------------|-------------------------------|
| Hybrid connection name | ContosoHybridOnPremisesServer |
| Endpoint Host | finance1 |
| Endpoint Port | 5986 |
| Servicebus namespace | Create New |
| Location | Pick an available location |
| Name | contosopowershellhybrid |

- Select OK to create the hybrid connection.

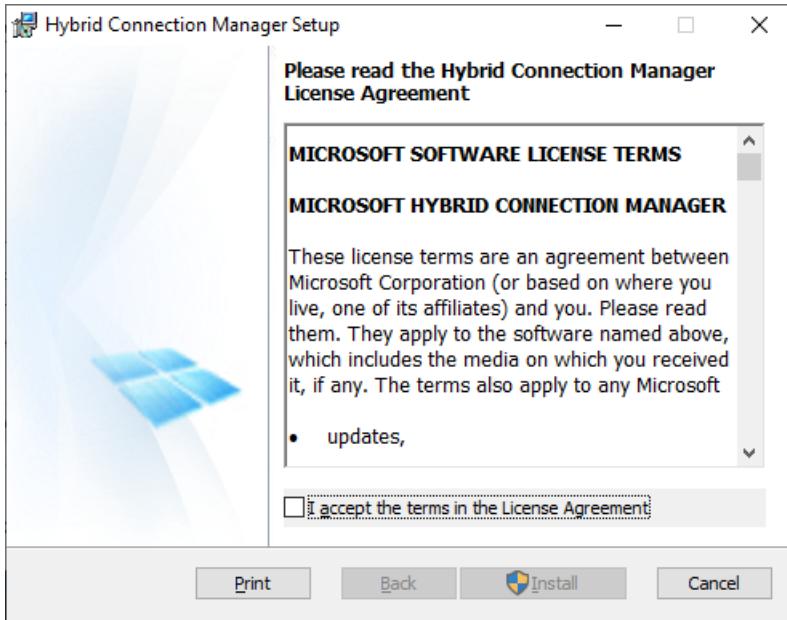
Download and install the hybrid connection

- Select **Download connection manager** to save the **.msi** file locally on your computer.

| NAME | STATUS | ENDPOINT | NAMESPACE |
|---------------------------------|---------------|-----------------|-------------------------|
| ContosoFinance1OnPremisesServer | Not connected | finance1 : 5986 | contosopowershellhybrid |

- Copy the **.msi** file from your local computer to the on-premises server.

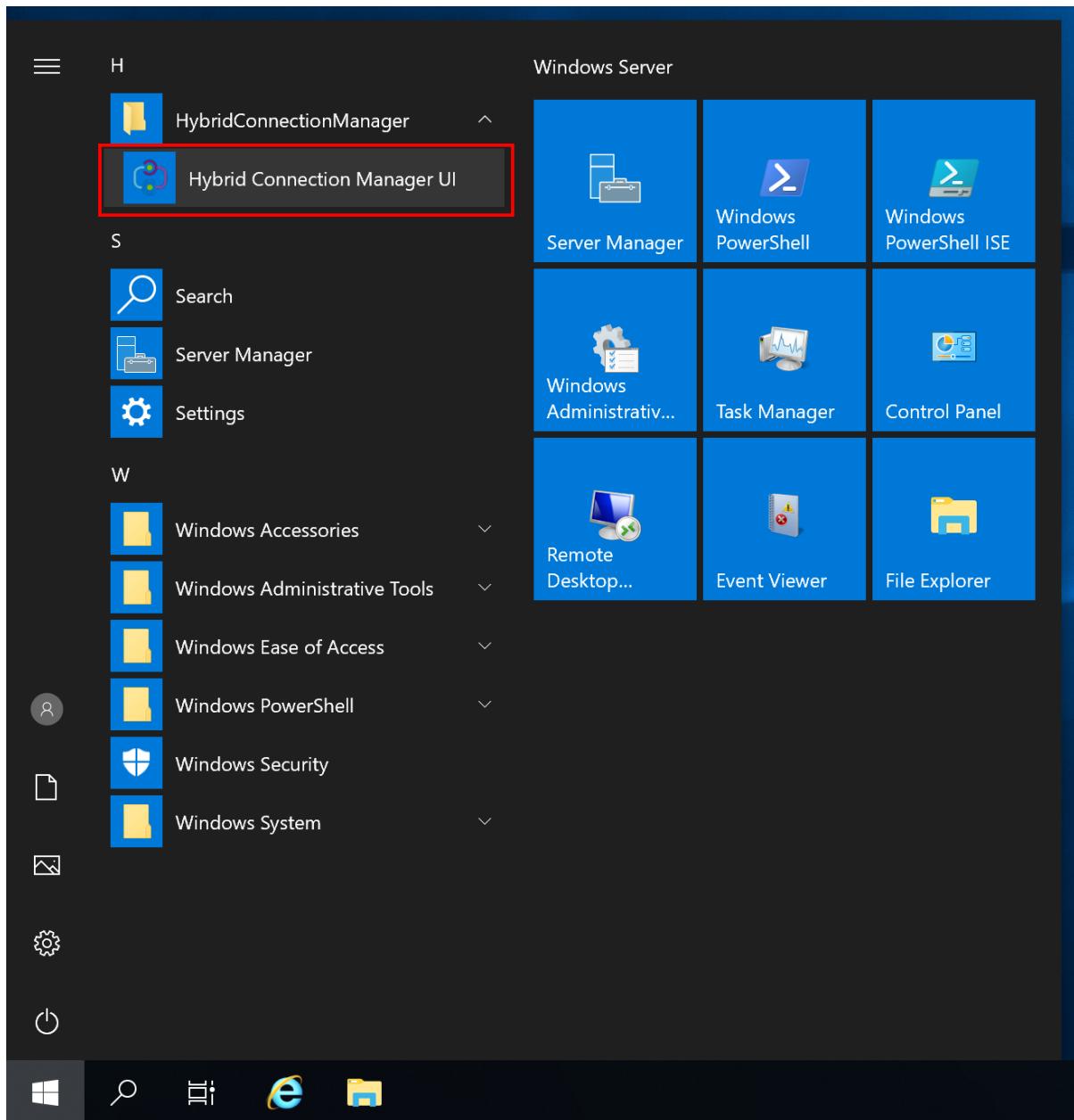
- Run the Hybrid Connection Manager installer to install the service on the on-premises server.



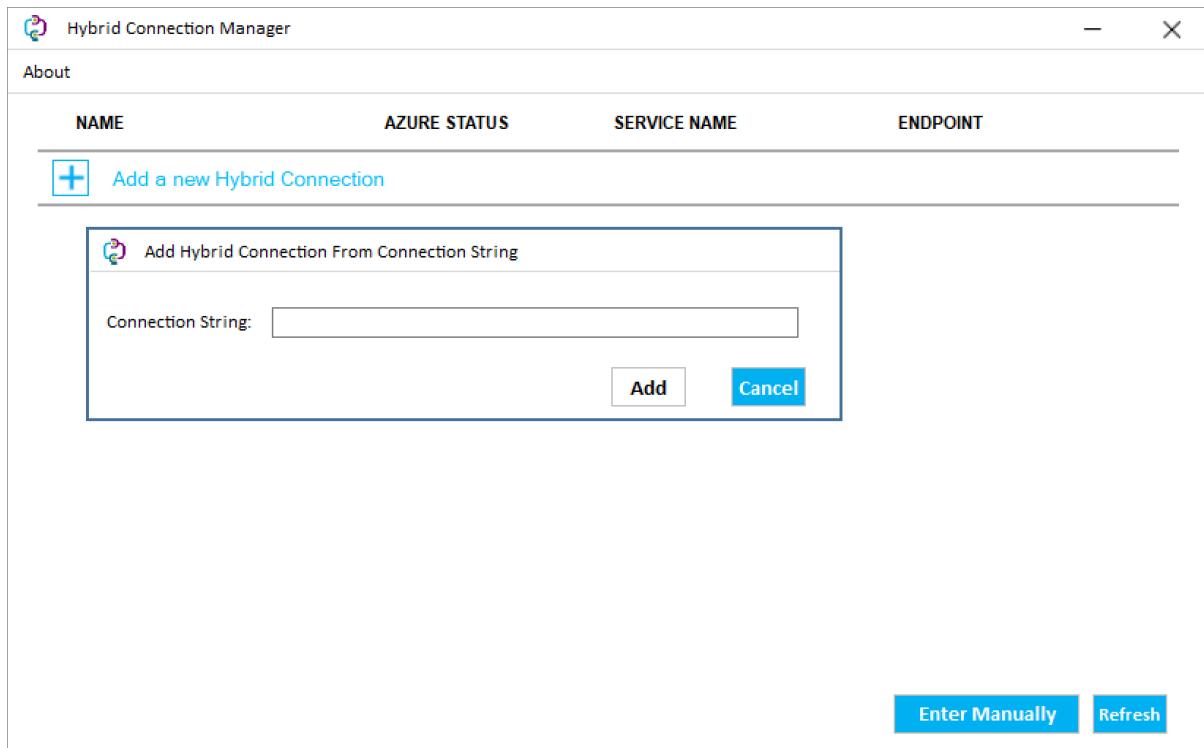
- From the portal, open the hybrid connection and then copy the gateway connection string to the clipboard.

The screenshot shows the Azure portal's 'Hybrid connections' blade for a function app named 'ContosoPowerShellHybrid'. The blade displays basic information like the app service plan ('ServicePlanb818da2f-b9c1 (Standard)'), location ('Central US'), and connection usage ('Connections used: 1', 'Connections quota: 25'). On the right, a 'Properties' panel is open, showing details such as 'HYBRID CONNECTION NAME: ContosoFinance1OnPremisesServer', 'ENDPOINT HOST: finance1', 'ENDPOINT PORT: 5986', 'SERVICE BUS NAMESPACE: contosopowershellhybrid', 'NAMESPACE LOCATION: Central US', and 'HYBRID CONNECTION MANAGERS: 0 connected'. A red box highlights the 'Copy to clipboard' button next to the gateway endpoint URL 'Endpoint=sb://contoso...'.

- 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.

```
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 | Deployments |
|------------------------------------|-------|------|-------------|
| (no connection strings to display) | | | |

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

Search (Ctrl+ /)

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

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 | 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**.

Search (Ctrl+I)[+ Add](#)[Develop Locally](#)[Refresh](#)[Enable](#)[Disable](#)[Delete](#)[Overview](#)[Activity log](#)[Access control \(IAM\)](#)[Tags](#)[Diagnose and solve problems](#)[Security](#)[Events](#)

Functions

[Fx 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](#) Filter by name...[Name ↑↓](#)[Trigger ↑↓](#)[Status ↑↓](#)

No results.

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:

```

# 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 `

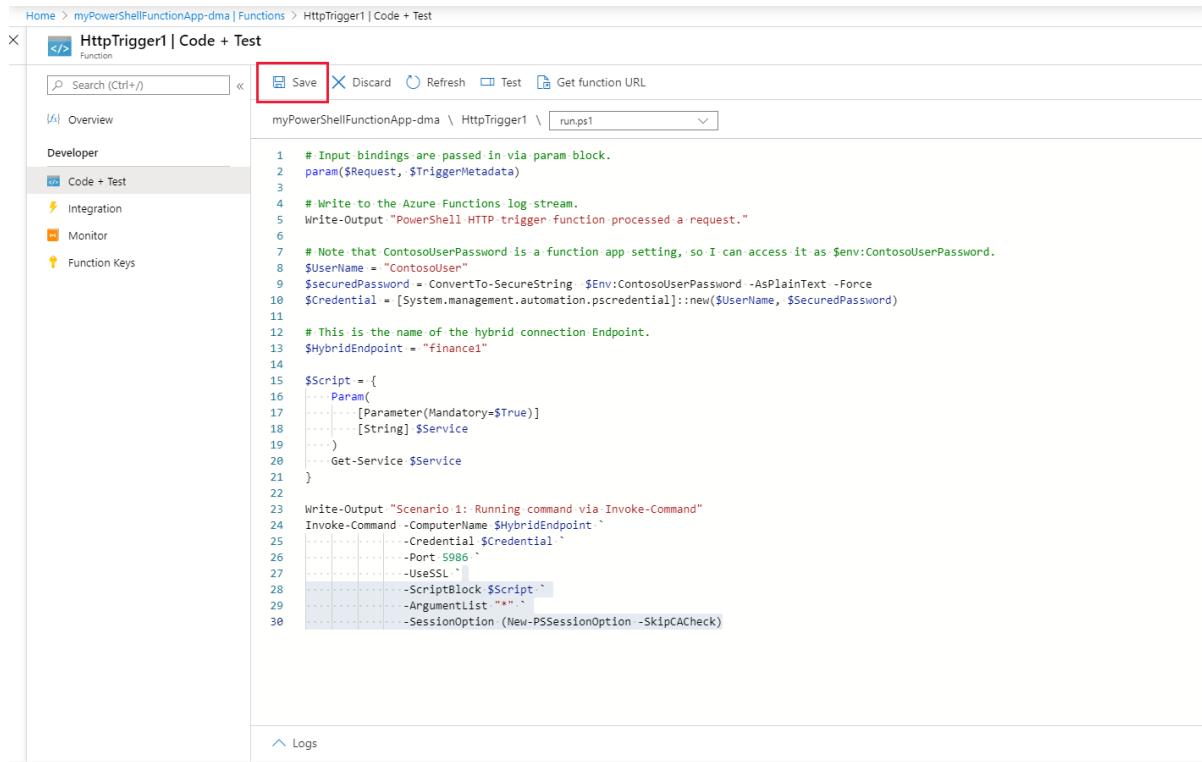
    -ScriptBlock $Script `

    -ArgumentList "" `

    -SessionOption (New-PSSessionOption -SkipCACheck)

```

2. Select Save.



3. Select Test, and then select Run to test the function. Review the logs to verify that the test was successful.

```
1 Input bindings are passed in via param block.
2 $request, $triggerMetadata
3
4 Write to the Azure Functions log stream.
5 $output = "PowerShell HTTP trigger function processed a request"
6
7 Note that ContosoUserPassword is a function app setting.
8 $contosoUser = "ContosoUser"
9 $contosoPassword = ConvertTo-SecureString $Env:ContosoUserPassword -AsPlainText -Force
10 $contosoCredential = [System.Management.Automation.PSCredential]::New($contosoUser, $contosoPassword)
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 $output = "Scenario 1: Running command via Invoke-Command"
24 Invoke-Command -ComputerName $hybridEndpoint `-
25     -Credential $contosoCredential `-
26     -Port 5986 `-
27     -UseSSL `-
28     -ScriptBlock $script `-
29     -ArgumentList "Azure" `-
30     -SessionOption (New-PSSessionOption -SkipCACheck)
```

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.

```

# 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"

12/4/2020 • 3 minutes to read • [Edit Online](#)

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 Azure Functions Runtime can't start. The most common reason for the issue 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 the following 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 function 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 wrapped up in 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 Consumption and Premium 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 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](#).

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 runtime 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 App Service Environment configuration:

1. Go to the network security group (NSG) of the subnet where the App Service Environment 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](#).

Next steps

Learn about monitoring your function apps:

[Monitor Azure Functions](#)

Troubleshoot Python errors in Azure Functions

11/2/2020 • 7 minutes to read • [Edit Online](#)

Following is a list of troubleshooting guides for common issues in Python functions:

- [ModuleNotFoundError and ImportError](#)
- [Cannot import 'cygrpc'](#)

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 issue 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 only supports Windows or 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 `WEBSITE_RUN_FROM_PACKAGE` app setting and its value is a URL, download the file by copy and paste the URL into your browser.
- If the function app has `WEBSITE_RUN_FROM_PACKAGE` and it is set to `1`, navigate 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 the app setting mentioned above, navigate to `https://<app-name>.scm.azurewebsites.net/api/settings` and find the URL under `SCM_RUN_FROM_PACKAGE`. Download the file by copy and paste the URL into your browser.
- If none of these works for you, navigate to `https://<app-name>.scm.azurewebsites.net/DebugConsole` and reveal 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

Browse 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 may cause this issue.

See [Enable remote build](#) or [Build native dependencies](#) for mitigation.

The package isn't resolved with 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. If the value of the tag doesn't contain `linux`, this could be the issue.

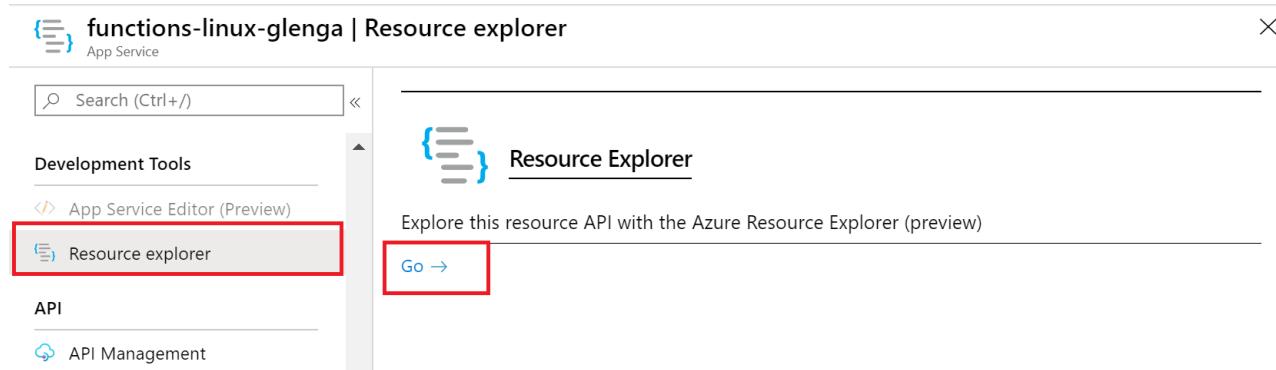
Python functions run only on Linux in Azure: Functions runtime v2.x runs on Debian Stretch and the v3.x runtime on Debian Buster. The artifact is expected to contain the correct Linux binaries. Using `--build local` flag in Core Tools, third-party, or outdated tools may cause older binaries to be used.

See [Enable remote build](#) or [Build native dependencies](#) for mitigation.

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`. Using a text editor, open the **METADATA** file and check the **Classifiers:** section. If the section doesn't contains `Python :: 3`, `Python :: 3.6`, `Python :: 3.7`, or `Python :: 3.8`, this means the package version is either too old, or most likely, the package is already out of maintenance.

You can check the Python version of your function app from the [Azure portal](#). Navigate to your function app, choose **Resource explorer**, and select **Go**.



After the explorer loads, search for **LinuxFxVersion**, which shows the Python version.

See [Update your package to the latest version](#) or [Replace the package with equivalents](#) for mitigation.

The package conflicts with other packages

If you have verified that the package is resolved correctly with the proper Linux wheels, there may be a conflict with other packages. In certain packages, the PyPi documentations may clarify the incompatible modules. For example in `azure 4.0.0`, there's a statement as follows:

```
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>>.

See [Update your package to the latest version](#) or [Replace the package with equivalents](#) for mitigation.

The package only supports Windows or macOS platforms

Open the `requirements.txt` with a text editor and check the package in <https://pypi.org/project/<package-name>>. Some packages only run on Windows or macOS platforms. For example, `pywin32` only runs on Windows.

The `Module Not Found` error may 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 is likely to be caused by using `pip freeze` to export virtual environment into `requirements.txt` from your Windows or macOS machine during project initialization.

See [Replace the package with equivalents](#) or [Handcraft requirements.txt](#) for mitigation.

Mitigate ModuleNotFoundError

The following are potential mitigations for module-related issues. Use the [diagnoses above](#) to determine which of these mitigations to try.

Enable remote build

Make sure that remote build is enabled. The way that you do this depends on your deployment method.

- [Visual Studio Code](#)
- [Azure Functions Core Tools](#)
- [Manual publishing](#)

Make sure that the latest version of the [Azure Functions extension for Visual Studio Code](#) is installed. Verify that `.vscode/settings.json` exists and it contains the setting `"azureFunctions.scmDoBuildDuringDeployment": true`. If not, please create this file with the `azureFunctions.scmDoBuildDuringDeployment` setting enabled and redeploy the project.

Build native dependencies

Make sure that the latest version of both [docker](#) and [Azure Functions Core Tools](#) is 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

Browse the latest package version in <https://pypi.org/project/<package-name>> and 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`, or `Python :: 3.8`.

If these 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 running on Linux.

The best practice is to check the import statement from each .py file in your project source code and only check-in those modules in `requirements.txt` file. This guarantees the resolution of packages can be handled properly on different operating systems.

Replace the package with equivalents

First, we should take a look into the latest version of the package in <https://pypi.org/project/<package-name>>. Usually, this package has their own GitHub page, go to the **Issues** section on GitHub and search if your issue has been fixed. If so, update the package to the latest version.

Sometimes, the package may have been integrated into [Python Standard Library](#) (such as `pathlib`). If so, since we provide a certain Python distribution in Azure Functions (Python 3.6, Python 3.7, and Python 3.8), the package in your `requirements.txt` should be removed.

However, if you're facing an issue that it has not been fixed and you're on a deadline. I encourage you to do some research and find a similar package for your project. Usually, the Python community will provide you with a wide

variety of similar libraries that you can use.

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 issue 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 is not supported by Azure Functions Python Worker](#)

Diagnose 'cygrpc' reference error

The Python interpreter mismatches OS architecture

This is most likely caused by a 32-bit Python interpreter is installed on your 64-bit operating system.

If you're running on an x64 operating system, please ensure your Python 3.6, 3.7, or 3.8 interpreter is also on 64-bit version.

You can check your Python interpreter bitness by the following commands:

On Windows in PowerShell: `py -c 'import platform; print(platform.architecture()[0])'`

On Unix-like shell: `python3 -c 'import platform; print(platform.architecture()[0])'`

If there's a mismatch between Python interpreter bitness and operating system architecture, please download a proper Python interpreter from [Python Software Foundation](#).

The Python interpreter is not supported by Azure Functions Python Worker

The Azure Functions Python Worker only supports Python 3.6, 3.7, and 3.8. Please check if your Python interpreter matches our expected version by `py --version` in Windows or `python3 --version` in Unix-like systems. Ensure the return result is Python 3.6.x, Python 3.7.x, or Python 3.8.x.

If your Python interpreter version does not meet our expectation, please download the Python 3.6, 3.7, or 3.8 interpreter from [Python Software Foundation](#).

Next steps

If you're unable to resolve your issue, please report this to the Functions team:

[Report an unresolved issue](#)

App settings reference for Azure Functions

12/4/2020 • 10 minutes to read • [Edit Online](#)

App settings in a function app contain global configuration options that affect all functions for that function app. When you run locally, these settings are accessed as local [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](#).

There are other global configuration options in the [host.json](#) file and in the [local.settings.json](#) file.

NOTE

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](#).

APPINSIGHTS_INSTRUMENTATIONKEY

The instrumentation key for Application Insights. Only use one of `APPINSIGHTS_INSTRUMENTATIONKEY` or `APPLICATIONINSIGHTS_CONNECTION_STRING`. When Application Insights runs in a sovereign cloud, use `APPLICATIONINSIGHTS_CONNECTION_STRING`. For more information, see [How to configure monitoring for Azure Functions](#).

| KEY | SAMPLE VALUE |
|--------------------------------|-------------------------------------|
| APPINSIGHTS_INSTRUMENTATIONKEY | 55555555-af77-484b-9032-64f83bb83bb |

APPLICATIONINSIGHTS_CONNECTION_STRING

The connection string for Application Insights. Use `APPLICATIONINSIGHTS_CONNECTION_STRING` instead of `APPINSIGHTS_INSTRUMENTATIONKEY` 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](#).

| KEY | SAMPLE VALUE |
|---------------------------------------|--|
| APPLICATIONINSIGHTS_CONNECTION_STRING | InstrumentationKey=[key];IngestionEndpoint=[url];LiveEndpoint=[url];ProfilerEndpoint=[url];SnapshotEndpoint=[url]; |

AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL

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.

| 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. This is the default value. |

AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES

This setting controls whether the characters `%2F` are decoded as slashes in route parameters when they are inserted into the backend URL.

| 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.

```
{  
    "$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

In version 2.x and later versions of the Functions runtime, configures app behavior based on the runtime environment. This value is [read during initialization](#). You can set `AZURE_FUNCTIONS_ENVIRONMENT` to any value, but [three values](#) are supported: [Development](#), [Staging](#), and [Production](#). When `AZURE_FUNCTIONS_ENVIRONMENT` isn't set, it defaults to `Development` on a local environment and `Production` on Azure. This setting should be used instead

of `ASPNETCORE_ENVIRONMENT` to set the runtime environment.

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](#).

AzureWebJobsDashboard

Optional storage account connection string for storing logs and displaying them in the **Monitor** tab in the portal. This setting is only valid for apps that target version 1.x of the Azure Functions runtime. The storage account must be a general-purpose one that supports blobs, queues, and tables. To learn more, see [Storage account requirements](#).

| KEY | SAMPLE VALUE |
|------------------------------------|--|
| <code>AzureWebJobsDashboard</code> | <code>DefaultEndpointsProtocol=https;AccountName=;AccountKey=</code> |

NOTE

For better performance and experience, runtime version 2.x and later versions use `APPINSIGHTS_INSTRUMENTATIONKEY` and App Insights for monitoring instead of `AzureWebJobsDashboard`.

AzureWebJobsDisableHomepage

`true` means disable the default landing page that is shown for the root URL of a function app. Default is `false`.

| KEY | SAMPLE VALUE |
|--|-------------------|
| <code>AzureWebJobsDisableHomepage</code> | <code>true</code> |

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`.

| KEY | SAMPLE VALUE |
|---|-------------------|
| <code>AzureWebJobsDotNetReleaseCompilation</code> | <code>true</code> |

AzureWebJobsFeatureFlags

A comma-delimited list of beta features to enable. Beta features enabled by these flags are not production ready, but can be enabled for experimental use before they go live.

| KEY | SAMPLE VALUE |
|---------------------------------------|--------------------------------|
| <code>AzureWebJobsFeatureFlags</code> | <code>feature1,feature2</code> |

AzureWebJobsSecretStorageType

Specifies the repository or provider to use for key storage. Currently, the supported repositories are blob storage ("Blob") and the local file system ("Files"). The default is blob in version 2 and file system in version 1.

| KEY | SAMPLE VALUE |
|--|--------------------|
| <code>AzureWebJobsSecretStorageType</code> | <code>Files</code> |

AzureWebJobsStorage

The Azure Functions runtime uses this storage account connection string for normal operation. Some uses of

this storage account 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. See [Storage account](#) and [Storage account requirements](#).

| KEY | SAMPLE VALUE |
|---------------------|--|
| AzureWebJobsStorage | DefaultEndpointsProtocol=https;AccountName=[name];AccountKey=[key] |

AzureWebJobs_TypeScriptPath

Path to the compiler used for TypeScript. Allows you to override the default if you need to.

| KEY | SAMPLE VALUE |
|-----------------------------|-------------------|
| AzureWebJobs_TypeScriptPath | %HOME%\typescript |

FUNCTION_APP_EDIT_MODE

Dictates whether editing in the Azure portal is enabled. Valid values are "readwrite" and "readonly".

| KEY | SAMPLE VALUE |
|------------------------|--------------|
| FUNCTION_APP_EDIT_MODE | readonly |

FUNCTIONS_EXTENSION_VERSION

The version of the Functions runtime to use in this function app. A tilde with major version means use the latest version of that major version (for example, "~2"). When new versions for the same major version are available, they are automatically installed in the function app. To pin the app to a specific version, use the full version number (for example, "2.0.12345"). Default is "~2". A value of `~1` pins your app to version 1.x of the runtime.

| KEY | SAMPLE VALUE |
|-----------------------------|--------------|
| FUNCTIONS_EXTENSION_VERSION | ~2 |

FUNCTIONS_V2_COMPATIBILITY_MODE

This setting enables your function app to run in a version 2.x compatible mode on the version 3.x runtime. Use this setting only if encountering issues when [upgrading your function app from version 2.x to 3.x of the runtime](#).

IMPORTANT

This setting is intended only as a short-term workaround while you update your app to run correctly on version 3.x. This setting is supported as long as the [2.x runtime is supported](#). If you encounter issues that prevent your app from running on version 3.x without using this setting, please [report your issue](#).

Requires that [FUNCTIONS_EXTENSION_VERSION](#) be set to `~3`.

| KEY | SAMPLE VALUE |
|---------------------------------|--------------|
| FUNCTIONS_V2_COMPATIBILITY_MODE | true |

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 is not 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 non-.NET languages.

| KEY | SAMPLE VALUE |
|--------------------------------|--------------|
| FUNCTIONS_WORKER_PROCESS_COUNT | 2 |

FUNCTIONS_WORKER_RUNTIME

The language worker runtime to load in the function app. This will correspond to the language being used in your application (for example, "dotnet"). For functions in multiple languages you will need to publish them to multiple apps, each with a corresponding worker runtime value. Valid values are `dotnet` (C#/F#), `node` (JavaScript/TypeScript), `java` (Java), `powershell` (PowerShell), and `python` (Python).

| KEY | SAMPLE VALUE |
|--------------------------|--------------|
| FUNCTIONS_WORKER_RUNTIME | dotnet |

PIP_EXTRA_INDEX_URL

The value for this setting indicates a custom package index URL for Python apps. Use this setting when you need to run a remote build using custom dependencies that are found in an extra package index.

| KEY | SAMPLE VALUE |
|---------------------|---|
| PIP_EXTRA_INDEX_URL | <code>http://my.custom.package.repo/simple</code> |

To learn more, see [Custom dependencies](#) in the Python developer reference.

SCALE_CONTROLLER_LOGGING_ENABLE

This setting is currently in preview.

This setting controls logging from the Azure Functions scale controller. For more information, see [Scale controller logs](#).

| KEY | SAMPLE VALUE |
|---------------------------------|----------------------------------|
| SCALE_CONTROLLER_LOGGING_ENABLE | <code>AppInsights:Verbose</code> |

The value for this key is supplied in the format `<DESTINATION>:<VERBOSITY>`, which is defined as follows:

| | |
|---|--|
| <p><code><DESTINATION></code></p> | <p>The destination to which logs are sent. Valid values are <code>AppInsights</code> and <code>Blob</code>. When you use <code>AppInsights</code>, make sure 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.</p> |
| <p><code><VERBOSITY></code></p> | <p>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, as well as 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.

WEBSITE_CONTENTAZUREFILECONNECTIONSTRING

For Consumption & Premium plans only. Connection string for storage account where the function app code and configuration are stored. See [Create a function app](#).

| KEY | SAMPLE VALUE |
|---|---|
| <code>WEBSITE_CONTENTAZUREFILECONNECTIONSTRING</code> | <code>DefaultEndpointsProtocol=https;AccountName=[name];AccountKey=[key]</code> |

WEBSITE_CONTENTOVERVNET

For Premium plans only. 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. To learn more, see [Restrict your storage account to a virtual network](#).

| KEY | SAMPLE VALUE |
|--------------------------------------|----------------|
| <code>WEBSITE_CONTENTOVERVNET</code> | <code>1</code> |

WEBSITE_CONTENTSHARE

For Consumption & Premium plans only. The file path to the function app code and configuration. Used with `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`. Default is a unique string that begins with the function app name. See [Create a function app](#).

| KEY | SAMPLE VALUE |
|----------------------|---------------------|
| WEBSITE_CONTENTSHARE | functionapp091999e2 |

WEBSITE_MAX_DYNAMIC_APPLICATION_SCALE_OUT

The maximum number of instances that the function 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.

| KEY | SAMPLE VALUE |
|---|--------------|
| WEBSITE_MAX_DYNAMIC_APPLICATION_SCALE_OUT | 5 |

WEBSITE_NODE_DEFAULT_VERSION

Windows only.

Sets the version of Nodejs 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 `~10`, the latest version of Node.js 10 is used. When a major version is targeted with a tilde, you don't have to manually update the minor version.

| KEY | SAMPLE VALUE |
|------------------------------|------------------|
| WEBSITE_NODE_DEFAULT_VERSION | <code>~10</code> |

WEBSITE_RUN_FROM_PACKAGE

Enables your function app to run from a mounted package file.

| KEY | SAMPLE VALUE |
|--------------------------|--------------|
| WEBSITE_RUN_FROM_PACKAGE | 1 |

Valid values are either a URL that resolves to the location of a deployment package file, or `1`. When set to `1`, the package must be in the `d:\home\data\SitePackages` folder. When using zip deployment with this setting, 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](#).

WEBSITE_TIME_ZONE

Allows you to set the timezone for your function app.

| 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.

| 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 Dedicated | Set the value to the name of the desired time zone as shown in the tz database . |

NOTE

`WEBSITE_TIME_ZONE` is not currently supported on the Linux Consumption plan.

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.

Next steps

[Learn how to update app settings](#)

[See global settings in the host.json file](#)

[See other app settings for App Service apps](#)

Azure Blob storage bindings for Azure Functions overview

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 |

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

Azure Storage SDK version in Functions 1.x

In Functions 1.x, the Storage triggers and bindings use version 7.2.1 of the Azure Storage SDK ([WindowsAzure.Storage](#) NuGet package). If you reference a different version of the Storage SDK, and you bind to a Storage SDK type in your function signature, the Functions runtime may report that it can't bind to that type. The solution is to make sure your project references [WindowsAzure.Storage 7.2.1](#).

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

12/4/2020 • 12 minutes to read • [Edit Online](#)

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](#).

The Azure Blob storage trigger requires a general-purpose storage account. Storage V2 accounts with [hierarchical namespaces](#) are also supported. To use a blob-only account, or if your application has specialized needs, review the alternatives to using this trigger.

For information on setup and configuration details, see the [overview](#).

Polling

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.

WARNING

In addition, [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, consider creating a [queue message](#) when you create the blob. Then use a [queue trigger](#) instead of a blob trigger to process the blob. Another option is to use Event Grid; see the tutorial [Automate resizing uploaded images using Event Grid](#).

Alternatives

Event Grid trigger

The [Event Grid trigger](#) also has built-in support for [blob events](#). Use Event Grid instead of the Blob storage trigger for the following scenarios:

- **Blob-only storage accounts:** [Blob-only storage accounts](#) are supported for blob input and output bindings but not for blob triggers.
- **High-scale:** 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.
- **Minimizing latency:** 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. To avoid this latency, you can switch to an App Service plan with Always On enabled. You can also use an [Event Grid trigger](#) with your Blob storage account. For an example, see the [Event Grid tutorial](#).

See the [Image resize with Event Grid](#) tutorial of an Event Grid example.

Queue storage trigger

Another approach to processing blobs is to write queue messages that correspond to blobs being created or modified and then use a [Queue storage trigger](#) to begin processing.

Example

- [C#](#)

- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that writes a log when a blob is added or updated in the `samples-workitems` container.

```
[FunctionName("BlobTriggerCSharp")]
public static void Run([BlobTrigger("samples-workitems/{name}")]) Stream myBlob, string name, ILogger log
{
    log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");
}
```

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](#) later in this article.

For more information about the `BlobTrigger` attribute, see [attributes and annotations](#).

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the following attributes to configure a blob trigger:

- [BlobTriggerAttribute](#)

The attribute's constructor takes a path string that indicates the container to watch and optionally a [blob name pattern](#). Here's an example:

```
[FunctionName("ResizeImage")]
public static void Run(
    [BlobTrigger("sample-images/{name}")] Stream image,
    [Blob("sample-images-md/{name}", FileAccess.Write)] Stream imageSmall)
{
    ....
}
```

You can set the `Connection` property to specify the storage account to use, as shown in the following example:

```
[FunctionName("ResizeImage")]
public static void Run(
    [BlobTrigger("sample-images/{name}", Connection = "StorageConnectionAppSetting")] Stream image,
    [Blob("sample-images-md/{name}", FileAccess.Write)] Stream imageSmall)
{
    ....
}
```

For a complete example, see [Trigger example](#).

- [StorageAccountAttribute](#)

Provides another way to specify the storage account to use. 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:

```
[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 ("AzureWebJobsStorage" app setting).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `BlobTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|-----------------------|---|
| <code>type</code> | n/a | Must be set to <code>blobTrigger</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>direction</code> | n/a | Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal. Exceptions are noted in the usage section. |
| <code>name</code> | n/a | The name of the variable that represents the blob in function code. |
| <code>path</code> | <code>BlobPath</code> | The container to monitor. May be a blob name pattern . |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| connection | Connection | <p>The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code>.</p> <p>The connection string must be for a general-purpose storage account, not a Blob storage account.</p> |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

You can use the following parameter types for the triggering blob:

- `Stream`
- `TextReader`
- `string`
- `Byte[]`
- `ICloudBlob`¹
- `CloudBlockBlob`¹
- `CloudPageBlob`¹
- `CloudAppendBlob`¹

¹ Requires "inout" binding `direction` in `function.json` or `FileAccess.ReadWrite` in a C# class library.

If you try to bind to one of the Storage SDK types and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

Binding to `string`, or `Byte[]` is only recommended if the blob size is small, as the entire blob contents are loaded into memory. Generally, it is preferable to use a `Stream` or `CloudBlockBlob` type. For more information, see [Concurrency and memory usage](#) later in this article.

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.

Get file name and extension

The following example shows how to bind to the blob file name and extension separately:

```
"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-":

```
"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:

```
"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:

```
"path": "images/{{20140101}}-{name}",
```

If the blob is named *{20140101}-soundfile.mp3*, the `name` variable value in the function code is *soundfile.mp3*.

Metadata

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

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.

| 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> | BlobProperties | The blob's system properties. |

| PROPERTY | TYPE | DESCRIPTION |
|----------|-----------------------------|---|
| Metadata | IDictionary<string, string> | The user-defined metadata for the blob. |

For example, the following C# script and JavaScript examples log the path to the triggering blob, including the container:

```
public static void Run(string myBlob, string blobTrigger, ILogger log)
{
    log.LogInformation($"Full blob path: {blobTrigger}");
}
```

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 5 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")

Concurrency and memory usage

The blob trigger uses a queue internally, so the maximum number of concurrent function invocations is

controlled by the [queues configuration in host.json](#). The default settings limit concurrency to 24 invocations. This limit applies separately to each function that uses a blob trigger.

The [Consumption plan](#) limits a function app on one virtual machine (VM) to 1.5 GB of memory. Memory is used by each concurrently executing function instance and by the Functions runtime itself. If a blob-triggered function loads the entire blob into memory, the maximum memory used by that function just for blobs is $24 * \text{maximum blob size}$. For example, a function app with three blob-triggered functions and the default settings would have a maximum per-VM concurrency of $3 * 24 = 72$ function invocations.

JavaScript and Java functions load the entire blob into memory, and C# functions do that if you bind to `string`, or `Byte[]`.

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

12/4/2020 • 7 minutes to read • [Edit Online](#)

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

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example is a [C# function](#) that uses a queue trigger and an input blob binding. The queue message contains the name of the blob, and the function logs the size of the blob.

```
[FunctionName("BlobInput")]
public static void Run(
    [QueueTrigger("myqueue-items")] string myQueueItem,
    [Blob("samples-workitems/{queueTrigger}", FileAccess.Read)] Stream myBlob,
    ILogger log)
{
    log.LogInformation($"BlobInput processed blob\n Name:{myQueueItem} \n Size: {myBlob.Length} bytes");
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the [BlobAttribute](#).

The attribute's constructor takes the path to the blob and a `FileAccess` parameter indicating read or write, as shown in the following example:

```
[FunctionName("BlobInput")]
public static void Run(
    [QueueTrigger("myqueue-items")] string myQueueItem,
    [Blob("samples-workitems/{queueTrigger}", FileAccess.Read)] Stream myBlob,
    ILogger log)
{
    log.LogInformation($"BlobInput processed blob\n Name:{myQueueItem} \n Size: {myBlob.Length} bytes");
}
```

You can set the `Connection` property to specify the storage account to use, as shown in the following example:

```
[FunctionName("BlobInput")]
public static void Run(
    [QueueTrigger("myqueue-items")] string myQueueItem,
    [Blob("samples-workitems/{queueTrigger}", FileAccess.Read, Connection = "StorageConnectionAppSetting")]
    Stream myBlob,
    ILogger log)
{
    log.LogInformation($"BlobInput processed blob\n Name:{myQueueItem} \n Size: {myBlob.Length} bytes");
}
```

You can use the `StorageAccount` attribute to specify the storage account at class, method, or parameter level. For more information, see [Trigger - attributes and annotations](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Blob` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------|-------------------------|--|
| <code>type</code> | n/a | Must be set to <code>blob</code> . |
| <code>direction</code> | n/a | Must be set to <code>in</code> . Exceptions are noted in the usage section. |
| <code>name</code> | n/a | The name of the variable that represents the blob in function code. |
| <code>path</code> | <code>BlobPath</code> | The path to the blob. |
| <code>connection</code> | <code>Connection</code> | <p>The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage". If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code>.</p> <p>The connection string must be for a general-purpose storage account, not a blob-only storage account.</p> |
| <code>dataType</code> | n/a | For dynamically typed languages, specifies the underlying data type. Possible values are <code>string</code> , <code>binary</code> , or <code>stream</code> . For more detail, refer to the triggers and bindings concepts . |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| n/a | Access | Indicates whether you will be reading or writing. |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

You can use the following parameter types for the blob input binding:

- `Stream`
- `TextReader`
- `string`
- `Byte[]`
- `CloudBlobContainer`
- `CloudBlobDirectory`
- `ICloudBlob`¹
- `CloudBlockBlob`¹
- `CloudPageBlob`¹
- `CloudAppendBlob`¹

¹ Requires "inout" binding `direction` in `function.json` or `FileAccess.ReadWrite` in a C# class library.

If you try to bind to one of the Storage SDK types and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

Binding to `string` or `Byte[]` is only recommended if the blob size is small, as the entire blob contents are loaded into memory. Generally, it is preferable to use a `Stream` or `CloudBlockBlob` type. For more information, see [Concurrency and memory usage](#) earlier in this article.

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

11/2/2020 • 8 minutes to read • [Edit Online](#)

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

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example is a [C# function](#) that uses a blob trigger and two output blob bindings. The function is triggered by the creation of an image blob in the *sample-images* container. It creates small and medium size copies of the image blob.

```

using System.Collections.Generic;
using System.IO;
using Microsoft.Azure.WebJobs;
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats;
using SixLabors.ImageSharp.PixelFormats;
using SixLabors.ImageSharp.Processing;

public class ResizeImages
{
    [FunctionName("ResizeImage")]
    public static void Run([BlobTrigger("sample-images/{name}")] Stream image,
        [Blob("sample-images-sm/{name}", FileAccess.Write)] Stream imageSmall,
        [Blob("sample-images-md/{name}", FileAccess.Write)] Stream imageMedium)
    {
        IImageFormat format;

        using (Image<Rgba32> input = Image.Load<Rgba32>(image, out format))
        {
            ResizeImage(input, imageSmall, ImageSize.Small, format);
        }

        image.Position = 0;
        using (Image<Rgba32> input = Image.Load<Rgba32>(image, out format))
        {
            ResizeImage(input, imageMedium, ImageSize.Medium, format);
        }
    }

    public static void ResizeImage(Image<Rgba32> input, Stream output, ImageSize size, IImageFormat format)
    {
        var dimensions = imageDimensionsTable[size];

        input.Mutate(x => x.Resize(dimensions.Item1, dimensions.Item2));
        input.Save(output, format);
    }

    public enum ImageSize { ExtraSmall, Small, Medium }

    private static Dictionary<ImageSize, (int, int)> imageDimensionsTable = new Dictionary<ImageSize, (int, int)>()
    {
        { ImageSize.ExtraSmall, (320, 200) },
        { ImageSize.Small, (640, 400) },
        { ImageSize.Medium, (800, 600) }
    };
}

```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the [BlobAttribute](#).

The attribute's constructor takes the path to the blob and a [FileAccess](#) parameter indicating read or write, as shown in the following example:

```
[FunctionName("ResizeImage")]
public static void Run(
    [BlobTrigger("sample-images/{name}")] Stream image,
    [Blob("sample-images-md/{name}", FileAccess.Write)] Stream imageSmall)
{
    ...
}
```

You can set the `Connection` property to specify the storage account to use, as shown in the following example:

```
[FunctionName("ResizeImage")]
public static void Run(
    [BlobTrigger("sample-images/{name}")] Stream image,
    [Blob("sample-images-md/{name}", FileAccess.Write, Connection = "StorageConnectionAppSetting")] Stream
imageSmall)
{
    ...
}
```

For a complete example, see [Output example](#).

You can use the `StorageAccount` attribute to specify the storage account at class, method, or parameter level.

For more information, see [Trigger - attributes](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Blob` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|-----------------------|--|
| <code>type</code> | n/a | Must be set to <code>blob</code> . |
| <code>direction</code> | n/a | Must be set to <code>out</code> for an output binding. Exceptions are noted in the usage section. |
| <code>name</code> | n/a | The name of the variable that represents the blob in function code. Set to <code>\$return</code> to reference the function return value. |
| <code>path</code> | <code>BlobPath</code> | The path to the blob container. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| connection | Connection | <p>The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code>.</p> <p>The connection string must be for a general-purpose storage account, not a blob-only storage account.</p> |
| n/a | Access | Indicates whether you will be reading or writing. |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

You can bind to the following types to write blobs:

- `TextWriter`
- `out string`
- `out Byte[]`
- `CloudBlobStream`
- `Stream`
- `CloudBlobContainer`¹
- `CloudBlobDirectory`
- `ICloudBlob`²
- `CloudBlockBlob`²
- `CloudPageBlob`²
- `CloudAppendBlob`²

¹ Requires "in" binding `direction` in `function.json` or `FileAccess.Read` in a C# class library. However, you can use the container object that the runtime provides to do write operations, such as uploading blobs to the container.

² Requires "inout" binding `direction` in `function.json` or `FileAccess.ReadWrite` in a C# class library.

If you try to bind to one of the Storage SDK types and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

Binding to `string` or `Byte[]` is only recommended if the blob size is small, as the entire blob contents are loaded into memory. Generally, it is preferable to use a `Stream` or `CloudBlockBlob` type. For more information, see [Concurrency and memory usage](#) earlier in this article.

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](#)

Azure Cosmos DB bindings for Azure Functions 1.x

11/2/2020 • 26 minutes to read • [Edit Online](#)

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 Functions version 1.x, only the trigger was renamed Cosmos DB; the input binding, output binding, and NuGet package retain the DocumentDB name.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: [C#](#), [JavaScript](#), [Java](#), or [Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#](#), [C# script](#), [F#](#), [Java](#), [JavaScript](#), or [Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

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's API for MongoDB](#), [Cassandra API](#), [Gremlin API](#), and [Table API](#).

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#](#)
- [C# Script](#)
- [JavaScript](#)

The following example shows a [C# function](#) that is invoked when there are inserts or updates in the specified database and collection.

```
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#](#)
- [C# Script](#)
- [JavaScript](#)

In [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:

```
[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 create the trigger in the Azure portal. |
| <code>name</code> | n/a | The variable name used in function code that represents the list of documents with changes. |
| <code>connectionStringSetting</code> | <code>ConnectionStringSetting</code> | The name of an app setting that contains the connection string used to connect to the Azure Cosmos DB account being monitored. |
| <code>databaseName</code> | <code>DatabaseName</code> | The name of the Azure Cosmos DB database with the collection being monitored. |
| <code>collectionName</code> | <code>CollectionName</code> | The name of the collection being monitored. |
| <code>leaseConnectionStringSetting</code> | <code>LeaseConnectionStringSetting</code> | (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. |
| <code>leaseDatabaseName</code> | <code>LeaseDatabaseName</code> | (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. |
| <code>leaseCollectionName</code> | <code>LeaseCollectionName</code> | (Optional) The name of the collection used to store leases. When not set, the value <code>leases</code> is used. |
| <code>createLeaseCollectionIfNotExists</code> | <code>CreateLeaseCollectionIfNotExists</code> | (Optional) When set to <code>true</code> , the leases collection is automatically created when it doesn't already exist. The default value is <code>false</code> . |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-----------------------------------|-----------------------------------|--|
| 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 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. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|---------------------------|--|
| 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, app settings go into the [local.settings.json file](#).

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 a 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#](#)
- [C# Script](#)
- [JavaScript](#)

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:

```

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.

```

namespace CosmosDBSamplesV1
{
    public class ToDoItemLookup
    {
        public string ToDoItemId { get; set; }
    }
}

```

```

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 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.

```

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.

```

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.

```
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.

```

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.

```

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

- [C#](#)
- [C# Script](#)
- [JavaScript](#)

In [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> | n/a | Must be set to <code>documentdb</code> . |
| <code>direction</code> | n/a | Must be set to <code>in</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 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, app settings go into the [local.settings.json file](#).

Input - usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)

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#](#)
- [C# Script](#)
- [JavaScript](#)

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:

```
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.

```
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.

```

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 ToDoItem in ToDoItemsIn)
            {
                log.Info($"Description={ToDoItem.Description}");
                await ToDoItemsOut.AddAsync(ToDoItem);
            }
        }
    }
}

```

Output - attributes

- [C#](#)
- [C# Script](#)
- [JavaScript](#)

In [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:

```

[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 |
|-----------------------------|--------------------------------|--|
| type | n/a | Must be set to <code>documentdb</code> . |
| direction | n/a | Must be set to <code>out</code> . |
| name | n/a | Name of the binding parameter that represents the document in the function. |
| databaseName | DatabaseName | The database containing the collection where the document is created. |
| collectionName | CollectionName | The name of the collection where the document is created. |
| createIfNotExists | CreateIfNotExists | A boolean value to indicate whether the collection is created when it doesn't exist. The default is <code>false</code> because new collections are created with reserved throughput, which has cost implications. For more information, see the pricing page . |
| partitionKey | PartitionKey | When <code>CreateIfNotExists</code> is true, defines the partition key path for the created collection. |
| collectionThroughput | CollectionThroughput | When <code>CreateIfNotExists</code> is true, defines the throughput of the created collection. |
| connection | ConnectionStringSetting | The name of the app setting containing your Azure Cosmos DB connection string. |

When you're developing locally, app settings go into the [local.settings.json file](#).

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 |
|----------|--------------------------------------|
| CosmosDB | CosmosDB Error Codes |

Next steps

- [Learn more about serverless database computing with Cosmos DB](#)
- [Learn more about Azure functions triggers and bindings](#)

Azure Cosmos DB trigger and bindings for Azure Functions 2.x and higher overview

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 Functions version 2.x and higher, the trigger, bindings, and package are all named Cosmos DB.

Supported APIs

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's API for MongoDB](#), [Cassandra API](#), [Gremlin API](#), and [Table API](#).

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference to the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

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

12/4/2020 • 6 minutes to read • [Edit Online](#)

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.

For information on setup and configuration details, see the [overview](#).

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that is invoked when there are inserts or updates in the specified database and collection.

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class CosmosTrigger
    {
        [FunctionName("CosmosTrigger")]
        public static void Run([CosmosDBTrigger(
            databaseName: "ToDoItems",
            collectionName: "Items",
            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}");
            }
        }
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In 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:

```
[FunctionName("DocumentUpdates")]
public static void Run([CosmosDBTrigger("database", "collection", ConnectionStringSetting =
"myCosmosDB")]
    IReadOnlyList<Document> documents,
    ILogger log)
{
    ...
}
```

For a complete example, see [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 |
|-------------------------------------|-------------------------------------|---|
| type | n/a | Must be set to <code>cosmosDBTrigger</code> . |
| direction | n/a | Must be set to <code>in</code> . This parameter is set automatically when you 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 Azure Cosmos DB account that 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. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|----------------------------------|----------------------------------|--|
| 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 created when it doesn't already exist. The default value is <code>false</code> . |
| leasesCollectionThroughput | LeasesCollectionThroughput | (Optional) Defines the number 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, the value is added as a prefix to the leases created in the Lease collection for this Function. Using a prefix allows two separate Azure Functions to share the same Lease collection by using different prefixes. |
| feedPollDelay | 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 | 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). |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|-----------------------|---|
| leaseRenewInterval | 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). |
| 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, this property sets the maximum number of items received per Function call. If operations in the monitored collection 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 | StartFromBeginning | (Optional) This option tells the Trigger to read changes from the beginning of the collection'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. |
| preferredLocations | 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". |

When you're developing locally, app settings go into the [local.settings.json file](#).

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 a 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.

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

12/4/2020 • 24 minutes to read • [Edit Online](#)

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

If the collection is [partitioned](#), lookup operations need to also specify the partition key value.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

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 SqlCommand](#)
- [HTTP trigger, get multiple docs, using SqlCommand](#)
- [HTTP trigger, get multiple docs, using DocumentClient](#)

The examples refer to a simple `ToDoItem` type:

```
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.

```

namespace CosmosDBSamplesV2
{
    public class ToDoItemLookup
    {
        public string ToDoItemId { get; set; }

        public string ToDoItemPartitionKeyValue { get; set; }
    }
}

```

```

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.

```

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.

```

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.

```

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 toItem in ToDoItems)
            {
                log.LogInformation(toItem.Description);
            }
            return new OkResult();
        }
    }
}

```

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.

```

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 toItem in ToDoItems)
            {
                log.LogInformation(toItem.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.

```

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();
        }
    }
}

```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the [CosmosDB](#) 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](#).

Configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the `CosmosDB` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|--------------------------------------|--------------------------------------|--|
| <code>type</code> | n/a | Must be set to <code>cosmosDB</code> . |
| <code>direction</code> | n/a | Must be set to <code>in</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 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>connectionStringSetting</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. It is required for lookups in partitioned collections. |
| <code>preferredLocations</code> | <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". |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

When the function exits successfully, any changes made to the input document via named input parameters are automatically persisted.

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

12/4/2020 • 10 minutes to read • [Edit Online](#)

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](#).

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

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:

```
namespace CosmosDBSamplesV2
{
    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.

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using System;

namespace CosmosDBSamplesV2
{
    public static class WriteOneDoc
    {
        [FunctionName("WriteOneDoc")]
        public static void Run(
            [QueueTrigger("todoqueueforwrite")] string queueMessage,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")]
            out dynamic document,
            ILogger log)
        {
            document = new { Description = queueMessage, id = Guid.NewGuid() };

            log.LogInformation($"C# Queue trigger function inserted one row");
            log.LogInformation($"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.

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class WriteDocsIAsyncCollector
    {
        [FunctionName("WriteDocsIAsyncCollector")]
        public static async Task Run(
            [QueueTrigger("todoqueueforwritemulti")] ToDoItem[] ToDoItemsIn,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")]
                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);
            }
        }
    }
}

```

Attributes and annotations

- [C#](#)

- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `CosmosDB` 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 `CosmosDB` attribute example in a method signature:

```
[FunctionName("QueueToDocDB")]
public static void Run(
    [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] string myQueueItem,
    [CosmosDB("ToDoList", "Items", Id = "id", ConnectionStringSetting = "myCosmosDB")] out dynamic
    document)
{
    ...
}
```

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `CosmosDB` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|--------------------------------|--------------------------------|--|
| <code>type</code> | n/a | Must be set to <code>cosmosDB</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 <code>false</code> 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, it defines the partition key path for the created collection. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|---------------------------|---|
| collectionThroughput | CollectionThroughput | When <code>CreateIfNotExists</code> is true, it defines the throughput of the created collection. |
| connectionStringSetting | ConnectionStringSetting | The name of the app setting containing your Azure Cosmos DB connection string. |
| preferredLocations | 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". |
| useMultipleWriteLocations | UseMultipleWriteLocations | (Optional) When set to <code>true</code> along with <code>PreferredLocations</code> , it can leverage multi-region writes in the Azure Cosmos DB service. |

When you're developing locally, app settings go into the [local.settings.json file](#).

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 |
|----------|--------------------------------------|
| CosmosDB | CosmosDB Error Codes |

host.json settings

This section describes the global configuration settings available for this binding in version 2.x. For more information about global configuration settings in version 2.x, see [host.json reference for Azure Functions version 2.x](#).

```
{
  "version": "2.0",
  "extensions": {
    "cosmosDB": {
      "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 Direct and Gateway |
| 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. |

Next steps

- Run a function when an Azure Cosmos DB document is created or modified (Trigger)
- Read an Azure Cosmos DB document (Input binding)

Azure Event Grid bindings for Azure Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

This reference explains how to handle [Event Grid](#) events in Azure Functions. For details on how to handle Event Grid messages in an HTTP end point, see [Receive events to an HTTP endpoint](#).

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](#). In this reference, you learn to use an Event Grid trigger to invoke a function when an event is received from Event Grid, and to use the output binding to send events to an [Event Grid custom topic](#).

If you prefer, you can use an HTTP trigger to handle Event Grid Events; see [Receive events to an HTTP endpoint](#). Currently, you can't use an Event Grid trigger for an Azure Functions app when the event is delivered in the [CloudEvents schema](#). Instead, use an HTTP trigger.

| ACTION | TYPE |
|---|--------------------------------|
| Run a function when an Event Grid event is dispatched | Trigger |
| Sends an Event Grid event | Output binding |

The code in this reference 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](#).

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 2.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

Next steps

- [Run a function when an Event Grid event is dispatched](#)
- [Dispatch an Event Grid event](#)

Azure Event Grid trigger for Azure Functions

12/4/2020 • 11 minutes to read • [Edit Online](#)

Use the function trigger to respond to an event sent to an Event Grid topic.

For information on setup and configuration details, see the [overview](#).

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

For an HTTP trigger example, see [Receive events to an HTTP endpoint](#).

C# (2.x and higher)

The following example shows a C# function that binds to `EventGridEvent`:

```
using Microsoft.Azure.EventGrid.Models;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.EventGrid;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace Company.Function
{
    public static class EventGridTriggerCSharp
    {
        [FunctionName("EventGridTest")]
        public static void EventGridTest([EventGridTrigger]EventGridEvent eventGridEvent, ILogger log)
        {
            log.LogInformation(eventGridEvent.Data.ToString());
        }
    }
}
```

For more information, see [Packages](#), [Attributes](#), [Configuration](#), and [Usage](#).

Version 1.x

The following example shows a Functions 1.x C# function that binds to `Object`:

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.EventGrid;
using Microsoft.Azure.WebJobs.Host;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using Microsoft.Extensions.Logging;

namespace Company.Function
{
    public static class EventGridTriggerCSharp
    {
        [FunctionName("EventGridTriggerCSharp")]
        public static void Run([EventGridTrigger] JObject eventGridEvent, ILogger log)
        {
            log.LogInformation(eventGridEvent.ToString(Formatting.Indented));
        }
    }
}

```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `EventGridTrigger` attribute.

Here's an `EventGridTrigger` attribute in a method signature:

```

[FunctionName("EventGridTest")]
public static void EventGridTest([EventGridTrigger] JObject eventGridEvent, ILogger log)
{
    ...
}

```

For a complete example, see [C# example](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file. There are no constructor parameters or properties to set in the `EventGridTrigger` attribute.

| FUNCTION.JSON PROPERTY | DESCRIPTION |
|------------------------|--|
| <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. |

Usage

- [C#](#)

- C# Script
- JavaScript
- Python
- Java

In Azure Functions 1.x, you can use the following parameter types for the Event Grid trigger:

- `JObject`
- `string`

In Azure Functions 2.x and higher, you also have the option to use the following parameter type for the Event Grid trigger:

- `Microsoft.Azure.EventGrid.Models.EventGridEvent` - Defines properties for the fields common to all event types.

NOTE

In Functions v1 if you try to bind to `Microsoft.Azure.WebJobs.Extensions.EventGrid.EventGridEvent`, the compiler will display a "deprecated" message and advise you to use `Microsoft.Azure.EventGrid.Models.EventGridEvent` instead. To use the newer type, reference the [Microsoft.Azure.EventGrid](#) NuGet package and fully qualify the `EventGridEvent` type name by prefixing it with `Microsoft.Azure.EventGrid.Models`.

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:

```
[{
  "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": "0x8D562831044DD0",
    "contentType": "application/octet-stream",
    "contentLength": 2248,
    "blobType": "BlockBlob",
    "url": "https://egblobstore.blob.core.windows.net/{containername}/blobname.jpg",
    "sequencer": "00000000000272D00000000003D60F",
    "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.

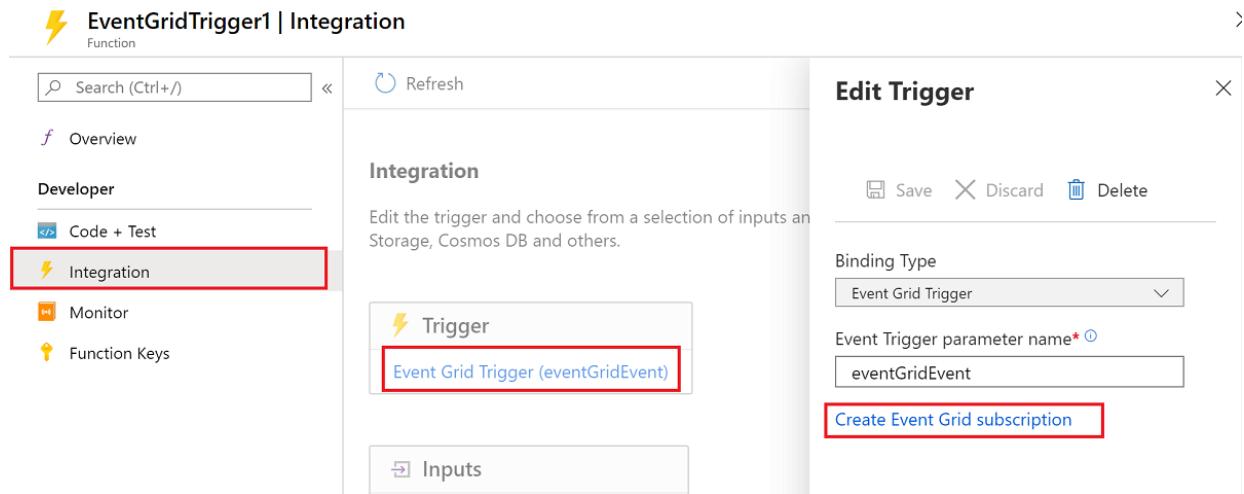
The `EventGridEvent` type defines only the top-level properties; the `Data` property is a `JObject`.

Create a subscription

To start receiving Event Grid HTTP requests, create an Event Grid subscription that specifies the endpoint URL that invokes the function.

Azure 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**.



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

Event Grid Schema

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Types

Event Hubs Namespaces

Subscription

Resource Group

Resource

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Filter to Event Types

Capture File Created

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type

Azure Function (change)

Endpoint

EventGridTrigger1 (change)

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.

Azure CLI

To create a subscription by using [the Azure CLI](#), use the `az eventgrid event-subscription create` command.

The command requires the endpoint URL that invokes the function. The following example shows the version-specific URL pattern:

Version 2.x (and higher) runtime

```
https://{{functionappname}}.azurewebsites.net/runtime/webhooks/eventgrid?functionName={{functionname}}&code={{systemkey}}
```

Version 1.x runtime

```
https://{{functionappname}}.azurewebsites.net/admin/extensions/EventGridExtensionConfig?functionName={{functionname}}&code={{systemkey}}
```

The system key is an authorization key that has to be included in the endpoint URL for an Event Grid trigger. The following section explains how to get the system key.

Here's an example that subscribes to a blob storage account (with a placeholder for the system key):

Version 2.x (and higher) runtime

- [Bash](#)
- [Cmd](#)

```
az eventgrid resource event-subscription create -g myResourceGroup \
--provider-namespace Microsoft.Storage --resource-type storageAccounts \
--resource-name myblobstorage12345 --name myFuncSub \
--included-event-types Microsoft.Storage.BlobCreated \
--subject-begins-with /blobServices/default/containers/images/blobs/ \
--endpoint https://mystoragetriggeredfunction.azurewebsites.net/runtime/webhooks/eventgrid?
functionName=imageresizefunc&code=<key>
```

Version 1.x runtime

- [Bash](#)
- [Cmd](#)

```
az eventgrid resource event-subscription create -g myResourceGroup \
--provider-namespace Microsoft.Storage --resource-type storageAccounts \
--resource-name myblobstorage12345 --name myFuncSub \
--included-event-types Microsoft.Storage.BlobCreated \
--subject-begins-with /blobServices/default/containers/images/blobs/ \
--endpoint
https://mystoragetriggeredfunction.azurewebsites.net/admin/extensions/EventGridExtensionConfig?
functionName=imageresizefunc&code=<key>
```

For more information about how to create a subscription, see [the blob storage quickstart](#) or the other Event Grid quickstarts.

Get the system key

You can get the system key by using the following API (HTTP GET):

Version 2.x (and higher) runtime

```
http://{functionappname}.azurewebsites.net/admin/host/systemkeys/eventgrid_extension?code={masterkey}
```

Version 1.x runtime

```
http://{functionappname}.azurewebsites.net/admin/host/systemkeys/eventgridextensionconfig_extension?code=
{masterkey}
```

This is an admin 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}"  
    }  
  ]  
}
```

You can get the master key for your function app from the **Function app settings** tab in the portal.

IMPORTANT

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 [Authorization keys](#) in the HTTP trigger reference article.

Alternatively, you can send an HTTP PUT to specify the key value yourself.

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.

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

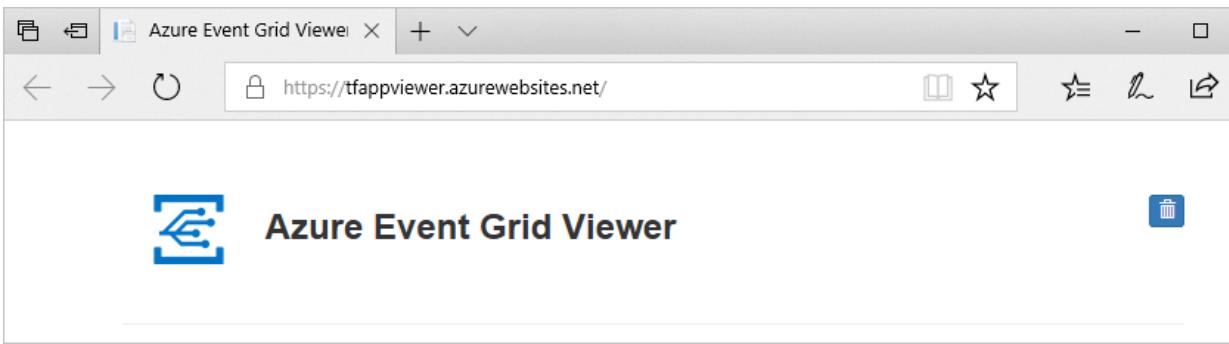
To simplify capturing event messages, you can deploy a [pre-built web app](#) that displays the event messages. The deployed solution includes an App Service plan, an App Service web app, and source code from GitHub.

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.

Use a tool such as [Postman](#) or [curl](#) 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.
- Post to the URL of your Event Grid trigger function.
 - For 2.x and higher use the following pattern:

```
http://localhost:7071/runtime/webhooks/eventgrid?functionName={FUNCTION_NAME}
```

- For 1.x use:

```
http://localhost:7071/admin/extensions/EventGridExtensionConfig?functionName={FUNCTION_NAME}
```

The `functionName` parameter must be the name specified in the `FunctionName` attribute.

The following screenshots show the headers and request body in Postman:

POST http://localhost:7071/admin/extensions/EventGridExtensionConfig?functionNa...

| Key | Value |
|--|------------------|
| <input checked="" type="checkbox"/> Content-Type | application/json |
| <input checked="" type="checkbox"/> aeg-event-type | Notification |

POST http://localhost:7071/admin/extensions/EventGridExtensionConfig?functionNa...

| | | | | |
|---------------|-----------------------|--------|--------------------|---------------------------|
| Authorization | Headers (2) | Body ● | Pre-request Script | Tests |
| form-data | x-www-form-urlencoded | raw | binary | JSON (application/json) ▾ |

```
[{"topic": "/subscriptions/{subscriptionid}/resourceGroups/eg0122/providers/Microsoft BlobStorage/egblobstor0122", "subject": "/blobServices/default/containers/test0123/blobs/Default.rdp", "eventType": "Microsoft.Storage.BlobCreated", "eventTime": "2018-01-23T17:02:19.6069787Z", "id": "44d4f022-001e-003c-466b-940cba0612e5", "data": { "api": "PutBlockList", "clientRequestId": "2c169f2f-7b3b-4d99-839b-c92a2d25801b", "requestId": "44d4f022-001e-003c-466b-940cba000000", "eTag": "0x8D562831044DD0", "contentType": "application/octet-stream", "contentLength": 2248, "blobType": "BlockBlob", "url": "https://egblobstor0122.blob.core.windows.net/test0123/Default.rdp", "sequencer": "00000000000272D000000000003D60F", "storageDiagnostics": { "batchId": "b4229b3a-4d50-4ff4-a9f2-039ccf26efe9" }, "dataVersion": "", "metadataVersion": "1" }}
```

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 2018-01-26T12:1
6:46.0126069-08:00', 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": "000000000000272D0000000000003D60F",
[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, Duratio
```

Next steps

- Dispatch an Event Grid event

Azure Event Grid output binding for Azure Functions

11/2/2020 • 4 minutes to read • [Edit Online](#)

Use the Event Grid output binding to write events to a custom topic. You must have a valid [access key for the custom topic](#).

For information on setup and configuration details, see the [overview](#).

NOTE

The Event Grid output binding does not support shared access signatures (SAS tokens). You must use the topic's access key.

IMPORTANT

The Event Grid output binding is only available for Functions 2.x and higher.

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that writes a message to an Event Grid custom topic, using the `method return value` as the output:

```
[FunctionName("EventGridOutput")]
[return: EventGrid(TopicEndpointUri = "MyEventGridTopicUriSetting", TopicKeySetting =
"MyEventGridTopicKeySetting")]
public static EventGridEvent Run([TimerTrigger("0 */5 * * * *")] TimerInfo myTimer, ILogger log)
{
    return new EventGridEvent("message-id", "subject-name", "event-data", "event-type", DateTime.UtcNow,
    "1.0");
}
```

The following example shows how to use the `IAsyncCollector` interface to send a batch of messages.

```
[FunctionName("EventGridAsyncOutput")]
public static async Task Run(
    [TimerTrigger("0 */5 * * *")] TimerInfo myTimer,
    [EventGrid(TopicEndpointUri = "MyEventGridTopicUriSetting", TopicKeySetting =
"MyEventGridTopicKeySetting")]IAsyncCollector<EventGridEvent> outputEvents,
    ILogger log)
{
    for (var i = 0; i < 3; i++)
    {
        var myEvent = new EventGridEvent("message-id-" + i, "subject-name", "event-data", "event-type",
DateTime.UtcNow, "1.0");
        await outputEvents.AddAsync(myEvent);
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

For [C# class libraries](#), use the [EventGridAttribute](#) attribute.

The attribute's constructor takes the name of an app setting that contains the name of the custom topic, and the name of an app setting that contains the topic key. For more information about these settings, see [Output - configuration](#). Here's an `EventGrid` attribute example:

```
[FunctionName("EventGridOutput")]
[return: EventGrid(TopicEndpointUri = "MyEventGridTopicUriSetting", TopicKeySetting =
"MyEventGridTopicKeySetting")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, ILogger log)
{
    ...
}
```

For a complete example, see [example](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `EventGrid` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| <code>type</code> | n/a | Must be set to "eventGrid". |
| <code>direction</code> | n/a | Must be set to "out". This parameter is set automatically when you create the binding in the Azure portal. |
| <code>name</code> | n/a | The variable name used in function code that represents the event. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------|-------------------------|--|
| topicEndpointUri | TopicEndpointUri | The name of an app setting that contains the URI for the custom topic, such as <code>MyTopicEndpointUri</code> . |
| topicKeySetting | TopicKeySetting | The name of an app setting that contains an access key for the custom topic. |

When you're developing locally, app settings go into the [local.settings.json](#) file.

IMPORTANT

Ensure that you set the value of the `TopicEndpointUri` configuration property to the name of an app setting that contains the URI of the custom topic. Do not specify the URL of the custom topic directly in this property.

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Send messages by using a method parameter such as `out EventGridEvent paramName`. To write multiple messages, you can use `ICollector<EventGridEvent>` or `IAsyncCollector<EventGridEvent>` in place of `out EventGridEvent`.

Next steps

- [Dispatch an Event Grid event](#)

Azure Event Hubs trigger and bindings for Azure Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 |

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

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

12/4/2020 • 10 minutes to read • [Edit Online](#)

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.

Scaling

Each instance of an event triggered function is backed by a single [EventProcessorHost](#) instance. The trigger (powered by Event Hubs) ensures that only one [EventProcessorHost](#) instance can get a lease on a given partition.

For example, consider an Event Hub as follows:

- 10 partitions
- 1,000 events distributed evenly across all partitions, with 100 messages in each partition

When your function is first enabled, there is only one instance of the function. Let's call the first function instance `Function_0`. The `Function_0` function has a single instance of [EventProcessorHost](#) that holds a lease on all ten partitions. This instance is reading events from partitions 0-9. From this point forward, one of the following happens:

- **New function instances are not needed:** `Function_0` is able to process all 1,000 events before the Functions scaling logic take effect. In this case, all 1,000 messages are processed by `Function_0`.
- **An additional function instance is added:** If the Functions scaling logic determines that `Function_0` has more messages than it can process, a new function app instance (`Function_1`) is created. This new function also has an associated instance of [EventProcessorHost](#). As the underlying Event Hubs detect that a new host instance is trying read messages, it load balances the partitions across the host instances. For example, partitions 0-4 may be assigned to `Function_0` and partitions 5-9 to `Function_1`.
- **N more function instances are added:** If the Functions scaling logic determines that both `Function_0` and `Function_1` have more messages than they can process, new `Functions_N` function app instances are created. Apps are created to the point where `N` is greater than the number of event hub partitions. In our example, Event Hubs again load balances the partitions, in this case across the instances `Function_0 ... Functions_9`.

As scaling occurs, `N` instances is a number greater than the number of event hub partitions. This pattern is used to ensure [EventProcessorHost](#) instances are available to obtain locks on partitions as they become available from other instances. You are only charged for the resources used when the function instance executes. In other words, you are not charged for this over-provisioning.

When all function execution completes (with or without errors), checkpoints are added to the associated storage account. When check-pointing succeeds, all 1,000 messages are never retrieved again.

- [C#](#)
- [C# Script](#)

- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that logs the message body of the event hub trigger.

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")]
string myEventHubMessage, ILogger log)
{
    log.LogInformation($"C# function triggered to process a message: {myEventHubMessage}");
}
```

To get access to [event metadata](#) in function code, bind to an [EventData](#) object (requires a using statement for `Microsoft.Azure.EventHubs`). 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:

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run(
    [EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")] EventData
    myEventHubMessage,
    DateTime enqueuedTimeUtc,
    Int64 sequenceNumber,
    string offset,
    ILogger log)
{
    log.LogInformation($"Event: {Encoding.UTF8.GetString(myEventHubMessage.Body)}");
    // Metadata accessed by binding to EventData
    log.LogInformation($"EnqueuedTimeUtc={myEventHubMessage.SystemProperties.EnqueuedTimeUtc}");
    log.LogInformation($"SequenceNumber={myEventHubMessage.SystemProperties.SequenceNumber}");
    log.LogInformation($"Offset={myEventHubMessage.SystemProperties.Offset}");
    // Metadata accessed by using binding expressions in method parameters
    log.LogInformation($"EnqueuedTimeUtc={enqueuedTimeUtc}");
    log.LogInformation($"SequenceNumber={sequenceNumber}");
    log.LogInformation($"Offset={offset}");
}
```

To receive events in a batch, make `string` or `EventData` an array.

NOTE

When receiving in a batch you cannot bind to method parameters like in the above example with `DateTime enqueuedTimeUtc` and must receive these from each `EventData` object

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")]
EventData[] eventHubMessages, ILogger log)
{
    foreach (var message in eventHubMessages)
    {
        log.LogInformation($"C# function triggered to process a message:
{Encoding.UTF8.GetString(message.Body)}");
        log.LogInformation($"EnqueuedTimeUtc={message.SystemProperties.EnqueuedTimeUtc}");
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `EventHubTriggerAttribute` attribute.

The attribute's constructor takes the name of the event hub, the name of the consumer group, and the name of an app setting that contains the connection string. For more information about these settings, see the [trigger configuration section](#). Here's an `EventHubTriggerAttribute` attribute example:

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")]
string myEventHubMessage, ILogger log)
{
    ...
}
```

For a complete example, see [Trigger - C# example](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `EventHubTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|---------------------------|--|
| <code>type</code> | n/a | Must be set to <code>eventHubTrigger</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>direction</code> | n/a | Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>name</code> | n/a | The name of the variable that represents the event item in function code. |
| <code>path</code> | <code>EventHubName</code> | Functions 1.x only. 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. |
| <code>eventHubName</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 app settings <code>%eventHubName%</code> |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| consumerGroup | ConsumerGroup | An optional property that sets the consumer group used to subscribe to events in the hub. If omitted, the <code>\$Default</code> consumer group is used. |
| cardinality | n/a | Used for all non-C# languages. Set to <code>many</code> in order to enable batching. If omitted or set to <code>one</code> , a single message is passed to the function. In C#, this property is automatically assigned whenever the trigger has an array for the type. |
| connection | Connection | The name of an app setting that contains the connection string to the event hub's namespace. Copy this connection string by clicking the Connection Information button for the namespace , not the event hub itself. This connection string must have at least read permissions to activate the trigger. |

When you're developing locally, app settings go into the [local.settings.json file](#).

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> | <code>DateTime</code> | 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.

host.json properties

The `host.json` file contains settings that control Event Hubs trigger behavior. The configuration is different depending on the Azure Functions version.

Functions 2.x and higher

```
{
  "version": "2.0",
  "extensions": {
    "eventHubs": {
      "batchCheckpointFrequency": 5,
      "eventProcessorOptions": {
        "maxBatchSize": 256,
        "prefetchCount": 512
      }
    }
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------------|---------|---|
| maxBatchSize | 10 | The maximum event count received per receive loop. |
| prefetchCount | 300 | The default pre-fetch count used by the underlying <code>EventProcessorHost</code> . The minimum allowed value is 10. |
| batchCheckpointFrequency | 1 | The number of event batches to process before creating an EventHub cursor checkpoint. |

NOTE

For a reference of `host.json` in Azure Functions 2.x and beyond, see [host.json reference for Azure Functions](#).

Functions 1.x

```
{
  "eventHub": {
    "maxBatchSize": 64,
    "prefetchCount": 256,
    "batchCheckpointFrequency": 1
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------|---------|--|
| maxBatchSize | 64 | The maximum event count received per receive loop. |

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------------|---------|---|
| prefetchCount | n/a | The default pre-fetch that will be used by the underlying <code>EventProcessorHost</code> . |
| batchCheckpointFrequency | 1 | The number of event batches to process before creating an EventHub cursor checkpoint. |

NOTE

For a reference of host.json in Azure Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

Next steps

- [Write events to an event stream \(Output binding\)](#)

Azure Event Hubs output binding for Azure Functions

12/4/2020 • 6 minutes to read • [Edit Online](#)

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.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that writes a message to an event hub, using the method return value as the output:

```
[FunctionName("EventHubOutput")]
[return: EventHub("outputEventHubMessage", Connection = "EventHubConnectionAppSetting")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, ILogger log)
{
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
    return $"{DateTime.Now}";
}
```

The following example shows how to use the `IAsyncCollector` interface to send a batch of messages. This scenario is common when you are processing messages coming from one Event Hub and sending the result to another Event Hub.

```
[FunctionName("EH2EH")]
public static async Task Run(
    [EventHubTrigger("source", Connection = "EventHubConnectionAppSetting")] EventData[] events,
    [EventHub("dest", Connection = "EventHubConnectionAppSetting")] IAsyncCollector<string> outputEvents,
    ILogger log)
{
    foreach (EventData eventData in events)
    {
        // do some processing
        var myProcessedEvent = DoSomething(eventData);

        // then send the message
        await outputEvents.AddAsync(JsonConvert.SerializeObject(myProcessedEvent));
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

For [C# class libraries](#), use the [EventHubAttribute](#) attribute.

The attribute's constructor takes the name of the event hub and the name of an app setting that contains the connection string. For more information about these settings, see [Output - configuration](#). Here's an [EventHub](#) attribute example:

```
[FunctionName("EventHubOutput")]
[return: EventHub("outputEventHubMessage", Connection = "EventHubConnectionAppSetting")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, ILogger log)
{
    ...
}
```

For a complete example, see [Output - C# example](#).

Configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the [EventHub](#) attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|---------------------|---|
| type | n/a | Must be set to "eventHub". |
| direction | n/a | Must be set to "out". This parameter is set automatically when you create the binding in the Azure portal. |
| name | n/a | The variable name used in function code that represents the event. |
| path | EventHubName | Functions 1.x only. 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. |
| eventHubName | 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. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| connection | Connection | The name of an app setting that contains the connection string to the event hub's namespace. Copy this connection string by clicking the Connection Information button for the <i>namespace</i> , not the event hub itself. This connection string must have send permissions to send the message to the event stream. |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Send messages by using a method parameter such as `out string paramName`. In C# script, `paramName` is the value specified in the `name` property of *function.json*. To write multiple messages, you can use `ICollector<string>` or `IAsyncCollector<string>` in place of `out string`.

Exceptions and return codes

| BINDING | REFERENCE |
|-----------|----------------------------------|
| Event Hub | Operations Guide |

Next steps

- [Respond to events sent to an event hub event stream \(Trigger\)](#)

Azure IoT Hub bindings for Azure Functions

2/24/2020 • 2 minutes to read • [Edit Online](#)

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 |
| Write events to an IoT event stream | Output binding |

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

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

11/2/2020 • 10 minutes to read • [Edit Online](#)

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.

Scaling

Each instance of an event triggered function is backed by a single [EventProcessorHost](#) instance. The trigger (powered by Event Hubs) ensures that only one [EventProcessorHost](#) instance can get a lease on a given partition.

For example, consider an Event Hub as follows:

- 10 partitions
- 1,000 events distributed evenly across all partitions, with 100 messages in each partition

When your function is first enabled, there is only one instance of the function. Let's call the first function instance `Function_0`. The `Function_0` function has a single instance of [EventProcessorHost](#) that holds a lease on all ten partitions. This instance is reading events from partitions 0-9. From this point forward, one of the following happens:

- **New function instances are not needed:** `Function_0` is able to process all 1,000 events before the Functions scaling logic take effect. In this case, all 1,000 messages are processed by `Function_0`.
- **An additional function instance is added:** If the Functions scaling logic determines that `Function_0` has more messages than it can process, a new function app instance (`Function_1`) is created. This new function also has an associated instance of [EventProcessorHost](#). As the underlying Event Hubs detect that a new host instance is trying read messages, it load balances the partitions across the host instances. For example, partitions 0-4 may be assigned to `Function_0` and partitions 5-9 to `Function_1`.
- **N more function instances are added:** If the Functions scaling logic determines that both `Function_0` and `Function_1` have more messages than they can process, new `Functions_N` function app instances are created. Apps are created to the point where `N` is greater than the number of event hub partitions. In our example, Event Hubs again load balances the partitions, in this case across the instances `Function_0` ... `Functions_9`.

As scaling occurs, `N` instances is a number greater than the number of event hub partitions. This pattern is used to ensure [EventProcessorHost](#) instances are available to obtain locks on partitions as they become available from other instances. You are only charged for the resources used when the function instance executes. In other words, you are not charged for this over-provisioning.

When all function execution completes (with or without errors), checkpoints are added to the associated storage account. When check-pointing succeeds, all 1,000 messages are never retrieved again.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that logs the message body of the event hub trigger.

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")]
string myEventHubMessage, ILogger log)
{
    log.LogInformation($"C# function triggered to process a message: {myEventHubMessage}");
}
```

To get access to [event metadata](#) in function code, bind to an [EventData](#) object (requires a using statement for `Microsoft.Azure.EventHubs`). 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:

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run(
    [EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")] EventData
    myEventHubMessage,
    DateTime enqueuedTimeUtc,
    Int64 sequenceNumber,
    string offset,
    ILogger log)
{
    log.LogInformation($"Event: {Encoding.UTF8.GetString(myEventHubMessage.Body)}");
    // Metadata accessed by binding to EventData
    log.LogInformation($"EnqueuedTimeUtc={myEventHubMessage.SystemProperties.EnqueuedTimeUtc}");
    log.LogInformation($"SequenceNumber={myEventHubMessage.SystemProperties.SequenceNumber}");
    log.LogInformation($"Offset={myEventHubMessage.SystemProperties.Offset}");
    // Metadata accessed by using binding expressions in method parameters
    log.LogInformation($"EnqueuedTimeUtc={enqueuedTimeUtc}");
    log.LogInformation($"SequenceNumber={sequenceNumber}");
    log.LogInformation($"Offset={offset}");
}
```

To receive events in a batch, make `string` or `EventData` an array.

NOTE

When receiving in a batch you cannot bind to method parameters like in the above example with `DateTime enqueuedTimeUtc` and must receive these from each `EventData` object

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")]
EventData[] eventHubMessages, ILogger log)
{
    foreach (var message in eventHubMessages)
    {
        log.LogInformation($"C# function triggered to process a message:
{Encoding.UTF8.GetString(message.Body)}");
        log.LogInformation($"EnqueuedTimeUtc={message.SystemProperties.EnqueuedTimeUtc}");
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `EventHubTriggerAttribute` attribute.

The attribute's constructor takes the name of the event hub, the name of the consumer group, and the name of an app setting that contains the connection string. For more information about these settings, see the [trigger configuration section](#). Here's an `EventHubTriggerAttribute` attribute example:

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnectionAppSetting")]
string myEventHubMessage, ILogger log)
{
    ...
}
```

For a complete example, see [Trigger - C# example](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `EventHubTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| <code>type</code> | n/a | Must be set to <code>eventHubTrigger</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>direction</code> | n/a | Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>name</code> | n/a | The name of the variable that represents the event item in function code. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| path | EventHubName | Functions 1.x only. 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. |
| eventHubName | 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. Can be referenced via app settings <code>%eventHubName%</code> |
| consumerGroup | ConsumerGroup | An optional property that sets the consumer group used to subscribe to events in the hub. If omitted, the <code>\$Default</code> consumer group is used. |
| cardinality | n/a | Used for all non-C# languages. Set to <code>many</code> in order to enable batching. If omitted or set to <code>one</code> , a single message is passed to the function. In C#, this property is automatically assigned whenever the trigger has an array for the type. |
| connection | Connection | The name of an app setting that contains the connection string to the event hub's namespace. Copy this connection string by clicking the Connection Information button for the namespace , not the event hub itself. This connection string must have at least read permissions to activate the trigger. |

When you're developing locally, app settings go into the [local.settings.json file](#).

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> | <code>DateTime</code> | The enqueued time in UTC. |

| PROPERTY | TYPE | DESCRIPTION |
|------------------|-----------------------------|--|
| 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.

host.json properties

The [host.json](#) file contains settings that control Event Hubs trigger behavior. The configuration is different depending on the Azure Functions version.

Functions 2.x and higher

```
{
  "version": "2.0",
  "extensions": {
    "eventHubs": {
      "batchCheckpointFrequency": 5,
      "eventProcessorOptions": {
        "maxBatchSize": 256,
        "prefetchCount": 512
      }
    }
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------------|---------|---|
| maxBatchSize | 10 | The maximum event count received per receive loop. |
| prefetchCount | 300 | The default pre-fetch count used by the underlying <code>EventProcessorHost</code> . The minimum allowed value is 10. |
| batchCheckpointFrequency | 1 | The number of event batches to process before creating an <code>EventHub</code> cursor checkpoint. |

NOTE

For a reference of host.json in Azure Functions 2.x and beyond, see [host.json reference for Azure Functions](#).

Functions 1.x

```
{  
  "eventHub": {  
    "maxBatchSize": 64,  
    "prefetchCount": 256,  
    "batchCheckpointFrequency": 1  
  }  
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------------|---------|---|
| maxBatchSize | 64 | The maximum event count received per receive loop. |
| prefetchCount | n/a | The default pre-fetch that will be used by the underlying <code>EventProcessorHost</code> . |
| batchCheckpointFrequency | 1 | The number of event batches to process before creating an EventHub cursor checkpoint. |

NOTE

For a reference of host.json in Azure Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

Next steps

- [Write events to an event stream \(Output binding\)](#)

Azure IoT Hub output binding for Azure Functions

2/24/2020 • 6 minutes to read • [Edit Online](#)

This article explains how to work with Azure Functions output 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 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.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that writes a message to an event hub, using the method return value as the output:

```
[FunctionName("EventHubOutput")]
[return: EventHub("outputEventHubMessage", Connection = "EventHubConnectionAppSetting")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, ILogger log)
{
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
    return $"{DateTime.Now}";
}
```

The following example shows how to use the `IAsyncCollector` interface to send a batch of messages. This scenario is common when you are processing messages coming from one Event Hub and sending the result to another Event Hub.

```
[FunctionName("EH2EH")]
public static async Task Run(
    [EventHubTrigger("source", Connection = "EventHubConnectionAppSetting")] EventData[] events,
    [EventHub("dest", Connection = "EventHubConnectionAppSetting")] IAsyncCollector<string> outputEvents,
    ILogger log)
{
    foreach (EventData eventData in events)
    {
        // do some processing:
        var myProcessedEvent = DoSomething(eventData);

        // then send the message
        await outputEvents.AddAsync(JsonConvert.SerializeObject(myProcessedEvent));
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

For [C# class libraries](#), use the [EventHubAttribute](#) attribute.

The attribute's constructor takes the name of the event hub and the name of an app setting that contains the connection string. For more information about these settings, see [Output - configuration](#). Here's an [EventHub](#) attribute example:

```
[FunctionName("EventHubOutput")]
[return: EventHub("outputEventHubMessage", Connection = "EventHubConnectionAppSetting")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, ILogger log)
{
    ...
}
```

For a complete example, see [Output - C# example](#).

Configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the [EventHub](#) attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|---------------------|---|
| type | n/a | Must be set to "eventHub". |
| direction | n/a | Must be set to "out". This parameter is set automatically when you create the binding in the Azure portal. |
| name | n/a | The variable name used in function code that represents the event. |
| path | EventHubName | Functions 1.x only. 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. |
| eventHubName | 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. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| connection | Connection | The name of an app setting that contains the connection string to the event hub's namespace. Copy this connection string by clicking the Connection Information button for the <i>namespace</i> , not the event hub itself. This connection string must have send permissions to send the message to the event stream. |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Send messages by using a method parameter such as `out string paramName`. In C# script, `paramName` is the value specified in the `name` property of `function.json`. To write multiple messages, you can use `ICollector<string>` or `IAsyncCollector<string>` in place of `out string`.

Exceptions and return codes

| BINDING | REFERENCE |
|-----------|----------------------------------|
| Event Hub | Operations Guide |

Next steps

- [Respond to events sent to an event hub event stream \(Trigger\)](#)

Azure Functions HTTP triggers and bindings overview

11/2/2020 • 2 minutes to read • [Edit Online](#)

Azure Functions may be invoked via HTTP requests to build serverless APIs and respond to [webhooks](#).

| ACTION | TYPE |
|---|--------------------------------|
| Run a function from an HTTP request | Trigger |
| Return an HTTP response from a function | Output binding |

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](#).

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

Next steps

- [Run a function from an HTTP request](#)
- [Return an HTTP response from a function](#)

Azure Functions HTTP trigger

11/2/2020 • 20 minutes to read • [Edit Online](#)

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

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

The following example shows a [C# function](#) that looks for a `name` parameter either in the query string or the body of the HTTP request. Notice that the return value is used for the output binding, but a return value attribute isn't required.

```
[FunctionName("HttpTriggerCSharp")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "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;

    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}
```

Attributes and annotations

In C# class libraries and Java, the `HttpTrigger` attribute is available to configure the function.

You can set the authorization level and allowable HTTP methods in attribute constructor parameters, webhook type, and a route template. For more information about these settings, see [configuration](#).

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

This example demonstrates how to use the `HttpTrigger` attribute.

```
[FunctionName("HttpTriggerCSharp")]
public static Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequest req)
{
    ...
}
```

For a complete example, see the [trigger example](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `HttpTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|------------------------|---|
| <code>type</code> | n/a | Required - must be set to <code>httpTrigger</code> . |
| <code>direction</code> | n/a | Required - must be set to <code>in</code> . |
| <code>name</code> | n/a | Required - the variable name used in function code for the request or request body. |
| <code>authLevel</code> | <code>AuthLevel</code> | <p>Determines what keys, if any, need to be present on the request in order to invoke the function. The authorization level can be one of the following values:</p> <ul style="list-style-type: none">• <code>anonymous</code> —No API key is required.• <code>function</code> —A function-specific API key is required. This is the default value if none is provided.• <code>admin</code> —The master key is required. <p>For more information, see the section about authorization keys.</p> |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| methods | Methods | 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 . |
| route | Route | Defines the route template, controlling to which request URLs your function responds. The default value if none is provided is <functionname>. For more information, see customize the HTTP endpoint . |
| webHookType | WebHookType | <p><i>Supported only for the version 1.x runtime.</i></p> <p>Configures the HTTP trigger to act as a webhook receiver for the specified provider. Don't set the <code>methods</code> property if you set this property. The webhook type can be one of the following values:</p> <ul style="list-style-type: none"> • <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. Do not use the <code>authLevel</code> property with GitHub webhooks. For more information, see the GitHub webhooks section later in this article. • <code>slack</code> —The function responds to Slack webhooks. Do not use the <code>authLevel</code> property with Slack webhooks. For more information, see the Slack webhooks section later in this article. |

Payload

The trigger input type is declared as either `HttpRequest` or a custom type. If you choose `HttpRequest`, you get full access to the request object. For a custom type, the runtime tries to parse the JSON request body to set the object properties.

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://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>
```

You can customize this route using the optional `route` property on the HTTP trigger's input binding. As an example, the following `function.json` file defines a `route` property for an HTTP trigger:

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "name": "req",
      "direction": "in",
      "methods": [ "get" ],
      "route": "products/{category:alpha}/{id:int?}"
    },
    {
      "type": "http",
      "name": "res",
      "direction": "out"
    }
  ]
}
```

Using this configuration, the function is now addressable with the following route instead of the original route.

```
http://<APP_NAME>.azurewebsites.net/api/products/electronics/357
```

This configuration allows the function code to support two parameters in the address, *category* and *id*.

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

You can use any [Web API Route Constraint](#) with your parameters. The following C# function code makes use of both parameters.

```
using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;

public static IActionResult Run(HttpContext req, string category, int? id, ILogger log)
{
    var message = String.Format($"Category: {category}, ID: {id}");
    return (ActionResult)new OkObjectResult(message);
}
```

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.

```
{
  "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.

The following configuration shows how the `{id}` parameter is passed to the binding's `rowKey`.

```
{
  "type": "table",
  "direction": "in",
  "name": "product",
  "partitionKey": "products",
  "tableName": "products",
  "rowKey": "{id}"
}
```

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. This capability is only available to the Functions runtime in 2.x and higher. It is also currently only available for .NET languages.

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

Information regarding authenticated clients is available as a [ClaimsPrincipal](#). The ClaimsPrincipal is available as part of the request context as shown in the following example:

```
using System.Net;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;

public static IActionResult Run(HttpContext req, ILogger log)
{
    ClaimsPrincipal identities = req.HttpContext.User;
    // ...
    return new OkObjectResult();
}
```

Alternatively, the `ClaimsPrincipal` can simply be included as an additional parameter in the function signature:

```
using System.Net;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using Newtonsoft.Json.Linq;

public static void Run(JObject input, ClaimsPrincipal principal, ILogger log)
{
    // ...
    return;
}
```

Function access keys

Functions lets you use keys to make it harder to access your HTTP function endpoints during development. Unless the HTTP access level on an HTTP triggered function is set to `anonymous`, requests must include an API access key in the request.

While keys provide a default security mechanism, you may want to consider additional options to secure an HTTP endpoint in production. For example, it's generally not a good practice to distribute shared secret in public apps. If your function is being called from a public client, you may want to consider implementing another security mechanism. To learn more, see [Secure an HTTP endpoint in production](#).

When you renew your function key values, you must manually redistribute the updated key values to all clients that call your function.

Authorization scopes (function-level)

There are two access scopes for function-level keys:

- **Function:** These keys apply only to the specific functions under which they are defined. When used as an API key, these only allow access to that function.
- **Host:** Keys with a host scope can be used to access all functions within the function app. When used as an API key, these allow access to any function within the function app.

Each key is named for reference, and there is 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.

Master key (admin-level)

Each function app also has an admin-level host key named `_master`. In addition to providing host-level access to all functions in the app, the master key also provides administrative access to the runtime REST APIs. This key cannot be revoked. When you set an access level of `admin`, requests must use the master key; any other key results in access failure.

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 native client applications. Use caution when choosing the admin access level.

Obtaining keys

Keys are stored as part of your function app in Azure and are encrypted at rest. To view your keys, create new ones, or roll keys to new values, navigate to one of your HTTP-triggered functions in the [Azure portal](#) and select **Manage**.

The screenshot shows the Azure portal interface for managing function keys. On the left, there's a sidebar with a search bar and a dropdown for 'Visual Studio Enterprise'. Below that are sections for 'Function Apps', 'myfunctionapp' (with 'Functions' and 'HttpTriggerCSharp1' expanded), 'Integrate', 'Manage' (which is highlighted with a red box), 'Monitor', 'HttpTriggerJS1', 'Proxies', and 'Slots (preview)'. The main content area has a 'Function State' section with 'Enabled' selected. Below it is a 'Function Keys' table:

| NAME | VALUE | ACTIONS |
|----------|---------------|---|
| default | Click to show | Copy Renew Revoke |
| namedKey | Click to show | Copy Renew Revoke |

Below this is a blue button labeled 'Add new function key'. Further down is a 'Host Keys (All functions)' table:

| NAME | VALUE | ACTIONS |
|---------|---------------|---|
| _master | Click to show | Copy Renew |
| default | Click to show | Copy Renew Revoke |

Below this is another blue 'Add new host key' button.

You may obtain function keys programmatically by using [Key management APIs](#).

API key authorization

Most HTTP trigger templates require an API key in the request. So your HTTP request normally looks like the following URL:

```
https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>?code=<API_KEY>
```

The key can be included in a query string variable named `code`, as above. 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 do not 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. For more information, see [Trigger - configuration](#).

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](#).

Secure an HTTP endpoint in production

To fully secure your function endpoints in production, you should consider implementing one of the following function app-level security options. When using one of these function app-level security methods, you should set the HTTP-triggered function authorization level to `anonymous`.

Enable App Service Authentication/Authorization

The App Service platform lets you use Azure Active Directory (AAD) 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 a variety of 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](#).

Deploy your function app in isolation

Azure App Service Environment (ASE) provides a dedicated hosting environment in which to run your functions. ASE lets 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](#).

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 additional validation for webhook payloads. In version 2.x and higher, the base HTTP trigger still works and is the recommended approach for webhooks.

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.

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

[Redacted input field]

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](#)

Azure Functions HTTP output bindings

2/14/2020 • 3 minutes to read • [Edit Online](#)

Use the HTTP output binding to respond to the HTTP request sender. 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

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file. For C# class libraries, there are no attribute properties that correspond to these `function.json` properties.

| 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. |

Usage

To send an HTTP response, use the language-standard response patterns. In C# or C# script, make the function return type `IActionResult` or `Task<IActionResult>`. In C#, a return value attribute isn't required.

For example responses, see the [trigger example](#).

host.json settings

This section describes the global configuration settings available for this binding in versions 2.x and higher. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about global configuration settings in versions 2.x and beyond, 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](#).

```
{
  "extensions": {
    "http": {
      "routePrefix": "api",
      "maxOutstandingRequests": 200,
      "maxConcurrentRequests": 100,
      "dynamicThrottlesEnabled": true,
      "hsts": {
        "isEnabled": true,
        "maxAge": "10"
      },
      "customHeaders": {
        "X-Content-Type-Options": "nosniff"
      }
    }
  }
}
```

| 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. |
| dynamicThrottlesEnabled | true* | <p>When enabled, this setting causes the request processing pipeline to periodically check system performance counters like <code>connections/thread/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.</p> <p>*The default in a Consumption plan is <code>true</code>. The default in a Dedicated plan is <code>false</code>.</p> |

| PROPERTY | DEFAULT | DESCRIPTION | | | | | | | | | | |
|--------------------------------|---|---|----------|-------------|----------------------------|---|--------------------------------|---|---------------------|---|----------------------|---|
| hsts | not enabled | <p>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 <code>HstsOptions</code> class. The above example also sets the <code>maxAge</code> property to 10 days. Supported properties of <code>hsts</code> are:</p> <table border="1"> <thead> <tr> <th>PROPERTY</th><th>DESCRIPTION</th></tr> </thead> <tbody> <tr> <td><code>excludedHosts</code></td><td>A string array of host names for which the HSTS header isn't added.</td></tr> <tr> <td><code>includeSubDomains</code></td><td>Boolean value that indicates whether the <code>includeSubDomains</code> parameter of the Strict-Transport-Security header is enabled.</td></tr> <tr> <td><code>maxAge</code></td><td>String that defines the <code>max-age</code> parameter of the Strict-Transport-Security header.</td></tr> <tr> <td><code>preload</code></td><td>Boolean that indicates whether the <code>preload</code> parameter of the Strict-Transport-Security header is enabled.</td></tr> </tbody> </table> | PROPERTY | DESCRIPTION | <code>excludedHosts</code> | A string array of host names for which the HSTS header isn't added. | <code>includeSubDomains</code> | Boolean value that indicates whether the <code>includeSubDomains</code> parameter of the Strict-Transport-Security header is enabled. | <code>maxAge</code> | String that defines the <code>max-age</code> parameter of the Strict-Transport-Security header. | <code>preload</code> | Boolean that indicates whether the <code>preload</code> parameter of the Strict-Transport-Security header is enabled. |
| PROPERTY | DESCRIPTION | | | | | | | | | | | |
| <code>excludedHosts</code> | A string array of host names for which the HSTS header isn't added. | | | | | | | | | | | |
| <code>includeSubDomains</code> | Boolean value that indicates whether the <code>includeSubDomains</code> parameter of the Strict-Transport-Security header is enabled. | | | | | | | | | | | |
| <code>maxAge</code> | String that defines the <code>max-age</code> parameter of the Strict-Transport-Security header. | | | | | | | | | | | |
| <code>preload</code> | Boolean that indicates whether the <code>preload</code> parameter of the Strict-Transport-Security header is enabled. | | | | | | | | | | | |
| maxConcurrentRequests | 100* | <p>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.</p> <p>*The default for a Consumption plan is 100. The default for a Dedicated plan is unbounded (<code>-1</code>).</p> | | | | | | | | | | |

| PROPERTY | DEFAULT | DESCRIPTION |
|------------------------|---------|---|
| maxOutstandingRequests | 200* | <p>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.</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](#)

Mobile Apps bindings for Azure Functions

11/2/2020 • 8 minutes to read • [Edit Online](#)

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.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: [C#](#), [JavaScript](#), [Java](#), or [Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#](#), [C# script](#), [F#](#), [Java](#), [JavaScript](#), or [Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

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 \(.csx\)](#)
- [JavaScript](#)

Input - C# script example

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:

```
{  
  "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:

```
#r "Newtonsoft.Json"  
using Newtonsoft.Json.Linq;  
  
public static void Run(string myQueueItem, JObject record)  
{  
  if (record != null)  
  {  
    record["Text"] = "This has changed."  
  }  
}
```

Input - JavaScript

The following example shows a Mobile Apps input binding in a *function.json* file and a [JavaScript 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:

```
{
  "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 JavaScript code:

```
module.exports = function (context, myQueueItem) {
  context.log(context.bindings.record);
  context.done();
};
```

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 |
|------------------------|--------------------|--|
| type | n/a | Must be set to "mobileTable" |
| direction | n/a | Must be set to "in" |
| name | n/a | Name of input parameter in function signature. |
| tableName | TableName | Name of the mobile app's data table |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| id | Id | 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. |
| connection | Connection | 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 | ApiKey | 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, app settings go into the [local.settings.json file](#).

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

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

Output - C# example

The following example shows a [C# function](#) that is triggered by a queue message and creates a record in a mobile app table.

```
[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 - C# script example

The following example shows a Mobile Apps output binding in a *function.json* file and a [C# script function](#) that uses the binding. The function is triggered by a queue message and creates a new record with hard-coded value for the `Text` property.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "name": "myQueueItem",
      "queueName": "myqueue-items",
      "connection": "",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "record",
      "type": "mobileTable",
      "tableName": "MyTable",
      "connection": "My_MobileApp_Url",
      "apiKey": "My_MobileApp_Key",
      "direction": "out"
    }
  ]
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
public static void Run(string myQueueItem, out object record)
{
    record = new {
        Text = $"I'm running in a C# function! {myQueueItem}"
    };
}
```

Output - JavaScript example

The following example shows a Mobile Apps output binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function is triggered by a queue message and creates a new record with hard-coded value for the `Text` property.

Here's the binding data in the `function.json` file:

```
{  
  "bindings": [  
    {  
      "name": "myQueueItem",  
      "queueName": "myqueue-items",  
      "connection": "",  
      "type": "queueTrigger",  
      "direction": "in"  
    },  
    {  
      "name": "record",  
      "type": "mobileTable",  
      "tableName": "MyTable",  
      "connection": "My_MobileApp_Url",  
      "apiKey": "My_MobileApp_Key",  
      "direction": "out"  
    }  
  "disabled": false  
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function (context, myQueueItem) {  
  
  context.bindings.record = {  
    text : "I'm running in a Node function! Data: '" + 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:

```
[FunctionName("MobileAppsOutput")]  
[return: MobileTable(ApiKeySetting = "MyMobileAppKey", TableName = "MyTable", MobileAppUriSetting =  
"MyMobileAppUri")]  
public static object Run(  
  [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] string myQueueItem,  
  TraceWriter log)  
{  
  ...  
}
```

For a complete example, see [Output - C# example](#).

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 |
| <code>connection</code> | <code>MobileAppUriSetting</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>ApiKeySetting</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 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, app settings go into the `local.settings.json` file.

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](#)

Notification Hubs output binding for Azure Functions

11/2/2020 • 7 minutes to read • [Edit Online](#)

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.

Azure Notification Hubs must be configured for the Platform Notifications Service (PNS) you want to use. To learn how to get push notifications in your client app from Notification Hubs, see [Getting started with Notification Hubs](#) and select your target client platform from the drop-down list near the top of the page.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: [C#, JavaScript, Java, or Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#, C# script, F#, Java, JavaScript, or Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

IMPORTANT

Google has [deprecated Google Cloud Messaging \(GCM\)](#) in favor of [Firebase Cloud Messaging \(FCM\)](#). This output binding 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

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 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 |

Packages - Functions 2.x and higher

This binding is not available in Functions 2.x and higher.

Example - template

The notifications you send can be native notifications or [template notifications](#). Native notifications target a

specific client platform as configured in the `platform` property of the output binding. A template notification can be used to target multiple platforms.

See the language-specific example:

- [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.

```
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 are using asynchronous code, out parameters are not allowed. In this case use `IAsyncCollector` to return your template notification. The following code is an asynchronous example of the code above.

```
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.

```

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](#).

```

#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`.

```

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`.

```

module.exports = 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!"
    };
    context.done();
};

```

Example - APNS native

This C# script example shows how to send a native APNS notification.

```
#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 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 WNS toast notification.

```

#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 explains the binding configuration properties that you set in the `function.json` file and the `NotificationHub` attribute:

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| <code>type</code> | n/a | Must be set to <code>notificationHub</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 the notification hub message. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|-------------------------|--|
| tagExpression | TagExpression | Tag expressions allow you to specify that notifications be delivered to a set of devices that have registered to receive notifications that match the tag expression. For more information, see Routing and tag expressions . |
| hubName | HubName | Name of the notification hub resource in the Azure portal. |
| connection | ConnectionStringSetting | The name of an app setting that contains a Notification Hubs connection string. The connection string must be set to the <i>DefaultFullSharedAccessSignature</i> value for your notification hub. See Connection string setup later in this article. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| 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 on using templates in general to send cross platform notifications with an Azure Notification Hub, see Templates. When set, platform 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 Sending push notifications to iOS with Azure Notification Hubs. • <code>adm</code>—Amazon Device Messaging. For more information on configuring the notification hub for ADM and receiving the notification in a Kindle app, see Getting Started with Notification Hubs for Kindle apps. • <code>wns</code>—Windows Push Notification Services targeting Windows platforms. Windows Phone 8.1 and later is also supported by WNS. For more information, see Getting started with Notification Hubs for Windows Universal Platform Apps. • <code>mpns</code>—Microsoft Push Notification Service. This platform supports Windows Phone 8 and earlier Windows Phone platforms. For more information, see Sending push notifications with Azure Notification Hubs on Windows Phone. |

When you're developing locally, app settings go into the `local.settings.json` file.

function.json file example

Here's an example of a Notification Hubs binding in a `function.json` file.

```
{
  "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 right 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. This copies the connection string for the **DefaultFullSharedAccessSignature** policy to your notification hub. This connection string lets your function send notification messages to the hub.

The screenshot shows the 'Access Policies' page for a notification hub. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick Start, Properties, and others. The 'Access Policies' option is highlighted with a red box. The main area shows a table with two rows:

| POLICY NAME | PERMISSION | CONNECTION STRING |
|------------------------------------|--------------------|---------------------------------------|
| DefaultListenSharedAccessSignat... | Listen | Endpoint=sb://glengafunchub-ns.ser... |
| DefaultFullSharedAccessSignature | Listen,Manage,Send | Endpoint=sb://glengafunchub-ns.ser... |

2. Navigate to your function app in the Azure portal, choose **Application settings**, add a key such as **MyHubConnectionString**, paste the copied **DefaultFullSharedAccessSignature** for your notification hub as the value, and then click **Save**.

The name of this application setting is what goes in the output binding connection setting in `function.json` or the .NET attribute. See the [Configuration section](#) earlier in this article.

When you're developing locally, app settings go into the `local.settings.json` file.

Exceptions and return codes

| BINDING | REFERENCE |
|------------------|----------------------------------|
| Notification Hub | Operations Guide |

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Queue storage trigger and bindings for Azure Functions overview

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 |

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

Azure Storage SDK version in Functions 1.x

In Functions 1.x, the Storage triggers and bindings use version 7.2.1 of the Azure Storage SDK ([WindowsAzure.Storage](#) NuGet package). If you reference a different version of the Storage SDK, and you bind to a Storage SDK type in your function signature, the Functions runtime may report that it can't bind to that type. The solution is to make sure your project references [WindowsAzure.Storage 7.2.1](#).

Next steps

- [Run a function as queue storage data changes \(Trigger\)](#)
- [Write queue storage messages \(Output binding\)](#)

Azure Queue storage trigger for Azure Functions

12/4/2020 • 9 minutes to read • [Edit Online](#)

The queue storage trigger runs a function as messages are added to Azure Queue storage.

Encoding

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.

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.

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

The following example shows a [C# function](#) that polls the `myqueue-items` queue and writes a log each time a queue item is processed.

```
public static class QueueFunctions
{
    [FunctionName("QueueTrigger")]
    public static void QueueTrigger(
        [QueueTrigger("myqueue-items")] string myQueueItem,
        ILogger log)
    {
        log.LogInformation($"C# function processed: {myQueueItem}");
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

In [C# class libraries](#), use the following attributes to configure a queue trigger:

- [QueueTriggerAttribute](#)

The attribute's constructor takes the name of the queue to monitor, as shown in the following example:

```
[FunctionName("QueueTrigger")]
public static void Run(
    [QueueTrigger("myqueue-items")] string myQueueItem,
    ILogger log)
{
    ...
}
```

You can set the `Connection` property to specify the app setting that contains the storage account connection string to use, as shown in the following example:

```
[FunctionName("QueueTrigger")]
public static void Run(
    [QueueTrigger("myqueue-items", Connection = "StorageConnectionAppSetting")] string myQueueItem,
    ILogger log)
{
    ....
}
```

For a complete example, see [example](#).

- [StorageAccountAttribute](#)

Provides another way to specify the storage account to use. 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:

```
[StorageAccount("ClassLevelStorageAppSetting")]
public static class AzureFunctions
{
    [FunctionName("QueueTrigger")]
    [StorageAccount("FunctionLevelStorageAppSetting")]
    public static void Run( //...
    {
        ...
    }
}
```

The storage account to use is determined in the following order:

- The `QueueTrigger` attribute's `connection` property.
- The `StorageAccount` attribute applied to the same parameter as the `QueueTrigger` attribute.
- The `StorageAccount` attribute applied to the function.
- The `StorageAccount` attribute applied to the class.
- The "AzureWebJobsStorage" app setting.

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `QueueTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| <code>type</code> | n/a | Must be set to <code>queueTrigger</code> . This property is set automatically when you create the trigger in the Azure portal. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| direction | n/a | In the <code>function.json</code> file only. Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal. |
| name | n/a | The name of the variable that contains the queue item payload in the function code. |
| queueName | QueueName | The name of the queue to poll. |
| connection | Connection | The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "MyStorage." If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code> . |

When you're developing locally, app settings go into the [local.settings.json](#) file.

Usage

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

Access the message data by using a method parameter such as `string paramName`. You can bind to any of the following types:

- Object - The Functions runtime deserializes a JSON payload into an instance of an arbitrary class defined in your code.
- `string`
- `byte[]`
- [CloudQueueMessage](#)

If you try to bind to `CloudQueueMessage` and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

Message 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. The properties are members of the [CloudQueueMessage](#) class.

| 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 | int | 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. |

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 `<originalqueuename>-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.

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 two seconds and then checks for another message
- When no message is found, it waits about four seconds 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.

In regard to billing, time spent polling by the runtime is "free" and not counted against your account.

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

12/4/2020 • 9 minutes to read • [Edit Online](#)

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

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

The following example shows a [C# function](#) that creates a queue message for each HTTP request received.

```
[StorageAccount("MyStorageConnectionAppSetting")]
public static class QueueFunctions
{
    [FunctionName("QueueOutput")]
    [return: Queue("myqueue-items")]
    public static string QueueOutput([HttpTrigger] dynamic input, ILogger log)
    {
        log.LogInformation($"C# function processed: {input.Text}");
        return input.Text;
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

In [C# class libraries](#), use the [QueueAttribute](#).

The attribute applies to an `out` parameter or the return value of the function. The attribute's constructor takes the name of the queue, as shown in the following example:

```
[FunctionName("QueueOutput")]
[return: Queue("myqueue-items")]
public static string Run([HttpTrigger] dynamic input, ILogger log)
{
    ...
}
```

You can set the `Connection` property to specify the storage account to use, as shown in the following example:

```
[FunctionName("QueueOutput")]
[return: Queue("myqueue-items", Connection = "StorageConnectionAppSetting")]
public static string Run([HttpTrigger] dynamic input, ILogger log)
{
    ...
}
```

For a complete example, see [Output example](#).

You can use the `StorageAccount` attribute to specify the storage account at class, method, or parameter level.

For more information, see [Trigger - attributes](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Queue` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------|-------------------------|---|
| <code>type</code> | n/a | Must be set to <code>queue</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>direction</code> | n/a | Must be set to <code>out</code> . This property is set automatically when you create the trigger in the Azure portal. |
| <code>name</code> | n/a | The name of the variable that represents the queue in function code. Set to <code>\$return</code> to reference the function return value. |
| <code>queueName</code> | <code>QueueName</code> | The name of the queue. |
| <code>connection</code> | <code>Connection</code> | The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "MyStorage." If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code> . |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

Write a single queue message by using a method parameter such as `out T paramName`. You can use the method return type instead of an `out` parameter, and `T` can be any of the following types:

- An object serializable as JSON
- `string`
- `byte[]`
- [CloudQueueMessage](#)

If you try to bind to `CloudQueueMessage` and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

In C# and C# script, write multiple queue messages by using one of the following types:

- `ICollector<T>` or `IAsyncCollector<T>`
- [CloudQueue](#)

Exceptions and return codes

| BINDING | REFERENCE |
|--------------------|-------------------------------------|
| Queue | Queue Error Codes |
| Blob, Table, Queue | Storage Error Codes |
| Blob, Table, Queue | Troubleshooting |

host.json settings

This section describes the global configuration settings available for this binding in versions 2.x and higher. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about global configuration settings in versions 2.x and beyond, 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](#).

```
{
  "version": "2.0",
  "extensions": {
    "queues": {
      "maxPollingInterval": "00:00:02",
      "visibilityTimeout" : "00:00:30",
      "batchSize": 16,
      "maxDequeueCount": 5,
      "newBatchThreshold": 8
    }
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------|----------|---|
| maxPollingInterval | 00:00:01 | The maximum interval between queue polls. Minimum is 00:00:00.100 (100 ms) and increments up to 00:01:00 (1 min). In 1.x the data type is milliseconds, and in 2.x and higher it is a TimeSpan. |
| 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 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. |

| PROPERTY | DEFAULT | DESCRIPTION |
|-------------------|-------------|---|
| newBatchThreshold | batchSize/2 | Whenever the number of messages being processed concurrently gets down to this number, the runtime retrieves another batch. |

Next steps

- Run a function as queue storage data changes (Trigger)

Azure Functions SendGrid bindings

12/4/2020 • 6 minutes to read • [Edit Online](#)

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:

- Create your first function: [C#, JavaScript, Java, or Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#, C# script, F#, Java, JavaScript, or Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

Packages - Functions 1.x

The SendGrid bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.SendGrid](#) NuGet package, version 2.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 |

Packages - Functions 2.x and higher

The SendGrid bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.SendGrid](#) NuGet package, version 3.x. Source code for the package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

Add support in your preferred development environment using the following methods.

| DEVELOPMENT ENVIRONMENT | APPLICATION TYPE | TO ADD SUPPORT |
|-------------------------|-------------------------------------|---|
| Visual Studio | C# class library | Install the NuGet package |
| Visual Studio Code | Based on core tools | Register the extension bundle Installing the Azure Tools extension is recommended. |
| Any other editor/IDE | Based on core tools | Register the extension bundle |

| DEVELOPMENT ENVIRONMENT | APPLICATION TYPE | TO ADD SUPPORT |
|-------------------------|-----------------------|---|
| Azure Portal | Online only in portal | Installs when adding a binding See Update your extensions to update existing binding extensions without having to republish your function app. |

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that uses a Service Bus queue trigger and a SendGrid output binding.

Synchronous

```
using SendGrid.Helpers.Mail;
using System.Text.Json;

...

[FunctionName("SendEmail")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")] Message email,
    [SendGrid(ApiKey = "CustomSendGridKeyAppSettingName")] out SendGridMessage message)
{
    var emailObject = JsonSerializer.Deserialize<OutgoingEmail>(Encoding.UTF8.GetString(email.Body));

    message = new SendGridMessage();
    message.AddTo(emailObject.To);
    message.AddContent("text/html", emailObject.Body);
    message.SetFrom(new EmailAddress(emailObject.From));
    message.SetSubject(emailObject.Subject);
}

public class OutgoingEmail
{
    public string To { get; set; }
    public string From { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }
}
```

Asynchronous

```

using SendGrid.Helpers.Mail;
using System.Text.Json;

...

[FunctionName("SendEmail")]
public static async Task Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")] Message email,
    [SendGrid(ApiKey = "CustomSendGridKeyAppSettingName")] IAsyncCollector<SendGridMessage> messageCollector)
{
    var emailObject = JsonSerializer.Deserialize<OutgoingEmail>(Encoding.UTF8.GetString(email.Body));

    var message = new SendGridMessage();
    message.AddTo(emailObject.To);
    message.AddContent("text/html", emailObject.Body);
    message.SetFrom(new EmailAddress(emailObject.From));
    message.SetSubject(emailObject.Subject);

    await messageCollector.AddAsync(message);
}

public class OutgoingEmail
{
    public string To { get; set; }
    public string From { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }
}

```

You can omit setting the attribute's `ApiKey` property if you have your API key in an app setting named "AzureWebJobsSendGridApiKey".

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `SendGrid` attribute.

For information about attribute properties that you can configure, see [Configuration](#). Here's a `SendGrid` attribute example in a method signature:

```

[FunctionName("SendEmail")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")] OutgoingEmail email,
    [SendGrid(ApiKey = "CustomSendGridKeyAppSettingName")] out SendGridMessage message)
{
    ...
}

```

For a complete example, see [C# example](#).

Configuration

The following table lists the binding configuration properties available in the `function.json` file and the `SendGrid` attribute/annotation.

| FUNCTION.JSON PROPERTY | ATTRIBUTE/ANNOTATION PROPERTY | DESCRIPTION | OPTIONAL |
|------------------------|-------------------------------|---|----------|
| type | n/a | Must be set to <code>sendGrid</code> . | No |
| direction | n/a | Must be set to <code>out</code> . | No |
| name | n/a | The variable name used in function code for the request or request body. This value is <code>\$return</code> when there is only one return value. | No |
| apiKey | ApiKey | The name of an app setting that contains your API key. If not set, the default app setting name is <code>AzureWebJobsSendGridApiKey</code> . | No |
| to | To | The recipient's email address. | Yes |
| from | From | The sender's email address. | Yes |
| subject | Subject | The subject of the email. | Yes |
| text | Text | The email content. | Yes |

Optional properties may have default values defined in the binding and either added or overridden programmatically.

When you're developing locally, app settings go into the [local.settings.json file](#).

host.json settings

This section describes the global configuration settings available for this binding in versions 2.x and higher. The example host.json file below contains only the version 2.x+ settings for this binding. For more information about global configuration settings in versions 2.x and beyond, 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](#).

```
{
  "version": "2.0",
  "extensions": {
    "sendGrid": {
      "from": "Azure Functions <samples@functions.com>"
    }
  }
}
```

| 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

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 |

Add to your Functions app

NOTE

The Service Bus binding doesn't currently support authentication using a managed identity. Instead, please use a [Service Bus shared access signature](#).

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 4.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

Functions 1.x

Functions 1.x apps automatically have a reference to the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x.

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

11/2/2020 • 8 minutes to read • [Edit Online](#)

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](#).

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that reads [message metadata](#) and logs a Service Bus queue message:

```
[FunctionName("ServiceBusQueueTriggerCSharp")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")]
    string myQueueItem,
    Int32 deliveryCount,
    DateTime enqueuedTimeUtc,
    string messageId,
    ILogger log)
{
    log.LogInformation($"C# ServiceBus queue trigger function processed message: {myQueueItem}");
    log.LogInformation($"EnqueuedTimeUtc={enqueuedTimeUtc}");
    log.LogInformation($"DeliveryCount={deliveryCount}");
    log.LogInformation($"MessageId={messageId}");
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the following attributes to configure a Service Bus trigger:

- [ServiceBusTriggerAttribute](#)

The attribute's constructor takes the name of the queue or the topic and subscription. In Azure Functions version 1.x, you can also specify the connection's access rights. If you don't specify access rights, the default is `Manage`. For more information, see the [Trigger - configuration](#) section.

Here's an example that shows the attribute used with a string parameter:

```
[FunctionName("ServiceBusQueueTriggerCSharp")]
public static void Run(
    [ServiceBusTrigger("myqueue")] string myQueueItem, ILogger log)
{
    ...
}
```

Since 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, as shown in the following example:

```
[FunctionName("ServiceBusQueueTriggerCSharp")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")]
    string myQueueItem, ILogger log)
{
    ...
}
```

For a complete example, see [Trigger - example](#).

- [ServiceBusAccountAttribute](#)

Provides another way to specify the Service Bus account to use. The constructor takes the name of an app setting that contains a Service Bus connection string. The attribute can be applied at the parameter, method, or class level. The following example shows class level and method level:

```
[ServiceBusAccount("ClassLevelServiceBusAppSetting")]
public static class AzureFunctions
{
    [ServiceBusAccount("MethodLevelServiceBusAppSetting")]
    [FunctionName("ServiceBusQueueTriggerCSharp")]
    public static void Run(
        [ServiceBusTrigger("myqueue", AccessRights.Manage)]
        string myQueueItem, ILogger log)
    {
        ...
    }
}
```

The Service Bus account to use is determined in the following order:

- The `ServiceBusTrigger` attribute's `Connection` property.
- The `ServiceBusAccount` attribute applied to the same parameter as the `ServiceBusTrigger` attribute.
- The `ServiceBusAccount` attribute applied to the function.
- The `ServiceBusAccount` attribute applied to the class.
- The "AzureWebJobsServiceBus" app setting.

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `ServiceBusTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|-------------|
|------------------------|--------------------|-------------|

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------|-------------------------|---|
| type | n/a | Must be set to "serviceBusTrigger". This property is set automatically when you create the trigger in the Azure portal. |
| direction | n/a | Must be set to "in". This property is set automatically when you create the trigger in the Azure portal. |
| name | n/a | The name of the variable that represents the queue or topic message in function code. |
| queueName | QueueName | Name of the queue to monitor. Set only if monitoring a queue, not for a topic. |
| topicName | TopicName | Name of the topic to monitor. Set only if monitoring a topic, not for a queue. |
| subscriptionName | SubscriptionName | Name of the subscription to monitor. Set only if monitoring a topic, not for a queue. |
| connection | Connection | <p>The name of an app setting that contains the Service Bus connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name. For example, if you set <code>connection</code> to "MyServiceBus", the Functions runtime looks for an app setting that is named "AzureWebJobsMyServiceBus". If you leave <code>connection</code> empty, the Functions runtime uses the default Service Bus connection string in the app setting that is named "AzureWebJobsServiceBus".</p> <p>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.</p> |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| accessRights | Access | 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 | 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, app settings go into the [local.settings.json](#) file.

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following parameter types are available for the queue or topic message:

- `string` - If the message is text.
- `byte[]` - Useful for binary data.
- A custom type - If the message contains JSON, Azure Functions tries to deserialize the JSON data.
- `BrokeredMessage` - Gives you the deserialized message with the `BrokeredMessage.GetBody<T>()` method.
- `MessageReceiver` - Used to receive and acknowledge messages from the message container (required when `autoComplete` is set to `false`)

These parameter types are for Azure Functions version 1.x; for 2.x and higher, use `Message` instead of `BrokeredMessage`.

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 `OnMessageOptions.MaxAutoRenewDuration`. The maximum allowed for this setting is 5 minutes according to the Service Bus documentation, whereas you can increase the Functions time limit from the default of 5 minutes to 10 minutes. For Service Bus functions you wouldn't want to do that then, because you'd exceed the Service Bus renewal limit.

Message metadata

The Service Bus 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 properties are members of the [Message](#) class.

| PROPERTY | TYPE | DESCRIPTION |
|-------------------------------|--|---|
| <code>ContentType</code> | <code>string</code> | A content type identifier utilized by the sender and receiver for application-specific logic. |
| <code>CorrelationId</code> | <code>string</code> | The correlation ID. |
| <code>DeadLetterSource</code> | <code>string</code> | The dead letter source. |
| <code>DeliveryCount</code> | <code>Int32</code> | The number of deliveries. |
| <code>EnqueuedTimeUtc</code> | <code>DateTime</code> | The enqueued time in UTC. |
| <code>ExpiresAtUtc</code> | <code>DateTime</code> | The expiration time in UTC. |
| <code>Label</code> | <code>string</code> | The application-specific label. |
| <code>MessageId</code> | <code>string</code> | A user-defined value that Service Bus can use to identify duplicate messages, if enabled. |
| <code>MessageReceiver</code> | <code>MessageReceiver</code> | Service Bus message receiver. Can be used to abandon, complete, or deadletter the message. |
| <code>MessageSession</code> | <code>MessageSession</code> | A message receiver specifically for session-enabled queues and topics. |
| <code>ReplyTo</code> | <code>string</code> | The reply to queue address. |
| <code>SequenceNumber</code> | <code>long</code> | The unique number assigned to a message by the Service Bus. |
| <code>To</code> | <code>string</code> | The send to address. |
| <code>UserProperties</code> | <code>IDictionary<string, object></code> | Properties set by the sender. |

See [code examples](#) that use these properties earlier in this article.

Next steps

- Send Azure Service Bus messages from Azure Functions (Output binding)

Azure Service Bus output binding for Azure Functions

11/2/2020 • 10 minutes to read • [Edit Online](#)

Use Azure Service Bus output binding to send queue or topic messages.

For information on setup and configuration details, see the [overview](#).

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that sends a Service Bus queue message:

```
[FunctionName("ServiceBusOutput")]
[return: ServiceBus("myqueue", Connection = "ServiceBusConnection")]
public static string ServiceBusOutput([HttpTrigger] dynamic input, ILogger log)
{
    log.LogInformation($"C# function processed: {input.Text}");
    return input.Text;
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the [ServiceBusAttribute](#).

The attribute's constructor takes the name of the queue or the topic and subscription. You can also specify the connection's access rights. How to choose the access rights setting is explained in the [Output - configuration](#) section. Here's an example that shows the attribute applied to the return value of the function:

```
[FunctionName("ServiceBusOutput")]
[return: ServiceBus("myqueue")]
public static string Run([HttpTrigger] dynamic input, ILogger log)
{
    ...
}
```

You can set the `Connection` property to specify the name of an app setting that contains the Service Bus connection string to use, as shown in the following example:

```
[FunctionName("ServiceBusOutput")]
[return: ServiceBus("myqueue", Connection = "ServiceBusConnection")]
public static string Run([HttpTrigger] dynamic input, ILogger log)
{
    ...
}
```

For a complete example, see [Output - example](#).

You can use the `ServiceBusAccount` attribute to specify the Service Bus account to use at class, method, or parameter level. For more information, see [Trigger - attributes](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `ServiceBus` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|------------------------|---|
| <code>type</code> | n/a | Must be set to "serviceBus". This property is set automatically when you create the trigger in the Azure portal. |
| <code>direction</code> | n/a | Must be set to "out". This property is set automatically when you create the trigger in the Azure portal. |
| <code>name</code> | n/a | The name of the variable that represents the queue or topic message in function code. Set to "\$return" to reference the function return value. |
| <code>queueName</code> | <code>QueueName</code> | Name of the queue. Set only if sending queue messages, not for a topic. |
| <code>topicName</code> | <code>TopicName</code> | Name of the topic. Set only if sending topic messages, not for a queue. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------------------|-------------------------|---|
| <code>connection</code> | <code>Connection</code> | <p>The name of an app setting that contains the Service Bus connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name. For example, if you set <code>connection</code> to "MyServiceBus", the Functions runtime looks for an app setting that is named "AzureWebJobsMyServiceBus". If you leave <code>connection</code> empty, the Functions runtime uses the default Service Bus connection string in the app setting that is named "AzureWebJobsServiceBus".</p> <p>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.</p> |
| <code>accessRights</code> (v1 only) | <code>Access</code> | <p>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.</p> |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

In Azure Functions 1.x, the runtime creates the queue if it doesn't exist and you have set `accessRights` to `manage`. In 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 will fail.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Use the following parameter types for the output binding:

- `out T paramName` - `T` can be any JSON-serializable type. If the parameter value is null when the function exits, Functions creates the message with a null object.

- `out string` - If the parameter value is null when the function exits, Functions does not create a message.
- `out byte[]` - If the parameter value is null when the function exits, Functions does not create a message.
- `out BrokeredMessage` - If the parameter value is null when the function exits, Functions does not create a message (for Functions 1.x)
- `out Message` - If the parameter value is null when the function exits, Functions does not create a message (for Functions 2.x and higher)
- `ICollector<T>` or `IAsyncCollector<T>` (for async methods) - For creating multiple messages. A message is created when you call the `Add` method.

When working with C# functions:

- Async functions need a return value or `IAsyncCollector` instead of an `out` parameter.
- To access the session ID, bind to a `Message` type and use the `sessionId` property.

Exceptions and return codes

| BINDING | REFERENCE |
|-------------|---|
| Service Bus | Service Bus Error Codes |
| Service Bus | Service Bus Limits |

host.json settings

This section describes the global configuration settings available for this binding in versions 2.x and higher. The example host.json file below contains only the settings for this binding. For more information about global configuration settings, see [host.json reference for Azure Functions version](#).

NOTE

For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

```
{
  "version": "2.0",
  "extensions": {
    "serviceBus": {
      "prefetchCount": 100,
      "messageHandlerOptions": {
        "autoComplete": true,
        "maxConcurrentCalls": 32,
        "maxAutoRenewDuration": "00:05:00"
      },
      "sessionHandlerOptions": {
        "autoComplete": false,
        "messageWaitTimeout": "00:00:30",
        "maxAutoRenewDuration": "00:55:00",
        "maxConcurrentSessions": 16
      }
    }
  }
}
```

If you have `isSessionsEnabled` set to `true`, the `sessionHandlerOptions` will be honored. If you have `isSessionsEnabled` set to `false`, the `messageHandlerOptions` will be honored.

| PROPERTY | DEFAULT | DESCRIPTION |
|-----------------------|----------|--|
| prefetchCount | 0 | Gets or sets the number of messages that the message receiver can simultaneously request. |
| maxAutoRenewDuration | 00:05:00 | The maximum duration within which the message lock will be renewed automatically. |
| autoComplete | true | <p>Whether the trigger should automatically call complete after processing, or if the function code will manually call complete.</p> <p>Setting to <code>false</code> is only supported in C#.</p> <p>If set to <code>true</code>, the trigger completes the message automatically if the function execution completes successfully, and abandons the message otherwise.</p> <p>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 MessageReceiver 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.</p> <p>In non-C# functions, exceptions in the function results in the runtime calls <code>abandonAsync</code> in the background. If no exception occurs, then <code>completeAsync</code> is called in the background.</p> |
| maxConcurrentCalls | 16 | The maximum number of concurrent calls to the callback that the message pump should initiate per scaled instance. By default, the Functions runtime processes multiple messages concurrently. |
| maxConcurrentSessions | 2000 | The maximum number of sessions that can be handled concurrently per scaled instance. |

Next steps

- Run a function when a Service Bus queue or topic message is created (Trigger)

SignalR Service bindings for Azure Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 supports input and output bindings for SignalR Service.

| ACTION | TYPE |
|--|--------------------------------|
| Return the service endpoint URL and access token | Input binding |
| Send SignalR Service messages | Output binding |

Add to your Functions app

Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

| LANGUAGE | ADD BY... | REMARKS |
|---|--|---|
| C# | Installing the NuGet package , version 3.x | |
| C# Script, Java, JavaScript, Python, PowerShell | Registering the extension bundle | The Azure Tools extension is recommended to use with Visual Studio Code. |
| C# Script (online-only in Azure portal) | Adding a binding | To update existing binding extensions without having to republish your function app, see Update your extensions . |

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](#).

Annotations library (Java only)

To use the SignalR Service annotations in Java functions, you need to add a dependency to the *azure-functions-java-library-signalr* artifact (version 1.0 or higher) to your *pom.xml* file.

```
<dependency>
    <groupId>com.microsoft.azure.functions</groupId>
    <artifactId>azure-functions-java-library-signalr</artifactId>
    <version>1.0.0</version>
</dependency>
```

Next steps

- [Return the service endpoint URL and access token \(Input binding\)](#)
- [Send SignalR Service messages \(Output binding\)](#)

SignalR Service trigger binding for Azure Functions

11/2/2020 • 4 minutes to read • [Edit Online](#)

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.

For information on setup and configuration details, see the [overview](#).

Example

The following example shows a function that receives a message using the trigger binding and log the message.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)

SignalR Service trigger binding for C# has two programming models. Class based model and traditional model. Class based model can provide a consistent SignalR server-side programming experience. And traditional model provides more flexibility and similar with other function bindings.

With Class based model

See [Class based model](#) for details.

```
public class SignalRTTestHub : ServerlessHub
{
    [FunctionName("SignalRTTest")]
    public async Task SendMessage([SignalRTrigger]InvocationContext invocationContext, string message,
ILogger logger)
    {
        logger.LogInformation($"Receive {message} from {invocationContext.ConnectionId}.");
    }
}
```

With Traditional model

Traditional model obeys the convention of Azure Function developed by C#. If you're not familiar with it, you can learn from [documents](#).

```
[FunctionName("SignalRTTest")]
public static async Task Run([SignalRTrigger("SignalRTTest", "messages", "SendMessage", parameterNames: new
string[] {"message"})]InvocationContext invocationContext, string message, ILogger logger)
{
    logger.LogInformation($"Receive {message} from {invocationContext.ConnectionId}.");
}
```

Use attribute `[SignalRParameter]` to simplify `ParameterNames`

As it's bit cumbersome to use `ParameterNames`, `SignalRParameter` is provided to achieve the same purpose.

```
[FunctionName("SignalRTTest")]
public static async Task Run([SignalRTrigger("SignalRTTest", "messages", "SendMessage")]InvocationContext
invocationContext, [SignalRParameter]string message, ILogger logger)
{
    logger.LogInformation($"Receive {message} from {invocationContext.ConnectionId}.");
}
```

Configuration

SignalRTrigger

The following table explains the binding configuration properties that you set in the `function.json` file and the `SignalRTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|--------------------------------------|--------------------------------------|--|
| <code>type</code> | n/a | Must be set to <code>SignalRTrigger</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 trigger invocation context object. |
| <code>hubName</code> | <code>HubName</code> | This value must be set to the name of the SignalR hub for the function to be triggered. |
| <code>category</code> | <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> | <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> | <code>ParameterNames</code> | (Optional) A list of names that binds to the parameters. |
| <code>connectionStringSetting</code> | <code>ConnectionStringSetting</code> | The name of the app setting that contains the SignalR Service connection string (defaults to "AzureSignalRConnectionString") |

Payload

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 send from SignalR Service.

| PROPERTY IN INVOCATIONCONTEXT | 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 which the message belongs to. |
| Category | The category of the message. |
| Event | The event of the message. |
| ConnectionId | The connection ID of the client which sends the message. |
| UserId | The user identity of the client which 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` allows you to bind arguments of invocation messages to the parameters of functions. 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.

```
await connection.invoke("broadcast", message1, message2);
```

You can access these two arguments from parameter as well as assign type of parameter for them by using `ParameterNames`.

Remarks

For the parameter binding, the order matters. If you are using `ParameterNames`, the order in `ParameterNames` matches the order of the arguments you invoke in the client. If you are 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 will get an exception.

Send messages to SignalR Service trigger binding

Azure Function generates a URL for SignalR Service trigger binding and it is formatted as following:

```
https://<APP_NAME>.azurewebsites.net/runtime/webhooks/signalr?code=<API_KEY>
```

The `<API_KEY>` is generated by Azure Function. You can get the `<API_KEY>` from Azure portal as you're using SignalR Service trigger binding.

The screenshot shows the Azure portal's 'System keys' page under the 'App keys' section. A single key named 'signair_extension' is listed with its value being a hidden string. There are buttons for 'Show values', 'Filter', 'Renew key value', and a delete icon.

| Name | Value |
|-------------------|-------------------------------------|
| signair_extension | (Hidden value. Click to show value) |

You should set this URL in `UrlTemplate` in the upstream settings of SignalR Service.

Next steps

- [Azure Functions development and configuration with Azure SignalR Service](#)
- [SignalR Service Trigger binding sample](#)

SignalR Service input binding for Azure Functions

11/2/2020 • 4 minutes to read • [Edit Online](#)

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. Because the token is time-limited and can be used to authenticate a specific user to a connection, you should not cache the token or share it between clients. An HTTP trigger using this binding can be used by clients to retrieve the connection information.

For more information on how this binding is used to create a "negotiate" function that can be consumed by a SignalR client SDK, see the [Azure Functions development and configuration article](#) in the SignalR Service concepts documentation.

For information on setup and configuration details, see the [overview](#).

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that acquires SignalR connection information using the input binding and returns it over HTTP.

```
[FunctionName("negotiate")]
public static SignalRConnectionInfo Negotiate(
    [HttpTrigger(AuthorizationLevel.Anonymous)]HttpRequest req,
    [SignalRConnectionInfo(HubName = "chat")]SignalRConnectionInfo connectionInfo)
{
    return connectionInfo;
}
```

Authenticated tokens

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.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

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}`.

```
[FunctionName("negotiate")]
public static SignalRConnectionInfo Negotiate(
    [HttpTrigger(AuthorizationLevel.Anonymous)]HttpRequest req,
    [SignalRConnectionInfo]
        (HubName = "chat", UserId = "{headers.x-ms-client-principal-id}")]
    SignalRConnectionInfo connectionInfo)
{
    // connectionInfo contains an access key token with a name identifier claim set to the authenticated
    user
    return connectionInfo;
}
```

Next steps

- [Send SignalR Service messages \(Output binding\)](#)

SignalR Service output binding for Azure Functions

12/4/2020 • 8 minutes to read • [Edit Online](#)

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 authenticated to a specific user

The output binding also allows you to manage groups.

For information on setup and configuration details, see the [overview](#).

Broadcast to all clients

The following example shows a function that sends a message using the output binding to all connected clients. The *target* is the name of the method to be invoked on each client. The *Arguments* property is an array of zero or more objects to be passed to the client method.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("SendMessage")]
public static Task SendMessage(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]object message,
    [SignalR(HubName = "chat")]IAsyncCollector<SignalRMessage> signalRMessages)
{
    return signalRMessages.AddAsync(
        new SignalRMessage
        {
            Target = "newMessage",
            Arguments = new [] { message }
        });
}
```

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.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("SendMessage")]
public static Task SendMessage(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]object message,
    [SignalR(HubName = "chat")]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 }
        });
}
```

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.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("SendMessage")]
public static Task SendMessage(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]object message,
    [SignalR(HubName = "chat")]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 }
        });
}
```

Group management

SignalR Service allows users to be added to groups. Messages can then be sent to a group. You can use the `SignalR` output binding to manage a user's group membership.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Add user to a group

The following example adds a user to a group.

```
[FunctionName("addToGroup")]
public static Task AddToGroup(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]HttpRequest req,
    ClaimsPrincipal claimsPrincipal,
    [SignalR(HubName = "chat")]
    IAsyncCollector<SignalRGroupAction> signalRGroupActions)
{
    var userIdClaim = claimsPrincipal.FindFirst(ClaimTypes.NameIdentifier);
    return signalRGroupActions.AddAsync(
        new SignalRGroupAction
        {
            UserId = userIdClaim.Value,
            GroupName = "myGroup",
            Action = GroupAction.Add
        });
}
```

Remove user from a group

The following example removes a user from a group.

```
[FunctionName("removeFromGroup")]
public static Task RemoveFromGroup(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]HttpRequest req,
    ClaimsPrincipal claimsPrincipal,
    [SignalR(HubName = "chat")]
    IAsyncCollector<SignalRGroupAction> signalRGroupActions)
{
    var userIdClaim = claimsPrincipal.FindFirst(ClaimTypes.NameIdentifier);
    return signalRGroupActions.AddAsync(
        new SignalRGroupAction
        {
            UserId = userIdClaim.Value,
            GroupName = "myGroup",
            Action = GroupAction.Remove
        });
}
```

NOTE

In order to get the `ClaimsPrincipal` correctly bound, you must have configured the authentication settings in Azure Functions.

Configuration

SignalRConnectionInfo

The following table explains the binding configuration properties that you set in the `function.json` file and the `SignalRConnectionInfo` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| <code>type</code> | n/a | Must be set to <code>signalRConnectionInfo</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 connection info object. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------|-------------------------|--|
| hubName | HubName | This value must be set to the name of the SignalR hub for which the connection information is generated. |
| userId | UserId | Optional: The value of the user identifier claim to be set in the access key token. |
| connectionStringSetting | ConnectionStringSetting | The name of the app setting that contains the SignalR Service connection string (defaults to "AzureSignalRConnectionString") |

SignalR

The following table explains the binding configuration properties that you set in the *function.json* file and the `SignalR` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|-------------------------|-------------------------|--|
| type | n/a | Must be set to <code>signalR</code> . |
| direction | n/a | Must be set to <code>out</code> . |
| name | n/a | Variable name used in function code for connection info object. |
| hubName | HubName | This value must be set to the name of the SignalR hub for which the connection information is generated. |
| connectionStringSetting | ConnectionStringSetting | The name of the app setting that contains the SignalR Service connection string (defaults to "AzureSignalRConnectionString") |

When you're developing locally, app settings go into the [local.settings.json](#) file.

Next steps

- [Return the service endpoint URL and access token \(Input binding\)](#)

Azure Table storage bindings for Azure Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Azure Functions integrates with [Azure Storage](#) via [triggers and bindings](#). Integrating with Table storage allows you to build functions that read and write Table storage data.

| ACTION | TYPE |
|--|--------------------------------|
| Read table storage data in a function | Input binding |
| Allow a function to write table storage data | Output binding |

Packages - Functions 2.x and higher

The Table storage bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.Storage](#) NuGet package, version 3.x. Source code for the package is in the [azure-webjobs-sdk](#) GitHub repository.

Add support in your preferred development environment using the following methods.

| DEVELOPMENT ENVIRONMENT | APPLICATION TYPE | TO ADD SUPPORT |
|-------------------------|-------------------------------------|---|
| Visual Studio | C# class library | Install the NuGet package |
| Visual Studio Code | Based on core tools | Register the extension bundle Installing the Azure Tools extension is recommended. |
| Any other editor/IDE | Based on core tools | Register the extension bundle |
| Azure Portal | Online only in portal | Installs when adding a binding See Update your extensions to update existing binding extensions without having to republish your function app. |

Packages - Functions 1.x

The Table storage bindings are provided in the [Microsoft.Azure.WebJobs](#) NuGet package, version 2.x. Source code for the package is in the [azure-webjobs-sdk](#) GitHub repository.

Support for this binding is automatically provided in all development environments. You don't have to manually install the package or register the extension.

Azure Storage SDK version in Functions 1.x

In Functions 1.x, the Storage triggers and bindings use version 7.2.1 of the Azure Storage SDK ([WindowsAzure.Storage](#) NuGet package). If you reference a different version of the Storage SDK, and you bind to a Storage SDK type in your function signature, the Functions runtime may report that it can't bind to that type. The solution is to make sure your project references [WindowsAzure.Storage 7.2.1](#).

Next steps

- [Read table storage data when a function runs](#)
- [Write table storage data from a function](#)

Azure Table storage input bindings for Azure Functions

11/2/2020 • 11 minutes to read • [Edit Online](#)

Use the Azure Table storage input binding to read a table in an Azure Storage account.

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

One entity

The following example shows a [C# function](#) that reads a single table row. For every message send to the queue, the function will be triggered.

The row key value "{queueTrigger}" indicates that the row key comes from the queue message string.

```
public class TableStorage
{
    public class MyPoco
    {
        public string PartitionKey { get; set; }
        public string RowKey { get; set; }
        public string Text { get; set; }
    }

    [FunctionName("TableInput")]
    public static void TableInput(
        [QueueTrigger("table-items")] string input,
        [Table("MyTable", "MyPartition", "{queueTrigger}")] MyPoco poco,
        ILogger log)
    {
        log.LogInformation($"PK={poco.PartitionKey}, RK={poco.RowKey}, Text={poco.Text}");
    }
}
```

CloudTable

`CloudTable` is only supported in the [Functions v2 and and higher runtimes](#).

Use a `CloudTable` method parameter to read the table by using the Azure Storage SDK. Here's an example of a function that queries an Azure Functions log table:

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using Microsoft.Azure.Cosmos.Table;
using System;
using System.Threading.Tasks;

namespace FunctionAppCloudTable2
{
    public class LogEntity : TableEntity
    {
        public string OriginalName { get; set; }
    }
    public static class CloudTableDemo
    {
        [FunctionName("CloudTableDemo")]
        public static async Task Run(
            [TimerTrigger("0 */1 * * * *")] TimerInfo myTimer,
            [Table("AzureWebJobsHostLogscommon")] CloudTable cloudTable,
            ILogger log)
        {
            log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");

            TableQuery<LogEntity> rangeQuery = new TableQuery<LogEntity>().Where(
                TableQuery.CombineFilters(
                    TableQuery.GenerateFilterCondition("PartitionKey", QueryComparisons.Equal,
                        "FD2"),
                    TableOperators.And,
                    TableQuery.GenerateFilterCondition("RowKey", QueryComparisons.GreaterThan,
                        "t")));
        }

        // Execute the query and loop through the results
        foreach (LogEntity entity in
            await cloudTable.ExecuteQuerySegmentedAsync(rangeQuery, null))
        {
            log.LogInformation(
                $"{entity.PartitionKey}\t{entity.RowKey}\t{entity.Timestamp}\t{entity.OriginalName}");
        }
    }
}

```

For more information about how to use CloudTable, see [Get started with Azure Table storage](#).

If you try to bind to `CloudTable` and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

IQueryable

`IQueryable` is only supported in the [Functions v1 runtime](#).

The following example shows a [C# function](#) that reads multiple table rows where the `MyPoco` class derives from `TableEntity`.

```

public class TableStorage
{
    public class MyPoco : TableEntity
    {
        public string Text { get; set; }
    }

    [FunctionName("TableInput")]
    public static void TableInput(
        [QueueTrigger("table-items")] string input,
        [Table("MyTable", "MyPartition")] IQueryables<MyPoco> pocos,
        ILogger log)
    {
        foreach (MyPoco poco in pocos)
        {
            log.LogInformation($"PK={poco.PartitionKey}, RK={poco.RowKey}, Text={poco.Text}");
        }
    }
}

```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the following attributes to configure a table input binding:

- [TableAttribute](#)

The attribute's constructor takes the table name, partition key, and row key. The attribute can be used on an `out` parameter or on the return value of the function, as shown in the following example:

```

[FunctionName("TableInput")]
public static void Run(
    [QueueTrigger("table-items")] string input,
    [Table("MyTable", "Http", "{queueTrigger}")] MyPoco poco,
    ILogger log)
{
    ...
}

```

You can set the `Connection` property to specify the storage account to use, as shown in the following example:

```

[FunctionName("TableInput")]
public static void Run(
    [QueueTrigger("table-items")] string input,
    [Table("MyTable", "Http", "{queueTrigger}", Connection = "StorageConnectionAppSetting")] MyPoco
    poco,
    ILogger log)
{
    ...
}

```

For a complete example, see the [C# example](#).

- [StorageAccountAttribute](#)

Provides another way to specify the storage account to use. 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:

```
[StorageAccount("ClassLevelStorageAppSetting")]
public static class AzureFunctions
{
    [FunctionName("TableInput")]
    [StorageAccount("FunctionLevelStorageAppSetting")]
    public static void Run( //...
    {
        ...
    }
}
```

The storage account to use is determined in the following order:

- The `Table` attribute's `Connection` property.
- The `StorageAccount` attribute applied to the same parameter as the `Table` attribute.
- The `StorageAccount` attribute applied to the function.
- The `StorageAccount` attribute applied to the class.
- The default storage account for the function app ("AzureWebJobsStorage" app setting).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Table` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|---------------------------|--|
| <code>type</code> | n/a | Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal. |
| <code>direction</code> | n/a | Must be set to <code>in</code> . This property is set automatically when you create the binding in the Azure portal. |
| <code>name</code> | n/a | The name of the variable that represents the table or entity in function code. |
| <code>tableName</code> | <code>TableName</code> | The name of the table. |
| <code>partitionKey</code> | <code>PartitionKey</code> | Optional. The partition key of the table entity to read. See the usage section for guidance on how to use this property. |
| <code>rowKey</code> | <code>RowKey</code> | Optional. The row key of the table entity to read. See the usage section for guidance on how to use this property. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| take | Take | Optional. The maximum number of entities to read in JavaScript. See the usage section for guidance on how to use this property. |
| filter | Filter | Optional. An OData filter expression for table input in JavaScript. See the usage section for guidance on how to use this property. |
| connection | Connection | The name of an app setting that contains the Storage connection string to use for this binding. The setting can be the name of an "AzureWebJobs" prefixed app setting or connection string name. For example, if your setting name is "AzureWebJobsMyStorage", you can specify "MyStorage" here. The Functions runtime will automatically look for an app setting that named "AzureWebJobsMyStorage". If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code> . |

When you're developing locally, app settings go into the [local.settings.json](#) file.

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)
- **Read one row in**

Set `partitionKey` and `rowKey`. Access the table data by using a method parameter `T <paramName>`. In C# script, `paramName` is the value specified in the `name` property of `function.json`. `T` is typically a type that implements `ITableEntity` or derives from `TableEntity`. The `filter` and `take` properties are not used in this scenario.

- **Read one or more rows**

Access the table data by using a method parameter `IQueryable<T> <paramName>`. In C# script, `paramName` is the value specified in the `name` property of `function.json`. `T` must be a type that implements `ITableEntity` or derives from `TableEntity`. You can use `IQueryable` methods to do any filtering required. The `partitionKey`, `rowKey`, `filter`, and `take` properties are not used in this scenario.

NOTE

`IQueryable` isn't supported in the [Functions v2 runtime](#). An alternative is to [use a CloudTable paramName method parameter](#) to read the table by using the Azure Storage SDK. If you try to bind to `CloudTable` and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

Next steps

- [Write table storage data from a function](#)

Azure Table storage output bindings for Azure Functions

11/2/2020 • 7 minutes to read • [Edit Online](#)

Use an Azure Table storage output binding to write entities to a table in an Azure Storage account.

NOTE

This output binding does not support updating existing entities. Use the `TableOperation.Replace` operation [from the Azure Storage SDK](#) to update an existing entity.

Example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that uses an HTTP trigger to write a single table row.

```
public class TableStorage
{
    public class MyPoco
    {
        public string PartitionKey { get; set; }
        public string RowKey { get; set; }
        public string Text { get; set; }
    }

    [FunctionName("TableOutput")]
    [return: Table("MyTable")]
    public static MyPoco TableOutput([HttpTrigger] dynamic input, ILogger log)
    {
        log.LogInformation($"C# http trigger function processed: {input.Text}");
        return new MyPoco { PartitionKey = "Http", RowKey = Guid.NewGuid().ToString(), Text = input.Text };
    }
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `TableAttribute`.

The attribute's constructor takes the table name. The attribute can be used on an `out` parameter or on the return value of the function, as shown in the following example:

```
[FunctionName("TableOutput")]
[return: Table("MyTable")]
public static MyPoco TableOutput(
    [HttpTrigger] dynamic input,
    ILogger log)
{
    ...
}
```

You can set the `Connection` property to specify the storage account to use, as shown in the following example:

```
[FunctionName("TableOutput")]
[return: Table("MyTable", Connection = "StorageConnectionAppSetting")]
public static MyPoco TableOutput(
    [HttpTrigger] dynamic input,
    ILogger log)
{
    ...
}
```

For a complete example, see the [C# example](#).

You can use the `StorageAccount` attribute to specify the storage account at class, method, or parameter level. For more information, see [Input - attributes](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Table` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|---------------------------|---|
| <code>type</code> | n/a | Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal. |
| <code>direction</code> | n/a | Must be set to <code>out</code> . This property is set automatically when you create the binding in the Azure portal. |
| <code>name</code> | n/a | The variable name used in function code that represents the table or entity. Set to <code>\$return</code> to reference the function return value. |
| <code>tableName</code> | <code>TableName</code> | The name of the table. |
| <code>partitionKey</code> | <code>PartitionKey</code> | The partition key of the table entity to write. See the usage section for guidance on how to use this property. |
| <code>rowKey</code> | <code>RowKey</code> | The row key of the table entity to write. See the usage section for guidance on how to use this property. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| connection | Connection | The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "MyStorage". If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code> . |

When you're developing locally, app settings go into the [local.settings.json file](#).

Usage

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Access the output table entity by using a method parameter `ICollector<T> paramName` or `IAsyncCollector<T> paramName` where `T` includes the `PartitionKey` and `RowKey` properties. These properties are often accompanied by implementing `ITableEntity` or inheriting `TableEntity`.

Alternatively you can use a `CloudTable` method parameter to write to the table by using the Azure Storage SDK. If you try to bind to `CloudTable` and get an error message, make sure that you have a reference to [the correct Storage SDK version](#).

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

12/4/2020 • 11 minutes to read • [Edit Online](#)

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:

- Create your first function: [C#](#), [JavaScript](#), [Java](#), or [Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#](#), [C# script](#), [F#](#), [Java](#), [JavaScript](#), or [Python](#).
- [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](#).

Packages - Functions 2.x and higher

The timer trigger is provided in the [Microsoft.Azure.WebJobs.Extensions](#) NuGet package, version 3.x. Source code for the package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

Support for this binding is automatically provided in all development environments. You don't have to manually install the package or register the extension.

Packages - Functions 1.x

The timer trigger is provided in the [Microsoft.Azure.WebJobs.Extensions](#) NuGet package, version 2.x. Source code for the package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

Support for this binding is automatically provided in all development environments. You don't have to manually install the package or register the extension.

Example

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

The following example shows a [C# function](#) that is executed each time the minutes have a value divisible by five (eg if the function starts at 18:57:00, the next performance will be at 19:00:00). The [TimerInfo](#) object is passed into the function.

```
[FunctionName("TimerTriggerCSharp")]
public static void Run([TimerTrigger("0 */5 * * *")]TimerInfo myTimer, ILogger log)
{
    if (myTimer.IsPastDue)
    {
        log.LogInformation("Timer is running late!");
    }
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
}
```

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

In [C# class libraries](#), use the [TimerTriggerAttribute](#).

The attribute's constructor takes a CRON expression or a `TimeSpan`. You can use `TimeSpan` only if the function app is running on an App Service plan. `TimeSpan` is not supported for Consumption or Elastic Premium Functions.

The following example shows a CRON expression:

```
[FunctionName("TimerTriggerCSharp")]
public static void Run([TimerTrigger("0 */5 * * *")]TimerInfo myTimer, ILogger log)
{
    if (myTimer.IsPastDue)
    {
        log.LogInformation("Timer is running late!");
    }
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
}
```

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `TimerTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| <code>type</code> | n/a | Must be set to "timerTrigger". This property is set automatically when you create the trigger in the Azure portal. |
| <code>direction</code> | n/a | Must be set to "in". This property is set automatically when you create the trigger in the Azure portal. |
| <code>name</code> | n/a | The name of the variable that represents the timer object in function code. |

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|---|
| schedule | ScheduleExpression | 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%". |
| runOnStartup | 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. So runOnStartup should rarely if ever be set to <code>true</code> , especially in production. |
| useMonitor | 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, app settings go into the [local.settings.json file](#).

Caution

We recommend against setting **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 Consumption plans. 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.

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.

```
{
  "schedule": {
    "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:

| 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>*</code>) | <code>0 * 5 * * * *</code> | At every minute in the hour, beginning at hour 5 |
| A range (<code>-</code> operator) | <code>5-7 * * * * *</code> | Three times a minute - at seconds 5 through 7 during every minute of every hour of each day |
| A set of values (<code>,</code> operator) | <code>5,8,10 * * * * *</code> | Three times a minute - at seconds 5, 8, and 10 during every minute of every hour of each day |
| An interval value (<code>/</code> operator) | <code>0 */5 * * * *</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.

| 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 require a **six field** format. The sixth field position is a value for seconds which is placed at the beginning of the expression. Five field cron expressions are not supported in Azure.

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.

| 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 Dedicated | Set the value to the name of the desired time zone as shown in the tz database . |

NOTE

`WEBSITE_TIME_ZONE` is not currently supported on the Linux Consumption plan.

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:

| EXAMPLE | WHEN TRIGGERED |
|--------------|----------------|
| "01:00:00" | every hour |
| "00:01:00" | every minute |
| "24:00:00" | every 24 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.

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.

| 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.

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

11/2/2020 • 7 minutes to read • [Edit Online](#)

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:

- Create your first function: [C#, JavaScript, Java, or Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#, C# script, F#, Java, JavaScript, or Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

Packages - Functions 1.x

The Twilio bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.Twilio](#) NuGet package, version 1.x. Source code for the package is in the [azure-webjobs-sdk](#) 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 |

Packages - Functions 2.x and higher

The Twilio bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.Twilio](#) NuGet package, version 3.x. Source code for the package is in the [azure-webjobs-sdk](#) GitHub repository.

Add support in your preferred development environment using the following methods.

| DEVELOPMENT ENVIRONMENT | APPLICATION TYPE | TO ADD SUPPORT |
|-------------------------|-------------------------------------|---|
| Visual Studio | C# class library | Install the NuGet package |
| Visual Studio Code | Based on core tools | Register the extension bundle Installing the Azure Tools extension is recommended. |
| Any other editor/IDE | Based on core tools | Register the extension bundle |

| DEVELOPMENT ENVIRONMENT | APPLICATION TYPE | TO ADD SUPPORT |
|-------------------------|-----------------------|---|
| Azure Portal | Online only in portal | Installs when adding a binding See Update your extensions to update existing binding extensions without having to republish your function app. |

Example - Functions 2.x and higher

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that sends a text message when triggered by a queue message.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;
namespace TwilioQueueOutput
{
    public static class QueueTwilio
    {
        [FunctionName("QueueTwilio")]
        [return: TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting = "TwilioAuthToken",
From = "+1425XXXXXXX")]
        public static CreateMessageOptions Run(
            [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] JObject order,
            ILogger log)
        {
            log.LogInformation($"C# Queue trigger function processed: {order}");

            var message = new CreateMessageOptions(new PhoneNumber(order["mobileNumber"].ToString()))
            {
                Body = $"Hello {order["name"]}, thanks for your order!"
            };

            return message;
        }
    }
}
```

This example uses the `TwilioSms` attribute with the method return value. An alternative is to use the attribute with an `out CreateMessageOptions` parameter or an `ICollector<CreateMessageOptions>` or `IAsyncCollector<CreateMessageOptions>` parameter.

Attributes and annotations

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

In [C# class libraries](#), use the `TwilioSms` attribute.

For information about attribute properties that you can configure, see [Configuration](#). Here's a `TwilioSms` attribute example in a method signature:

```
[FunctionName("QueueTwilio")]
[return: TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting = "TwilioAuthToken", From =
"+1425XXXXXX")]
public static CreateMessageOptions Run(
[QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] JObject order, ILogger log)
{
    ...
}
```

For a complete example, see [C# example](#).

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `TwilioSms` attribute.

| V1 FUNCTION.JSON PROPERTY | V2 FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|--------------------------------|--|--|
| <code>type</code> | <code>type</code> | must be set to <code>twilioSms</code> . | |
| <code>direction</code> | <code>direction</code> | must be set to <code>out</code> . | |
| <code>name</code> | <code>name</code> | Variable name used in function code for the Twilio SMS text message. | |
| <code>accountSid</code> | <code>accountSidSetting</code> | <code>AccountSidSetting</code> | This value must be set to the name of an app setting that holds your Twilio Account Sid (<code>TwilioAccountSid</code>). If not set, the default app setting name is "AzureWebJobsTwilioAccountSid". |
| <code>authToken</code> | <code>authTokenSetting</code> | <code>AuthTokenSetting</code> | This value must be set to the name of an app setting that holds your Twilio authentication token (<code>TwilioAccountAuthToken</code>). If not set, the default app setting name is "AzureWebJobsTwilioAuthToKen". |
| <code>to</code> | N/A - specify in code | <code>To</code> | This value is set to the phone number that the SMS text is sent to. |

| V1 FUNCTION.JSON PROPERTY | V2 FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|---------------------------|---------------------------|--------------------|---|
| from | from | From | This value is set to the phone number that the SMS text is sent from. |
| body | body | Body | This value can be used to hard code the SMS text message if you don't need to set it dynamically in the code for your function. |

When you're developing locally, app settings go into the [local.settings.json file](#).

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Functions warm-up trigger

12/4/2020 • 4 minutes to read • [Edit Online](#)

This article explains how to work with the warmup trigger in Azure Functions. The warmup trigger is supported only for function apps running in a [Premium plan](#). A warmup trigger is invoked when an instance is added to scale a running function app. You can use a warmup trigger to pre-load custom dependencies during the [pre-warming process](#) so that your functions are ready to start processing requests immediately.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: [C#](#), [JavaScript](#), [Java](#), or [Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#](#), [C# script](#), [F#](#), [Java](#), [JavaScript](#), or [Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

Packages - Functions 2.x and higher

The [Microsoft.Azure.WebJobs.Extensions](#) NuGet package, version [3.0.5 or higher](#) is required. Source code for the package is in the [azure-webjobs-sdk-extensions](#) GitHub repository.

Support for this binding is automatically provided in all development environments. You don't have to manually install the package or register the extension.

Trigger

The warmup trigger lets you define a function that will be run on a new instance when it is added to your running app. You can use a warmup function to open connections, load dependencies, or run any other custom logic before your app will begin receiving traffic.

The warmup trigger is intended to create shared dependencies that will be used by the other functions in your app. See [examples of shared dependencies here](#).

Note that the warmup trigger is only called during scale-out operations, not during restarts or other non-scale startups. You must ensure your logic can load all necessary dependencies without using the warmup trigger. Lazy loading is a good pattern to achieve this.

Trigger - example

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

The following example shows a [C# function](#) that will run on each new instance when it is added to your app. A return value attribute isn't required.

- Your function must be named `warmup` (case-insensitive) and there may only be one warmup function per app.
- To use warmup as a .NET class library function, please make sure you have a package reference to

Microsoft.Azure.WebJobs.Extensions >= 3.0.5

- <PackageReference Include="Microsoft.Azure.WebJobs.Extensions" Version="3.0.5" />

Placeholder comments show where in the application to declare and initialize shared dependencies. [Learn more about shared dependencies here.](#)

```
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;

namespace WarmupSample
{

    //Declare shared dependencies here

    public static class Warmup
    {
        [FunctionName("Warmup")]
        public static void Run([WarmupTrigger()] WarmupContext context,
            ILogger log)
        {
            //Initialize shared dependencies here

            log.LogInformation("Function App instance is warm 🌞");
        }
    }
}
```

Trigger - attributes

In [C# class libraries](#), the `WarmupTrigger` attribute is available to configure the function.

- [C#](#)
- [C# Script](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

This example demonstrates how to use the `warmup` attribute.

Note that your function must be called `Warmup` and there can only be one warmup function per app.

```
[FunctionName("Warmup")]
public static void Run(
    [WarmupTrigger()] WarmupContext context, ILogger log)
{
    ...
}
```

For a complete example, see the [trigger example](#).

Trigger - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `WarmupTrigger` attribute.

| FUNCTION.JSON PROPERTY | ATTRIBUTE PROPERTY | DESCRIPTION |
|------------------------|--------------------|--|
| type | n/a | Required - must be set to <code>warmupTrigger</code> . |
| direction | n/a | Required - must be set to <code>in</code> . |
| name | n/a | Required - the variable name used in function code. |

Trigger - usage

No additional information is provided to a warmup triggered function when it is invoked.

Trigger - limits

- The warmup trigger is only available to apps running on the [Premium plan](#).
- The warmup trigger is only called during scale-out operations, not during restarts or other non-scale startups. You must ensure your logic can load all necessary dependencies without using the warmup trigger. Lazy loading is a good pattern to achieve this.
- The warmup trigger cannot be invoked once an instance is already running.
- There can only be one warmup trigger function per function app.

Next steps

[Learn more about Azure functions triggers and bindings](#)

host.json reference for Azure Functions 2.x and later

12/4/2020 • 14 minutes to read • [Edit Online](#)

The *host.json* metadata file contains global configuration options that affect all functions for a function app. 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 in your [app settings](#) (for deployed apps) or your [local.settings.json](#) file (for local development).

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).

```
{
    "version": "2.0",
    "aggregator": {
        "batchSize": 1000,
        "flushTimeout": "00:00:30"
    },
    "extensions": {
        "cosmosDb": {},
        "durableTask": {},
        "eventHubs": {},
        "http": {},
        "queues": {},
        "sendGrid": {},
        "serviceBus": {}
    },
    "extensionBundle": {
        "id": "Microsoft.Azure.Functions.ExtensionBundle",
        "version": "[1.*, 2.0.0)"
    },
    "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"
            },
            "enableLiveMetrics": true,
            "enableDependencyTracking": true,
            "enablePerformanceCountersCollection": true,
            "httpAutoCollectionOptions": {
                "enableHttpTriggerExtendedInfoCollection": true,
                "enableW3CDistributedTracing": true,
                "enableResponseHeaderInjection": true
            },
            "snapshotConfiguration": {
                "agentEndpoint": null,
                "captureSnapshotMemoryWeight": 0.5,
                "failedRequestLimit": 3,
                "handleUntrackedExceptions": true,
                "isEnabled": true,
                "isEnabledInDeveloperMode": false,
                "isEnabledWhenProfiling": true,
                "isExceptionSnapsEnabled": 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
    },
    "retry": {
        "strategy": "fixedDelay",
        "maxRetryCount": 5,
        "delayInterval": "00:00:05"
    },
    "singleton": {
        "lockPeriod": "00:00:15",
        "listenerLockPeriod": "00:01:00",
        "listenerLockRecoveryPollingInterval": "00:01:00",
        "lockAcquisitionTimeout": "00:01:00",
        "lockAcquisitionPollingInterval": "00:00:03"
    },
    "watchDirectories": [ "Shared", "Test" ]
}

```

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.

```
{  
  "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

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.

| PROPERTY | DEFAULT | DESCRIPTION |
|-------------------------------------|----------|---|
| samplingSettings | n/a | See applicationInsights.samplingSettings . |
| 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. |
| httpAutoCollectionOptions | n/a | See applicationInsights.httpAutoCollectionOptions . |

| PROPERTY | DEFAULT | DESCRIPTION |
|-----------------------|---------|---|
| snapshotConfiguration | n/a | See applicationInsights.snapshotConfiguration . |

applicationInsights.samplingSettings

For more information about these settings, see [Sampling in Application Insights](#).

| 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. |
| maxSamplingPercentage | 100.0 | As sampling percentage varies, this property determines the maximum allowed sampling percentage. |

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------|---------|--|
| 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

| 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 is false, this flag applies to outgoing requests only, not incoming requests. |

| PROPERTY | DEFAULT | DESCRIPTION |
|-------------------------------|---------|---|
| 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.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](#).

| 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 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. |

| PROPERTY | DEFAULT | DESCRIPTION |
|-------------------------------|----------|--|
| 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 9999. |
| 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 problems with the Snapshot Debugger. It is 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. |

| PROPERTY | DEFAULT | DESCRIPTION |
|-----------------------------|---------|---|
| 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 is 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 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. |

| PROPERTY | DEFAULT | DESCRIPTION |
|---------------|---------|---|
| 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 System.Net.WebRequest.DefaultWebProxy 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. |

cosmosDb

Configuration setting can be found in [Cosmos DB triggers and bindings](#).

durableTask

Configuration setting can be found in [bindings for Durable Functions](#).

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](#).

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[1.*, 2.0.0)"
  }
}
```

The following properties are available in `extensionBundle`:

| PROPERTY | DESCRIPTION |
|----------|---|
| id | The namespace for Microsoft Azure Functions extension bundles. |
| version | The version of the bundle to install. The Functions runtime always picks the maximum permissible version defined by the version range or interval. The version value above allows all bundle versions from 1.0.0 up to but not including 2.0.0. 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.

```
{
  "functions": [ "QueueProcessor", "GitHubWebHook" ]
}
```

functionTimeout

Indicates the timeout duration for all functions. It follows the timespan string format.

| 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.

```
{
  "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 can be found in [http triggers and bindings](#).

logging

Controls the logging behaviors of the function app, including Application Insights.

```
"logging": {
    "fileLoggingMode": "debugOnly",
    "logLevel": {
        "Function.MyFunction": "Information",
        "default": "None"
    },
    "console": {
        ...
    },
    "applicationInsights": {
        ...
    }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|-----------------|-----------|--|
| fileLoggingMode | debugOnly | Defines what level of file logging is enabled. Options are <code>never</code> , <code>always</code> , <code>debugOnly</code> . |
| logLevel | n/a | Object that defines the log category filtering for functions in the app. Versions 2.x and later follow the ASP.NET Core layout for log category filtering. This setting lets you filter logging for specific functions. For more information, see Log filtering in the ASP.NET Core documentation. |

| PROPERTY | DEFAULT | DESCRIPTION |
|---------------------|---------|--|
| console | n/a | The console logging setting. |
| applicationInsights | n/a | The applicationInsights setting. |

console

This setting is a child of [logging](#). It controls the console logging when not in debugging mode.

```
{
  "logging": {
    ...
    "console": {
      "isEnabled": "false"
    },
    ...
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|-----------|---------|--------------------------------------|
| isEnabled | false | Enables or disables console logging. |

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.

```
{
  "managedDependency": {
    "enabled": true
  }
}
```

queues

Configuration settings can be found in [Storage queue triggers and bindings](#).

retry

Controls the [retry policy](#) options for all executions in the app.

```
{
  "retry": {
    "strategy": "fixedDelay",
    "maxRetryCount": 2,
    "delayInterval": "00:00:03"
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|-----------------|---------|--|
| strategy | null | Required. The retry strategy to use. Valid values are <code>fixedDelay</code> or <code>exponentialBackoff</code> . |
| maxRetryCount | null | Required. The maximum number of retries allowed per function execution. <code>-1</code> means to retry indefinitely. |
| delayInterval | null | The delay that's used between retries with a <code>fixedDelay</code> strategy. |
| minimumInterval | null | The minimum retry delay when using <code>exponentialBackoff</code> strategy. |
| maximumInterval | null | The maximum retry delay when using <code>exponentialBackoff</code> strategy. |

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](#).

```
{
  "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. |

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------------------|----------|--|
| 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. |

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.

```
{
  "watchDirectories": [ "Shared" ]
}
```

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:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "{storage-account-connection-string}",
    "FUNCTIONS_WORKER_RUNTIME": "{language-runtime}",
    "AzureFunctionsJobHost__logging__applicationInsights__samplingSettings__isEnabled": "false"
  }
}
```

Next steps

[Learn how to update the host.json file](#)

[See global settings in environment variables](#)

host.json reference for Azure Functions 1.x

12/4/2020 • 13 minutes to read • [Edit Online](#)

The *host.json* metadata file contains global configuration options that affect all functions for a function app. This article lists the settings that are available for the v1 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.

```
{
  "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](#).

```
{
    "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](#).

```
{  
    "applicationInsights": {  
        "sampling": {  
            "isEnabled": true,  
            "maxTelemetryItemsPerSecond" : 5  
        }  
    }  
}
```

| 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](#).

```
{  
    "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 Direct and Gateway |
| 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

```
{
  "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
    }
  }
}
```

Durable Functions 1.x

```
{
  "extensions": {
    "durableTask": {
      "hubName": "MyTaskHub",
      "controlQueueBatchSize": 32,
      "partitionCount": 4,
      "controlQueueVisibilityTimeout": "00:05:00",
      "workItemQueueVisibilityTimeout": "00:05:00",
      "maxConcurrentActivityFunctions": 10,
      "maxConcurrentOrchestratorFunctions": 10,
      "maxQueuePollingInterval": "00:00:30",
      "azureStorageConnectionStringName": "AzureWebJobsStorage",
      "trackingStoreConnectionStringName": "TrackingStorage",
      "trackingStoreNamePrefix": "DurableTask",
      "traceInputsAndOutputs": false,
      "logReplayEvents": false,
      "eventGridTopicEndpoint": "https://topic_name.westus2-1.eventgrid.azure.net/api/events",
      "eventGridKeySettingName": "EventGridKey",
      "eventGridPublishRetryCount": 3,
      "eventGridPublishRetryInterval": "00:00:30",
      "eventGridPublishEventTypes": ["Started", "Completed", "Failed", "Terminated"]
    }
  }
}
```

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 **DurableFunctionsHub**. For more information, see [Task hubs](#).

| PROPERTY | DEFAULT | DESCRIPTION |
|--------------------------------|---|--|
| hubName | DurableFunctionsHub | 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 | 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 | 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. |

| PROPERTY | DEFAULT | DESCRIPTION |
|------------------------------------|---|---|
| maxConcurrentOrchestratorFunctions | 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. |
| 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. |
| azureStorageConnectionStringName | AzureWebJobsStorage | The name of the app setting that has the Azure Storage connection string used to manage the underlying Azure Storage resources. |
| trackingStoreConnectionStringName | | The name of a connection string to use for the History and Instances tables. 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. |
| 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. |
| logReplayEvents | false | A value indicating whether to write orchestration replay events to Application Insights. |

| PROPERTY | DEFAULT | DESCRIPTION |
|-------------------------------|-----------|--|
| 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> . |
| 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 | true | 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. The default will be changed to <code>false</code> in a future release. |
| useGracefulShutdown | false | (Preview) Enable gracefully shutting down to reduce the chance of host shutdowns failing in-process function executions. |

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.

```
{  
    "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.

```
{  
    "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](#).

```
{
  "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/memor y/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. |
| 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.

```
{  
    "id": "9f4ea53c5136457d883d685e57164f08"  
}
```

logger

Controls filtering for logs written by an [ILogger](#) object or by `context.log`.

```
{  
    "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](#).

```
{
  "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. |
| 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](#)

```
{
  "sendGrid": {
    "from": "Contoso Group <admin@contoso.com>"
  }
}
```

| PROPERTY | DEFAULT | DESCRIPTION |
|----------|---------|--|
| from | n/a | The sender's email address across all functions. |

serviceBus

Configuration setting for Service Bus triggers and bindings.

```
{
  "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](#).

```
{
  "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].

```
{
  "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.

```
{  
  "watchDirectories": [ "Shared" ]  
}
```

Next steps

[Learn how to update the host.json file](#)

[See global settings in environment variables](#)

Frequently asked questions about networking in Azure Functions

12/4/2020 • 2 minutes to read • [Edit Online](#)

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 currently the only way to have a static inbound and outbound IP for your function. For details on using an App Service Environment, start with the article [Create and use an internal load balancer with an App Service Environment](#).

How do I restrict internet access to my function?

You can restrict internet access in a couple of ways:

- [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 an [private endpoints](#) with outbound virtual network integration or an App Service Environment.

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](#)

OpenAPI 2.0 metadata support in Azure Functions (preview)

11/2/2020 • 3 minutes to read • [Edit Online](#)

OpenAPI 2.0 (formerly Swagger) metadata support in Azure Functions is a preview feature that you can use to write an OpenAPI 2.0 definition inside a function app. You can then host that file by using the function app.

IMPORTANT

The OpenAPI preview feature is only available today in the 1.x runtime. Information on how to create a 1.x function app [can be found here](#).

[OpenAPI metadata](#) allows a function that's hosting a REST API to be consumed by a wide variety of other software. This software includes Microsoft offerings like PowerApps and the [API Apps feature of Azure App Service](#), third-party developer tools like [Postman](#), and [many more packages](#).

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: [C#, JavaScript, Java, or Python](#).
- [Azure Functions developer reference](#).
- Language-specific reference: [C#, C# script, F#, Java, JavaScript, or Python](#).
- [Azure Functions triggers and bindings concepts](#).
- [Code and test Azure Functions locally](#).

TIP

We recommend starting with the [getting started tutorial](#) and then returning to this document to learn more about specific features.

Enable OpenAPI definition support

You can configure all OpenAPI settings on the [API Definition](#) page in your function app's [Platform features](#).

NOTE

Function API definition feature is not supported for beta runtime currently.

To enable the generation of a hosted OpenAPI definition and a quickstart definition, set [API definition source](#) to [Function \(Preview\)](#). [External URL](#) allows your function to use an OpenAPI definition that's hosted elsewhere.

Generate a Swagger skeleton from your function's metadata

A template can help you start writing your first OpenAPI definition. The definition template feature creates a sparse OpenAPI definition by using all the metadata in the function.json file for each of your HTTP trigger functions. You'll need to fill in more information about your API from the [OpenAPI specification](#), such as request and response templates.

For step-by-step instructions, see the [getting started tutorial](#).

Available templates

| NAME | DESCRIPTION |
|----------------------|--|
| Generated Definition | An OpenAPI definition with the maximum amount of information that can be inferred from the function's existing metadata. |

Included metadata in the generated definition

The following table represents the Azure portal settings and corresponding data in `function.json` as it is mapped to the generated Swagger skeleton.

| SWAGGER.JSON | PORTAL UI | FUNCTION.JSON |
|--------------|---|-----------------------|
| Host | Function app settings > App Service settings > Overview > URL | <i>Not present</i> |
| Paths | Integrate > Selected HTTP methods | Bindings: Route |
| Path Item | Integrate > Route template | Bindings: Methods |
| Security | Keys | <i>Not present</i> |
| operationID* | Route + Allowed verbs | Route + Allowed Verbs |

*The operation ID is required only for integrating with PowerApps and Flow.

NOTE

The `x-ms-summary` extension provides a display name in Logic Apps, PowerApps, and Flow.

To learn more, see [Customize your Swagger definition for PowerApps](#).

Use CI/CD to set an API definition

You must enable API definition hosting in the portal before you enable source control to modify your API definition from source control. Follow these instructions:

1. Browse to [API Definition \(preview\)](#) in your function app settings.
 - a. Set **API definition source** to **Function**.
 - b. Click **Generate API definition template** and then **Save** to create a template definition for modifying later.
 - c. Note your API definition URL and key.
2. [Set up continuous integration/continuous deployment \(CI/CD\)](#).
3. Modify `swagger.json` in source control at `\site\wwwroot.azurefunctions\swagger\swagger.json`.

Now, changes to `swagger.json` in your repository are hosted by your function app at the API definition URL and key that you noted in step 1.c.

Next steps

- [Getting started tutorial](#). Try our walkthrough to see an OpenAPI definition in action.
- [Azure Functions GitHub repository](#). Check out the Functions repository to give us feedback on the API definition support preview. Make a GitHub issue for anything you want to see updated.
- [Azure Functions developer reference](#). Learn about coding functions and defining triggers and bindings.

Azure Functions pricing

12/4/2020 • 2 minutes to read • [Edit Online](#)

Azure Functions has three kinds of pricing plans. Choose the one that best fits your needs:

- **Consumption plan:** Azure provides all of the necessary computational resources. You don't have to worry about resource management, and only pay for the time that your code runs.
- **Premium plan:** You specify a number of pre-warmed instances that are always online and ready to immediately respond. When your function runs, Azure provides any additional computational resources that are needed. You pay for the pre-warmed instances running continuously and any additional instances you use as Azure scales your app in and out.
- **App Service plan:** Run your functions just like your web apps. If you use App Service for your other applications, your functions can run on the same plan at no additional cost.

For more information about hosting plans, see [Azure Functions hosting plan comparison](#). Full pricing details are available on the [Functions Pricing page](#).

Azure Functions scale and hosting

12/4/2020 • 19 minutes to read • [Edit Online](#)

When you create a function app in Azure, you must choose a hosting plan for your app. There are three basic hosting plans available for Azure Functions: [Consumption plan](#), [Premium plan](#), and [Dedicated \(App Service\) plan](#). All hosting plans are generally available (GA) on both Linux and Windows virtual machines.

The hosting plan you choose dictates the following behaviors:

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

Both Consumption and Premium plans automatically add compute power when your code is running. Your app is scaled out when needed to handle load, and scaled in when code stops running. For the Consumption plan, you also don't have to pay for idle VMs or reserve capacity in advance.

Premium plan provides additional features, such as premium compute instances, the ability to keep instances warm indefinitely, and VNet connectivity.

App Service plan allows you to take advantage of dedicated infrastructure, which you manage. Your function app doesn't scale based on events, which means it never scales in to zero. (Requires that [Always on](#) is enabled.)

For a detailed comparison between the various hosting plans (including Kubernetes-based hosting), see the [Hosting plans comparison section](#).

Consumption plan

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. This serverless plan scales automatically, and 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.

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](#).

The Consumption plan is the default hosting plan and offers the following benefits:

- Pay only when your functions are running
- Scale out automatically, even during periods of high load

Function apps in the same region can be assigned to the same Consumption plan. There's no downside or impact to having multiple apps running in the same Consumption plan. Assigning multiple apps to the same Consumption plan has no impact on resilience, scalability, or reliability of each app.

To learn more about how to estimate costs when running in a Consumption plan, see [Understanding Consumption plan costs](#).

Premium plan

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. Premium plan supports the following features:

- Perpetually warm instances to avoid any cold start
- VNet connectivity
- Unlimited execution duration (60 minutes guaranteed)
- Premium instance sizes (one core, two core, and four core instances)
- More predictable pricing
- High-density app allocation for plans with multiple function apps

To learn how you can create a function app in a Premium plan, see [Azure Functions Premium plan](#).

Instead of billing per execution and memory consumed, billing for the Premium plan is based on the number of core seconds and memory allocated across instances. There is no execution charge with the Premium plan. At least one instance must be allocated at all times per plan. This 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.

Consider the Azure Functions Premium plan in the following situations:

- Your function apps run continuously, or nearly continuously.
- You have a high number of small executions and have a high execution bill but low GB second bill in the Consumption plan.
- You need more CPU or memory options than what is provided by the Consumption plan.
- Your code needs to run longer than the [maximum execution time allowed](#) on the Consumption plan.
- You require features that are only available on a Premium plan, such as virtual network connectivity.

Dedicated (App Service) plan

Your function apps can also run on the same dedicated VMs as other App Service apps (Basic, Standard, Premium, and Isolated SKUs).

Consider an 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.

You pay the same for function apps in an App Service Plan as you would for other App Service resources, like web apps. For details about how the App Service plan works, see the [Azure App Service plans in-depth overview](#).

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](#).

When running JavaScript 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](#).

Running in an [App Service Environment](#) (ASE) lets you fully isolate your functions and take advantage of higher number of instances than an App Service Plan.

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. Always on is available only on an App Service plan. On a Consumption plan, the platform activates function apps automatically.

Function app timeout duration

The timeout duration of a function app is defined by the `functionTimeout` property in the [host.json](#) project file. The following table shows the default and maximum values in minutes for both plans and the different runtime versions:

| PLAN | RUNTIME VERSION | DEFAULT | MAXIMUM |
|-------------|-----------------|-----------|-----------|
| Consumption | 1.x | 5 | 10 |
| Consumption | 2.x | 5 | 10 |
| Consumption | 3.x | 5 | 10 |
| Premium | 1.x | 30 | Unlimited |
| Premium | 2.x | 30 | Unlimited |
| Premium | 3.x | 30 | Unlimited |
| App Service | 1.x | Unlimited | Unlimited |
| App Service | 2.x | 30 | Unlimited |
| App Service | 3.x | 30 | Unlimited |

NOTE

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](#).

Even with Always On enabled, the execution timeout for individual functions is controlled by the `functionTimeout` setting in the [host.json](#) project file.

Determine the hosting plan of an existing application

To determine the hosting plan used by your function app, see **App Service plan** in the **Overview** tab for the function app in the [Azure portal](#). To see the pricing tier, select the name of the **App Service Plan**, and then select **Properties** from the left pane.

The screenshot shows the Azure portal's Overview page for a function app. The 'App Service Plan' section is highlighted with a red box, showing 'ASP-myResourceGroup-a285 (Y1: 0)'. Other details include Resource group: myResourceGroup, Status: Running, Location: Central US, Subscription: Visual Studio Enterprise, Subscription ID: 11111111-1111-1111-1111-111111111111, Tags: Click here to add tags, and Metrics tab selected.

You can also use the Azure CLI to determine the plan, as follows:

```
appServicePlanId=$(az functionapp show --name <my_function_app_name> --resource-group <my_resource_group> --query appServicePlanId --output tsv)
az appservice plan list --query "[?id=='$appServicePlanId'].sku.tier" --output tsv
```

When the output from this command is `dynamic`, your function app is in the Consumption plan.
When the output from this command is `ElasticPremium`, your function app is in the Premium plan.
All other values indicate different tiers of an App Service plan.

Storage account requirements

On any plan, a function app requires a general Azure Storage account, which supports Azure Blob, Queue, Files, and Table storage. This is because Azure Functions relies on Azure Storage for operations such as managing triggers and logging function executions, but some storage accounts don't support queues and tables. These accounts, which include blob-only storage accounts (including premium storage) and general-purpose storage accounts with zone-redundant storage replication, are filtered-out from your existing **Storage Account** selections when you create a function app.

The same storage account used by your function app can also be used by your triggers and bindings to store your application data. However, for storage-intensive operations, you should use a separate storage account.

It's possible for multiple function apps to share the same storage account without any issues. (A good example of this is when you develop multiple apps in your local environment using the Azure Storage Emulator, which acts like one storage account.)

To learn more about storage account types, see [Introducing the Azure Storage services](#).

In Region Data Residency

When necessary for all customer data to 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 would also need to be used with [Azure Durable Functions](#) for Durable Functions.

Other platform-managed customer data will only be stored within the region when hosting in an Internal Load Balancer App Service Environment (or ILB ASE). Details can be found in [ASE zone redundancy](#).

How the Consumption and Premium plans work

In the Consumption and Premium plans, the Azure Functions infrastructure scales CPU and memory

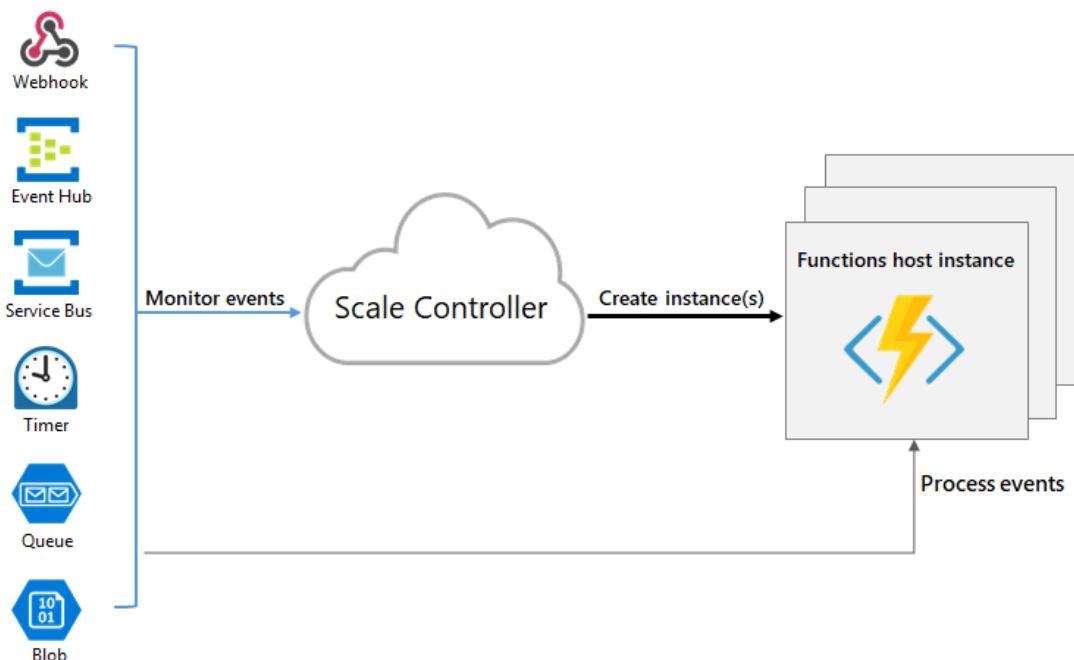
resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. Each instance of the Functions host in the Consumption plan is limited to 1.5 GB of memory and one CPU. An instance of the host is the entire function app, meaning all functions within a function app share resource within an instance and scale at the same time. Function apps that share the same Consumption plan are scaled independently. In the Premium plan, your plan size will determine the available memory and CPU for all apps in that plan on that instance.

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 cannot 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 scales based on the queue length and the age of the oldest queue message.

The unit of scale for Azure Functions is the function app. When the function app is scaled out, additional 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* to zero when no functions are running within a function app.



Cold Start

After your function app has been idle for a number of minutes, the platform may 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 that must be loaded by your function app can impact 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 running in a Premium plan or in a Dedicated plan with Always on enabled.

Understanding scaling behaviors

Scaling can vary on a number of factors, and scale differently based on the trigger and language selected. There are a few intricacies of scaling behaviors to be aware of:

- A single function app only scales out to a maximum of 200 instances. A single instance may

process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions. You can [specify a lower maximum](#) to throttle scale as required.

- 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](#).
- For Service Bus triggers, use *Manage* rights on resources for the most efficient scaling. With *Listen* rights, scaling isn't as accurate because the queue 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](#).
- For Event Hub triggers, see the [scaling guidance](#) in the reference article.

Limit scale out

You may wish to restrict the number of instances an app scales out to. This is most common for cases where a downstream component like a database has limited throughput. By default, consumption plan functions will scale out to as many as 200 instances, and premium plan functions will scale out to as many as 100 instances. You can specify a lower maximum for a specific app by modifying the `functionAppScaleLimit` value. The `functionAppScaleLimit` can be set to 0 or null for unrestricted, or a valid value between 1 and the app maximum.

```
az resource update --resource-type Microsoft.Web/sites -g <resource_group> -n <function_app_name>/config/web --set properties.functionAppScaleLimit=<scale_limit>
```

Best practices and patterns for scalable apps

There are many aspects of a function app that will impact how well it will scale, 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](#).

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](#).

Billing model

Billing for the different plans is described in detail on the [Azure Functions pricing page](#). Usage is aggregated at the function app level and counts only the time that function code is executed. The following are units for billing:

- **Resource consumption in gigabyte-seconds (GB-s)**. Computed as a combination of memory size and execution time for all functions within a function app.
- **Executions**. Counted each time a function is executed in response to an event trigger.

Useful queries and information on how to understand your consumption bill can be found [on the billing FAQ](#).

Hosting plans comparison

The following comparison table shows all important aspects to help the decision of Azure Functions App hosting plan choice:

Plan summary

| | |
|-----------------------------------|---|
| Consumption plan | Scale automatically and only pay for compute resources when your functions are running. On the Consumption plan, instances of the Functions host are dynamically added and removed based on the number of incoming events. <ul style="list-style-type: none"> ✓ Default hosting plan. ✓ Pay only when your functions are running. ✓ scale-out automatically, even during periods of high load. |
| Premium plan | While automatically scaling based on demand, use pre-warmed workers to run applications with no delay after being idle, run on more powerful instances, and connect to VNETs. Consider the Azure Functions Premium plan in the following situations, in addition to all features of the App Service plan: <ul style="list-style-type: none"> ✓ Your function apps run continuously, or nearly continuously. ✓ You have a high number of small executions and have a high execution bill but low GB second bill in the Consumption plan. ✓ You need more CPU or memory options than what is provided by the Consumption plan. ✓ Your code needs to run longer than the maximum execution time allowed on the Consumption plan. ✓ You require features that are only available on a Premium plan, such as virtual network connectivity. |
| Dedicated plan¹ | Run your functions within an App Service plan at regular App Service plan rates. Good fit for long running operations, as well as when more predictive scaling and costs are required. Consider an App Service plan in the following situations: <ul style="list-style-type: none"> ✓ 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. |
| ASE¹ | App Service Environment (ASE) is an App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale. ASEs are appropriate for application workloads that require: <ul style="list-style-type: none"> ✓ Very high scale. ✓ Full compute isolation and secure network access. ✓ High memory utilization. |
| Kubernetes | Kubernetes provides a fully isolated and dedicated environment running on top of the Kubernetes platform. Kubernetes is appropriate for application workloads that require: <ul style="list-style-type: none"> ✓ Custom hardware requirements. ✓ Isolation and secure network access. ✓ Ability to run in hybrid or multi-cloud environment. ✓ Run alongside existing Kubernetes applications and services. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

| | LINUX ¹ CODE-ONLY | WINDOWS ² CODE-ONLY | LINUX ^{1,3} DOCKER CONTAINER |
|-----------------------------------|--|---|---|
| Consumption plan | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | No support |
| Premium plan | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | .NET Core Node.js Java PowerShell Core Python |
| Dedicated plan⁴ | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | .NET Core Node.js Java PowerShell Core Python |
| ASE⁴ | .NET Core Node.js Java Python | .NET Core Node.js Java PowerShell Core | .NET Core Node.js Java PowerShell Core Python |
| Kubernetes | n/a | n/a | .NET Core Node.js Java PowerShell Core Python |

¹Linux is the only supported operating system for the Python runtime stack.

²Windows is the only supported operating system for the PowerShell runtime stack.

³Linux is the only supported operating system for Docker containers. ⁴ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Scale

| | SCALE OUT | MAX # INSTANCES |
|-------------------------|--|-----------------|
| Consumption plan | Event driven. Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. | 200 |
| Premium plan | Event driven. Scale out automatically, even during periods of high load. Azure Functions infrastructure scales CPU and memory resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. | 100 |

| | SCALE OUT | MAX # INSTANCES |
|-----------------------------|--|--------------------|
| Dedicated plan ¹ | Manual/autoscale | 10-20 |
| ASE ¹ | Manual/autoscale | 100 |
| Kubernetes | Event-driven autoscale for Kubernetes clusters using KEDA. | Varies by cluster. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Cold start behavior

| | |
|-----------------------------|--|
| Consumption plan | Apps may scale to zero if idle for a period of time, meaning some requests may have additional latency at startup. The consumption plan does have some optimizations to help decrease cold start time, including pulling from pre-warmed placeholder functions that already have the function host and language processes running. |
| Premium plan | Perpetually warm instances to avoid any cold start. |
| Dedicated plan ¹ | When running in a Dedicated plan, the Functions host can run continuously, which means that cold start isn't really an issue. |
| ASE ¹ | When running in a Dedicated plan, the Functions host can run continuously, which means that cold start isn't really an issue. |
| Kubernetes | Depends on KEDA configuration. Apps can be configured to always run and never have cold start, or configured to scale to zero, which results in cold start on new events. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Service limits

| RESOURCE | CONSUMPTION PLAN | PREMIUM PLAN | DEDICATED PLAN | ASE | KUBERNETES |
|---|-------------------------|------------------------|------------------------|-----------|--------------------|
| Default timeout duration (min) | 5 | 30 | 30 ¹ | 30 | 30 |
| Max timeout duration (min) | 10 | unbounded ⁷ | unbounded ² | unbounded | unbounded |
| Max outbound connections (per instance) | 600 active (1200 total) | unbounded | unbounded | unbounded | unbounded |
| Max request size (MB) ³ | 100 | 100 | 100 | 100 | Depends on cluster |

| RESOURCE | CONSUMPTION PLAN | PREMIUM PLAN | DEDICATED PLAN | ASE | KUBERNETES |
|---|---------------------------------------|---|---|---|-----------------------------|
| Max query string length ³ | 4096 | 4096 | 4096 | 4096 | Depends on cluster |
| Max request URL length ³ | 8192 | 8192 | 8192 | 8192 | Depends on cluster |
| ACU per instance | 100 | 210-840 | 100-840 | 210-250 ⁸ | AKS pricing |
| Max memory (GB per instance) | 1.5 | 3.5-14 | 1.75-14 | 3.5 - 14 | Any node is supported |
| Function apps per plan | 100 | 100 | unbounded ⁴ | unbounded | unbounded |
| App Service plans | 100 per region | 100 per resource group | 100 per resource group | - | - |
| Storage ⁵ | 5 TB | 250 GB | 50-1000 GB | 1 TB | n/a |
| Custom domains per app | 500 ⁶ | 500 | 500 | 500 | n/a |
| 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 | n/a |

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. Consumption plan uses Azure Files for temporary storage.

⁶ 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.

⁷ Guaranteed for up to 60 minutes.

⁸ 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.

Networking features

| FEATURE | CONSUMPTION PLAN | PREMIUM PLAN | DEDICATED PLAN | ASE | KUBERNETES |
|---|---|---|---|---|---|
| Inbound IP restrictions and private site access | <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 |
| Virtual network integration | <input type="checkbox"/> No | <input checked="" type="checkbox"/> Yes (Regional) | <input checked="" type="checkbox"/> Yes (Regional and Gateway) | <input checked="" type="checkbox"/> Yes | <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 |
| Hybrid connections (Windows only) | <input type="checkbox"/> No | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes |
| Outbound IP restrictions | <input type="checkbox"/> No | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes |

Billing

| | |
|-----------------------------------|--|
| Consumption plan | Pay only for the time your functions run. Billing is based on number of executions, execution time, and memory used. |
| Premium plan | Premium plan is based on the number of core seconds and memory used across needed and pre-warmed instances. At least one instance per plan must be kept warm at all times. This plan provides more predictable pricing. |
| Dedicated plan¹ | You pay the same for function apps in an App Service Plan as you would for other App Service resources, like web apps. |
| ASE¹ | there's a flat monthly rate for an ASE that pays for the infrastructure and doesn't change with the size of the ASE. In addition, there's a cost per App Service plan vCPU. All apps hosted in an ASE are in the Isolated pricing SKU. |
| Kubernetes | You pay only the costs of your Kubernetes cluster; no additional billing for Functions. Your function app runs as an application workload on top of your cluster, just like a regular app. |

¹ For specific limits for the various App Service plan options, see the [App Service plan limits](#).

Next steps

- [Quickstart: Create an Azure Functions project using Visual Studio Code](#)

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