

Contents

[Functions Documentation](#)

[Overview](#)

[About Azure Functions](#)

[Durable Functions](#)

[Serverless comparison](#)

[Quickstarts](#)

[Create function - Visual Studio Code](#)

[Create function - Visual Studio](#)

[Create function - Java/Maven](#)

[Create function - Python](#)

[Create function - PowerShell](#)

[Create function - Azure CLI](#)

[Create function - portal](#)

[Create function - Linux](#)

[Triggers](#)

[Azure Cosmos DB](#)

[Blob storage](#)

[Queue storage](#)

[Timer](#)

[Integrate](#)

[Azure Cosmos DB](#)

[Storage - Python](#)

[Storage - portal](#)

[Storage - Visual Studio Code](#)

[Storage - Visual Studio](#)

[Tutorials](#)

[Functions with Logic Apps](#)

[Create a serverless API](#)

[Develop Python functions with VS Code](#)

- [Create an OpenAPI definition](#)
- [Connect to a Virtual Network](#)
- [Image resize with Event Grid](#)
- [Create a serverless web app](#)
- [Create a custom Linux image](#)
- [Functions on IoT Edge device](#)
- [Samples](#)
 - [Azure Serverless Community Library](#)
 - [Azure Samples](#)
 - [Azure CLI](#)
- [Concepts](#)
 - [Compare versions 1.x and 2.x](#)
 - [Scale and hosting](#)
 - [Premium plan](#)
 - [Deployments](#)
 - [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](#)
 - [Diagnostics](#)

[Performance considerations](#)

[Functions Proxies](#)

[Networking options](#)

[IP addresses](#)

[Functions in Kubernetes](#)

[How-to guides](#)

[Develop](#)

[Developer guide](#)

[Testing functions](#)

[Develop and debug locally](#)

[Visual Studio Code development](#)

[Visual Studio development](#)

[Core Tools development](#)

[IntelliJ IDEA development](#)

[Eclipse development](#)

[Create Linux function app](#)

[Debug local PowerShell functions](#)

[Dependency injection](#)

[Manage connections](#)

[Error handling](#)

[Manually run a non HTTP-triggered function](#)

[Debug Event Grid trigger locally](#)

[Azure for Students Starter](#)

[Deploy](#)

[Continuous deployment](#)

[Build and deploy using Azure Pipelines](#)

[Zip deployment](#)

[Run from package](#)

[Automate resource deployment](#)

[On-premises functions](#)

[Deploy using the Jenkins plugin](#)

[Configure](#)

- [Manage a function app](#)
- [Set the runtime version](#)
- [Manually register an extension](#)
- [Disable a function](#)
- [Monitor](#)
- [Secure](#)
 - [Buy SSL cert](#)
 - [Authenticate users
 - \[Authenticate with Azure AD\]\(#\)
 - \[Authenticate with Facebook\]\(#\)
 - \[Authenticate with Google\]\(#\)
 - \[Authenticate with Microsoft account\]\(#\)
 - \[Authenticate with Twitter\]\(#\)](#)
 - [Advanced auth](#)
 - [Restrict IPs](#)
 - [Use a managed identity](#)
 - [Reference secrets from Key Vault](#)
- [Integrate](#)
 - [Connect to SQL Database](#)
 - [Connect to a virtual Network](#)
 - [Create an Open API 2.0 definition](#)
 - [Export to PowerApps and Microsoft Flow](#)
 - [Call a function from PowerApps](#)
 - [Call a function from Microsoft Flow](#)
 - [Use a managed identity](#)
- [Troubleshoot](#)
 - [Troubleshoot storage](#)
- [Reference](#)
 - [API references](#)
 - [Java](#)
 - [Python](#)
 - [App settings reference](#)

Bindings

[Blob storage](#)

[Azure Cosmos DB](#)

[Functions 1.x](#)

[Functions 2.x](#)

[Event Grid](#)

[Event Hubs](#)

[IoT Hub](#)

[HTTP and webhooks](#)

[Microsoft Graph](#)

[Mobile Apps](#)

[Notification Hubs](#)

[Queue storage](#)

[SendGrid](#)

[Service Bus](#)

[SignalR Service](#)

[Table storage](#)

[Timer](#)

[Twilio](#)

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

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

[Networking FAQ](#)

[OpenAPI reference](#)

Resources

[Build your skills with Microsoft Learn](#)

[Azure Roadmap](#)

[Pricing](#)

[Pricing calculator](#)

[Quota information](#)

[Regional availability](#)

[Videos](#)

[MSDN forum](#)

[Stack Overflow](#)

[Twitter](#)

[Provide product feedback](#)

[Azure Functions GitHub repository](#)

[Service updates](#)

An introduction to Azure Functions

7/19/2019 • 4 minutes to read • [Edit Online](#)

Azure Functions is a solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development language of choice, such as C#, Java, JavaScript, Python, or PHP. Pay only for the time your code runs and trust Azure to scale as needed. Azure Functions lets you develop [serverless](#) applications on Microsoft Azure.

This topic provides a high-level overview of Azure Functions. If you want to jump right in and get started with Functions, start with [Create your first Azure Function](#). If you are looking for more technical information about Functions, see the [developer reference](#).

Features

Here are some key features of Functions:

- **Choice of language** - Write functions using your choice of C#, Java, Javascript, Python, and other languages. See [Supported languages](#) for the complete list.
- **Pay-per-use pricing model** - Pay only for the time spent running your code. See the Consumption hosting plan option in the [pricing section](#).
- **Bring your own dependencies** - Functions supports NuGet and NPM, so you can use your favorite libraries.
- **Integrated security** - Protect HTTP-triggered functions with OAuth providers such as Azure Active Directory, Facebook, Google, Twitter, and Microsoft Account.
- **Simplified integration** - Easily leverage Azure services and software-as-a-service (SaaS) offerings. See the [integrations section](#) for some examples.
- **Flexible development** - Code your functions right in the portal or set up continuous integration and deploy your code through [GitHub](#), [Azure DevOps Services](#), and other [supported development tools](#).
- **Open-source** - The Functions runtime is open-source and [available on GitHub](#).

What can I do with Functions?

Functions is a great solution for processing data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and microservices. Consider Functions for tasks like image or order processing, file maintenance, or for any tasks that you want to run on a schedule.

Functions provides templates to get you started with key scenarios, including the following:

- **HTTPTrigger** - Trigger the execution of your code by using an HTTP request. For an example, see [Create your first function](#).
- **TimerTrigger** - Execute cleanup or other batch tasks on a predefined schedule. For an example, see [Create a function triggered by a timer](#).
- **CosmosDBTrigger** - Process Azure Cosmos DB documents when they are added or updated in collections in a NoSQL database. For more information, see [Azure Cosmos DB bindings](#).
- **BlobTrigger** - Process Azure Storage blobs when they are added to containers. You might use this function for image resizing. For more information, see [Blob storage bindings](#).
- **QueueTrigger** - Respond to messages as they arrive in an Azure Storage queue. For more information, see [Azure Queue storage bindings](#).
- **EventGridTrigger** - Respond to events delivered to a subscription in Azure Event Grid. Supports a

subscription-based model for receiving events, which includes filtering. A good solution for building event-based architectures. For an example, see [Automate resizing uploaded images using Event Grid](#).

- **EventHubTrigger** - Respond to events delivered to an Azure Event Hub. Particularly useful in application instrumentation, user experience or workflow processing, and internet-of-things (IoT) scenarios. For more information, see [Event Hubs bindings](#).
- **ServiceBusQueueTrigger** - Connect your code to other Azure services or on-premises services by listening to message queues. For more information, see [Service Bus bindings](#).
- **ServiceBusTopicTrigger** - Connect your code to other Azure services or on-premises services by subscribing to topics. For more information, see [Service Bus bindings](#).

Azure Functions supports *triggers*, which are ways to start execution of your code, and *bindings*, which are ways to simplify coding for input and output data. For a detailed description of the triggers and bindings that Azure Functions provides, see [Azure Functions triggers and bindings developer reference](#).

Integrations

Azure Functions integrates with various Azure and 3rd-party services. These services can trigger your function and start execution, or they can serve as input and output for your code. The following service integrations are supported by Azure Functions:

- Azure Cosmos DB
- Azure Event Hubs
- Azure Event Grid
- Azure Notification Hubs
- Azure Service Bus (queues and topics)
- Azure Storage (blob, queues, and tables)
- On-premises (using Service Bus)
- Twilio (SMS messages)

How much does Functions cost?

Azure Functions has two kinds of pricing plans. Choose the one that best fits your needs:

- **Consumption plan** - When your function runs, Azure provides all of the necessary computational resources. You don't have to worry about resource management, and you only pay for the time that your code runs.
- **App Service plan** - Run your functions just like your web apps. When you are already using App Service for your other applications, you can run your functions 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](#).

Next Steps

- [Create your first Azure Function](#)
Jump right in and create your first function using the Azure Functions quickstart.
- [Azure Functions developer reference](#)
Provides more technical information about the Azure Functions runtime and a reference for coding functions and defining triggers and bindings.
- [Testing Azure Functions](#)
Describes various tools and techniques for testing your functions.
- [How to scale Azure Functions](#)
Discusses service plans available with Azure Functions, including the Consumption hosting plan, and how to choose the right plan.

- [Learn more about Azure App Service](#)

Azure Functions leverages Azure App Service for core functionality like deployments, environment variables, and diagnostics.

What are Durable Functions?

7/7/2019 • 2 minutes to read • [Edit Online](#)

Durable Functions are an extension of [Azure Functions](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

Benefits

The extension lets you define stateful workflows using an *orchestrator function*, which can provide the following benefits:

- You can define your workflows in code. No JSON schemas or designers are needed.
- Other functions can be called both synchronously and asynchronously. Output from called functions can be saved to local variables.
- Progress is automatically checkpointed when the function awaits. Local state is never lost when the process recycles or the VM reboots.

Application patterns

The primary use case for Durable Functions is simplifying complex, stateful coordination requirements in serverless applications. The following are some typical application patterns that can benefit from Durable Functions:

- [Chaining](#)
- [Fan-out/fan-in](#)
- [Async HTTP APIs](#)
- [Monitoring](#)
- [Human interaction](#)

Supported languages

Durable Functions currently supports the following languages:

- **C#**: both [precompiled class libraries](#) and [C# script](#).
- **F#**: precompiled class libraries and F# script. F# script is only supported for version 1.x of the Azure Functions runtime.
- **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.

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](#).

Billing

Durable Functions are billed the same as Azure Functions. For more information, see [Azure Functions pricing](#).

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](#)

In both 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:

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

Next steps

[Durable Functions patterns and technical concepts](#)

What are Microsoft Flow, Logic Apps, Functions, and WebJobs?

7/19/2019 • 6 minutes to read • [Edit Online](#)

This article compares the following Microsoft cloud services:

- [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.

Compare Microsoft Flow and Azure Logic Apps

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

Microsoft Flow is built on top of Logic Apps. They share the same workflow designer and the same [connectors](#).

Microsoft Flow 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 Microsoft Flow or Logic Apps is best for a particular integration:

	MICROSOFT FLOW	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 Microsoft Flow environments and data loss prevention (DLP) policies, track licensing: Microsoft Flow Admin Center	Manage resource groups, connections, access management, and logging: Azure portal

	MICROSOFT FLOW	LOGIC APPS
Security	Office 365 Security and Compliance 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 services:

	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 (preview) Python (preview)	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.

Microsoft Flow, 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 Microsoft Flow, 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](#)
- [Microsoft Flow frequently asked questions](#)

Next steps

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

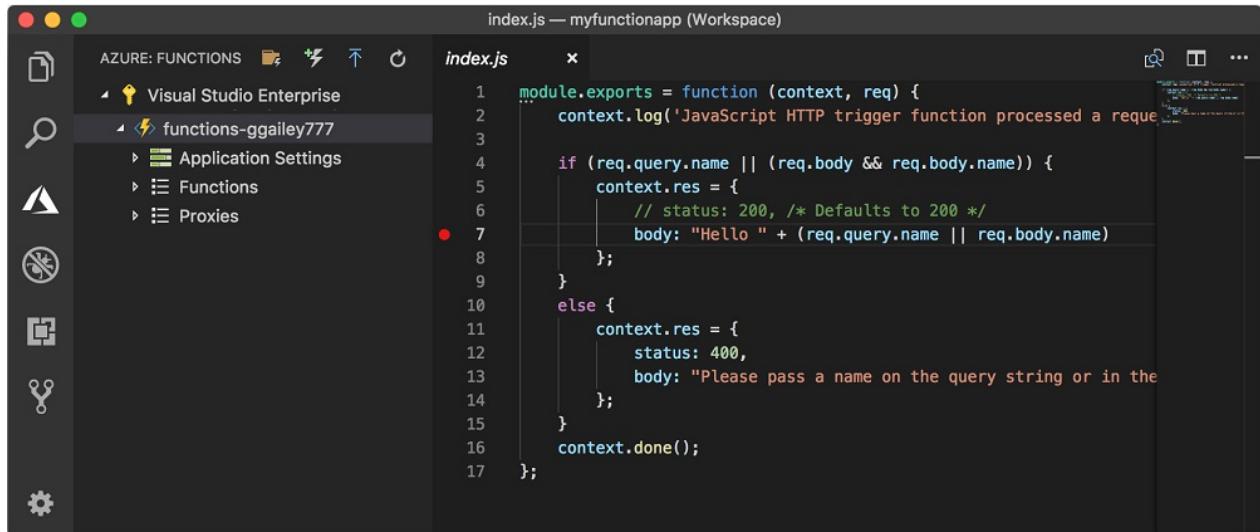
- [Get started with Microsoft Flow](#)
- [Create a logic app](#)
- [Create your first Azure function](#)

Create your first function using Visual Studio Code

6/26/2019 • 6 minutes to read • [Edit Online](#)

Azure Functions lets you execute your code in a [serverless](#) environment without having to first create a VM or publish a web application.

In this article, you learn how to use the [Azure Functions extension for Visual Studio Code](#) to create and test a "hello world" function on your local computer using Microsoft Visual Studio Code. You then publish the function code to Azure from Visual Studio Code.



The extension currently supports C#, JavaScript, and Java functions, with Python support currently in Preview. The steps in this article and the article that follows support only JavaScript and C# functions. To learn how to use Visual Studio Code to create and publish Python functions, see [Deploy Python to Azure Functions](#). To learn how to use Visual Studio Code to create and publish PowerShell functions, see [Create your first PowerShell function in Azure](#).

The extension is currently in preview. To learn more, see the [Azure Functions extension for Visual Studio Code](#) extension page.

Prerequisites

To complete this quickstart:

- Install [Visual Studio Code](#) on one of the [supported platforms](#).
- Install version 2.x of the [Azure Functions Core Tools](#).
- Install the specific requirements for your chosen language:

LANGUAGE	REQUIREMENT
C#	C# extension
JavaScript	Node.js*

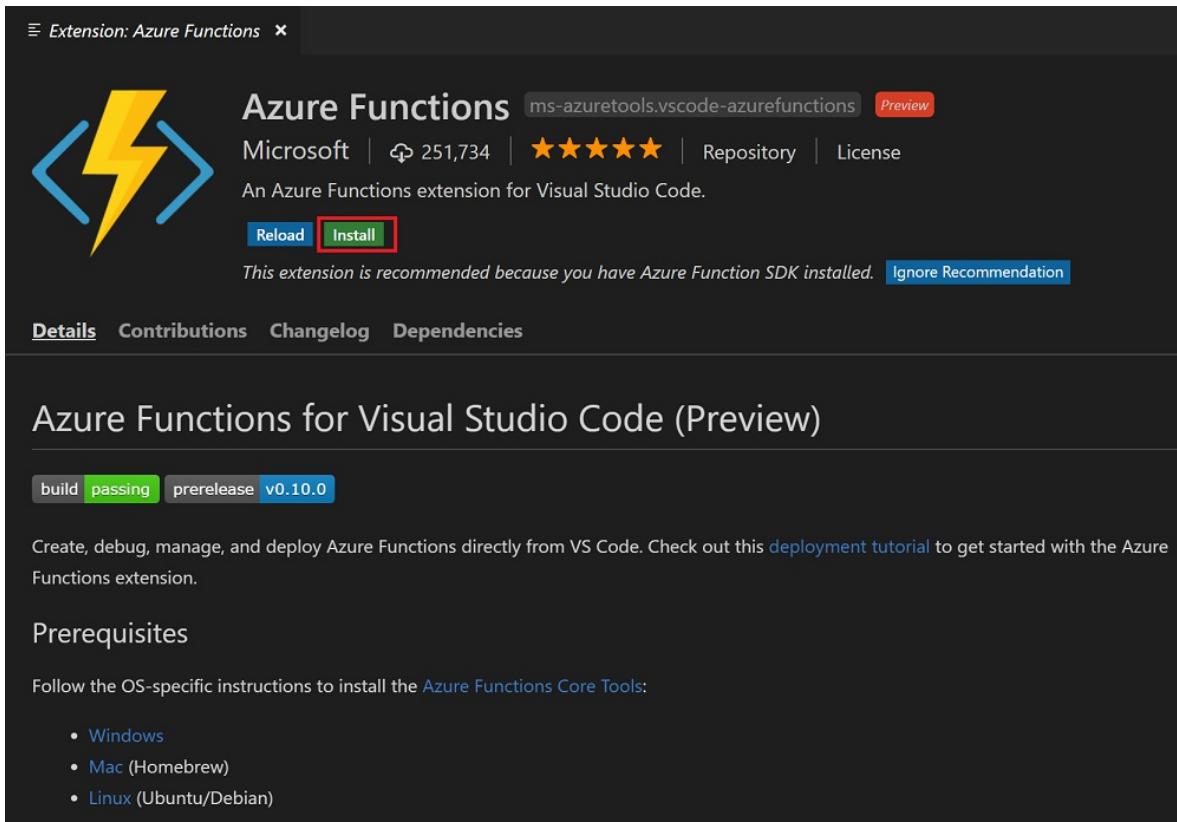
*Active LTS and Maintenance LTS versions (8.11.1 and 10.14.1 recommended).

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

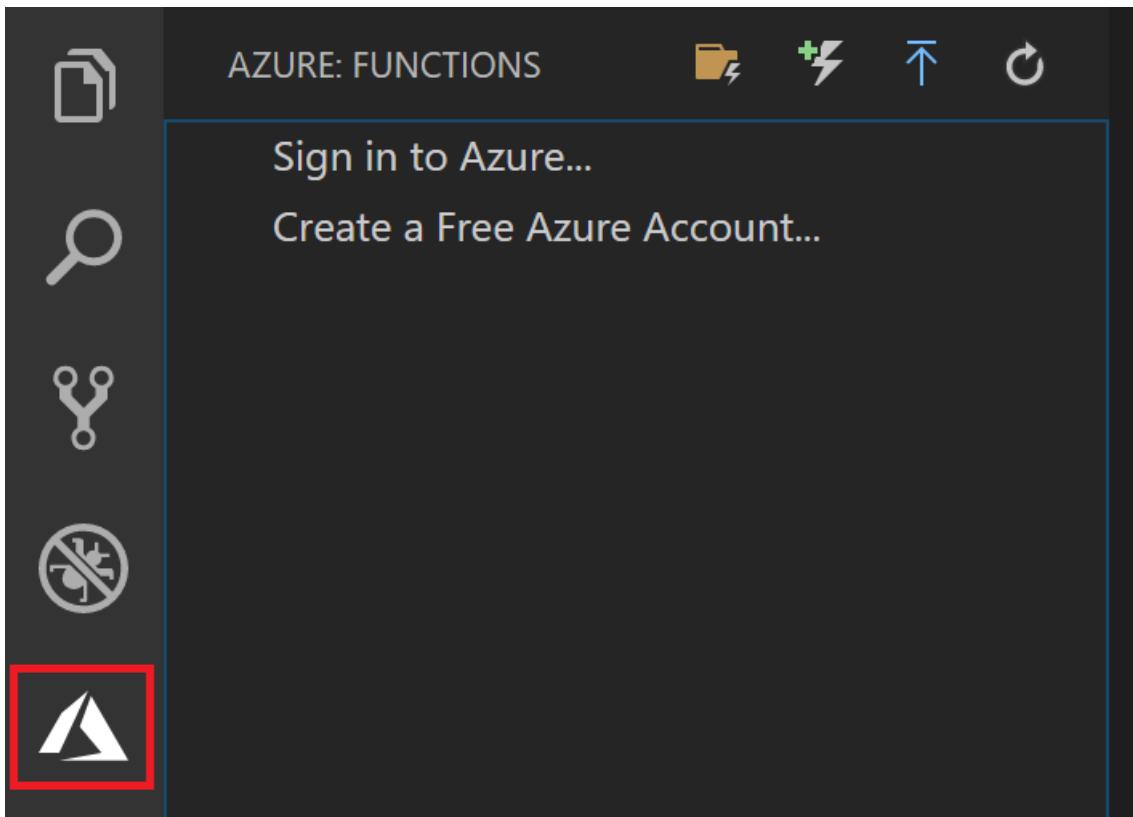
Install the Azure Functions extension

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](#).
2. Select **Install** to install the extension for Visual Studio Code:



3. Restart Visual Studio Code and select the Azure icon on the Activity bar. You should see an Azure Functions area in the Side Bar.



Create your Functions project with a function

The Azure Functions project template in Visual Studio Code creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for management, deployment, and sharing of resources.

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Create new project...`.
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. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a language for your function app project	C# or JavaScript	This article supports C# and JavaScript. For Python, see this Python article , and for PowerShell, see this PowerShell article .
Select a template for your project's first function	HTTP trigger	Create an HTTP triggered function in the new function app.
Provide a function name	HttpTrigger	Press Enter to use the default name.
Provide a namespace	My.Functions	(C# only) C# class libraries must have a namespace.
Authorization level	Function	Requires a function key to call the function's HTTP endpoint.
Select how you would like to open your project	Add to workspace	Creates the function app in the current workspace.

Visual Studio Code creates the function app project in a new workspace. This project contains the `host.json` and `local.settings.json` configuration files, plus any language-specific project files.

A new HTTP triggered function is also created in the `HttpTrigger` folder of the function app project.

Run the function locally

Azure Functions Core Tools lets you run an Azure Functions project on your local development computer.

1. To test your function, set a breakpoint in the function code and 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. This URL includes the function key, which is passed to the `code` query parameter.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2: Task - runFunctionsH + ⌂ ⌄ ⌁ ⌈ ⌉ ×
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

HttpStart: [POST] http://localhost:7071/api/orchestrators/{functionName}
```

3. Paste the URL for the HTTP request into your browser's address bar. Append the query string `?name=<yourname>` to this URL and execute the request. Execution is paused when the breakpoint is hit.
4. When you continue the execution, the following shows the response in the browser to the GET request:



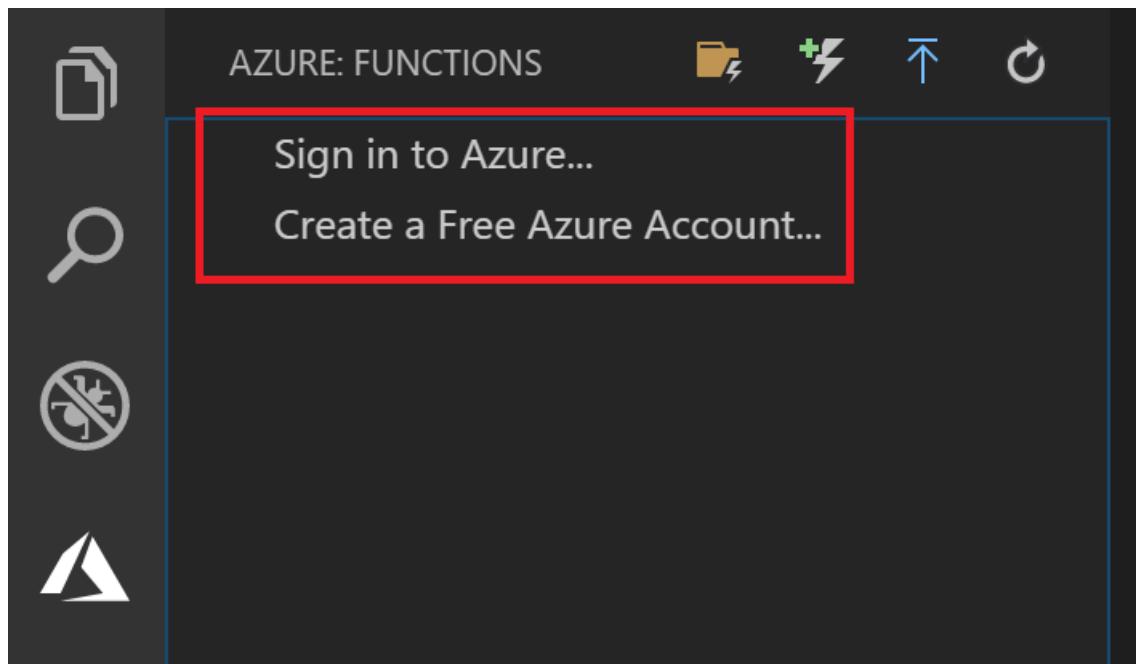
5. To stop debugging, press Shift + F5.

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

Sign in to Azure

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

1. In the **Azure: Functions** area, choose **Sign in to Azure...**. If you don't already have one, you can **Create a free Azure account**.



2. When prompted, select **Copy & Open**, or copy the displayed code and open <https://aka.ms/devicelogin> in your browser.

3. Paste the copied code in the **Device Login** page, verify the sign in for Visual Studio Code, then select **Continue**.
4. Complete the sign in using your Azure account credentials. After you have successfully signed in, you can close the browser.

Publish the project 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.

By default, Visual Studio creates all of the Azure resources required to create your function app. The names of these resources are based on the function app name you choose. If you need to have full control of the created resources, you can instead [publish using advanced options](#).

This section assumes that you are creating a new function app in Azure.

IMPORTANT

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

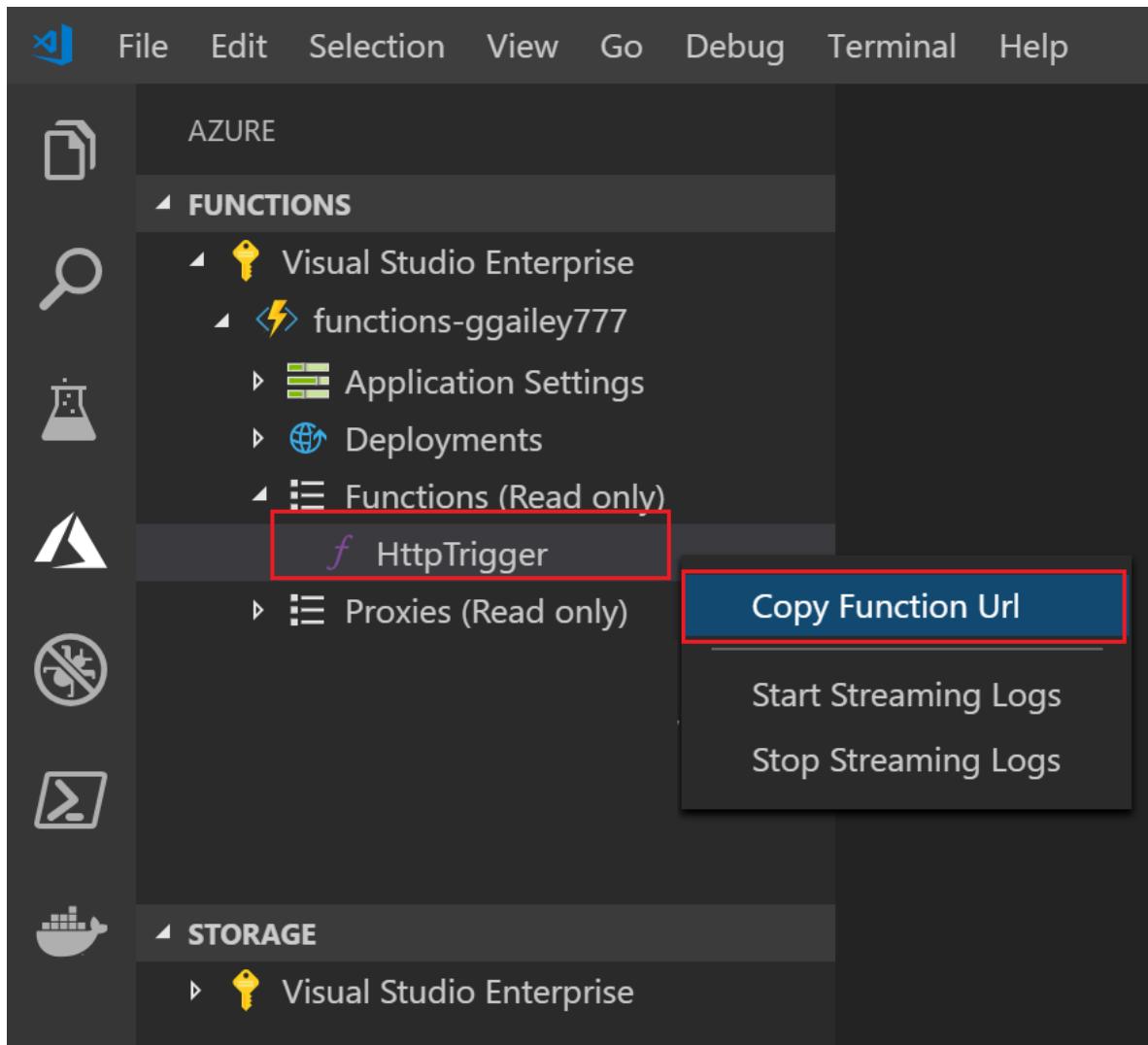
1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Deploy to function app...`.
2. If not signed-in, you are prompted to **Sign in to Azure**. You can also [Create a free Azure account](#). After successful sign in from the browser, go back to Visual Studio Code.
3. If you have multiple subscriptions, **Select a subscription** for the function app, then choose **+ Create New Function App in Azure**.
4. Type a globally unique name that identifies your function app and press Enter. Valid characters for a function app name are `a-z`, `0-9`, and `-`.

When you press Enter, the following Azure resources are created in your subscription:

- **Resource group**: Contains all of the created Azure resources. The name is based on your function app name.
- **Storage account**: A standard Storage account is created with a unique name that is based on your function app name.
- **Hosting plan**: A consumption plan is created in the West US region to host your serverless function app.
- **Function app**: Your project is deployed to and runs in this new function app.

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.

5. Back in the **Azure: Functions** area, expand the new function app under your subscription. Expand **Functions**, right-click **HttpTrigger**, and then choose **Copy function URL**.



Run the function in Azure

1. Copy the URL of the HTTP trigger from the **Output** panel. As before, make sure to add the query string `?name=<yourusername>` to the end of this URL and execute the request.

The URL that calls your HTTP-triggered function should be in the following format:

```
http://<functionappname>.azurewebsites.net/api/<functionname>?name=<yourname>
```

2. Paste this new URL for the HTTP request into your browser's address bar. The following shows the response in the browser to the remote GET request returned by the function:



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. The next article also shows you how to clean up these new Azure resources by removing the resource group you created.

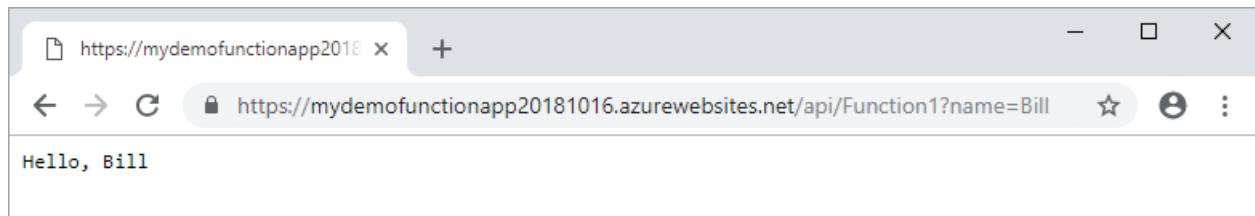
[Add an Azure Storage queue binding to your function](#)

Create your first function using Visual Studio

7/28/2019 • 6 minutes to read • [Edit Online](#)

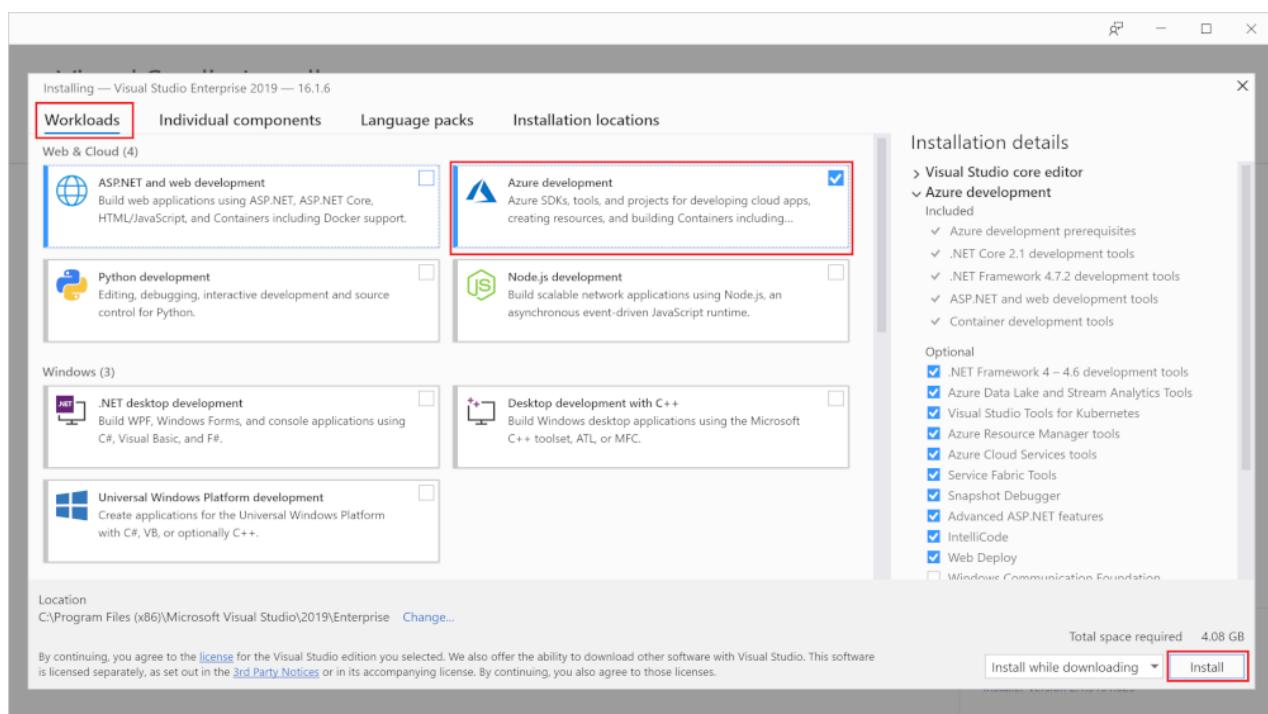
Azure Functions lets you execute your code in a [serverless](#) environment without having to first create a VM or publish a web application.

In this article, you learn how to use Visual Studio 2019 to locally create and test a "hello world" function and then publish it to Azure. This quickstart is designed for Visual Studio 2019. When creating a Functions project using Visual Studio 2017, you must first install the [latest Azure Functions tools](#).



Prerequisites

To complete this tutorial, you must first install [Visual Studio 2019](#). Make sure that the **Azure development** workload is also installed.



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 can be published to a function app in Azure. You can use a function app to group functions as a logical unit for management, deployment, and sharing of resources.

1. In Visual Studio, on the **File** menu, select **New > Project**.
2. In the **Create a new project** dialog box, search for `functions`, choose the **Azure Functions** template, and

select **Next**.

3. Enter a name for your project, and 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. In **Create a new Azure Functions application**, use the following options:

- **Azure Functions v2 (.NET Core)**
- **HTTP trigger**
- **Storage Account: Storage Emulator**
- **Authorization level: Anonymous**

OPTION	SUGGESTED VALUE	DESCRIPTION
Functions runtime	Azure Functions 2.x (.NET Core)	This setting creates a function project that uses the version 2.x runtime of Azure Functions, which supports .NET Core. Azure Functions 1.x supports the .NET Framework. For more information, see Target Azure Functions runtime version .
Function template	HTTP trigger	This setting creates a function triggered by an HTTP request.
Storage Account	Storage Emulator	An HTTP trigger doesn't use the Azure Storage account connection. All other trigger types require a valid Storage account connection string. Because Functions requires a storage account, one is assigned or created when you publish your project to Azure.
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 in the HTTP and webhook bindings .

NOTE

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-triggered function.

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

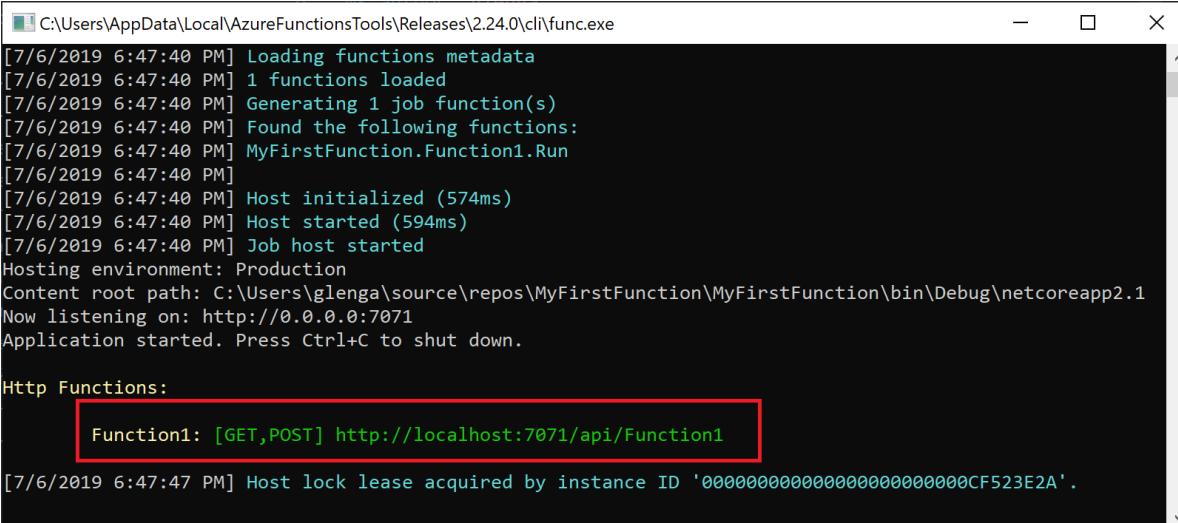
You can expand the capabilities of your function using input and output bindings by applying the appropriate attributes to the method. For more information, see the [Triggers and bindings](#) section of the [Azure Functions C# developer reference](#).

Now that you've created your function project and an HTTP-triggered 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 Functions runtime.

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

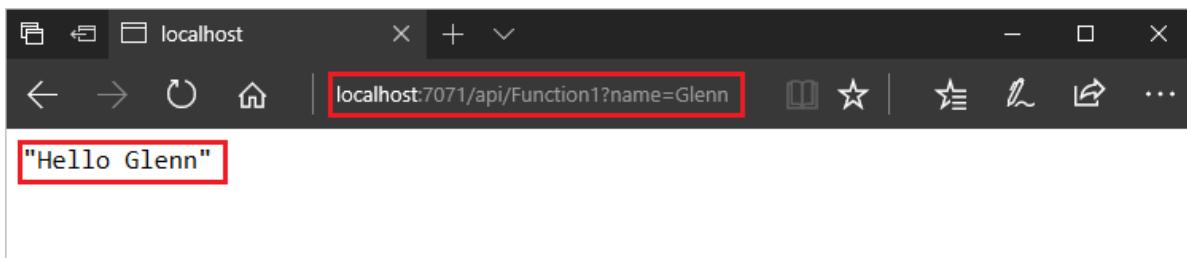


```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.24.0\cli\func.exe
[7/6/2019 6:47:40 PM] Loading functions metadata
[7/6/2019 6:47:40 PM] 1 functions loaded
[7/6/2019 6:47:40 PM] Generating 1 job function(s)
[7/6/2019 6:47:40 PM] Found the following functions:
[7/6/2019 6:47:40 PM] MyFirstFunction.Function1.Run
[7/6/2019 6:47:40 PM]
[7/6/2019 6:47:40 PM] Host initialized (574ms)
[7/6/2019 6:47:40 PM] Host started (594ms)
[7/6/2019 6:47:40 PM] Job host started
Hosting environment: Production
Content root path: C:\Users\glenga\source\repos\MyFirstFunction\MyFirstFunction\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:
    Function1: [GET,POST] http://localhost:7071/api/Function1

[7/6/2019 6:47:47 PM] Host lock lease acquired by instance ID '0000000000000000000000000000CF523E2A'.
```

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 execute the request. The following shows the response in the browser to the local GET request returned by the function:



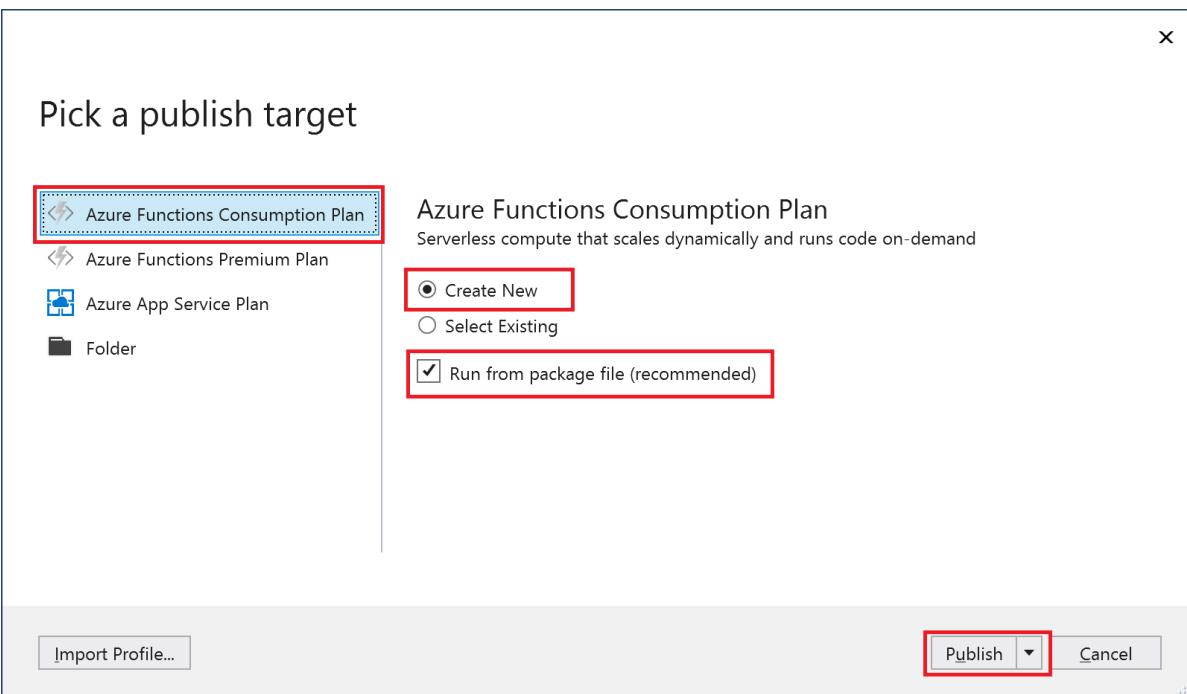
4. To stop debugging, press **Shift + F5**.

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

Publish the project to Azure

You must have a function app in your Azure subscription before you can publish your project. 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 the **Pick a publish target** dialog, use the publish options as specified in the table below the image:



OPTION	DESCRIPTION
Azure Functions Consumption plan	When you publish your project to a function app that runs in a Consumption plan , you only pay for executions of your functions app. Other hosting plans incur higher costs. To learn more, see Azure Functions scale and hosting .
Create new	A new function app, with related resources, is created in Azure. When you choose Select Existing , all files in the existing function app in Azure are overwritten by files from the local project. Only use this option when republishing updates to an existing function app.
Run from package file	Your function app is deployed using Zip Deploy with Run-From-Package mode enabled. This is the recommended way of running your functions, which results in better performance.

3. Select **Publish**. If you haven't already signed-in to your Azure account from Visual Studio, select **Sign-in**. You can also create a free Azure account.
4. In the **App Service: Create new** dialog, use the **Hosting** settings as specified in the table below the image:

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Globally unique name	Name that uniquely identifies your new function app. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Choose your subscription	The Azure subscription to use.
Resource Group	myResourceGroup	Name of the resource group in which to create your function app. Choose New to create a new resource group.
Hosting Plan	Consumption plan	Make sure to choose the Consumption under Size after you select New to create a serverless plan. Also, choose a Location in a region near you or near other services your functions access. When you run in a plan other than Consumption , you must manage the scaling of your function app .
Azure Storage	General-purpose storage account	An Azure storage account is required by the Functions runtime. Select New to create a general-purpose storage account. You can also use an existing account that meets the storage account requirements .

- Select **Create** to create a function app and related resources in Azure with these settings and deploy your function project code.
- After the deployment is complete, make a note of the **Site URL** value, which is the address of your function

app in Azure.

Publish

Publish your app to Azure or another host. [Learn more](#)

New Profile... Actions▼

Site URL	https://mydemofunctiona...	Manage Application Settings...
Configuration	Release	Manage Profile Settings...
Delete existing files	True	
Username	\$myDemoFunctionApp20181016	
Password	*****	

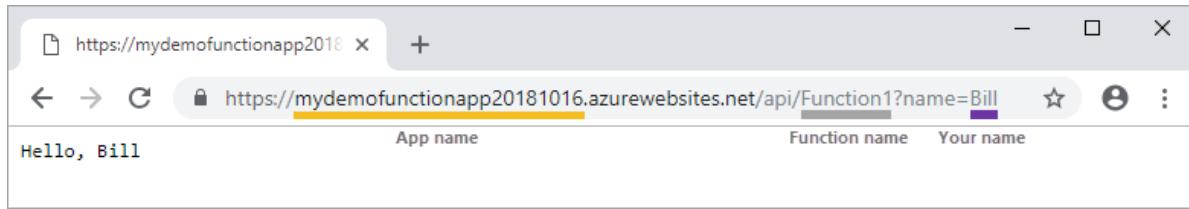
Test your function in Azure

1. Copy the base URL of the function app from the Publish profile page. Replace the `localhost:port` portion of the URL you used when testing the function locally with the new base URL. As before, make sure to append the query string `?name=<YOUR_NAME>` to this URL and execute the request.

The URL that calls your HTTP triggered function should be in the following format:

```
http://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>?name=<YOUR_NAME>
```

2. Paste this new URL for the HTTP request into your browser's address bar. The following shows the response in the browser to the remote GET request returned by the function:



Next steps

You have used Visual Studio to create and publish a C# function app in Azure with a simple HTTP triggered function. To learn more about developing functions as .NET class libraries, see [Azure Functions C# developer reference](#).

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

Create your first function with Java and Maven

7/26/2019 • 5 minutes to read • [Edit Online](#)

This article guides you through using the Maven command-line tool to build and publish a Java function 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 Java, 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](#).

Linux/macOS

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

Windows

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

```
mvn archetype:generate ^
-DarchetypeGroupId=com.microsoft.azure" ^
-DarchetypeArtifactId=azure-functions-archetype"
```

Maven will ask 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 Java 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 are encouraged to supply your own values to Maven in this step.

```
Define value for property 'groupId' (should match expression '[A-Za-z0-9_\-\\.]+'): com.fabrikam.functions
Define value for property 'artifactId' (should match expression '[A-Za-z0-9_\-\\.]+'): fabrikam-functions
Define value for property 'version' 1.0-SNAPSHOT :
Define value for property 'package': com.fabrikam.functions
Define value for property 'appName' fabrikam-functions-20170927220323382:
Define value for property 'appRegion' westus: :
Define value for property 'resourceGroup' java-functions-group: :
Confirm properties configuration: Y
```

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 an [HTTP triggered](#) function that echoes the body of the request. Replace `src/main/java/com/fabrikam/functions/Function.java` with the following code:

```
package com.fabrikam.functions;

import java.util.*;
import com.microsoft.azure.functions.annotation.*;
import com.microsoft.azure.functions.*;

public class Function {
    /**
     * This function listens at endpoint "/api/hello". Two ways to invoke it using "curl" command in bash:
     * 1. curl -d "HTTP Body" {your host}/api/hello
     * 2. curl {your host}/api/hello?name=HTTP%20Query
     */
    @FunctionName("hello")
    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
        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 {
            return request.createResponseBuilder(HttpStatus.OK).body("Hello, " + name).build();
        }
    }
}
```

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)"  
    }  
}
```

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:

```
Listening on http://localhost:7071  
Hit CTRL-C to exit...  
  
Http Functions:  
  
hello: http://localhost:7071/api/hello
```

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

```
curl -w "\n" http://localhost:7071/api/hello -d LocalFunction
```

```
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. This performs a [Zip Deploy with Run From Package](#) mode enabled.

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/hello -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.createResponse(200, "Hello, " + name);
```

To the following:

```
return request.createResponse(200, "Hi, " + name);
```

Save the changes. Run `mvn clean package` 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/hello
```

Will have updated output:

```
Hi, AzureFunctionsTest
```

Next steps

You have created a Java 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 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.

Create an HTTP triggered function in Azure

6/27/2019 • 6 minutes to read • [Edit Online](#)

NOTE

Python for Azure Functions is currently in preview. To receive important updates, subscribe to the [Azure App Service announcements](#) repository on GitHub.

This article shows you how to use command-line tools to create a Python project that runs in Azure Functions. The function you create is triggered by HTTP requests. Finally, you publish your project to run as a [serverless function](#) in Azure.

This article is the first of two quickstarts for Azure Functions. After you complete this article, you [add an Azure Storage queue output binding](#) to your function.

Prerequisites

Before you start, you must have the following:

- Install [Python 3.6](#).
- Install [Azure Functions Core Tools](#) version 2.6.1071 or a later version.
- Install the [Azure CLI](#) version 2.x or a later version.
- An active Azure subscription.

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

Create and activate a virtual environment

To locally develop and test Python functions, you must work in a Python 3.6 environment. Run the following commands to create and activate a virtual environment named `.env`.

Bash:

```
python3.6 -m venv .venv  
source .venv/bin/activate
```

PowerShell or a Windows command prompt:

```
py -3.6 -m venv .venv  
.venv\scripts\activate
```

The remaining commands are run inside the virtual environment.

Create a local Functions project

A Functions project is the equivalent of a function app in Azure. It can have multiple functions that all share the same local and hosting configurations.

In the virtual environment, run the following command, choosing **python** as your worker runtime.

```
func init MyFunctionProj
```

A folder named *MyFunctionProj* is created, which contains the following three files:

- `local.settings.json` is used to store app settings and connection strings when running locally. This file doesn't get published to Azure.
- `requirements.txt` contains the list of packages to be installed on publishing to Azure.
- `host.json` contains global configuration options that affect all functions in a function app. This file does get published to Azure.

Navigate to the new *MyFunctionProj* folder:

```
cd MyFunctionProj
```

Next, you update the `host.json` file to enable extension bundles.

Create a function

To add a function to your project, run the following command:

```
func new
```

Choose the **HTTP trigger** template, type `HttpTrigger` as the name for the function, then press Enter.

A subfolder named *HttpTrigger* is created, which contains the following files:

- **function.json**: configuration file that defines the function, trigger, and other bindings. Review this file and see that the value for `scriptFile` points to the file containing the function, while the invocation trigger and bindings are defined in the `bindings` array.

Each binding requires a direction, type and a unique name. The HTTP trigger has an input binding of type `httpTrigger` and output binding of type `http`.

- **__init__.py**: script file that is your HTTP triggered function. Review this script and see that it contains a default `main()`. HTTP data from the trigger is passed to this function using the `req` named binding parameter. 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](#).

Run the function locally

The following command starts the function app, which runs locally using the same Azure Functions runtime that is in Azure.

```
func host start
```

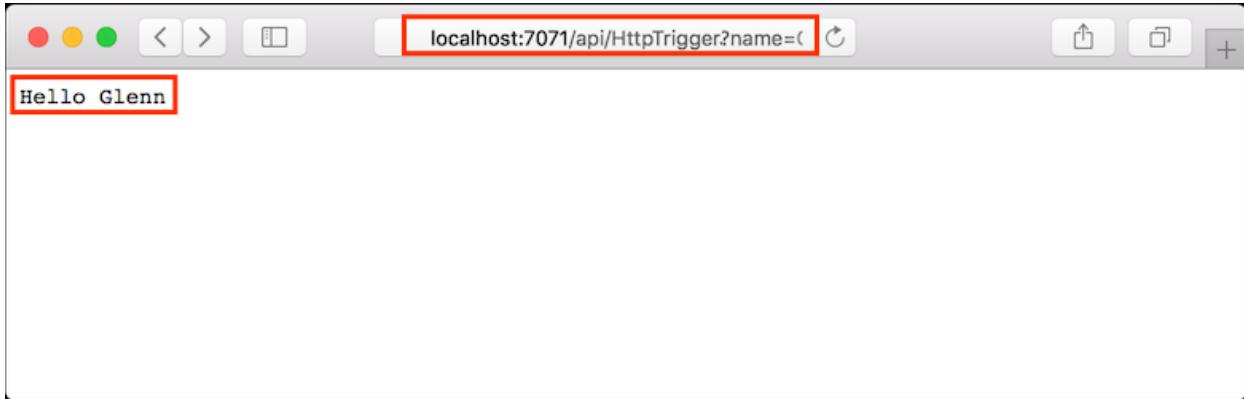
When the Functions host starts, it writes something like the following output, which has been truncated for readability:

```
%%%%%
%%%%%
@  %%%%%%  @
@@  %%%%  @@%
@@@  %%%%%%%  @@@
@@  %%%%%%%%
@@  %%%  @@
@@  %%  @@
@@  %
%
%
...
Content root path: C:\functions\MyFunctionProj
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
...
Http Functions:

HttpTrigger: http://localhost:7071/api/MyHttpTrigger

[8/27/2018 10:38:27 PM] Host started (29486ms)
[8/27/2018 10:38:27 PM] Job host started
```

Copy the URL of your `HttpTrigger` function from the runtime output and paste it into your browser's address bar. Append the query string `?name=<yourname>` to this URL and execute the request. The following shows the response in the browser to the GET request returned by the local function:



Now that you have run your function locally, you can create the function app and other required resources in Azure.

Create a resource group

Create a resource group with the `az group create`. An Azure resource group is a logical container into which Azure resources like function apps, databases, and storage accounts are deployed and managed.

The following example creates a resource group named `myResourceGroup`.

If you are not using Cloud Shell, sign in first using `az login`.

```
az group create --name myResourceGroup --location westeurope
```

You generally create your resource group and the resources in a `region` near you.

Create an Azure Storage account

Functions uses a general-purpose account in Azure Storage to maintain state and other information about your functions. Create a general-purpose storage account in the resource group you created by using the [az storage account create](#) command.

In the following command, substitute a globally unique storage account name where you see the `<storage_name>` placeholder. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

```
az storage account create --name <storage_name> --location westeurope --resource-group myResourceGroup --sku Standard_LRS
```

Create a function app in Azure

A function app provides an environment for executing your function code. It lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Run the following command using a unique function app name in place of the `<APP_NAME>` placeholder and the storage account name for `<STORAGE_NAME>`. The `<APP_NAME>` is also the default DNS domain for the function app. This name needs to be unique across all apps in Azure.

```
az functionapp create --resource-group myResourceGroup --os-type Linux \  
--consumption-plan-location westeurope --runtime python \  
--name <APP_NAME> --storage-account <STORAGE_NAME>
```

NOTE

Azure Functions, Consumption plan for Linux is currently in preview and only available on following regions: West US, East US, West Europe, East Asia. Moreover, Linux and Windows apps cannot be hosted in the same resource group. If you have an existing resource group named `myResourceGroup` with a Windows function app or web app, you must use a different resource group.

You're now ready to publish your local functions project to the function app in Azure.

Deploy the function app project to Azure

After the function app is created in Azure, you can use the [func azure functionapp publish](#) Core Tools command to deploy your project code to Azure. In the following command, replace `<APP_NAME>` with the name of your app from the previous step.

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

You'll see output similar to the following, which has been truncated for readability:

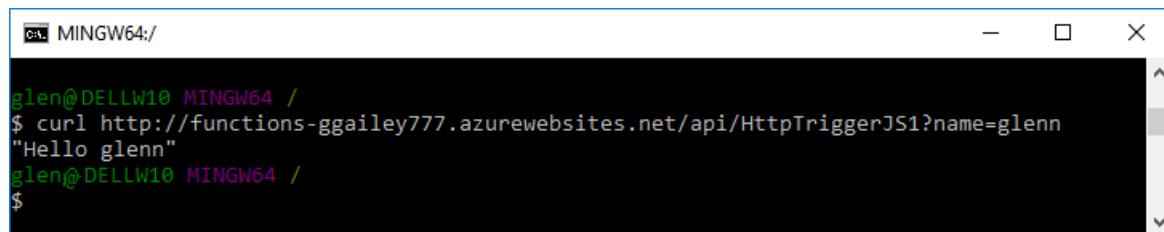
```
Getting site publishing info...
...
Preparing archive...
Uploading content...
Upload completed successfully.
Deployment completed successfully.
Syncing triggers...
Functions in myfunctionapp:
HttpTrigger - [httpTrigger]
    Invoke url: https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....
```

Copy the `Invoke url` value for your `HttpTrigger`, which you can now use to test your function in Azure. The URL contains a `code` query string value that is your function key. This key makes it difficult for others to call your HTTP trigger endpoint in Azure.

Test the function in Azure

Use cURL to test the deployed function. Using the URL that you copied from the previous step, append the query string `&name=<yourusername>` to the URL, as in the following example:

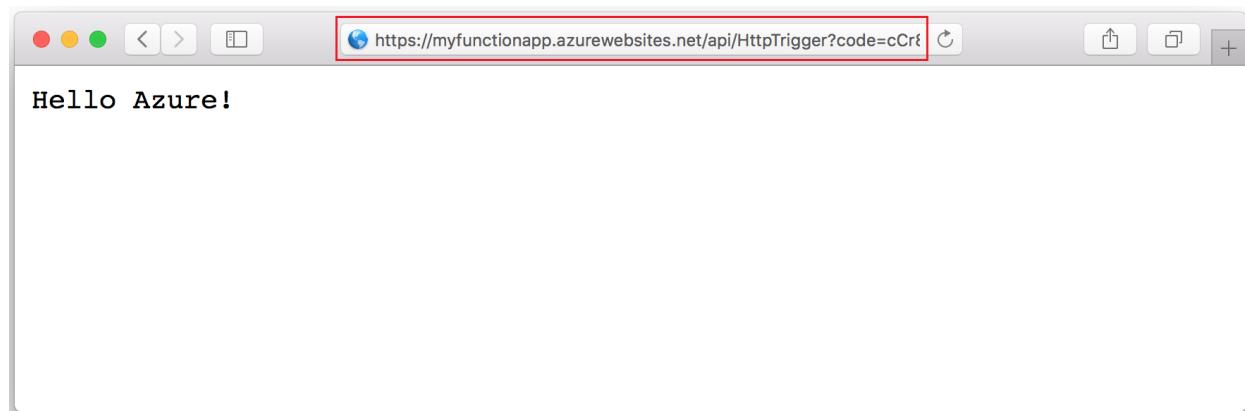
```
curl https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....&name=<yourusername>
```



```
curl https://functions-ggailey777.azurewebsites.net/api/HttpTriggerJS1?name=glen
"Hello glenn"
```

A screenshot of a terminal window titled "MINGW64:". It shows a command being run: "curl https://functions-ggailey777.azurewebsites.net/api/HttpTriggerJS1?name=glen". The output of the command, "Hello glenn", is displayed below the command line.

You can also paste the copied URL in to the address of your web browser. Again, append the query string `&name=<yourusername>` to the URL before you execute the request.



Next steps

You've created a Python 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](#)

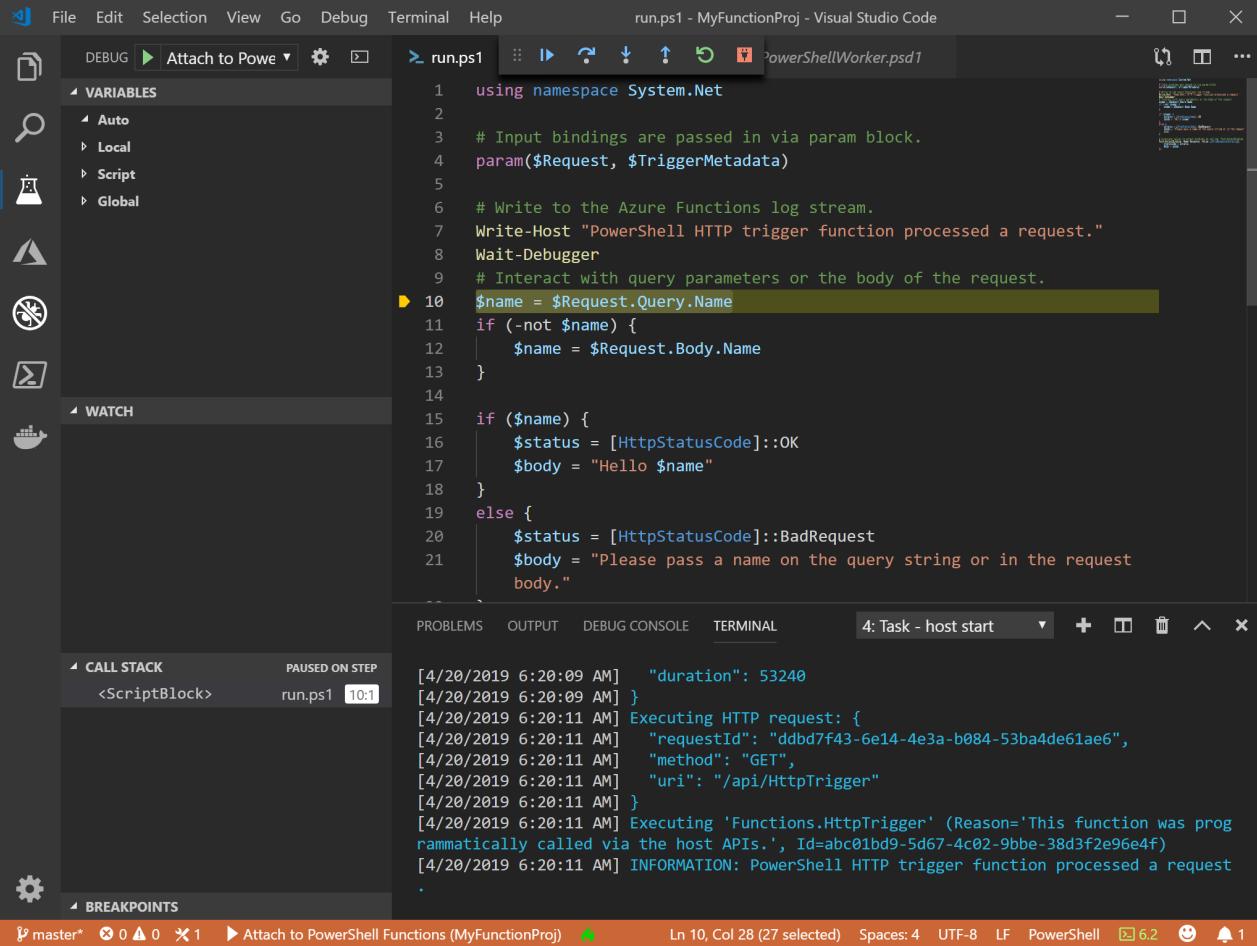
Create your first PowerShell function in Azure (preview)

7/9/2019 • 6 minutes to read • [Edit Online](#)

NOTE

PowerShell for Azure Functions is currently in preview. To receive important updates, subscribe to the [Azure App Service announcements](#) repository on GitHub.

This quickstart article walks you through how to create your first [serverless](#) PowerShell function using Visual Studio Code.



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Debug, Terminal, Help.
- Editor:** The file `run.ps1` is open, showing PowerShell code for an Azure Function. The code handles an HTTP trigger and returns a response based on the name parameter.
- Sidebar:** Variables, Watch, Call Stack, Breakpoints.
- Bottom Status Bar:** master*, 0▲0 ▾1, Attach to PowerShell Functions (MyFunctionProj), Ln 10, Col 28 (27 selected), Spaces: 4, UTF-8, LF, PowerShell, 6.2, 1.

```
1 using namespace System.Net
2
3 # Input bindings are passed in via param block.
4 param($Request, $TriggerMetadata)
5
6 # Write to the Azure Functions log stream.
7 Write-Host "PowerShell HTTP trigger function processed a request."
8 Wait-Debugger
9 # Interact with query parameters or the body of the request.
10 $name = $Request.Query.Name
11 if (-not $name) {
12     $name = $Request.Body.Name
13 }
14
15 if ($name) {
16     $status = [HttpStatusCode]::OK
17     $body = "Hello $name"
18 }
19 else {
20     $status = [HttpStatusCode]::BadRequest
21     $body = "Please pass a name on the query string or in the request body."
```

You use the [Azure Functions extension for Visual Studio Code](#) to create a PowerShell function locally and then deployed it to a new function app in Azure. The extension is currently in preview. To learn more, see the [Azure Functions extension for Visual Studio Code](#) extension page.

NOTE

PowerShell support for the [Azure Functions extension](#) is currently disabled by default. Enabling PowerShell support is one of the steps in this article.

The following steps are supported on macOS, Windows, and Linux-based operating systems.

Prerequisites

To complete this quickstart:

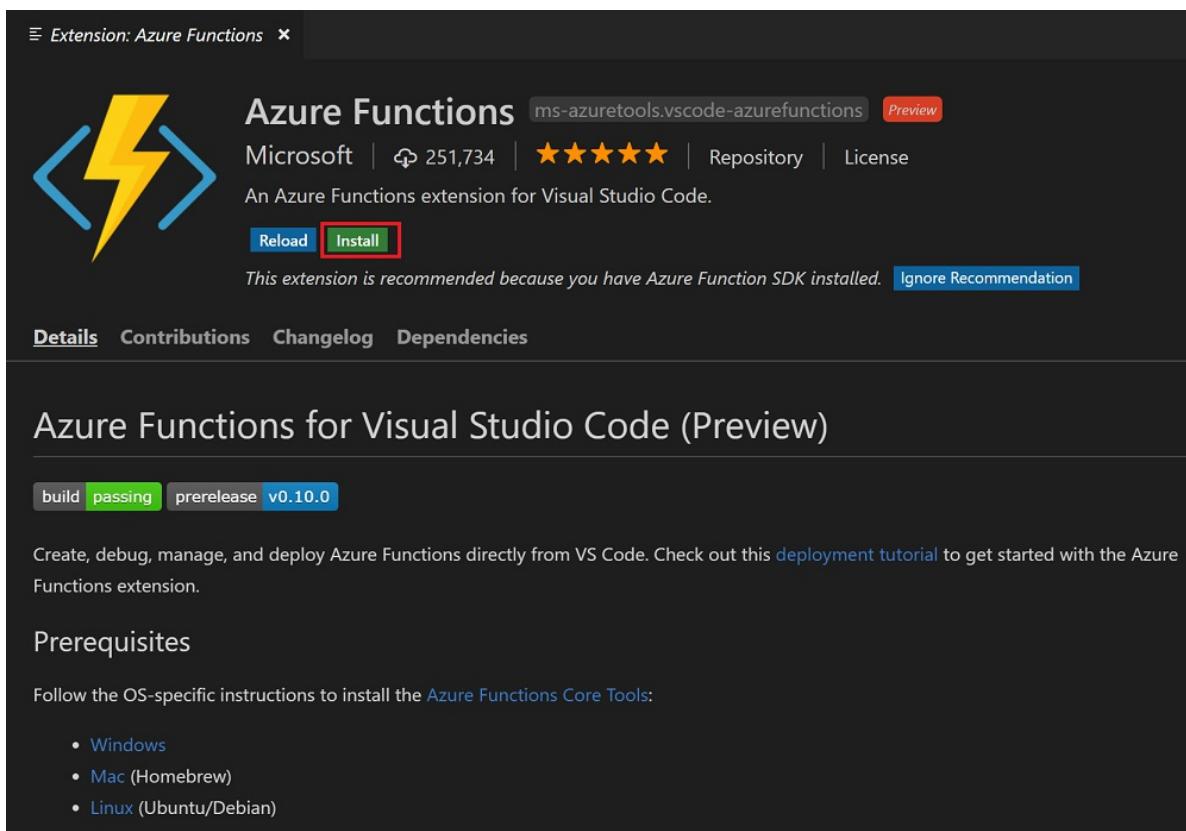
- Install [PowerShell Core](#)
- Install [Visual Studio Code](#) on one of the [supported platforms](#).
- Install [PowerShell extension for Visual Studio Code](#).
- Install [.NET Core SDK 2.2+](#) (required by Azure Functions Core Tools and available on all supported platforms).
- Install version 2.x of the [Azure Functions Core Tools](#).
- You also need an active Azure subscription.

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

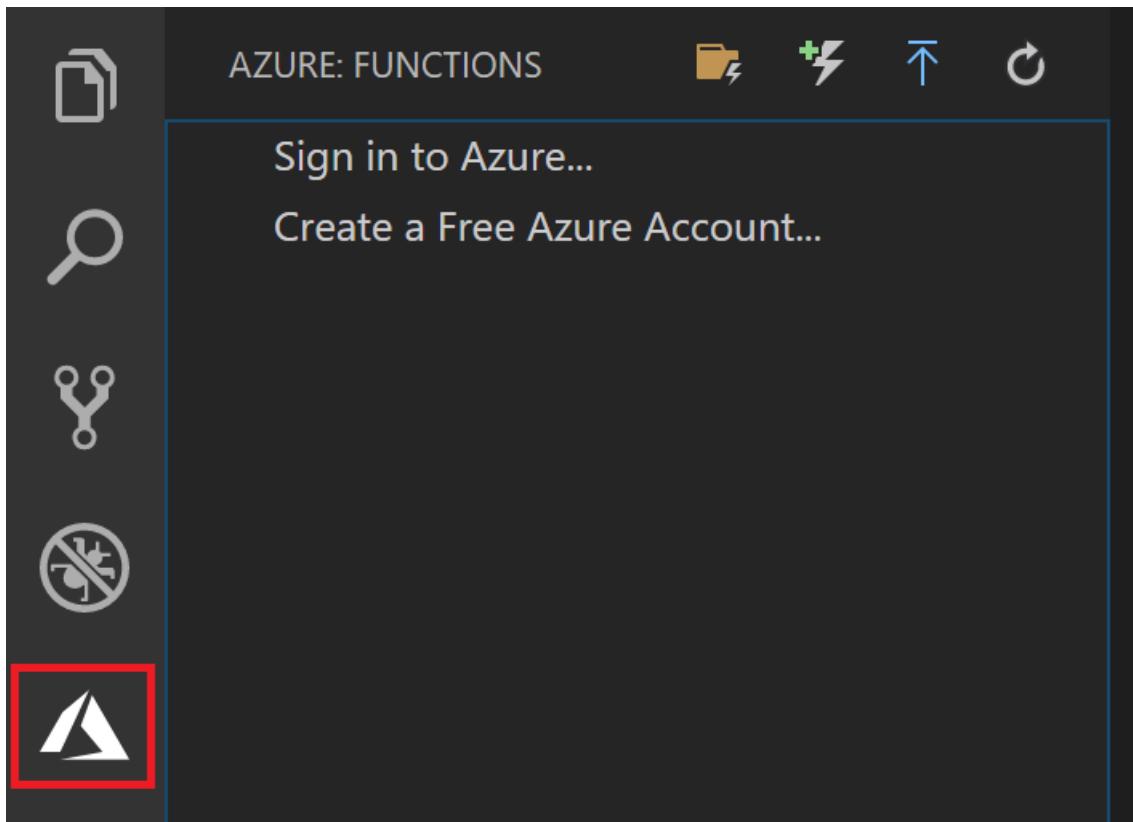
Install the Azure Functions extension

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](#).
2. Select **Install** to install the extension for Visual Studio Code:



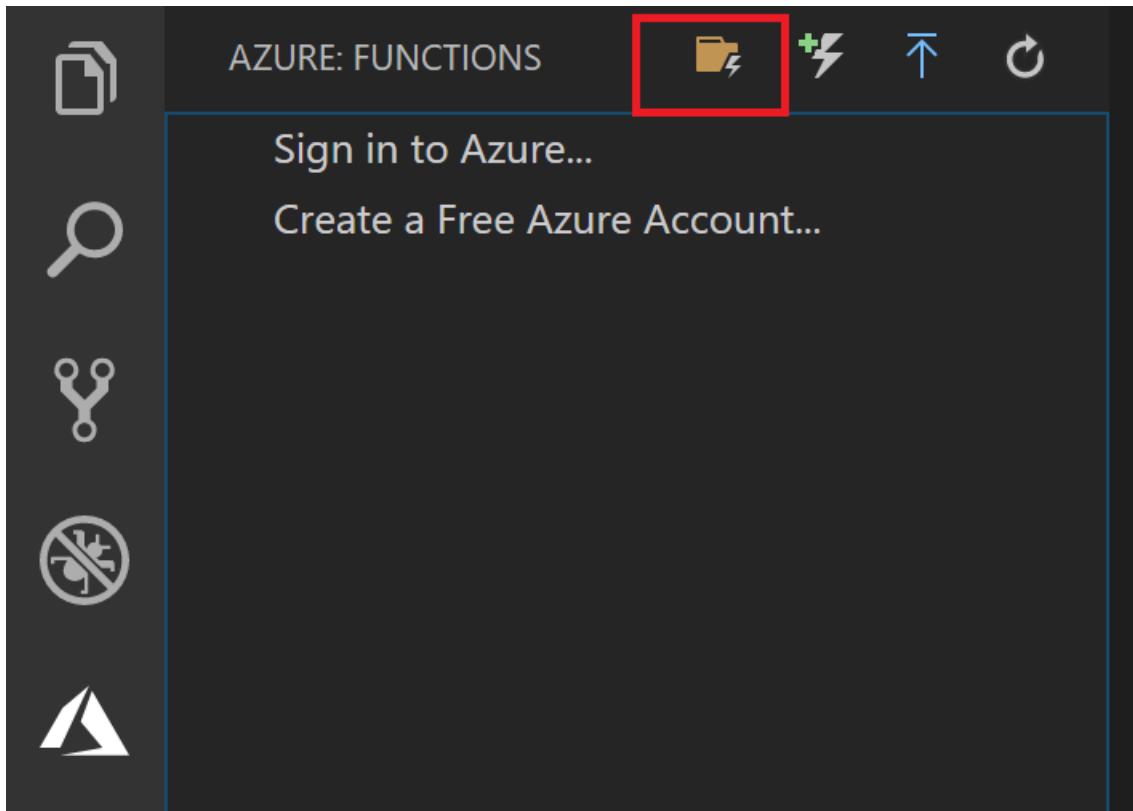
3. Restart Visual Studio Code and select the Azure icon on the Activity bar. You should see an Azure Functions area in the Side Bar.



Create a function app project

The Azure Functions project template in Visual Studio Code creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for management, deployment, and sharing of resources.

1. In Visual Studio Code, select the Azure logo to display the **Azure: Functions** area, and then select the Create New Project icon.



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

NOTE

This article was designed to be completed outside of a workspace. In this case, do not select a project folder that is part of a workspace.

3. Select the **Powershell (preview)** as the language for your function app project and then **Azure Functions v2**.
4. Choose **HTTP Trigger** as the template for your first function, use `HTTPTrigger` as the function name, and choose an authorization level of **Function**.

NOTE

The **Function** authorization level requires a [function key](#) value when calling the function endpoint in Azure. This makes it harder for just anyone to call your function.

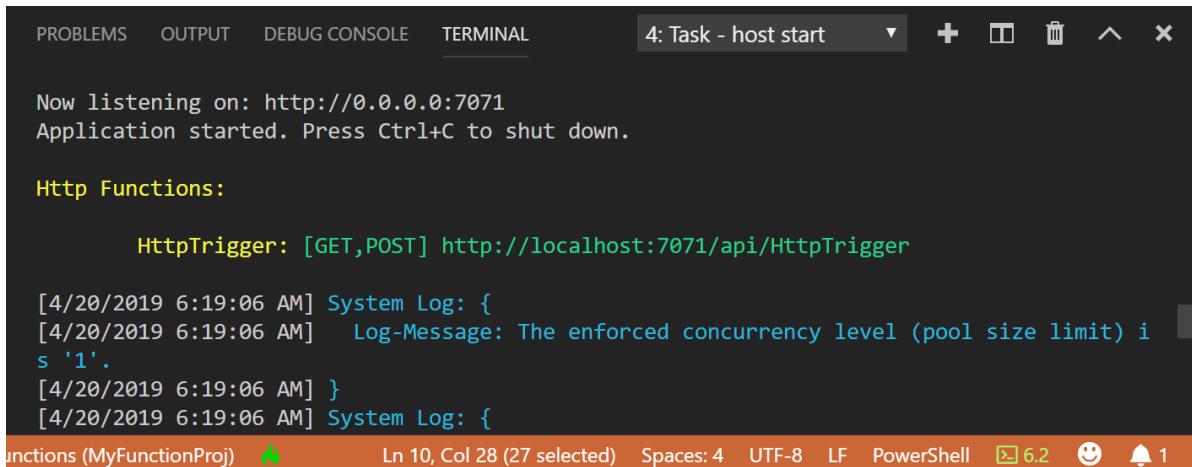
5. When prompted, choose **Add to workspace**.

Visual Studio Code creates the PowerShell function app project in a new workspace. This project contains the `host.json` and `local.settings.json` configuration files, which apply to all function in the project. This [PowerShell project](#) is the same as a function app running in Azure.

Run the function locally

Azure Functions Core Tools integrates with Visual Studio Code to let you run and debug an Azure Functions project locally.

1. To debug your function, insert a call to the `Wait-Debugger` cmdlet in the function code before you want to attach the debugger, then press F5 to start the function app project and attach the debugger. 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: {
```

Junctions (MyFunctionProj) 4 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` 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.

When you call the `HttpTrigger` endpoint without passing a `name` parameter either as a query parameter or in the body, the function returns a 500 error. When you review the code in `run.ps1`, you see that this error occurs by design.

4. To stop debugging, press Shift + F5.

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

NOTE

Remember to remove any calls to `Wait-Debugger` before you publish your functions to Azure.

NOTE

Creating a Function App in Azure will only prompt for Function App name. Set `azureFunctions.advancedCreation` to true to be prompted for all other values.

Publish the project 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.

By default, Visual Studio creates all of the Azure resources required to create your function app. The names of these resources are based on the function app name you choose. If you need to have full control of the created resources, you can instead [publish using advanced options](#).

This section assumes that you are creating a new function app in Azure.

IMPORTANT

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

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Deploy to function app...`.
2. If not signed-in, you are prompted to **Sign in to Azure**. You can also **Create a free Azure account**. After successful sign in from the browser, go back to Visual Studio Code.
3. If you have multiple subscriptions, **Select a subscription** for the function app, then choose **+ Create New Function App in Azure**.
4. Type a globally unique name that identifies your function app and press Enter. Valid characters for a function app name are `a-z`, `0-9`, and `-`.

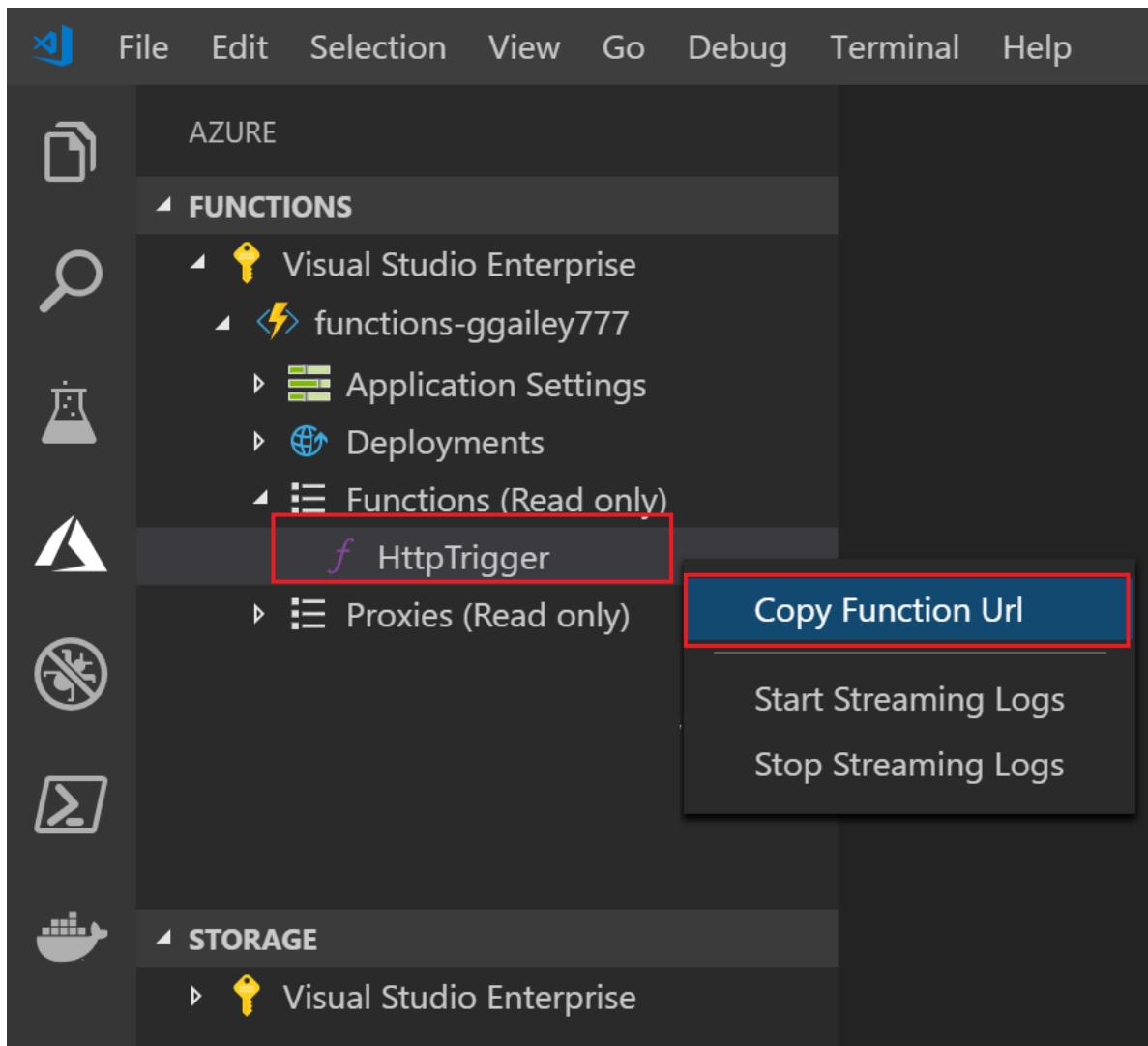
When you press Enter, the following Azure resources are created in your subscription:

- **Resource group**: Contains all of the created Azure resources. The name is based on your function app name.
- **Storage account**: A standard Storage account is created with a unique name that is based on your function app name.
- **Hosting plan**: A consumption plan is created in the West US region to host your serverless function app.

- **Function app:** Your project is deployed to and runs in this new function app.

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.

5. Back in the **Azure: Functions** area, expand the new function app under your subscription. Expand **Functions**, right-click **HttpTrigger**, and then choose **Copy function URL**.



Run the function in Azure

To verify that your published function runs in Azure, execute the following PowerShell command, replacing the `Uri` parameter with the URL of the HTTPTrigger function from the previous step. As before, append the query string `&name=<yourname>` to the URL, as in the following example:

```
PS > Invoke-WebRequest -Method Get -Uri "https://glengatest-vscode-
powershell.azurewebsites.net/api/HttpTrigger?code=nrY05eZutfPqLo0som...&name=PowerShell"

StatusCode      : 200
StatusDescription : OK
Content         : Hello PowerShell
RawContent      : HTTP/1.1 200 OK
                  Content-Length: 16
                  Content-Type: text/plain; charset=utf-8
                  Date: Thu, 25 Apr 2019 16:01:22 GMT

                  Hello PowerShell
Forms           : {}
Headers         : {[Content-Length, 16], [Content-Type, text/plain; charset=utf-8], [Date, Thu, 25 Apr 2019
16:01:22 GMT]}
Images          : {}
InputFields     : {}
Links           : {}
ParsedHtml      : mshtml.HTMLDocumentClass
RawContentLength : 16
```

Next steps

You have used Visual Studio Code to create a PowerShell function app with a simple HTTP-triggered function. You may also want to learn more about [debugging a PowerShell function locally](#) using the Azure Functions Core Tools. Check out the [Azure Functions PowerShell developer guide](#).

[Enable Application Insights integration](#)

Create your first function from the command line

6/26/2019 • 6 minutes to read • [Edit Online](#)

This quickstart topic walks you through how to create your first function from the command line or terminal. You use the Azure CLI to create a function app, which is the [serverless](#) infrastructure that hosts your function. The function code project is generated from a template by using the [Azure Functions Core Tools](#), which is also used to deploy the function app project to Azure.

You can follow the steps below using a Mac, Windows, or Linux computer.

Prerequisites

Before running this sample, you must have the following:

- Install [Azure Functions Core Tools](#) version 2.6.666 or later.
- Install the [Azure CLI](#). This article requires the Azure CLI version 2.0 or later. Run `az --version` to find the version you have. You can also use the [Azure Cloud Shell](#).
- An active Azure subscription.

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

Create the local function app project

Run the following command from the command line to create a function app project in the `MyFunctionProj` folder of the current local directory. A GitHub repo is also created in `MyFunctionProj`.

```
func init MyFunctionProj
```

When prompted, select a worker runtime from the following language choices:

- `dotnet` : creates a .NET class library project (.csproj).
- `node` : creates a JavaScript project.

When the command executes, you see something like the following output:

```
Writing .gitignore
Writing host.json
Writing local.settings.json
Initialized empty Git repository in C:/functions/MyFunctionProj/.git/
```

Use the following command to navigate to the new `MyFunctionProj` project folder.

```
cd MyFunctionProj
```

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)"  
    }  
}
```

Create a function

The following command creates an HTTP-triggered function named `MyHttpTrigger`.

```
func new --name MyHttpTrigger --template "HttpTrigger"
```

When the command executes, you see something like the following output:

```
The function "MyHttpTrigger" was created successfully from the "HttpTrigger" template.
```

Run the function locally

The following command starts the function app. The app runs using the same Azure Functions runtime that is in Azure.

```
func host start --build
```

The `--build` option is required to compile C# projects. You don't need this option for a JavaScript project.

When the Functions host starts, it writes something like the following output, which has been truncated for readability:

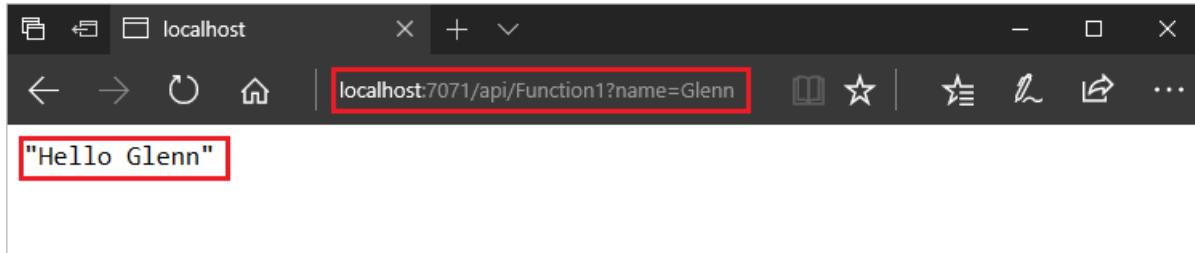
```
%%%%%
%%%%%
@ %%%% @
@@ %%%% @@%
@@@ %%%%%%%%%@@%
@@ %%%%%@@%
@@ %%@@%
@@ %@@%
%
...
Content root path: C:\functions\MyFunctionProj
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

...
Http Functions:

HttpTrigger: http://localhost:7071/api/MyHttpTrigger

[8/27/2018 10:38:27 PM] Host started (29486ms)
[8/27/2018 10:38:27 PM] Job host started
```

Copy the URL of your `HttpTrigger` function from the runtime output and paste it into your browser's address bar. Append the query string `?name=<yourname>` to this URL and execute the request. The following shows the response in the browser to the GET request returned by the local function:



Now that you have run your function locally, you can create the function app and other required resources in Azure.

Create a resource group

Create a resource group with the `az group create`. An Azure resource group is a logical container into which Azure resources like function apps, databases, and storage accounts are deployed and managed.

The following example creates a resource group named `myResourceGroup`.

If you are not using Cloud Shell, sign in first using `az login`.

```
az group create --name myResourceGroup --location westeurope
```

You generally create your resource group and the resources in a `region` near you.

Create an Azure Storage account

Functions uses a general-purpose account in Azure Storage to maintain state and other information about your functions. Create a general-purpose storage account in the resource group you created by using the `az storage`

[account create](#) command.

In the following command, substitute a globally unique storage account name where you see the `<storage_name>` placeholder. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

```
az storage account create --name <storage_name> --location westeurope --resource-group myResourceGroup --sku Standard_LRS
```

Create a function app

You must have a function app to host the execution of your functions. The function app provides an environment for serverless execution of your function code. It lets you group functions as a logic unit for easier management, deployment, and sharing of resources. Create a function app by using the [az functionapp create](#) command.

In the following command, substitute a unique function app name where you see the `<APP_NAME>` placeholder and the storage account name for `<STORAGE_NAME>`. The `<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. You should also set the `<language>` runtime for your function app, from `dotnet` (C#) or `node` (JavaScript).

```
az functionapp create --resource-group myResourceGroup --consumption-plan-location westeurope \
--name <APP_NAME> --storage-account <STORAGE_NAME> --runtime <language>
```

Setting the `consumption-plan-location` parameter means that the function app is hosted in a Consumption hosting plan. In this serverless plan, resources are added dynamically as required by your functions and you only pay when functions are running. For more information, see [Choose the correct hosting plan](#).

After the function app has been created, the Azure CLI shows information similar to the following example:

```
{
  "availabilityState": "Normal",
  "clientAffinityEnabled": true,
  "clientCertEnabled": false,
  "containerSize": 1536,
  "dailyMemoryTimeQuota": 0,
  "defaultHostName": "quickstart.azurewebsites.net",
  "enabled": true,
  "enabledHostNames": [
    "quickstart.azurewebsites.net",
    "quickstart.scm.azurewebsites.net"
  ],
  ...
  // Remaining output has been truncated for readability.
}
```

Deploy the function app project to Azure

After the function app is created in Azure, you can use the `func azure functionapp publish` Core Tools command to deploy your project code to Azure. In the following command, replace `<APP_NAME>` with the name of your app from the previous step.

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

You'll see output similar to the following, which has been truncated for readability:

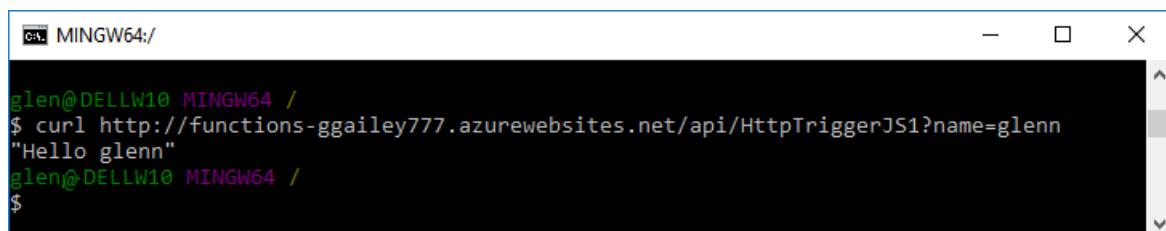
```
Getting site publishing info...
...
Preparing archive...
Uploading content...
Upload completed successfully.
Deployment completed successfully.
Syncing triggers...
Functions in myfunctionapp:
HttpTrigger - [httpTrigger]
    Invoke url: https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....
```

Copy the `Invoke url` value for your `HttpTrigger`, which you can now use to test your function in Azure. The URL contains a `code` query string value that is your function key. This key makes it difficult for others to call your HTTP trigger endpoint in Azure.

Test the function in Azure

Use cURL to test the deployed function. Using the URL that you copied from the previous step, append the query string `&name=<yourusername>` to the URL, as in the following example:

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



```
MINGW64:/ 
glen@DELLW10 MINGW64 / 
$ curl http://functions-ggailey777.azurewebsites.net/api/HttpTriggerJS1?name=glen 
"Hello glenn" 
glen@DELLW10 MINGW64 / 
$
```

You can also paste the copied URL in to the address of your web browser. Again, append the query string `&name=<yourusername>` to the URL before you execute the request.



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

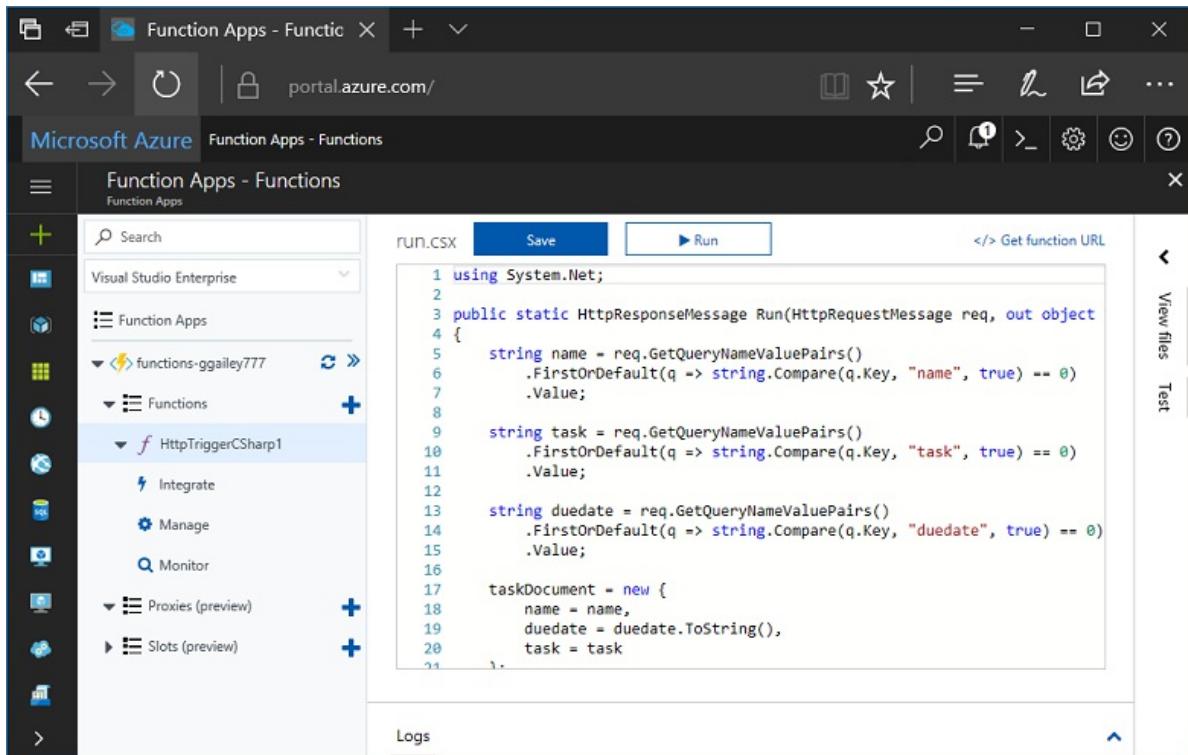
Learn more about developing Azure Functions locally using the Azure Functions Core Tools.

[Code and test Azure Functions locally](#)

Create your first function in the Azure portal

5/17/2019 • 4 minutes to read • [Edit Online](#)

Azure Functions lets you execute your code in a [serverless](#) environment without having to first create a VM or publish a web application. In this article, learn how to use Functions to create a "hello world" function in the Azure portal.



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

NOTE

C# developers should consider [creating your first function in Visual Studio 2019](#) instead of in the portal.

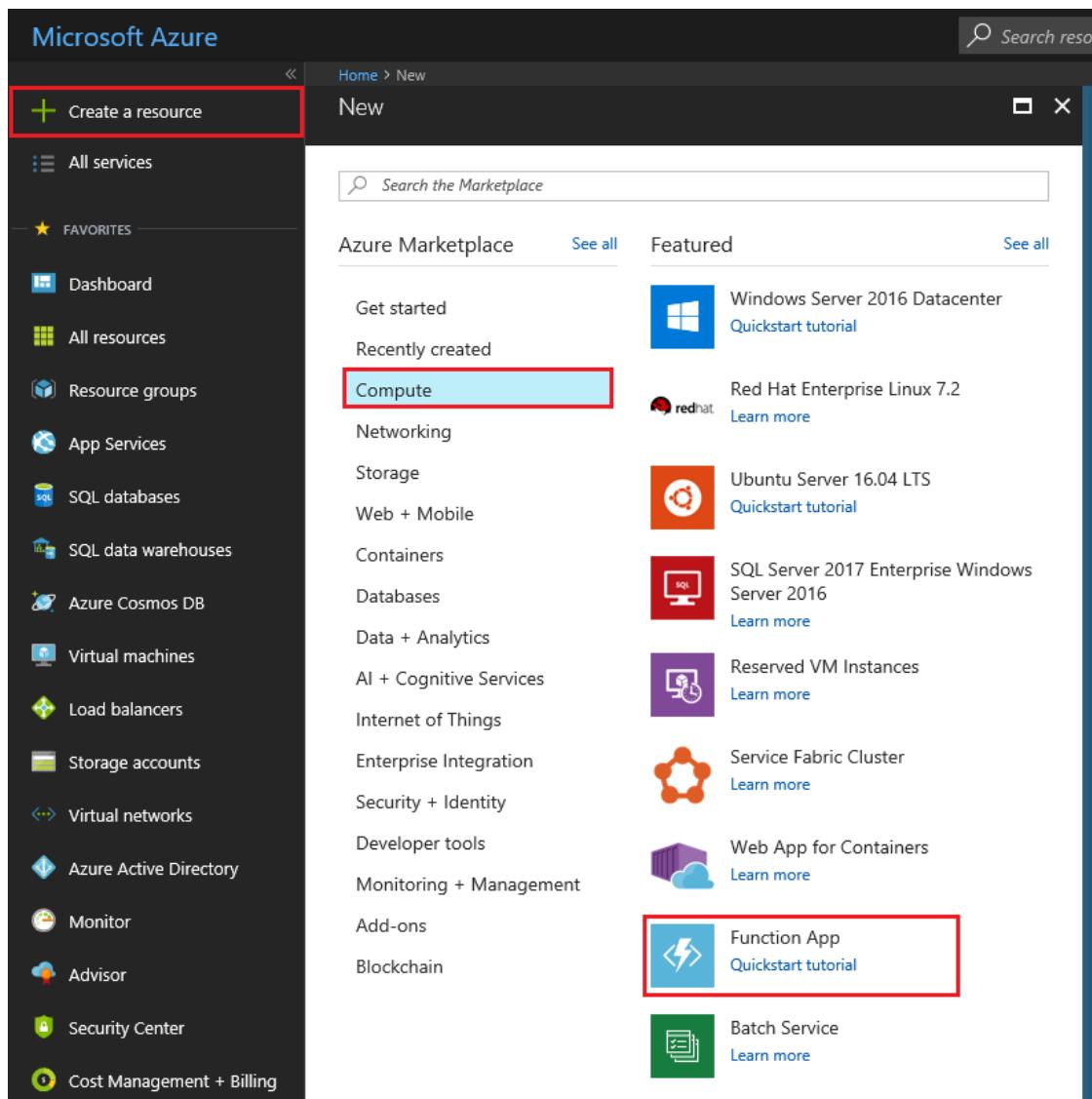
Log in to Azure

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. A function app lets you group functions as a logic unit for easier management, deployment, and sharing of resources.

1. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.



2. Use the function app settings as specified in the table below the image.

Function App

Create

*** App name**
myfunctionapp

.azurewebsites.net

*** Subscription**
Visual Studio Enterprise

*** Resource Group i**
 Create new Use existing

myResourceGroup

*** OS**
 Windows Linux

*** Hosting Plan i**
Consumption Plan

*** Location**
Central US

*** Runtime Stack**
.NET

*** Storage i**
 Create new Use existing

myfunctionapp99761

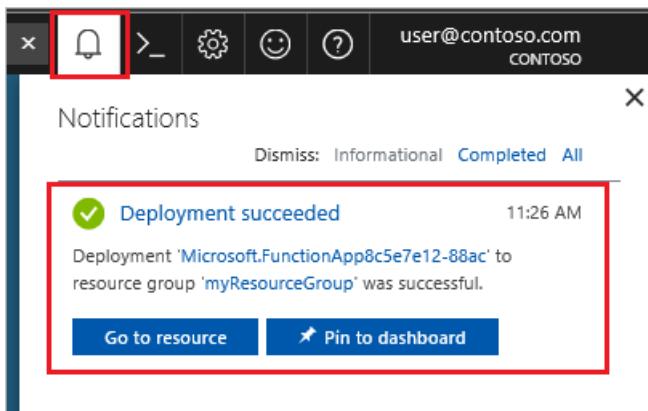
Application Insights >
myfunctionapp9

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z, 0-9, and -.
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.

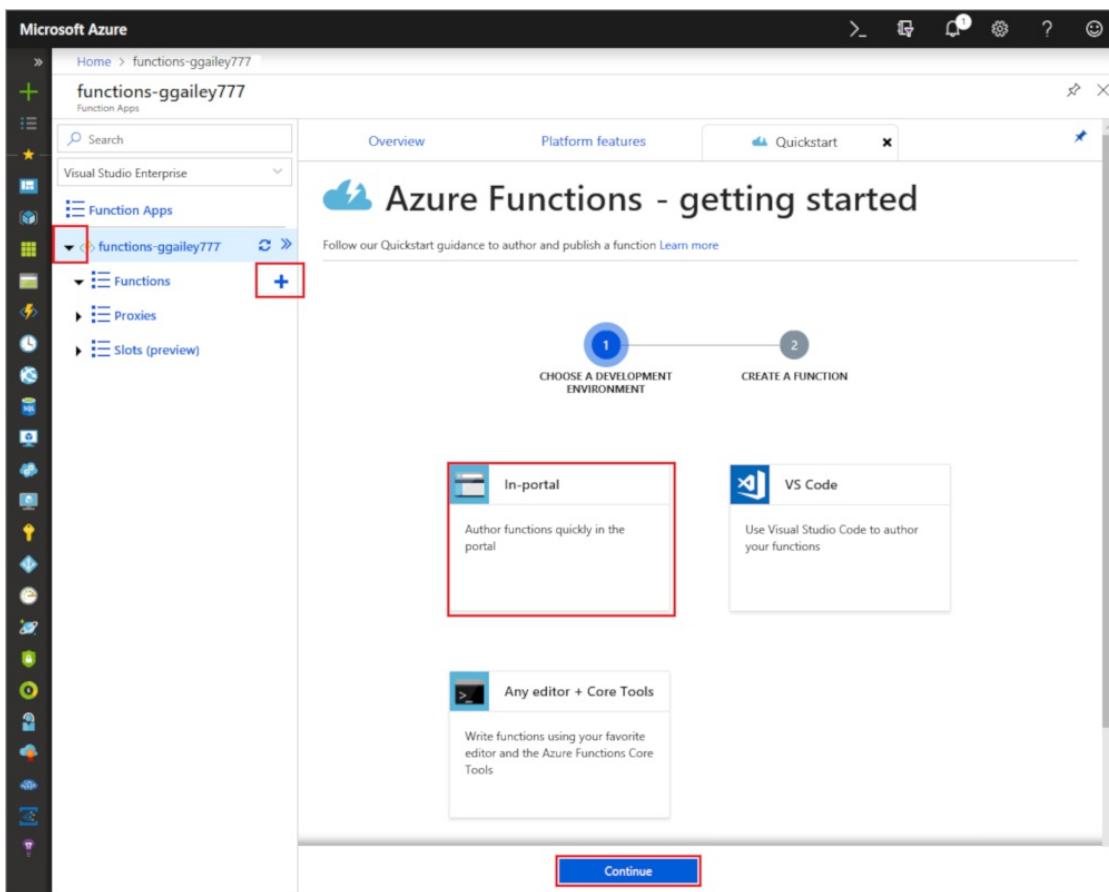


5. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

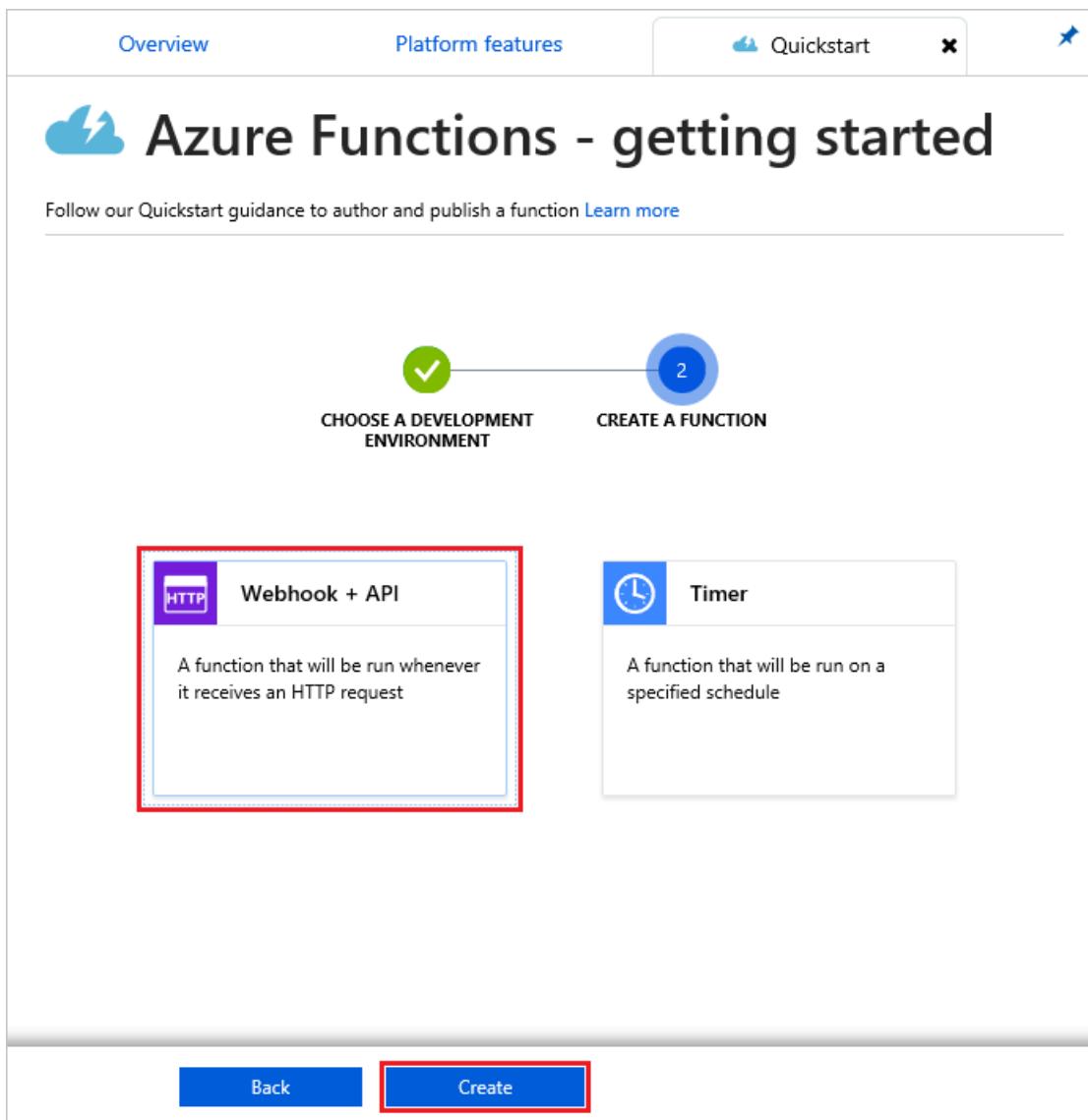
Next, you create a function in the new function app.

Create an HTTP triggered function

1. Expand your new function app, then select the + button next to **Functions**, choose **In-portal**, and select **Continue**.



2. Choose **WebHook + API** and then select **Create**.

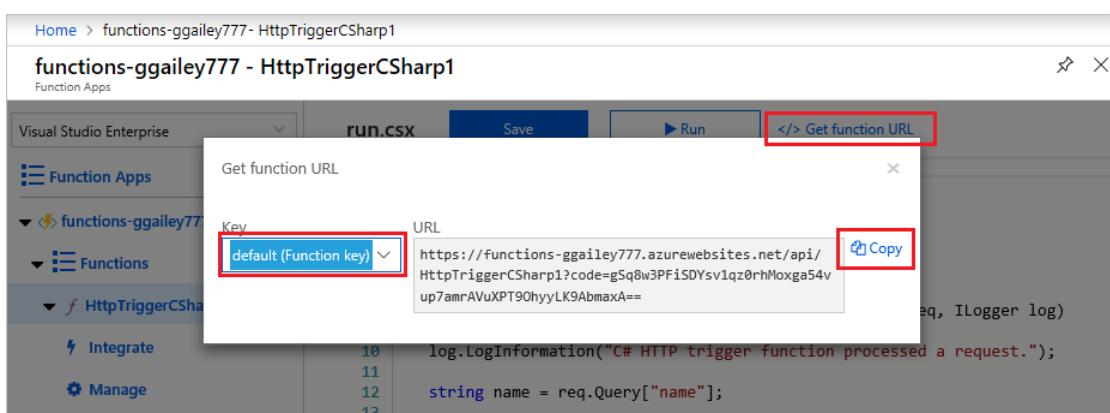


A function is created using a language-specific template for an HTTP triggered function.

Now, you can run the new function by sending an HTTP request.

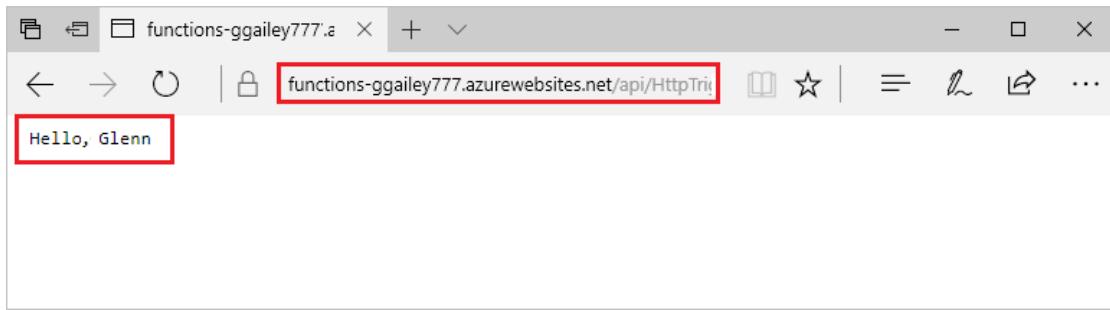
Test the function

1. In your new function, click **</> Get function URL** at the top right, select **default (Function key)**, and then click **Copy**.



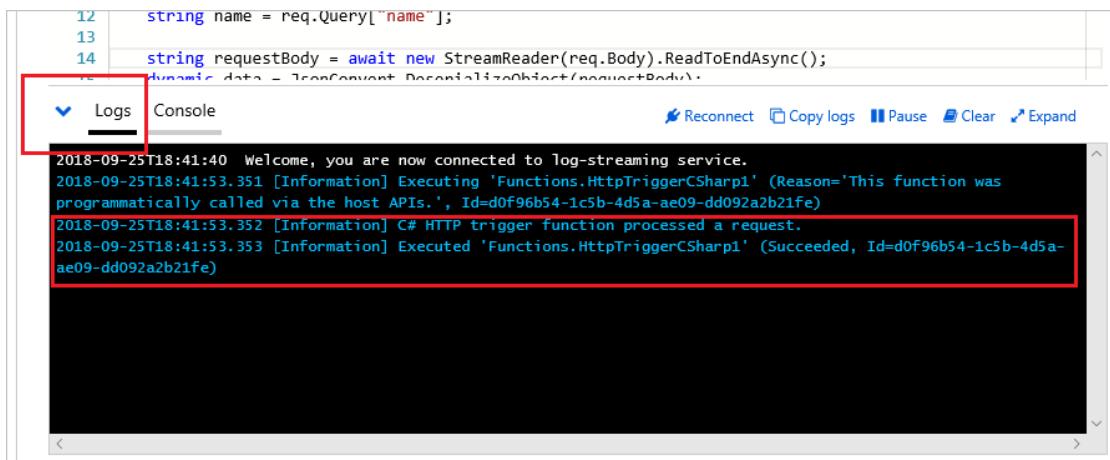
2. Paste the function URL into your browser's address bar. Add the query string value `&name=<yourusername>` to the end of this URL and press the **Enter** key on your keyboard to execute the request. You should see the response returned by the function displayed in the browser.

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.

3. When your function runs, trace information is written to the logs. To see the trace output from the previous execution, return to your function in the portal and click the arrow at the bottom of the screen to expand the **Logs**.



Clean up resources

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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 Microsoft Azure Functions dashboard. In the top navigation bar, the URL is 'Microsoft Azure functions ggailey777'. The main area is titled 'functions-ggailey777 Function Apps'. On the left, there's a sidebar with icons for creating new resources and managing existing ones. The main content area has tabs: 'Overview' (which is selected and highlighted with a red box), 'Settings', 'Platform features', and 'API definition (preview)'. Below these tabs are buttons for 'Stop', 'Swap', 'Restart', 'Download publish profile', 'Reset publish credentials', and 'Download ap'. The 'Overview' section displays the following details:

- Status: Running
- Subscription: Visual Studio Enterprise
- Resource group: functions-ggailey777 (highlighted with a red box)
- Subscription ID
- Location: South Central US
- URL: https://fun...
- App Service: SouthCent

A section titled 'Configured features' is present with the note 'Quick links to your features will show up here after'.

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 are 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 app with a simple HTTP triggered function.

Now that you have 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](#).

Create your first function hosted on Linux using Core Tools and the Azure CLI (preview)

6/26/2019 • 6 minutes to read • [Edit Online](#)

Azure Functions lets you execute your code in a [serverless](#) Linux environment without having to first create a VM or publish a web application. Linux-hosting requires [the Functions 2.0 runtime](#). Support to run a function app on Linux in the serverless [Consumption plan](#) is currently in preview. To learn more, see [this preview considerations article](#).

This quickstart article walks you through how to use the Azure CLI to create your first function app running on Linux. The function code is created locally and then deployed to Azure by using the [Azure Functions Core Tools](#).

The following steps are supported on a Mac, Windows, or Linux computer. This article shows you how to create functions in either JavaScript or C#. To learn how to create Python functions, see [Create your first Python function using Core Tools and the Azure CLI \(preview\)](#).

Prerequisites

Before running this sample, you must have the following:

- Install [Azure Functions Core Tools](#) version 2.6.666 or above.
- Install the [Azure CLI](#). This article requires the Azure CLI version 2.0 or later. Run `az --version` to find the version you have. You can also use the [Azure Cloud Shell](#).
- An active Azure subscription.

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

Create the local function app project

Run the following command from the command line to create a function app project in the `MyFunctionProj` folder of the current local directory. A GitHub repo is also created in `MyFunctionProj`.

```
func init MyFunctionProj
```

When prompted, use the arrow keys to select a worker runtime from the following language choices:

- `dotnet` : creates a .NET class library project (.csproj).
- `node` : creates a JavaScript or TypeScript project. When prompted, choose `JavaScript`.
- `python` : creates a Python project. For Python functions, see the [Python quickstart](#).

When the command executes, you see something like the following output:

```
Writing .gitignore
Writing host.json
Writing local.settings.json
Initialized empty Git repository in C:/functions/MyFunctionProj/.git/
```

Use the following command to navigate to the new `MyFunctionProj` project folder.

```
cd MyFunctionProj
```

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)"  
    }  
}
```

Create a function

The following command creates an HTTP-triggered function named `MyHttpTrigger`.

```
func new --name MyHttpTrigger --template "HttpTrigger"
```

When the command executes, you see something like the following output:

```
The function "MyHttpTrigger" was created successfully from the "HttpTrigger" template.
```

Run the function locally

The following command starts the function app. The app runs using the same Azure Functions runtime that is in Azure.

```
func host start --build
```

The `--build` option is required to compile C# projects. You don't need this option for a JavaScript project.

When the Functions host starts, it writes something like the following output, which has been truncated for readability:

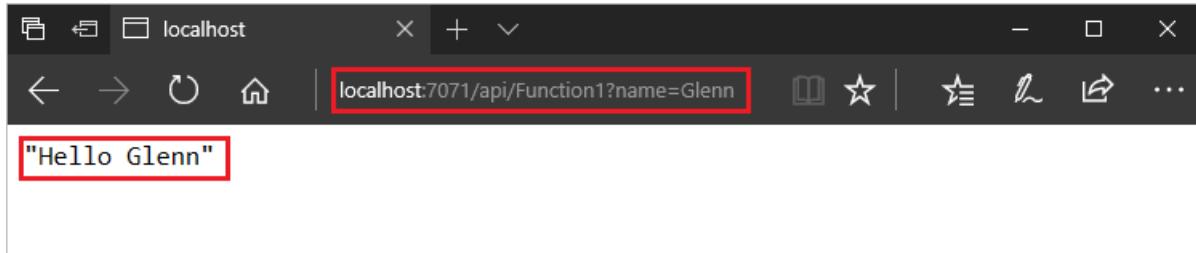
```
%%%%%
%%%%%
@ %%%% @
@@ %%%% @@%
@@@ %%%%%%%%%@@%
@@ %%%%%@@%
@@ %%@@%
@@ %@@%
%
...
Content root path: C:\functions\MyFunctionProj
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

...
Http Functions:

HttpTrigger: http://localhost:7071/api/MyHttpTrigger

[8/27/2018 10:38:27 PM] Host started (29486ms)
[8/27/2018 10:38:27 PM] Job host started
```

Copy the URL of your `HttpTrigger` function from the runtime output and paste it into your browser's address bar. Append the query string `?name=<yourname>` to this URL and execute the request. The following shows the response in the browser to the GET request returned by the local function:



Now that you have run your function locally, you can create the function app and other required resources in Azure.

Create a resource group

Create a resource group with the `az group create`. An Azure resource group is a logical container into which Azure resources like function apps, databases, and storage accounts are deployed and managed.

The following example creates a resource group named `myResourceGroup`.

If you are not using Cloud Shell, sign in first using `az login`.

```
az group create --name myResourceGroup --location westeurope
```

You generally create your resource group and the resources in a `region` near you.

Create an Azure Storage account

Functions uses a general-purpose account in Azure Storage to maintain state and other information about your functions. Create a general-purpose storage account in the resource group you created by using the `az storage`

[account create](#) command.

In the following command, substitute a globally unique storage account name where you see the `<storage_name>` placeholder. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

```
az storage account create --name <storage_name> --location westeurope --resource-group myResourceGroup --sku Standard_LRS
```

Create a Linux function app in Azure

You must have a function app to host the execution of your functions on Linux. The function app provides a serverless environment for executing your function code. It lets you group functions as a logic unit for easier management, deployment, and sharing of resources. Create a function app running on Linux by using the [az functionapp create](#) command.

In the following command, use a unique function app name where you see the `<app_name>` placeholder and the storage account name for `<storage_name>`. The `<app_name>` is also the default DNS domain for the function app. This name needs to be unique across all apps in Azure. You should also set the `<language>` runtime for your function app, from `dotnet` (C#), `node` (JavaScript/TypeScript), or `python`.

```
az functionapp create --resource-group myResourceGroup --consumption-plan-location westus --os-type Linux \
--name <app_name> --storage-account <storage_name> --runtime <language>
```

After the function app has been created, you see the following message:

```
Your serverless Linux function app 'myfunctionapp' has been successfully created.  
To active this function app, publish your app content using Azure Functions Core Tools or the Azure portal.
```

Now, you can publish your project to the new function app in Azure.

Deploy the function app project to Azure

After the function app is created in Azure, you can use the `func azure functionapp publish` Core Tools command to deploy your project code to Azure. In the following command, replace `<APP_NAME>` with the name of your app from the previous step.

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

You'll see output similar to the following, which has been truncated for readability:

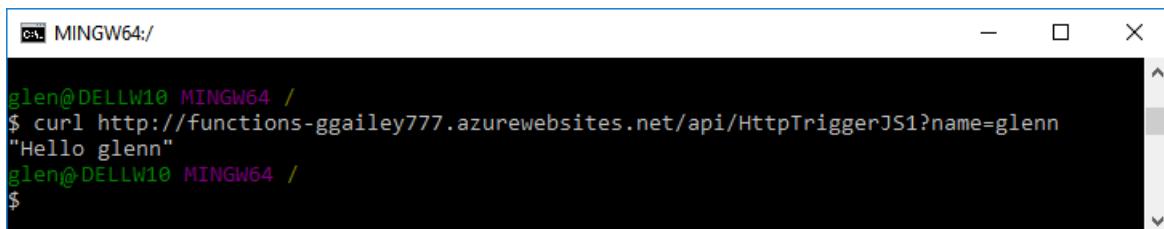
```
Getting site publishing info...
...
Preparing archive...
Uploading content...
Upload completed successfully.
Deployment completed successfully.
Syncing triggers...
Functions in myfunctionapp:
HttpTrigger - [httpTrigger]
    Invoke url: https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....
```

Copy the `Invoke url` value for your `HttpTrigger`, which you can now use to test your function in Azure. The URL contains a `code` query string value that is your function key. This key makes it difficult for others to call your HTTP trigger endpoint in Azure.

Test the function in Azure

Use cURL to test the deployed function. Using the URL that you copied from the previous step, append the query string `&name=<yourusername>` to the URL, as in the following example:

```
curl https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiov548FBDSL1....&name=<yourusername>
```



```
MINGW64:/  
glen@DELLW10 MINGW64 /  
$ curl http://functions-ggailey777.azurewebsites.net/api/HttpTriggerJS1?name=glen  
"Hello glenn"  
glen@DELLW10 MINGW64 /  
$
```

You can also paste the copied URL in to the address of your web browser. Again, append the query string `&name=<yourusername>` to the URL before you execute the request.



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

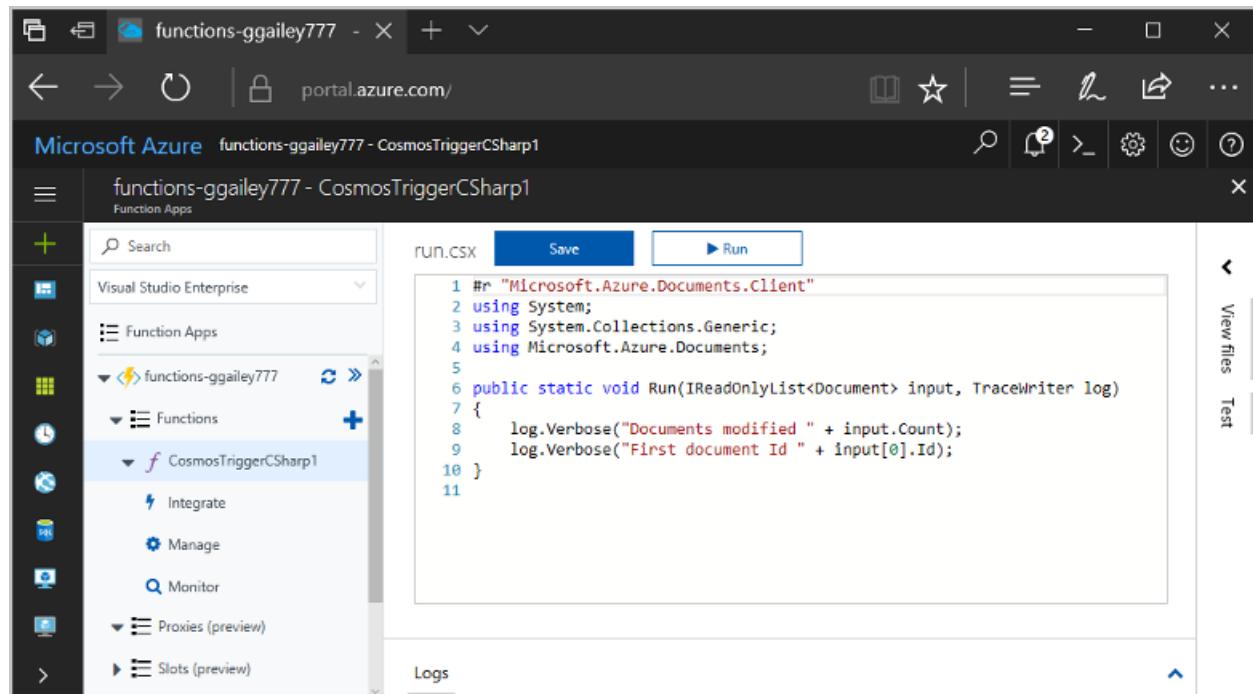
Learn more about developing Azure Functions locally using the Azure Functions Core Tools.

[Code and test Azure Functions locally](#)

Create a function triggered by Azure Cosmos DB

3/18/2019 • 8 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](#).



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](#).

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. Sign in to the [Azure portal](#).
2. Select **Create a resource > Databases > Azure Cosmos DB**.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons and links. At the top, there's a navigation bar with a search bar containing 'https://portal.azure.com/'. The main area is titled 'New' and shows the 'Azure Marketplace' with a 'Featured' tab selected. A search bar at the top of the marketplace lists says 'Search the Marketplace'. Below it, there are several service categories: Get started, Recently created, Compute, Networking, Storage, Web, Mobile, Containers, Databases, Analytics, AI + Machine Learning, Internet of Things, Mixed Reality, Integration, Security, Identity, and Developer Tools. The 'Databases' category is highlighted with a red box. To the right of each category is a small icon and a link to a 'Quickstart tutorial'. The 'Azure Cosmos DB' service is also highlighted with a red box.

3. 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	Enter a unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the ID that you provide to create your URI, use a unique ID.</p> <p>The ID can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	Core (SQL)	<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> <p>Select Core (SQL) to create a document database and query by using SQL syntax.</p> <p>Learn more about the SQL API.</p>
Location	Select 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.

Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [learn more](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription	Contoso Subscription	▼
* Resource Group	(New) myResourceGroup	▼
	Create new	

INSTANCE DETAILS

* Account Name	mysqlapicosmosdb	✓
	documents.azure.com	
* API ⓘ	Core (SQL)	▼
* Location	West US	▼
Geo-Redundancy ⓘ	Enable	Disable
Multi-region Writes ⓘ	Enable	Disable

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

5. 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 Microsoft Azure CosmosDB - Overview page for a deployment named "Microsoft.Azure.CosmosDB-20190321000000". The deployment status is shown as "Your deployment is complete" with a green checkmark icon. Below this, there is a summary of the deployment details: Deployment name: Microsoft.Azure.CosmosDB-20190321000000, Subscription: Contoso Subscription, and Resource group: myResourceGroup. The deployment started at 3/21/2019, 5:00:03 PM and took 5 minutes 38 seconds. The Correlation ID is 8e0be948-0c60-4da0-0000-000000000000. A table at the bottom lists the resource "mysqlapicosmosdb" with type Microsoft.DocumentDb/databaseAcc... and status OK, along with a link to "Operation details".

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

mysqlapicosmosdb - Quick start
Azure Cosmos DB account

Search (Ctrl+ /)

Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Quick start
Notifications
Data Explorer
Settings
Replicate data globally
Default consistency
Firewall and virtual networks

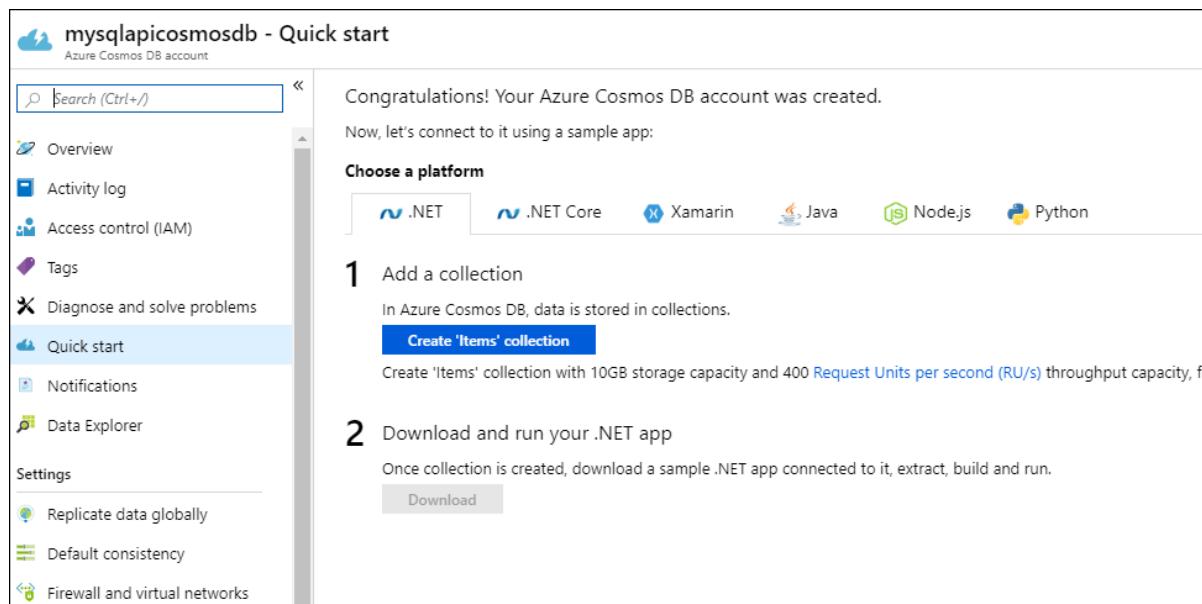
Congratulations! Your Azure Cosmos DB account was created.
Now, let's connect to it using a sample app:

Choose a platform

.NET .NET Core Xamarin Java Node.js Python

1 Add a collection
In Azure Cosmos DB, data is stored in collections.
Create 'Items' collection
Create 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, fo

2 Download and run your .NET app
Once collection is created, download a sample .NET app connected to it, extract, build and run.
Download



Create an Azure Function app

1. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.

Microsoft Azure

Create a resource

All services
FAVORITES
Dashboard
All resources
Resource groups
App Services
SQL databases
SQL data warehouses
Azure Cosmos DB
Virtual machines
Load balancers
Storage accounts
Virtual networks
Azure Active Directory
Monitor
Advisor
Security Center
Cost Management + Billing

Home > New

Search the Marketplace

Azure Marketplace See all
Get started
Recently created
Compute

Featured See all

Windows Server 2016 Datacenter
Quickstart tutorial

Red Hat Enterprise Linux 7.2
Learn more

Ubuntu Server 16.04 LTS
Quickstart tutorial

SQL Server 2017 Enterprise Windows Server 2016
Learn more

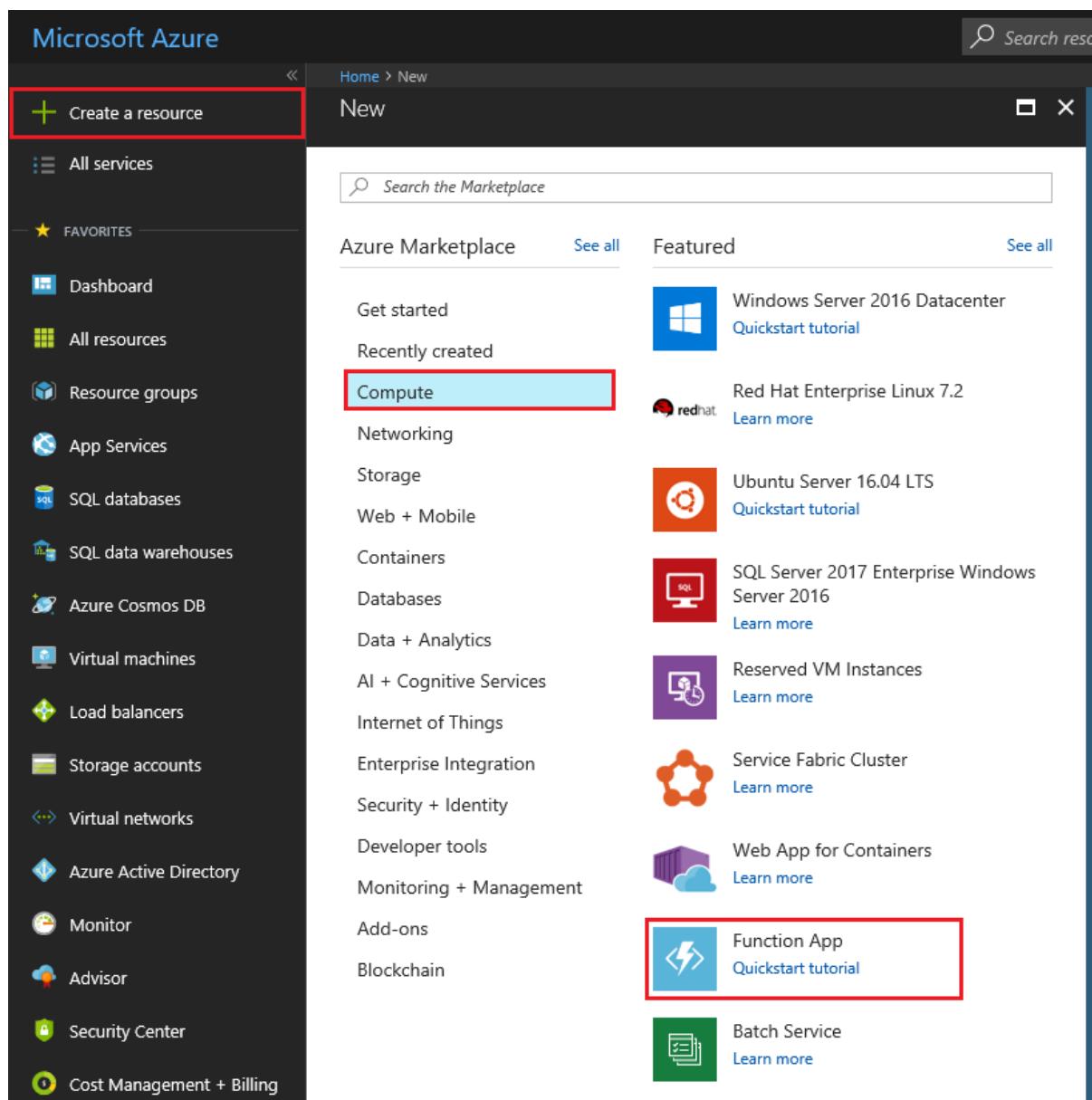
Reserved VM Instances
Learn more

Service Fabric Cluster
Learn more

Web App for Containers
Learn more

Function App
Quickstart tutorial

Batch Service
Learn more



2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
myfunctionapp .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
myResourceGroup

* OS
 Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
Central US

* Runtime Stack
.NET

* Storage ⓘ
 Create new Use existing
myfunctionapp99761

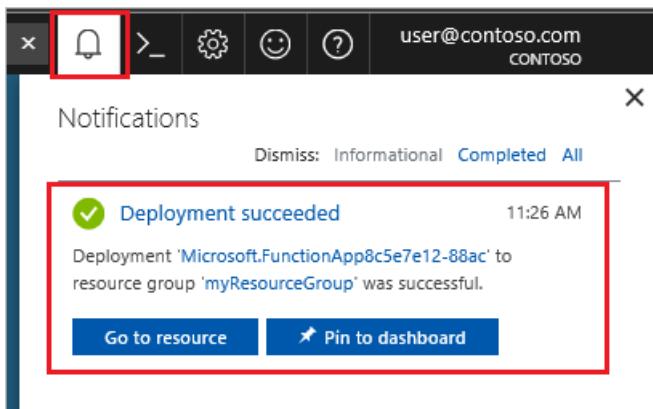
Application Insights >
myfunctionapp9

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Your subscription	The subscription under which this new function app is created.

Setting	Suggested Value	Description
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.

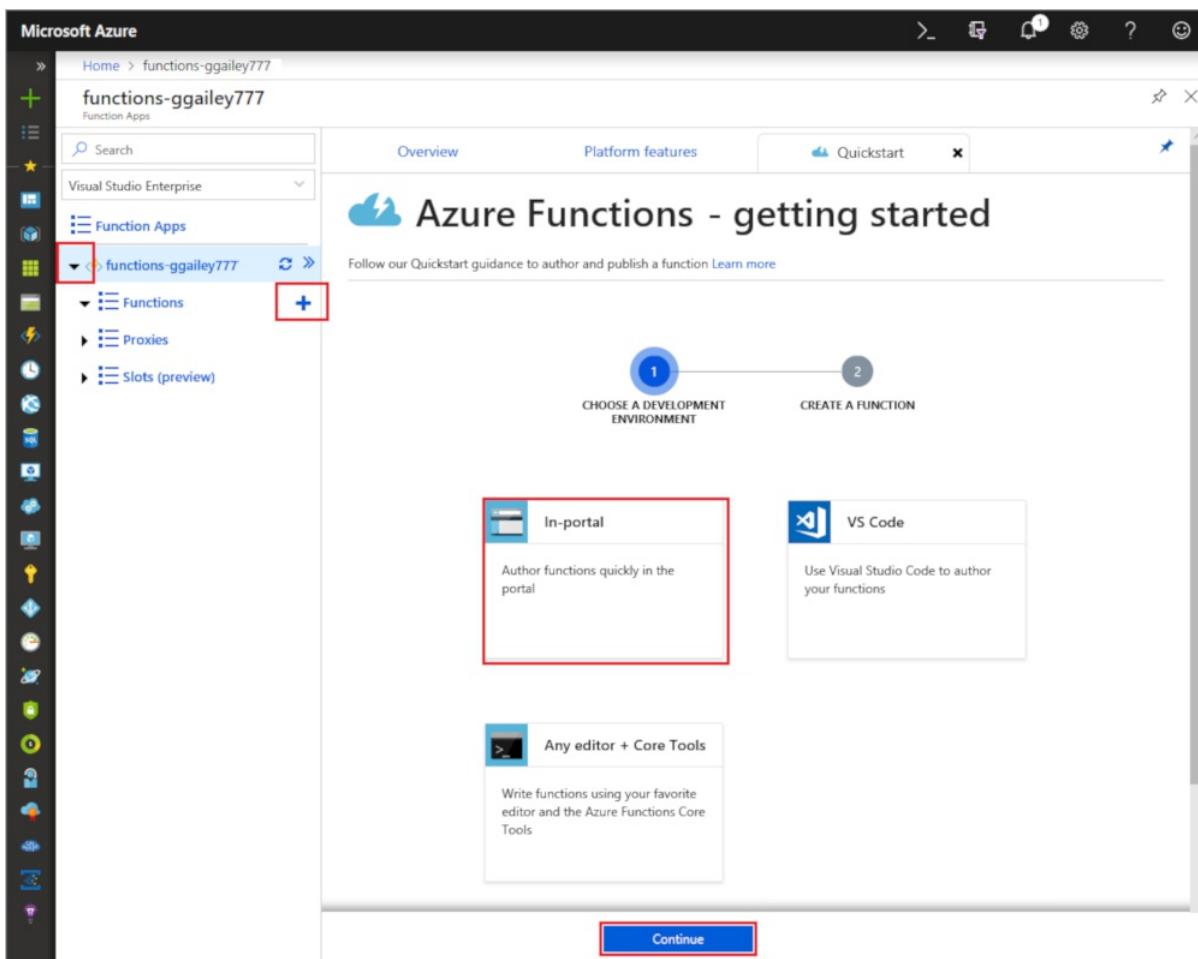


5. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

Next, you create a function in the new function app.

Create Azure Cosmos DB trigger

1. Expand your function app and click the + button next to **Functions**. If this is the first function in your function app, select **In-portal** then **Continue**. Otherwise, go to step three.



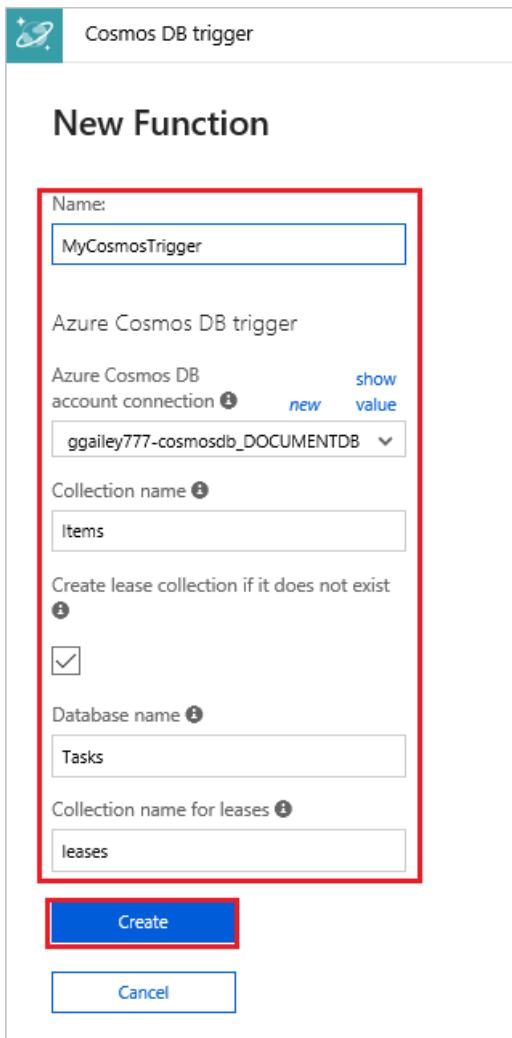
2. Choose **More templates** then **Finish and view templates**.

The screenshot shows the second step of the Azure Functions Getting Started wizard. At the top, there are three tabs: 'Overview', 'Platform features', and 'Quickstart'. Below the tabs, the title 'Azure Functions - getting started' is displayed with a cloud icon. A sub-instruction 'Follow our Quickstart guidance to author and publish a function' with a 'Learn more' link is present. The main area has two steps: 'CHOOSE A DEVELOPMENT ENVIRONMENT' (marked with a green checkmark) and 'CREATE A FUNCTION' (marked with a blue circle containing the number 2). Below these steps are three template cards: 'Webhook + API' (HTTP trigger), 'Timer' (Timer trigger), and 'More templates...'. The 'More templates...' card is highlighted with a red box. At the bottom, there are 'Back' and 'Finish and view templates' buttons, with 'Finish and view templates' also highlighted with a red box.

3. In the search field, type `cosmos` and then choose the **Azure Cosmos DB trigger** template.
4. If prompted, select **Install** to install the Azure Cosmos DB extension in the function app. After installation succeeds, select **Continue**.

The screenshot shows a 'Choose a template' dialog. At the top, it says 'Choose a template below or [go to the quickstart](#)'. A search bar contains the text 'cosmos'. On the right, a preview window for the 'Cosmos DB trigger' template is shown, featuring a teal icon with a white planet and the text 'Cosmos DB trigger'. Below the preview, a note says 'A function that will be run whenever documents are added to a document collection'. To the right of the preview, the text 'Extensions not Installed' is displayed, along with a note that the template requires the 'Microsoft.Azure.WebJobs.Extensions.CosmosDB' extension. There are 'Install' and 'Close' buttons, with 'Install' highlighted with a red box. A blue callout box at the bottom right provides a link to learn more about troubleshooting extension installation.

5. Configure the new trigger with the settings as specified in the table below the image.



SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Default	Use the default function name suggested by the template.
Azure Cosmos DB account connection	New setting	Select New , then choose your Subscription , the Database account you created earlier, and Select . This creates an application setting for your account connection. This setting is used by the binding to connection to the database.
Collection name	Items	Name of collection to be monitored.
Create lease collection if it doesn't exist	Checked	The collection doesn't already exist, so create it.
Database name	Tasks	Name of database with the collection to be monitored.

- Click **Create** to create your Azure Cosmos DB triggered function. After the function is created, the template-based function code is displayed.

The screenshot shows the Azure Functions portal interface. On the left, the sidebar lists 'Visual Studio Enterprise' and categories like 'Function Apps', 'Functions', and 'Proxies (preview)'. Under 'Functions', 'CosmosTriggerCSharp1' is selected. The main area displays the 'RUN.CSX' code:

```
1 #r "Microsoft.Azure.Documents.Client"
2 using System;
3 using System.Collections.Generic;
4 using Microsoft.Azure.Documents;
5
6 public static void Run(IReadOnlyList<Document> input, TraceWriter log)
7 {
8     log.Verbose("Documents modified " + input.Count);
9     log.Verbose("First document Id " + input[0].Id);
10 }
11
```

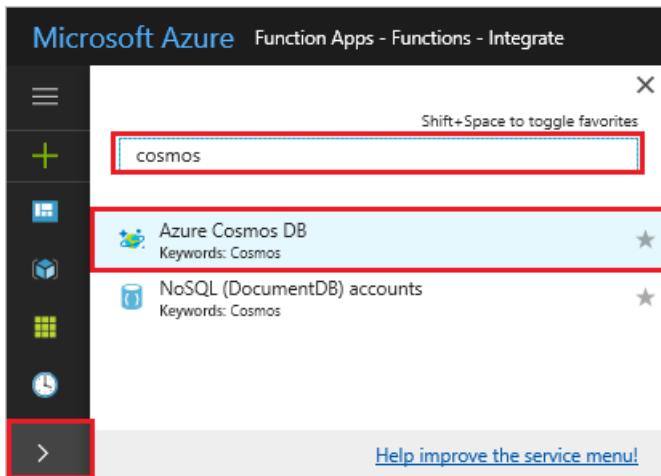
At the top right, there are 'Save' and 'Run' buttons. To the right of the code editor, there are 'View files' and 'Test' buttons.

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` collection in the `Tasks` database.

Create the Items collection

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. In **Collections**, choose **taskDatabase** and select **New Collection**.

The screenshot shows the Microsoft Azure Data Explorer (Preview) interface. At the top, the title bar reads "Microsoft Azure < ggailey777 - Data Explorer (Preview)". The main area displays a list of collections under the database "taskDatabase". A red box highlights the "New Collection" button in the top right corner of the interface.

5. In **Add Collection**, use the settings shown in the table below the image.

The screenshot shows the "Add Collection" dialog box. It contains the following fields:

- Database id**: Tasks (highlighted by a red box)
- Collection Id**: Items
- Storage capacity**: Fixed (10 GB) (highlighted by a red box)
- Throughput (400 - 10,000 RU/s)**: 400
- Partition key**: /category

At the bottom right of the dialog is a blue "OK" button, which is also highlighted by a red box.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	The name for your new database. This must match the name defined in your function binding.

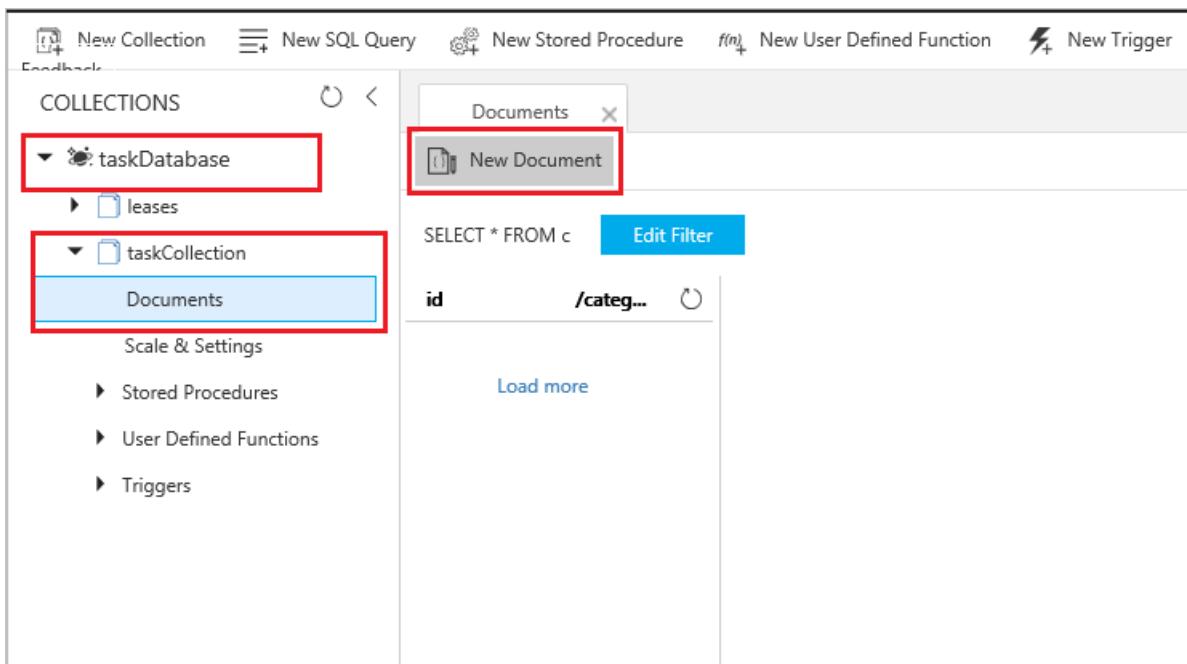
SETTING	SUGGESTED VALUE	DESCRIPTION
Collection ID	Items	The name for the new collection. This must match the name defined in your function binding.
Storage capacity	Fixed (10 GB)	Use the default value. This value is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you want to reduce latency, you can scale up the throughput later.
Partition key	/category	A partition key that distributes data evenly to each partition. Selecting the correct partition key is important in creating a performant collection.

- Click **OK** to create the Items collection. It may take a short time for the collection to get created.

After the collection specified in the function binding exists, you can test the function by adding documents to this new collection.

Test the function

- Expand the new **taskCollection** collection in Data Explorer, choose **Documents**, then select **New Document**.

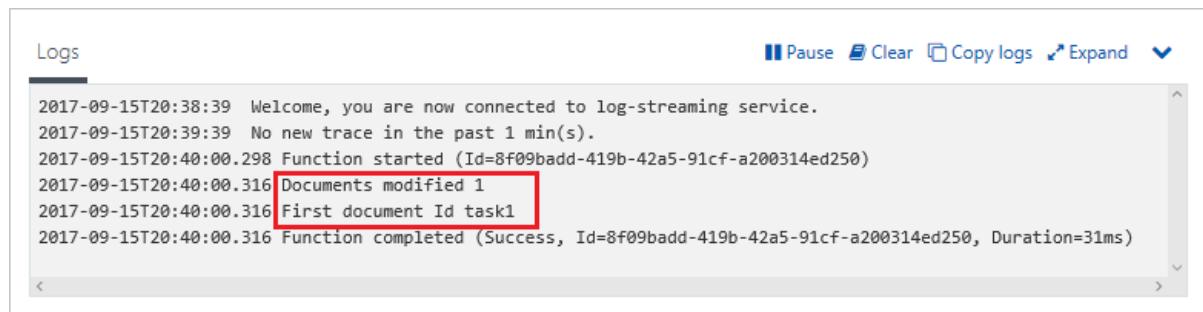


- Replace the contents of the new document with the following content, then choose **Save**.

```
{
  "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.



The screenshot shows the Azure Functions logs interface. At the top, there are buttons for Pause, Clear, Copy logs, and Expand. The log entries are listed below:

```
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)
```

The log entry "First document Id task1" is highlighted with a red box.

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

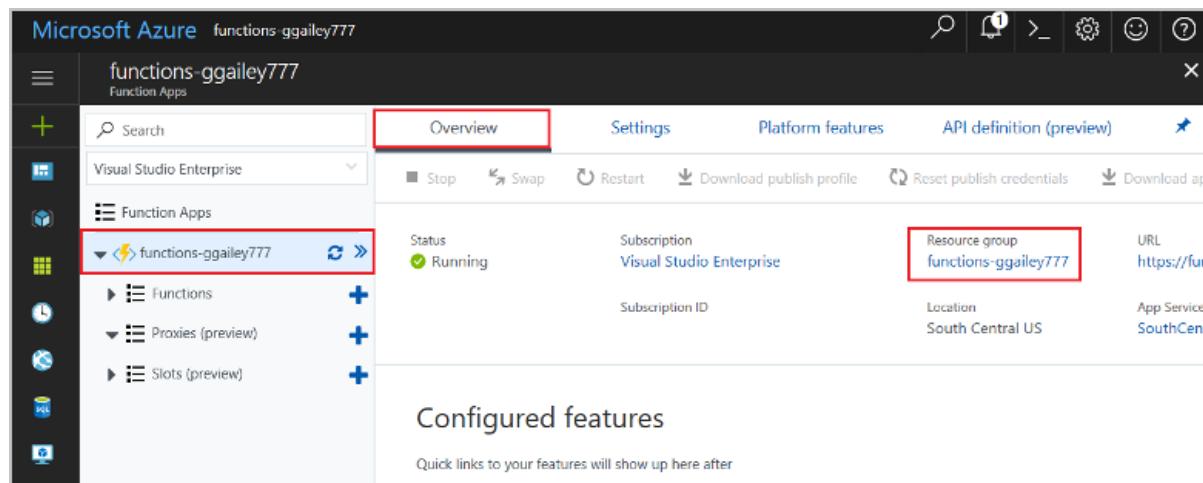
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 Microsoft Azure portal with the URL [functions-ggailey777](#). The left sidebar shows a list of resources under "Function Apps", with "functions-ggailey777" selected. The main area is the "Overview" tab, which displays the following information:

Resource Group	URL
functions-ggailey777	https://fun

Below this, there is a section titled "Configured features" with the note "Quick links to your features will show up here after".

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 are 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](#).

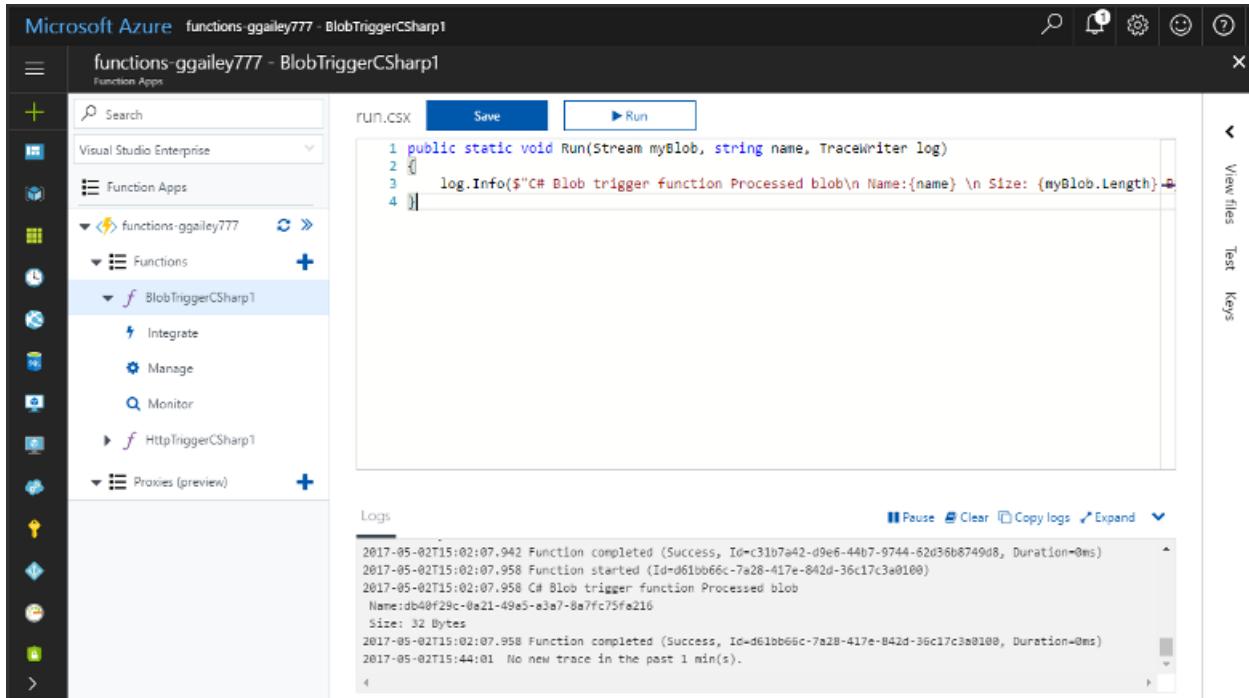
Now that you have created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Create a function triggered by Azure Blob storage

1/24/2019 • 5 minutes to read • [Edit Online](#)

Learn how to create a function triggered when files are uploaded to or updated in Azure Blob storage.



Prerequisites

- Download and install the [Microsoft Azure Storage Explorer](#).
- An Azure subscription. If you don't have one, create a [free account](#) before you begin.

Create an Azure Function app

1. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.

Microsoft Azure

Home > New

Create a resource

All services

FAVORITES

- Dashboard
- All resources
- Resource groups
- App Services
- SQL databases
- SQL data warehouses
- Azure Cosmos DB
- Virtual machines
- Load balancers
- Storage accounts
- Virtual networks
- Azure Active Directory
- Monitor
- Advisor
- Security Center
- Cost Management + Billing

New

Search the Marketplace

Azure Marketplace

Get started

Recently created

Compute

Networking

Storage

Web + Mobile

Containers

Databases

Data + Analytics

AI + Cognitive Services

Internet of Things

Enterprise Integration

Security + Identity

Developer tools

Monitoring + Management

Add-ons

Blockchain

Featured

See all

Windows Server 2016 Datacenter
Quickstart tutorial

Red Hat Enterprise Linux 7.2
Learn more

Ubuntu Server 16.04 LTS
Quickstart tutorial

SQL Server 2017 Enterprise Windows Server 2016
Learn more

Reserved VM Instances
Learn more

Service Fabric Cluster
Learn more

Web App for Containers
Learn more

Function App
Quickstart tutorial

Batch Service
Learn more

The screenshot shows the Microsoft Azure 'New' blade. On the left is a sidebar with various service icons and names. At the top right is a search bar. Below the search bar is a 'Create a resource' button, which is highlighted with a red box. The main area is titled 'New' and contains a 'Search the Marketplace' input field. Below it are two tabs: 'Azure Marketplace' and 'Featured'. Under 'Azure Marketplace', there's a 'Compute' category, which is also highlighted with a red box. Other categories listed include Get started, Recently created, Networking, Storage, Web + Mobile, Containers, Databases, Data + Analytics, AI + Cognitive Services, Internet of Things, Enterprise Integration, Security + Identity, Developer tools, Monitoring + Management, Add-ons, and Blockchain. To the right of these categories are several featured items, each with a thumbnail icon, a title, and a 'Quickstart tutorial' link.

2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
myfunctionapp ✓ .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
myResourceGroup ✓

* OS
Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
Central US

* Runtime Stack
.NET

* Storage ⓘ
 Create new Use existing
myfunctionapp99761 ✓

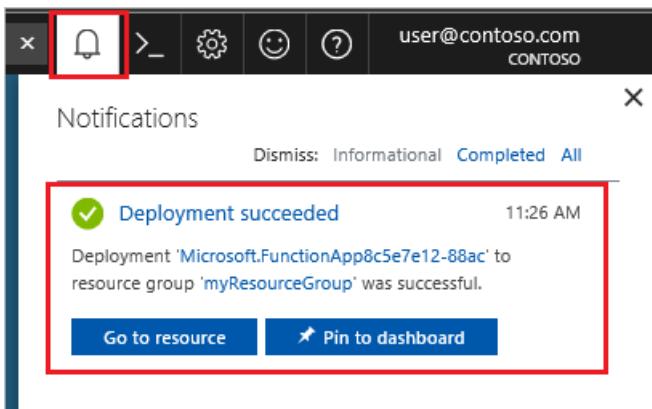
Application Insights >
myfunctionapp9

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z, 0-9, and -.
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



5. 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.

A screenshot of the Microsoft Azure Function App overview page for the app 'functions-ggalley777'. The left sidebar shows a tree view with 'Visual Studio Enterprise' selected under 'Function Apps', and 'functions-ggalley777' is expanded to show 'Functions' and 'Proxies (preview)'. The main pane has tabs for 'Overview', 'Settings', 'Platform features', and 'API definition (preview)'. The 'Overview' tab is active, showing basic details: Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggalley777), Location (South Central US), and URL (https://functions-ggalley777.azurewebsites.net). Below this, there's a section titled 'Configured features' with a note: 'Quick links to your features will show up here after you've configured them from the "Platform features" tab above.'

Next, you create a function in the new function app.

Create a Blob storage triggered function

1. Expand your function app and click the + button next to **Functions**. If this is the first function in your function app, select **In-portal** then **Continue**. Otherwise, go to step three.

Microsoft Azure

Home > functions-ggailey777

functions-ggailey777
Function Apps

Search
Visual Studio Enterprise

Function Apps

functions-ggailey777 **+**

Functions Proxies Slots (preview)

Overview Platform features Quickstart

Azure Functions - getting started

Follow our Quickstart guidance to author and publish a function [Learn more](#)

1 CHOOSE A DEVELOPMENT ENVIRONMENT 2 CREATE A FUNCTION

In-portal Author functions quickly in the portal

VS Code Use Visual Studio Code to author your functions

Any editor + Core Tools Write functions using your favorite editor and the Azure Functions Core Tools

Continue

2. Choose **More templates** then **Finish and view templates**.

Overview Platform features Quickstart

Azure Functions - getting started

Follow our Quickstart guidance to author and publish a function [Learn more](#)

CHOOSE A DEVELOPMENT ENVIRONMENT CREATE A FUNCTION

Webhook + API A function that will be run whenever it receives an HTTP request

Timer A function that will be run on a specified schedule

More templates... View all templates available to this function app

Back Finish and view templates

3. In the search field, type **blob** and then choose the **Blob trigger** template.
4. If prompted, select **Install** to install the Azure Storage extension any dependencies in the function app. After installation succeeds, select **Continue**.

Choose a template below or [go to the quickstart](#)

blob Scenario: All

Blob trigger

A function that will be run whenever a blob is added to a specified container

Extensions not Installed

This template requires the following extensions.

Not finding Serverless C

Microsoft.Azure.WebJobs.Extensions.Storage

Install (button highlighted with a red box)

Close

[Click here to learn more about troubleshooting extension installation](#)

5. Use the settings as specified in the table below the image.

Blob trigger

New Function

Name:

Azure Blob Storage trigger

Path ?

Storage account connection ? new value
AzureWebJobsStorage

Create (button highlighted with a red box)

Cancel

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	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.

6. Click **Create** to create your function.

Next, you connect to your Azure Storage account and create the **samples-workitems** container.

Create the container

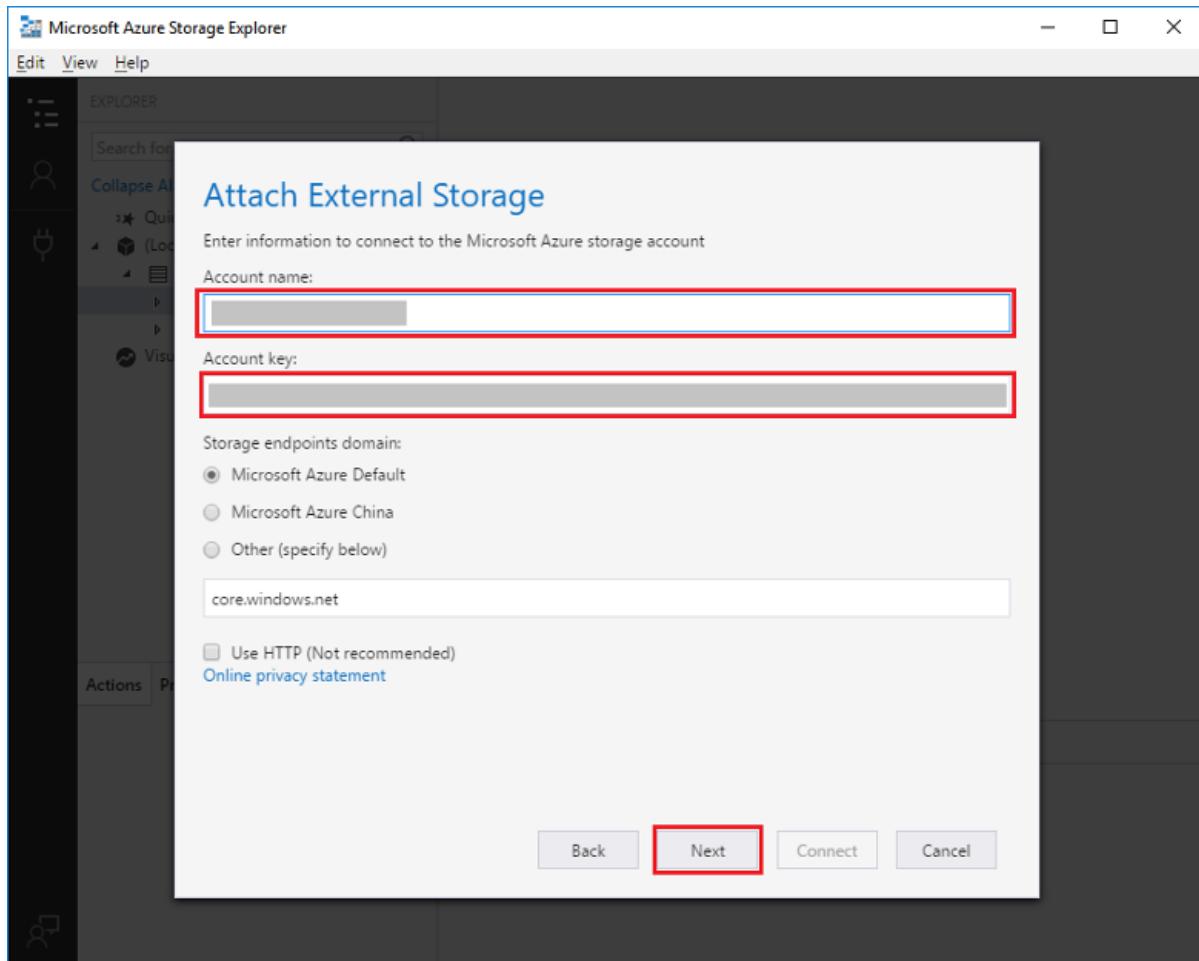
1. In your function, click **Integrate**, expand **Documentation**, and copy both **Account name** and **Account key**. You use these credentials to connect to the storage account. If you have already connected your storage account, skip to step 4.

The screenshot shows the Azure Functions portal for the function app 'functions-ggailey777 - MyBlobTrigger'. On the left, the navigation menu is visible with 'Integrate' and 'Documentation' highlighted by red boxes. The main area displays an 'Azure Blob Storage trigger' configuration. It includes fields for 'Blob parameter name' (set to 'myBlob'), 'Path' (set to 'samples-workitems/{name}'), and 'Storage account connection' (set to 'AzureWebJobsStorage'). A 'Documentation' link is also highlighted with a red box. At the bottom, there's a section for connecting to a storage account, showing 'Account Name' ('myfunctionstorageaccount') and 'Account Key' (redacted), along with a 'Connection String' field.

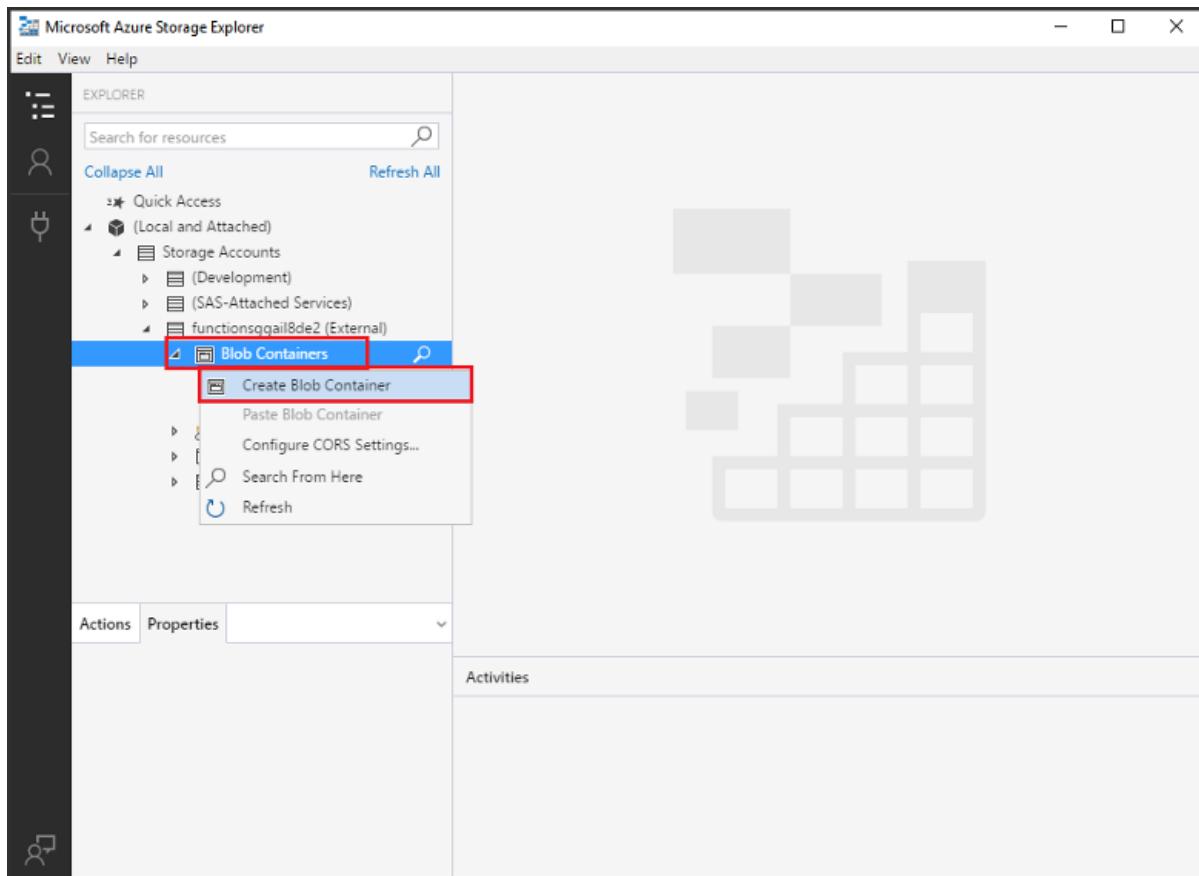
2. Run the **Microsoft Azure Storage Explorer** tool, click the connect icon on the left, choose **Use a storage account name and key**, and click **Next**.

The screenshot shows the 'Microsoft Azure Storage Explorer' application. On the left, the sidebar has a 'Connect' icon highlighted with a red box. The main window displays a 'Connect to Azure Storage' dialog. It asks 'How do you want to connect to your Storage Account or service?' with two options: 'Add an Azure Account' (radio button) and 'Use a shared access signature (SAS) URI or connection string' (radio button). Below these is a third option, 'Use a storage account name and key' (radio button), which is also highlighted with a red box. At the bottom of the dialog are 'Next', 'Connect', and 'Cancel' buttons, with 'Next' highlighted with a red box.

3. Enter the **Account name** and **Account key** from step 1, click **Next** and then **Connect**.



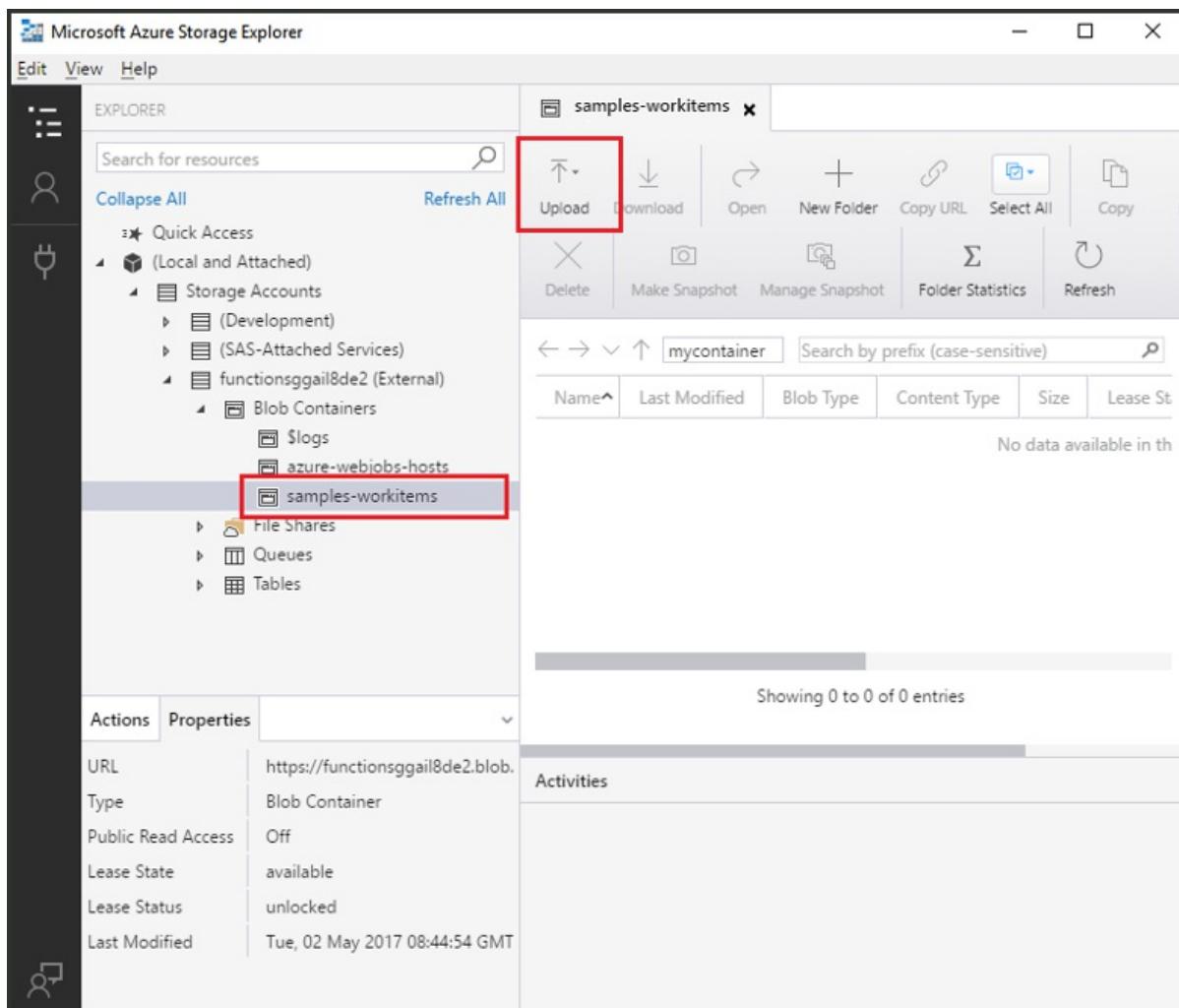
4. Expand the attached storage account, right-click **Blob containers**, click **Create blob container**, type `samples-workitems`, and then press enter.



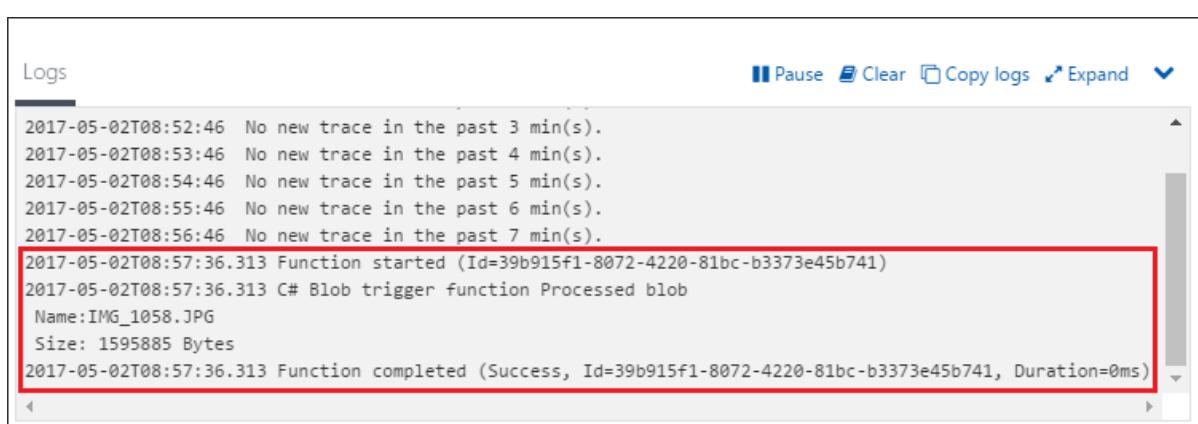
Now that you have a blob container, you can test the function by uploading a file to the container.

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.
2. In Storage Explorer, expand your storage account, **Blob containers**, and **samples-workitems**. Click **Upload** and then **Upload files....**



3. In the **Upload files** dialog box, click the **Files** field. Browse to a file on your local computer, such as an image file, select it and click **Open** and then **Upload**.
4. Go back to your function logs and verify that the blob has been read.



NOTE

When your function app runs in the default Consumption plan, there may be a delay of up to several minutes between the blob being added or updated and the function being triggered. If you need low latency in your blob triggered functions, consider running your function app in an App Service plan.

Clean up resources

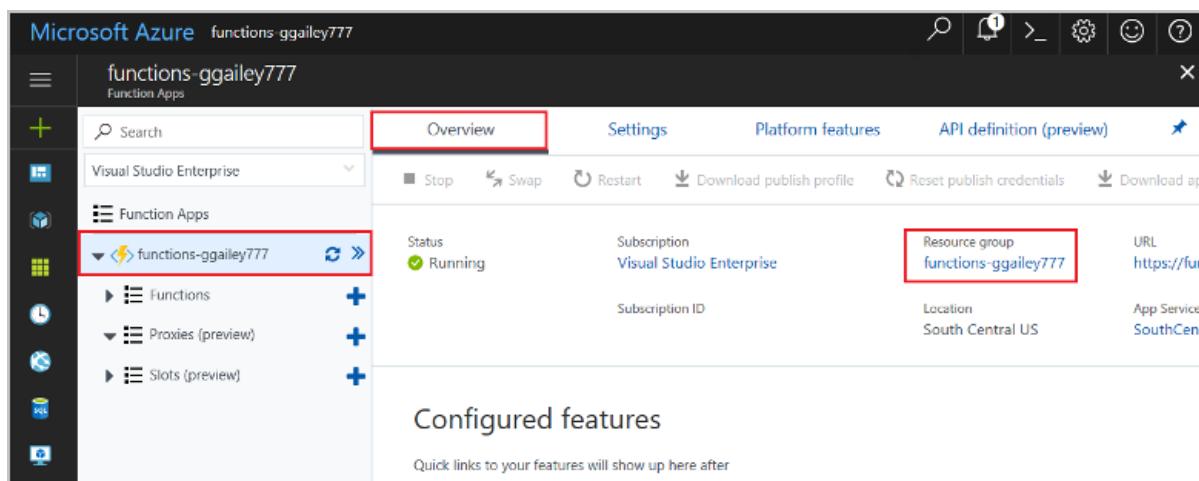
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 'functions-ggailey777'. The 'Overview' tab is selected. In the top right, there is a section labeled 'Resource group' containing the text 'functions-ggailey777'. This entire section is highlighted with a red box. Below this, there are other details like 'Status: Running', 'Subscription: Visual Studio Enterprise', and 'URL: https://fun...'. On the left sidebar, there is a tree view under 'Function Apps' with 'functions-ggailey777' expanded, showing 'Functions', 'Proxies (preview)', and 'Slots (preview)'. These items are also highlighted with a red box.

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 are 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](#).

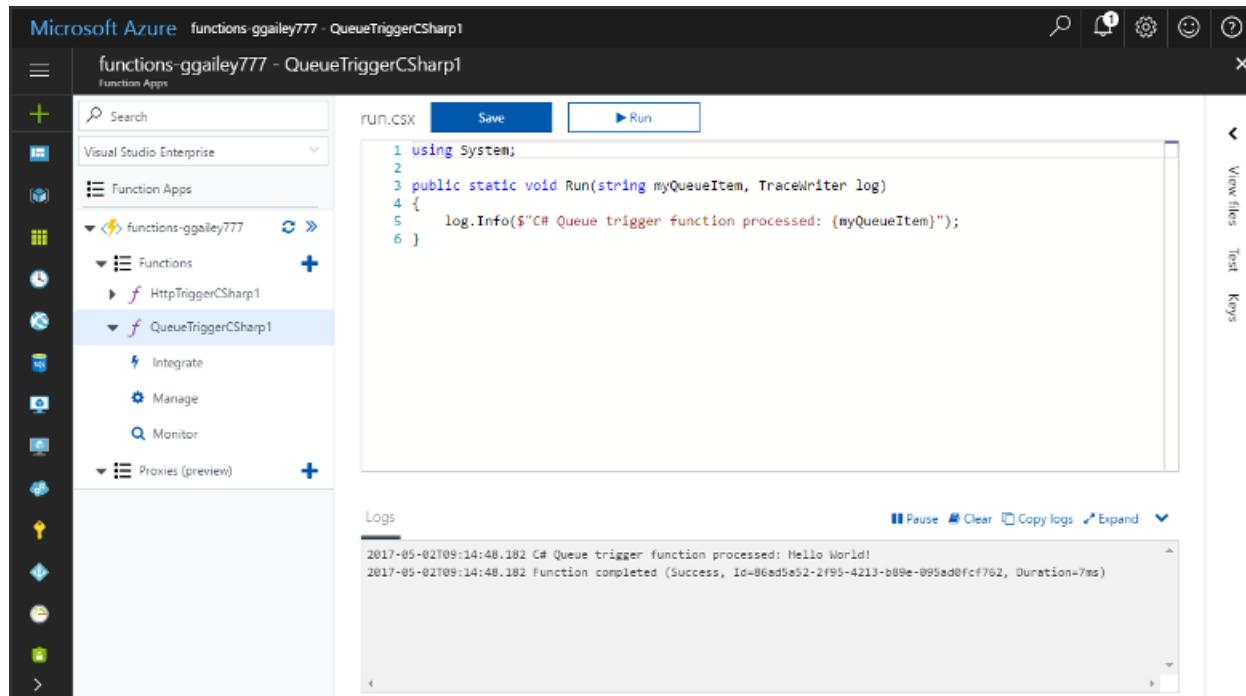
Now that you have created your first function, let's add an output binding to the function that writes a message to a Storage queue.

[Add messages to an Azure Storage queue using Functions](#)

Create a function triggered by Azure Queue storage

1/24/2019 • 5 minutes to read • [Edit Online](#)

Learn how to create a function that is triggered when messages are submitted to an Azure Storage queue.

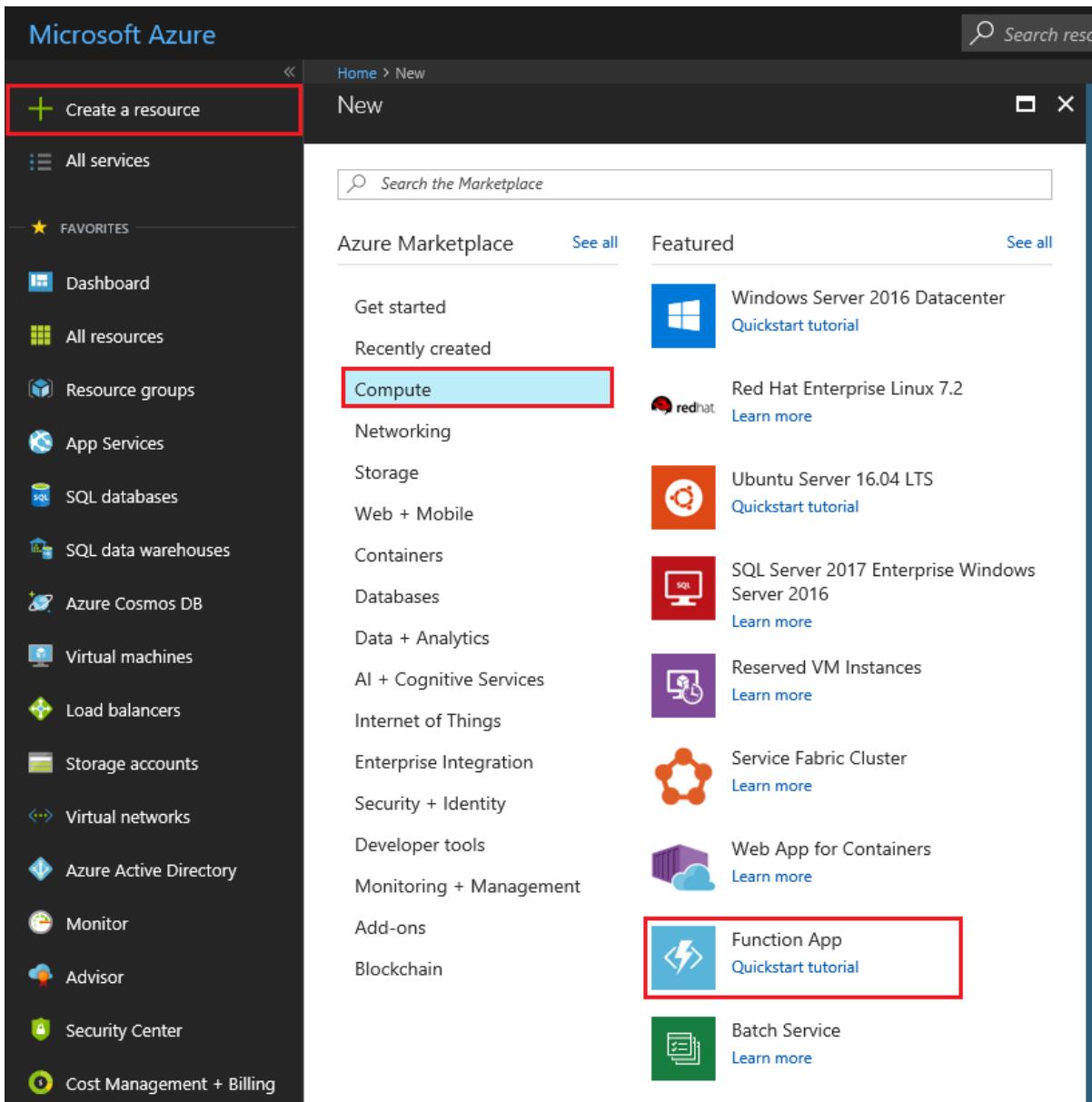


Prerequisites

- Download and install the [Microsoft Azure Storage Explorer](#).
- An Azure subscription. If you don't have one, create a [free account](#) before you begin.

Create an Azure Function app

1. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.



2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
myfunctionapp .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
myResourceGroup ✓

* OS
Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
Central US

* Runtime Stack
.NET

* Storage ⓘ
 Create new Use existing
myfunctionapp99761 ✓

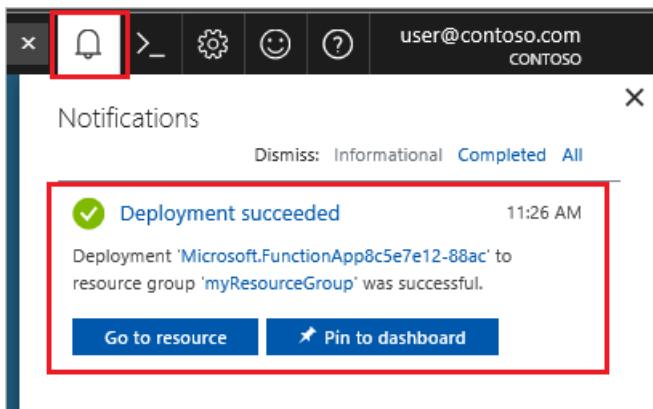
Application Insights >
myfunctionapp9

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z , 0-9 , and - .
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



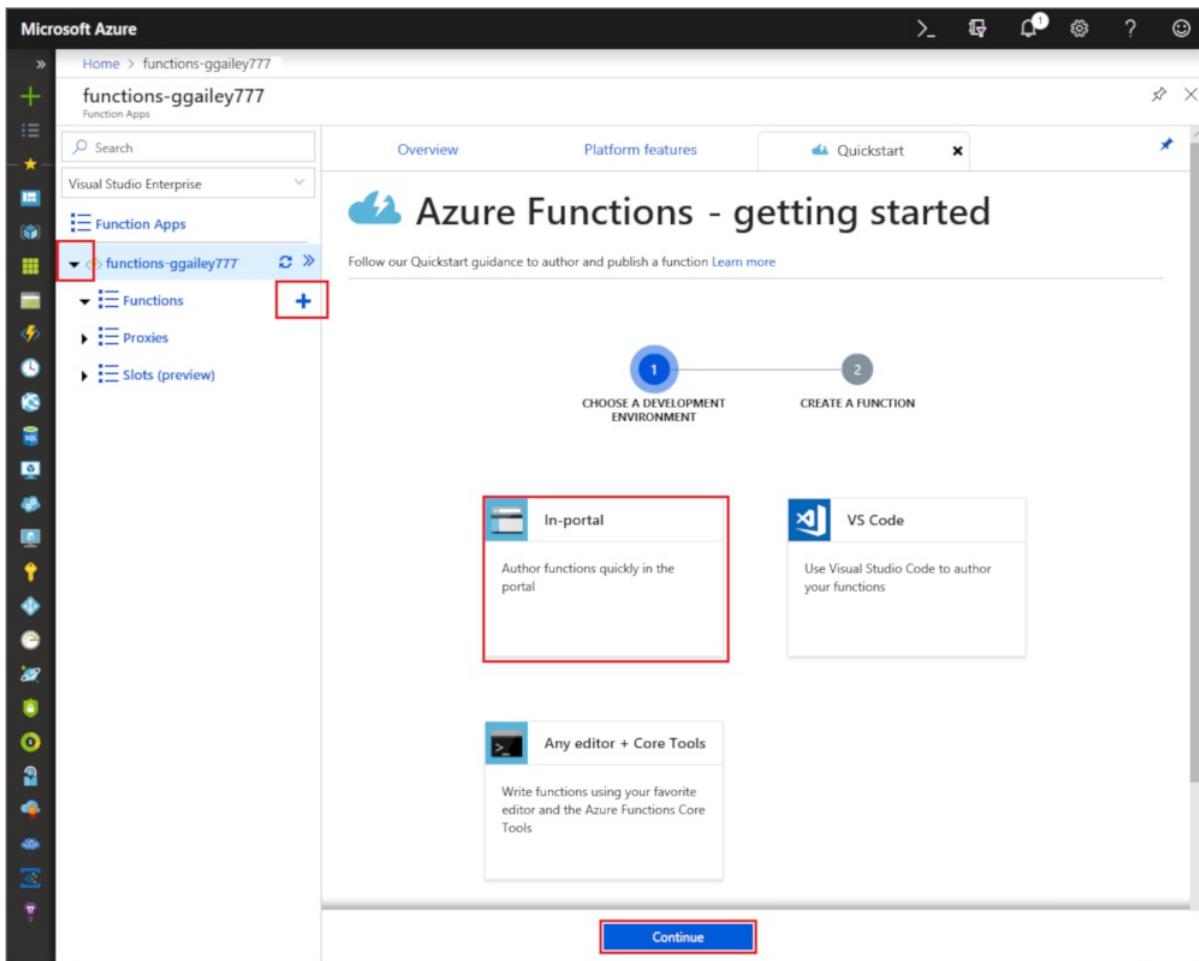
5. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

A screenshot of the Microsoft Azure Function App overview page for the app 'functions-ggaley777'. The left sidebar shows a tree view with 'Visual Studio Enterprise' selected under 'Function Apps', and 'functions-ggaley777' is expanded to show 'Functions' and 'Proxies (preview)'. The main content area has tabs for 'Overview', 'Settings', 'Platform features', and 'API definition (preview)', with 'Overview' selected. Under 'Overview', there are sections for 'Status' (Running), 'Subscription' (Visual Studio Enterprise), 'Resource group' (functions-ggaley777), 'URL' (https://functions-ggaley777.azurewebsites.net), 'Location' (South Central US), and 'App Service plan / pricing tier' (SouthCentralUSPlan (Consumption)). Below this, a section titled 'Configured features' contains the note: 'Quick links to your features will show up here after you've configured them from the "Platform features" tab above.'

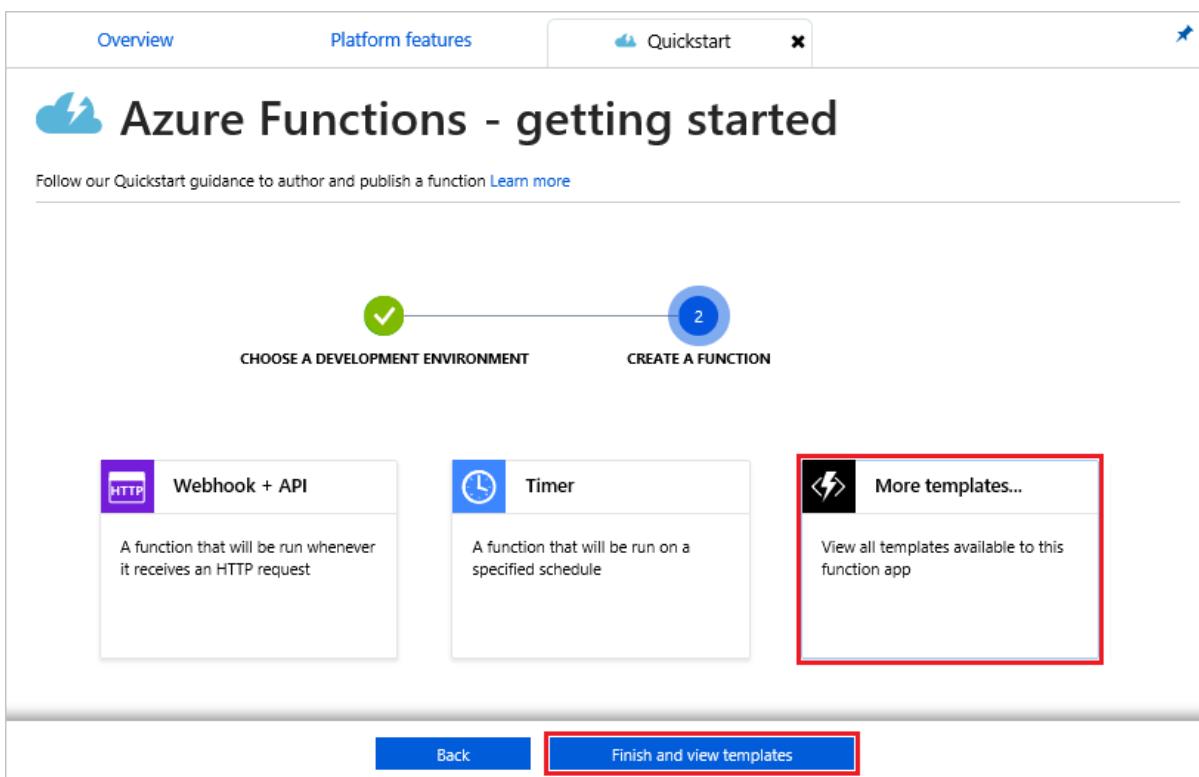
Next, you create a function in the new function app.

Create a Queue triggered function

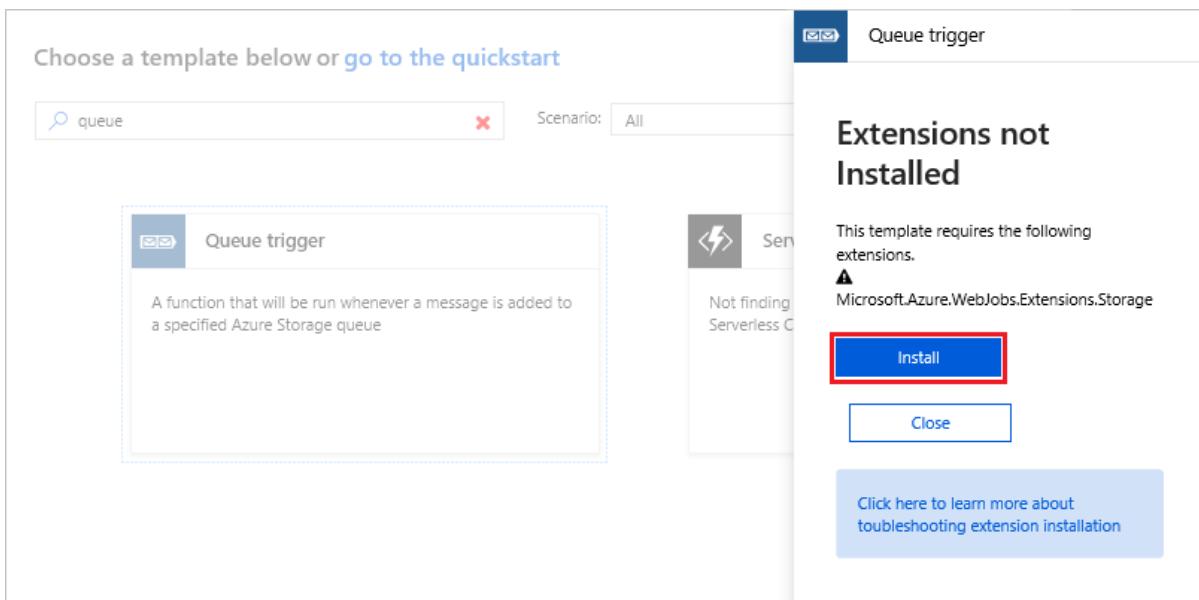
1. Expand your function app and click the + button next to **Functions**. If this is the first function in your function app, select **In-portal** then **Continue**. Otherwise, go to step three.



2. Choose **More templates** then **Finish and view templates**.



3. In the search field, type `queue` and then choose the **Queue trigger** template.
4. If prompted, select **Install** to install the Azure Storage extension any dependencies in the function app. After installation succeeds, select **Continue**.



5. Use the settings as specified in the table below the image.

New Function

Name: MyQueueTrigger

Azure Queue Storage trigger

Queue name: myqueue-items

Storage account connection: AzureWebJobsStorage

Create

Cancel

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	AzureWebJobStorage	You can use the storage account connection already being used by your function app, or create a new one.

6. Click **Create** to create your function.

Next, you connect to your Azure Storage account and create the **myqueue-items** storage queue.

Create the queue

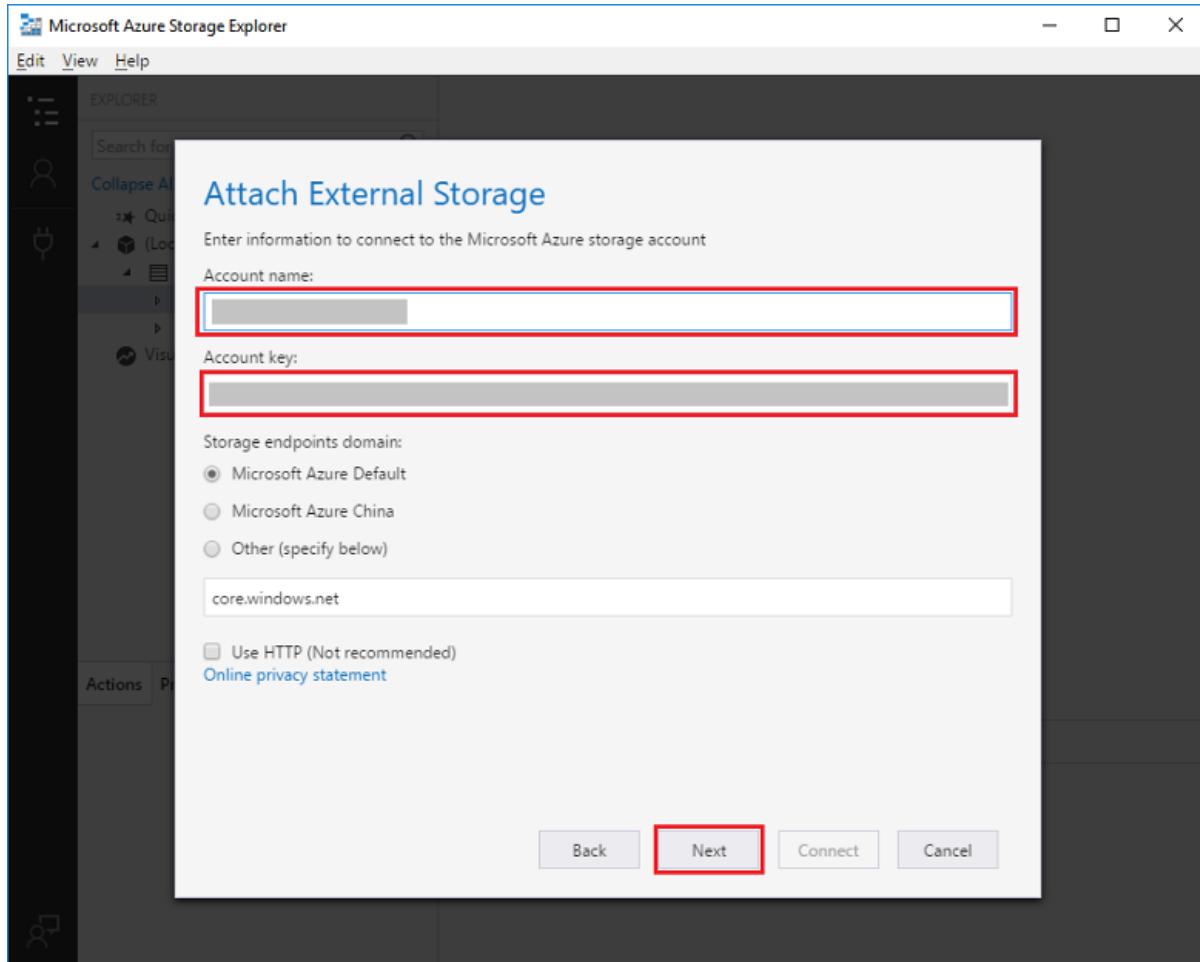
1. In your function, click **Integrate**, expand **Documentation**, and copy both **Account name** and **Account key**. You use these credentials to connect to the storage account in Azure Storage Explorer. If you have already connected your storage account, skip to step 4.

The screenshot shows the Azure Functions portal for the function app 'functions-ggailey777'. On the left, the navigation menu is visible with options like 'Visual Studio Enterprise', 'Function Apps', 'Functions', 'MyQueueTrigger', 'Integrate' (which is highlighted with a red box), 'Manage', 'Monitor', 'Proxies', and 'Slots (preview)'. Under 'MyQueueTrigger', there is an 'Azure Queue Storage trigger' configuration. It includes fields for 'Message parameter name' (set to 'myQueueItem'), 'Queue name' (set to 'myqueue-items'), 'Storage account connection' (set to 'AzureWebJobsStorage'), and a 'Documentation' section. The 'Documentation' section contains instructions to download Storage Explorer from <http://storageexplorer.com> and connects using 'Account Name' (set to 'myfunctionstorageaccount') and 'Account Key' (redacted). A 'Connection String' field also contains a redacted value.

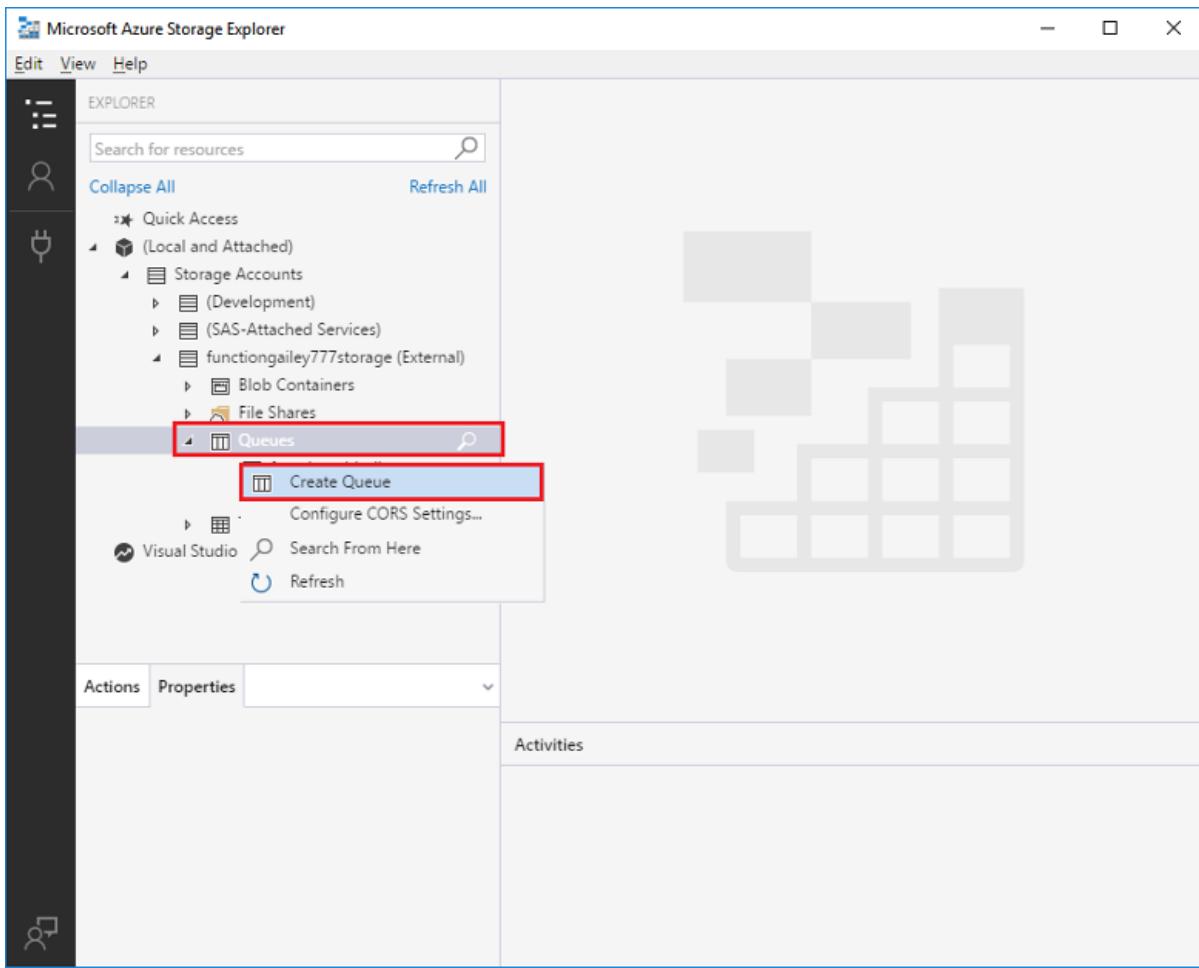
2. Run the **Microsoft Azure Storage Explorer** tool, click the connect icon on the left, choose **Use a storage account name and key**, and click **Next**.

The screenshot shows the 'Microsoft Azure Storage Explorer' application. On the left, there is a sidebar with icons for 'File', 'Edit', 'View', 'Help', and a 'Connect' icon (which is highlighted with a red box). The main window displays a 'Connect to Azure Storage' dialog. It asks 'How do you want to connect to your Storage Account or service?'. There are two radio button options: 'Add an Azure Account' (unchecked) and 'Use a shared access signature (SAS) URI or connection string' (unchecked). Below these is a checked radio button 'Use a storage account name and key' (highlighted with a red box). At the bottom of the dialog are three buttons: 'Next' (highlighted with a red box), 'Connect', and 'Cancel'.

3. Enter the **Account name** and **Account key** from step 1, click **Next** and then **Connect**.



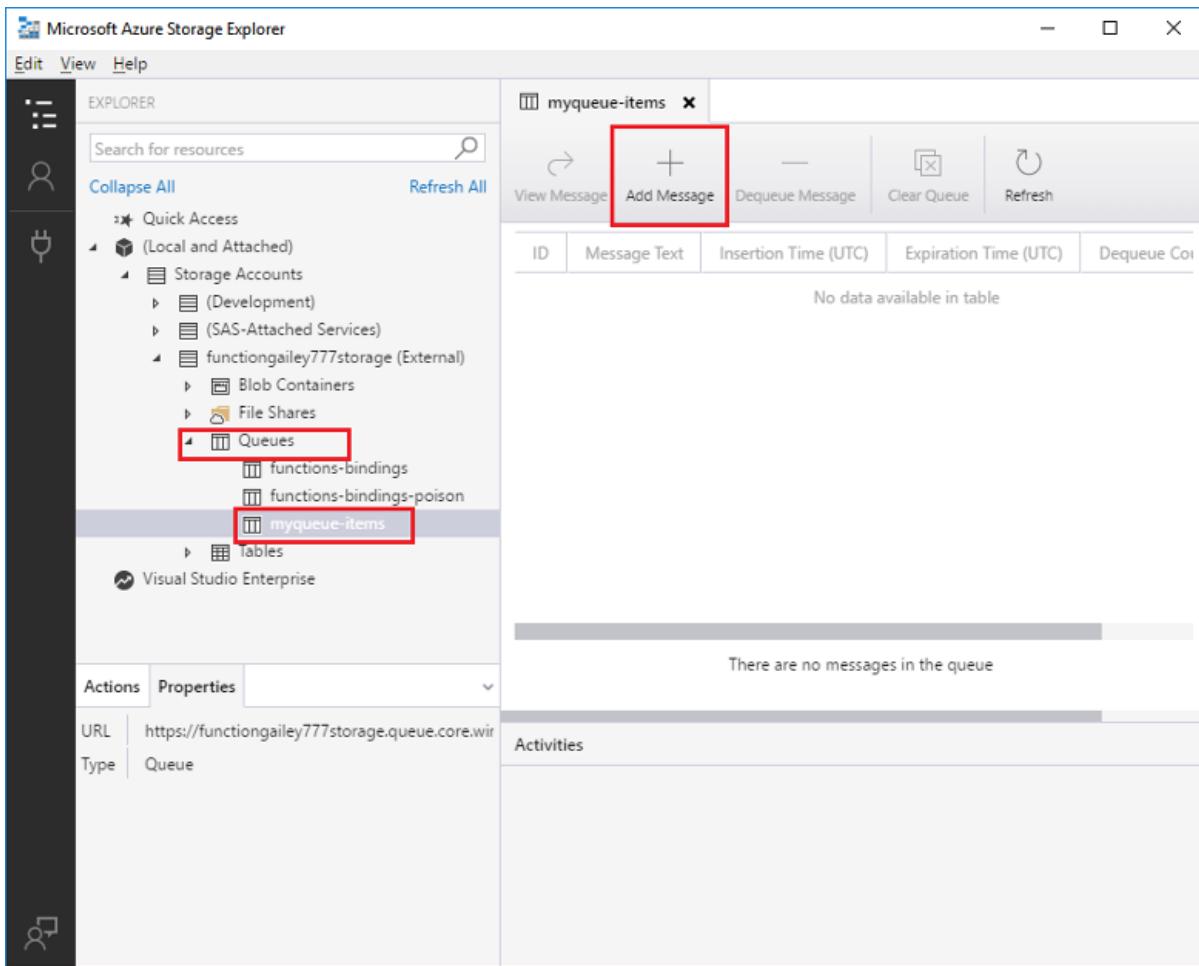
4. Expand the attached storage account, right-click **Queues**, click **Create Queue**, type `myqueue-items`, and then press enter.



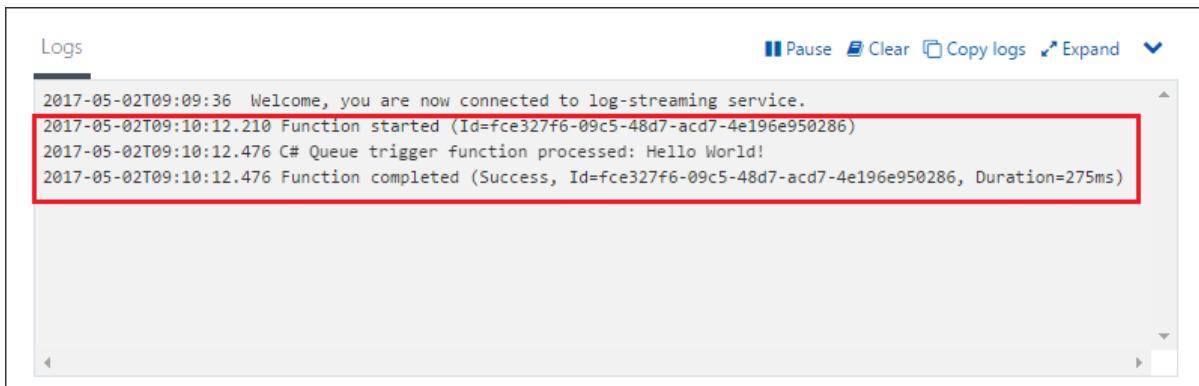
Now that you have a storage queue, you can test the function by adding a message to the queue.

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.
2. In Storage Explorer, expand your storage account, **Queues**, and **myqueue-items**, then click **Add message**.



3. Type your "Hello World!" message in **Message text** and click **OK**.
4. Wait for a few seconds, then go back to your function logs and verify that the new message has been read from the queue.



5. Back in Storage Explorer, click **Refresh** and verify that the message has been processed and is no longer in the queue.

Clean up resources

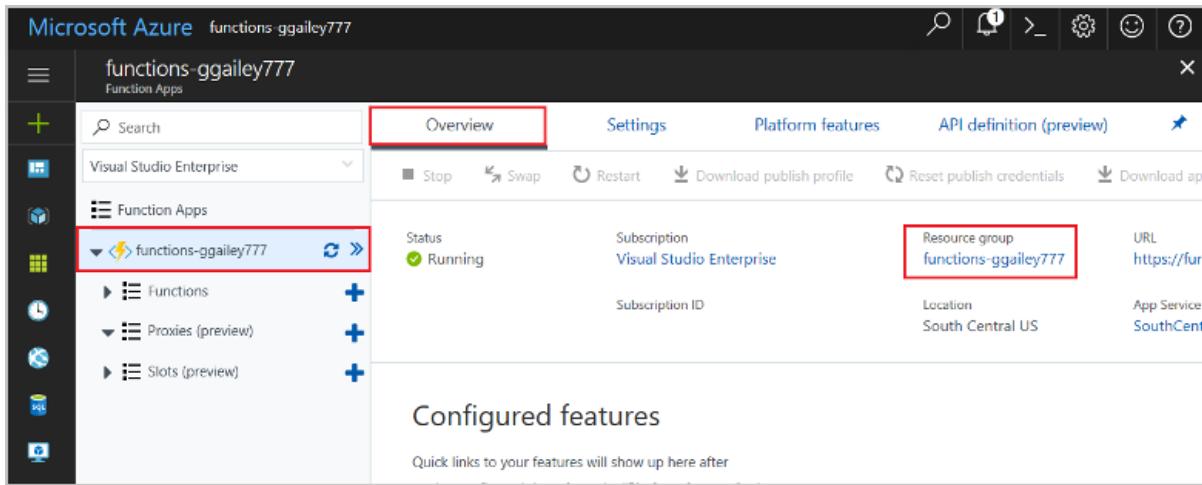
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 'functions-ggailey777'. The 'Overview' tab is selected. On the left sidebar, under 'Function Apps', the 'functions-ggailey777' item is expanded, and its 'Resource group' link is highlighted with a red box. The main content area displays basic information such as status (Running), subscription (Visual Studio Enterprise), location (South Central US), and URL (https://fun). Below this, there's a section titled 'Configured features' with a note: 'Quick links to your features will show up here after'.

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 are 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](#).

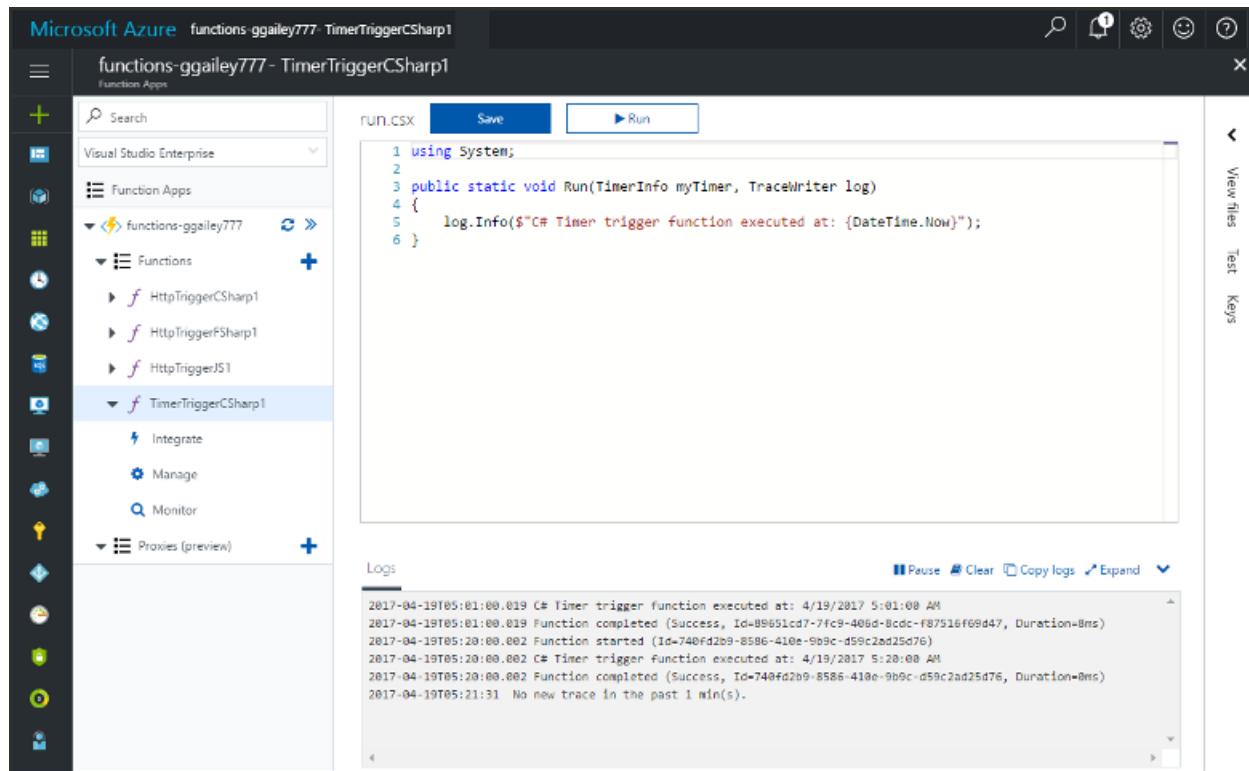
Now that you have a created your first function, let's add an output binding to the function that writes a message back to another queue.

[Add messages to an Azure Storage queue using Functions](#)

Create a function in Azure that is triggered by a timer

1/24/2019 • 4 minutes to read • [Edit Online](#)

Learn how to use Azure Functions to create a [serverless](#) function that runs based 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. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.

Microsoft Azure

Home > New

Create a resource

All services

FAVORITES

- Dashboard
- All resources
- Resource groups
- App Services
- SQL databases
- SQL data warehouses
- Azure Cosmos DB
- Virtual machines
- Load balancers
- Storage accounts
- Virtual networks
- Azure Active Directory
- Monitor
- Advisor
- Security Center
- Cost Management + Billing

New

Search the Marketplace

Azure Marketplace See all

Get started Recently created **Compute**

Networking Storage Web + Mobile Containers Databases Data + Analytics AI + Cognitive Services Internet of Things Enterprise Integration Security + Identity Developer tools Monitoring + Management Add-ons Blockchain

Featured See all

Windows Server 2016 Datacenter Quickstart tutorial

Red Hat Enterprise Linux 7.2 Learn more

Ubuntu Server 16.04 LTS Quickstart tutorial

SQL Server 2017 Enterprise Windows Server 2016 Learn more

Reserved VM Instances Learn more

Service Fabric Cluster Learn more

Web App for Containers Learn more

Function App Quickstart tutorial

Batch Service Learn more

The screenshot shows the Microsoft Azure 'New' blade. On the left is a sidebar with various service icons and names. At the top right is a search bar. Below the search bar is a 'Create a resource' button, which is highlighted with a red box. The main area is titled 'New' and contains two tabs: 'Azure Marketplace' and 'Featured'. Under 'Azure Marketplace', there's a 'See all' link, followed by a list of categories: Get started, Recently created, Compute (which is highlighted with a blue box), Networking, Storage, Web + Mobile, Containers, Databases, Data + Analytics, AI + Cognitive Services, Internet of Things, Enterprise Integration, Security + Identity, Developer tools, Monitoring + Management, Add-ons, and Blockchain. To the right of these categories are cards for different services, each with an icon, a title, and a 'Quickstart tutorial' link. One card for 'Function App' is also highlighted with a red box.

2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
myfunctionapp .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group i
 Create new Use existing
myResourceGroup

* OS
 Windows Linux

* Hosting Plan i
Consumption Plan

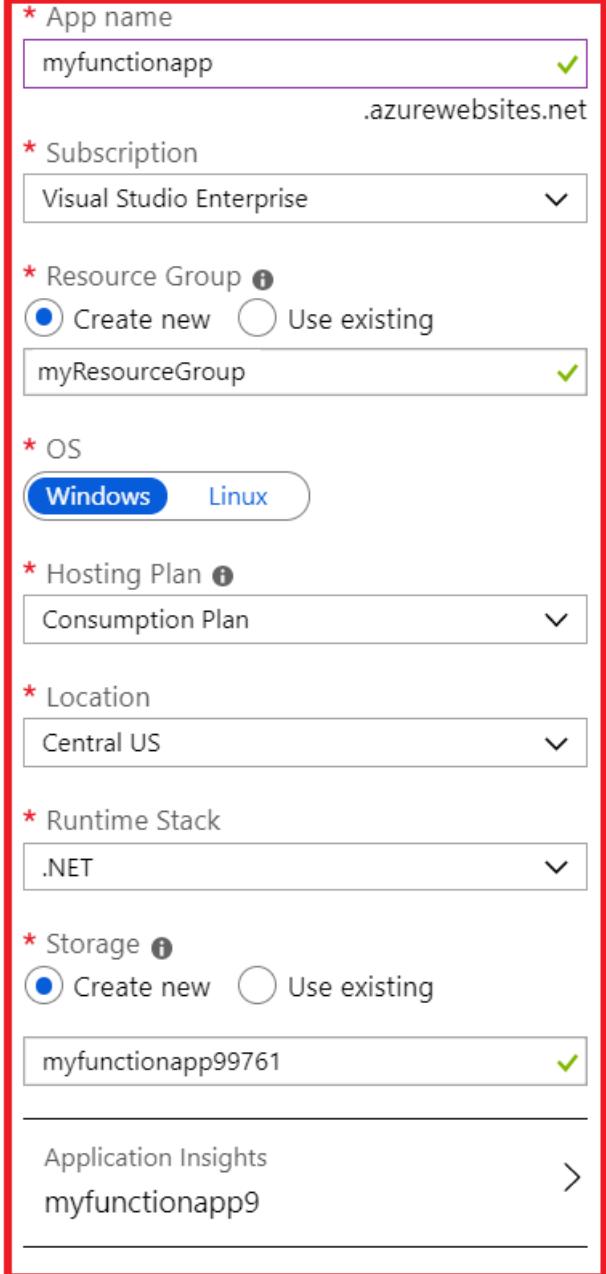
* Location
Central US

* Runtime Stack
.NET

* Storage i
 Create new Use existing
myfunctionapp99761

Application Insights >
myfunctionapp9

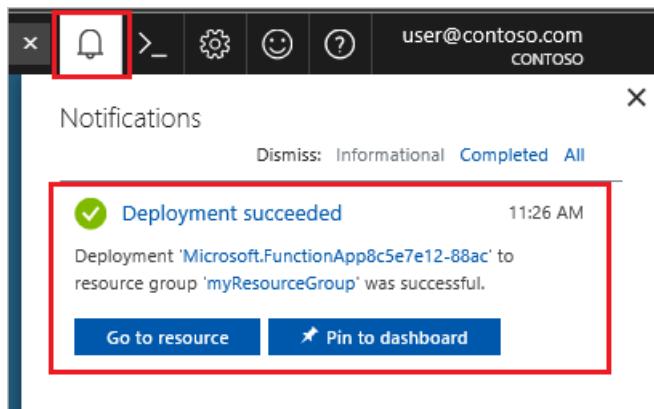
Create Automation options



SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z, 0-9, and -.
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



5. 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.

A screenshot of the Microsoft Azure Function App overview page for the app 'functions-ggaley777'. The left sidebar shows a search bar and a dropdown for 'Visual Studio Enterprise'. Under 'Function Apps', the app 'functions-ggaley777' is listed with a status of 'Running'. Below the sidebar, there are tabs for 'Overview', 'Settings', 'Platform features', and 'API definition (preview)'. The 'Overview' tab is active, showing details like 'Subscription: Visual Studio Enterprise', 'Resource group: functions-ggaley777', 'Status: Running', 'Subscription ID', 'Location: South Central US', and 'URL: https://functions-ggaley777.azurewebsites.net'. Below these details, a section titled 'Configured features' contains a note: 'Quick links to your features will show up here after you've configured them from the "Platform features" tab above.'

Next, you create a function in the new function app.

Create a timer triggered function

1. Expand your function app and click the + button next to **Functions**. If this is the first function in your function app, select **In-portal** then **Continue**. Otherwise, go to step three.

Microsoft Azure

Home > functions-ggailey777

functions-ggailey777

Visual Studio Enterprise

Function Apps

functions-ggailey777

+ Functions

Proxies

Slots (preview)

Overview Platform features Quickstart

Azure Functions - getting started

Follow our Quickstart guidance to author and publish a function [Learn more](#)

1 CHOOSE A DEVELOPMENT ENVIRONMENT

2 CREATE A FUNCTION

In-portal Author functions quickly in the portal

VS Code Use Visual Studio Code to author your functions

Any editor + Core Tools Write functions using your favorite editor and the Azure Functions Core Tools

Continue

2. Choose **More templates** then **Finish and view templates**.

Overview Platform features Quickstart

Azure Functions - getting started

Follow our Quickstart guidance to author and publish a function [Learn more](#)

CHOOSE A DEVELOPMENT ENVIRONMENT

CREATE A FUNCTION

Webhook + API HTTP A function that will be run whenever it receives an HTTP request

Timer A function that will be run on a specified schedule

More templates...

View all templates available to this function app

Back Finish and view templates

3. In the search field, type `timer` and configure the new trigger with the settings as specified in the table below the image.

Timer trigger

New Function

Name: TimerTrigger1

Timer trigger

Schedule ⓘ
0 */1 * * *

Create **Cancel**

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.

4. Click **Create**. A function is created in your chosen language that runs every minute.

5. Verify execution by viewing trace information written to the logs.

```

Logs
2017-04-27T17:51:00.189 JavaScript timer trigger function ran! 2017-04-27T17:51:00.189Z
2017-04-27T17:51:00.204 Function completed (Success, Id=a2830e1b-7d62-4ad0-81ce-9e4f857bf175, Duration=189ms)
2017-04-27T17:52:00.688 Function started (Id=c3491caf-28af-4db0-af2e-7698e49da6dc)
2017-04-27T17:52:00.688 JavaScript timer trigger function ran! 2017-04-27T17:52:00.688Z
2017-04-27T17:52:00.797 Function completed (Success, Id=c3491caf-28af-4db0-af2e-7698e49da6dc, Duration=118ms)
2017-04-27T17:53:00.004 Function started (Id=880e4a28-106c-4ac5-9237-4911421638bd)
2017-04-27T17:53:00.004 JavaScript timer trigger function ran! 2017-04-27T17:53:00.005Z
2017-04-27T17:53:00.004 Function completed (Success, Id=880e4a28-106c-4ac5-9237-4911421638bd, Duration=2ms)
4

```

Now, you change the function's schedule so that it runs once every hour instead of every minute.

Update the timer schedule

1. Expand your function and click **Integrate**. This is where you define input and output bindings for your function and also set the schedule.
2. Enter a new hourly **Schedule** value of `0 0 */1 * * *` and then click **Save**.

The screenshot shows the Azure portal interface for a function app named 'functions-ggailey777'. In the left sidebar, under 'Functions', a 'TimerTriggerCSharp1' function is selected. A red box highlights the 'Integrate' button in the sidebar. The main panel shows the 'Triggers' tab with 'Timer (myTimer)' selected. The 'Schedule' field contains the value '0 0 */1 * * *'. The 'Save' button at the bottom of the configuration pane is also highlighted with a red box.

You now have a function that runs once every hour.

Clean up resources

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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 'Overview' tab of the 'functions-ggailey777' function app. A red box highlights the 'Overview' tab in the top navigation bar. Another red box highlights the 'Resource group' field, which contains the value 'functions-ggailey777'. The URL field shows 'https://fun...'.

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 are 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 based on a schedule. For more information about timer triggers, see [Schedule code execution with Azure Functions](#).

Now that you have 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](#)

Store unstructured data using Azure Functions and Azure Cosmos DB

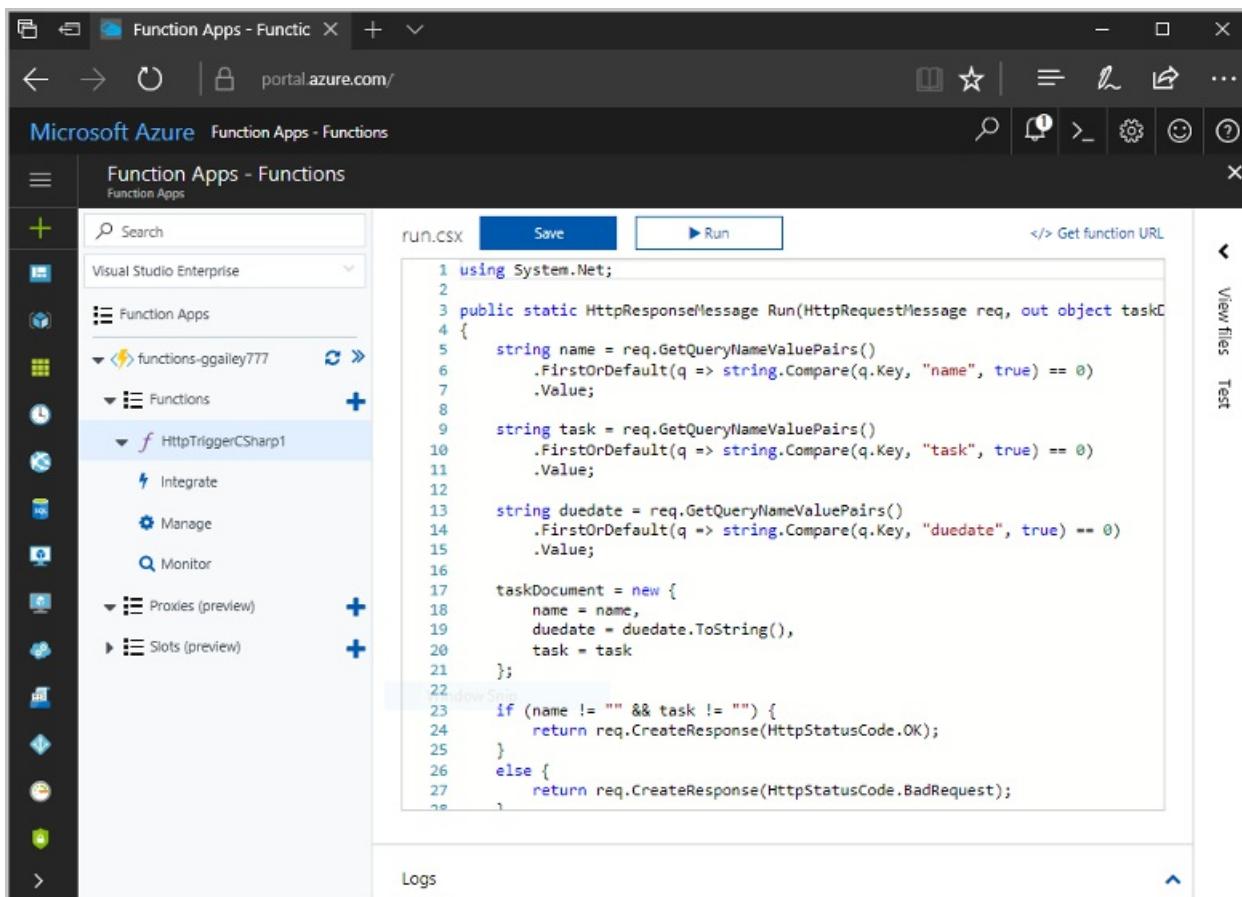
1/24/2019 • 5 minutes to read • [Edit Online](#)

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.



The screenshot shows the Microsoft Azure Function Apps - Functions portal. On the left, there's a sidebar with icons for Visual Studio Enterprise, Function Apps, Proxies (preview), and Slots (preview). The main area shows a file named 'run.csx' with the following C# code:

```
1 using System.Net;
2
3 public static HttpResponseMessage Run(HttpRequestMessage req, out object taskDocument)
4 {
5     string name = req.GetQueryNameValuePairs()
6         .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
7         .Value;
8
9     string task = req.GetQueryNameValuePairs()
10        .FirstOrDefault(q => string.Compare(q.Key, "task", true) == 0)
11        .Value;
12
13     string duedate = req.GetQueryNameValuePairs()
14        .FirstOrDefault(q => string.Compare(q.Key, "duedate", true) == 0)
15        .Value;
16
17     taskDocument = new {
18         name = name,
19         duedate = duedate.ToString(),
20         task = task
21     };
22
23     if (name != "" && task != "") {
24         return req.CreateResponse(HttpStatusCode.OK);
25     }
26     else {
27         return req.CreateResponse(HttpStatusCode.BadRequest);
28     }
}
```

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. Sign in to the [Azure portal](#).
2. Select **Create a resource > Databases > Azure Cosmos DB**.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a navigation sidebar with various service icons. A red box highlights the 'Create a resource' button at the top of this sidebar. In the main content area, the 'New' blade is open under 'Azure Marketplace'. A blue dashed box highlights the 'Databases' category in the list of services. Another red box highlights the 'Azure Cosmos DB' service item in the list, which includes a 'Quickstart tutorial' link.

3. 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	Enter a unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the ID that you provide to create your URI, use a unique ID.</p> <p>The ID can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	Core (SQL)	<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> <p>Select Core (SQL) to create a document database and query by using SQL syntax.</p> <p>Learn more about the SQL API.</p>
Location	Select 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.

Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [learn more](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription	Contoso Subscription	▼
* Resource Group	(New) myResourceGroup	▼
	Create new	

INSTANCE DETAILS

* Account Name	mysqlapicosmosdb	✓
	documents.azure.com	
* API ⓘ	Core (SQL)	▼
* Location	West US	▼
Geo-Redundancy ⓘ	Enable	Disable
Multi-region Writes ⓘ	Enable	Disable

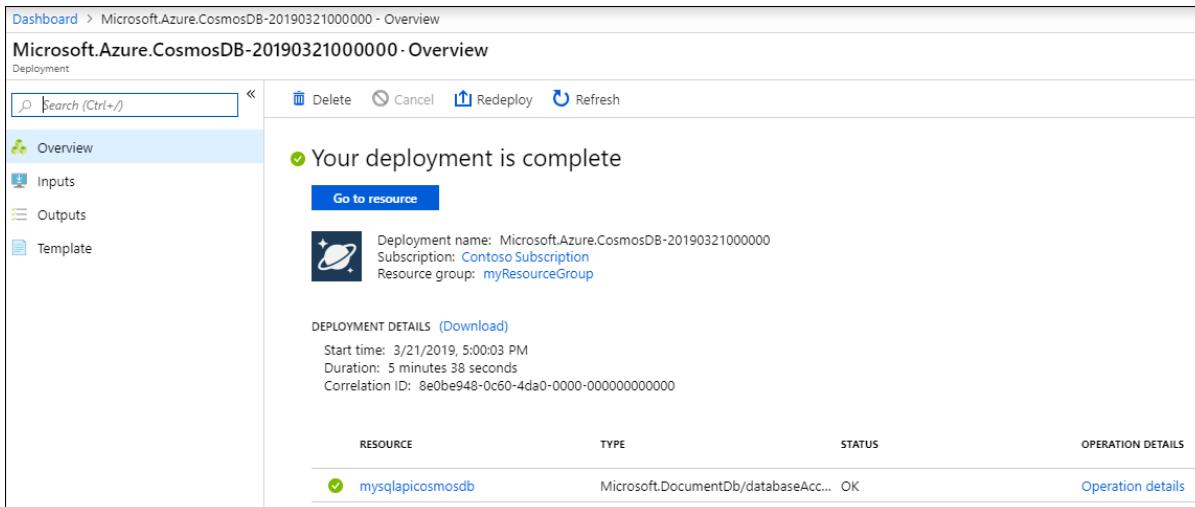
[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

5. 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 Microsoft Azure CosmosDB - Overview page for a deployment named "Microsoft.Azure.CosmosDB-20190321000000". The deployment status is shown as "Your deployment is complete" with a green checkmark icon. Below this, deployment details are listed: Deployment name: Microsoft.Azure.CosmosDB-20190321000000, Subscription: Contoso Subscription, Resource group: myResourceGroup. Under DEPLOYMENT DETAILS, it shows Start time: 3/21/2019, 5:00:03 PM, Duration: 5 minutes 38 seconds, and Correlation ID: 8e0be948-0c60-4da0-0000-000000000000. A table at the bottom lists the resource "mysqlapicosmosdb" with type Microsoft.DocumentDb/databaseAcc... and status OK, along with a link to "Operation details".

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

Choose a platform

- .NET**
- .NET Core**
- Xamarin
- Java
- Node.js
- Python

- 1 Add a collection**

In Azure Cosmos DB, data is stored in collections.

Create 'Items' collection

Create 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, fo

- 2 Download and run your .NET app**

Once collection is created, download a sample .NET app connected to it, extract, build and run.

Download

Add an output binding

1. In the portal, navigate to the function app you created previously and expand both your function app and your function.
2. Select **Integrate** and **+ New Output**, which is at the top right of the page. Choose **Azure Cosmos DB**, and click **Select**.

Home > functions-ggailey777 - HttpTrigger

functions-ggailey777 - HttpTrigger
Function Apps

Triggers 1 Inputs 1 Outputs 1 Advanced editor

HTTP (req)

+ New Input

HTTP (\$return)

+ New Output

Integrate

Manage Monitor Proxies Slots (preview)

Azure Event Hubs SendGrid Excel table Outlook mail message

OneDrive file Microsoft Graph webhook subscription **Azure Cosmos DB**

Select Cancel

3. If you get an **Extensions not installed** message, choose **Install** to install the Azure Cosmos DB bindings extension in the function app. Installation may take a minute or two.

Azure Cosmos DB output

⚠ Extensions not installed

This integration requires the following extensions.

⚠ Microsoft.Azure.WebJobs.Extensions.CosmosDB

Install

4. Use the **Azure Cosmos DB output** settings as specified in the table:

Azure Cosmos DB output

Document parameter name ⓘ <input type="text" value="taskDocument"/>	Database name ⓘ <input type="text" value="taskDatabase"/>
<input type="checkbox"/> Use function return value Collection Name ⓘ <input type="text" value="TaskCollection"/>	
Azure Cosmos DB account connection ⓘ show value new <input type="text" value="ggailey777-cosmosdb_DOCUMENTDB"/>	
Collection throughput (optional) ⓘ <input type="text" value="400"/>	

Save Cancel

SETTING	SUGGESTED VALUE	DESCRIPTION
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	Checked	The collection doesn't already exist, so create it.
Azure Cosmos DB account connection	New setting	Select New , then choose your Subscription , the Database account you created earlier, and Select . Creates an application setting for your account connection. This setting is used by the binding to connection to the database.
Collection throughput	400 RU	If you want to reduce latency, you can scale up the throughput later.

5. Select **Save** 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. Expand the right window and select **Test**. Under **Query**, click **+ Add parameter** and add the following parameters to the query string:

- `name`
- `task`
- `duedate`

2. Click **Run** and verify that a 200 status is returned.

The screenshot shows the Azure Functions portal. On the left, the navigation bar includes 'Function Apps' and 'Integrate'. The main area displays the code for 'HttpTriggerCSharp1.cs' (RUN.CSX) which retrieves query parameters from an incoming HTTP request. To the right, the 'Test' tab is selected, showing a query string with 'name=Maria Anders', 'task=Shopping', and 'duedate=07/27/2017'. The 'Request body' field contains a JSON object: { "name": "Azure" }. The 'Output' section shows a successful response with status 200 OK. A red box highlights the 'Test' tab, the query parameters, the request body, and the 'Run' button at the bottom.

- On the left side of the Azure portal, expand the icon bar, type `cosmos` in the search field, and select **Azure Cosmos DB**.

The screenshot shows the 'Integrate' section of the Azure Functions portal. A search bar at the top contains the text 'cosmos'. Below it, a list of services is shown: 'Azure Cosmos DB' (selected and highlighted with a red box), 'NoSQL (DocumentDB) accounts', and a link to 'Help improve the service menu!'. A red box highlights the search bar and the 'cosmos' entry in the list.

- Choose your Azure Cosmos DB account, then select the **Data Explorer**.
- Expand the **Collections** nodes, select the new document, and confirm that the document contains your query string values, along with some additional metadata.

The screenshot shows the Data Explorer (Preview) interface. The left sidebar has a red box around the 'Data Explorer (Preview)' item. The main area shows a collection named 'taskDatabase' with a sub-collection 'TaskCollection'. Under 'Documents', a specific document is selected, showing its properties and raw JSON content. The JSON content is as follows:

```

1 {
2   "name": "Maria Anders",
3   "duedate": "07/27/2017",
4   "task": "Shopping",
5   "id": "e53fc38d-cb6c-485f-825b-3e636b244e4",
6   "_rid": "JUQpAH-9cQACAAAAAA==",
7   "_self": "dbs/JUQpAH-/colls/JUQpAH-9cQA=/docs/JUQpAH-9cQ",
8   "_etag": "\"0a00660b-0000-0000-0000-597fd7e40800\"",
9   "_attachments": "attachments/",
10  "_ts": 1501558566
11 }

```

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 left menu in the Azure portal, select **Resource groups** and then select **myResourceGroup**.

On the resource group page, make sure that the listed resources are the ones you want to delete.

Select **Delete**, type **myResourceGroup** in the text box, 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.

Add an Azure Storage queue binding to your Python function

7/30/2019 • 7 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 Python quickstart](#).

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. 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

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 Functions, each type of binding requires that a `direction`, a `type`, and a unique `name` be defined in the `function.json` file. Depending on the binding type, additional properties might be required. The [queue output configuration](#) describes the fields required for an Azure Storage queue binding.

To create a binding, you add a binding configuration object to the `function.json` file. Edit the `function.json` file in your `HttpTrigger` folder to add an object to the `bindings` array that has these properties:

PROPERTY	VALUE	DESCRIPTION
<code>name</code>	<code>msg</code>	The name that identifies the binding parameter referenced in your code.
<code>type</code>	<code>queue</code>	The binding is an Azure Storage queue binding.
<code>direction</code>	<code>out</code>	The binding is an output binding.
<code>queueName</code>	<code>outqueue</code>	The name of the queue that the binding writes to. When the <code>queueName</code> doesn't exist, the binding creates it on first use.
<code>connection</code>	<code>AzureWebJobsStorage</code>	The name of an app 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.

Your `function.json` file should now look like this example:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "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"
    }
  ]
}
```

Add code that uses the output binding

After the `name` is configured, you can start using it to access the binding as a method attribute in the function signature. In the following example, `msg` is an instance of the [azure.functions.InputStream class](#).

```
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
        )
```

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.

Run the function locally

As before, use the following command to start the Functions runtime locally:

```
func host start
```

NOTE

Because in the previous quickstart 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.

Copy the URL of your `HttpTrigger` function from the runtime output and paste it into your browser's address bar. Append the query string `?name=<yourusername>` to this URL and run the request. You should see the same response in the browser as you did in the previous article.

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:

```
export 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](#).

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

Deploy the function app project to Azure

After the function app is created in Azure, you can use the `func azure functionapp publish` Core Tools command to deploy your project code to Azure. In the following command, replace `<APP_NAME>` with the name of your app from the previous step.

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

You'll see output similar to the following, which has been truncated for readability:

```
Getting site publishing info...
...
Preparing archive...
Uploading content...
Upload completed successfully.
Deployment completed successfully.
Syncing triggers...
Functions in myfunctionapp:
HttpTrigger - [httpTrigger]
    Invoke url: https://myfunctionapp.azurewebsites.net/api/httptrigger?code=cCr8sAxfBiow548FBDSL1....
```

Copy the `Invoke url` value for your `HttpTrigger`, which you can now use to test your function in Azure. The URL contains a `code` query string value that is your function key. This key makes it difficult for others to call your HTTP trigger endpoint in Azure.

Again, you can use cURL or a browser to test the deployed function. As before, append the query string `&name=<yourname>` to the URL, as in this example:

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

You can [examine the Storage queue message](#) 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 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 Python, see the [Azure Functions Python developer guide](#) and [Azure Functions triggers and bindings](#).

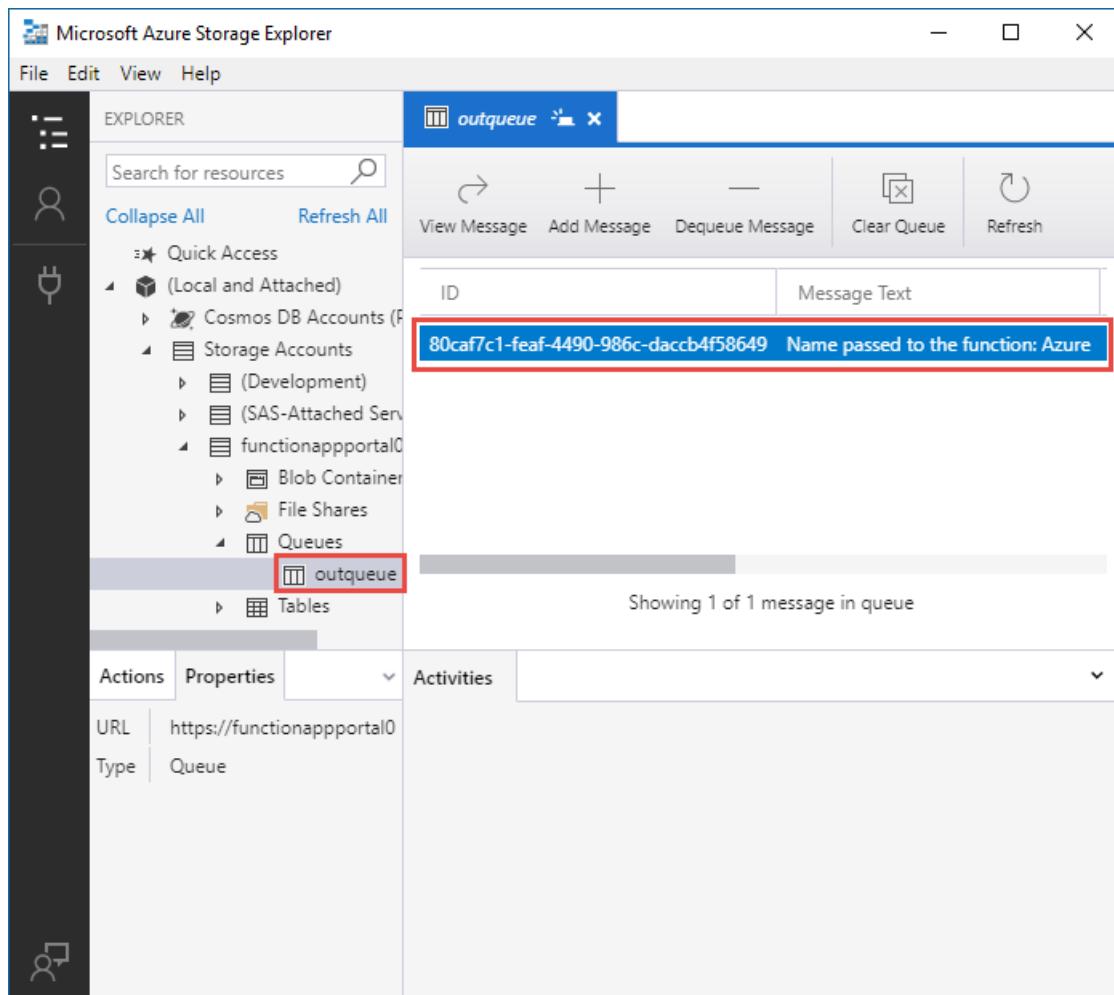
Next, you should enable Application Insights monitoring for your function app:

[Enable Application Insights integration](#)

Add messages to an Azure Storage queue using Functions

3/15/2019 • 6 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 Explorer to view the queue messages that your function creates:



Prerequisites

To complete this quickstart:

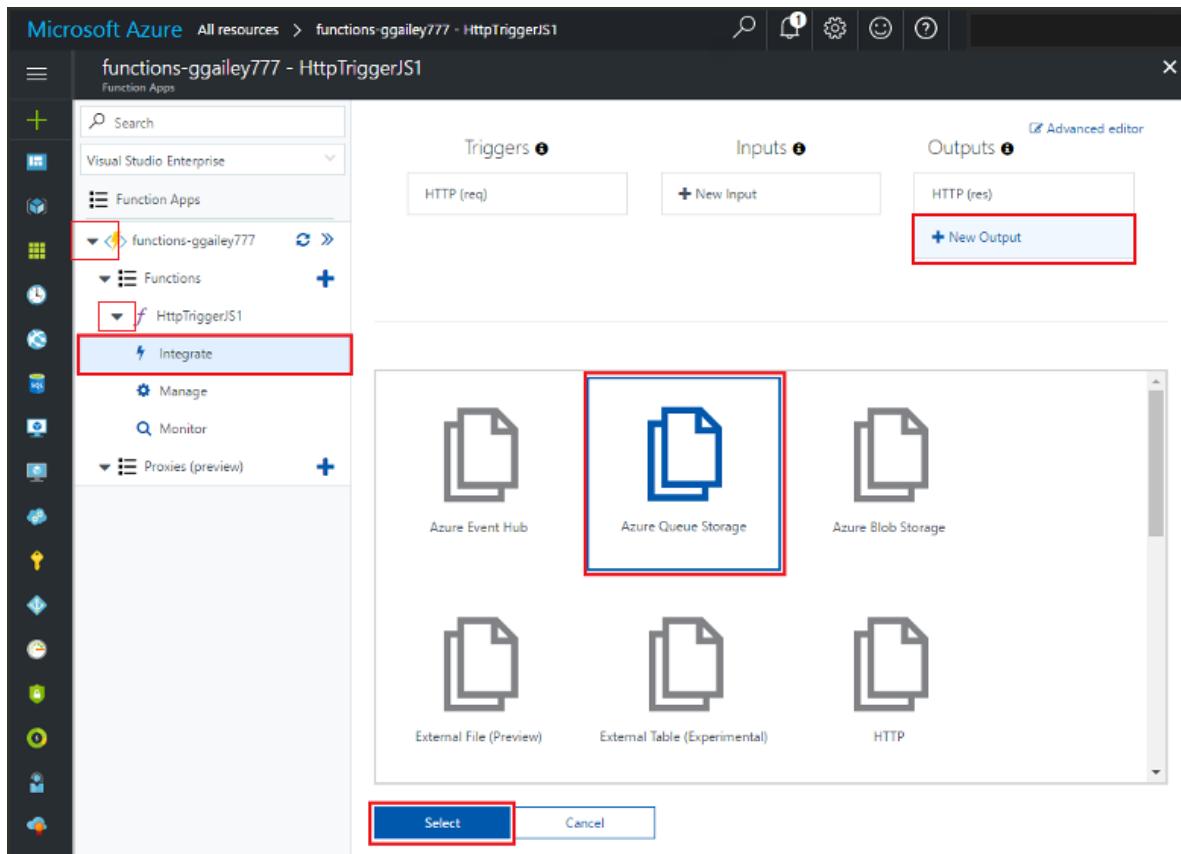
- Follow the directions in [Create your first function from the Azure portal](#) and don't do the **Clean up resources** step. That quickstart creates the function app and function that you use here.
- Install [Microsoft Azure Storage Explorer](#). This is a tool you'll use to examine queue messages that your output binding creates.

Add an output binding

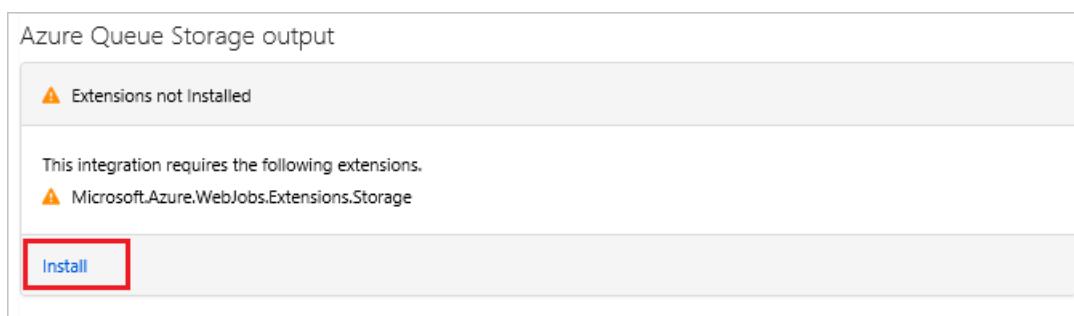
In this section, you use the portal UI to add a queue storage output binding to the function you created earlier. This binding will make it possible to write minimal code to create a message in a queue. You don't have to write

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.

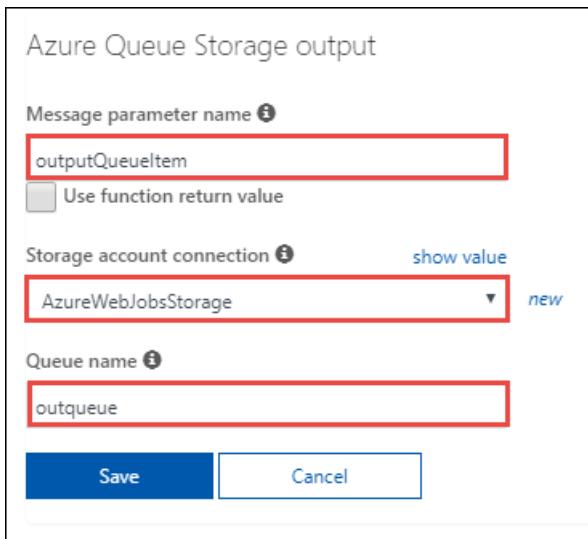
1. In the Azure portal, open the function app page for the function app that you created in [Create your first function from the Azure portal](#). To do this, select **All services > Function Apps**, and then select your function app.
2. Select the function that you created in that earlier quickstart.
3. Select **Integrate > New output > Azure Queue storage**.
4. Click **Select**.



5. If you get an **Extensions not installed** message, choose **Install** to install the Storage bindings extension in the function app. This may take a minute or two.



6. Under **Azure Queue Storage output**, use the settings as specified in the table that follows this screenshot:



SETTING	SUGGESTED VALUE	DESCRIPTION
Message parameter name	outputQueueItem	The name of the output binding parameter.
Storage account connection	AzureWebJobsStorage	You can use the storage account connection already being used by your function app, or create a new one.
Queue name	outqueue	The name of the queue to connect to in your Storage account.

7. Click **Save** 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. Select your function 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(HttpRequest req,
    ICollector<string> outputQueueItem, ILogger log)
{
    ...
}
```

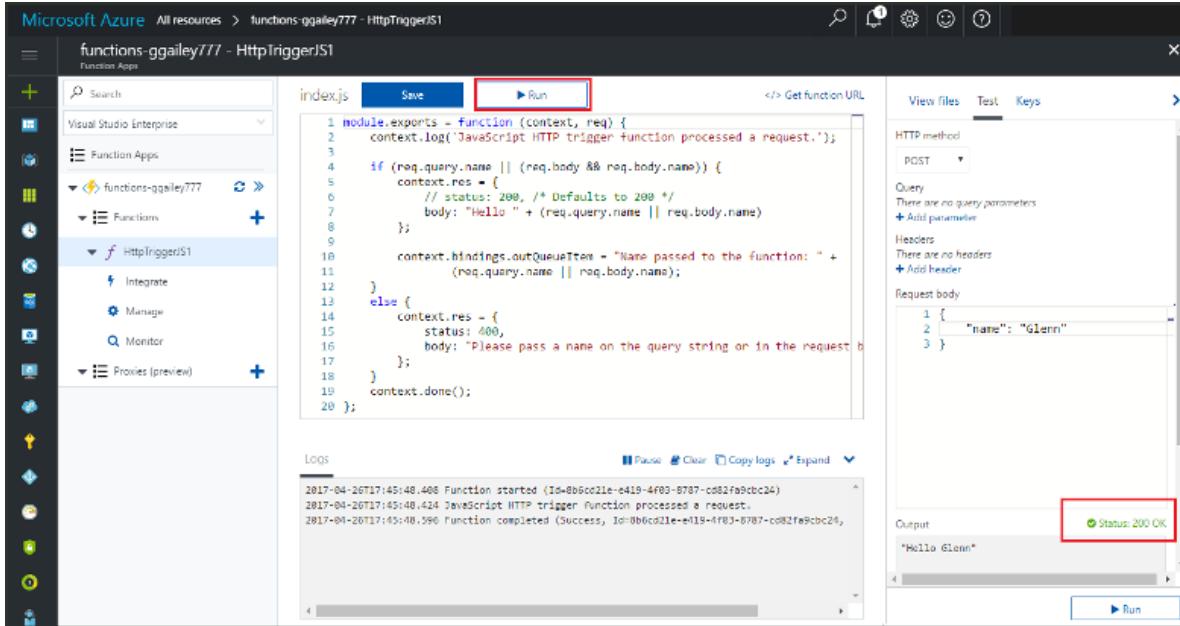
In the body of the function just before the `return` statement, add code that uses the parameter to create a queue message.

```
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 **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](#).

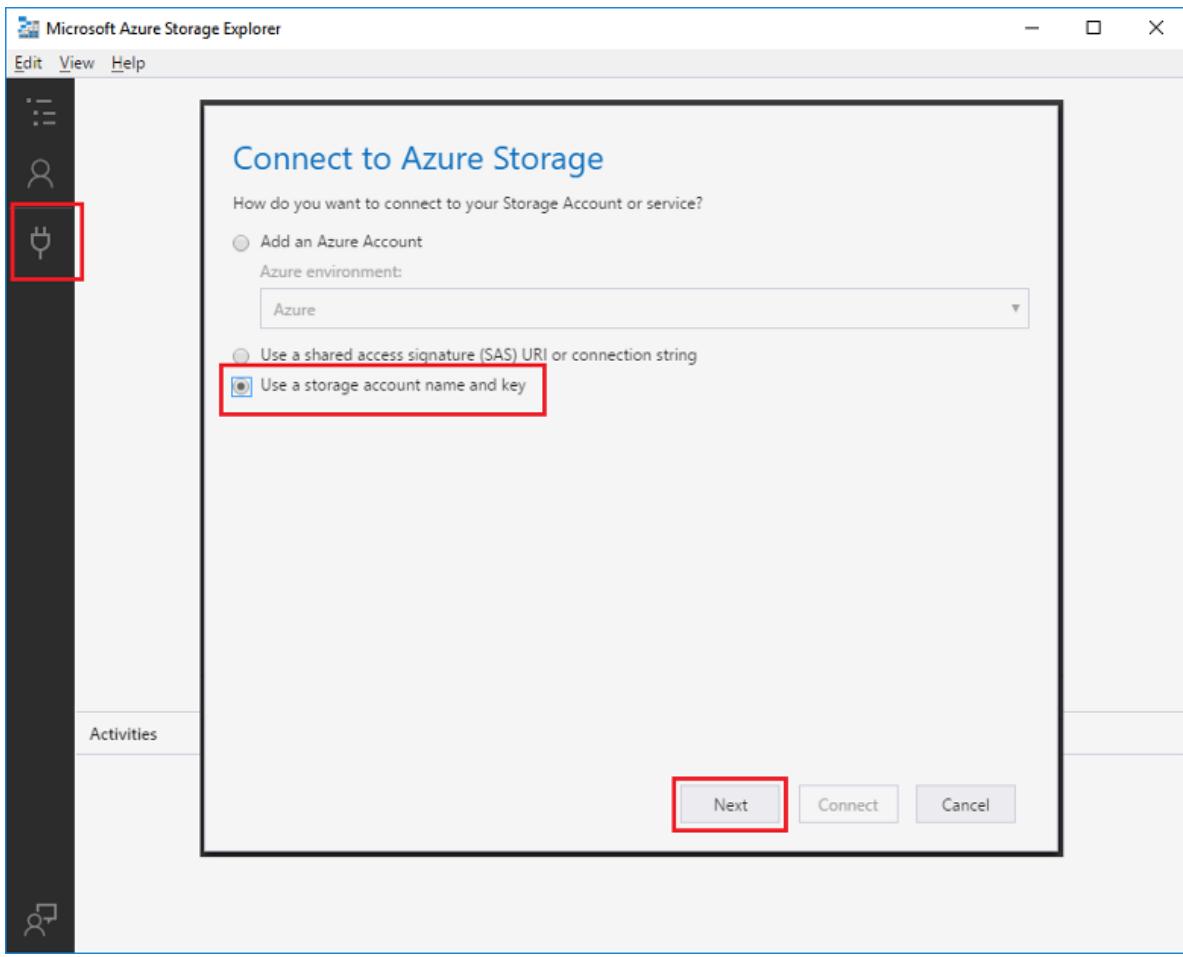
2. Check the logs to make sure that the function succeeded.

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 Explorer to verify that the queue and a message in it were created.

Connect Storage Explorer to your account

Skip this section if you have already installed Storage Explorer and connected it to the storage account that you're using with this quickstart.

1. Run the [Microsoft Azure Storage Explorer](#) tool, select the connect icon on the left, choose **Use a storage account name and key**, and select **Next**.

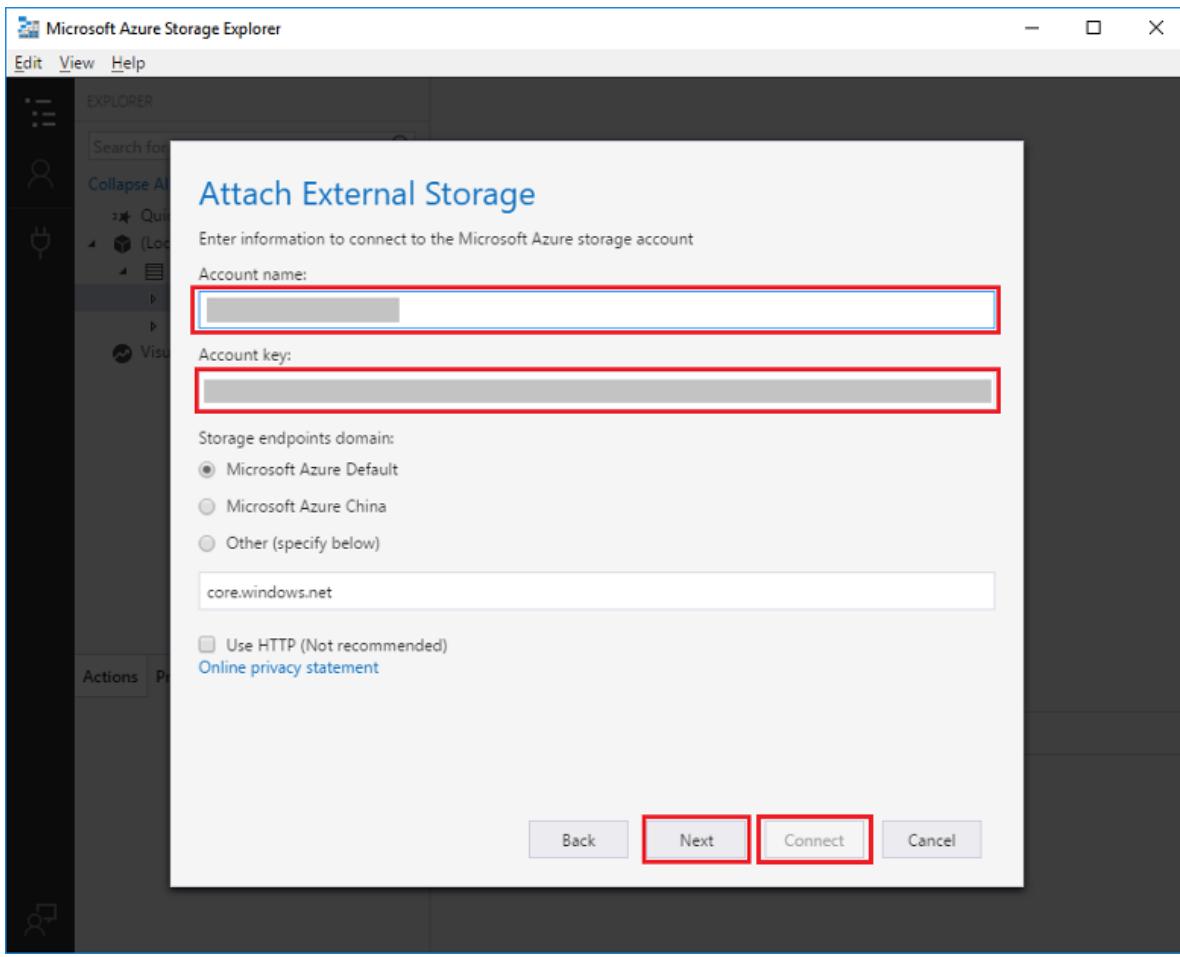


2. In the Azure portal, on the function app page, select your function and then select **Integrate**.
3. Select the **Azure Queue storage** output binding that you added in an earlier step.
4. Expand the **Documentation** section at the bottom of the page.

The portal shows credentials that you can use in Storage Explorer to connect to the storage account.

The screenshot shows the Azure Functions blade for the 'HttpTriggerJS1' function. On the left sidebar, the 'Integrate' option is selected and highlighted with a red box. In the main area, under 'Outputs', the 'Azure Queue Storage (outputQueueItem)' option is also highlighted with a red box. The 'Message parameter name' is set to 'outputQueueItem' and the 'Queue name' is set to 'outqueue'. A 'Use function return value' checkbox is unchecked. The 'Storage account connection' dropdown is set to 'AzureWebJobsDashboard'. At the bottom of the blade, there is a 'Documentation' link and a 'Cancel' button.

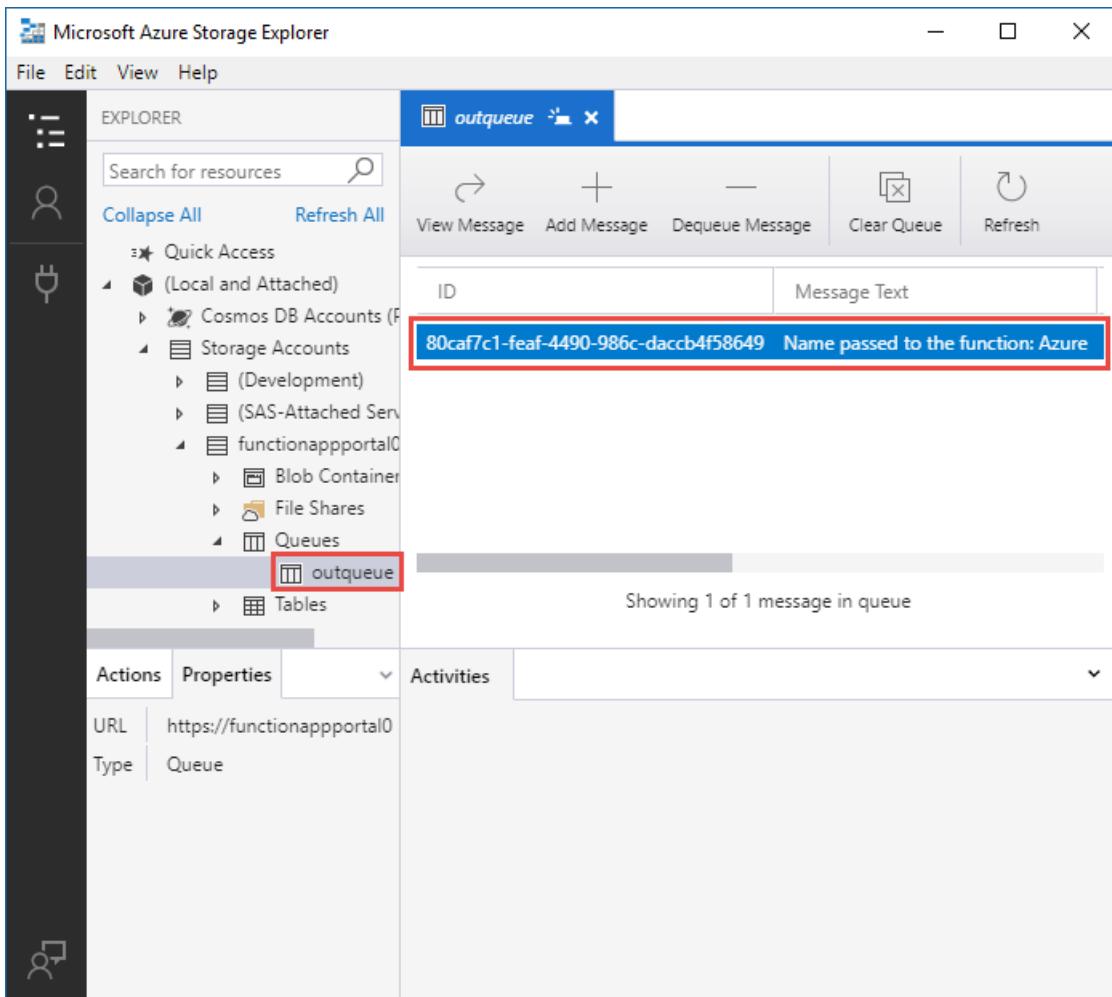
5. Copy the **Account Name** value from the portal and paste it in the **Account name** box in Storage Explorer.
6. Click the show/hide icon next to **Account Key** to display the value, and then copy the **Account Key** value and paste it in the **Account key** box in Storage Explorer.
7. Select **Next > Connect**.



Examine the output queue

1. In Storage Explorer, select the storage account that you're using for this quickstart.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default `name` value of `Azure`, the queue message is *Name passed to the function: Azure*.



3. Run the function again, and you'll see a new message appear in the queue.

Clean up resources

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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 Microsoft Azure Functions overview page for a function app named 'functions-ggailey777'. The 'Overview' tab is active. Key details shown include:

- Status:** Running
- Subscription:** Visual Studio Enterprise
- Resource group:** functions-ggailey777 (highlighted by a red box)
- Location:** South Central US
- URL:** https://fun... (partially visible)
- App Service:** SouthCent

On the left sidebar, under 'Function Apps', the specific app 'functions-ggailey777' is expanded, showing 'Functions', 'Proxies (preview)', and 'Slots (preview)'.

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 are 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 functions to Azure Storage using Visual Studio Code

7/28/2019 • 10 minutes to read • [Edit Online](#)

Azure Functions lets you connect functions to Azure services and other 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. While 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 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 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](#) (C# projects only).
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you are 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 `Azure Functions: Download Remote Settings....`.
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

IMPORTANT

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

3. Copy the value `AzureWebJobsStorage`, which is the key for the Storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you are using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

JavaScript

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)"  
    }  
}
```

C# class library

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

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.

JavaScript

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.
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.

PROMPT	VALUE	DESCRIPTION
Select setting from "local.setting.json"	AzureWebJobsStorage	The name of an application setting that contains the connection string for the Storage account. The AzureWebJobsStorage 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` file, which should now look like the following example:

```
{
  ...
  "bindings": [
    {
      "authLevel": "function",
      "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"
    }
  ]
}
```

C# class library

In a C# class library project, the bindings are defined as binding attributes on the function method. The `function.json` file is then auto-generated based on these attributes.

Open the `HttpTrigger.cs` project file and add the following `using` statement:

```
using Microsoft.Azure.WebJobs.Extensions.Storage;
```

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("HttpTrigger")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "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.

JavaScript

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.

```
// Add a message to the Storage queue.
context.bindings.msg = "Name passed to the function: " +
(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.
        context.bindings.msg = "Name passed to the function: " +
        (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"
        };
    }
};
```

C#

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("HttpTrigger")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "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"];

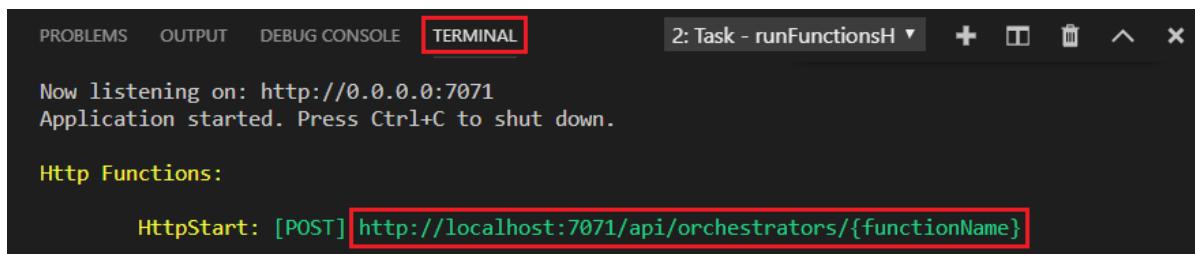
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    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");
}
```

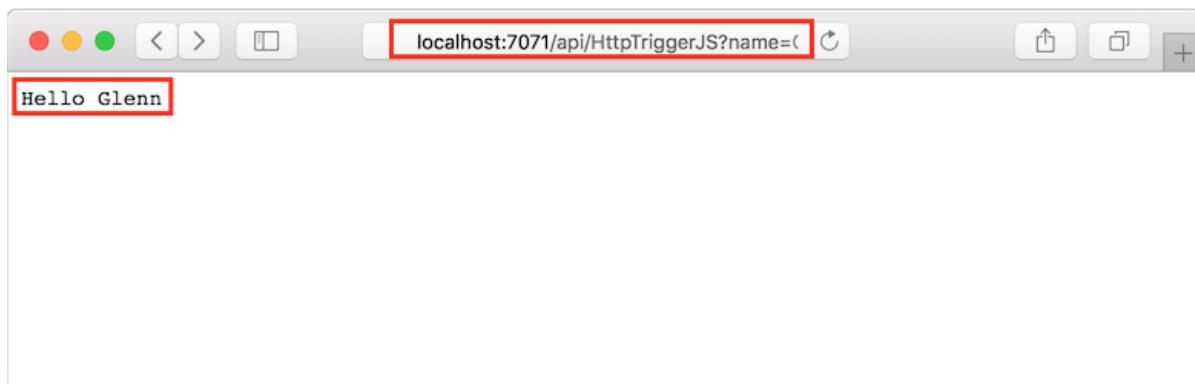
Run the function locally

Azure Functions Core Tools lets you run an Azure Functions project on your local development computer.

1. To test your function, set a breakpoint in the function code and 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. This URL includes the function key, which is passed to the `code` query parameter.



- Paste the URL for the HTTP request into your browser's address bar. Append the query string `?name=<yourname>` to this URL and execute the request. Execution is paused when the breakpoint is hit.
 - When you continue the execution, the following shows the response in the browser to the GET request:



5. To stop debugging, press Shift + F5.

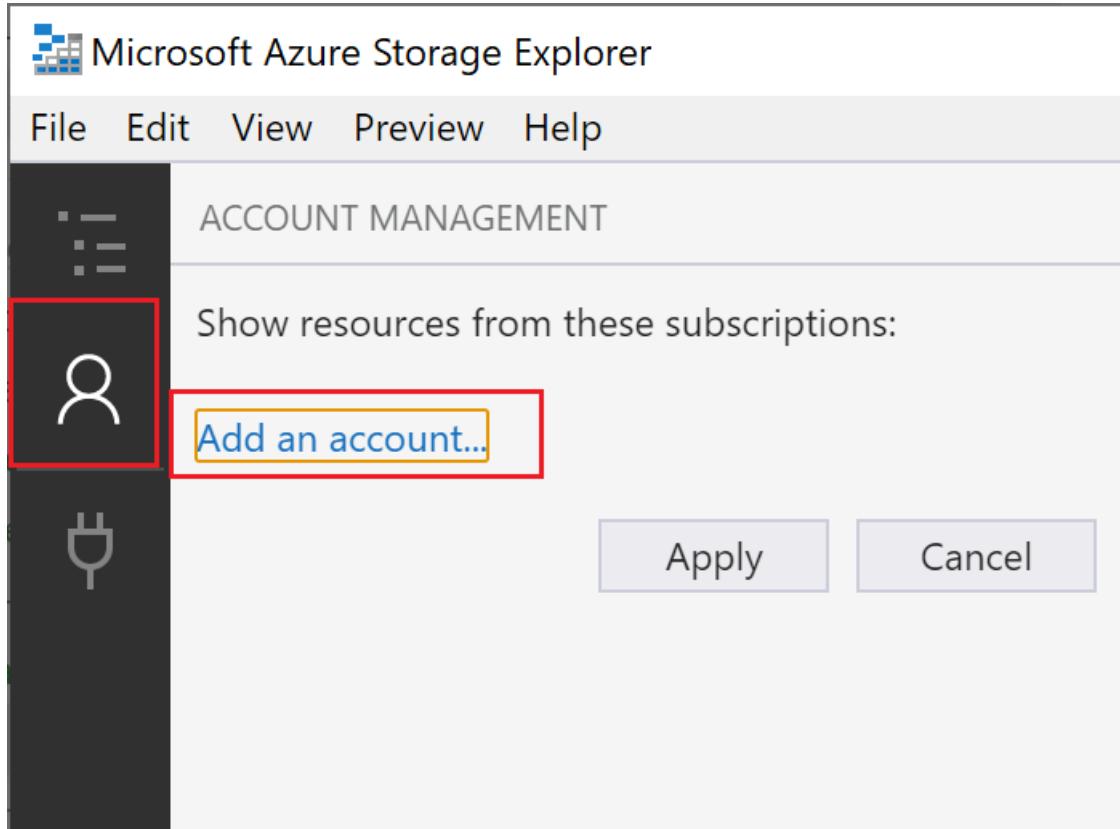
A new queue named **outqueue** is created in your storage account by the Functions runtime when the output

binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

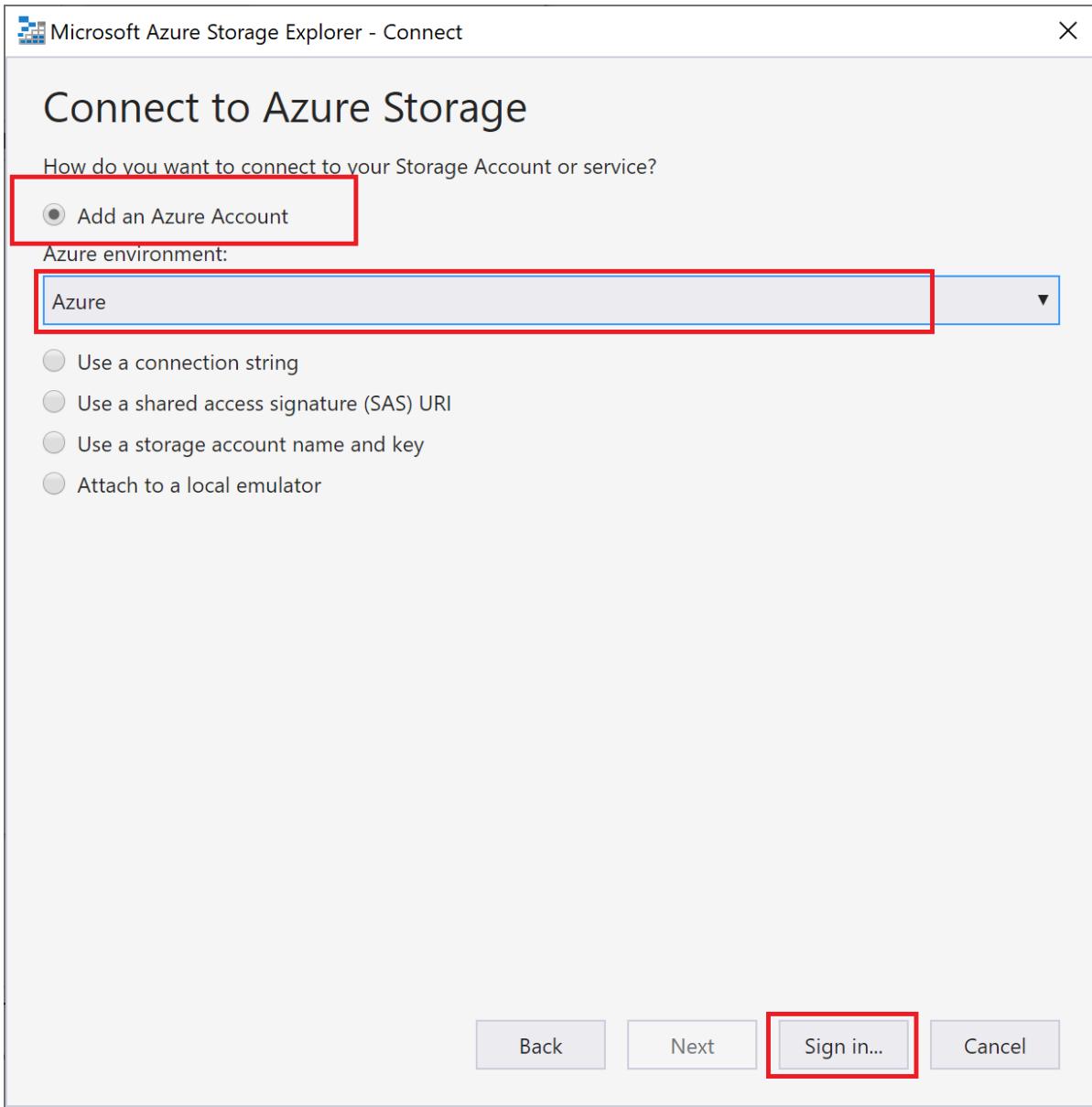
Connect Storage Explorer to your account

Skip this section if you 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...**.

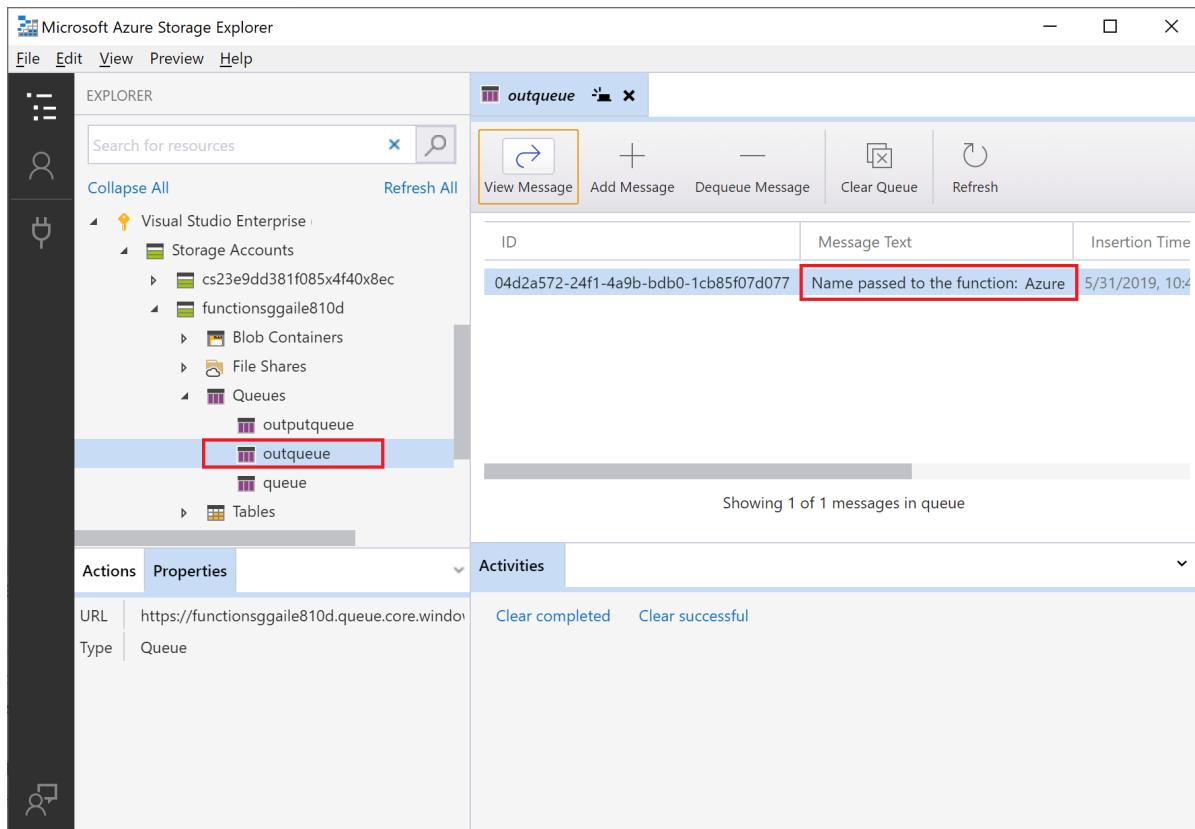


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account.

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 are 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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 is opened in the [Azure portal](#).

3. In the **Overview** tab, select the named link under **Resource group**.

The screenshot shows the Microsoft Azure Functions Overview page. At the top, there's a navigation bar with icons for search, notifications, and settings. Below that is a header with the function app name 'functions-ggailey777' and a 'Function Apps' dropdown. The main content area has tabs for 'Overview' (which is selected and highlighted with a red box), 'Settings', 'Platform features', and 'API definition (preview)'. Under the 'Overview' tab, there are sections for 'Status' (Running), 'Subscription' (Visual Studio Enterprise), 'Resource group' (functions-ggailey777, also highlighted with a red box), and 'Configured features' (with a note: 'Quick links to your features will show up here after'). On the left sidebar, there are icons for creating new resources like a function app, storage account, and database, followed by a 'Function Apps' section containing 'functions-ggailey777' (with a red box around it) and three sub-options: 'Functions', 'Proxies (preview)', and 'Slots (preview)'. Each sub-option has a '+' icon to its 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.

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 Code](#).

Next, you should enable Application Insights monitoring for your function app:

[Enable Application Insights integration](#)

Connect functions to Azure Storage using Visual Studio

7/28/2019 • 6 minutes to read • [Edit Online](#)

Azure Functions lets you connect functions to Azure services and other resources without having to write your own integration code. These pre-defined connections are called *bindings*. Bindings allow a function to use Azure services and other resources as input and output to a function.

Function execution is started by a single *trigger*. A trigger is a special type of input binding. While 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][./functions-create-first-function-vs-code.md].
- 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**.

The screenshot shows the 'Publish' blade in the Azure Functions portal. At the top, there's a dropdown menu set to 'FunctionAppPart2 - Zip Deploy' and a 'Publish' button. Below this, there are tabs for 'New', 'Edit', 'Rename', and 'Delete'. The main area has two sections: 'Summary' and 'Actions'. In the 'Summary' section, there are four items: 'Site URL' (https://functionapppart2.azurewebsites.net), 'Configuration' (Release), 'Username' (\$FunctionAppPart2), and 'Password' (redacted). In the 'Actions' section, there are two buttons: 'Manage in Cloud Explorer' and 'Edit Azure App Service settings', with the latter being highlighted by a red box.

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 is then auto-generated based on these attributes.

Open the `HttpTrigger.cs` project file and add the following `using` statement:

```
using Microsoft.Azure.WebJobs.Extensions.Storage;
```

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("HttpTrigger")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "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("HttpTrigger")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "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**. You may need to enable a firewall exception so that the tools can handle HTTP requests. Authorization levels are never enforced when running locally.
2. Copy the URL of your function from the Azure Functions runtime output.

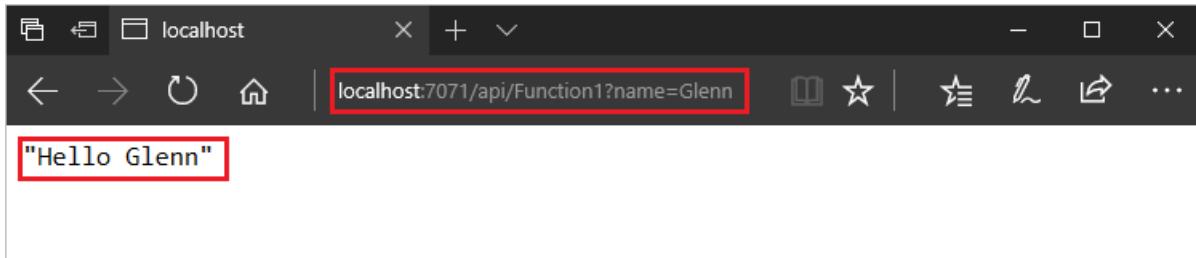
```
C:\Users\AppData\Local\AzureFunctionsTools\Releases\2.24.0\cli\func.exe
[7/6/2019 6:47:40 PM] Loading functions metadata
[7/6/2019 6:47:40 PM] 1 functions loaded
[7/6/2019 6:47:40 PM] Generating 1 job function(s)
[7/6/2019 6:47:40 PM] Found the following functions:
[7/6/2019 6:47:40 PM] MyFirstFunction.Function1.Run
[7/6/2019 6:47:40 PM]
[7/6/2019 6:47:40 PM] Host initialized (574ms)
[7/6/2019 6:47:40 PM] Host started (594ms)
[7/6/2019 6:47:40 PM] Job host started
Hosting environment: Production
Content root path: C:\Users\glenga\source\repos\MyFirstFunction\MyFirstFunction\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:
    Function1: [GET,POST] http://localhost:7071/api/Function1

[7/6/2019 6:47:47 PM] Host lock lease acquired by instance ID '00000000000000000000000000000000CF523E2A'.
```

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 execute the request. The following shows the response in the browser to the local GET request returned by the function:



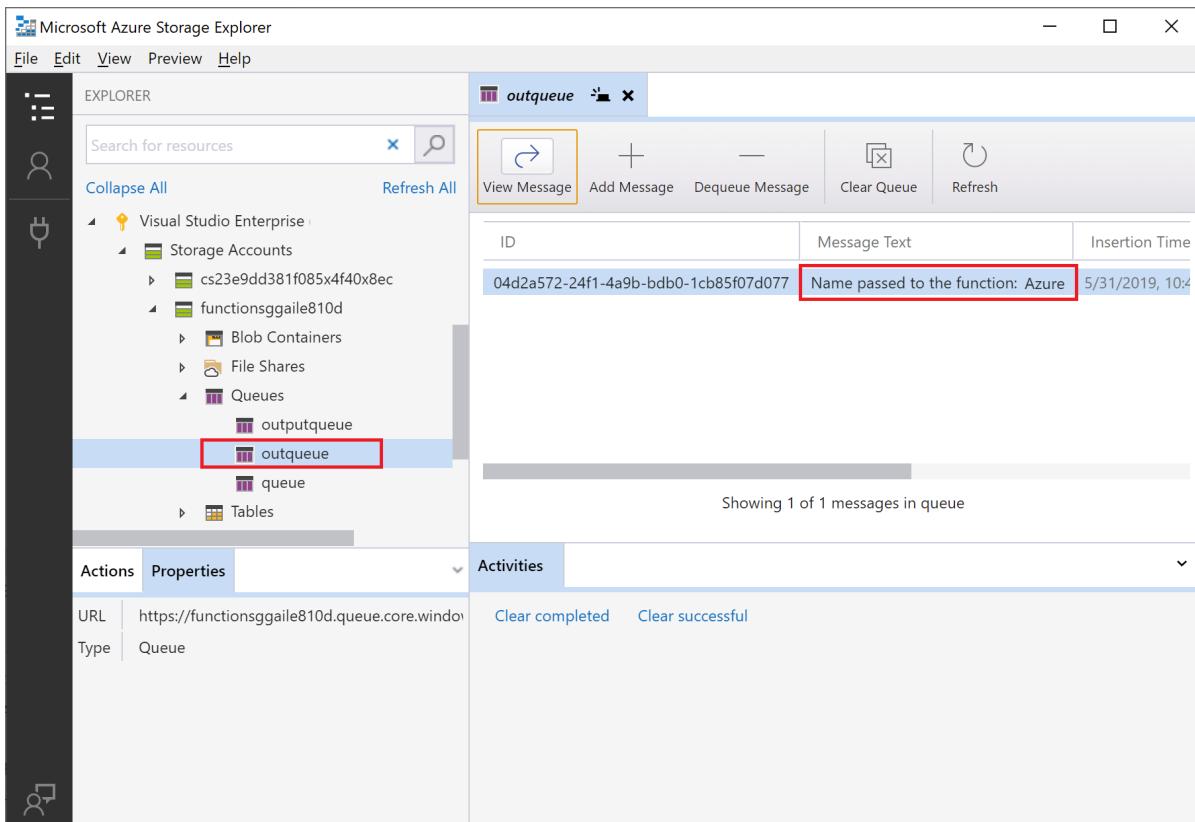
4. To stop debugging, press **Shift + F5**.

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*.



- 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 **Solution Explorer**, right-click the project and select **Publish**, then choose **Publish** to republish the project to Azure.
- After deployment completes, you can again use the browser to test the redeployed function. As before, append the query string `&name=<yourname>` to the URL.
- Again [view the message in the storage queue](#) to verify that the output binding again generates a new message in the queue.

Clean up resources

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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 Microsoft Azure Functions overview page. At the top, there's a navigation bar with icons for search, notifications, and settings. Below that is a header with the text 'functions-ggailey777' and 'Function Apps'. The main area has tabs for 'Overview' (which is highlighted with a red box), 'Settings', 'Platform features', and 'API definition (preview)'. Under 'Overview', there are sections for 'Status' (Running), 'Subscription' (Visual Studio Enterprise), 'Resource group' (functions-ggailey777, also highlighted with a red box), 'Subscription ID', 'Location' (South Central US), and 'URL' (https://fun). On the left, there's a sidebar with icons for Visual Studio Enterprise, Function Apps, and three specific function apps: 'functions-ggailey777' (highlighted with a red box), 'Proxies (preview)', and 'Slots (preview)'. Below the sidebar is a section titled 'Configured features' with the subtext 'Quick links to your features will show up here after'.

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 are 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](#)

Create a function that integrates with Azure Logic Apps

7/1/2019 • 9 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 Functions with Logic Apps and Cognitive Services on Azure to run sentiment analysis from Twitter posts. An HTTP triggered 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, TweetSentiment, 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, and Properties. The main area is titled 'TweetSentiment' and shows the 'Overview' tab selected. It displays the trigger ('TWITTER When a new tweet is posted'), actions ('COUNT 4 actions View in Logic Apps designer'), and runs history. The runs history table has columns for STATUS, START TIME, IDENTIFIER, and DURATION, showing three successful runs.

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

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).
- 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.

The screenshot shows the Azure portal's 'Create a resource' interface. On the left, a sidebar lists various services like Function Apps, Storage accounts, and App Services. The 'AI + Machine Learning' section is expanded, and 'Text Analytics' is highlighted with a red box. The main pane shows a search bar and a grid of service cards. The 'Text Analytics' card includes a 'PREVIEW' badge, a thumbnail, a title, and a 'Quickstart tutorial' link. Other visible cards include Machine Learning service workspace (preview), Data Science Virtual Machine - Windows 2016, Web App Bot, Computer Vision, Face, Language Understanding, Bing Search v7, and Azure Search.

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.

- Click on **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.

Overview

Resource group (change) myResourceGroup
Status Active
Location West Central US
Subscription name (change) Microsoft

API type Text Analytics
Pricing tier Free

Endpoint https://westcentralus.api.cognitive.microsoft.com/text/analytics/v2.0
Manage keys Show access keys ...

- 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.

Keys

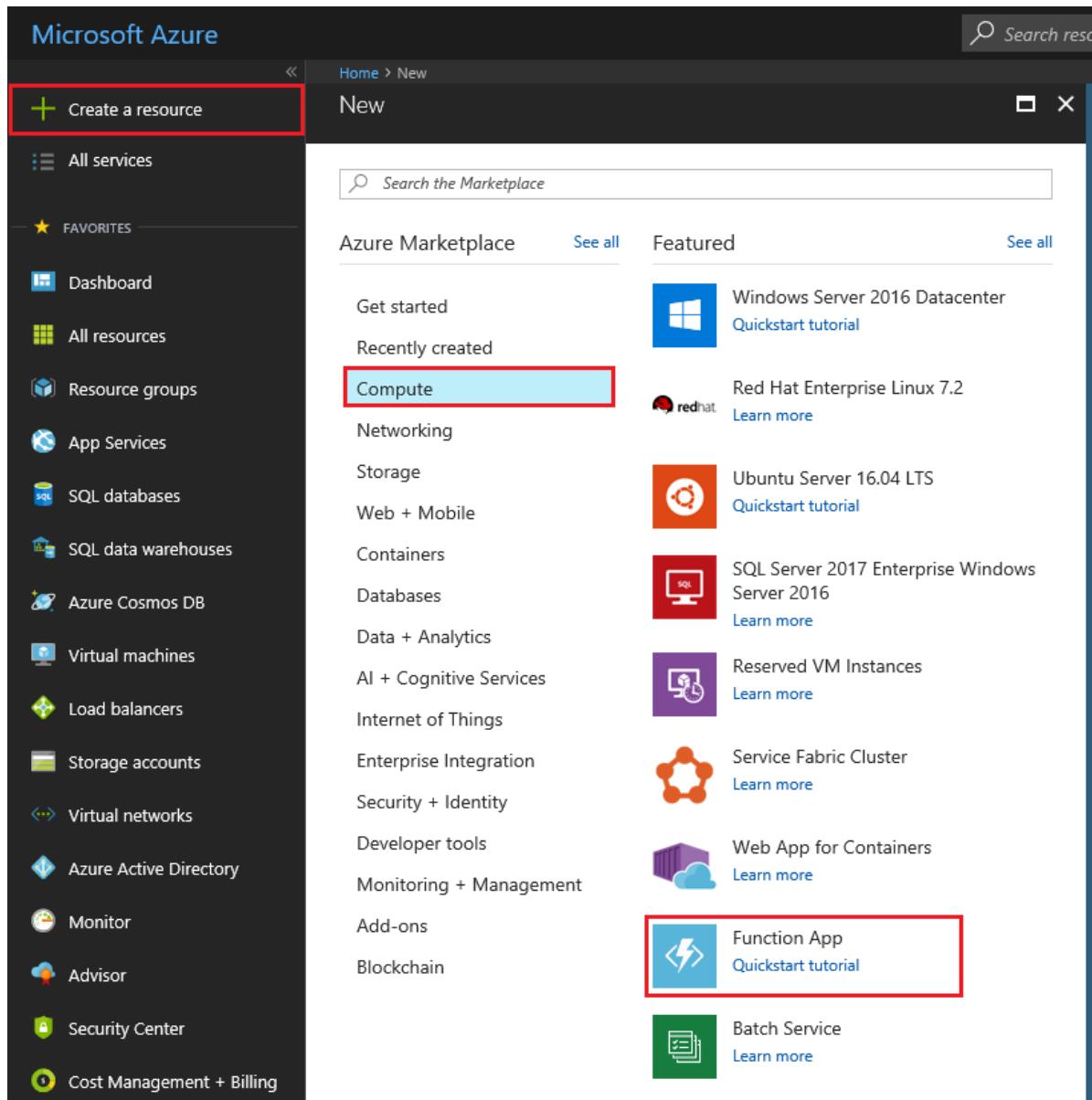
NAME MyCognitiveServicesAccnt

KEY 1
KEY 2

Create the function app

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

- Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.



2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
myfunctionapp .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
myResourceGroup ✓

* OS
 Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
Central US

* Runtime Stack
.NET

* Storage ⓘ
 Create new Use existing
myfunctionapp99761 ✓

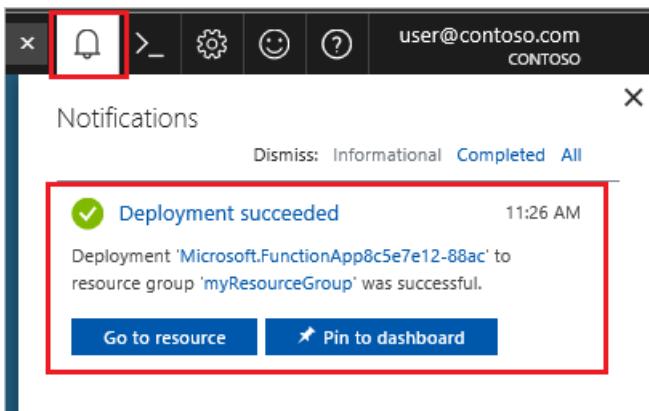
Application Insights >
myfunctionapp9

Create Automation options

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z , 0-9 , and - .
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

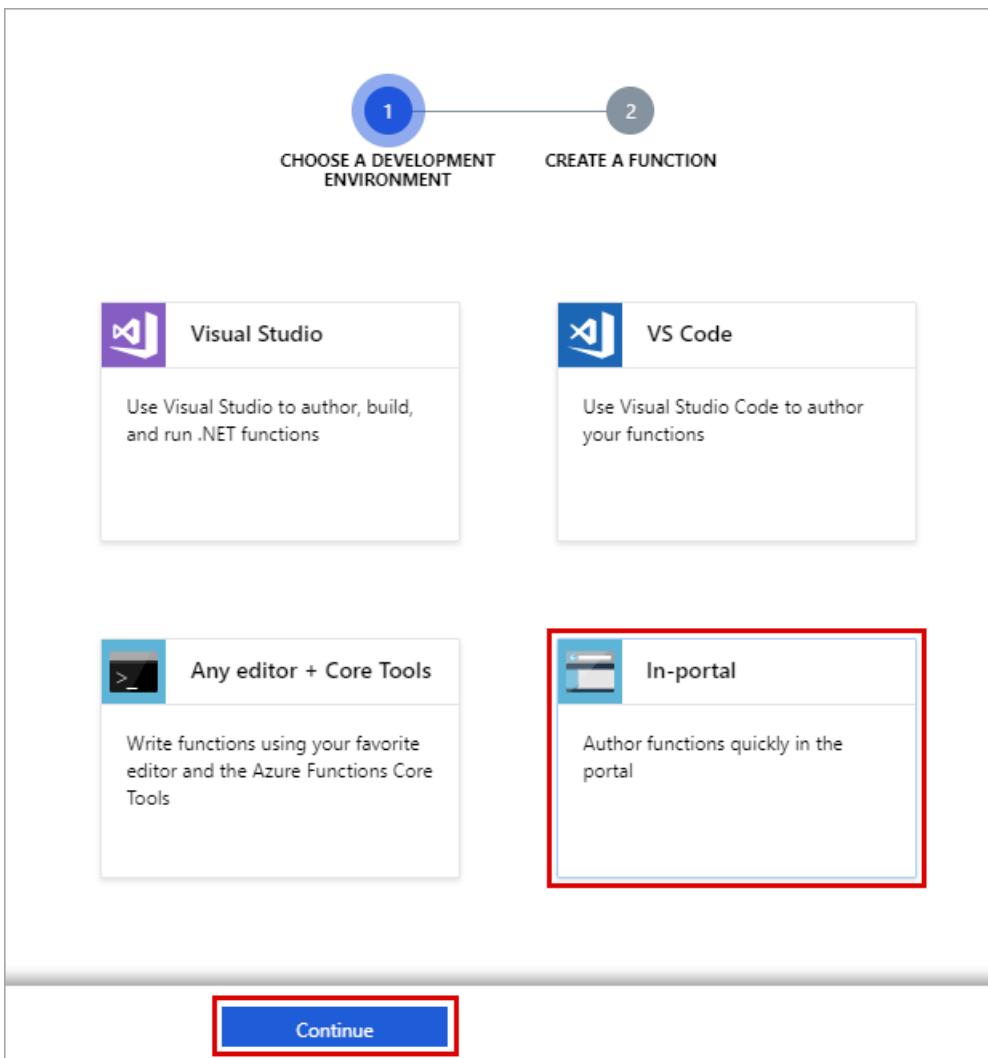
3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



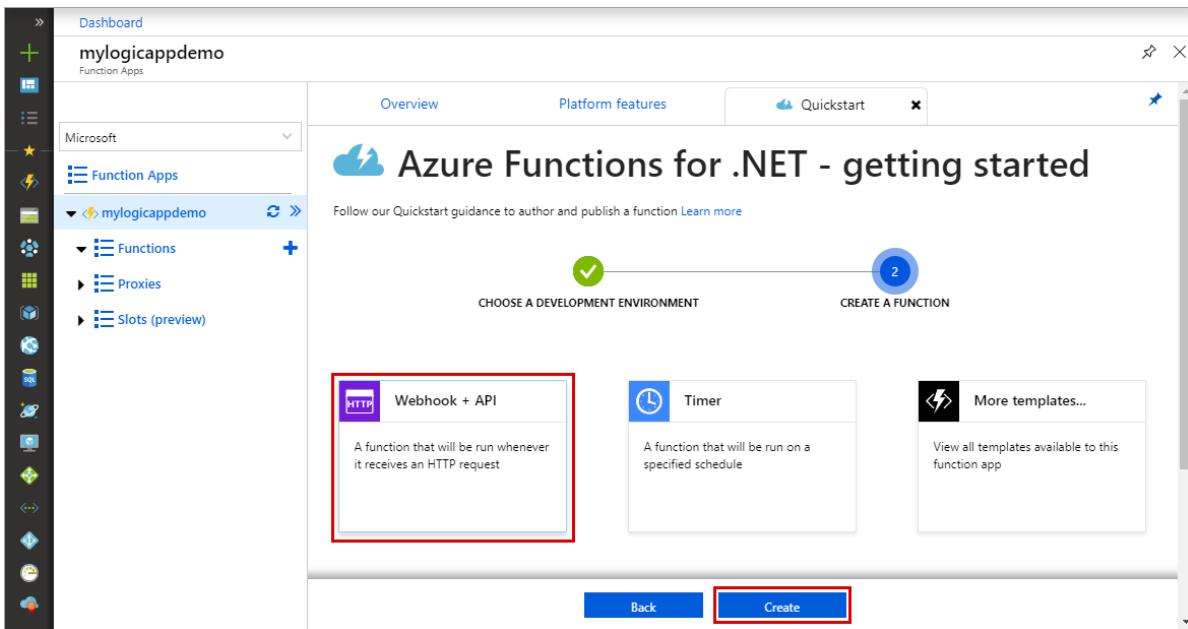
5. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

Create an HTTP triggered function

1. Expand your function app and click the + button next to **Functions**. If this is the first function in your function app, select **In-portal**.



2. Next, select **Webhook + API** and click **Create**.



- Replace the contents of the `run.csx` file with the following code, then click **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, click **Test** at the far right to expand the Test tab. Type a value of `0.2` for the **Request body**, and then click **Run**. A value of **RED** is returned in the body of the response.

The screenshot shows the Azure Functions developer tools interface. On the left, the code editor displays a C# file named 'run.csx' with the following content:

```

1 #r "Newtonsoft.Json"
2
3 using System;
4 using System.Net;
5 using Microsoft.AspNetCore.Mvc;
6 using Microsoft.Extensions.Logging;
7 using Microsoft.Extensions.Primitives;
8 using Newtonsoft.Json;
9
10 public static async Task<IActionResult> Run(HttpContext req, ILogger log)
11 {
12     string category = "GREEN";
13
14     string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
15     log.LogInformation(string.Format("The sentiment score received is '{0}'.", requestBody));
16
17     double score = Convert.ToDouble(requestBody);
18
19     if(score < .3)
20     {
21         category = "RED";
22     }
23     else if (score < .6)
24     {
25         category = "YELLOW";
26     }
27
28     return requestBody != null
29         ? (ActionResult)new OkObjectResult(category)
30         : new BadRequestObjectResult("Please pass a value on the query string or in the request body");

```

The right side of the interface includes a 'Test' tab, an 'HTTP method' dropdown set to 'POST', a 'Request body' input field containing '0.2', and an 'Output' pane showing the result 'RED'. The 'Logs' pane at the bottom contains the following log entries:

```

2018-11-05T15:22:53 Welcome, you are now connected to log-streaming service.
2018-11-05T15:23:10.886 [Information] Script for function "HttpTrigger1" changed. Reloading.
2018-11-05T15:23:11.732 [Information] Compilation succeeded.
2018-11-05T15:23:49.645 [Information] Executing "Functions.HttpTrigger1" (Reason="This function was programmatically called via the host APIs.", Id=e55153e4-02e0-48c2-9705-1f0a9b5a8995)
2018-11-05T15:23:49.845 [Information] The sentiment score received is '0.2'.
2018-11-05T15:23:49.845 [Information] Executed "Functions.HttpTrigger1" (Succeeded, Id=e55153e4-02e0-48c2-9705-1f0a9b5a8995)

```

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.

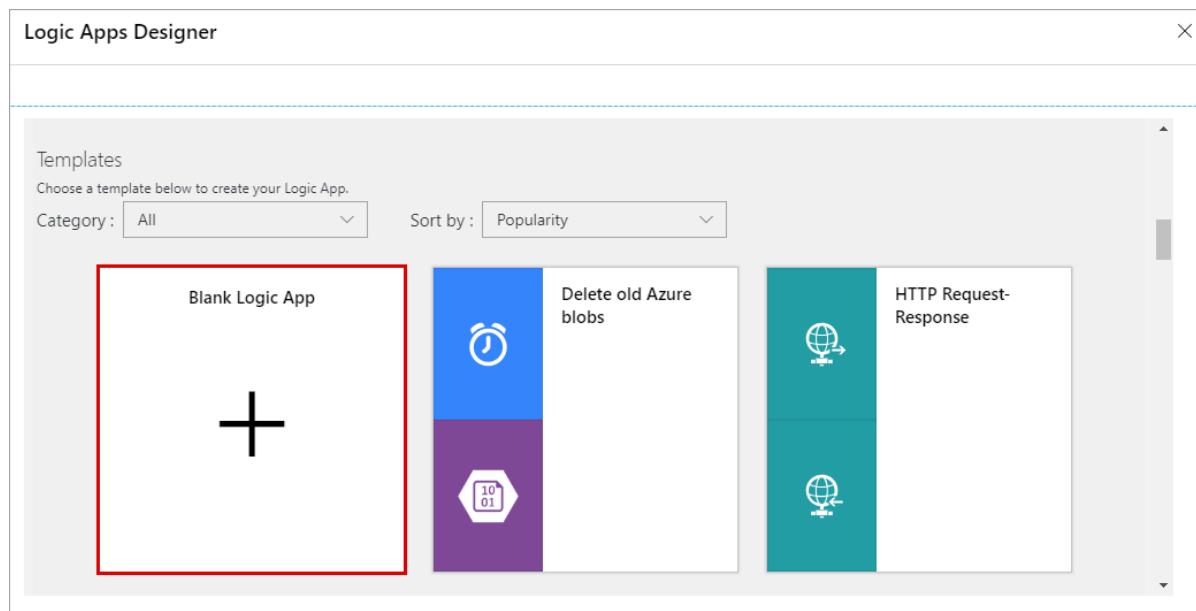
Create a logic app

1. In the Azure portal, click the **New** 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. On the left, a sidebar lists various services like Dashboard, All services, Favorites, Function Apps, Storage accounts, HDInsight clusters, All resources, Resource groups, App Services, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Virtual networks, Azure Active Directory, Monitor, Advisor, Security Center, Cost Management + Bill..., Help + support, Azure Databricks, Subscriptions, and App registrations. A red box highlights the '+ Create a resource' button at the top. In the main area, there's a search bar labeled 'Search the Marketplace'. Below it, the 'Azure Marketplace' section has tabs for 'See all' and 'Featured'. Under 'Featured', several items are listed with icons: 'Web App' (globe icon), 'Logic App' (person icon, highlighted with a red box), 'Web App for Containers' (cloud icon), 'CDN' (cloud with gear icon), 'Media Services' (camera icon), 'Azure Search' (cloud with magnifying glass icon), 'API App' (handshake icon), 'Template deployment' (cube icon), and 'API management' (cloud with gear icon). The 'Web' category is also highlighted with a red box.

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 'When a new tweet is posted' trigger configuration. The 'Save' button is highlighted with a red box. The 'Search text' field contains '#azure' and is also highlighted with a red box. The 'Interval' field is set to '15' and the 'Frequency' field is set to 'Minute', both of which are highlighted with red boxes.

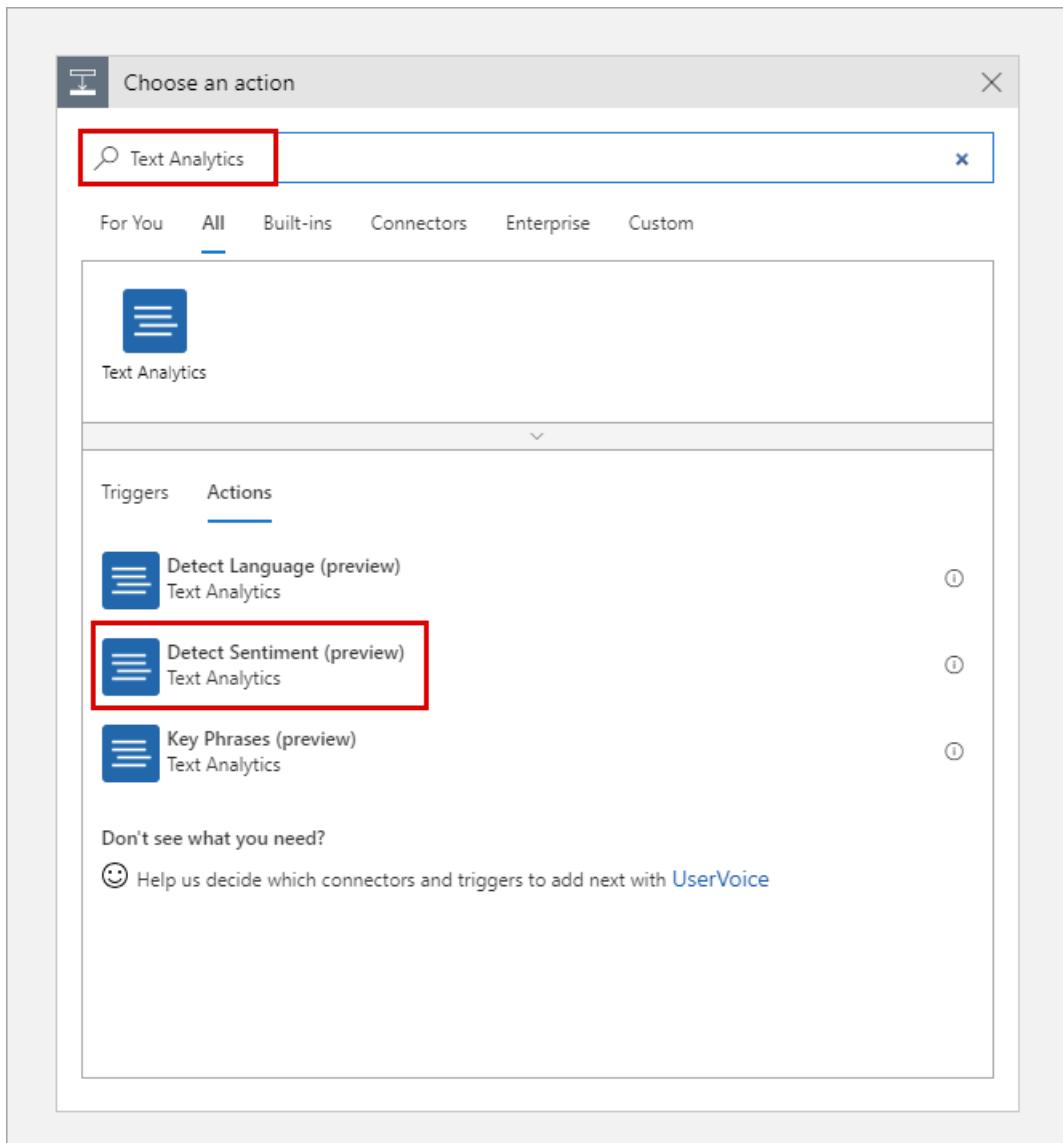
SETTING	SUGGESTED VALUE	DESCRIPTION
Search text	#Azure	Use a hashtag that is popular enough to generate new tweets in the chosen interval. When using the Free tier and your hashtag is too popular, you can quickly use up the transaction quota in your Cognitive Services API.
Interval	15	The time elapsed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

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

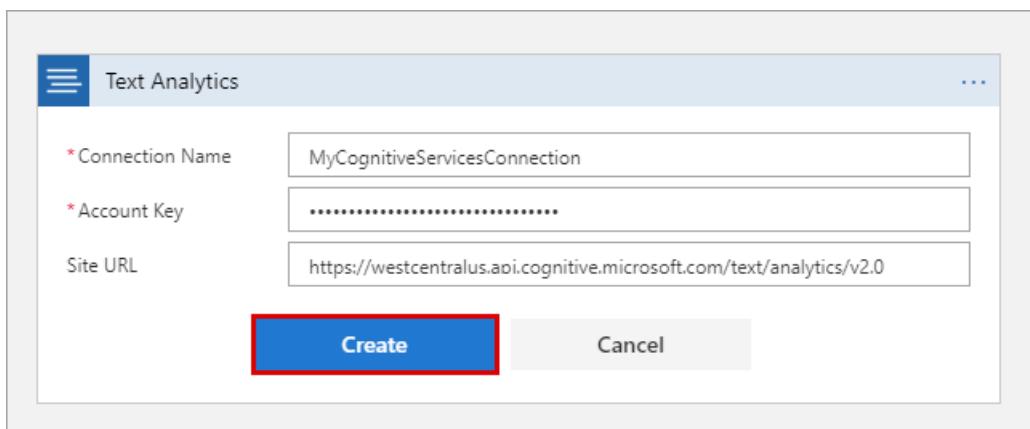
Now your app is connected to Twitter. Next, you connect to text analytics to detect the sentiment of collected tweets.

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.

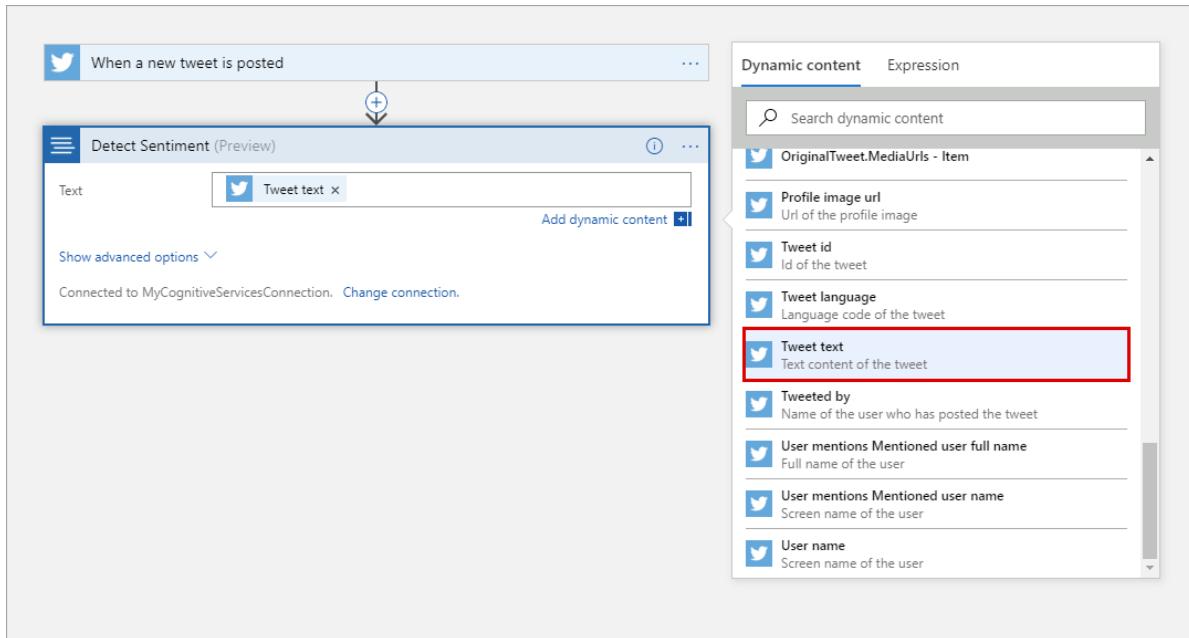


3. Type a connection name such as `MyCognitiveServicesConnection`, paste the key for your Cognitive Services API and the Cognitive Services endpoint you set aside in a text editor, and click **Create**.



*Connection Name	MyCognitiveServicesConnection
*Account Key
Site URL	https://westcentralus.api.cognitive.microsoft.com/text/analytics/v2.0

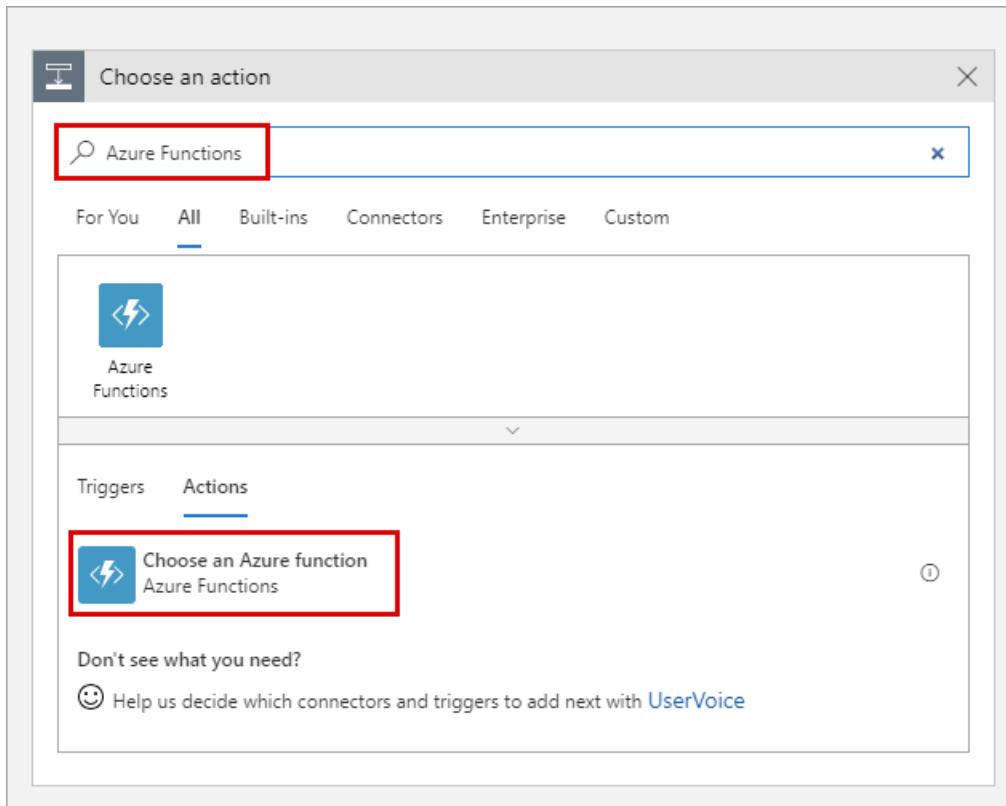
4. Next, enter **Tweet Text** in the text box and then click on **New Step**.



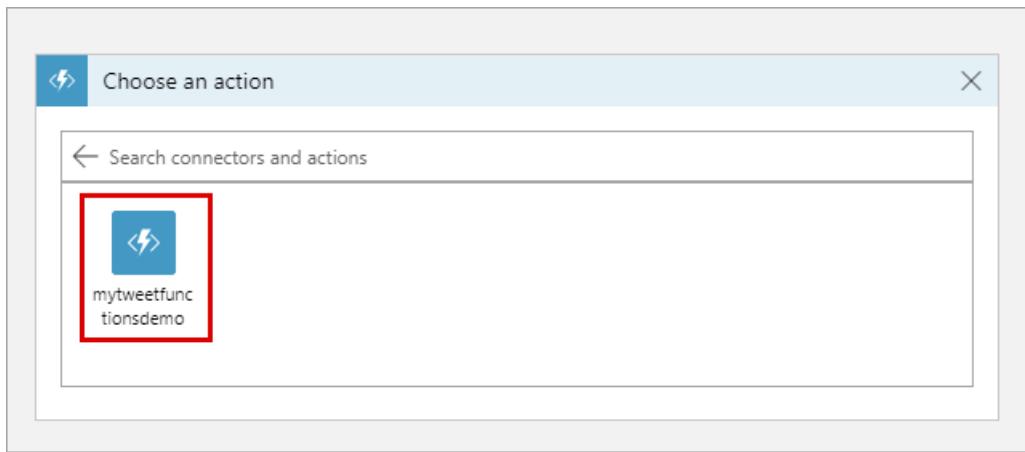
Now that sentiment detection is configured, you can add a connection to your function that consumes the sentiment score output.

Connect sentiment output to your function

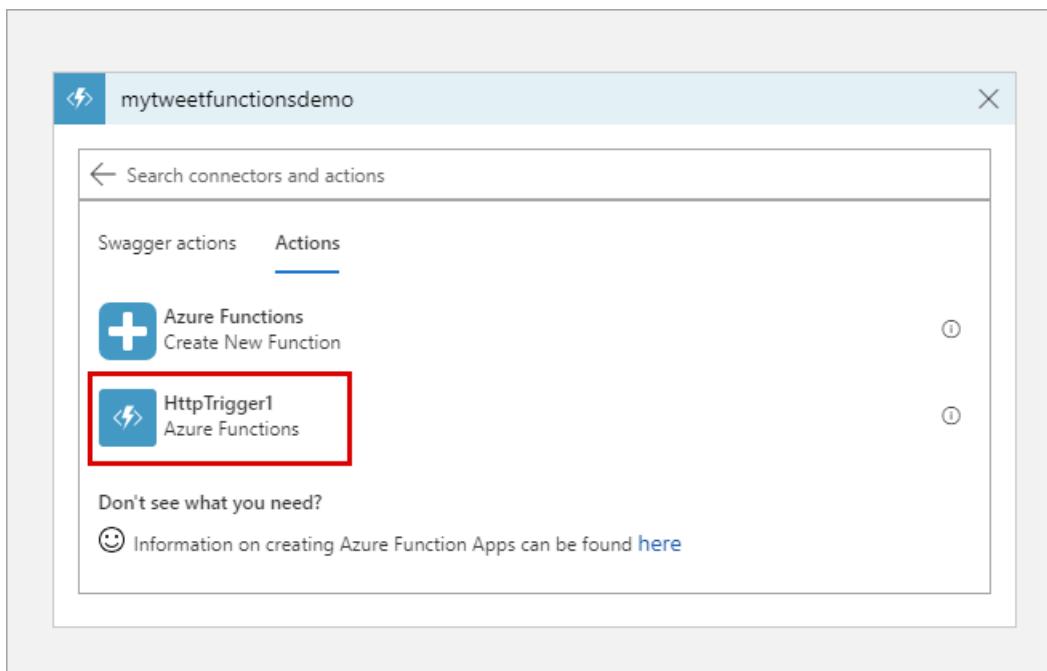
1. In the Logic Apps Designer, click **New step > Add an action**, filter on **Azure Functions** and click **Choose an Azure function**.



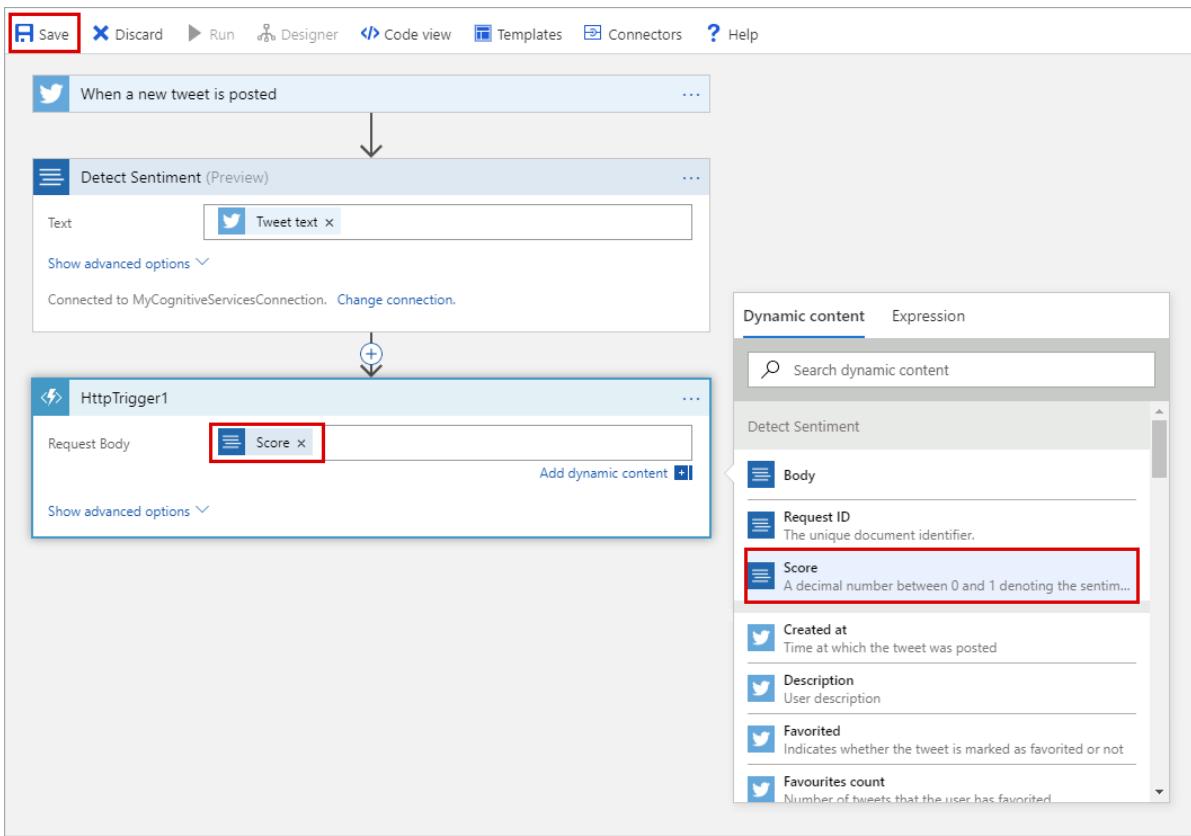
2. Select the function app you created earlier.



3. Select the function you created for this tutorial.



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

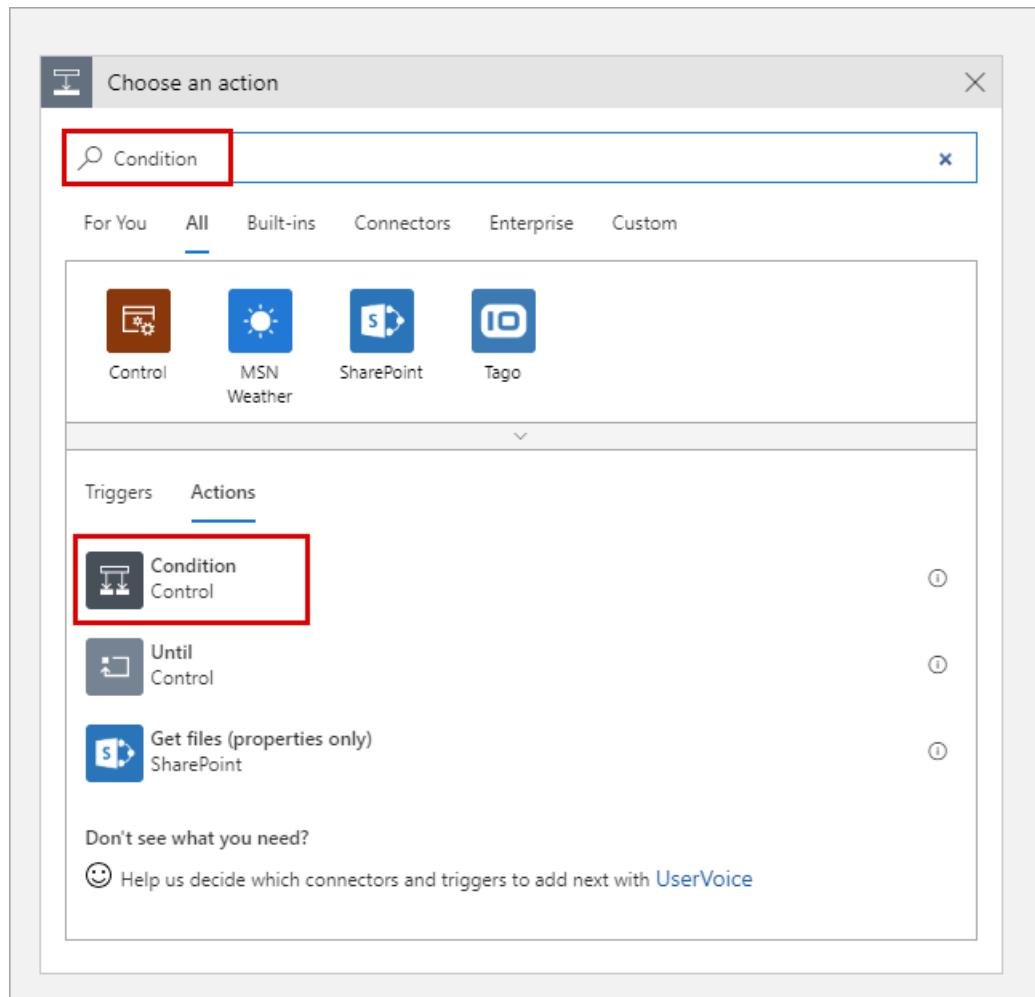


Now, your function is triggered when a sentiment score is sent from the logic app. A color-coded category is returned to the logic app by the function. Next, you add an email notification that is sent when a sentiment value of **RED** is returned from the function.

Add email notifications

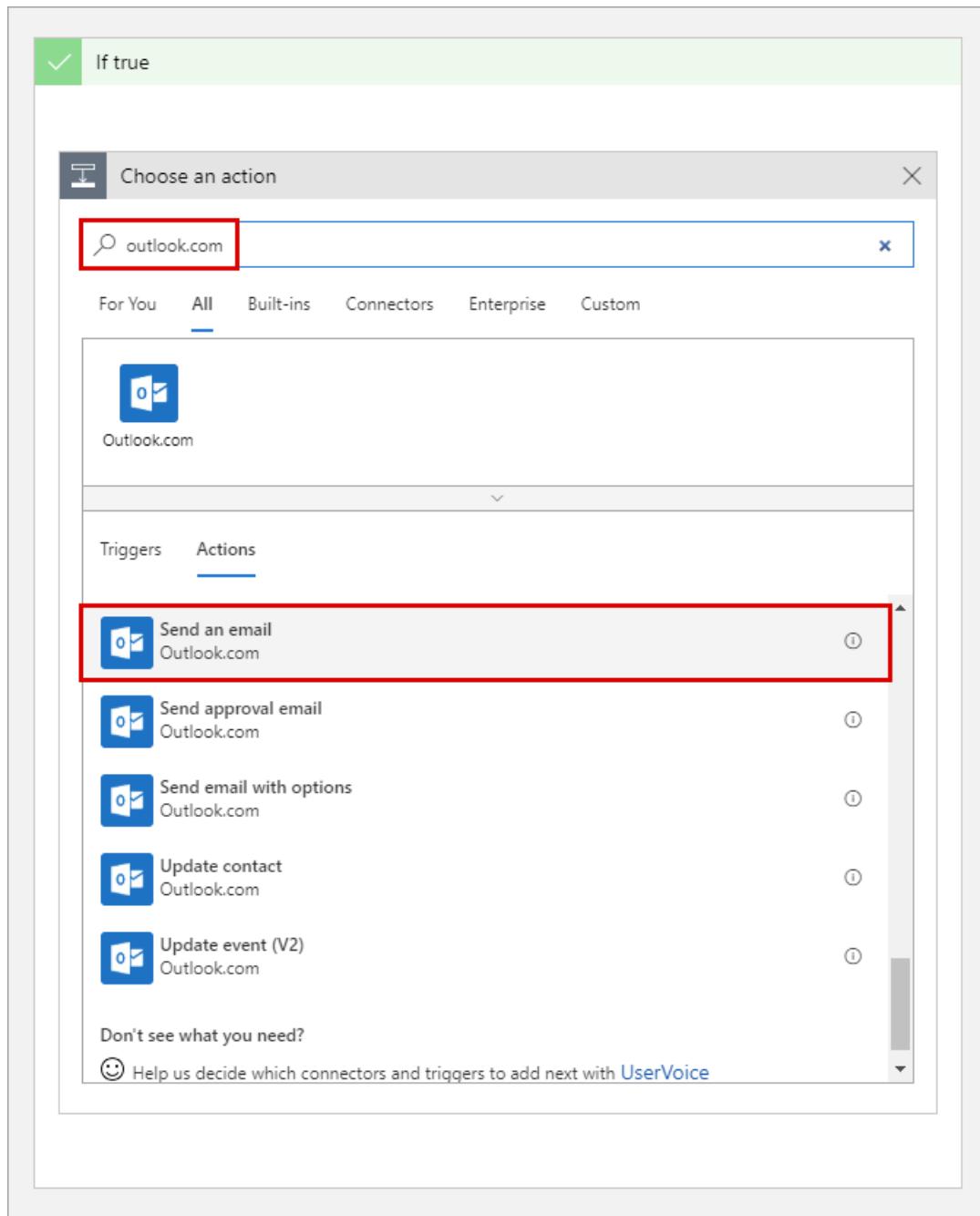
The last part of the workflow is to trigger an email when the sentiment is scored as **RED**. This topic uses an Outlook.com connector. You can perform similar steps to use a Gmail or Office 365 Outlook connector.

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**.

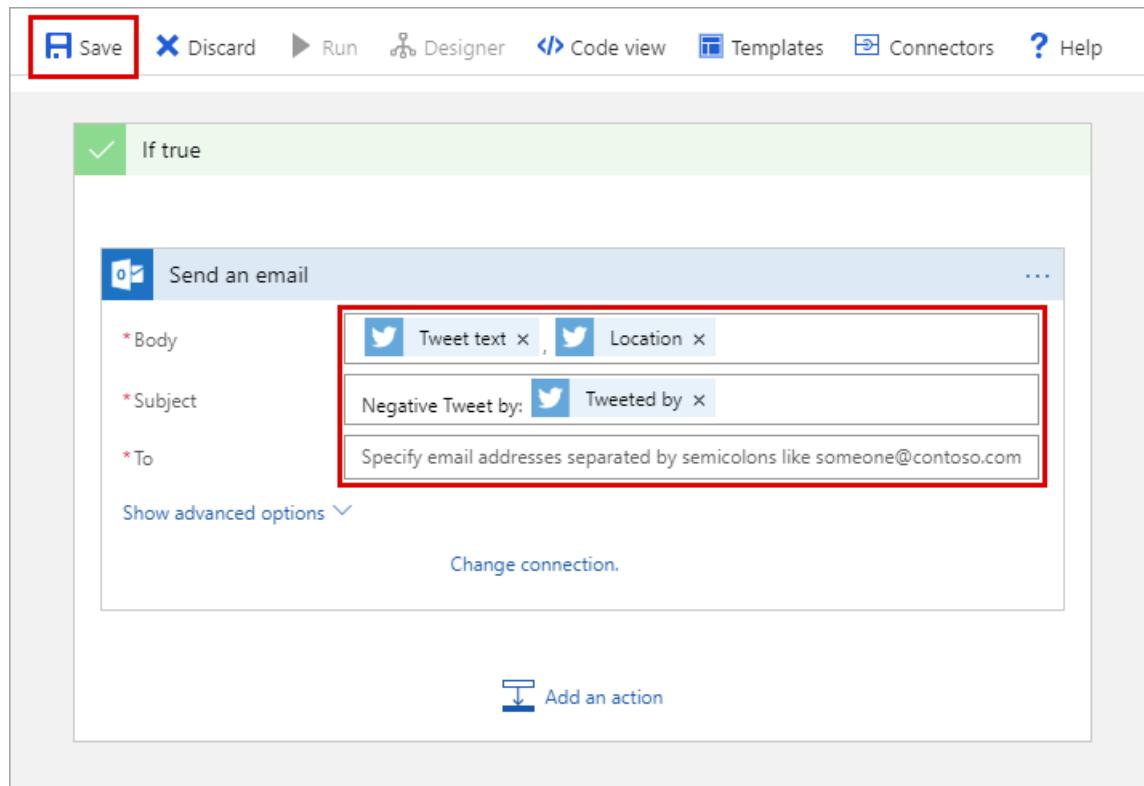
3. In **IF TRUE**, click **Add an action**, search for **outlook.com**, click **Send an email**, and sign in to your Outlook.com account.



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.

The screenshot shows the Azure Logic Apps portal. At the top, there are navigation icons: Run Trigger, Refresh, Edit, Delete, Disable, Update Schema, Clone, and Export. Below the header, the logic app 'Essentials' is selected. The 'Summary' section shows the trigger 'TWITTER' (When a new tweet is posted) and action 'COUNT' (4 actions). A link 'View in Logic Apps designer' is provided. The 'EVALUATION' section indicates 0 evaluations in the last 24 hours. The 'Runs history' section displays a table of execution logs for 11 runs on May 13, 2018, all of which succeeded. The table includes columns for Status, Start Time, Identifier, and Duration.

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.

The screenshot shows the Azure Functions logs viewer. The logs are displayed in a scrollable window with a light blue background. The log entries show the function starting, receiving a sentiment score, processing it, and then completing successfully.

```

Logs
Pause Clear Copy logs Expand
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");
```

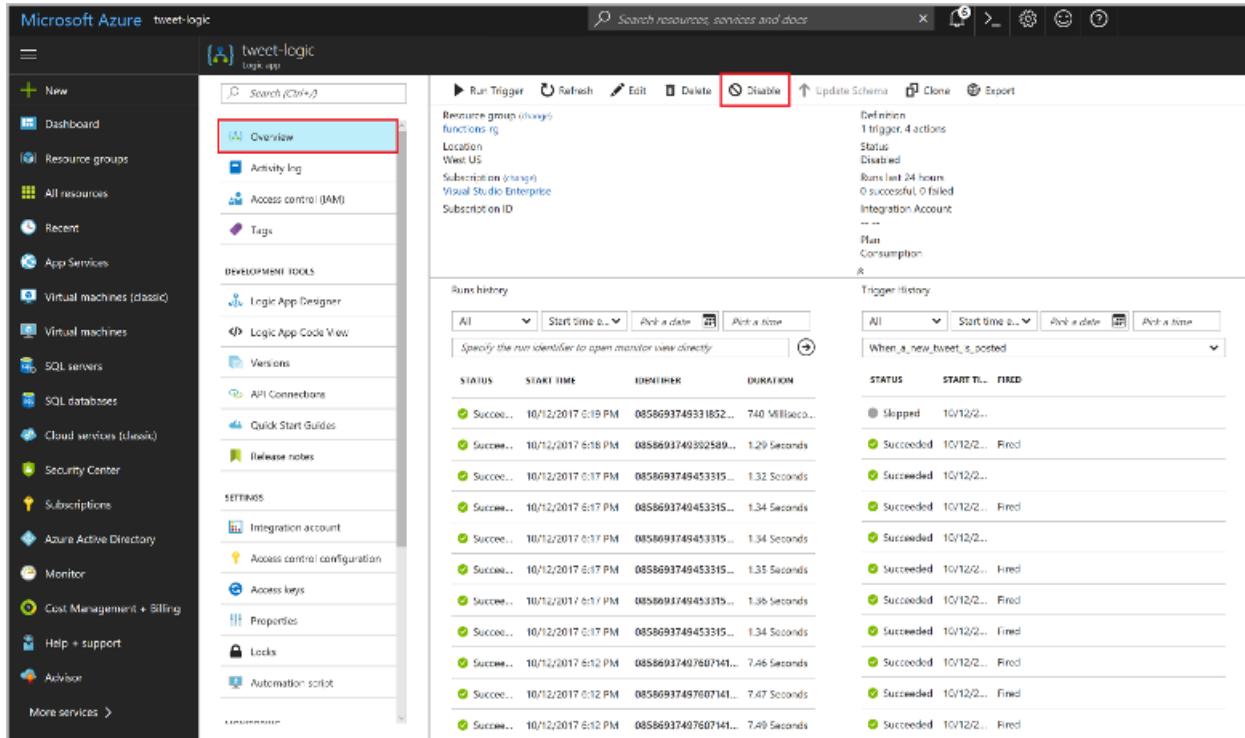
IMPORTANT

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 have 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 are buttons for Run Trigger, Refresh, Edit, Delete, and Disable (which is highlighted with a red box). To the right of these are Update Schema, Clone, and Export buttons. The main content area shows the logic app's definition, including triggers, actions, status (Disabled), and run history. The run history table lists multiple successful executions with start times ranging from 10/12/2017 6:17 PM to 10/12/2017 6:12 PM.

Status	Start Time	Identifier	Duration
Succeeded	10/12/2017 6:18 PM	085869374931852...	740 Milliseconds
Succeeded	10/12/2017 6:18 PM	0858693749392589...	1.29 Seconds
Succeeded	10/12/2017 6:17 PM	0858693749453315...	1.32 Seconds
Succeeded	10/12/2017 6:17 PM	0858693749453315...	1.34 Seconds
Succeeded	10/12/2017 6:17 PM	0858693749453315...	1.34 Seconds
Succeeded	10/12/2017 6:17 PM	0858693749453315...	1.35 Seconds
Succeeded	10/12/2017 6:17 PM	0858693749453315...	1.36 Seconds
Succeeded	10/12/2017 6:17 PM	0858693749453315...	1.34 Seconds
Succeeded	10/12/2017 6:12 PM	08586937497607141...	7.46 Seconds
Succeeded	10/12/2017 6:12 PM	08586937497607141...	7.47 Seconds
Succeeded	10/12/2017 6:12 PM	08586937497607141...	7.49 Seconds

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 a serverless API using Azure Functions

2/4/2019 • 6 minutes to read • [Edit Online](#)

In this tutorial, you will 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 tutorial, you will customize an HTTP trigger to handle specific actions in your API design. You will also prepare for growing your API by integrating it with Azure Functions Proxies and setting up mock APIs. All of this is 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 tutorial.

Sign in to Azure

Open the Azure portal. To do this, sign in to <https://portal.azure.com> with your Azure account.

Customize your HTTP function

By default, your HTTP-triggered function is configured to accept any HTTP method. There is also a default URL of the form `http://<yourapp>.azurewebsites.net/api/<funcname>?code=<functionkey>`. If you followed the quickstart, then `<funcname>` probably looks something like "HttpTriggerJS1". In this section, you will modify the function to respond only to GET requests against `/api/hello` route instead.

1. Navigate to your function in the Azure portal. Select **Integrate** in the left navigation.

The screenshot shows the Azure portal's left sidebar with the "ProxiesBackEnd" resource selected. Under the "Functions" section, the "HttpTriggerCSharp1" function is selected, and the "Integrate" option is highlighted. The main pane displays the "HTTP trigger" configuration. Key settings include:

- Allowed HTTP methods:** Selected methods (dropdown set to "Selected methods")
- Mode:** Standard
- Request parameter name:** req
- Route template:** /hello
- Authorization level:** Anonymous
- Selected HTTP methods:** GET (checkbox checked), POST, HEAD, PATCH, PUT, OPTIONS

At the bottom are "Save" and "Cancel" buttons.

2. Use the HTTP trigger settings as specified in the table.

FIELD	SAMPLE VALUE	DESCRIPTION
-------	--------------	-------------

FIELD	SAMPLE VALUE	DESCRIPTION
Allowed HTTP methods	Selected methods	Determines what HTTP methods may be used to invoke this function
Selected HTTP methods	GET	Allows only selected HTTP methods to be used to invoke this function
Route template	/hello	Determines what route is used to invoke this function
Authorization Level	Anonymous	Optional: Makes your function accessible without an API key

NOTE

Note that you did not include the `/api` base path prefix in the route template, as this is handled by a global setting.

3. Click **Save**.

You can learn more about customizing HTTP functions in [Azure Functions HTTP bindings](#).

Test your API

Next, test your function to see it working with the new API surface.

1. Navigate back to the development page by clicking on the function's name in the left navigation.
2. Click **Get function URL** and copy the URL. You should see that it uses the `/api/hello` route now.
3. Copy the URL into a new browser tab or your preferred REST client. Browsers will use GET by default.
4. Add parameters to the query string in your URL e.g. `/api/hello/?name=John`
5. Hit 'Enter' to confirm that it is 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 is not executed.
For this, you will need to use a REST client, such as cURL, Postman, or Fiddler.

Proxies overview

In the next section, you will 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, but instead of writing code to execute when that endpoint is called, you provide a URL to a remote implementation. This allows you to compose multiple API sources into a single API surface which is easy for clients to consume. This is particularly 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 will 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 will serve as the frontend for our API, and the function app you were previously editing will serve 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 key "HELLO_HOST". Set its value to the host of your backend function app, such as <YourBackendApp>.azurewebsites.net. This 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. Click **Save**.

Creating a proxy on the frontend

1. Navigate back to your frontend function app in the portal.
2. In the left-hand navigation, click the plus sign '+' next to "Proxies".

New proxy

Name

Route template

Allowed HTTP methods

Backend URL

[+ Request override](#)

[+ Response override](#)

Create

3. Use proxy settings as specified in the table.

FIELD	SAMPLE VALUE	DESCRIPTION
Name	HelloProxy	A friendly name used only for management
Route template	/api/remotehello	Determines what route is used to invoke this proxy

FIELD	SAMPLE VALUE	DESCRIPTION
Backend URL	https://%HELLO_HOST%/api/hello	Specifies the endpoint to which the request should be proxied

4. Note that Proxies does not provide the `/api` base path prefix, and this must be included in the route template.
5. The `%HELLO_HOST%` syntax will reference the app setting you created earlier. The resolved URL will point to your original function.
6. Click **Create**.
7. You can try out your new proxy by copying the Proxy URL and testing it in the browser or with your favorite HTTP client.
 - a. For an anonymous function use:
`https://YOURPROXYAPP.azurewebsites.net/api/remotehello?name="Proxies"`
 - b. For a function with authorization use:
`https://YOURPROXYAPP.azurewebsites.net/api/remotehello?code=YOURCODE&name="Proxies"`

Create a mock API

Next, you will use a proxy to create a mock API for your solution. This allows client development to progress, without needing the backend fully implemented. Later in development, you could create a new function app which supports this logic and redirect your proxy to it.

To create this mock API, we will 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**. Clicking this will open the App Service Editor in a new tab.

Select `proxies.json` in the left navigation. This is the file which stores the configuration for all of your proxies. If you use one of the [Functions deployment methods](#), this is the file you will maintain 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 the following:

```
{
  "$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:

```
{
    "$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 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 is particularly useful when proxying to a legacy system, where you might need to modify headers, query parameters, etc. 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 tutorial, 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 \(preview\)](#)

Deploy Python to Azure Functions with Visual Studio Code

7/30/2019 • 18 minutes to read • [Edit Online](#)

In this tutorial, you use Visual Studio Code and the Azure Functions extension to create a serverless HTTP endpoint with Python and to also add a connection (or "binding") to storage. Azure Functions runs your code in a serverless environment without needing to provision a virtual machine or publish a web app. The Azure Functions extension for Visual Studio Code greatly simplifies the process of using Functions by automatically handling many configuration concerns.

If you encounter issues with any of the steps in this tutorial, we'd love to hear about the details. Use the **I ran into an issue** button at the end of each section to submit detailed feedback.

Prerequisites

- An [Azure subscription](#).
- [Visual Studio Code with the Azure Functions extension](#).
- The [Azure Functions Core Tools](#).

Azure subscription

If you don't have an Azure subscription, [sign up now](#) for a free 30-day account with \$200 in Azure credits to try out any combination of services.

Visual Studio Code, Python, and the Azure Functions extension

Install the following software:

- Python 3.6.x as required by Azure Functions. [Python 3.6.8](#) is the latest 3.6.x version.
- [Visual Studio Code](#).
- The [Python extension](#) as described on [Visual Studio Code Python Tutorial - Prerequisites](#).
- The [Azure Functions extension](#). For general information, visit the [vscode-azurefunctions GitHub repository](#).

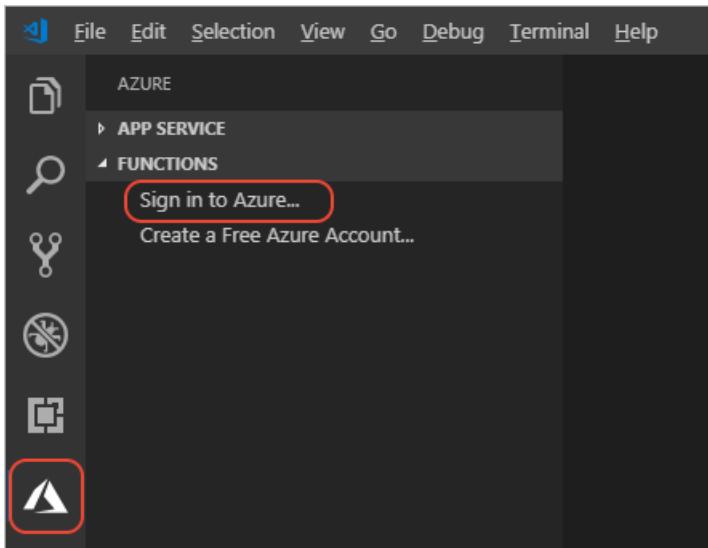
Azure Functions Core Tools

Follow the instructions for your operating system on [Work with Azure Functions Core Tools](#). The tools themselves are written in .NET Core, and the Core Tools package is best installed using the Node.js package manager, npm, which is why you need to install .NET Core and Node.js at present, even for Python code. You can, however bypass the .NET Core requirement using "extension bundles" as described in the aforementioned documentation.

Whatever the case, you need install these components only once, after which Visual Studio Code automatically prompts you to install any updates.

Sign in to Azure

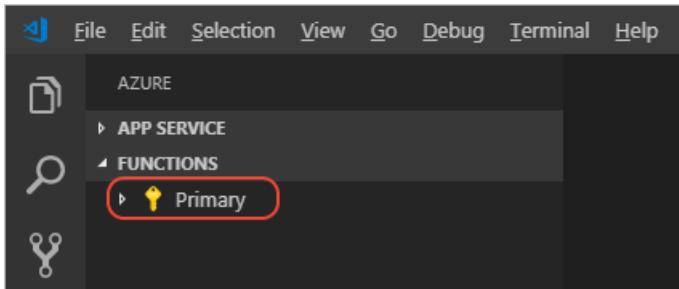
Once the Functions extension is installed, sign into your Azure account by navigating to the **Azure: Functions** explorer, select **Sign in to Azure**, and follow the prompts.



After signing in, verify that the email account of your Azure subscription appears in the Status Bar:



The name you've assigned to your subscription also appears in the **Azure: Functions** explorer ("Primary" in the image below):



NOTE

If you encounter the error "**Cannot find subscription with name [subscription ID]**", this may be because you are behind a proxy and unable to reach the Azure API. Configure `HTTP_PROXY` and `HTTPS_PROXY` environment variables with your proxy information in your terminal:

```
# macOS/Linux
export HTTPS_PROXY=https://username:password@proxy:8080
export HTTP_PROXY=http://username:password@proxy:8080
```

```
# Windows
set HTTPS_PROXY=https://username:password@proxy:8080
set HTTP_PROXY=http://username:password@proxy:8080
```

Verify prerequisites

To verify that all the Azure Functions tools are installed, open the Visual Studio Code Command Palette (F1), select the **Terminal: Create New Integrated Terminal** command, and once the terminal opens, run the command

```
func --version
```

```

PROBLEMS TERMINAL OUTPUT DEBUG CONSOLE 1: cmd + 

D:\Examples\Python\AzureFunctions>func

%%%%%%
%%%%%
@ %%%%%% @
@@ %%%%%% @@ 
@% %%%%%% @%
@@ %%%%%% @@%
@@ %%% @@@%
@@ %% @@@%
@% %
@

Azure Functions Core Tools (2.4.419 Commit hash: c9c1724d002bd90b2e6b41393915ea3a26bcf0ce)
Function Runtime Version: 2.0.12332.0
Usage: func [context] [context] <action> [-/-options]

```

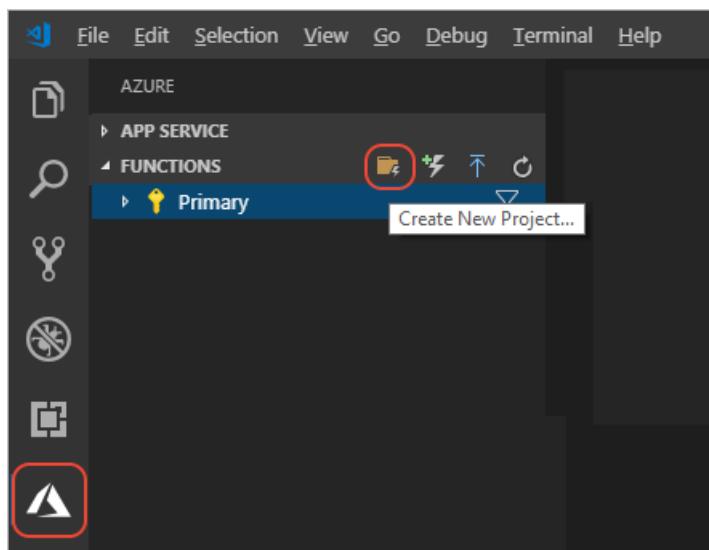
The output that starts with the Azure Functions logo (you need to scroll the output upwards) indicates that the Azure Functions Core Tools are present.

If the `func` command isn't recognized, then verify that the folder where you installed the Azure Functions Core Tools is included in your PATH environment variable.

[I ran into an issue](#)

Create the function

1. Code for Azure Functions is managed within a Functions *project*, which you create first before creating the code. In **Azure: Functions** explorer (opened using the Azure icon on the left side), select the **New Project** command icon, or open the Command Palette (F1) and select **Azure Functions: Create New Project**.

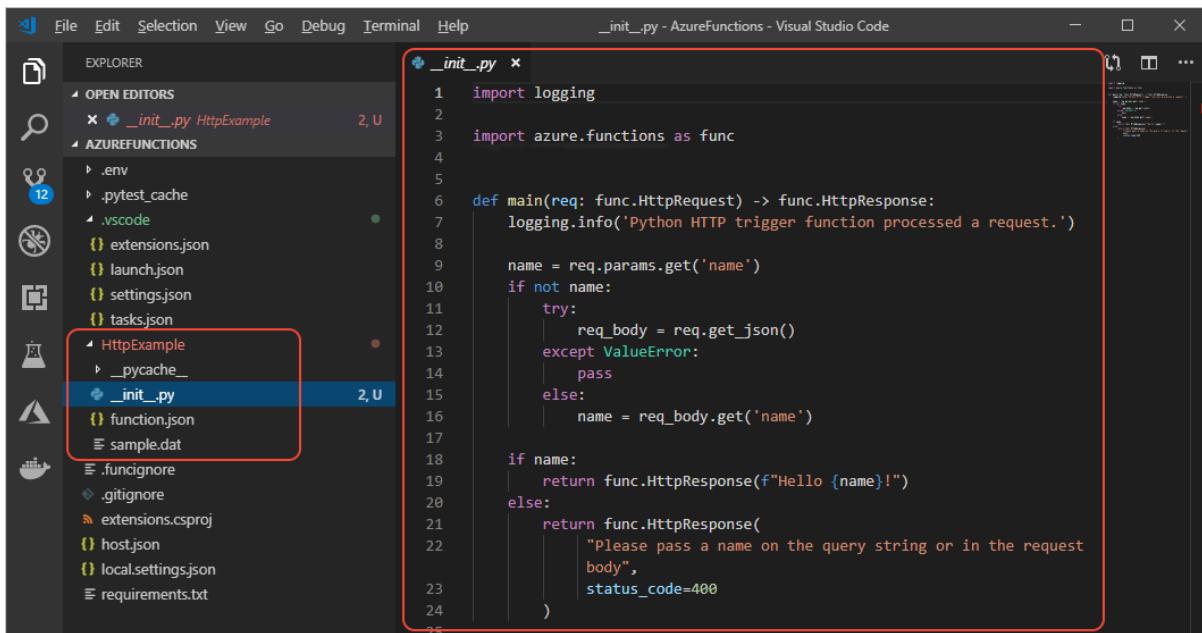


2. In the prompts that follow:

PROMPT	VALUE	DESCRIPTION
Specify a folder for the project	Current open folder	The folder in which to create the project. You may want to create the project in a subfolder.
Select a language for your function app project	Python	The language to use for the function, which determines the template used for the code.

PROMPT	VALUE	DESCRIPTION
Select a template for your project's first function	HTTP trigger	A function that uses an HTTP trigger is run whenever there's an HTTP request made to the function's endpoint. (There are a variety of other triggers for Azure Functions. To learn more, see What can I do with Functions? .)
Provide a function name	HttpExample	The name is used for a subfolder that contains the function's code along with configuration data, and also defines the name of the HTTP endpoint. Use "HttpExample" rather than accepting the default "HTTPTrigger" to distinguish the function itself from the trigger.
Authorization level	Anonymous	Anonymous authorization makes the function publicly accessible to anyone.
Select how you would like to open your project	Open in current window	Opens the project in the current Visual Studio Code window.

3. After a short time, a message to indicate that the new project was created. In the **Explorer**, there's the subfolder created for the function, and Visual Studio Code opens the `__init__.py` file that contains the default function code:



```

File Edit Selection View Go Debug Terminal Help _init_.py - AzureFunctions - Visual Studio Code

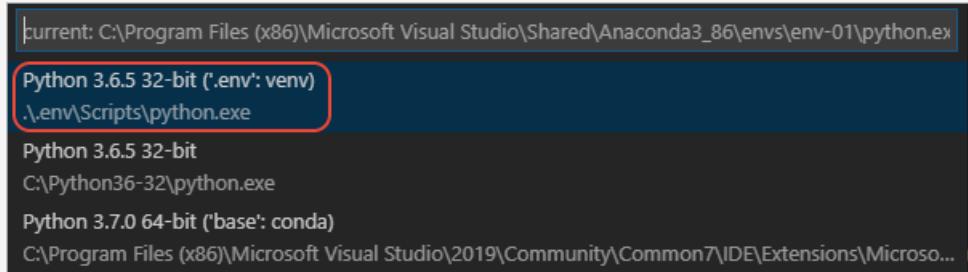
EXPLORER
OPEN EDITORS
x _init_.py HttpExample 2, U
AZUREFUNCTIONS
.env
.pytest_cache
vscode
extensions.json
launch.json
settings.json
tasks.json
HttpExample
._pycache_
_init_.py 2, U
function.json
sample.dat
.funcignore
.gitignore
extensions.csproj
host.json
local.settings.json
requirements.txt

_init_.py *
1 import logging
2
3 import azure.functions as func
4
5
6 def main(req: func.HttpRequest) -> func.HttpResponse:
7     logging.info('Python HTTP trigger function processed a request.')
8
9     name = req.params.get('name')
10    if not name:
11        try:
12            req_body = req.get_json()
13        except ValueError:
14            pass
15        else:
16            name = req_body.get('name')
17
18    if name:
19        return func.HttpResponse(f"Hello {name}!")
20    else:
21        return func.HttpResponse(
22            "Please pass a name on the query string or in the request
23            body",
24            status_code=400
25)

```

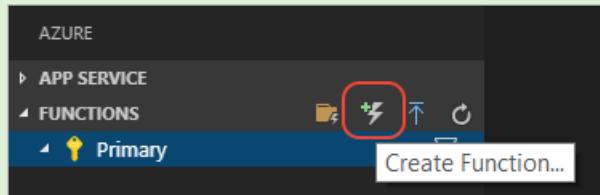
NOTE

If Visual Studio Code tells you that you don't have a Python interpreter selected when it opens `__init__.py`, open the Command Palette (F1), select the **Python: Select Interpreter** command, and then select the virtual environment in the local `.env` folder (which was created as part of the project). The environment must be based on Python 3.6x specifically, as noted earlier under [Prerequisites](#).



TIP

Whenever you want to create another function in the same project, use the **Create Function** command in the **Azure: Functions** explorer, or open the Command Palette (F1) and select the **Azure Functions: Create Function** command. Both commands prompt you for a function name (which is the name of the endpoint), then creates a subfolder with the default files.



I ran into an issue

Examine the code files

In the newly created function subfolder are three files: `__init__.py` contains the function's code, `function.json` describes the function to Azure Functions, and `sample.dat` is a sample data file. You can delete `sample.dat` if you want, as it exists only to show that you can add other files to the subfolder.

Let's look at `function.json` first, then the code in `__init__.py`.

function.json

The `function.json` file provides the necessary configuration information for the Azure Functions endpoint:

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

The `scriptFile` property identifies the startup file for the code, and that code must contain a Python function named `main`. You can factor your code into multiple files so long as the file specified here contains a `main` function.

The `bindings` element contains two objects, one to describe incoming requests, and the other to describe the HTTP response. For incoming requests (`"direction": "in"`), the function responds to HTTP GET or POST requests and doesn't require authentication. The response (`"direction": "out"`) is an HTTP response that returns whatever value is returned from the `main` Python function.

__init__.py

When you create a new function, Azure Functions provides default Python code in `__init__.py`:

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

The important parts of the code are as follows:

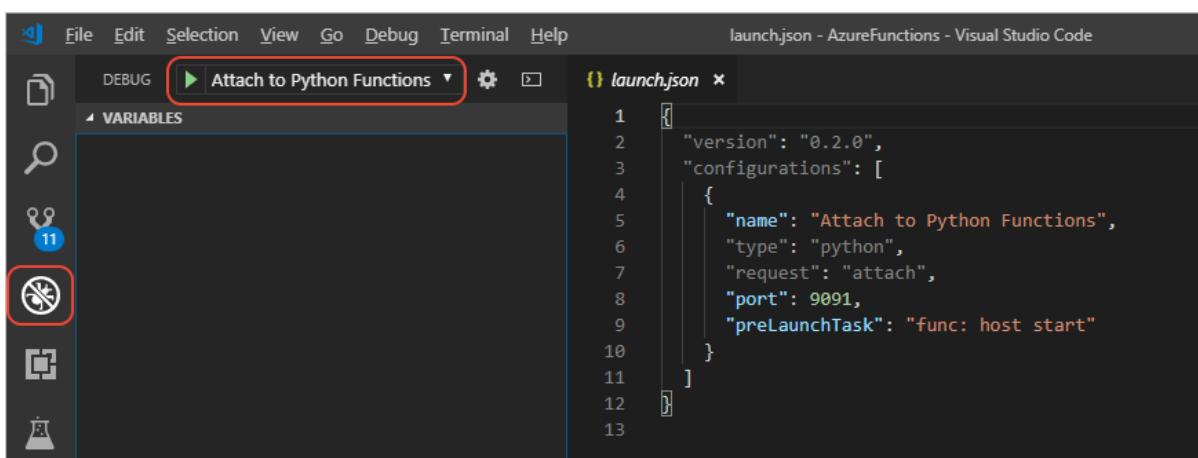
- You must import `func` from `azure.functions`; importing the `logging` module is optional but recommended.

- The required `main` Python function receives a `func.request` object named `req`, and returns a value of type `func.HttpResponse`. You can learn more about the capabilities of these objects in the `func.HttpRequest` and `func.HttpResponse` references.
- The body of `main` then processes the request and generates a response. In this case, the code looks for a `name` parameter in the URL. Failing that, it checks if the request body contains JSON (using `func.HttpRequest.get_json`) and that the JSON contains a `name` value (using the `get` method of the JSON object returned by `get_json`).
- If a name is found, the code returns the string "Hello" with the name appended; otherwise it returns an error message.

I ran into an issue

Debug locally

- When you create the Functions project, the Visual Studio Code extension also creates a launch configuration in `.vscode/launch.json` that contains a single configuration named **Attach to Python Functions**. This configuration means you can just press F5 or use the Debug explorer to start the project:



- When you start the debugger, a terminal opens showing output from Azure Functions, including a summary of the available endpoints. Your URL might be different if you used a name other than "HttpExample":

```

Http Functions:

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

```

- Use **Ctrl+click** or **Cmd+click** on the URL in the Visual Studio Code **Output** window to open a browser to that address, or start a browser and paste in the same URL. In either case, the endpoint is `api/<function_name>`, in this case `api/HttpExample`. However, because that URL doesn't include a name parameter, the browser window should just show, "Please pass a name on the query string or in the request body" as appropriate for that path in the code.

- Now try adding a name parameter to the URL, such as `http://localhost:7071/api/HttpExample?name=VS%20Code`, and the browser window should display the message, "Hello Visual Studio Code!", demonstrating that you've run that code path.
- To pass the name value in a JSON request body, you can use a tool like curl with the JSON inline:

```

# Mac OS/Linux: modify the URL if you're using a different function name
curl --header "Content-Type: application/json" --request POST \
--data {"name":"Visual Studio Code"} http://localhost:7071/api/HttpExample

```

```
# Windows (escaping on the quotes is necessary; also modify the URL
# if you're using a different function name)
curl --header "Content-Type: application/json" --request POST \
--data {"name":"Visual Studio Code"} http://localhost:7071/api/HttpExample
```

Alternately, create a file like `data.json` that contains `{"name": "Visual Studio Code"}` and use the command

```
curl --header "Content-Type: application/json" --request POST --data @data.json
http://localhost:7071/api/HttpExample
```

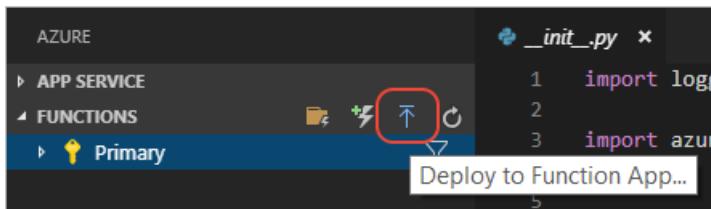
6. To test debugging the function, set a breakpoint on the line that reads `name = req.params.get('name')` and make a request to the URL again. The Visual Studio Code debugger should stop on that line, allowing you to examine variables and step through the code. (For a short walkthrough of basic debugging, see [Visual Studio Code Tutorial - Configure and run the debugger](#).)
7. When you're satisfied that you've thoroughly tested the function locally, stop the debugger (with the **Debug > Stop Debugging** menu command or the **Disconnect** command on the debugging toolbar).

[I ran into an issue](#)

Deploy to Azure Functions

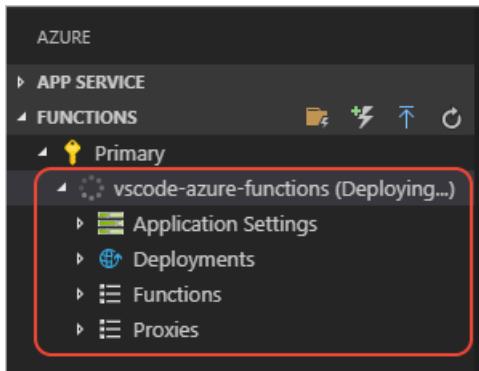
In these steps, you use the Functions extension to create a function app in Azure, along with other required Azure resources. A function app lets you group functions as a logic unit for easier management, deployment, and sharing of resources. It also requires an Azure Storage account for data and a [hosting plan](#). All of these resources are organized within a single resource group.

1. In the **Azure: Functions** explorer, select the **Deploy to Function App** command, or open the Command Palette (F1) and select the **Azure Functions: Deploy to Function App** command. Again, the function app is where your Python project runs in Azure.

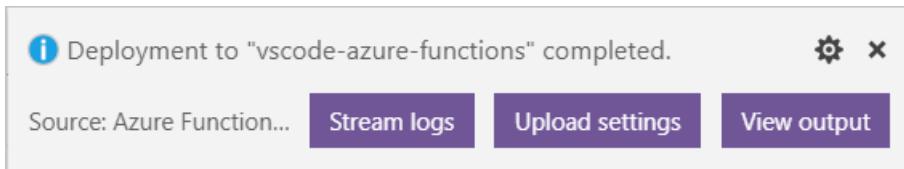


2. When prompted, select **Create New Function App in Azure**, and provide a name that's unique across Azure (typically using your personal or company name along with other unique identifiers; you can use letters, numbers, and hyphens). If you previously created a Function App, its name appears in this list of options.
3. The extension performs the following actions, which you can observe in Visual Studio Code popup messages and the **Output** window (the process takes a few minutes):
 - Create a resource group using the name you gave (removing hyphens).
 - In that resource group, create the storage account, hosting plan, and function app. By default, a [Consumption plan](#) is created. To run your functions in a dedicated plan, you need to [enable publishing with advanced create options](#).
 - Deploy your code to the function app.

The **Azure: Functions** explorer also shows progress:

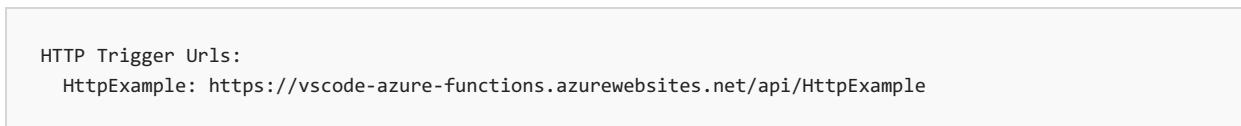


- Once deployment is complete, the Azure Functions extension displays a message with buttons for three additional actions:



For **Stream logs** and **Upload settings**, see the next sections. For **View output**, see step 5 that follows.

- After deployment, the **Output** window also shows the public endpoint on Azure:



Use this endpoint to run the same tests you did locally, using URL parameters and/or requests with JSON data in the request body. The results of the public endpoint should match the results of the local endpoint you tested earlier.

[I ran into an issue](#)

Stream logs

Support for log streaming is currently in development, as described on [Issue 589](#) for the Azure Functions extension. The **Stream logs** button in the deployment message popup will eventually connect the log output on Azure to Visual Studio Code. You will also be able to start and stop the log stream on the **Azure Functions** explorer by right-clicking the Functions project and selecting **Start streaming logs** or **Stop streaming logs**.

At present, however, these commands aren't yet operational. Log streaming is instead available in a browser by running the following command, replacing `<app_name>` with the name of your Functions app on Azure:

```
# Replace <app_name> with the name of your Functions app on Azure
func azure functionapp logstream <app_name> --browser
```

Sync local settings to Azure

The **Upload settings** button in the deployment message popup applies any changes you've made to your `local.settings.json` file to Azure. You can also invoke the command on the **Azure Functions** explorer by expanding the Functions project node, right-clicking **Application Settings**, and selecting **Upload local settings...**. You can also use the Command Palette to select the **Azure Functions: Upload Local Settings** command.

Uploading settings updates any existing settings and adds any new settings defined in `local.settings.json`. Uploading doesn't remove any settings from Azure that aren't listed in the local file. To remove those settings, expand the **Applications Settings** node in the **Azure Functions** explorer, right-click the setting, and select **Delete Setting...**. You can also edit settings directly on the Azure portal.

To apply any changes you make through the portal or through the **Azure Explorer** to the *local.settings.json* file, right-click the **Application Settings** node and select the **Download remote settings...** command. You can also use the Command Palette to select the **Azure Functions: Download Remote Settings** command.

Add a second Function

After your first deployment, you can make changes to your code, such as adding additional functions, and redeploy to the same Functions App.

1. In the **Azure: Functions** explorer, select the **Create Function** command or use **Azure Functions: Create Function** from the Command Palette. Specify the following details for the function:

- Template: HTTP trigger
- Name: "DigitsOfPi"
- Authorization level: Anonymous

2. In the Visual Studio Code file explorer is a subfolder with your function name that again contains files named *_init_.py*, *function.json*, and *sample.dat*.
3. Replace the contents of *_init_.py* to match the following code, which generates a string containing the value of PI to a number of digits specified in the URL (this code uses only a URL parameter)

```

import logging

import azure.functions as func

""" Adapted from the second, shorter solution at
http://www.codexcodex.com/wiki/Calculate_digits_of_pi#Python
"""

def pi_digits_Python(digits):
    scale = 10000
    maxarr = int((digits / 4) * 14)
    arrinit = 2000
    carry = 0
    arr = [arrinit] * (maxarr + 1)
    output = ""

    for i in range(maxarr, 1, -14):
        total = 0
        for j in range(i, 0, -1):
            total = (total * j) + (scale * arr[j])
            arr[j] = total % ((j * 2) - 1)
            total = total / ((j * 2) - 1)

        output += "%04d" % (carry + (total / scale))
        carry = total % scale

    return output;

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('DigitsOfPi HTTP trigger function processed a request.')

    digits_param = req.params.get('digits')

    if digits_param is not None:
        try:
            digits = int(digits_param)
        except ValueError:
            digits = 10    # A default

        if digits > 0:
            digit_string = pi_digits_Python(digits)

            # Insert a decimal point in the return value
            return func.HttpResponse(digit_string[:1] + '.' + digit_string[1:])

    return func.HttpResponse(
        "Please pass the URL parameter ?digits= to specify a positive number of digits.",
        status_code=400
    )

```

- Because the code supports only HTTP GET, modify *function.json* so that the `"methods"` collection contains only `"get"` (that is, remove `"post"`). The whole file should appear as follows:

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

- Start the debugger by pressing F5 or selecting the **Debug > Start Debugging** menu command. The **Output** window should now show both endpoints in your project:

```
Http Functions:

DigitsOfPi: [GET] http://localhost:7071/api/DigitsOfPi

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

- In a browser, or from curl, make a request to `http://localhost:7071/api/DigitsOfPi?digits=125` and observe the output. (You might notice that the code algorithm isn't entirely accurate, but we'll leave the improvements to you!) Stop the debugger when you're finished.
- Redeploy the code by using the **Deploy to Function App** in the **Azure: Functions** explorer. If prompted, select the Function App created previously.
- Once deployment finishes (it takes a few minutes!), the **Output** window shows the public endpoints with which you can repeat your tests.

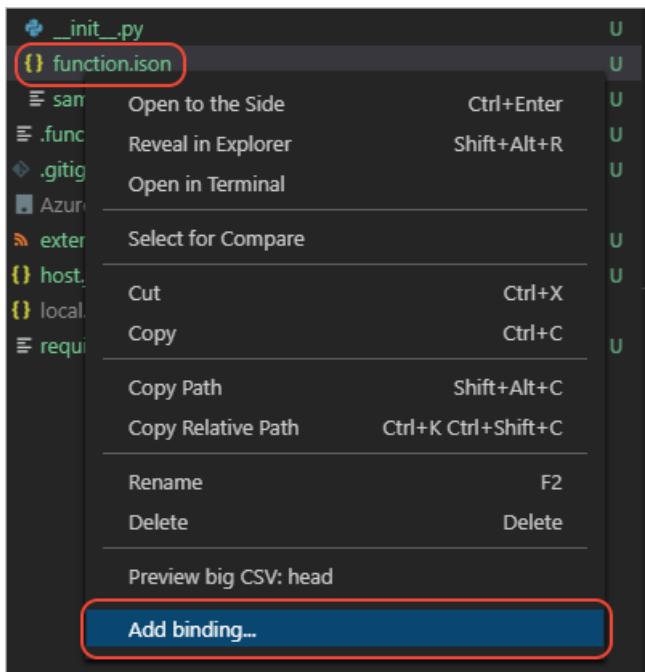
[I ran into an issue](#)

Add a binding to write messages to Azure storage

A *binding* let you connect your function code to resources, such as Azure storage, without writing any data access code. A binding is defined in the `function.json` file and can represent both input and output. A function can use multiple input and output bindings, but only one trigger. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this section, you add a storage binding to the `HttpExample` function created earlier in this tutorial. The function uses this binding to write messages to storage with each request. The storage in question uses the same default storage account used by the function app. If you plan on making heavy use of storage, however, you would want to consider creating a separate account.

- Sync the remote settings for your Azure Functions project into your `local.settings.json` file by opening the Command Palette and selecting **Azure Functions: Download Remote Settings**. Open `local.settings.json` and check that it contains a value for `AzureWebJobsStorage`. That value is the connection string for the storage account.
- In the `HttpExample` folder, right-click the `function.json`, select **Add binding**:



3. In the prompts that follow in Visual Studio Code, select or provide the following values:

PROMPT	VALUE TO PROVIDE
Set binding direction	out
Select binding with direction out	Azure Queue Storage
The name used to identify this binding in your code	msg
The queue to which the message will be sent	outqueue
Select setting from <i>local.settings.json</i> (asking for the storage connection)	AzureWebJobsStorage

4. After making these selections, verify that the following binding is added to your *function.json* file:

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

5. Now that you've configured the binding, you can use it in your function code. Again, the newly defined binding appears in your code as an argument to the `main` function in `__init__.py`. For example, you can modify the `__init__.py` file in `HttpExample` to match the following, which shows using the `msg` argument to write a timestamped message with the name used in the request. The comments explain the specific changes:

```

import logging
import datetime # MODIFICATION: added import
import azure.functions as func

# MODIFICATION: the added binding appears as an argument; func.Out[func.QueueMessage]
# is the appropriate type for an output binding with "type": "queue" (in function.json).
def main(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        # MODIFICATION: write the a message to the message queue, using msg.set
        msg.set(f"Request made for {name} at {datetime.datetime.now()}")

        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
        )
)

```

6. To test these changes locally, start the debugger again in Visual Studio Code by pressing F5 or selecting the **Debug > Start Debugging** menu command. As before the **Output** window should show the endpoints in your project.
7. In a browser, visit the URL <http://localhost:7071/api/HttpExample?name=VS%20Code> to create a request to the HttpExample endpoint, which should also write a message to the queue.
8. To verify that the message was written to the "outqueue" queue (as named in the binding), you can use one of three methods:
 - a. Sign into the [Azure portal](#), and navigate to the resource group containing your functions project. Within that resource group, local and navigate into the storage account for the project, then navigate into **Queues**. On that page, navigate into "outqueue", which should display all the logged messages.
 - b. Navigate and examine the queue with either the Azure Storage Explorer, which integrates with Visual Studio, as described on [Connect Functions to Azure Storage using Visual Studio Code](#), especially the [Examine the output queue](#) section.
 - c. Use the Azure CLI to query the storage queue, as described on [Query the storage queue](#).
9. To test in the cloud, redeploy the code by using the **Deploy to Function App** in the **Azure: Functions** explorer. If prompted, select the Function App created previously. Once deployment finishes (it takes a few minutes!), the **Output** window again shows the public endpoints with which you can repeat your tests.

[I ran into an issue](#)

Clean up resources

The Function App you created includes resources that can incur minimal costs (refer to [Functions Pricing](#)). To clean up the resources, right-click the Function App in the **Azure: Functions** explorer and select **Delete Function App**. You can also visit the [Azure portal](#), select **Resource groups** from the left-side navigation pane, select the resource group that was created in the process of this tutorial, and then use the **Delete resource group** command.

Next steps

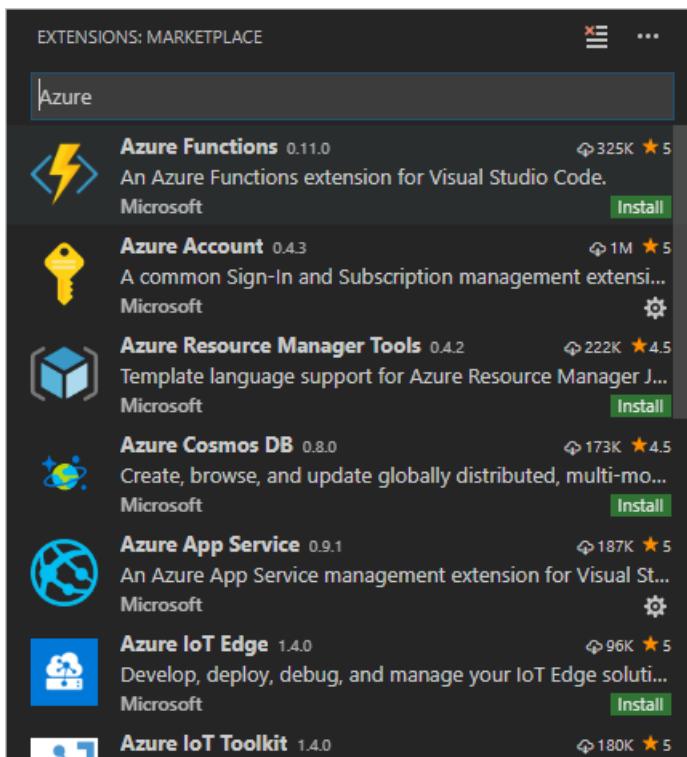
Congratulations on completing this walkthrough of deploying Python code to Azure Functions! You're now ready to create many more serverless functions.

As noted earlier, you can learn more about the Functions extension by visiting its GitHub repository, [vscode-azurefunctions](#). Issues and contributions are also welcome.

Read the [Azure Functions Overview](#) to explore the different triggers you can use.

To learn more about Azure services that you can use from Python, including data storage along with AI and Machine Learning services, visit [Azure Python Developer Center](#).

There are also other Azure extensions for Visual Studio Code that you may find helpful. Just search on "Azure" in the Extensions explorer:



Some popular extensions are:

- [Cosmos DB](#)
- [Azure App Service](#)
- [Azure CLI Tools](#)
- [Azure Resource Manager Tools](#)

Create an OpenAPI definition for a function with Azure API Management

5/9/2019 • 6 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.

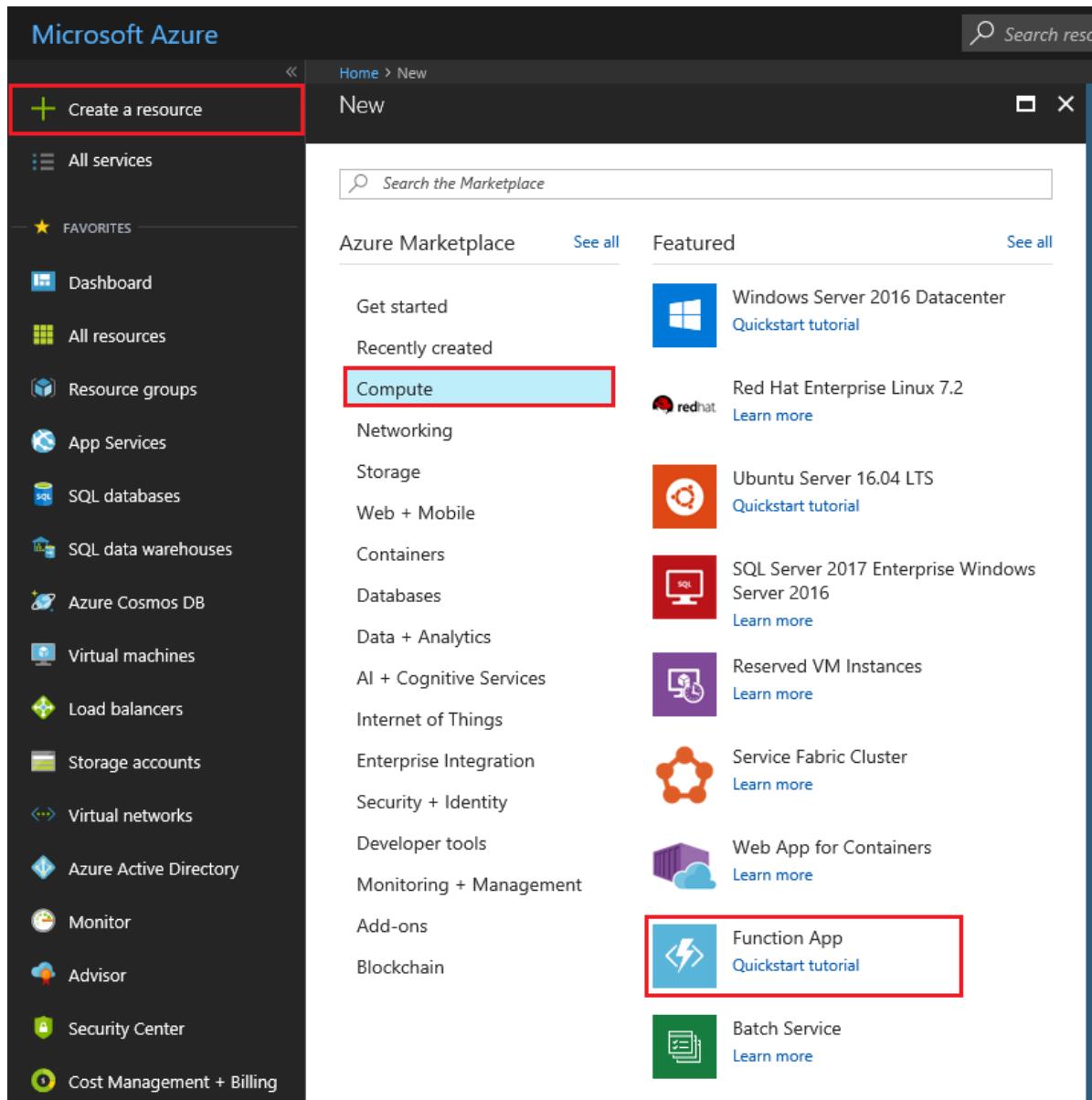
In this tutorial, you learn how to:

- 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. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.



2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
myfunctionapp .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
myResourceGroup ✓

* OS
 Windows Linux

* Hosting Plan ⓘ
Consumption Plan

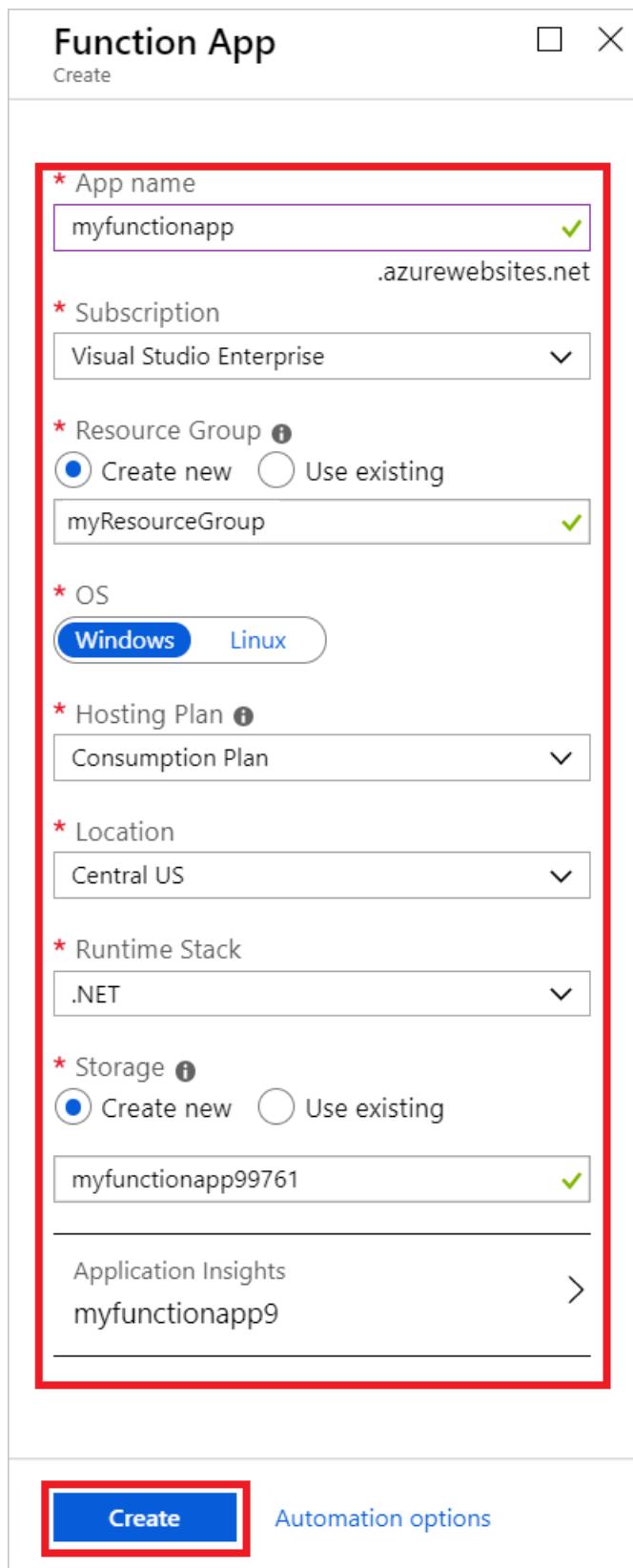
* Location
Central US

* Runtime Stack
.NET

* Storage ⓘ
 Create new Use existing
myfunctionapp99761 ✓

Application Insights >
myfunctionapp9

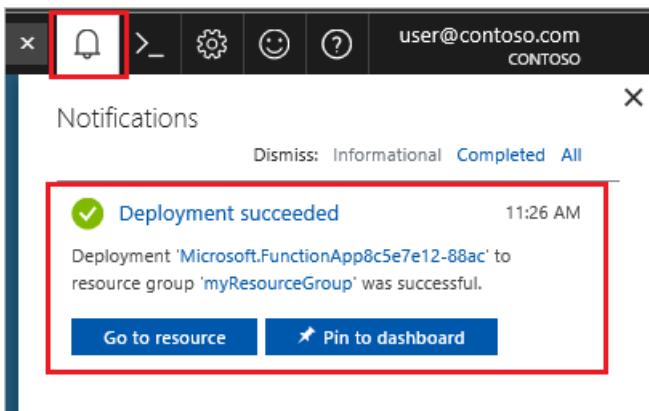
Create Automation options



SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z , 0-9 , and - .
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting on Linux is currently in preview. For more information, see this considerations article .
Hosting plan	Consumption plan	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 only pay for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .
Location	West Europe	Choose a region near you or near other services your functions access.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.
Storage	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 .
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 where you want to store your data.

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



5. 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 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. Expand your function app and select the + button next to **Functions**. Select **In-portal > Continue**.
2. Select **More templates...**, then select **Finish and view templates**
3. Select HTTP trigger, type `TurbineRepair` for the function **Name**, choose `Function` for **Authentication level**, and then select **Create**.



HTTP trigger

New Function

Name:

TurbineRepair

HTTP trigger

Authorization level i

Function

Create

Cancel

4. Replace the contents of the run.csx C# script file with the following code, then choose **Save**:

```

#r "Newtonsoft.Json"

using System;
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, as well as the revenue opportunity that the turbine represents, and the cost to fix the turbine.

- To test the function, click **Test** at the far right to expand the test tab. Enter the following value for the **Request body**, and then click **Run**.

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

The screenshot shows the Azure Functions Test interface. On the left, the code for 'run.csx' is displayed:

```

1 using System.Net;
2
3 const double revenuePerkW = 0.12;
4 const double technicianCost = 250;
5 const double turbineCost = 100;
6
7 //Summary: Check cost effectiveness of repair
8 //Description: This operation determines if a technician should be sent for an emergency repair
9 //Operation ID: check_cost_effectiveness
10
11 public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
12 {
13
14     //Get request body
15     dynamic data = await req.Content.ReadAsAsync<object>();
16     int hours = data.hours;
17     int capacity = data.capacity;
18
19     //Formulas to calculate revenue and cost
20     double revenueOpportunity = capacity * revenuePerkW * 24;
21     double costToFix = (hours * technicianCost) + turbineCost;
22     string repairTurbine;
23
24     if (revenueOpportunity > costToFix){
25         repairTurbine = "Yes";
26     }
27     else {
28         repairTurbine = "No";
29     }
30
31     return req.CreateResponse(HttpStatusCode.OK, new{
32         message = repairTurbine,
33         revenueOpportunity = "$"+ revenueOpportunity,
34         costToFix = "$"+ costToFix
35     });
36 }
37

```

The 'Test' tab is selected. The 'HTTP method' is set to 'POST'. The 'Request body' contains the following JSON:

```

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

```

The 'Output' section shows the response: 'Status: 200 OK' with the JSON object:

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

The following value is returned in the body of the response.

```
{"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

Now you're ready to generate the OpenAPI definition.

1. Select the function app, then in **Platform features**, choose **API Management** and select **Create new** under **API Management**.

The screenshot shows the 'Platform features' section of the Azure portal. The 'API Management' option is highlighted with a red box.

General Settings	Networking	API
Function app settings	Networking	API Management
Configuration	SSL	API definition
Properties	Custom domains	CORS
Backups	Authentication / Authorization	App Service plan
All settings	Identity	App Service plan
Code Deployment	Push notifications	Scale up
Deployment Center		Scale out
Monitoring		

2. Use the API Management settings as specified in the table below the image.

API Management service



* Name



.azure-api.net

* Subscription



* Resource group



[Create new](#)

* Location



* Organization name ⓘ



* Administrator email ⓘ



Pricing tier ([View full pricing details](#))

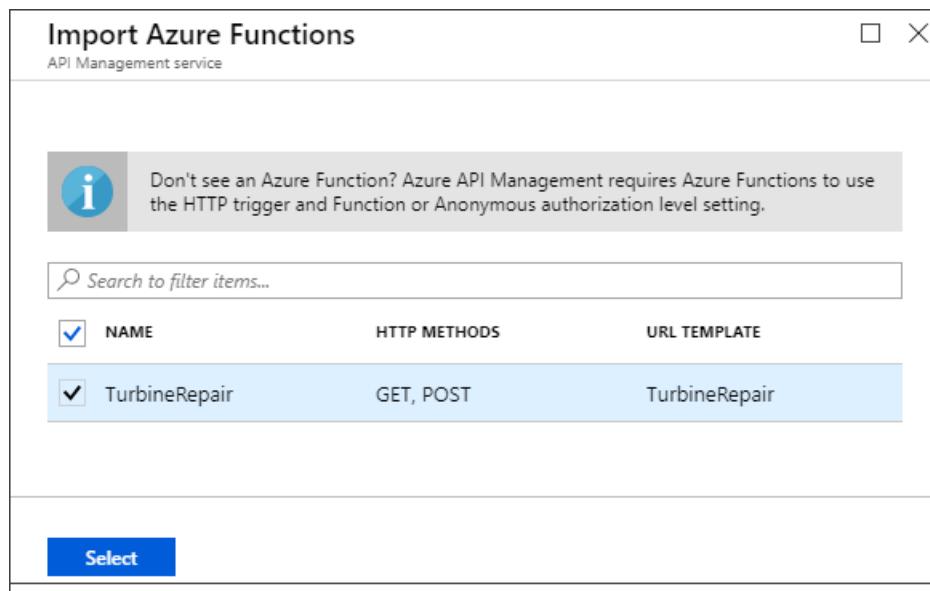


Create

[Automation options](#)

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.
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 (preview)	Consumption tier is in preview and isn't available in all regions. For complete pricing details, see the API Management pricing page

3. Choose **Create** to create the API Management instance, which may take several minutes.
4. Select **Enable Application Insights** to send logs to the same place as the function application, then accept the remaining defaults and select **Link API**.
5. The **Import Azure Functions** opens with the **TurbineRepair** function highlighted. Choose **Select** to continue.



6. In the **Create from Function App** page, accept the defaults and select **Create**

The screenshot shows the 'Create from Function App' dialog box. It has a 'Basic | Full' tab selection. There are three main input fields:

- * Function App: The value 'myfunctionapp' is entered.
- * Display name: The value 'myfunctionapp' is entered.
- * Name: The value 'myfunctionapp' is entered.

Below these are two optional fields:

- API URL suffix: The value 'myfunctionapp' is entered.
- Products: A message says 'No products selected'.

At the bottom right are 'Create' and 'Cancel' buttons, with 'Create' being highlighted with a red border.

The API is now created for the function.

Test the API

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

1. On the **Test** tab of your function, select **POST** operation.
2. Enter values for **hours** and **capacity**

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

3. Click **Send**, then view the HTTP response.

The screenshot shows the Azure portal interface for managing APIs. The left sidebar has a tree view with 'ContosoAPIM - API Management' selected. Under 'API', 'API Management' is also selected. The main area shows the 'TurbineRepair' API. The 'Test' tab is active. It displays a 'Query parameters' table with columns NAME, VALUE, TYPE, and DESCRIPTION. A 'POST' request is selected. In the 'Request body' section, the raw JSON input is:

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

Below the request body, there are sections for 'Apply product scope' (No products), 'Request URL', and 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. Open the downloaded JSON file and 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 left menu in the Azure portal, select **Resource groups** and then select **myResourceGroup**.

On the resource group page, make sure that the listed resources are the ones you want to delete.

Select **Delete**, type **myResourceGroup** in the text box, 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

7/7/2019 • 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.

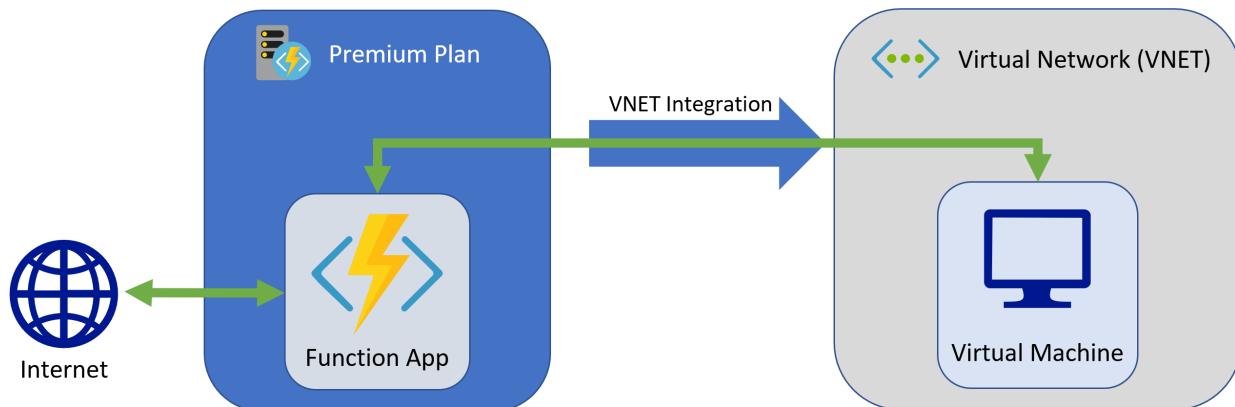
- Create a function app in the Premium plan
- Deploy a WordPress site to VM in a virtual network
- Connect the function app to the virtual network
- Create a function proxy to access WordPress resources
- Request a WordPress file from inside the virtual network

NOTE

This tutorial creates a function app in the Premium plan. This hosting plan is currently in preview. For more information, see [Premium plan](#).

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. Go to the [Azure portal](#).
2. Select **+ Create a resource** on the left hand side, then choose **Function app**.
3. For **Hosting plan**, choose **App Service plan**, then select **App Service plan/Location**.

The screenshot shows the Azure portal interface. On the left, the sidebar has a 'Create a resource' button highlighted with a red box. The main area shows a 'New' blade with a search bar and a 'Popular' section containing various service quickstart tutorials. On the right, a 'Function App' creation dialog is open. It includes fields for 'App name' (with a placeholder 'Enter a name for your App' and a suffix '.azurewebsites.net'), 'Subscription' (set to 'Visual Studio Enterprise'), 'Resource Group' (radio buttons for 'Create new' or 'Use existing'), 'OS' (set to 'Windows'), 'Hosting Plan' (set to 'App Service Plan', highlighted with a red box), 'App Service plan/Location' (set to 'Premium-Plan(West Europe)', highlighted with a red box), 'Runtime Stack' (set to '.NET'), 'Storage' (radio buttons for 'Create new' or 'Use existing'), and 'Application Insights'.

4. Select **Create new**, type an **App Service plan** name, choose a **Location** in a **region** near you or near other services your functions access, and then select **Pricing tier**.

The screenshot shows two dialogs. The left one is titled 'App Service plan' with a note about what it is. The right one is titled 'New App Service Plan' with a note to 'Create a plan for the web app'. Both dialogs have 'OK' buttons at the bottom. The 'App Service plan' dialog has a 'Create new' button highlighted with a red box. The 'New App Service Plan' dialog has three fields highlighted with red boxes: 'App Service plan' (set to 'Premium-Plan'), 'Location' (set to 'West Europe'), and 'Pricing tier' (set to 'EP1 Elastic Premium').

5. Choose the **EP1** (elastic Premium) plan, then select **Apply**.



Dev / Test

For less demanding workloads



Production

For most production workloads



Isolated

Advanced networking and scale

Recommended pricing tiers

S1

100 total ACU
1.75 GB memory
A-Series compute equivalent
44.64 USD/Month (Estimated)

P1V2

210 total ACU
3.5 GB memory
Dv2-Series compute equivalent
148.80 USD/Month (Estimated)

P2V2

420 total ACU
7 GB memory
Dv2-Series compute equivalent
297.60 USD/Month (Estimated)

P3V2

840 total ACU
14 GB memory
Dv2-Series compute equivalent
595.20 USD/Month (Estimated)

EP1

210 total ACU
3.5 GB memory
Dv2-Series compute equivalent
148.80 USD/Month (Estimated)

EP2

420 total ACU
7 GB memory
Dv2-Series compute equivalent
297.60 USD/Month (Estimated)

EP3

840 total ACU
14 GB memory
Dv2-Series compute equivalent
595.20 USD/Month (Estimated)

▼ See additional options

Apply

6. Select **OK** to create the plan, then use the remaining function app settings as specified in the table below the image.

Function App

Create

App name VNET-Function.azurewebsites.net

Subscription Visual Studio Enterprise

Resource Group Create new myResourceGroup

OS Windows

Hosting Plan App Service Plan

App Service plan/Location Premium-Plan(West Europe)

Runtime Stack .NET

Storage Create new vnetfunction1bed7

Application Insights VNET-Function

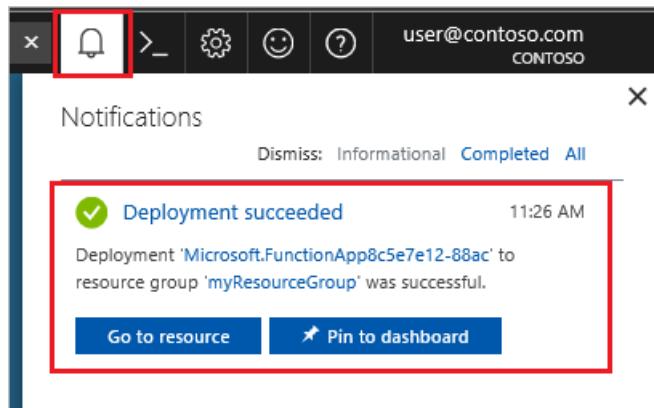
Create

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z, 0-9, and -.
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app. You can also use the suggested value.
OS	Preferred OS	Both Linux and Windows are supported on the Premium plan.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions. Only languages supported on your chosen OS are displayed.
Storage	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 .
Application Insights	Default	Creates an Application Insights resource of the same App name 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 where you want to store your data.

7. After your settings are validated, select **Create**.

8. Select the Notification 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:

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.

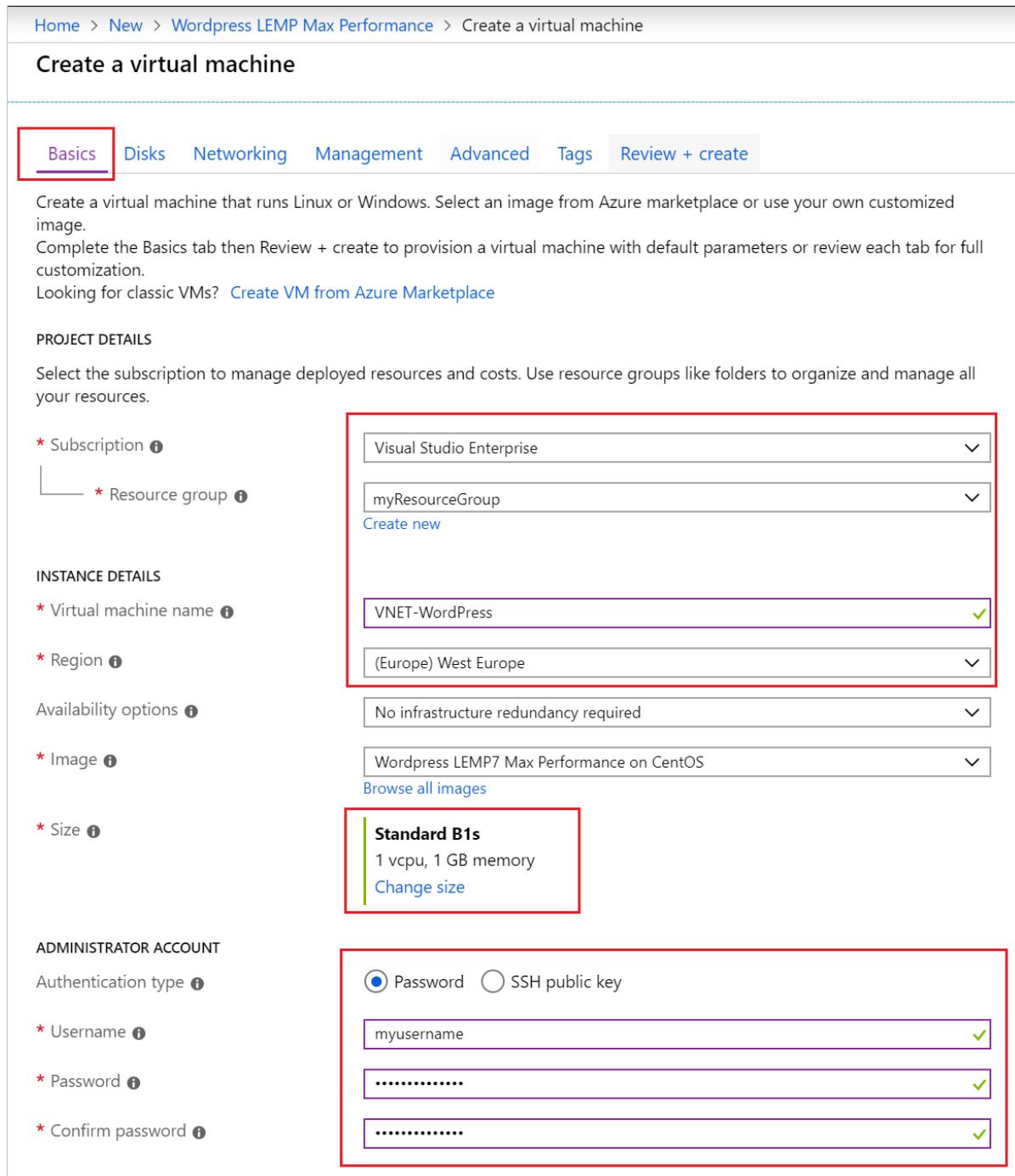
* Subscription [Visual Studio Enterprise](#)
└─ * Resource group [myResourceGroup](#) [Create new](#)

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.
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 are 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
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.
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.

VNET-Wordpress - Networking

Network Interface: vnet-wordpress178 Effective security rules Topology

Virtual network/subnet: myResourceGroup-vnet/Tutorial-Net Public IP: **None** Private IP: **10.10.1.4**

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 **Platform features > Networking**.

VNET-Function

Function Apps

Overview Platform features

General Settings Networking

- Function app settings
- Application settings
- Properties
- Backups
- All settings
- Networking
- SSL
- Custom domains
- Authentication / Authorization
- Identity
- Push notifications

2. Under **VNet Integration**, select **Click here to configure**.

Network Feature Status

VNET-Function

VNet Integration

Securely access resources available in or through your Azure VNet [Learn More](#)

Click here to configure

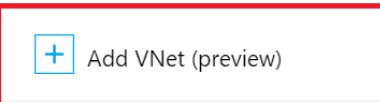
3. On the virtual network integration page, select **Add VNet (preview)**.

 **VNet Integration**
vnet-function

 Disconnect  Refresh

 **VNet Configuration**

Securely access resources available in or through your Azure VNet [Learn more](#)



 Add VNet (preview)

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



This VNet Integration experience is in preview.

Virtual Network

myResourceGroup-vnet (northeurope)

Subnet

Create New Subnet Select Existing

* Subnet Name

Function-Net

Virtual Network Address Block

10.10.0.0/16

* Subnet Address Block

10.10.2.0/24

SUBNET NAME

ADDRESS RANGE

Tutorial-Net

10.10.1.0 - 10.10.1.255

OK

SETTING	SUGGESTED VALUE	DESCRIPTION
Virtual Network	MyResourceGroup-vnet	This virtual network is the one you created earlier.
Subnet	Create New Subnet	Create a subnet in the virtual network for your function app to use. VNet Integration must be configured to use an empty subnet. 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.

5. 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.

1. In your function app, select **Proxies** > **+**, then use the proxy settings in the table below the image:

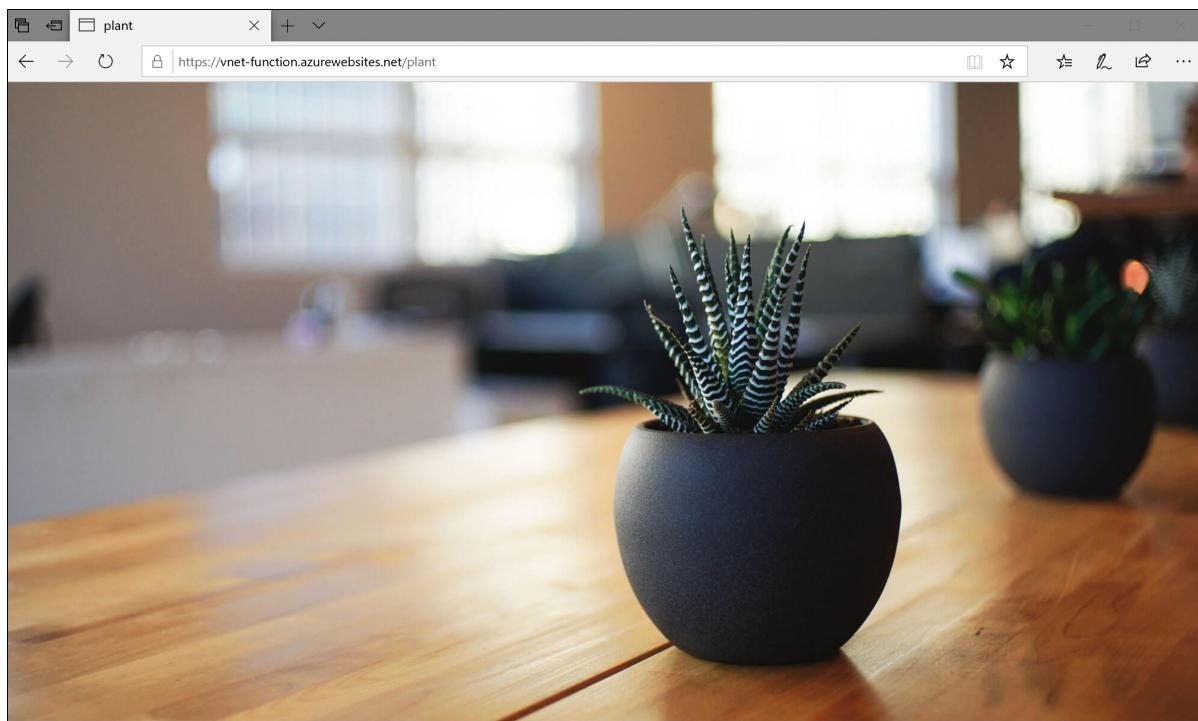
SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Plant	The name can be any value. It's used to identify the proxy.
Route Template	/plant	Route that maps to a VM resource.

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

2. Select **Create** to add the proxy to your function app.

Try it out

1. 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 left menu in the Azure portal, select **Resource groups** and then select **myResourceGroup**.

On the resource group page, make sure that the listed resources are the ones you want to delete.

Select **Delete**, type **myResourceGroup** in the text box, 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: Automate resizing uploaded images using Event Grid

4/16/2019 • 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](#)
- [Node.js V2 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, the text "Drop files here or click to upload." is displayed. Below this is another 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 a general 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.

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
```

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. Cloud Shell lets you use either `bash` or `PowerShell` to work with Azure services. You can use the Cloud Shell pre-installed commands to run the code in this article without having to install anything on your local environment.

To launch 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 top-right menu bar in the Azure portal .	

To run the code in this article in Azure Cloud Shell:

1. Launch Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session with **Ctrl+Shift+V** on Windows and Linux, or **Cmd+Shift+V** on macOS.
4. Press **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`.

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
```

2. 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>
```

3. Create the storage account for the Azure function.

```
az storage account create --name $functionstorage --location southeastasia \
--resource-group $resourceGroupName --sku Standard_LRS --kind storage
```

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>
```

2. Create the Azure function.

```
az functionapp create --name $functionapp --storage-account $functionstorage \
--resource-group $resourceGroupName --consumption-plan-location southeastasia
```

Now you must 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](#)
- [Node.js V2 SDK](#)
- [Node.js V10 SDK](#)

```
blobStorageAccount=<name of the Blob storage account you created in the previous tutorial>
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](#)
- [Node.js V2 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
```

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](#)
- [Node.js V2 SDK](#)
- [Node.js V10 SDK](#)

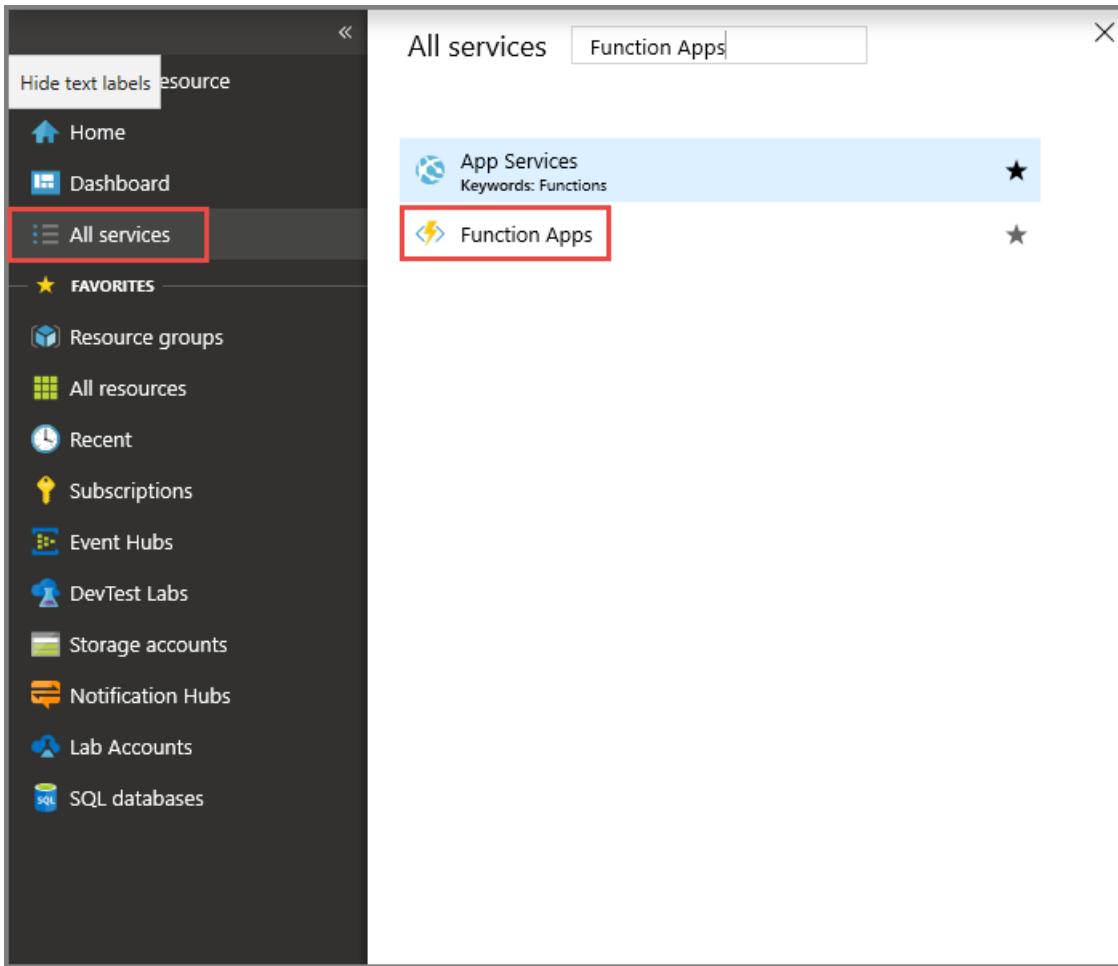
To learn more about this function, see the [function.json and run.csx files](#).

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](#), select **All Services** on the left menu, and then select **Function Apps**.



2. Expand your function app, choose the **Thumbnail** function, and then select **Add Event Grid subscription**.

A screenshot of the Azure Function App configuration page for 'myfuncapp0130'. The left sidebar shows a tree view with 'myfuncapp0130' expanded, 'Functions (Read Only)' selected, and 'Thumbnail' highlighted with a red box. The main area shows the 'function.json' file content:

```
1  "generatedBy": "Microsoft.NET.Sdk.Functions-1.0.22",
2  "configurationSource": "attributes",
3  "bindings": [
4    {
5      "type": "eventGridTrigger",
6      "name": "eventGridEvent"
7    }
8  ],
9  "disabled": false,
10 "scriptFile": "../bin/ImageFunctions.dll",
11 "entryPoint": "ImageFunctions.Thumbnail.Run"
12 ]
```

At the top right, there are 'Save', 'Run', and 'Add Event Grid subscription' buttons, with 'Add Event Grid subscription' highlighted with a red box.

3. Use the event subscription settings as specified in the table.

Home > Resource groups > spegridrg2 > myfuncapp0130 - Thumbnail > Create Event Subscription

Create Event Subscription

Event Grid

Basic Filters Additional Features

EVENT SUBSCRIPTION DETAILS

Name	imageresizersub	
Event Schema	Event Grid Schema	

TOPIC DETAILS

Pick a topic resource for which events should be generated and pushed. [Learn more](#)

Topic Type	Storage account
Topic Resource	mystorage0130 (change)

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

<input type="checkbox"/> Subscribe to all event types	
Defined Event Types	Blob Created

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type	Web Hook
Endpoint	https://myfuncapp0130.azurewebsites.net/runtime/webhooks/EventGrid...

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.
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.
Subscriber type	autogenerated	Pre-defined as Web Hook.

SETTING	SUGGESTED VALUE	DESCRIPTION
Subscriber endpoint	autogenerated	Use the endpoint URL that is generated for you.

4. Switch to the **Filter** tab, and do the following actions:

- Select **Enable subject filtering** option.
- 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 **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
- Node.js V2 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.

The screenshot shows a web browser window titled "Home Page - ImageResizer". The main content area is titled "ImageResizer". Below it, a section titled "Upload photos" contains a dashed blue rectangular area with the text "Drop files here or click to upload.". Below this, another section titled "Generated Thumbnails" displays a single thumbnail image of a person sitting at a desk with a laptop. At the bottom of the page, a note states: "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."

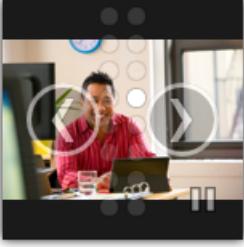
Home Page - ImageResizer

ImageResizer

Upload photos

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.

Create a function on Linux using a custom image

7/29/2019 • 11 minutes to read • [Edit Online](#)

Azure Functions lets you host your functions on Linux in your own custom container. You can also [host on a default Azure App Service container](#). This functionality requires [the Functions 2.x runtime](#).

In this tutorial, you learn how to deploy your functions to Azure as a custom Docker image. This pattern is useful when you need to customize the built-in container image. You may want to use a custom image when your functions need a specific language version or require a specific dependency or configuration that isn't provided within the built-in image. Supported base images for Azure Functions are found in the [Azure Functions base images repo](#). [Python support](#) is in preview at this time.

This tutorial walks you through how to use Azure Functions Core Tools to create a function in a custom Linux image. You publish this image to a function app in Azure, which was created using the Azure CLI. Later, you update your function to connect to Azure Queue storage. You also enable.

In this tutorial, you learn how to:

- Create a function app and Dockerfile using Core Tools.
- Build a custom image using Docker.
- Publish a custom image to a container registry.
- Create an Azure Storage account.
- Create a Premium hosting plan.
- Deploy a function app from Docker Hub.
- Add application settings to the function app.
- Enable continuous deployment.
- Add Application Insights monitoring.

The following steps are supported on a Mac, Windows, or Linux computer.

Prerequisites

Before running this sample, you must have the following:

- Install [Azure Core Tools version 2.x](#).
- Install the [Azure CLI](#). This article requires the Azure CLI version 2.0 or later. Run `az --version` to find the version you have.
You can also use the [Azure Cloud Shell](#).
- An active Azure subscription.

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

Create the local function app project

Run the following command from the command line to create a function app project in the `MyFunctionProj` folder of the current local directory.

```
func init MyFunctionProj --docker
```

When you include the `--docker` option, a dockerfile is generated for the project. This file is used to create a custom container in which to run the project. The base image used depends on the worker runtime language chosen.

When prompted, choose a worker runtime from the following languages:

- `dotnet` : creates a .NET Core class library project (.csproj).
- `node` : creates a JavaScript project.
- `python` : creates a Python project.

NOTE

Python for Azure Functions is currently in preview. To receive important updates, subscribe to the [Azure App Service announcements](#) repository on GitHub.

When the command executes, you see something like the following output:

```
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing Dockerfile
```

Use the following command to navigate to the new `MyFunctionProj` project folder.

```
cd MyFunctionProj
```

Create a function

The following command creates an HTTP-triggered function named `MyHttpTrigger`.

```
func new --name MyHttpTrigger --template "HttpTrigger"
```

When the command executes, you see something like the following output:

```
The function "MyHttpTrigger" was created successfully from the "HttpTrigger" template.
```

Run the function locally

The following command starts the function app. The app runs using the same Azure Functions runtime that is in Azure.

```
func host start --build
```

The `--build` option is required to compile C# projects. You don't need this option for a JavaScript project.

When the Functions host starts, it writes something like the following output, which has been truncated for readability:

```
%%%%%
%%%%%
@  %%%%%%  @
@@ %%%%  @@%
@@@ %%%%%%%  @@@
@@  %%%%  @@%
@@  %%  @@%
@@  %  @@%
%
%
```

...

```
Content root path: C:\functions\MyFunctionProj
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
```

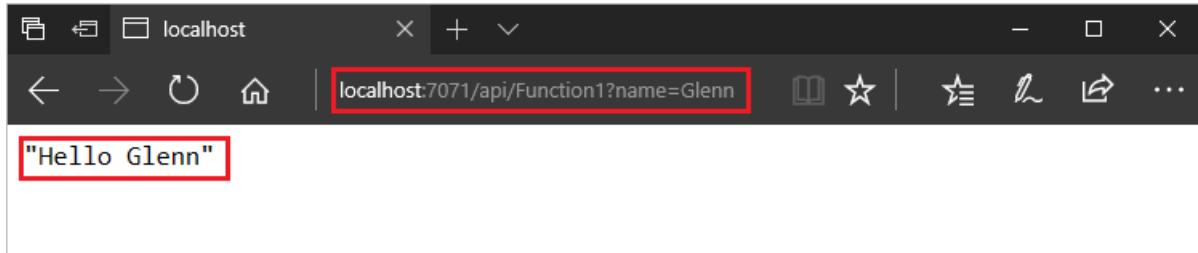
...

Http Functions:

```
HttpTrigger: http://localhost:7071/api/MyHttpTrigger
```

```
[8/27/2018 10:38:27 PM] Host started (29486ms)
[8/27/2018 10:38:27 PM] Job host started
```

Copy the URL of your `HttpTrigger` function from the runtime output and paste it into your browser's address bar. Append the query string `?name=<yourusername>` to this URL and execute the request. The following shows the response in the browser to the GET request returned by the local function:



Now that you have run your function locally, you can create the function app and other required resources in Azure.

Build the image from the Docker file

Take a look at the *Dockerfile* in the root folder of the project. This file describes the environment that is required to run the function app on Linux. The following example is a Dockerfile that creates a container that runs a function app on the JavaScript (Node.js) worker runtime:

```
FROM mcr.microsoft.com/azure-functions/node:2.0
ENV AzureWebJobsScriptRoot=/home/site/wwwroot
COPY . /home/site/wwwroot
```

NOTE

The complete list of supported base images for Azure Functions can be found in the [Azure Functions base image page](#).

Run the `build` command

In the root folder, run the `docker build` command, and provide a name, `mydockerimage`, 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>/mydockerimage:v1.0.0 .
```

When the command executes, you see something like the following output, which in this case is for a JavaScript worker runtime:

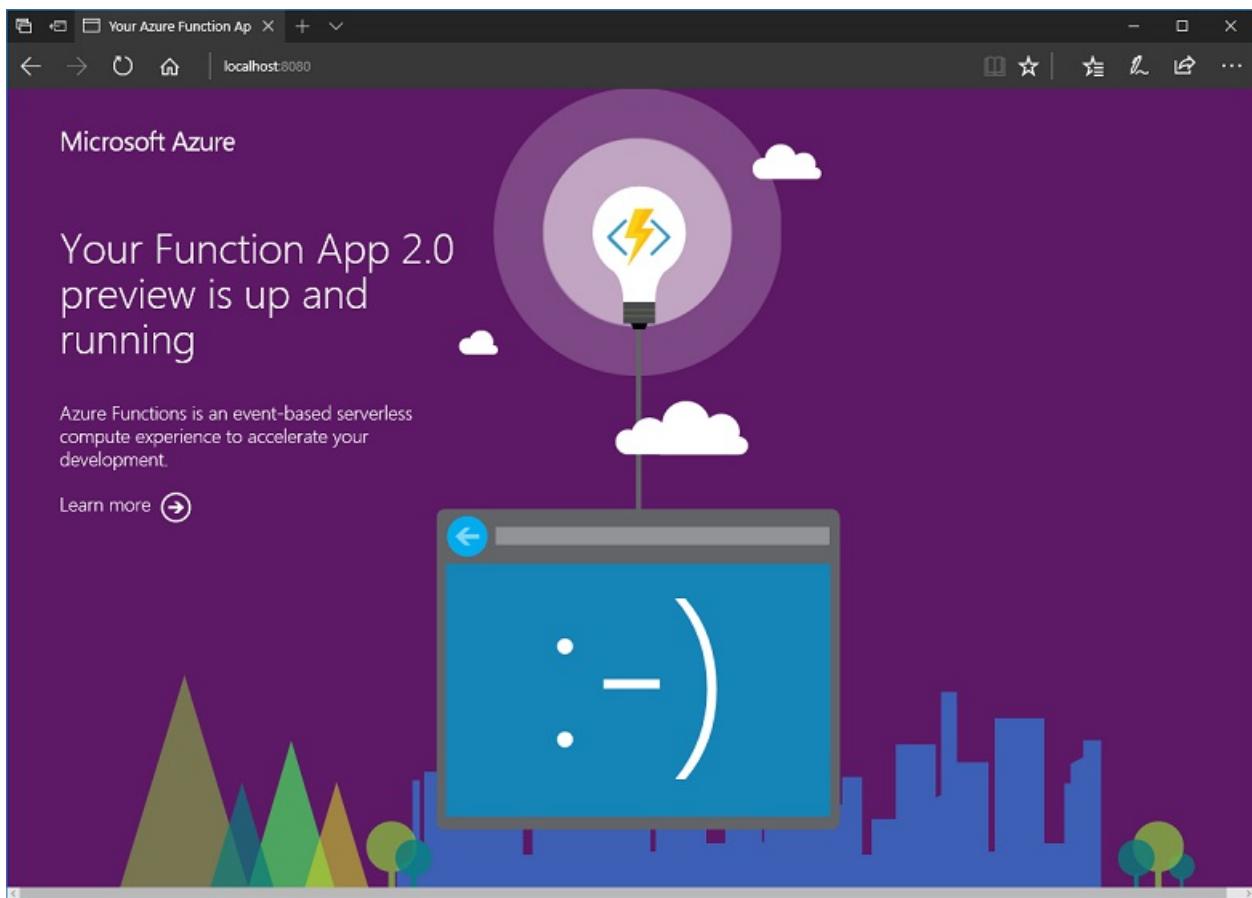
```
Sending build context to Docker daemon 17.41kB
Step 1/3 : FROM mcr.microsoft.com/azure-functions/node:2.0
2.0: Pulling from azure-functions/node
802b00ed6f79: Pull complete
44580ea7a636: Pull complete
73eebe8d57f9: Pull complete
3d82a67477c2: Pull complete
8bd51cd50290: Pull complete
7bd755353966: Pull complete
Digest: sha256:480e969821e9bef7c61dda353f63298f2c4b109e13032df5518e92540ea1d08
Status: Downloaded newer image for mcr.microsoft.com/azure-functions/node:2.0
--> 7c71671b838f
Step 2/3 : ENV AzureWebJobsScriptRoot=/home/site/wwwroot
--> Running in ed1e5809f0b7
Removing intermediate container ed1e5809f0b7
--> 39d9c341368a
Step 3/3 : COPY . /home/site/wwwroot
--> 5e196215935a
Successfully built 5e196215935a
Successfully tagged <docker-id>/mydockerimage:v1.0.0
```

Test the image locally

Verify that the image you built works by running the Docker image in a local container. Issue the `docker run` command and pass the name and tag of the image to it. Be sure to specify the port using the `-p` argument.

```
docker run -p 8080:80 -it <docker-ID>/mydockerimage:v1.0.0
```

With the custom image running in a local Docker container, verify the function app and container are functioning correctly by browsing to <http://localhost:8080>.



You can optionally test your function again, this time in the local container using the following URL:

```
http://localhost:8080/api/myhttpttrigger?name=<yourname>
```

After you have verified the function app in the container, stop the execution. Now, you can push the custom image to your Docker Hub account.

Push the custom image to Docker Hub

A registry is an application that hosts images and provides services image and container services. To share your image, you must push it to a registry. Docker Hub is a registry for Docker images that allows you to host your own repositories, either public or private.

Before you can push an image, you must sign in to Docker Hub using the [docker login](#) command. Replace `<docker-id>` with your account name and type in your password into the console at the prompt. For other Docker Hub password options, see the [docker login command documentation](#).

```
docker login --username <docker-id>
```

A "login succeeded" message confirms that you are logged in. After you have signed in, you push the image to Docker Hub by using the [docker push](#) command.

```
docker push <docker-id>/mydockerimage:v1.0.0
```

Verify that the push succeeded by examining the command's output.

```
The push refers to a repository [docker.io/<docker-id>/mydockerimage:v1.0.0]
24d81eb139bf: Pushed
fd9e998161c9: Mounted from <docker-id>/mydockerimage
e7796c35add2: Mounted from <docker-id>/mydockerimage
ae9a05b85848: Mounted from <docker-id>/mydockerimage
45c86e20670d: Mounted from <docker-id>/mydockerimage
v1.0.0: digest: sha256:be080d80770df71234eb893fbe4d... size: 1796
```

Now, you can use this image as the deployment source for a new function app in Azure.

Create a resource group

Create a resource group with the [az group create](#). An Azure resource group is a logical container into which Azure resources like function apps, databases, and storage accounts are deployed and managed.

The following example creates a resource group named `myResourceGroup`.

If you are not using Cloud Shell, sign in first using `az login`.

```
az group create --name myResourceGroup --location westeurope
```

You generally create your resource group and the resources in a [region](#) near you.

Create an Azure Storage account

Functions uses a general-purpose account in Azure Storage to maintain state and other information about your functions. Create a general-purpose storage account in the resource group you created by using the [az storage account create](#) command.

In the following command, substitute a globally unique storage account name where you see the `<storage_name>` placeholder. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

```
az storage account create --name <storage_name> --location westeurope --resource-group myResourceGroup --sku Standard_LRS
```

Create a Premium plan

Linux hosting for custom Functions containers supported on [Dedicated \(App Service\) plans](#) and [Premium plans](#). This tutorial uses a Premium plan, which can scale as needed. To learn more about hosting, see [Azure Functions hosting plans comparison](#).

The following example creates a Premium plan named `myPremiumPlan` in the **Elastic Premium 1** pricing tier (`--sku EP1`), in the West US region (`-location WestUS`), and in a Linux container (`--is-linux`).

```
az functionapp plan create --resource-group myResourceGroup --name myPremiumPlan \
--location WestUS --number-of-workers 1 --sku EP1 --is-linux
```

Create and deploy the custom image

The function app manages the execution of your functions in your hosting plan. Create a function app from a Docker Hub image by using the [az functionapp create](#) command.

In the following command, substitute a unique function app name where you see the `<app_name>` placeholder and

the storage account name for `<storage_name>`. The `<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. As before, `<docker-id>` is your Docker account name.

```
az functionapp create --name <app_name> --storage-account <storage_name> --resource-group myResourceGroup \
--plan myPremiumPlan --deployment-container-image-name <docker-id>/mydockerimage:v1.0.0
```

The `deployment-container-image-name` parameter indicates the image hosted on Docker Hub to use to create the function app. Use the [az functionapp config container show](#) command to view information about the image used for deployment. Use the [az functionapp config container set](#) command to deploy from a different image.

Configure the function app

The function needs the connection string to connect to the default storage account. When you are publishing your custom image to a private container account, you should instead set these application settings as environment variables in the Dockerfile using the [ENV instruction](#), or something similar.

In this case, `<storage_name>` is the name of the storage account you created. Get the connection string with the [az storage account show-connection-string](#) command. Add these application settings in the function app with the [az functionapp config appsettings set](#) command.

```
storageConnectionString=$(az storage account show-connection-string \
--resource-group myResourceGroup --name <storage_name> \
--query connectionString --output tsv)

az functionapp config appsettings set --name <app_name> \
--resource-group myResourceGroup \
--settings AzureWebJobsDashboard=$storageConnectionString \
AzureWebJobsStorage=$storageConnectionString
```

NOTE

If your container is private, you would have to set the following application settings as well

- `DOCKER_REGISTRY_SERVER_USERNAME`
- `DOCKER_REGISTRY_SERVER_PASSWORD`

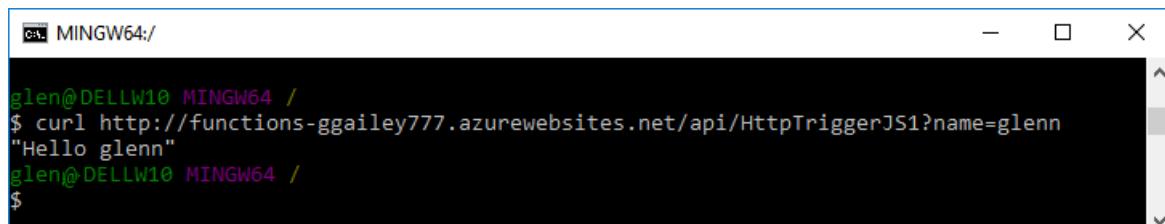
You will have to stop and then start your function app for these values to be picked up

You can now test your functions running on Linux in Azure.

Test the function in Azure

Use cURL to test the deployed function. Using the URL that you copied from the previous step, 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>
```



```
glen@DELLW10 MINGW64 /$ curl http://functions-ggailey777.azurewebsites.net/api/HttpTriggerJS1?name=glen
"Hello glenn"
glen@DELLW10 MINGW64 /$
```

You can also paste the copied URL in to the address of your web browser. Again, append the query string `&name=<yourusername>` to the URL before you execute the request.



Enable continuous deployment

One of the benefits of using containers is being able to automatically deploy updates when containers are updated in the registry. Enable continuous deployment with the [az functionapp deployment container config](#) command.

```
az functionapp deployment container config --enable-cd \
--query CI_CD_URL --output tsv \
--name <app_name> --resource-group myResourceGroup
```

This command returns the deployment webhook URL after continuous deployment is enabled. You can also use the [az functionapp deployment container show-cd-url](#) command to return this URL.

Copy the deployment URL and browse to your DockerHub repo, choose the **Webhooks** tab, type a **Webhook name** for the webhook, paste your URL in **Webhook URL**, and then choose the plus sign (+).

A screenshot of the DockerHub website showing the settings for a repository named "mydocker / mydockerimage". The top navigation bar includes "Explore", "Repositories", "Organizations", and "Get Help". Below the navigation, there are tabs for "General", "Tags", "Builds", "Timeline", "Collaborators", "Webhooks" (which is highlighted with a red box), and "Settings". The "Webhooks" section contains a heading "Webhooks" and a descriptive text about what webhooks are. It includes a link to "Learn More" and a "New Webhook" button. A "Continuous deployment" webhook is listed with its URL: "https://\$mylinuxfunction:ysoxbKdub9SpSf2cgHfcq7K". To the right of the URL is a red-bordered "+" button. Below this, a section titled "Current Webhooks" shows a message: "This repository does not have any webhooks.".

With the webhook set, any updates to the linked image in DockerHub result in the function app downloading and installing the latest image.

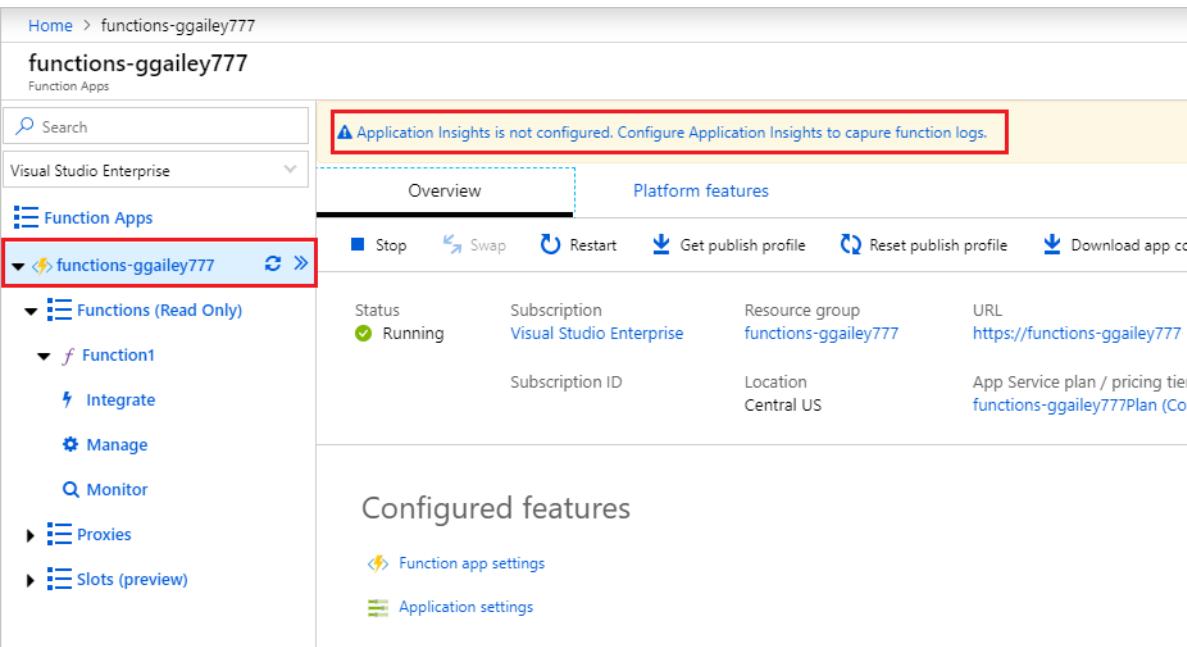
Enable Application Insights

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 by using the Azure CLI, the integration in your function app in Azure isn't done.

To enable Application Insights for your function app:

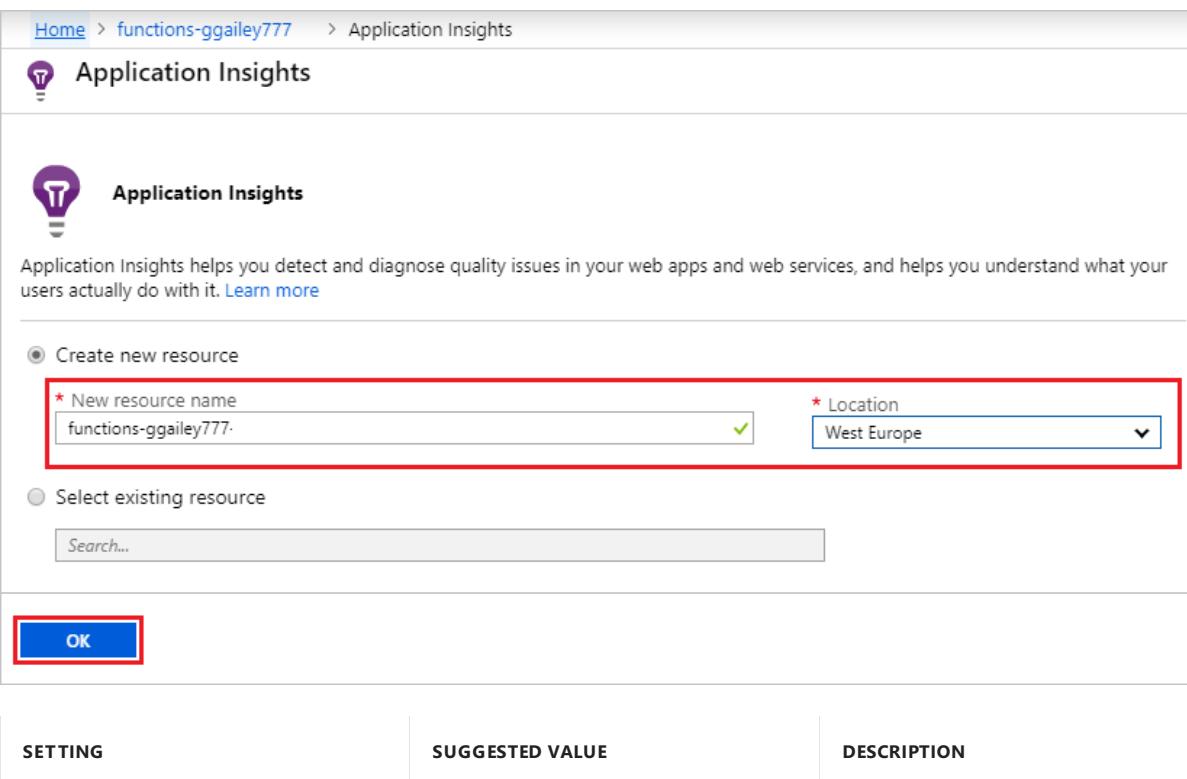
Functions makes it easy to add Application Insights integration to a function app from the [Azure portal](#).

1. In the [portal](#), select **All services > Function Apps**, select your function app, and then select the **Application Insights** banner at the top of the window



The screenshot shows the Azure portal interface for a function app named "functions-ggailey777". The left sidebar lists various settings like Functions (Read Only), Function1, Integrate, Manage, Monitor, Proxies, and Slots (preview). The main area has tabs for Overview and Platform features. The Overview tab displays basic information: Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggailey777), and URL (https://functions-ggailey777). Below this, there's a section for Configured features with links to Function app settings and Application settings. A prominent red box highlights the "Application Insights" banner at the top right, which contains the message "Application Insights is not configured. Configure Application Insights to capture function logs."

2. Create an Application Insights resource by using the settings specified in the table below the image.



The screenshot shows the "Application Insights" creation page. It starts with a brief introduction about Application Insights. There are two options: "Create new resource" (selected) and "Select existing resource". Under "Create new resource", fields for "New resource name" (containing "functions-ggailey777-") and "Location" (set to "West Europe") are filled out. A large red box highlights this entire section. At the bottom, there's an "OK" button, which is also highlighted with a red box. Below the form, there's a table with columns for Setting, Suggested Value, and Description.

SETTING	SUGGESTED VALUE	DESCRIPTION

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

3. Select **OK**. 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.
4. Back in your function app, select **Application settings**, and then scroll down to **Application settings**. If you see a setting named `APPINSIGHTS_INSTRUMENTATIONKEY`, Application Insights integration is enabled for your function app running in Azure.

To learn more, see [Monitor Azure Functions](#).

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

In this tutorial, you learned how to:

- Create a function app and Dockerfile using Core Tools.
- Build a custom image using Docker.
- Publish a custom image to a container registry.
- Create an Azure Storage account.
- Create a Linux Premium plan.
- Deploy a function app from Docker Hub.
- Add application settings to the function app.
- Enable continuous deployment.
- Add Application Insights monitoring.

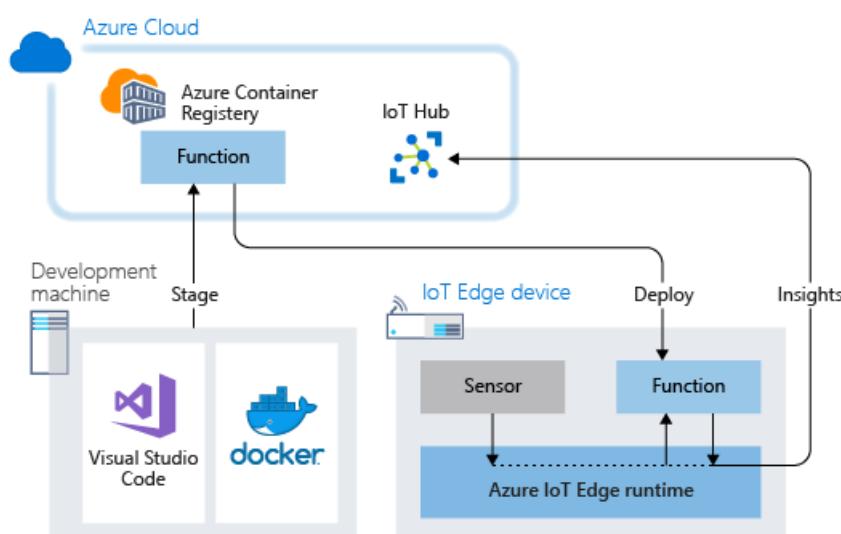
[Learn more about options for deploying functions to Azure](#)

Tutorial: Deploy Azure functions as IoT Edge modules

6/25/2019 • 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.



NOTE

Azure Function modules on Azure IoT Edge are in [public preview](#).

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

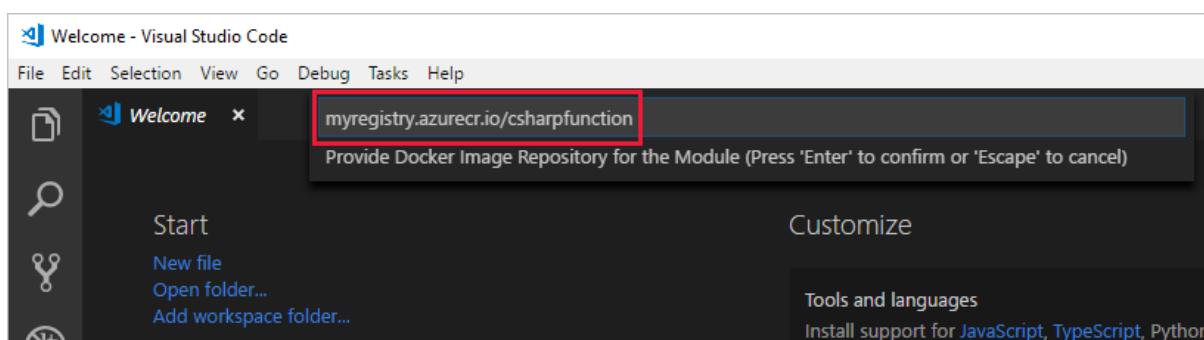
The Azure IoT Tools for Visual Studio Code that you installed in the prerequisites provides management capabilities as well as some code templates. In this section, you use Visual Studio Code to create an IoT Edge solution that contains an Azure function.

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.

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.

                    var filteredMessage = new Message(messageBytes);
                    // Copy the properties of the original message into the new Message object.
                    foreach (KeyValuePair<string, string> prop in messageReceived.Properties)
                    {
                        filteredMessage.Properties.Add(prop.Key, prop.Value);
                    }
                    await output.AddAsync(filteredMessage);
                }
            }
        }
    }
}
```

```

        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 your IoT Edge solution

In the previous section, you created an IoT Edge solution and added code to the **CSharpFunction** to filter out messages where the reported machine temperature is less than the acceptable threshold. Now you need to build the solution as a container image and push it to your container registry.

In this section, you provide the credentials for your container registry for the second time (the first was in the **.env** file of your IoT Edge solution) by signing in locally from your development machine so that Visual Studio Code can push images to your registry.

- Open the VS Code integrated terminal by selecting **View > Terminal**.
- Sign in to your container registry by entering the following command in the integrated terminal. Use the username and login server that you copied from your Azure container registry earlier.

```
docker login -u <ACR username> <ACR login server>
```

When you're prompted for the password, paste the password (it won't be visible in the terminal window) for your container registry and press **Enter**.

```
Password: <paste in the ACR password and press enter>
Login Succeeded
```

- In the VS Code explorer, right-click the deployment.template.json file and select **Build and Push IoT Edge solution**.

When you tell Visual Studio Code to build your solution, it first takes the information in the deployment template and generates a deployment.json file in a new folder named **config**. Then it runs two commands in the integrated

terminal: `docker build` and `docker push`. These two commands build your code, containerize the functions, and then push the code to the container registry that you specified when you initialized the solution.

View your container image

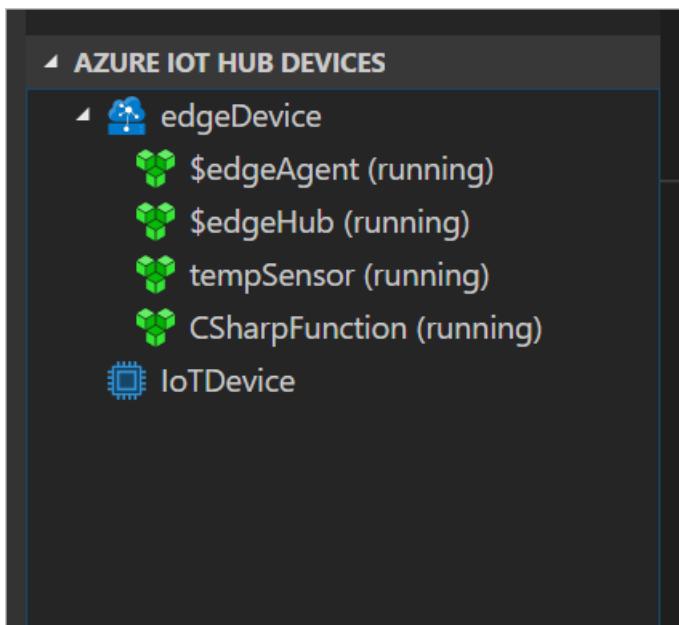
Visual Studio Code outputs a success message when your container image is pushed to your container registry. If you want to confirm the successful operation for yourself, you can view the image in the registry.

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 VS Code explorer, expand the **Azure IoT Hub Devices** section.
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.json** file, and then choose **Select Edge Deployment Manifest**.
4. Refresh the **Azure IoT Hub Devices** section. You should see the new **CSharpFunction** running along with the **TempSensor** 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 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](#)

Azure CLI Samples

6/4/2019 • 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	Creates a function app in a Consumption plan.
Create a function app in an 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.

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.

CONFIGURE APP	DESCRIPTION
Map a custom domain to a function app	Define a custom domain for your functions.
Bind an SSL certificate to a function app	Upload SSL certificates for functions in a custom domain.

Azure Functions runtime versions overview

3/15/2019 • 8 minutes to read • [Edit Online](#)

There are two major versions of the Azure Functions runtime: 1.x and 2.x. The current version where new feature work and improvements are being made is 2.x, though both are supported for production scenarios. The following details some of the differences between the two, how you can create each version, and upgrade from 1.x to 2.x.

NOTE

This article refers to the cloud service Azure Functions. For information about the preview product that lets you run Azure Functions on-premises, see the [Azure Functions Runtime Overview](#).

Cross-platform development

The version 2.x runtime runs on .NET Core 2, which enables it to run on all platforms supported by .NET Core, including macOS and Linux. Running on .NET Core enables cross-platform development and hosting scenarios.

By comparison, the version 1.x runtime only supports development and hosting in the Azure portal or on Windows computers.

Languages

The version 2.x runtime uses a new language extensibility model. In version 2.x, 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.

Azure Functions 1.x experimental languages won't be updated to use the new model, so they aren't supported in 2.x. The following table indicates which programming languages are currently supported in each runtime version.

LANGUAGE	1.X	2.X
C#	GA (.NET Framework 4.7)	GA (.NET Core 2.2)
JavaScript	GA (Node 6)	GA (Node 8 & 10)
F#	GA (.NET Framework 4.7)	GA (.NET Core 2.2)
Java	N/A	GA (Java 8)
PowerShell	Experimental	Preview (PowerShell Core 6)
Python	Experimental	Preview (Python 3.6)
TypeScript	Experimental	GA (supported through transpiling to JavaScript)
Bash	Experimental	N/A

Language	1.x	2.x
Batch (.cmd, .bat)	Experimental	N/A
PHP	Experimental	N/A

For information about planned changes to language support, see [Azure roadmap](#).

For more information, see [Supported languages](#).

Run on version 1.x

By default, function apps created in the Azure portal are set to version 2.x. When possible, you should use this runtime version, where new feature investments are being made. If you need to, you can still run a function app on the version 1.x runtime. You can only change the runtime version after you create your function app but before you add any functions. To learn how to pin the runtime version to 1.x, see [View and update the current runtime version](#).

Migrating from 1.x to 2.x

You may choose to migrate an existing app written to use the version 1.x runtime to instead use version 2.x. 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 2. 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 for version 2.x and port your existing version 1.x function code to the new app.

Changes in triggers and bindings

Version 2.x requires you to 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 have also been 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

A few features that have also been removed, updated, or replaced in the new version. This section details the changes you see in version 2.x 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 a locally developed application

You may have existing function app projects that you developed locally using the version 1.x runtime. To upgrade to version 2.x, you should create a local function app project against version 2.x and port your existing code into the new app. You could manually update the existing project and code, a sort of "in-place" upgrade. However, there are a number of other improvements between version 1.x and version 2.x that you may still need to make. For example, in C# the debugging object was changed from `TraceWriter` to `ILogger`. By creating a new version 2.x project, you start off with updated functions based on the latest version 2.x templates.

Visual Studio runtime versions

In Visual Studio, you select the runtime version when you create a project. Azure Functions tools for Visual Studio supports both 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>netcoreapp2.2</TargetFramework>
<AzureFunctionsVersion>v2</AzureFunctionsVersion>
```

When you debug or publish your project, the correct version of the runtime is used.

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 2.x, install version 2.x of the Core Tools. Version 1.x development requires version 1.x of the Core Tools. 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.

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. A value of `~2` targets the version 2.x runtime and `~1` targets the version 1.x runtime. Don't arbitrarily change this setting, because other app setting changes and code changes in your functions are likely required. To learn about the recommended way to migrate your function

app to a different runtime version, see [How to target Azure Functions runtime versions](#).

Bindings

The version 2.x 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 two major versions of the Azure Functions runtime:

TYPE	1.X	2.X ¹	TRIGGER	INPUT	OUTPUT
Blob storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓	✓	✓	✓
Event Grid	✓	✓	✓		
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
Microsoft Graph Excel tables		✓		✓	✓
Microsoft Graph OneDrive files		✓		✓	✓
Microsoft Graph Outlook email		✓			✓
Microsoft Graph events		✓	✓	✓	✓
Microsoft Graph Auth tokens		✓		✓	
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ In 2.x, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

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 in both runtime versions:

PLAN	RUNTIME VERSION	DEFAULT	MAXIMUM
Consumption	1.x	5	10
Consumption	2.x	5	10
App Service	1.x	Unlimited	Unlimited
App Service	2.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 scale and hosting

7/24/2019 • 11 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 hosting plans available for Azure Functions: [Consumption plan](#), [Premium plan](#), and [App Service plan](#).

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 VNET 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 down 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 down to zero. (Requires that [Always on](#) is enabled.)

NOTE

You can switch between Consumption and Premium plans by changing the plan property of the function app resource.

Hosting plan support

Feature support falls into the following two categories:

- *Generally available (GA)*: fully supported and approved for production use.
- *Preview*: not yet fully supported and approved for production use.

The following table indicates the current level of support for the three hosting plans, when running on either Windows or Linux:

	CONSUMPTION PLAN	PREMIUM PLAN	DEDICATED PLAN
Windows	GA	preview	GA
Linux	preview	preview	GA

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. Billing 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.

Premium plan (preview)

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
- Premium instance sizes (one core, two core, and four core instances)
- More predictable pricing
- High-density app allocation for plans with multiple function apps

Information on how you can configure these options can be found in the [Azure Functions premium plan document](#).

Instead of billing per execution and memory consumed, billing for the Premium plan is based on the number of core seconds, execution time, and memory used across needed and reserved instances. At least one instance must be warm at all times. This means that there is a fixed monthly cost per active plan, regardless of the number of executions.

Consider the Azure Functions premium plan in the following situations:

- Your function apps run continuously, or nearly continuously.
- 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 VNET/VPN connectivity.

When running JavaScript functions on a Premium plan, you should choose an instance that has fewer vCPUs. For more information, see the [Choose single-core Premium plans](#).

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](#).

With an App Service plan, you can manually scale out by adding more VM instances. You can also enable autoscale. 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](#).

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 in both runtime versions:

PLAN	RUNTIME VERSION	DEFAULT	MAXIMUM
Consumption	1.x	5	10
Consumption	2.x	5	10
App Service	1.x	Unlimited	Unlimited
App Service	2.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 / pricing tier** in the **Overview** tab for the function app in the [Azure portal](#). For App Service plans, the pricing tier is also indicated.

The screenshot shows the Azure portal interface for a function app named "myfunctionapp". The "Overview" tab is selected. Key details shown include:

- Status: Running
- Subscription: Visual Studio Enterprise
- Resource group: myresourcegroup
- URL: https://myfunctionapp.azurewebsites.net
- App Service plan / pricing tier: CentralUSPlan (Consumption)

The sidebar on the left lists other function apps and features:

- Function Apps: myfunctionapp (selected), HTTPTriggerCSharp
- Functions: HTTPTriggerCSharp
- Proxies
- Slots (preview)

The "Configured features" section lists:

- Function app settings
- Application settings
- Deployment options configured with VSTS/ARM
- Application Insights

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 Functions rely on Azure Storage for operations such as managing triggers and logging function executions, but some storage accounts do not 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.

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

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.

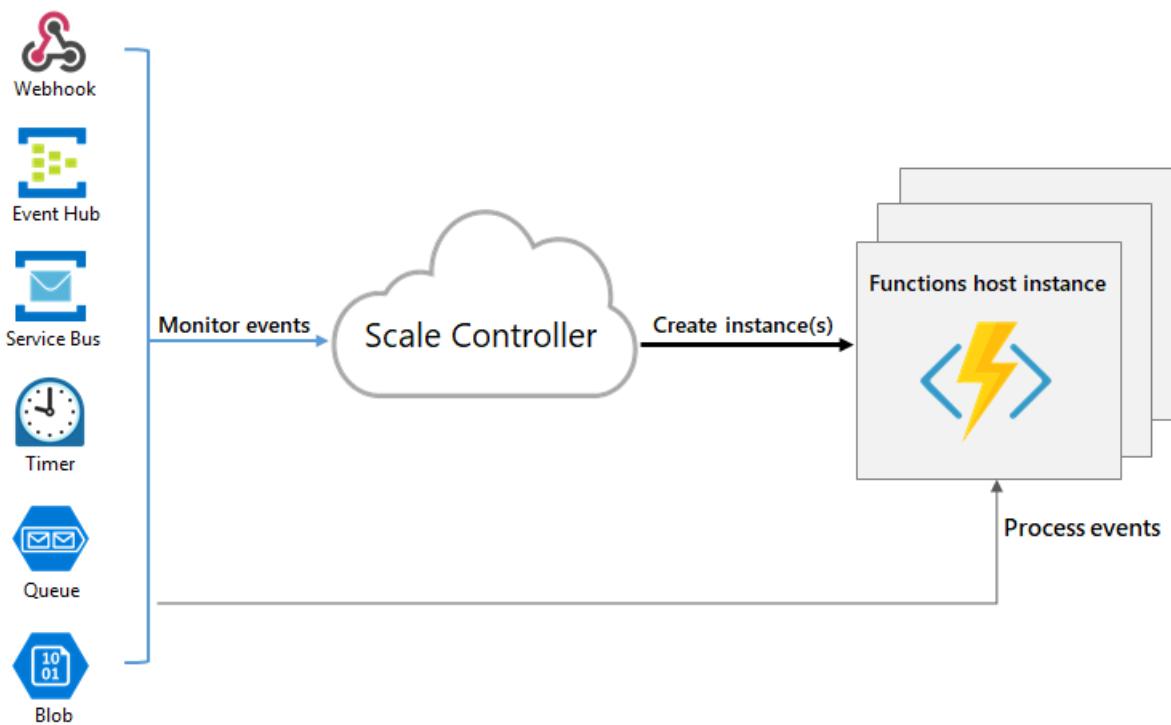
NOTE

When you're using a blob trigger on a Consumption plan, there can be up to a 10-minute delay in processing new blobs. This delay occurs when a function app has gone idle. After the function app is running, blobs are processed immediately. To avoid this cold-start delay, use the Premium plan, or use the [Event Grid trigger](#). For more information, see [the blob trigger binding reference article](#).

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 down to zero when no functions are running within a function app.



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 up 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.
- For HTTP triggers, new instances will only be allocated at most once every 1 second.
- For non-HTTP triggers, new instances will only be allocated at most once every 30 seconds.

Different triggers may also have different scaling limits as well as documented below:

- [Event Hub](#)

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](#).

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](#).

Service limits

The following table indicates the limits that apply to function apps when running in the various hosting plans:

RESOURCE	CONSUMPTION PLAN	PREMIUM PLAN	APP SERVICE PLAN ¹
Scale out	Event driven	Event driven	Manual/autoscale
Max instances	200	20	10-20
Default time out duration (min)	5	30	30 ²
Max time out duration (min)	10	unbounded	unbounded ³
Max outbound connections (per instance)	600 active (1200 total)	unbounded	unbounded
Max request size (MB) ⁴	100	100	100
Max query string length ⁴	4096	4096	4096
Max request URL length ⁴	8192	8192	8192
ACU per instance	100	210-840	100-840
Max memory (GB per instance)	1.5	3.5-14	1.75-14
Function apps per plan	100	100	unbounded ⁵
App Service plans	100 per region	100 per resource group	100 per resource group
Storage ⁶	1 GB	250 GB	50-1000 GB
Custom domains per app	500 ⁷	500	500
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

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

² By default, the timeout for the Functions 1.x runtime in an App Service plan is unbounded.

³ Requires the App Service plan be set to [Always On](#). Pay at standard [rates](#).

⁴ These limits are [set in the host](#).

⁵ 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.

⁶ 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.

⁷ When your function app is hosted in a [Consumption plan](#), only the CNAME option is supported. For function apps in a [Premium plan](#) or an [App Service plan](#), you can map a custom domain using either a CNAME or an A record.

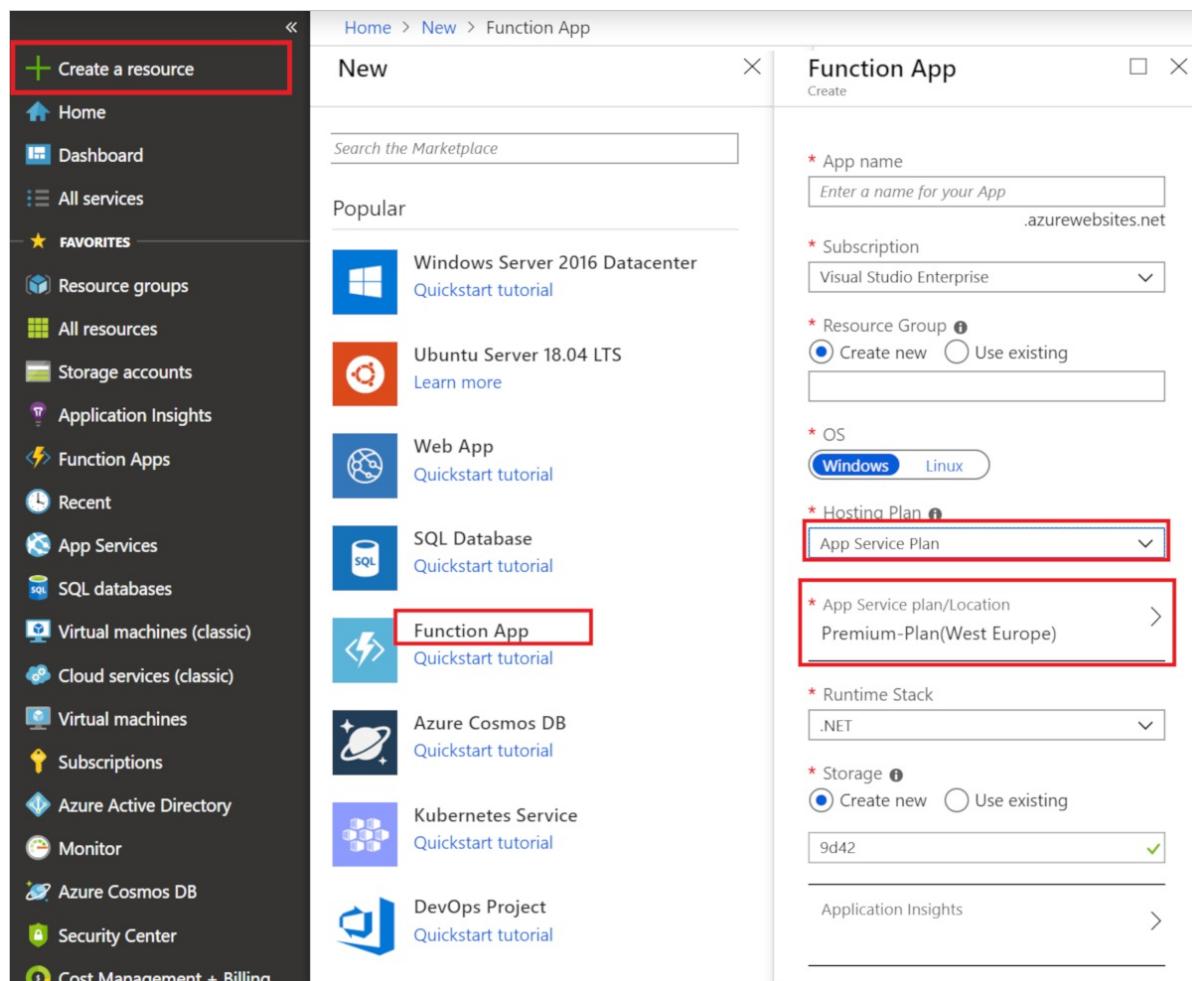
Azure Functions Premium plan (preview)

7/24/2019 • 6 minutes to read • [Edit Online](#)

The Azure Functions 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. Go to the [Azure portal](#).
2. Select **+ Create a resource** on the left hand side, then choose **Function app**.
3. For **Hosting plan**, choose **App Service plan**, then select **App Service plan/Location**.



4. Select **Create new**, type an **App Service plan** name, choose a **Location** in a **region** near you or near other services your functions access, and then select **Pricing tier**.

The image shows two overlapping windows. On the left is the 'App Service plan' dialog with a heading 'Select a plan for the web app'. It contains a grey info box with text about App Service plans and a red-bordered button labeled 'Create new'. On the right is the 'New App Service Plan' dialog with a heading 'Create a plan for the web app'. It has three dropdown fields: 'App Service plan' (set to 'Premium-Plan'), 'Location' (set to 'West Europe'), and 'Pricing tier' (set to 'EP1 Elastic Premium'). A red box highlights the 'Pricing tier' field. At the bottom is a blue 'OK' button.

- Choose the **EP1** (elastic Premium) plan, then select **Apply**.

The image shows the 'Recommended pricing tiers' section of the Azure portal. It displays six pricing tiers: S1, P1V2, P2V2, P3V2, EP1, and EP2. Each tier is represented by a colored box with its details. The 'EP1' tier is highlighted with a red double border. Below the tiers is a link 'See additional options' and a red-bordered 'Apply' button.

Tier	Total ACU	Memory	Compute Equivalent	Cost
S1	100 total ACU	1.75 GB memory	A-Series compute equivalent	44.64 USD/Month (Estimated)
P1V2	210 total ACU	3.5 GB memory	Dv2-Series compute equivalent	148.80 USD/Month (Estimated)
P2V2	420 total ACU	7 GB memory	Dv2-Series compute equivalent	297.60 USD/Month (Estimated)
P3V2	840 total ACU	14 GB memory	Dv2-Series compute equivalent	595.20 USD/Month (Estimated)
EP1	210 total ACU	3.5 GB memory	Dv2-Series compute equivalent	148.80 USD/Month (Estimated)
EP2	420 total ACU	7 GB memory	Dv2-Series compute equivalent	297.60 USD/Month (Estimated)
EP3	840 total ACU	14 GB memory	Dv2-Series compute equivalent	595.20 USD/Month (Estimated)

- Select **OK** to create the plan, then use the remaining function app settings as specified in the table below the image.

Function App X

Create

* App name
VNET-Function ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise ▼

* Resource Group i
 Create new Use existing
myResourceGroup ✓

* OS
Windows [Linux](#)

* Hosting Plan i
App Service Plan ▼

* App Service plan/Location
Premium-Plan(West Europe) >

* Runtime Stack
.NET ▼

* Storage i
 Create new Use existing
vnetfunction1bed7 ✓

Application Insights >
VNET-Function

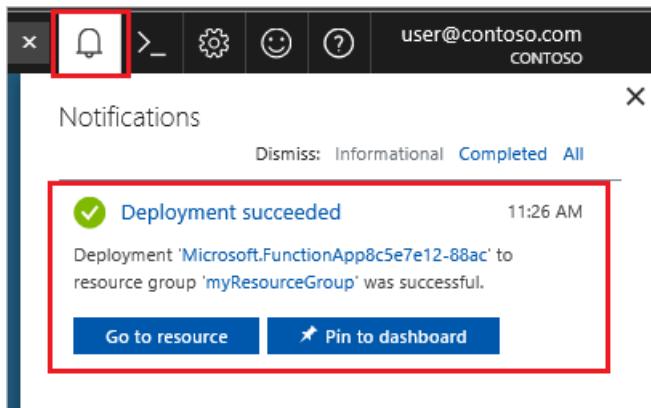
Create [Automation options](#)

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Your subscription	The subscription under which this new function app is created.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app. You can also use the suggested value.
OS	Preferred OS	Both Linux and Windows are supported on the Premium plan.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions. Only languages supported on your chosen OS are displayed.
Storage	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 .
Application Insights	Default	Creates an Application Insights resource of the same App name 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 where you want to store your data.

7. After your settings are validated, select **Create**.

8. Select the Notification 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.

Features

The following features are available to function apps deployed to a Premium plan.

Pre-warmed instances

If no events and executions occur today in the Consumption plan, your app may scale down 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 pre-warmed on a specified number of instances, up to your minimum plan size. Pre-warmed instances also let you pre-scale an app before high load. As the app scales out, it first scales into the pre-warmed instances. Additional instances continue to buffer out and warm immediately in preparation for the next scale operation. By having a buffer of pre-warmed instances, you can effectively avoid cold start latencies. Pre-warmed instances is a feature of the Premium plan, and you need to keep at least one instance running and available at all times the plan is active.

You can configure the number of pre-warmed 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, pre-warmed instances is specific to that app, but the minimum and maximum instances apply to your entire plan.



Elastic Scale out

This allows you to control the bounds that your Premium plan can scale within. [Learn more](#)

Plan Scale out

Minimum Instances

Maximum Burst

App Scale out

Pre-Warmed Instances

You can also configure pre-warmed instances for an app with 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
```

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. Though the maximum number of instances may vary during the preview, 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. To learn more about how scaling works, see [Function scale and hosting](#).

Unbounded 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 this unbounded for Premium plan apps.

In preview, your duration is not guaranteed past 12 minutes and will have the best chance of running beyond 30 minutes if your app is not scaled beyond its minimum worker count.

Plan and SKU settings

When you create the plan, you configure two settings: the minimum number of instances (or plan size) and the maximum burst limit. The minimum instances for a Premium plan is 1, and the maximum burst during the preview is 20. Minimum instances are reserved and always running.

IMPORTANT

You are charged for each instance allocated in the minimum instance count regardless if functions are executing or not.

If your app requires instances beyond your plan size, 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 rented to you. We will make a best effort at scaling your app out to its defined maximum limit, whereas the minimum plan instances are guaranteed for your app.

You can configure the plan size and maximums in the Azure portal by selected 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 resource update -g <resource_group> -n <premium_plan_name> --set properties.maximumElasticWorkerCount=<desired_max_burst> --resource-type Microsoft.Web/serverfarms
```

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 consumed per second. 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

Regions

Below are the currently supported regions for the public preview for each OS.

REGION	WINDOWS	LINUX
Australia East	✓	
Australia Southeast	✓	✓
Canada Central	✓	
Central US	✓	
East Asia	✓	
East US		✓
East US 2	✓	
France Central	✓	
Japan East		✓
Japan West	✓	
Korea Central	✓	
North Central US	✓	
North Europe	✓	✓
South Central US	✓	
South India	✓	
Southeast Asia	✓	✓
UK West	✓	
West Europe	✓	✓
West India	✓	
West US	✓	✓

Known Issues

You can track the status of known issues of the public preview on [GitHub](#).

Next steps

[Understand Azure Functions scale and hosting options](#)

Deployment technologies in Azure Functions

7/24/2019 • 7 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 exhaustive list of those technologies, describes which technologies are available for which flavors of Functions, explains what happens when you use each method, and provides recommendations for the best method to use in various scenarios. The various tools that support deploying to Azure Functions are tuned to the right technology based on their context.

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 (PREVIEW)	WINDOWS DEDICATED	LINUX CONSUMPTION (PREVIEW)	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

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.

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 (Preview).

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 in [Run From Package](#) mode.

How to use it: Deploy by using your favorite client tool: [VS Code](#), [Visual Studio](#), or the [Azure CLI](#). 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 in [Run From Package](#) mode. To set Run From Package mode, 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 Functions running on Windows and Linux in the Premium or Dedicated plan.

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 in an App Service plan.

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:

	WINDOWS CONSUMPTION	WINDOWS PREMIUM (PREVIEW)	WINDOWS DEDICATED	LINUX CONSUMPTION (PREVIEW)	LINUX PREMIUM (PREVIEW)	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 slots

When you deploy your function app to Azure, you can deploy to a separate deployment slot instead of deploying directly to production. For more information about deployment slots, see [Azure App Service slots](#).

Deployment slots levels of support

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	Preview
Windows Premium (preview)	Preview
Windows Dedicated	General availability
Linux Consumption	Unsupported
Linux Premium (preview)	Preview
Linux Dedicated	General availability

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 triggers and bindings concepts

7/1/2019 • 3 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.

Supported bindings

This table shows the bindings that are supported in the two major versions of the Azure Functions runtime:

TYPE	1.X	2.X ¹	TRIGGER	INPUT	OUTPUT
Blob storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓	✓	✓	✓
Event Grid	✓	✓	✓		
Event Hubs	✓	✓	✓		✓

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
HTTP & webhooks	✓	✓	✓		✓
Microsoft Graph Excel tables		✓		✓	✓
Microsoft Graph OneDrive files		✓		✓	✓
Microsoft Graph Outlook email		✓			✓
Microsoft Graph events		✓	✓	✓	✓
Microsoft Graph Auth tokens		✓		✓	
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ In 2.x, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

For information about which bindings are in preview or are approved for production use, see [Supported languages](#).

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

7/1/2019 • 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

7/8/2019 • 3 minutes to read • [Edit Online](#)

In Azure Functions version 2.x, [bindings](#) are available as separate packages from the functions runtime. While .NET functions access bindings through NuGet packages, extension bundles allow other functions access to all bindings through a configuration setting.

Consider the following items related to binding extensions:

- Binding extensions aren't explicitly registered in Functions 1.x except when [creating a C# class library using Visual Studio](#).
- HTTP and timer triggers are supported by default and don't require an extension.

The following table indicates when and how you register bindings.

DEVELOPMENT ENVIRONMENT	REGISTRATION IN FUNCTIONS 1.X	REGISTRATION IN FUNCTIONS 2.X
Azure portal	Automatic	Automatic
Non-.NET languages or local Azure Core Tools development	Automatic	Use Azure Functions Core Tools and extension bundles
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

Extension bundles for local development

Extension bundles is a local development technology for the version 2.x runtime that lets you add a compatible set of Functions binding extensions to your function app project. These extension packages are then included in the deployment package when you deploy to Azure. Bundles makes all bindings published by Microsoft available through a setting in the `host.json` file. Extension packages defined in a bundle are compatible with each other, which helps you avoid conflicts between packages. When developing locally, make sure you are using the latest version of [Azure Functions Core Tools](#).

Use extension bundles for all local development using Azure Functions Core Tools or Visual Studio Code.

If you don't use extension bundles, you must install the .NET Core 2.x SDK on your local computer before you install any binding extensions. Bundles removes this requirement for local development.

To use extension bundles, update the `host.json` file to include the following entry for `extensionBundle`:

```
{  
    "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 .

Bundle versions increment as packages in the bundle change. Major version changes occur when packages in the bundle increment by a major version, which usually coincides with a change in the major version of the Functions runtime.

The current set of extensions installed by the default bundle are enumerated in this [extensions.json file](#).

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 do not need to use [extension bundles](#). This approach is specific for class libraries built in Visual Studio.

C# class library with Visual Studio Code

NOTE

We recommend using [extension bundles](#) to have Functions automatically install a compatible set of binding extension packages.

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 provided in the reference article for your desired binding. 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

7/1/2019 • 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 that 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 such as the extension. 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}"
}
```

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}"  
}
```

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

7/1/2019 • 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 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](#).

See the language-specific example showing use of the return value:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)
- [Python](#)

C# example

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);
}
```

C# script example

Here's the output binding in the `function.json` file:

```
{
  "name": "$return",
  "type": "blob",
  "direction": "out",
  "path": "output-container/{id}"
}
```

Here's the C# script code, followed by an async example:

```
public static string Run(WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return json;
}
```

```
public static Task<string> Run(WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return Task.FromResult(json);
}
```

F# example

Here's the output binding in the *function.json* file:

```
{
  "name": "$return",
  "type": "blob",
  "direction": "out",
  "path": "output-container/{id}"
}
```

Here's the F# code:

```
let Run(input: WorkItem, log: ILogger) =
    let json = String.Format("{{ \"id\": \"{0}\" }}", input.Id)
    log.LogInformation(sprintf "F# script processed queue message '%s'" json)
    json
```

JavaScript example

Here's the output binding in the *function.json* file:

```
{
  "name": "$return",
  "type": "blob",
  "direction": "out",
  "path": "output-container/{id}"
}
```

In JavaScript, the return value goes in the second parameter for `context.done`:

```
module.exports = function (context, input) {
  var json = JSON.stringify(input);
  context.log('Node.js script processed queue message', json);
  context.done(null, json);
}
```

Python example

Here's the output binding in the *function.json* file:

```
{
  "name": "$return",
  "type": "blob",
  "direction": "out",
  "path": "output-container/{id}"
}
```

Here's the Python code:

```
def main(input: azure.functions.InputStream) -> str:
    return json.dumps({
        'name': input.name,
        'length': input.length,
        'content': input.read().decode('utf-8')
    })
```

Next steps

[Handle Azure Functions binding errors](#)

Handle Azure Functions binding errors

7/1/2019 • 2 minutes to read • [Edit Online](#)

Azure Functions [triggers and bindings](#) communicate with various Azure services. When integrating with these services, you may have errors raised that originate from the APIs of the underlying Azure services. Errors can also occur when you try to communicate with other services from your function code by using REST or client libraries. To avoid loss of data and ensure good behavior of your functions, it is important to handle errors from either source.

The following triggers have built-in retry support:

- [Azure Blob storage](#)
- [Azure Queue storage](#)
- [Azure Service Bus \(queue/topic\)](#)

By default, these triggers are retried up to five times. After the fifth retry, these triggers write a message to a special [poison queue](#).

For the other Functions triggers, there is no built-in retry when errors occur during function execution. To prevent loss of trigger information should an error occur in your function, we recommend that you use try-catch blocks in your function code to catch any errors. When an error occurs, write the information passed into the function by the trigger to a special "poison" message queue. This approach is the same one used by the [Blob storage trigger](#).

In this way, you can capture trigger events that could be lost due to errors and retry them at a later time using another function to process messages from the poison queue using the stored information.

For links to all relevant error topics for the various services supported by Functions, see the [Binding error codes](#) section of the [Azure Functions error handling](#) overview topic.

Supported languages in Azure Functions

5/6/2019 • 2 minutes to read • [Edit Online](#)

This article explains the levels of support offered for languages that you can use with Azure Functions.

Levels of support

There are three 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.
- **Experimental** - Not supported and might be abandoned in the future; no guarantee of eventual preview or GA status.

Languages in runtime 1.x and 2.x

Two 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
C#	GA (.NET Framework 4.7)	GA (.NET Core 2.2)
JavaScript	GA (Node 6)	GA (Node 8 & 10)
F#	GA (.NET Framework 4.7)	GA (.NET Core 2.2)
Java	N/A	GA (Java 8)
PowerShell	Experimental	Preview (PowerShell Core 6)
Python	Experimental	Preview (Python 3.6)
TypeScript	Experimental	GA (supported through transpiling to JavaScript)
Bash	Experimental	N/A
Batch (.cmd, .bat)	Experimental	N/A
PHP	Experimental	N/A

For information about planned changes to language support, see [Azure roadmap](#).

Experimental languages

The experimental languages in version 1.x don't scale well and don't support all bindings.

Don't use experimental features for anything that you rely on, as there is no official support for them. Support cases should not be opened for problems with experimental languages.

The version 2.x runtime doesn't support experimental languages. Support for new languages is added only when

the language can be supported in production.

Language extensibility

The 2.x 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 use one of the GA or preview languages in Azure Functions, see the following resources:

[C#](#)

[F#](#)

[JavaScript](#)

[Java](#)

[Python](#)

Azure Functions C# developer reference

7/16/2019 • 11 minutes to read • [Edit Online](#)

This article is an introduction to developing Azure Functions by using C# in .NET class libraries.

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](#).

This article assumes that you've already read the following articles:

- [Azure Functions developers guide](#)
- [Azure Functions Visual Studio 2019 Tools](#)

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:

Functions 1.x

```
<PropertyGroup>
  <TargetFramework>net461</TargetFramework>
</PropertyGroup>
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Sdk.Functions" Version="1.0.8" />
</ItemGroup>
```

Functions 2.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=)
```

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}");
    }
}
```

Logging

To log output to your streaming logs in C#, include an argument of type `ILogger`. We recommend that you name it `log`, as 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}");
    }
}
```

Avoid using `Console.WriteLine` in Azure Functions. For more information, see [Write logs in C# functions](#) in the [Monitor Azure Functions](#) article.

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);
        }
    }
}
```

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.WriteLineAsync("Hello World!!!");
        }
    }
}

```

Triggers and bindings

This table shows the bindings that are supported in the two major versions of the Azure Functions runtime:

TYPE	1.X	2.X ¹	TRIGGER	INPUT	OUTPUT
Blob storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓	✓	✓	✓

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
Event Grid	✓	✓	✓		
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
Microsoft Graph Excel tables		✓		✓	✓
Microsoft Graph OneDrive files		✓		✓	✓
Microsoft Graph Outlook email		✓			✓
Microsoft Graph events		✓	✓	✓	✓
Microsoft Graph Auth tokens		✓		✓	
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ In 2.x, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

Next steps

[Learn more about triggers and bindings](#)

[Learn more about best practices for Azure Functions](#)

Azure Functions C# script (.csx) developer reference

7/16/2019 • 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](#) 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;
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 [Write logs in C# functions](#) in the [Monitor Azure Functions](#) article.

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;
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 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	<code>Microsoft.Azure.WebJobs.DocumentDBAttribute</code> #r "Microsoft.Azure.WebJobs.Extensions.CosmosDB"	
Event Hubs	<code>Microsoft.Azure.WebJobs.ServiceBus.EventAttribute</code> #r "Microsoft.Azure.Jobs.ServiceBus" ,	<code>Microsoft.Azure.WebJobs.ServiceBusAccountAttribute</code>
Mobile Apps	<code>Microsoft.Azure.WebJobs.MobileTableAttribute</code> #r "Microsoft.Azure.WebJobs.Extensions.MobileApp"	
Notification Hubs	<code>Microsoft.Azure.WebJobs.NotificationHubAttribute</code> #r "Microsoft.Azure.WebJobs.Extensions.NotificationHubs"	
Service Bus	<code>Microsoft.Azure.WebJobs.ServiceBusAttribute</code> #r "Microsoft.Azure.WebJobs.ServiceBus" ,	<code>Microsoft.Azure.WebJobs.ServiceBusAccountAttribute</code>
Storage queue	<code>Microsoft.Azure.WebJobs.QueueAttribute</code> ,	<code>Microsoft.Azure.WebJobs.StorageAccountAttribute</code>
Storage blob	<code>Microsoft.Azure.WebJobs.BlobAttribute</code> ,	<code>Microsoft.Azure.WebJobs.StorageAccountAttribute</code>
Storage table	<code>Microsoft.Azure.WebJobs.TableAttribute</code> ,	<code>Microsoft.Azure.WebJobs.StorageAccountAttribute</code>
Twilio	<code>Microsoft.Azure.WebJobs.TwilioSmsAttribute</code> #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

6/26/2019 • 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 the version 2.x 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](#) 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 simple name (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\facavalfunc\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

6/28/2019 • 21 minutes to read • [Edit Online](#)

This guide contains information about the intricacies of writing Azure Functions with JavaScript.

A JavaScript 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.

This article assumes that you have already read the [Azure Functions developer reference](#). Complete the Functions quickstart to create your first function, using [Visual Studio Code](#) or [in the portal](#).

This article also supports [TypeScript app development](#).

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 to let you communicate with the runtime. The context object can be used for reading and setting data from bindings, writing logs, and using the `context.done` callback when your exported function is synchronous.

The `context` object is always the first parameter to a function. It should be included because it has important methods such as `context.done` and `context.log`. You can name the object whatever you would like (for example, `ctx` or `c`).

```
// You must include a context, but other arguments are optional
module.exports = function(ctx) {
    // function logic goes here :)
    ctx.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. On `context.log`, additional logging methods are available that let you write function logs at other trace levels:

METHOD	DESCRIPTION
<code>error(message)</code>	Writes to error level logging, or lower.
<code>warn(message)</code>	Writes to warning level logging, or lower.
<code>info(message)</code>	Writes to info level logging, or lower.
<code>verbose(message)</code>	Writes to verbose level logging.

The following example writes a log at the warning trace level:

```
context.log.warn("Something has happened.");
```

You can [configure the trace-level threshold for logging](#) in the host.json file. For more information on writing logs, see [writing trace outputs](#) below.

Read [monitoring Azure Functions](#) to learn more about viewing and querying function logs.

Writing trace output to the console

In Functions, you use the `context.log` methods to write trace output to the console. In Functions v2.x, trace outputs using `console.log` are captured at the Function App level. This means that outputs from `console.log` are not tied to a specific function invocation and aren't displayed in a specific function's logs. They do, however, propagate to Application Insights. In Functions v1.x, you cannot use `console.log` to write to the console.

When you call `context.log()`, your message is written to the console at the default trace level, which is the `info` trace level. The following code writes to the console at the `info` trace level:

```
context.log({hello: 'world'});
```

This code is equivalent to the code above:

```
context.log.info({hello: 'world'});
```

This code writes to the console at the `error` level:

```
context.log.error("An error has occurred.");
```

Because `error` is the highest trace level, this trace is written to the output at all trace levels as long as logging is enabled.

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));
```

Configure the trace level for console logging

Functions 1.x lets you define the threshold trace level for writing to the console, which makes it easy to control the way traces are written to the console from your function. To set the threshold for all traces written to the console, use the `tracing.consoleLevel` property in the host.json file. This setting applies to all functions in your function app. The following example sets the trace threshold to enable verbose logging:

```
{
  "tracing": {
    "consoleLevel": "verbose"
  }
}
```

Values of **consoleLevel** correspond to the names of the `context.log` methods. To disable all trace logging to the console, set **consoleLevel** to *off*. For more information, see [host.json reference](#).

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
<code>body</code>	An object that contains the body of the response.
<code>headers</code>	An object that contains the response headers.
<code>isRaw</code>	Indicates that formatting is skipped for the response.
<code>status</code>	The HTTP status code of the response.

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);
```

Node version

The following table shows the Node.js version used by each major version of the Functions runtime:

FUNCTIONS VERSION	NODE.JS VERSION
1.x	6.11.2 (locked by the runtime)
2.x	<i>Active LTS and Maintenance LTS</i> Node.js versions (8.11.1 and 10.14.1 recommended). Set the version by using the WEBSITE_NODE_DEFAULT_VERSION app setting .

You can see the current version that the runtime is using by checking the above app setting or by printing `process.version` from any function.

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

In Functions, [app settings](#), such as service connection strings, are exposed as environment variables during execution. You can access these settings 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) {
    var timeStamp = new Date().toISOString();

    context.log('Node.js timer trigger function ran!', timeStamp);
    context.log("AzureWebJobsStorage: " + process.env["AzureWebJobsStorage"]);
    context.log("WEBSITE_SITE_NAME: " + process.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.

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.

NOTE

Experimental support for TypeScript exists version 1.x of the Functions runtime. The experimental version transpiles TypeScript files into JavaScript files when the function is invoked. In version 2.x, this experimental support has been superseded by the tool-driven method that does transpilation before the host is initialized and during the deployment process.

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

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.

To run your function app code locally using Core Tools, use the `npm start` command, instead of `func host start`. The `npm start` command is equivalent to the following commands:

- `npm run build`
- `func extensions install`
- `tsc`
- `func start`

Before you use the `func azure functionapp publish` command to deploy to Azure, you must first run the `npm run build:production` command. This command creates a production-ready build of JavaScript files from the TypeScript source files that can be deployed using `func azure functionapp publish`.

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) {
    try {
        const 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

6/17/2019 • 9 minutes to read • [Edit Online](#)

The Azure Functions runtime supports [Java SE 8 LTS \(zulu8.31.0.2-jre8.0.181-win_x64\)](#). This guide contains information about the intricacies of writing Azure Functions with Java.

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.

This article assumes that you have already read the [Azure Functions developer reference](#). You should also complete the Functions quickstart to create your first function, by using [Visual Studio Code](#) or [Maven](#).

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.

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 = {"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"
    }
  ]
}
```

JDK runtime availability and support

For local development of Java function apps, download and use the [Azul Zulu Enterprise for Azure Java 8](#) JDKs from [Azul Systems](#). Azure Functions uses the Azul Java 8 JDK runtime when you deploy your function apps 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.

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:

```
az functionapp config appsettings set --name <APP_NAME> \
--resource-group <RESOURCE_GROUP> \
--settings "JAVA_OPTS=-Djava.awt.headless=true"
```

This example enables headless mode. Replace `<APP_NAME>` with the name of your function app, and `<RESOURCE_GROUP>` with the resource group.

WARNING

In the [Consumption plan](#), you must add the `WEBSITE_USE_PLACEHOLDER` setting with a value of `0`. This setting does increase the cold start times for Java functions.

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 = { "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
<code>HttpResponseMessage</code>	HTTP Output Binding	Returns status other than 200

Metadata

Few triggers send [trigger metadata](#) along with input data. You can use annotation `@BindingName` to bind to trigger metadata.

```

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 = { "get", "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.

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 = {"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:

```
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:

```
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")`.

For example, you can add [AppSetting](#), with the name `testAppSetting` and the value `testAppSettingValue`:

```
public class Function {
    public String echo(@HttpTrigger(name = "req", methods = {"post"}, authLevel =
AuthorizationLevel.ANONYMOUS) String req, ExecutionContext context) {
        context.getLogger().info("testAppSetting "+ System.getenv("testAppSettingValue"));
        return String.format(req);
    }
}
```

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 Azure Functions with 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

7/9/2019 • 17 minutes to read • [Edit Online](#)

This article provides details about how you write Azure Functions using PowerShell.

NOTE

PowerShell for Azure Functions is currently in preview. To receive important updates, subscribe to the [Azure App Service announcements](#) repository on GitHub.

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](#) 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.logLevel.default` property in the [host.json file](#). This setting applies to all functions in your function app.

The following example sets the threshold to enable verbose logging for all functions, but sets the threshold to enable debug logging for a function named `MyFunction`:

```
{  
    "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`
- `HttpResponseContext`

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
----------	-------------	------

PROPERTY	DESCRIPTION	TYPE
Body	An object that contains the body of the request. Body 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
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 `HttpContext`, 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 `HttpContext` 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 executes when the function app starts. Function apps start when first deployed and after being idled ([cold start](#)).

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 version

The following table shows the PowerShell version used by each major version of the Functions runtime:

FUNCTIONS VERSION	POWERSHELL VERSION
1.x	Windows PowerShell 5.1 (locked by the runtime)
2.x	PowerShell Core 6

You can see the current version by printing `$PSVersionTable` from any function.

Dependency management

PowerShell functions support managing Azure modules by the service. By modifying the host.json and setting the managedDependency enabled property to true, the requirements.psd1 file will be processed. The latest Azure modules will be automatically downloaded and made available to the function.

host.json

```
{
  "managedDependency": {
    "enabled": true
  }
}
```

requirements.psd1

```
@{
  Az = '1.*'
}
```

Leveraging your own custom modules or modules from the [PowerShell Gallery](#) is a little different than how you would do it normally.

When you install the module on your local machine, it goes in one of the globally available folders in your `$env:PSModulePath`. Since your function runs in Azure, you won't have access to the modules installed on your machine. This requires that the `$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 lives inside the PowerShell language worker.

Function app-level `Modules` folder

To use custom modules or PowerShell modules from the PowerShell Gallery, 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.

To take advantage of this feature, create a `Modules` folder in the root of your function app. Save the modules you want to use in your functions in this location.

```
mkdir ./Modules
Save-Module MyGalleryModule -Path ./Modules
```

Use `Save-Module` to save all of the modules your functions use, or copy your own custom modules to the `Modules` folder. With a `Modules` folder, your function app should have the following folder structure:

```
PSFunctionApp
| - MyFunction
| | - run.ps1
| | - function.json
| - Modules
| | - MyGalleryModule
| | - MyOtherGalleryModule
| | - MyCustomModule.psm1
| - local.settings.json
| - host.json
```

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.

The most recent version of these modules is used by Functions. To use a specific version of these modules, you can put the 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.

You can change this behavior by setting the following environment variable to an integer value:

```
PSWorkerInProcConcurrencyUpperBound
```

You set this environment variable in the [app settings](#) of your Function App.

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. This feature is how the Azure Functions PowerShell runtime works.

There are some drawbacks with this approach.

Concurrency is only as good as the machine it's running on

If your function app is running on an [App Service plan](#) that only supports a single core, then concurrency won't help much. That's because there are no additional cores to help balance the load. In this case, performance can vary when the single core has to context-switch between runspaces.

The [Consumption plan](#) runs using only one core, so you can't leverage concurrency. If you want to take full advantage of concurrency, instead deploy your functions to a function app running on a dedicated App Service plan with sufficient cores.

Azure PowerShell state

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 concurrency back to `1` and try the request again.

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

7/15/2019 • 8 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](#).

NOTE

Python for Azure Functions is currently in preview. To receive important updates, subscribe to the [Azure App Service announcements](#) repository on GitHub.

Programming model

An Azure Function should 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 change the default configuration by 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",
  ...
}
```

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`:

```
{
  "bindings": [
    {
      "name": "req",
      "direction": "in",
      "type": "httpTrigger",
      "authLevel": "anonymous"
    },
    {
      "name": "$return",
      "direction": "out",
      "type": "http"
    }
  ]
}
```

The `__init__.py` file contains the following function code:

```
def main(req):
    user = req.params.get('user')
    return f'Hello, {user}!'
```

Optionally, to leverage the intellisense and auto-complete features provided by your code editor, you can also declare the attribute types and return type in the function using Python type annotations.

```
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.

Folder structure

The folder structure for a Python Functions project looks like the following:

```
FunctionApp
| - MyFirstFunction
| | - __init__.py
| | - function.json
| - MySecondFunction
| | - __init__.py
| | - function.json
| - SharedCode
| | - myFirstHelperFunction.py
| | - mySecondHelperFunction.py
| - host.json
| - requirements.txt
```

There's a shared [host.json](#) file that can be used to configure the function app. Each function has its own code file and binding configuration file (function.json).

Shared code should be kept in a separate folder. To reference modules in the SharedCode folder, you can use the following syntax:

```
from __app__.SharedCode import myFirstHelperFunction
```

When deploying a Function project to your function app in Azure, the entire content of the *FunctionApp* folder should be included in the package, but not the folder itself.

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 `AzureWebJobsStorage` 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>logging.critical(message)</code>	Writes a message with level CRITICAL on the root logger.
<code>logging.error(message)</code>	Writes a message with level ERROR on the root logger.
<code>logging.warning(message)</code>	Writes a message with level WARNING on the root logger.

METHOD	DESCRIPTION
logging.info(<i>message</i>)	Writes a message with level INFO on the root logger.
logging.debug(<i>message</i>)	Writes a message with level DEBUG on the root logger.

Async

We recommend that you write your Azure Function as an asynchronous coroutine using the `async def` statement.

```
# Will be run with asyncio directly

async def main():
    await some_nonblocking_socket_io_op()
```

If the `main()` function is synchronous (no `async` qualifier) we automatically run the function in an `asyncio` thread-pool.

```
# Would be run in an asyncio thread-pool

def main():
    some_blocking_socket_io()
```

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 methods:

`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
```

Python version and package management

Currently, Azure Functions only supports Python 3.6.x (official CPython distribution).

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 dependencies are listed in the `requirements.txt` file, which is located at the root of your project directory. If you're using a package that requires a compiler and does not support the installation of manylinux-compatible wheels from PyPI, publishing to Azure will fail with the following error:

```
There was an error restoring dependencies.ERROR: cannot install <package name - version> dependency:
binary dependencies without wheels are not supported.
The terminal process terminated with exit code: 1
```

To automatically build and configure the required binaries, [install Docker](#) on your local machine and run the following command to publish using the [Azure Functions Core Tools](#) (`func`). Remember to replace `<app name>` with the name of your function app in Azure.

```
func azure functionapp publish <app name> --build-native-deps
```

Underneath the covers, Core Tools will use docker to run the mcr.microsoft.com/azure-functions/python image as a container on your local machine. Using this environment, it'll then build and install the required modules from source distribution, before packaging them up for final deployment to Azure.

To build your dependencies and publish using a continuous delivery (CD) system, [use Azure DevOps Pipelines](#).

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 [Python version and package management](#) section above.

For example, following is a mock test of an HTTP triggered function:

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

```
# myapp/httpfunc.py  
import azure.functions as func  
import logging  
  
def my_function(req: func.HttpRequest) -> func.HttpResponse:  
    logging.info('Python HTTP trigger function processed a request.')  
  
    name = req.params.get('name')  
    if not name:  
        try:  
            req_body = req.get_json()  
        except ValueError:  
            pass  
    else:  
        name = req_body.get('name')  
  
    if name:  
        return func.HttpResponse(f"Hello {name}")  
    else:  
        return func.HttpResponse(  
            "Please pass a name on the query string or in the request body",  
            status_code=400  
        )
```

```

# myapp/test_httpfunc.py
import unittest

import azure.functions as func
from httpfunc import my_function


class TestFunction(unittest.TestCase):
    def test_my_function(self):
        # Construct a mock HTTP request.
        req = func.HttpRequest(
            method='GET',
            body=None,
            url='/api/HttpTrigger',
            params={'name': 'Test'})

        # Call the function.
        resp = my_function(req)

        # Check the output.
        self.assertEqual(
            resp.get_body(),
            b'Hello Test',
        )

```

Here is another example, with a queue triggered function:

```

# myapp/__init__.py
import azure.functions as func


def my_function(msg: func.QueueMessage) -> str:
    return f'msg body: {msg.get_body().decode()}'
```

```

# myapp/test_func.py
import unittest

import azure.functions as func
from . import my_function


class TestFunction(unittest.TestCase):
    def test_my_function(self):
        # Construct a mock Queue message.
        req = func.QueueMessage(
            body=b'test')

        # Call the function.
        resp = my_function(req)

        # Check the output.
        self.assertEqual(
            resp,
            'msg body: test',
        )
```

Known issues and FAQ

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](#)

Azure App Service diagnostics overview

6/16/2019 • 4 minutes to read • [Edit Online](#)

When you're running a web application, you want to be prepared for any issues that may arise, from 500 errors to your users telling you that your site is down. App Service diagnostics is an intelligent and interactive experience to help you troubleshoot your app with no configuration required. When you do run into issues with your app, App Service diagnostics points out what's wrong to guide you to the right information to more easily and quickly troubleshoot and resolve the issue.

Although this experience is most helpful when you're having issues with your app within the last 24 hours, all the diagnostic graphs are always available for you to analyze.

App Service diagnostics works for not only your app on Windows, but also apps on [Linux/containers](#), [App Service Environment](#), and [Azure Functions](#).

Open App Service diagnostics

To access App Service diagnostics, navigate to your App Service web app or App Service Environment in the [Azure portal](#). In the left navigation, click on **Diagnose and solve problems**.

For Azure Functions, navigate to your function app, and in the top navigation, click on **Platform features**, and select **Diagnose and solve problems** from the **Resource management** section.

In the App Service diagnostics homepage, you can choose the category that best describes the issue with your app by using the keywords in each homepage tile. Also, this page is where you can find **Diagnostic Tools** for Windows apps. See [Diagnostic tools \(only for Windows app\)](#).

The screenshot shows the Microsoft Azure portal interface. The left sidebar includes 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (with items like All resources, Resource groups, App Services, Subscriptions, Function App, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, Monitor, Advisor, Security Center, Cost Management + Billing, and Help + support), and a 'Report a bug' button. The main content area is titled 'BuggyBakery' and shows the 'App Service Diagnostics' homepage. It features several tiles:

- Availability and Performance**: Describes downtime or slowness, with keywords: Health Check, Downtime, 5xx Errors, 4xx Errors, CPU, Memory.
- Configuration and Management**: Describes misconfiguration issues, with keywords: Scaling, Swaps, Failed Backups, IPs, Migration.
- SSL and Domains**: Describes certificate and domain management issues, with keywords: 4xx Errors, SSL, Domains, Permissions, Auth, Cert.
- Best Practices**: Describes running in production, with keywords: AutoScale, Traffic Manager, AlwaysOn, ARR Affinity.
- Diagnostic Tools**: Describes deeper investigation, with keywords: Profiler, Memory Dump, DaaS, AutoHeal, Metrics, Security.

The URL in the browser bar is https://ms.portal.azure.com.

Interactive interface

Once you select a homepage category that best aligns with your app's problem, App Service diagnostics' interactive interface, Genie, can guide you through diagnosing and solving problem with your app. You can use the tile shortcuts provided by Genie to view the full diagnostic report of the problem category that you are interested. 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 App Service Diagnostics! My name is Genie and I'm here to help you diagnose and solve problems." Another message box below it says, "Here are some issues related to Availability and Performance that I can help with. Please select the tile that best describes your issue." Six blue tiles are displayed in two rows: Application Logs, Container Issues, CPU Usage, Memory Usage, Port Usage, Process Full List; and Process List, Web App Down, Web App Restarted.

After clicking on these tiles, 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. You can click on any of these topics to investigate the issues further. Also, you can click on **View Full Report** to explore all the topics on a single page.

The screenshot shows the 'Availability and Performance' tab selected. A message box says, "Hello! Welcome to App Service Diagnostics! My name is Genie and I'm here to help you diagnose and solve problems." Another message box below it says, "Here are some issues related to Availability and Performance that I can help with. Please select the tile that best describes your issue." A blue button on the right says, "I am interested in Application Logs". Below, a message box says, "Okay give me a moment while I analyze your app for any issues related to this tile. Once the detectors load, feel free to click to investigate each topic further." Three items are listed in a red-bordered box: 1. Wrong port is exposed: 5000 != 8080 (with a red arrow icon) 2. Verbose application logging is off. (with a red arrow icon) 3. Link to Full Logs (with a red arrow icon).

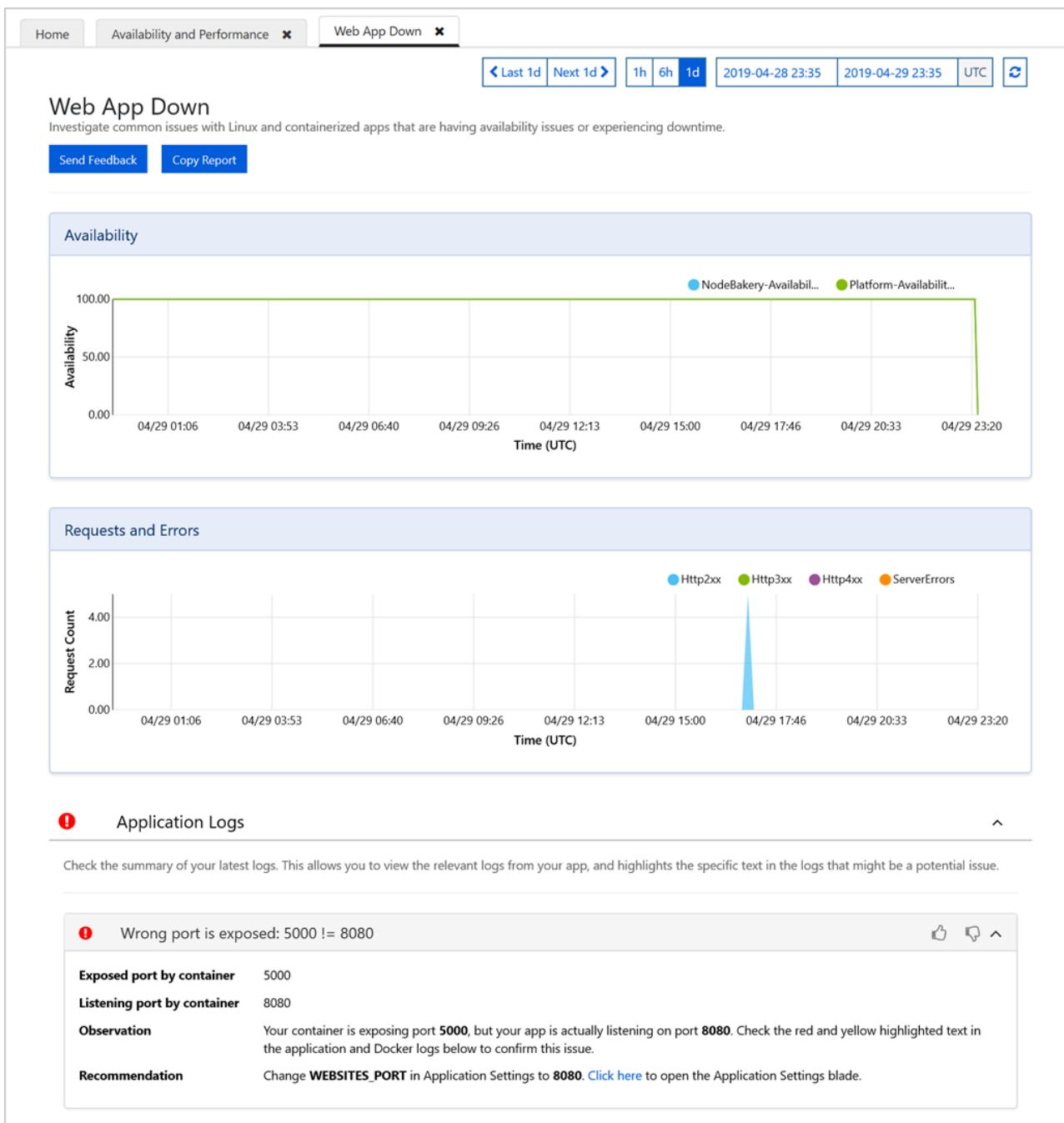
Web App Down

[View Full Report ➤](#)

- ❗ Application Logs ➤
- ❗ Container Issues ➤
- ℹ CPU Usage ➤
- ✓ Memory Usage ➤
- ℹ Port Usage ➤
- ℹ Process List ➤
- ⚠ Web App Restarted ➤

Diagnostic report

After you choose to investigate the issue further by clicking on a topic, you can view more details about the topic often supplemented with graphs and markdowns. Diagnostic report can be a powerful tool for pinpointing the problem with your app.



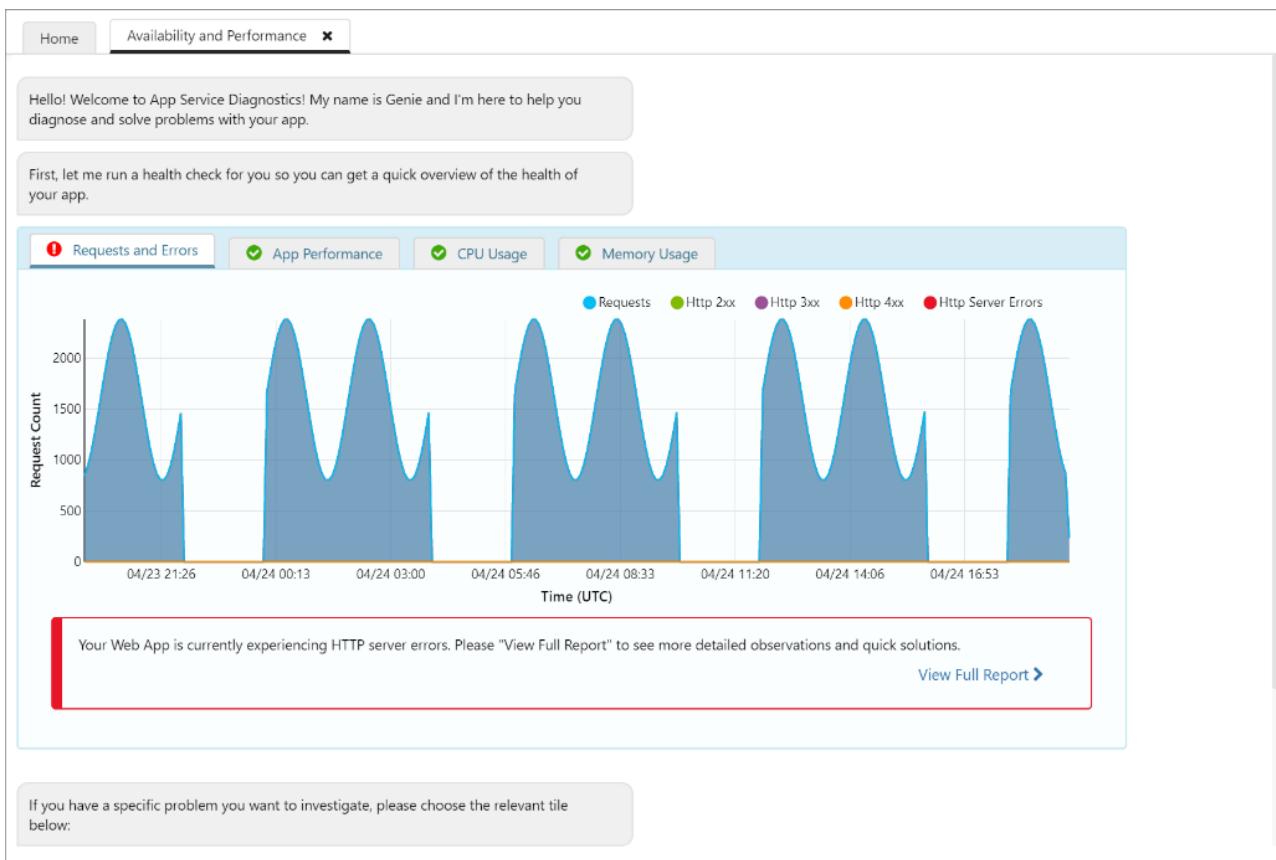
Health checkup

If you don't know what's wrong with your app or don't know where to start troubleshooting your issues, the health checkup is a good place to start. The health checkup analyzes your applications to give you a quick, interactive overview that points out what's healthy and what's wrong, telling you where to look to investigate the issue. Its intelligent and interactive interface provides you with guidance through the troubleshooting process. Health checkup is integrated with the Genie experience for Windows apps and web app down diagnostic report for Linux apps.

Health checkup graphs

There are four different graphs in the health checkup.

- requests and errors:** A graph that shows the number of requests made over the last 24 hours along with HTTP server errors.
- app performance:** A graph that shows response time over the last 24 hours for various percentile groups.
- CPU usage:** A graph that shows the overall percent CPU usage per instance over the last 24 hours.
- memory usage:** A graph that shows the overall percent physical memory usage per instance over the last 24 hours.



Investigate application code issues (only for Windows app)

Because many app issues are related to issues in your application code, App Service diagnostics integrates with [Application Insights](#) to highlight exceptions and dependency issues to correlate with the selected downtime. Application Insights has to be enabled separately.

Application Insights		
Application Exceptions that occurred during this time period		
Message	Exception	Count
Value cannot be null. Parameter name: UPDATE_BREAD_INFO_EXTERNALLY	System.ArgumentNullException at BuildBakery.Controllers.HomeController.CheckUpdateNeeded	4374
View More in App Insights		

To view Application Insights exceptions and dependencies, select the **web app down** or **web app slow** tile shortcuts.

Troubleshooting steps (only for Windows app)

If an issue is detected with a specific problem category within the last 24 hours, you can view the full diagnostic report, and App Service diagnostics may prompt you to view more troubleshooting advice and next steps for a more guided experience.



Troubleshooting and Next Steps

Next steps curated specifically for your app

Scale Out App Service Plan

Mitigation

Remote Profile App

Investigation

Scale out your App Service Plan

Mitigation

Increase the number of the instances in your app service plan. The requests to your app will be spread across all instances.

Current App Service Plan

S1 Standard 1
Tier Instance Count 2
App Count

Your current App Service Plan has only 1 instance. We suggest scaling out to at least two instances to make sure that when we do infrastructure upgrades, your app has the best chance of being highly available.

[Open Scale Out App Service Plan](#)

Why you should scale up

- Your app is consistently using high levels of CPU on every instance.

Diagnostic tools (only for Windows app)

Diagnostics Tools include more advanced diagnostic tools that help you investigate application code issues, slowness, connection strings, and more. and proactive tools that help you mitigate issues with CPU usage, requests, and memory.

Proactive CPU monitoring

Proactive CPU monitoring provides you an easy, proactive way to take an action when your app or child process for your app is consuming high CPU resources. You can set your own CPU threshold rules to temporarily mitigate a high CPU issue until the real cause for the unexpected issue is found.

- 1. Configure

Monitoring Enabled

Off On



CPU Threshold

This is the CPU threshold at which the rule will be triggered

75%



Threshold Seconds

For the rule to trigger, CPU should exceed 75% for this many seconds

30 seconds



Monitor Frequency

This is how frequently the rule will be evaluated

15 seconds



Configure Action

An action that you want to take when the above condition is met

CollectAndKill



Maximum Actions

Maximum number of memory dumps to be collected by this rule

2 dumps



Maximum Duration

Rule will be deactivated after this duration even if no data is collected

168 hours

Rule Configuration

When the site's process or any child processes of the site's process takes **75%** of CPU for more than **30** seconds, **collect a memory dump and kill the process**. Evaluate CPU usage every **15 seconds**. Collect a maximum of **2 memory dumps**. Monitoring will stop automatically after **7 days**.

Save

Proactive auto-healing

Like proactive CPU monitoring, proactive auto-healing offers an easy, proactive approach to mitigating unexpected behavior of your app. You can set your own rules based on request count, slow request, memory limit, and HTTP status code to trigger mitigation actions. This tool can be used to temporarily mitigate an unexpected behavior until the real cause for the issue is found. For more information on proactive auto-healing, visit [Announcing the new auto healing experience in app service diagnostics](#).

[Configure Mitigation Rules](#)
[ProActive Auto-Healing](#)
[View Auto-Healing History](#)

Auto-Healing Enabled
Off
On

1. Define Conditions

Request Duration
 Memory Limit
 Request Count
 Status Codes

2. Configure Actions

Recycle Process
 Log an Event
 Custom Action

Run Diagnostics
Run Any Executable

Memory Dump
Collects memory dumps of the process and the child processes hosting your app and analyzes them for errors

CLR Profiler
Profiles ASP.NET application code to identify exceptions and performance issues

CLR Profiler With Thread Stacks
Profiles ASP.NET application code to identify exceptions and performance issues and dumps stacks to identify deadlocks

JAVA Memory Dump
Collects a binary memory dump using jMap of all java.exe processes running for this web app

JAVA Thread Dump
Collects jStack output of all java.exe processes running for this app and analyzes the same

Change analysis (only for Windows app)

In a fast-paced development environment, sometimes it may be difficult to keep track of all the changes made to your app and let alone pinpoint on a change that caused unhealthy behavior. Change analysis can help you narrow down on the changes made to your app to facilitate trouble-shooting experience. Change analysis is found in **Application Changes** and also embedded in a diagnostic report such as **Application Crashes** so you can use it concurrently with other metrics.

Change analysis has to be enabled before using the feature. For more information on change analysis, visit [Announcing the new change analysis experience in App Service Diagnostics](#).

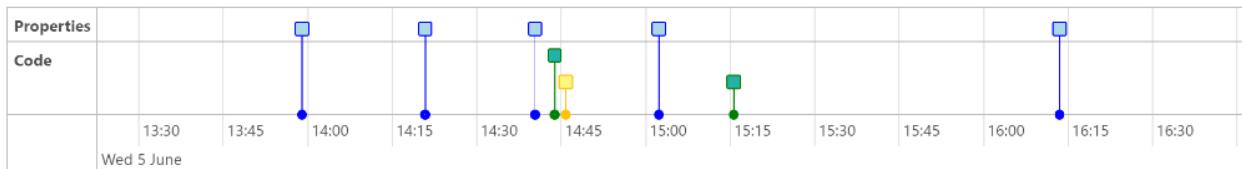
i 12 change groups have been detected

Changes were last scanned on Thu, Jun 6 2019, 7:15:36 am

[Go to Change Analysis Settings](#)

Click the below button to scan your web app and get the latest changes

[Scan changes now](#)



Search table



Normal Important Noise

Level	Time	Name	Description	Initiated By
>	Jun 5 2019, 2:45:49 pm	\Areas\HelpPage\Views\Web.config	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\Views\Web.config	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\Web.config	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\bin\changeanalysis-webapp.dll	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\bin\changeanalysis-webapp.pdb	Application file	someone@microsoft.com

To view other changes, select a change group in the timeline. To zoom in and out, scroll the timeline up and down

Level	Time	Name	Description	Initiated By
>	Jun 5 2019, 2:45:49 pm	\Areas\HelpPage\Views\Web.config	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\Views\Web.config	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\Web.config	Application file	someone@microsoft.com
<pre>1 <?xml version="1.0" encoding="utf-8"?> 2 <!-- 3 For more information on how to configure your ASP.NET 4 https://go.microsoft.com/fwlink/?LinkId=301879 5 --> 6 <configuration> 7 <connectionStrings> 8 - <add name="MyDbConnection" connectionString="Server=<!-- 9 <add name="StorageConnection" connectionString="C<!-- 10 </connectionStrings> 11 <appSettings> 12 <add key="webpages:Version" value="3.0.0.0" /> 13 <add key="webpages:Enabled" value="false" /></pre>				
>	Jun 5 2019, 2:45:49 pm	\bin\changeanalysis-webapp.dll	Application file	someone@microsoft.com
>	Jun 5 2019, 2:45:49 pm	\bin\changeanalysis-webapp.pdb	Application file	someone@microsoft.com

Optimize the performance and reliability of Azure Functions

3/13/2019 • 7 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. A function can become large due to 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 is 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 allows you to 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 is generally a best practice to use storage queues for cross function communication. The main reason is storage queues are cheaper and much easier to provision.

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 any time during the day with the same results. The function can exit when there is 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 db.
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 can have a serious impact on your work flow.

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 which 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

Re-use connections to external resources whenever possible. See [how to manage connections in Azure Functions](#).

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. Reference the assembly with a statement similar to the following example if using C# Scripts (.csx):

```
#r "..\Shared\MyAssembly.dll".
```

Otherwise, it is easy to accidentally deploy multiple test versions of the same binary that behave differently between functions.

Don't use verbose logging in production code. It has a negative performance impact.

Use async code but avoid blocking calls

Asynchronous programming is a recommended best practice. However, 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](#).

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 hosts file apply across all functions within the app, within a *single instance* of the function. For example, if you had a function app with 2 HTTP functions and concurrent requests set to 25, a request to either HTTP trigger would count towards the shared 25 concurrent requests. If that function app scaled to 10 instances, the 2 functions would effectively allow 250 concurrent requests (10 instances * 25 concurrent requests per instance).

HTTP concurrency host options

```
{  
    "http": {  
        "routePrefix": "api",  
        "maxOutstandingRequests": 200,  
        "maxConcurrentRequests": 100,  
        "dynamicThrottlesEnabled": true  
    }  
}
```

PROPERTY	DEFAULT	DESCRIPTION
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.
maxOutstandingRequests	200*	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows 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. *The default for version 1.x is unbounded (<code>-1</code>). The default for version 2.x in a consumption plan is 200. The default for version 2.x in a dedicated plan is unbounded (<code>-1</code>).

PROPERTY	DEFAULT	DESCRIPTION
maxConcurrentRequests	100*	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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 100. The default for version 2.x in a dedicated plan is unbounded (-1).
dynamicThrottlesEnabled	true*	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels. *The default for version 1.x is false. The default for version 2.x in a consumption plan is true. The default for version 2.x in a dedicated plan is false.

Other host configuration options can be found [in the host configuration document](#).

Next steps

For more information, see the following resources:

- [How to manage connections in Azure Functions](#)
- [Azure App Service best practices](#)

Work with Azure Functions Proxies

3/15/2019 • 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.

```
"backendurl": "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:

```
"backendurl": "https://localhost/api/httptriggerC#1?code=<keyvalue>&clientId=<keyname>"
```

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 }}	<code>{{ example }}</code> --> <code>{ example }</code>
\	\\\	<code>example.com\\text.html</code> --> <code>example.com\text.html</code>
"	\"	<code>\\"example\\"</code> --> <code>"example"</code>

Define a requestOverrides object

The requestOverrides object defines changes made to the request when the back-end resource is called. The object is defined by the following properties:

- **backend.request.method**: The HTTP method that's used to call the back-end.
- **backend.request.querystring.<ParameterName>**: A query string parameter that can be set for the call to the back-end. Replace <ParameterName> with the name of the parameter that you want to set. If the empty string is provided, the parameter is not included on the back-end request.
- **backend.request.headers.<HeaderName>**: A header that can be set for the call to the back-end. Replace <HeaderName> with the name of the header that you want to set. If you provide the empty string, the header is not 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

6/11/2019 • 5 minutes to read • [Edit Online](#)

This article describes the networking features available across the hosting options for Azure Functions. All of the following networking options provide some ability to access resources without using internet routable addresses, or restrict internet access to a function app.

The hosting models have different levels of network isolation available. Choosing the correct one will help you meet your network isolation requirements.

You can host function apps in a couple of ways:

- There's a set of plan options that run on a multitenant infrastructure, with various levels of virtual network connectivity and scaling options:
 - The [Consumption plan](#), which scales dynamically in response to load and offers minimal network isolation options.
 - The [Premium plan](#), which also scales dynamically, while offering more comprehensive network isolation.
 - The [Azure App Service plan](#), which operates at a fixed scale and offers similar network isolation 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

	CONSUMPTION PLAN	PREMIUM PLAN (PREVIEW)	APP SERVICE PLAN	APP SERVICE ENVIRONMENT
Inbound IP restrictions	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes
Outbound IP Restrictions	<input type="checkbox"/> No	<input type="checkbox"/> No	<input type="checkbox"/> No	<input type="checkbox"/> Yes
Virtual network integration	<input type="checkbox"/> No	<input type="checkbox"/> No	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes
Preview virtual network integration (Azure ExpressRoute and service endpoints outbound)	<input type="checkbox"/> No	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes
Hybrid Connections	<input type="checkbox"/> No	<input type="checkbox"/> No	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes
Private site access	<input type="checkbox"/> No	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes	<input type="checkbox"/> Yes

Inbound IP restrictions

You can use IP restrictions to define a priority-ordered list of IP addresses that are allowed/denied access to your app. The list can include IPv4 and IPv6 addresses. When there's one or more entries, an implicit "deny all" exists at the end of the list. IP restrictions work with all function-hosting options.

NOTE

To use the Azure portal editor, the portal must be able to directly access your running function app. Also, the device that you're using to access the portal must have its IP whitelisted. With network restrictions in place, you can still access any features on the **Platform features** tab.

To learn more, see [Azure App Service static access restrictions](#).

Outbound IP restrictions

Outbound IP restrictions are only available for functions deployed to an App Service Environment. You can configure outbound restrictions for the virtual network where your App Service Environment is deployed.

Virtual network integration

Virtual network integration allows your function app to access resources inside a virtual network. This feature is available in both the Premium plan and the App Service plan. If your app is in an App Service Environment, it's already in a virtual network and doesn't require the use of virtual network integration to reach resources in the same virtual network.

Virtual network integration gives your function app access to resources in your virtual network but doesn't grant [private site access](#) to your function app from the virtual network.

You can use virtual network integration to enable access from apps to databases and web services running in your virtual network. With virtual network integration, you don't need to expose a public endpoint for applications on your VM. You can use the private, non-internet routable addresses instead.

The generally available version of virtual network integration relies on a VPN gateway to connect function apps to a virtual network. It's available in functions hosted in an App Service plan. To learn how to configure this feature, see [Integrate your app with an Azure virtual network](#).

Preview version of virtual network integration

A new version of the virtual network integration feature is in preview. It doesn't depend on point-to-site VPN. It supports accessing resources across ExpressRoute or service endpoints. It's available in the Premium plan and in App Service plans scaled to PremiumV2.

Here are some characteristics of this version:

- You don't need a gateway to use it.
- You can access resources across ExpressRoute connections without any additional configuration beyond integrating with the ExpressRoute-connected virtual network.
- You can consume service-endpoint-secured resources from running functions. To do so, enable service endpoints on the subnet used for virtual network integration.
- You can't configure triggers to use service-endpoint-secured resources.
- Both the function app and the virtual network must be in the same region.
- The new feature requires an unused subnet in the virtual network that you deployed through Azure Resource Manager.
- Production workloads are not supported while the feature is in preview.
- Route tables and global peering are not yet available with the feature.
- One address is used for each potential instance of a function app. Because you can't change subnet size after assignment, use a subnet that can easily support your maximum scale size. For example, to support a Premium plan that can be scaled to 80 instances, we recommend a /25 subnet that provides 126 host addresses.

To learn more about using the preview version of virtual network integration, see [Integrate a function app with an](#)

Azure virtual network.

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 running in an [App Service plan](#) and an [App Service Environment](#).

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 does not know or care what the application protocol is, or what you're accessing. It simply provides network access.

To learn more, see the [App Service documentation for Hybrid Connections](#), which supports Functions in an App Service plan.

Private site access

Private site access refers to making your app accessible only from a private network such as from within an Azure virtual network.

- Private site access is available in the Premium and App Service plan when **Service Endpoints** are configured. For more information, see [virtual network service endpoints](#)
 - Keep in mind that with Service Endpoints, your function still has full outbound access to the internet, even with VNET integration configured.
- Private site access is available only with an App Service Environment configured with an internal load balancer (ILB). For more information, see [Create and use an internal load balancer with an App Service Environment](#).

There are many ways to access virtual network resources in other hosting options. But an App Service Environment is the only way to allow triggers for a function to occur over a virtual network.

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

12/3/2018 • 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](#) is scaled, a new range of outbound IP addresses may be assigned. When running on the Consumption plan, you may need to whitelist the entire data center.

Data center outbound IP addresses

If you need to whitelist the outbound IP addresses used by your function apps, another option is to whitelist the function apps' data center (Azure region). 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 an SSL binding, such as during [certificate renewal](#)).

When your function app runs in a [Consumption plan](#), the inbound IP address might also change when you haven't taken any actions such as the ones listed.

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](#), the outbound IP address might also change when you haven't taken any actions such as the ones listed.

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 on Kubernetes with KEDA

5/6/2019 • 2 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 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 down to 0 instances (when no events are occurring) and up 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 the Azure Functions Core Tools

By default, Core Tools installs both KEDA and Osiris components, which support event-driven and HTTP scaling, respectively. The installation uses `kubectl` running in the current context.

Install KEDA in your cluster by running the following install command:

```
func kubernetes install --namespace keda
```

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:

```
func init --docker-only
```

To build an image and deploy your functions to Kubernetes, run the following command:

```
func kubernetes deploy --name <name-of-function-deployment> --registry <container-registry-username>
```

Replace `<name-of-function-deployment>` with the name of your function app.

This creates a Kubernetes `Deployment` resource, a `ScaledObject` resource, and `Secrets`, which includes environment variables imported from your `local.settings.json` file.

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

You can run the following core tools command to remove KEDA from a Kubernetes cluster:

```
func kubernetes remove --namespace keda
```

Supported triggers in KEDA

KEDA is currently in beta with support for the following Azure Function triggers:

- [Azure Storage Queues](#)
- [Azure Service Bus Queues](#)
- [HTTP](#)
- [Apache Kafka](#)

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](#)

Azure Functions developers guide

7/4/2019 • 5 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
    }
  ]
}
```

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. 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, continuous deployment and runtime version. Think of a function app as a way to organize and collectively manage your functions.

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 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](#)

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.

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 "`~2`" indicates that your Function App will use 2.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 two major versions of the Azure Functions runtime:

TYPE	1.X	2.X ¹	TRIGGER	INPUT	OUTPUT
Blob storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓	✓	✓	✓
Event Grid	✓	✓	✓		
Event Hubs	✓	✓	✓		✓

Type	1.x	2.x	Trigger	Input	Output
HTTP & webhooks	✓	✓	✓		✓
Microsoft Graph Excel tables		✓		✓	✓
Microsoft Graph OneDrive files		✓		✓	✓
Microsoft Graph Outlook email		✓			✓
Microsoft Graph events		✓	✓	✓	✓
Microsoft Graph Auth tokens		✓		✓	
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ In 2.x, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

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
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 NodeJS developer reference](#)

Strategies for testing your code in Azure Functions

7/1/2019 • 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 function.

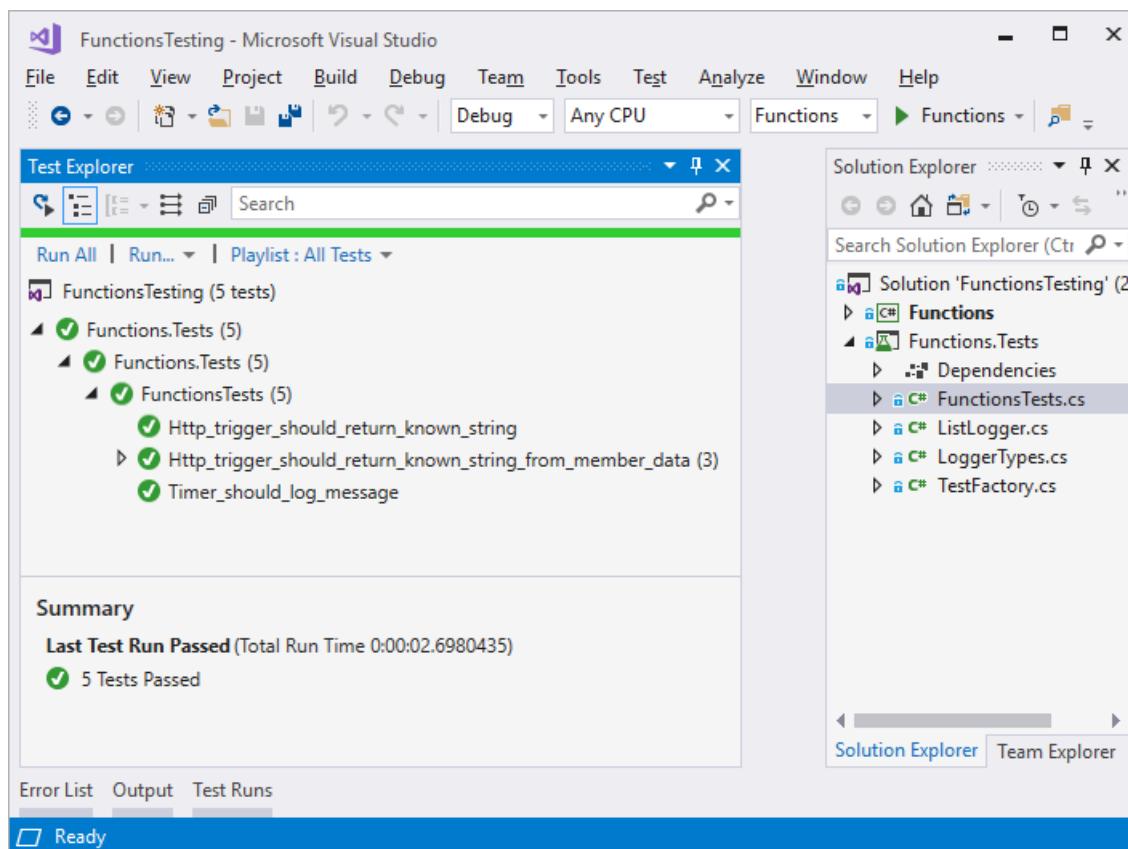
The content that follows is split into two different sections meant to target different languages and environments. You can learn to build tests in:

- [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 *HttpTrigger*.
3. [Create a timer function from the template](#) and name it *TimerTrigger*.

4. Create an xUnit Test app in Visual Studio by clicking **File > New > Project > Visual C# > .NET Core > xUnit Test Project** and name it *Functions.Test*.
5. Use Nuget to add a references from the test app [Microsoft.AspNetCore.Mvc](#)
6. Reference the *Functions* app from *Functions.Test* app.

Create test classes

Now that the applications 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.

The `ListLogger` class is meant to implement the `ILogger` interface and hold in internal list of messages for evaluation during a test.

Right-click on the *Functions.Test* application and select **Add > Class**, name it **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, **right-click** on the *Functions.Test* application and select **Add > Class**, name it **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, **right-click** on the `Functions.Test` application and select **Add > Class**, name it **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.

Next, **right-click** on the `Functions.Test` application and select **Add > Class**, name it **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 DefaultHttpRequest CreateHttpRequest(string queryStringKey, string queryStringValue)
        {
            var request = new DefaultHttpRequest(new DefaultHttpContext())
            {
                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.

Next, **right-click** on the `Functions.Test` application and select **Add > Class**, name it **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 HttpFunction.Run(request, logger);
            Assert.Equal("Hello, Bill", response.Value);
        }

        [Theory]
        [MemberData(nameof(TestFactory.Data), MemberType = typeof(TestFactory))]
        public async void Http_trigger_should_return_known_string_from_member_data(string queryStringKey,
string queryStringValue)
        {
            var request = TestFactory.CreateHttpRequest(queryStringKey, queryStringValue);
            var response = (OkObjectResult)await HttpFunction.Run(request, logger);
            Assert.Equal($"Hello, {queryStringValue}", response.Value);
        }

        [Fact]
        public void Timer_should_log_message()
        {
            var logger = (ListLogger)TestFactory.CreateLogger(LoggerTypes.List);
            TimerTrigger.Run(null, logger);
            var msg = logger.Logs[0];
            Assert.Contains("C# Timer trigger function executed at", msg);
        }
    }
}
```

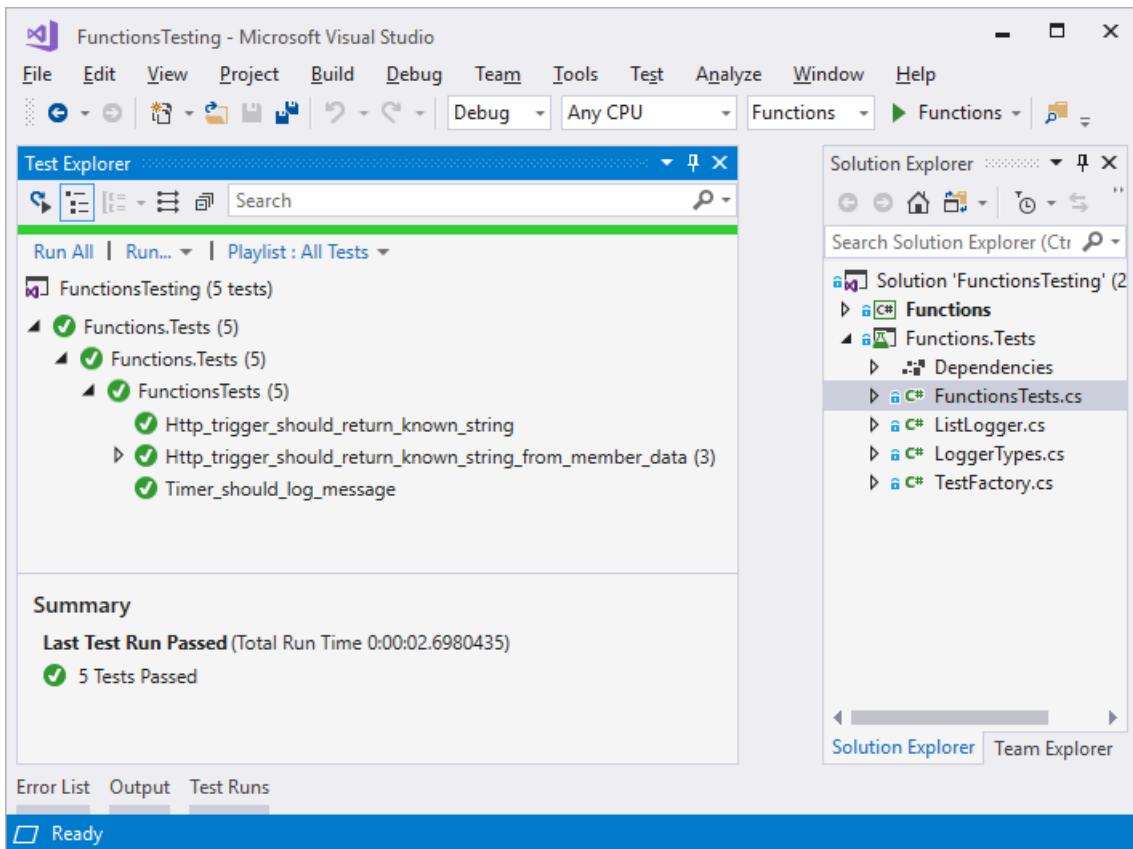
The members implemented in this class are:

- **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 use `System.Environment.GetEnvironmentVariable`.

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.

The screenshot shows the VS Code interface with the title bar "javascript-vscode - Visual Studio Code". The left sidebar has icons for Explorer, Search, Problems, and others. The "EXPLORER" section shows a file tree with ".vscode", "HttpTrigger", "node_modules", "testing", "TimerTrigger", "host.json", "local.settings.json", "package-lock.json", "package.json", and "proxies.json".

The right side shows the "TERMINAL" tab with the command "\$ npm test" followed by the output of a Jest test run:

```
$ npm test
> jest

PASS  HttpTrigger/index.test.js
PASS  TimerTrigger/index.test.js

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        14.435s, estimated 51s
Ran all test suites.
```

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 is as a fake timer instance.

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');

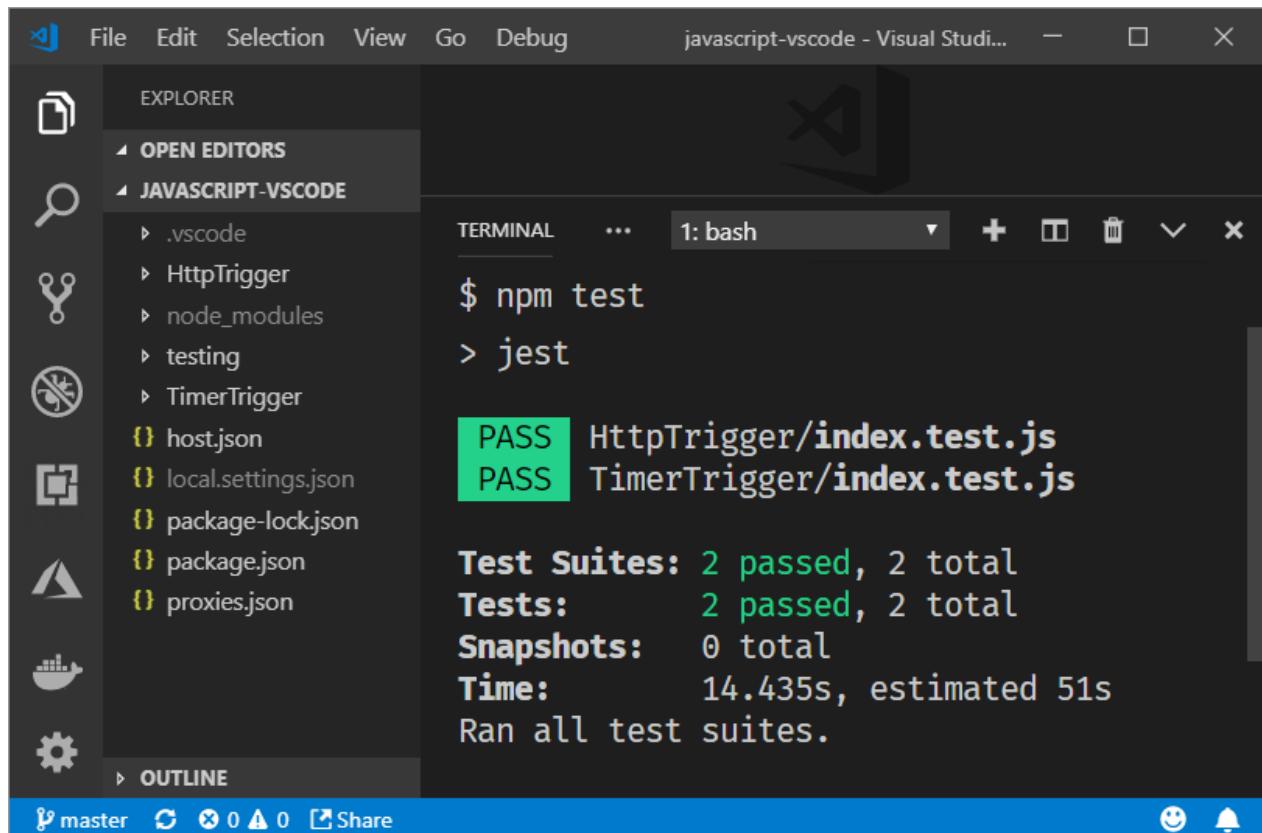
test('Timer trigger should log message', () => {
    timerFunction(context, timer);
    expect(context.log.mock.calls.length).toBe(1);
});
```

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",  
  "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](#)

Code and test Azure Functions locally

7/2/2019 • 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 the [VS Code documentation for Azure Functions](#).
- 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

7/30/2019 • 25 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 version 2.x runtime:

- [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](#).

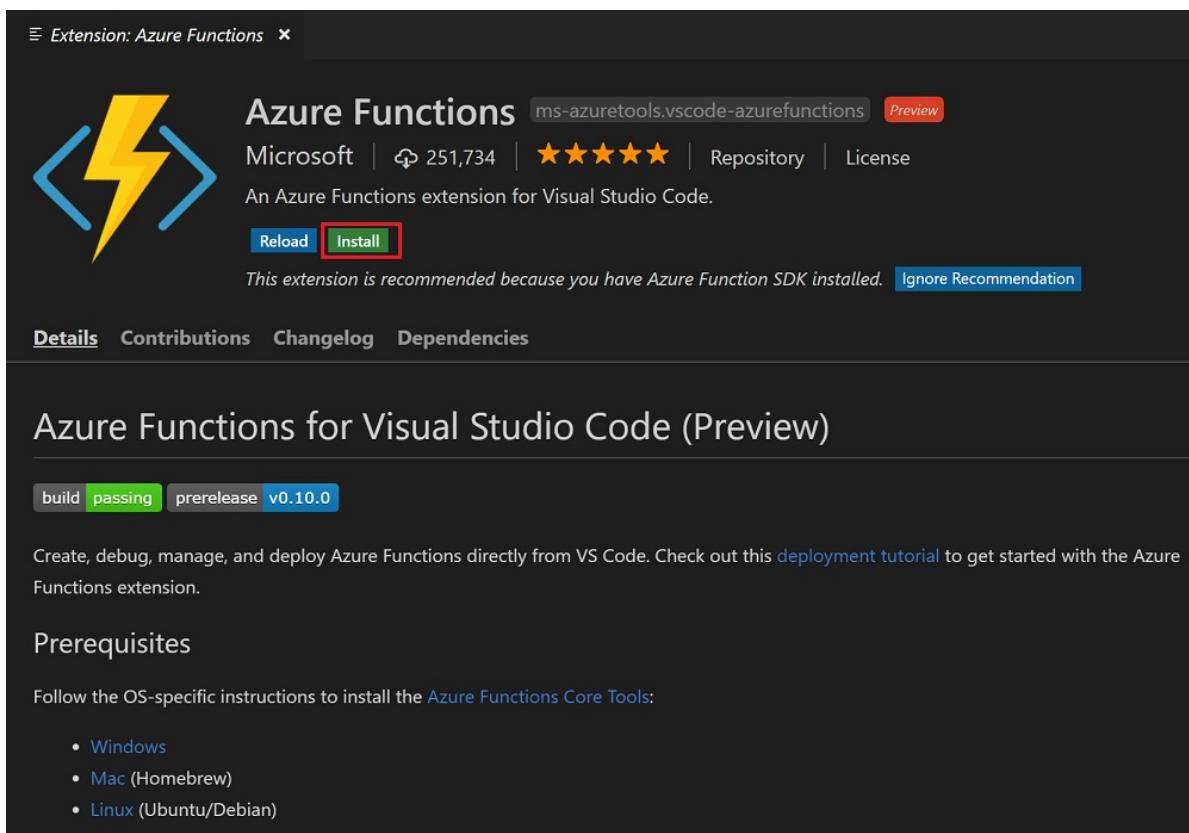
IMPORTANT

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](#).

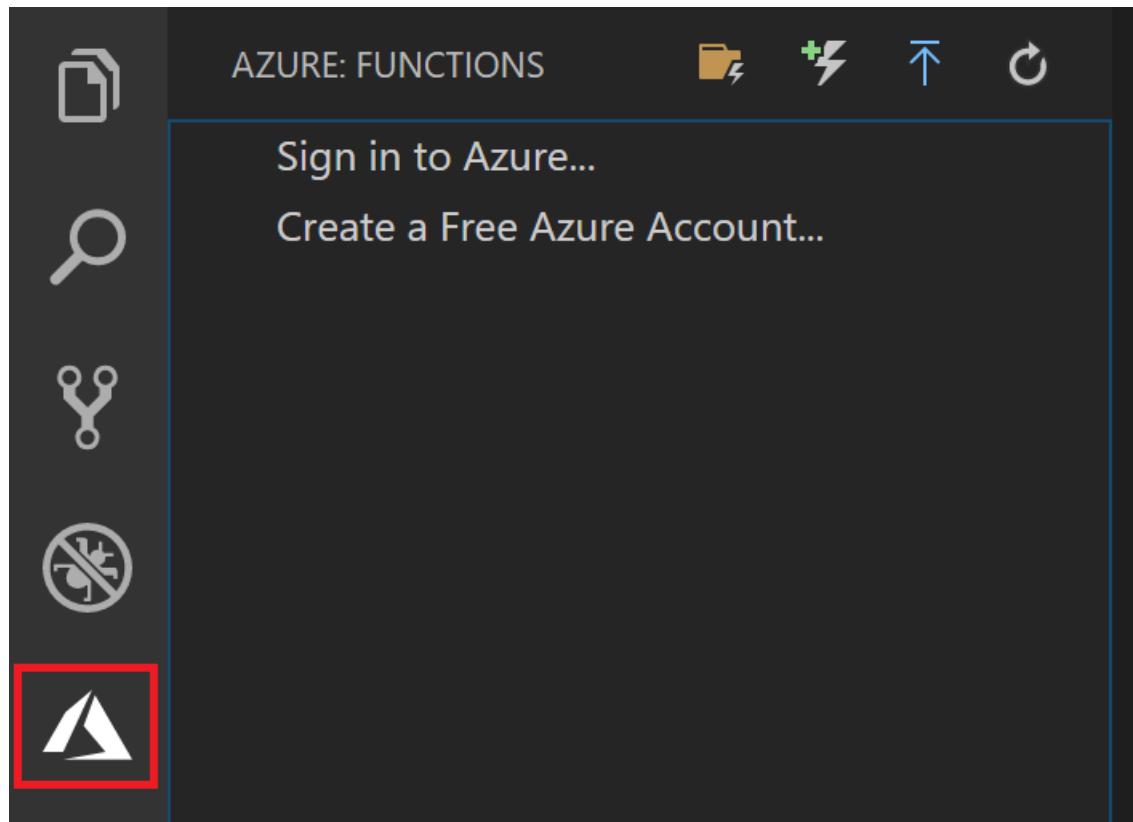
Install the Azure Functions extension

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](#).
2. Select **Install** to install the extension for Visual Studio Code:



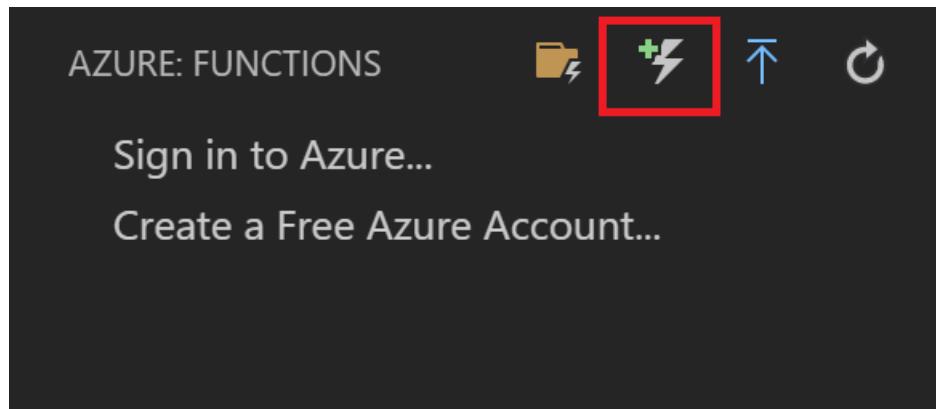
3. Restart Visual Studio Code and select the Azure icon on the Activity bar. You should see an Azure Functions area in the Side Bar.



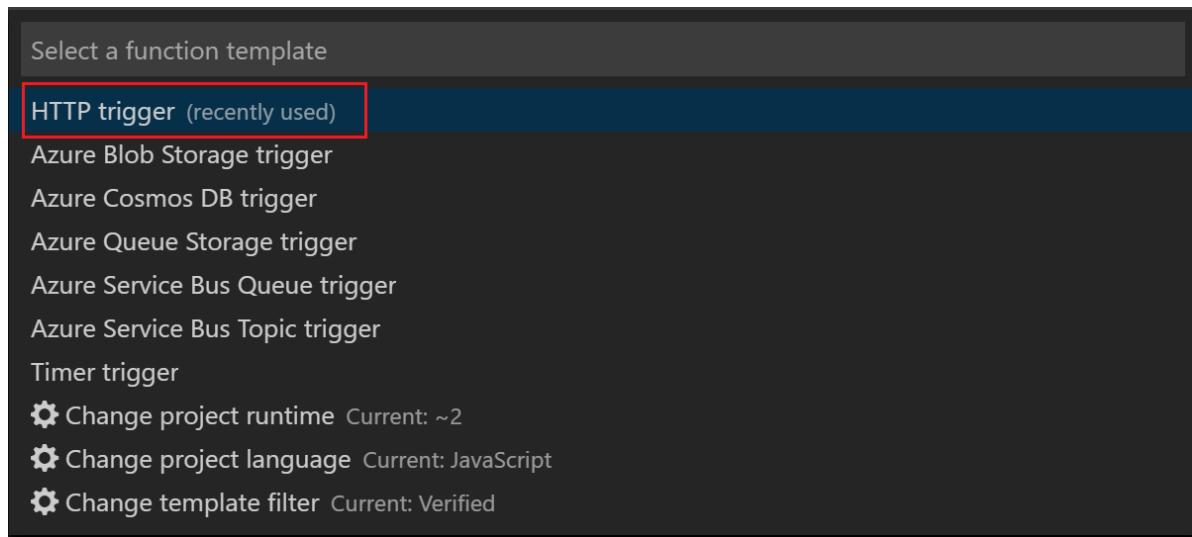
Create an Azure Functions project

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.

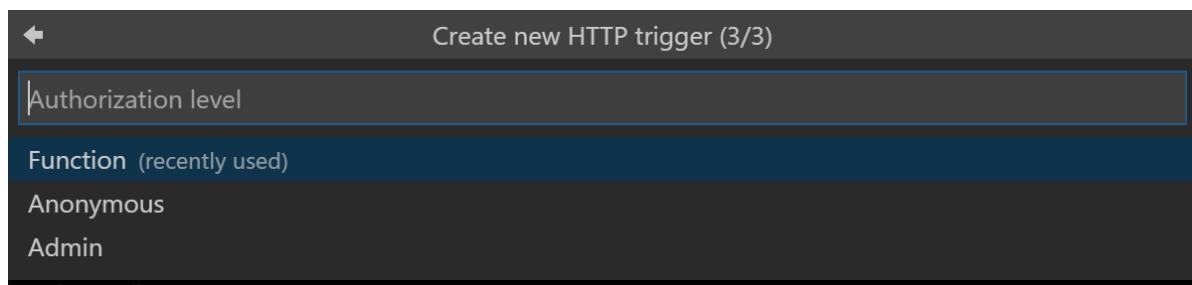
1. From **Azure: Functions**, select the **Create Function** icon:



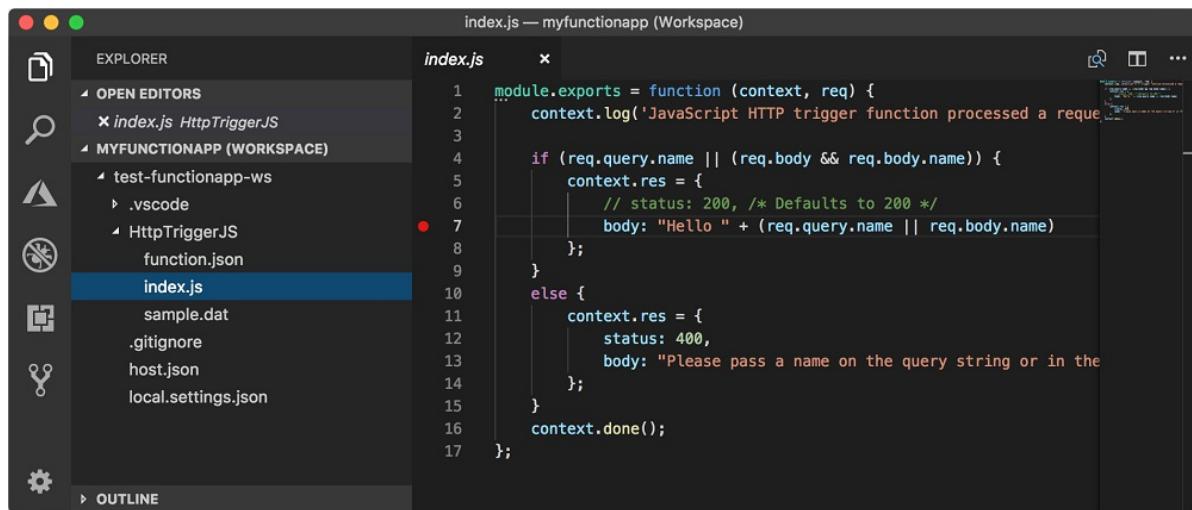
2. Select the folder for your function app project, and then [Select a language for your function project](#).
3. Select the **HTTP trigger** function template, or you can select **Skip for now** to create a project without a function. You can always [add a function to your project](#) later.



4. Type **HTTPTrigger** 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.



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.

At this point, you can add input and output bindings to your function by [modifying the function.json file](#) or 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.

JavaScript

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)"  
    }  
}
```

C# class library

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:

JavaScript

A new folder is created in the project. The folder contains a new function.json file and the new JavaScript code file.

C# class library

A new C# class library (.cs) file is added to your project.

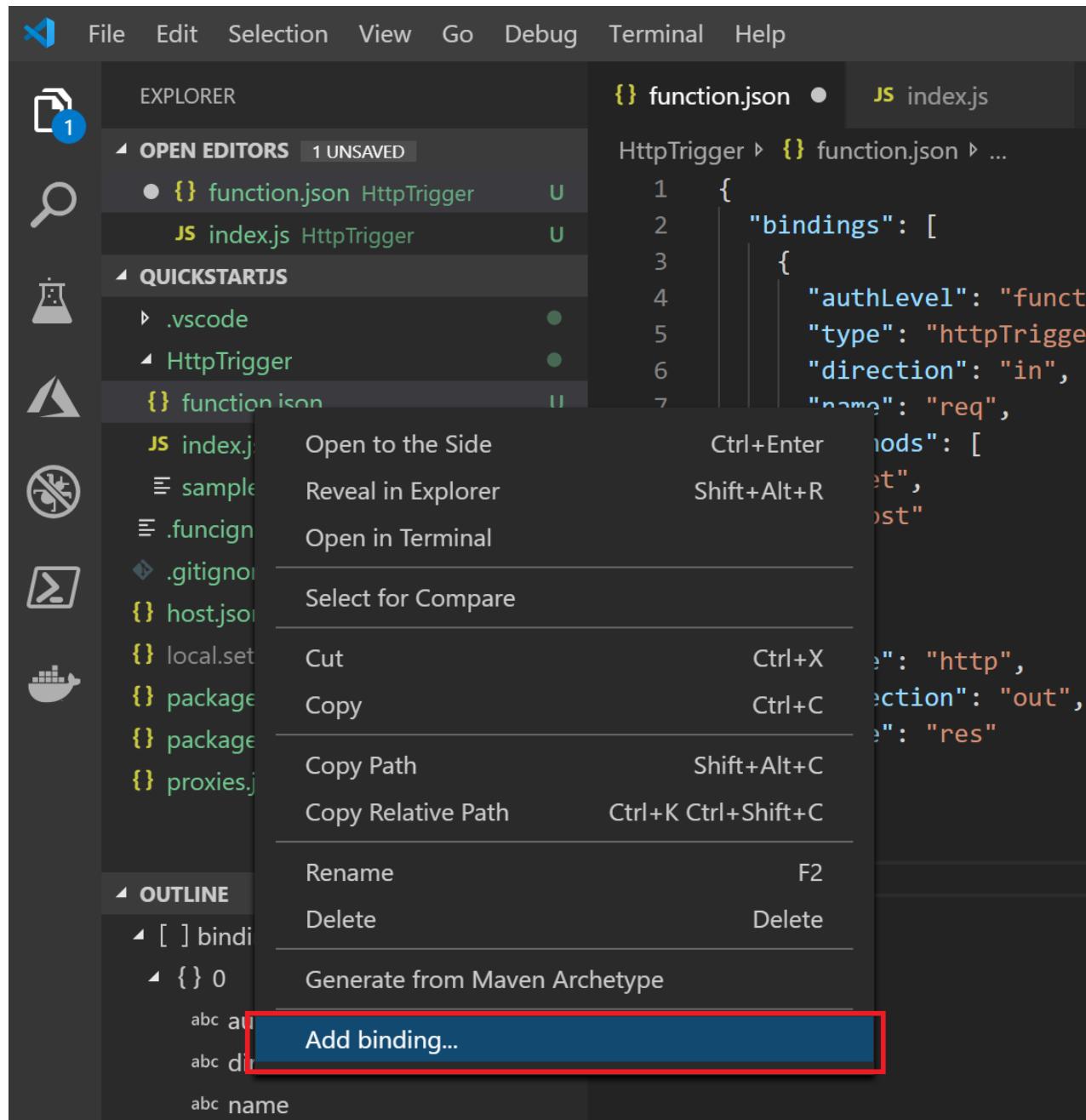
Add input and output bindings

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.

JavaScript

Visual Studio Code lets you add bindings to your function.json file by following a convenient set of prompts. To create a binding, right-click (Ctrl+click on macOS) the **function.json** file in your function folder and select **Add binding**:



Following are example prompts to define a new storage output binding:

PROMPT	VALUE	DESCRIPTION
Select binding direction	out	The binding is an output binding.
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.

PROMPT	VALUE	DESCRIPTION
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"	MyStorageConnection	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.

In this example, the following binding is added to the `bindings` array in your `function.json` file:

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

You can also add the same binding definition directly to your `function.json`.

In your function code, the `msg` binding is accessed from the `context`, as in this example:

```
context.bindings.msg = "Name passed to the function: " req.query.name;
```

To learn more, see the [Queue storage output binding](#) reference.

C# class library

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 two major versions of the Azure Functions runtime:

TYPE	1.X	2.X ¹	TRIGGER	INPUT	OUTPUT
Blob storage	✓	✓	✓	✓	✓

Type	1.x	2.x	Trigger	Input	Output
Cosmos DB	✓	✓	✓	✓	✓
Event Grid	✓	✓	✓		
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
Microsoft Graph Excel tables		✓		✓	✓
Microsoft Graph OneDrive files		✓		✓	✓
Microsoft Graph Outlook email		✓			✓
Microsoft Graph events		✓	✓	✓	✓
Microsoft Graph Auth tokens		✓		✓	
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ In 2.x, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

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, you can use one of two deployment methods:

- [Zip deploy with Run From Package enabled](#): Used for most Azure Functions deployments.

- [External package URL](#): Used for deployment to Linux apps on a [Consumption plan](#).

Quick function app creation

By default, Visual Studio Code 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 enable publishing with advanced options.

Enable publishing with advanced create options

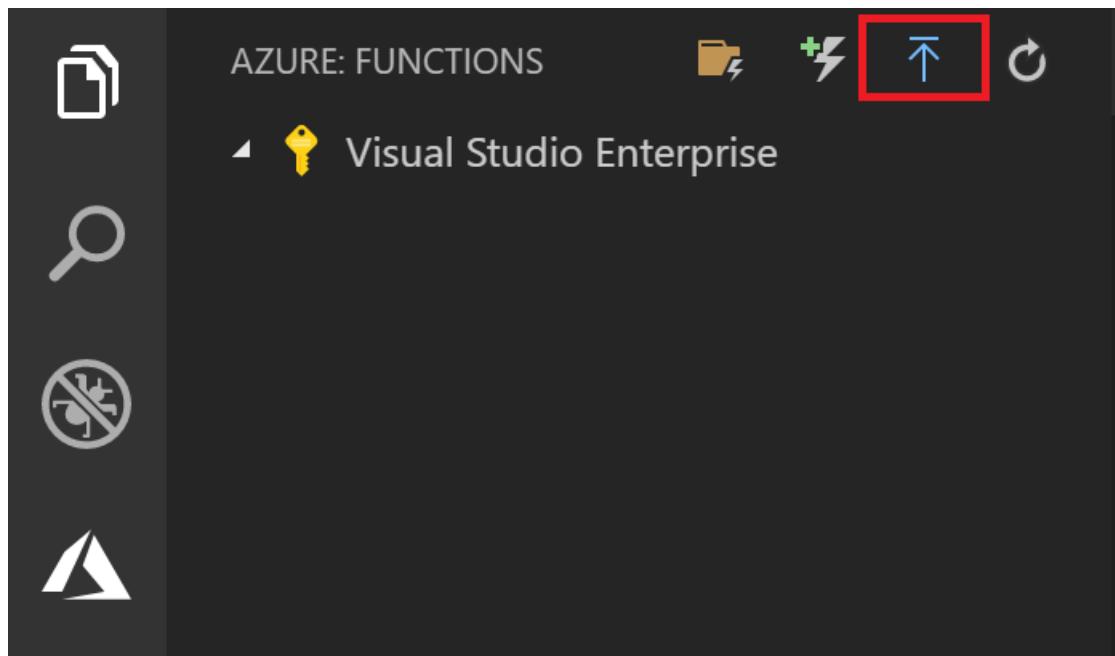
To get control over the settings associated with creating Azure Functions apps, update the Azure Functions extension to enable advanced settings:

1. Select **File > Preferences > Settings**.
2. Go to **User Settings > Extensions > Azure Functions**.
3. Select **Azure Function: Advanced Creation**.

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**.
4. Following the prompts, provide this information:

PROMPT	VALUE	DESCRIPTION
--------	-------	-------------

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.
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 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 or later

*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>"
  },
  "Host": {
    "LocalHttpPort": 7071,
    "CORS": "*",
    "CORSCredentials": false
  },
  "ConnectionStrings": {
    "SQLConnectionString": "<sqlclient-connection-string>"
  }
}
```

These settings are supported when you run projects locally:

SETTING	DESCRIPTION
---------	-------------

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> .
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 <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.</p> <p><code>AzureWebJobsStorage</code> is a required app setting for triggers other than HTTP.</p> <p>Version 2.x of the Functions runtime requires the <code>[FUNCTIONS_WORKER_RUNTIME]</code> setting, which is generated for your project by Core Tools.</p> <p>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.</p> <p>Values must be strings and not JSON objects or arrays.</p> <p>Setting names can't include a colon (<code>:</code>) or a double underline (<code>__</code>). These characters are reserved by the runtime.</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.
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 System.Data.SqlClient . 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.

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:

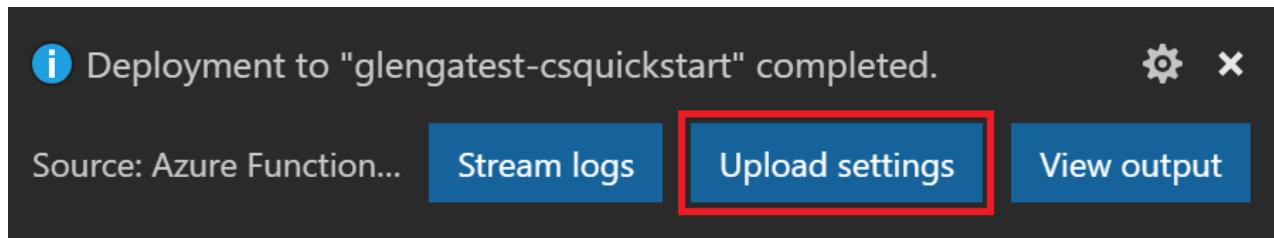
- [C# precompiled](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)

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:



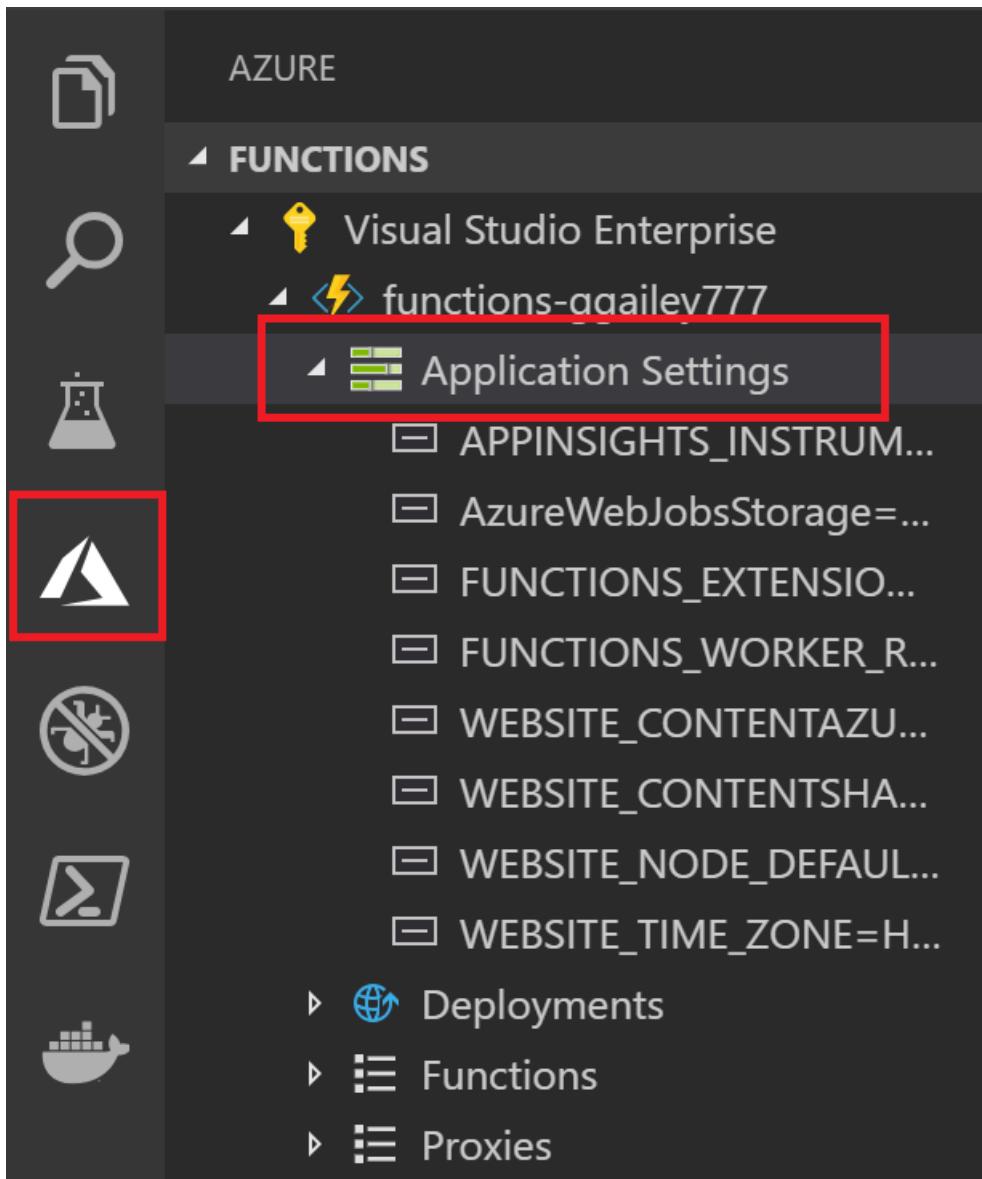
You can also publish settings by using the **Azure Functions: Upload Local Setting** command in the command palette. You can add individual settings to application settings in Azure by using the **Azure Functions: Add New Setting** command.

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 terminal tab active. The title bar says "functions-ggailey777 - ▾". The terminal window displays log output from an Azure Function named "HttpTrigger1". The logs show the function being executed via the host APIs, processing requests, and displaying its host status, which includes the ID "functions-ggailey777", state "Running", version "2.0.12507.0", and a commit hash.

```
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.

Functions makes it easy to add Application Insights integration to a function app from the [Azure portal](#).

1. In the [portal](#), select **All services > Function Apps**, select your function app, and then select the **Application Insights** banner at the top of the window

The screenshot shows the Azure Functions Overview page for the app 'functions-ggailey777'. The left sidebar lists various functions and features. The main area displays basic information about the app, including its status (Running), subscription (Visual Studio Enterprise), resource group (functions-ggailey777), location (Central US), and URL (https://functions-ggailey777). A prominent yellow box at the top right contains the message: 'Application Insights is not configured. Configure Application Insights to capture function logs.'

2. Create an Application Insights resource by using the settings specified in the table below the image.

The screenshot shows the 'Create new resource' page for Application Insights. It includes fields for 'New resource name' (set to 'functions-ggailey777') and 'Location' (set to 'West Europe'). Below these, there's an option to 'Select existing resource' with a search bar. At the bottom is a large blue 'OK' button.

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

3. Select **OK**. 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.
4. Back in your function app, select **Application settings**, and then scroll down to **Application settings**. If you see a setting named `APPINSIGHTS_INSTRUMENTATIONKEY`, Application Insights integration is enabled for your function app running in Azure.

To learn more, 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.

AZURE FUNCTIONS COMMAND	DESCRIPTION
Delete Function	Removes an existing function from a function app in Azure. Because this deletion doesn't affect your local project, instead consider removing the function locally and then republishing your project .
Delete Proxy	Removes an Azure Functions proxy from your function app in Azure. To learn more about proxies, see Work with Azure Functions Proxies .
Delete Setting	Deletes a function app setting in Azure. This deletion doesn't affect settings in your local.settings.json file.
Disconnect from Repo	Removes the continuous deployment connection between a function app in Azure and a source control repository.
Download Remote Settings	Downloads settings from the chosen function app in Azure into your local.settings.json file. If the local file is encrypted, it's decrypted, updated, and encrypted again. If there are settings that have conflicting values in the two locations, you're prompted to choose how to proceed. Be sure to save changes to your local.settings.json file before you run this command.
Edit settings	Changes the value of an existing function app setting in Azure. This command doesn't affect settings in your local.settings.json file.
Encrypt settings	Encrypts individual items in the <code>Values</code> array in the local settings . In this file, <code>IsEncrypted</code> is also set to <code>true</code> , which specifies that the local runtime 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.

AZURE FUNCTIONS COMMAND	DESCRIPTION
Set AzureWebJobsStorage	Sets the value of the <code>AzureWebJobsStorage</code> application setting. This setting is required by Azure Functions. It's set when a function app is created in Azure.
Start	Starts a stopped function app in Azure.
Start Streaming Logs	Starts the streaming logs for the function app in Azure. Use streaming logs during remote troubleshooting in Azure if you need to see logging information in near-real time. To learn more, see Streaming logs .
Stop	Stops a function app that's running in Azure.
Stop Streaming Logs	Stops the streaming logs for the function app in Azure.
Toggle as Slot Setting	When enabled, ensures that an application setting persists for a given deployment slot.
Uninstall Azure Functions Core Tools	Removes Azure Functions Core Tools, which is required by the extension.
Upload Local Settings	Uploads settings from your local.settings.json file to the chosen function app in Azure. If the local file is encrypted, it's decrypted, uploaded, and encrypted again. If there are settings that have conflicting values in the two locations, you're prompted to choose how to proceed. Be sure to save changes to your local.settings.json file before you run this command.
View Commit in GitHub	Shows you the latest commit in a specific deployment when your function app is connected to a repository.
View Deployment Logs	Shows you the logs for a specific deployment to the function app in Azure.

Next steps

To learn more about Azure Functions Core Tools, see [Work with Azure Functions Core Tools](#).

To learn more about developing functions as .NET class libraries, see [Azure Functions C# developer reference](#).

This article also provides links to examples of how to use attributes to declare the various types of bindings supported by Azure Functions.

Develop Azure Functions using Visual Studio

7/28/2019 • 16 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, you can learn more at [An introduction to Azure Functions](#).

Visual Studio provides the following benefits when developing 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, you should complete the [Functions quickstart for Visual Studio](#).

Unless otherwise noted, procedures and examples shown are for Visual Studio 2019.

Prerequisites

Azure Functions Tools is included in the Azure development workload of Visual Studio starting with Visual Studio 2017. Make sure you include the **Azure development** workload in your Visual Studio installation.

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

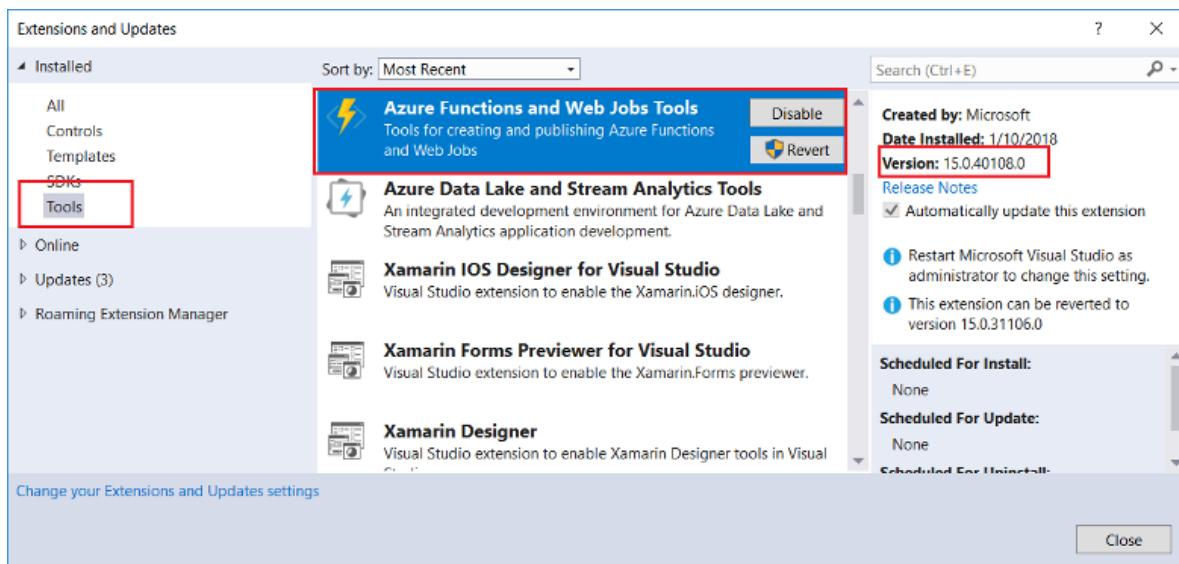
Other resources that you need, such as an Azure Storage account, are created in your subscription during the publishing process.

NOTE

In Visual Studio 2017, the Azure development workload installs the Azure Functions Tools as a separate extension. When you update your Visual Studio 2017, also make sure that you are 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.

Check your tools version in Visual Studio 2017

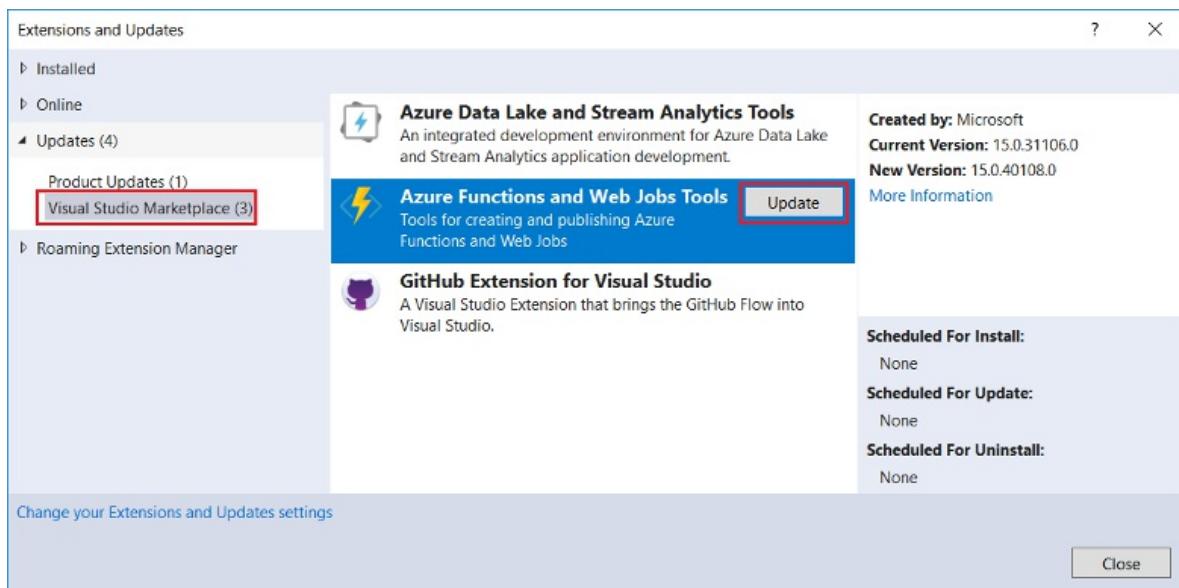
1. From the **Tools** menu, choose **Extensions and Updates**. Expand **Installed > Tools** and choose **Azure Functions and Web Jobs Tools**.



2. Note the installed **Version**. You can 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, close Visual Studio to trigger the tools update using the VSIX installer.
3. In the installer, choose **OK** to start and then **Modify** to update the tools.
4. After the update is complete, choose **Close** and 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 can be published to a function app in Azure. You can use a function app to group functions as a logical unit for management, deployment, and sharing.

of resources.

1. In Visual Studio, on the **File** menu, select **New > Project**.
2. In the **Create a new project** dialog box, search for `functions`, choose the **Azure Functions** template, and select **Next**.
3. Enter a name for your project, and 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. In **Create a new Azure Functions application**, use the following options:
 - **Azure Functions v2 (.NET Core) 1**
 - **HTTP trigger**
 - **Storage Account: Storage Emulator**
 - **Authorization level: Anonymous**

OPTION	SUGGESTED VALUE	DESCRIPTION
Functions runtime	Azure Functions 2.x (.NET Core)	This setting creates a function project that uses the version 2.x runtime of Azure Functions, which supports .NET Core. Azure Functions 1.x supports the .NET Framework. For more information, see Target Azure Functions runtime version .
Function template	HTTP trigger	This setting creates a function triggered by an HTTP request.
Storage Account	Storage Emulator	An HTTP trigger doesn't use the Azure Storage account connection. All other trigger types require a valid Storage account connection string. Because Functions requires a storage account, one is assigned or created when you publish your project to Azure.
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 in the HTTP and webhook bindings .

NOTE

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-triggered function.

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 are not 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. The **Copy to Output Directory** setting for this file should always be **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>"  
    },  
    "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> .

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 <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.</p> <p><code>AzureWebJobsStorage</code> is a required app setting for triggers other than HTTP.</p> <p>Version 2.x of the Functions runtime requires the [<code>FUNCTIONS_WORKER_RUNTIME</code>] setting, which is generated for your project by Core Tools.</p> <p>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.</p> <p>Values must be strings and not JSON objects or arrays. Setting names can't include a colon (:) or a double underline (_). These characters are reserved by the runtime.</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 <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 System.Data.SqlClient . 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.

Settings in local.settings.json are not uploaded automatically when you publish the project. To make sure that these settings also exist in your function app in Azure, you must upload them after you publish your project. To learn more, see [Function app settings](#).

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 [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, you must 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 is 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.

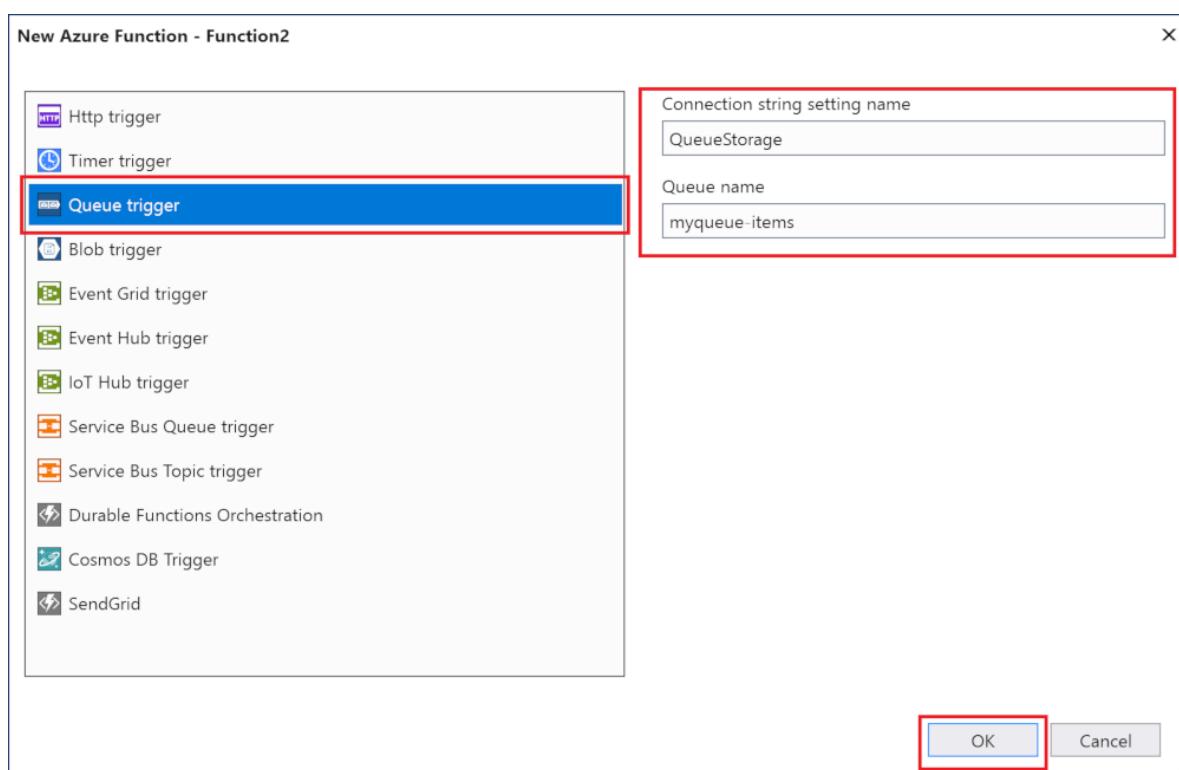
To set the storage account connection string:

1. In Visual Studio, open **Cloud Explorer**, expand **Storage Account > Your Storage Account**, then in the **Properties** tab 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.

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 on your project node and select **Add > New Item**. Select **Azure Function**, type a **Name** for the class, and click **Add**.
2. Choose your trigger, set the binding properties, and click **Create**. The following example shows the settings when creating a Queue storage triggered function.



This trigger example uses a connection string with a key named **QueueStorage**. This connection string setting must be defined in the [local.settings.json file](#).

3. Examine the newly added class. You see a static **Run** method, that is 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 triggered 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 queue triggered 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](#).

You can 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 have [configured the project for local development](#).
2. Add the appropriate NuGet extension package for the specific binding. For more information, see [Local C# development using Visual Studio](#) in the Triggers and Bindings article. The binding-specific NuGet package requirements are found 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](#). These values are used when the function runs locally. When the function runs in the function app in Azure, the [function app settings](#) are used.
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 two major versions of the Azure Functions runtime:

TYPE	1.X	2.X ¹	TRIGGER	INPUT	OUTPUT
Blob storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓	✓	✓	✓
Event Grid	✓	✓	✓		
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
Microsoft Graph Excel tables		✓		✓	✓
Microsoft Graph OneDrive files		✓		✓	✓
Microsoft Graph Outlook email		✓			✓
Microsoft Graph events		✓	✓	✓	✓
Microsoft Graph Auth tokens		✓		✓	
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

¹ In 2.x, all bindings except HTTP and Timer must be registered. See [Register binding extensions](#).

Testing functions

Azure Functions Core Tools lets you run Azure Functions project on your local development computer. You are prompted to install these tools the first time you start a function from Visual Studio.

To test your function, press F5. If prompted, accept the request from Visual Studio to download and install Azure Functions Core (CLI) tools. You may also need to enable a firewall exception so that the tools can handle HTTP requests.

With the project running, you can test your code as you would test deployed function. For more information, see [Strategies for testing your code in Azure Functions](#). When running in debug mode, breakpoints are hit in Visual Studio as expected.

To learn more about using the Azure Functions Core Tools, see [Code and test Azure functions locally](#).

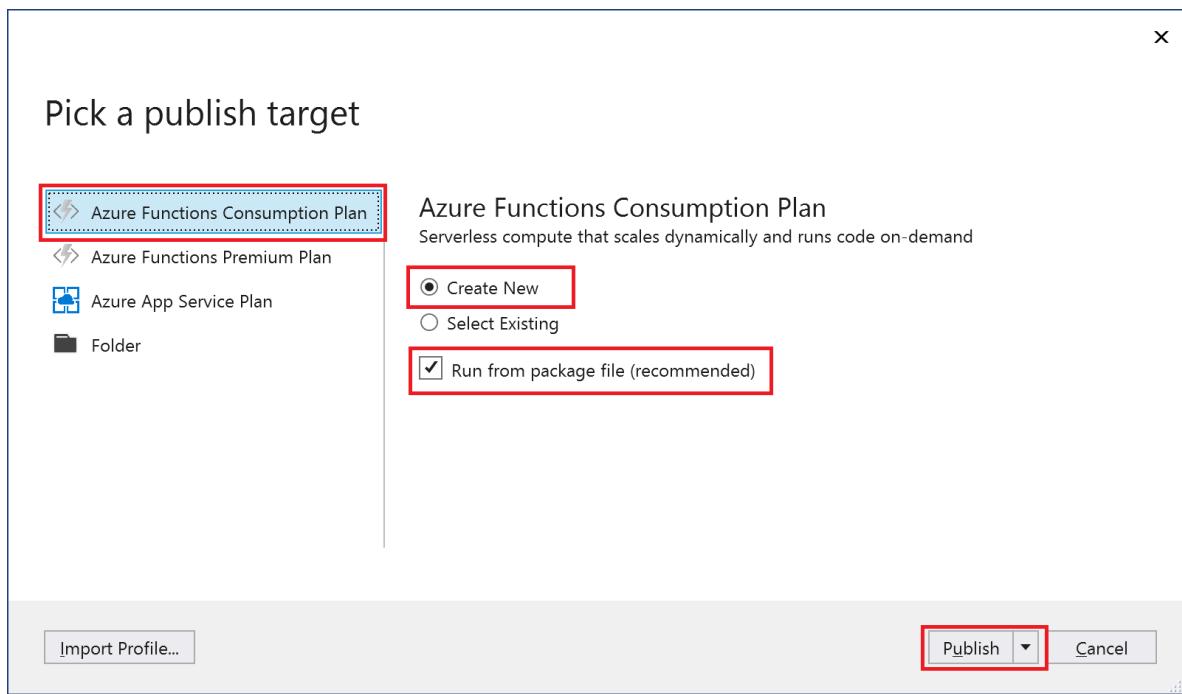
Publish to Azure

When publishing from Visual Studio, one of two deployment methods are used:

- [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 the **Pick a publish target** dialog, use the publish options as specified in the table below the image:



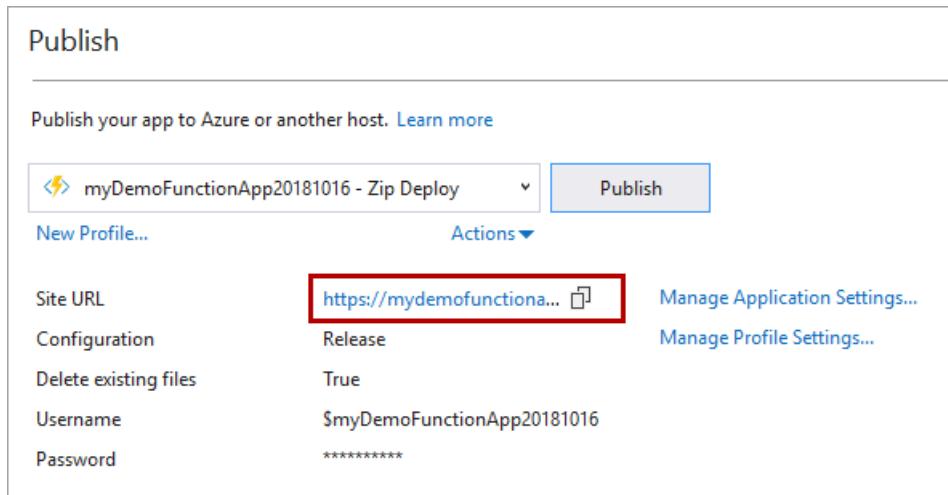
OPTION	DESCRIPTION
Azure Functions Consumption plan	When you publish your project to a function app that runs in a Consumption plan , you only pay for executions of your functions app. Other hosting plans incur higher costs. To learn more, see Azure Functions scale and hosting .
Create new	A new function app, with related resources, is created in Azure. When you choose Select Existing , all files in the existing function app in Azure are overwritten by files from the local project. Only use this option when republishing updates to an existing function app.
Run from package file	Your function app is deployed using Zip Deploy with Run-From-Package mode enabled. This is the recommended way of running your functions, which results in better performance.

3. Select **Publish**. If you haven't already signed-in to your Azure account from Visual Studio, select **Sign-in**. You can also create a free Azure account.
4. In the **App Service: Create new** dialog, use the **Hosting** settings as specified in the table below the image:

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Globally unique name	Name that uniquely identifies your new function app. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Choose your subscription	The Azure subscription to use.
Resource Group	myResourceGroup	Name of the resource group in which to create your function app. Choose New to create a new resource group.
Hosting Plan	Consumption plan	Make sure to choose the Consumption under Size after you select New to create a serverless plan. Also, choose a Location in a region near you or near other services your functions access. When you run in a plan other than Consumption , you must manage the scaling of your function app .
Azure Storage	General-purpose storage account	An Azure storage account is required by the Functions runtime. Select New to create a general-purpose storage account. You can also use an existing account that meets the storage account requirements .

5. Select **Create** to create a function app and related resources in Azure with these settings and deploy your function project code.

6. After the deployment is complete, make a note of the **Site URL** value, which is the address of your function app in Azure.



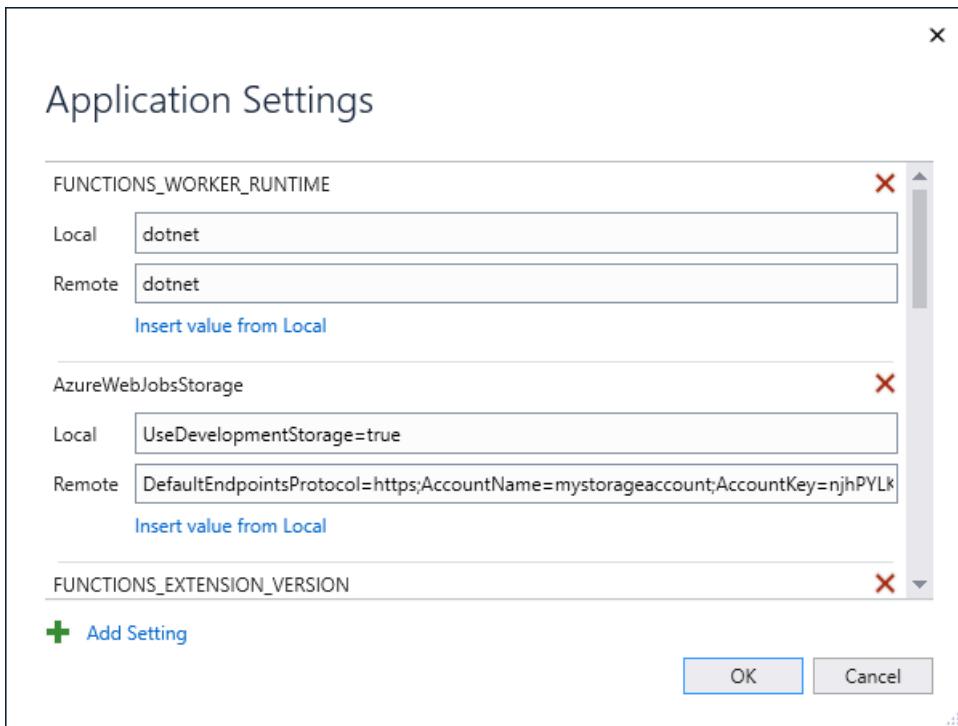
Function app settings

Any settings you added in the local.settings.json must be also added to the function app in Azure. These settings are not uploaded automatically when you publish the project.

The easiest way to upload the required settings to your function app in Azure is to use the **Manage Application Settings...** link that is displayed after you successfully publish your project.

The screenshot shows the 'Publish' dialog for a function app named 'FunctionApp8'. It displays a message: 'Azure successfully configured: How was your experience?'. Below it is a 'Create new profile' button. The 'Summary' section shows publishing details: Site URL (http://functionapp20180118122544.azurewebsites.net), Configuration (Release), Delete existing files (False), Username (\$FunctionApp20180118122544), and Password (*****). On the right side of the summary table, there are three links: 'Manage Application Settings...' (which is highlighted with a red box), 'Manage Profile Settings...', 'Rename profile...', and 'Delete profile'.

This displays the **Application Settings** dialog for the function app, where you can add new application settings or modify existing ones.



Local represents a setting value in the local.settings.json file, and **Remote** is the current setting 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**.

You can also manage application settings in one of these other ways:

- [Using the Azure portal](#).
- [Using the `--publish-local-settings` publish option in the Azure Functions Core Tools](#).
- [Using 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 enable Application Insights for your function app:

Functions makes it easy to add Application Insights integration to a function app from the [Azure portal](#).

1. In the [portal](#), select **All services > Function Apps**, select your function app, and then select the **Application Insights** banner at the top of the window

2. Create an Application Insights resource by using the settings specified in the table below the image.

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

3. Select **OK**. 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.
4. Back in your function app, select **Application settings**, and then scroll down to **Application settings**. If you see a setting named `APPINSIGHTS_INSTRUMENTATIONKEY`, Application Insights integration is enabled for your function app running in Azure.

To learn more, see [Monitor Azure Functions](#).

Next steps

To learn more about the Azure Functions Core Tools, see [Code and test Azure functions locally](#).

To learn more 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

7/24/2019 • 20 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 two 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 1.x: supports version 1.x of the runtime. This version of the tools is only supported on Windows computers and is installed from an [npm package](#). With this version, you can create functions in experimental languages that are not officially supported. For more information, see [Supported languages in Azure Functions](#)
- Version 2.x: supports [version 2.x of the runtime](#). This version supports [Windows](#), [macOS](#), and [Linux](#). Uses platform-specific package managers or npm for installation.

Unless otherwise noted, the examples in this article are for version 2.x.

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 2.x

Version 2.x of the tools uses the Azure Functions runtime 2.x that is built on .NET Core. This version is supported on all platforms .NET Core 2.x supports, including [Windows](#), [macOS](#), and [Linux](#).

IMPORTANT

You can bypass the requirement for installing the .NET Core 2.x SDK by using [extension bundles](#).

Windows

The following steps use npm to install Core Tools on Windows. You can also use [Chocolatey](#). For more information, see the [Core Tools readme](#).

1. Install [Node.js](#), which includes npm. For version 2.x of the tools, only Node.js 8.5 and later versions are supported.
2. Install the Core Tools package:

```
npm install -g azure-functions-core-tools
```

It may take a few minutes for npm to download and install the Core Tools package.

3. If you do not plan to use [extension bundles](#), install the [.NET Core 2.x SDK for Windows](#).

MacOS with Homebrew

The following steps use Homebrew to install the Core Tools on macOS.

1. Install [Homebrew](#), if it's not already installed.
2. Install the Core Tools package:

```
brew tap azure/functions
brew install azure-functions-core-tools
```

3. If you do not plan to use [extension bundles](#), install [.NET Core 2.x SDK for macOS](#).

Linux (Ubuntu/Debian) with APT

The following steps use [APT](#) to install Core Tools on your Ubuntu/Debian Linux distribution. For other Linux distributions, see the [Core Tools readme](#).

1. Register the Microsoft product key as trusted:

```
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg
sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg
```

2. Verify your Ubuntu server is running one of the appropriate versions from the table below. To add the apt source, run:

```
sudo sh -c 'echo "deb [arch=amd64] https://packages.microsoft.com/repos/microsoft-ubuntu-$(lsb_release -cs)-prod $(lsb_release -cs) main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-get update
```

LINUX DISTRIBUTION	VERSION
Ubuntu 18.10	cosmic
Ubuntu 18.04	bionic

LINUX DISTRIBUTION	VERSION
Ubuntu 17.04	zesty
Ubuntu 16.04/Linux Mint 18	xenial

3. Install the Core Tools package:

```
sudo apt-get install azure-functions-core-tools
```

4. If you do not plan to use [extension bundles](#), install [.NET Core 2.x SDK for Linux](#).

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 2.x requires you to select a default language for your project when it is initialized, and 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
```

When you provide a project name, a new folder with that name is created and initialized. Otherwise, the current folder is initialized.

In version 2.x, when you run the command you must choose a runtime for your project. If you plan to develop JavaScript functions, choose **node**:

```
Select a worker runtime:
dotnet
node
```

Use the up/down arrow keys to choose a language, then press Enter. 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 2.x-only, unless otherwise noted:

OPTION	DESCRIPTION
<code>--csx</code>	Initializes a C# script (.csx) project. You must specify <code>--csx</code> in subsequent commands.

OPTION	DESCRIPTION
--docker	Create a Dockerfile for a container using a base image that is based on the chosen --worker-runtime. Use this option when you plan to publish to a custom Linux container.
--force	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.
--no-source-control -n	Prevents the default creation of a Git repository in version 1.x. In version 2.x, the git repository isn't created by default.
--source-control	Controls whether a git repository is created. By default, a repository isn't created. When true, a repository is created.
--worker-runtime	Sets the language runtime for the project. Supported values are dotnet, node (JavaScript), java, and python. When not set, you are prompted to choose your runtime during initialization.

IMPORTANT

By default, version 2.x of the Core Tools creates 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 are implemented as extension packages. In version 2.x of the Azure Functions runtime, you have to explicitly register the extensions for the binding types used in your functions. The exceptions to this are HTTP bindings and timer triggers, which do not require extensions.

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.

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 functions.json file.

Register individual extensions

If you need to install extensions that aren't in a bundle, you can manually register individual extension packages for specific bindings.

NOTE

To manually register extensions by using `func extensions install`, you must have the .NET Core 2.x SDK installed.

After you have updated your `function.json` file to include all the bindings that your function needs, run the following command in the project folder.

```
func extensions install
```

The command reads the `function.json` file to see which packages you need, installs them, and rebuilds the extensions project. 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.

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>"
  },
  "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> .
<code>Values</code>	<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 <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.</p> <p><code>AzureWebJobsStorage</code> is a required app setting for triggers other than HTTP.</p> <p>Version 2.x of the Functions runtime requires the [<code>FUNCTIONS_WORKER_RUNTIME</code>] setting, which is generated for your project by Core Tools.</p> <p>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.</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>). These characters are reserved by the runtime.</p>
<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.
<code>CORScredentials</code>	When set to <code>true</code> , allows <code>withCredentials</code> requests.

SETTING	DESCRIPTION
<code>ConnectionStrings</code>	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.

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](#)

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 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](#). Navigate to your storage account, select **Access keys** in **Settings**, then copy one of the **Connection string** values.

The screenshot shows the Microsoft Azure Storage accounts settings page for a specific storage account. The left sidebar lists various services like All services, Storage accounts, Function Apps, App Services, etc. The main pane shows the storage account details, including its name (cs23e9dd381f085x4f40x8ec), access keys, and connection strings. The 'Access keys' section is highlighted with a red box. The 'key1' and 'key2' fields show long key values, and their corresponding connection strings are also shown.

- Use [Azure Storage Explorer](#) to connect to your Azure account. In the **Explorer**, expand your subscription, select your storage account, and copy the primary or secondary connection string.

The screenshot shows the Microsoft Azure Storage Explorer interface. The left sidebar has a tree view under 'EXPLORER' showing 'Visual Studio Enterprise' and 'Storage Accounts'. A specific storage account, 'cs23e9dd381f085x4f40x8ec', is selected and highlighted with a red box. The 'Properties' tab is selected in the bottom-left, displaying detailed information about the storage account, including its name, type, subscription, location, and connection strings. The 'Primary Connection String' field is highlighted with a red box.

- Use Core Tools 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 are not already signed in to Azure, you are prompted to do so.

Create a function

To create a function, run the following command:

```
func new
```

In version 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) Generates the same C# script (.csx) templates used in version 1.x and in the portal.
<code>--language -l</code>	The template programming language, such as C#, F#, or JavaScript. This option is required in version 1.x. In version 2.x, do not use this option or choose a language that matches the worker runtime.
<code>--name -n</code>	The function name.
<code>--template -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:

```
func host start
```

The `host` command is only required in version 1.x.

`func host 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. Version 2.x only.
<code>--cert</code>	The path to a .pfx file that contains a private key. Only used with <code>--useHttps</code> . Version 2.x only.
<code>--cors-credentials</code>	Allow cross-origin authenticated requests (i.e. cookies and the Authentication header) Version 2.x only.
<code>--cors</code>	A comma-separated list of CORS origins, with no spaces.
<code>--language-worker</code>	Arguments to configure the language worker. Version 2.x only.
<code>--nodeDebugPort -n</code>	The port for the node debugger to use. Default: A value from launch.json or 5858. Version 1.x only.
<code>--password</code>	Either the password or a file that contains the password for a .pfx file. Only used with <code>--cert</code> . Version 2.x only.
<code>--port -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).
<code>--script-root --prefix</code>	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 <code>MyProject/bin/Debug/netstandard2.0</code> . In this case, set the prefix as <code>--script-root MyProject/bin/Debug/netstandard2.0</code> . This is the root of the function app when running in Azure.

OPTION	DESCRIPTION
--timeout -t	The timeout for the Functions host to start, in seconds. Default: 20 seconds.
--useHttps	Bind to <code>https://localhost:{port}</code> rather than to <code>http://localhost:{port}</code> . By default, this option creates a trusted certificate on your computer.

For a C# class library project (.csproj), you must include the `--build` option to generate the library .dll.

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, authentication 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. On 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:

```
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, 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. 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.

```
curl --request POST -H "Content-Type:application/json" --data '{"input": "sample queue data"}'
http://localhost:7071/admin/functions/QueueTriggerJS
```

Using the `func run` command in version 1.x

IMPORTANT

The `func run` command is not supported in version 2.x of the tools. For more information, see the topic [How to target Azure Functions runtime versions](#).

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 -c</code>	Inline content.
<code>--debug -d</code>	Attach a debugger to the host process before running the function.
<code>--timeout -t</code>	Time to wait (in seconds) until the local Functions host is ready.
<code>--file -f</code>	The file name to use as content.

OPTION	DESCRIPTION
<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.

Deployment (project files)

To publish your local code to a function app in Azure, use the `publish` command:

```
func azure functionapp publish <FunctionAppName>
```

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 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 2.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 both versions, 1.x and 2.x:

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 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 only supported in version 2.x:

OPTION	DESCRIPTION
<code>--publish-settings-only -o</code>	Only publish settings and skip the content. Default is prompt.

OPTION	DESCRIPTION
--list-ignored-files	Displays a list of files that are ignored during publishing, which is based on the .funcignore file.
--list-included-files	Displays a list of files that are published, which is based on the .funcignore file.
--nozip	Turns the default Run-From-Package mode off.
--build-native-deps	Skips generating .wheels folder when publishing python function apps.
--additional-packages	List of packages to install when building native dependencies. For example: <code>python3-dev libevent-dev</code> .
--force	Ignore pre-publishing verification in certain scenarios.
--csx	Publish a C# script (.csx) project.
--no-build	Skip building dotnet functions.
--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.

Deployment (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. 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 by using the Azure CLI, the integration in your function app in Azure isn't done.

Functions makes it easy to add Application Insights integration to a function app from the [Azure portal](#).

1. In the [portal](#), select **All services > Function Apps**, select your function app, and then select the **Application Insights** banner at the top of the window

The screenshot shows the Azure portal interface for a function app named "functions-ggailey777". The main area is the "Overview" tab, which includes sections for Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggailey777), and URL (https://functions-ggailey777). Below this is a "Configured features" section with links to "Function app settings" and "Application settings". On the left, there's a sidebar with "Function Apps" selected, showing the function app details and various management options like "Integrate", "Manage", and "Monitor". A red box highlights the "Application Insights" banner at the top right of the main content area.

2. Create an Application Insights resource by using the settings specified in the table below the image.

The screenshot shows the "Create new resource" page for Application Insights. It has two main sections: "Create new resource" (selected) and "Select existing resource". Under "Create new resource", there are fields for "New resource name" (containing "functions-ggailey777:") and "Location" (set to "West Europe"). At the bottom is a large blue "OK" button. A red box highlights the "New resource name" field and the "OK" button.

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Unique app name	It's easiest to use the same name as your function app, which must be unique in your subscription.

SETTING	SUGGESTED VALUE	DESCRIPTION
Location	West Europe	If possible, use the same region as your function app, or one that's close to that region.

3. Select **OK**. 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.
4. Back in your function app, select **Application settings**, and then scroll down to **Application settings**. If you see a setting named `APPINSIGHTS_INSTRUMENTATIONKEY`, Application Insights integration is enabled for your function app running in Azure.

To learn more, see [Monitor Azure Functions](#).

Next steps

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 Azure function with Java and IntelliJ

6/27/2019 • 3 minutes to read • [Edit Online](#)

This article shows you:

- How to create a [serverless](#) function project with IntelliJ IDEA and Apache Maven
- 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:

- [Java Developer Kit](#) (JDK), version 8
- [Apache Maven](#), version 3.0 or higher
- [IntelliJ IDEA](#), Community or Ultimate versions with Maven
- [Azure CLI](#)

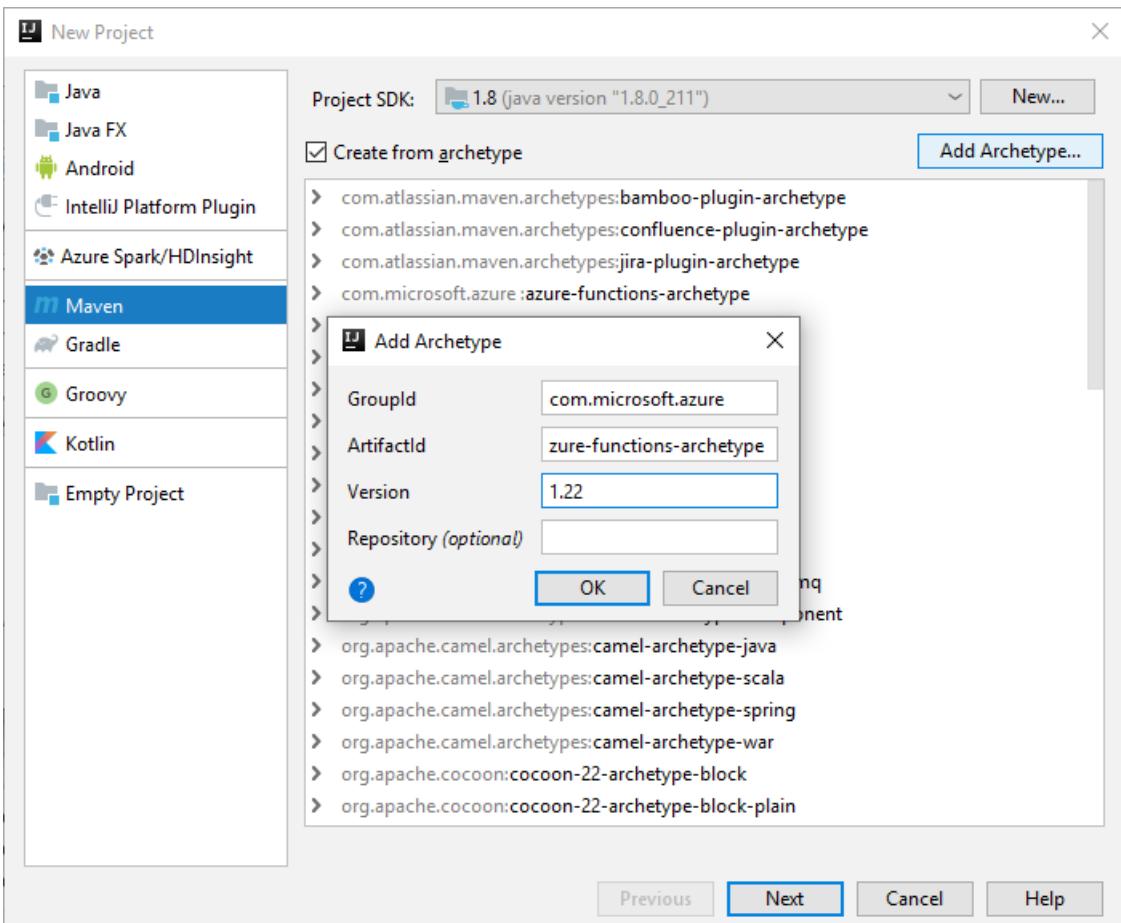
IMPORTANT

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

We recommend that you install [Azure Functions Core Tools, version 2](#). It provides a local development environment for writing, running, and debugging Azure Functions.

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-archetype](#).
4. In the **Add Archetype** window, complete the fields as follows:
 - *GroupId*: com.microsoft.azure
 - *ArtifactId*: azure-functions-archetype
 - *Version*: Use the latest version **1.22** from [the central repository](#)



5. Select **OK**, and then select **Next**.

6. Enter your details for current project, and select **Finish**.

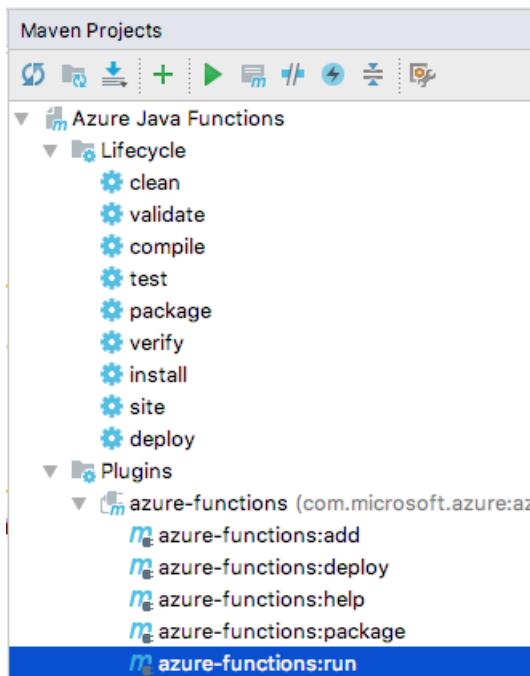
Maven creates the project files in a new folder with the same name as the *ArtifactId* value. The project's generated code is a simple [HTTP-triggered](#) function that echoes the body of the triggering HTTP request.

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.

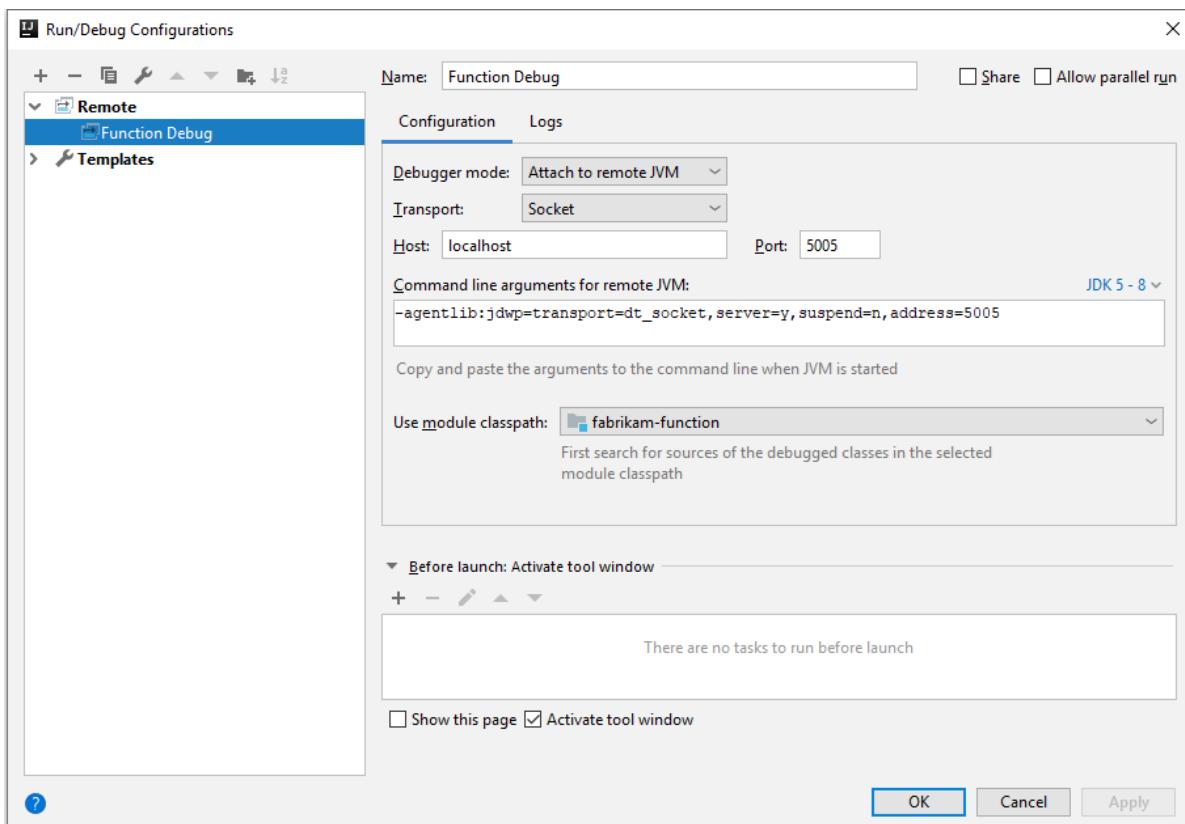
Debug the function in IntelliJ

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 [sign 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

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

Create your first function with Java and Eclipse

6/17/2019 • 3 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](#)

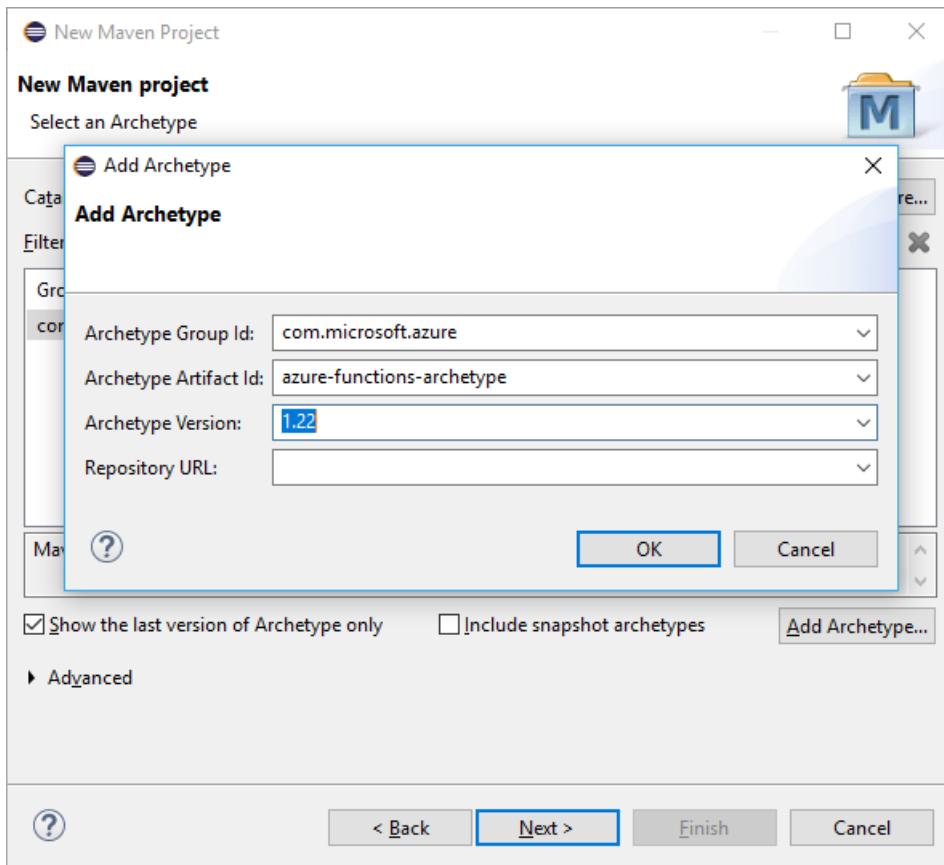
IMPORTANT

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

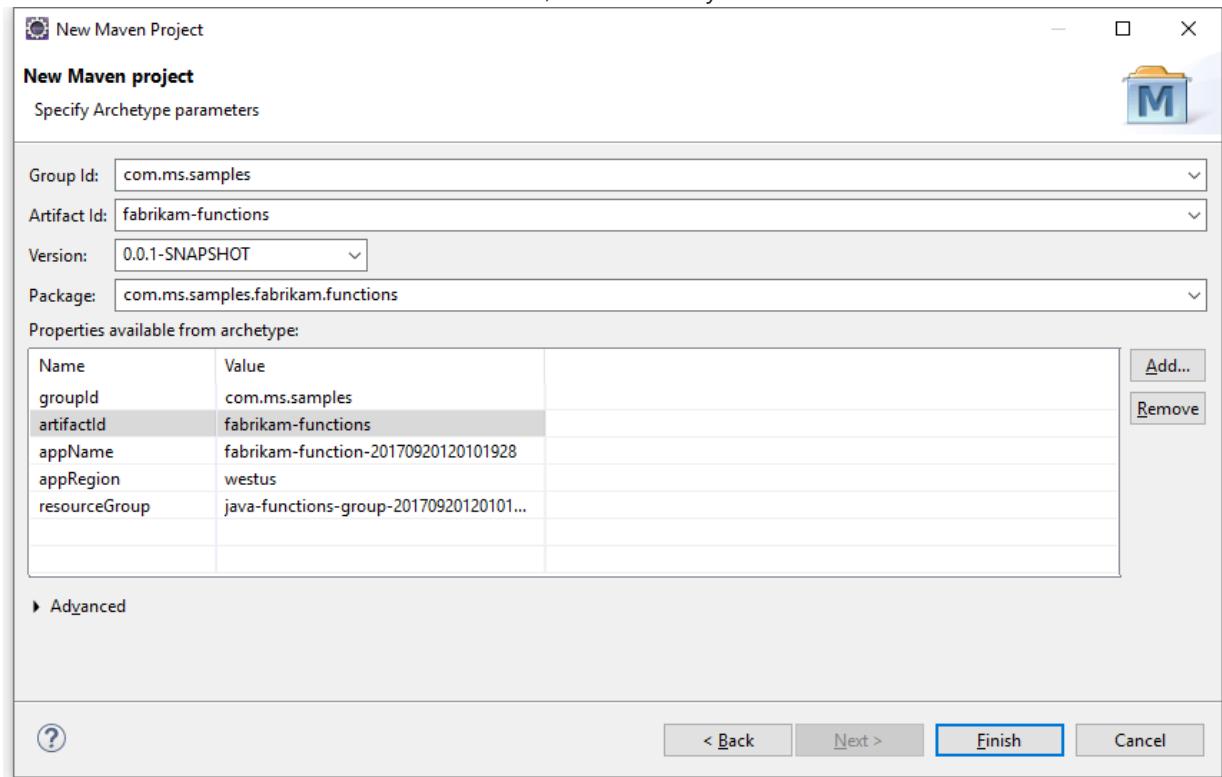
It's highly recommended to also install [Azure Functions Core Tools, version 2](#), which provide a local environment for running and debugging Azure Functions.

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. Select **Add Archetype** and add the entries for the [azure-functions-archetype](#).
 - Archetype Group ID: com.microsoft.azure
 - Archetype Artifact ID: azure-functions-archetype
 - Version: Use latest version **1.22** from [the central repository](#)



4. Click **OK** and then click **Next** to enter values like the following snapshot(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.

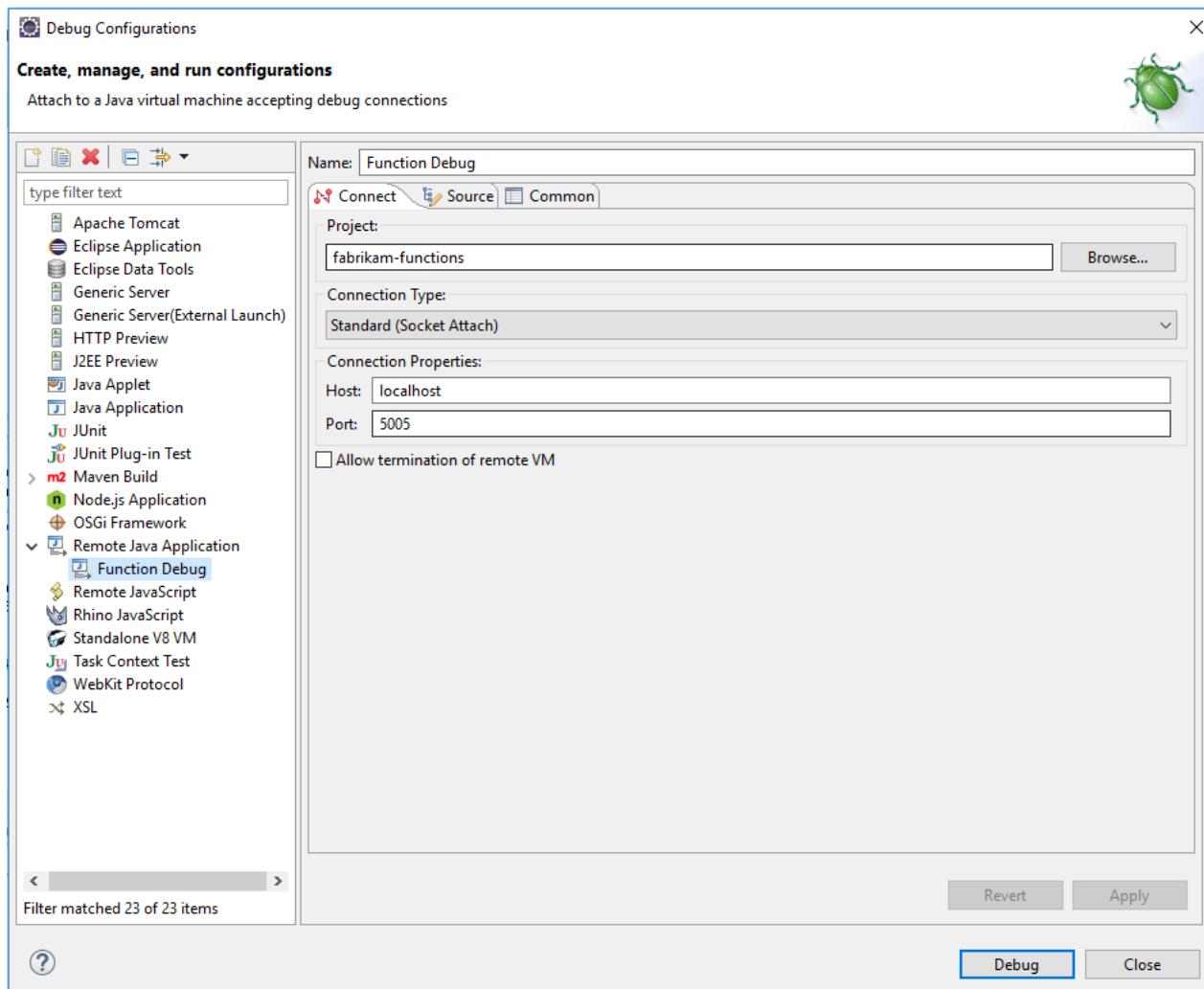
1. Right-click on the generated project, then choose **Run As** and **Maven build**.
2. In the **Edit Configuration** dialog, Enter `package` in the **Goals** and **Name** fields, then select **Run**. This will build and package the function code.
3. Once the build is complete, create another Run configuration 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] -----
```

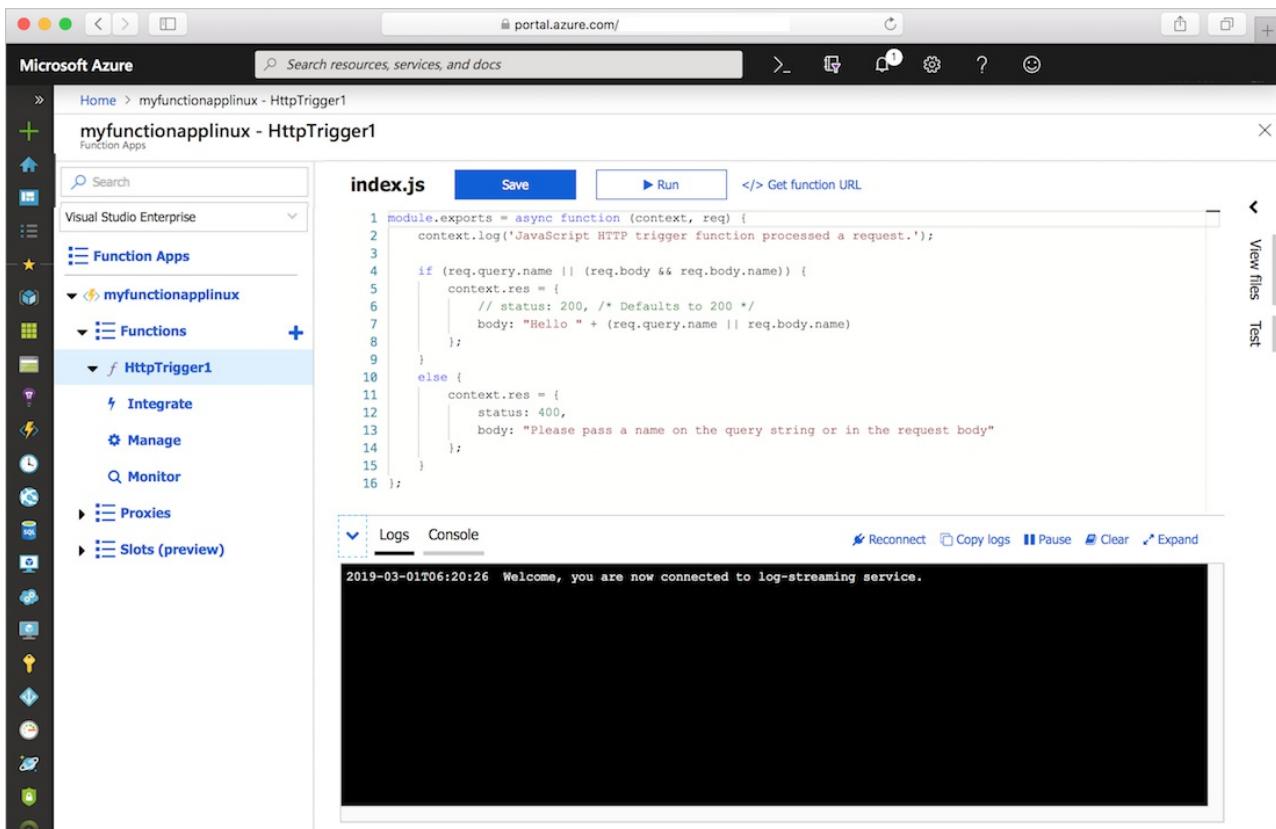
Next steps

- 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 a function app on Linux in an Azure App Service plan

5/6/2019 • 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](#).



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

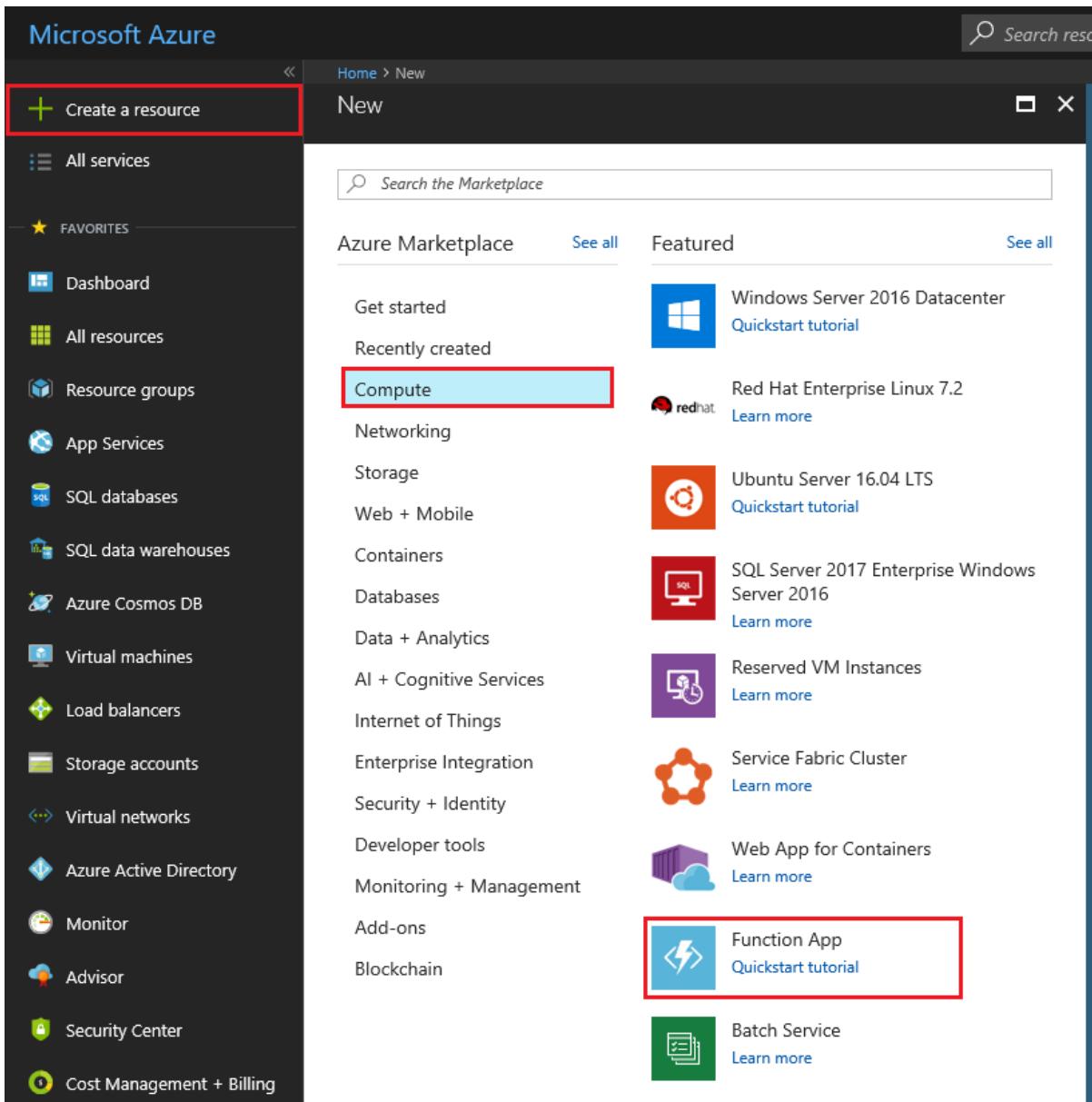
Sign in to Azure

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 logic unit for easier management, deployment, and sharing of resources. In this article, you create an App Service plan when you create your function app.

1. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.



2. Use the function app settings as specified in the table below the image.

The screenshot shows the 'Function App' creation wizard in the Azure portal. The configuration steps are outlined by a red border:

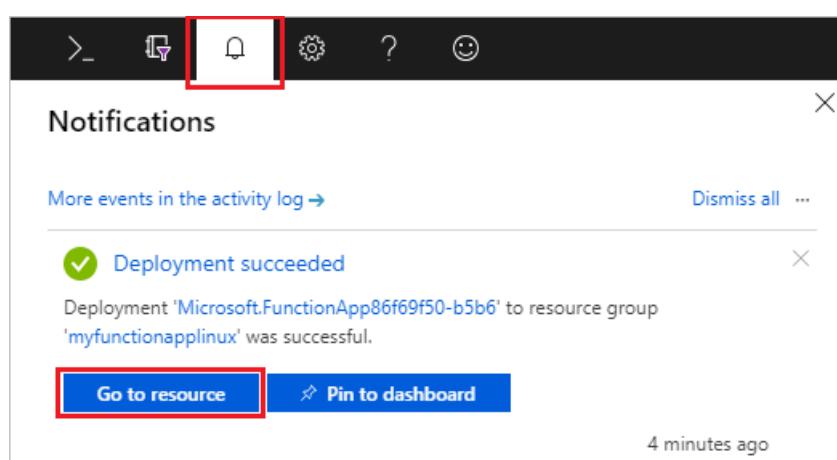
- App name:** myfunctionapplinux.azurewebsites.net
- Subscription:** Visual Studio Enterprise
- Resource Group:** Create new (myfunctionapplinux)
- OS:** Linux
- Publish:** Code (selected)
- Hosting Plan:** App Service Plan
- App Service plan/Location:** myAppServicePlanS1(North Euro...)
- Runtime Stack:** JavaScript
- Storage:** Create new (myfunctionapp1938e)
- Application Insights:** myfunctionapplinux

At the bottom, there are 'Create' and 'Automation options' buttons.

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are a-z, 0-9, and -.
Subscription	Your subscription	The subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Linux	The function app runs on Linux.
Publish	Code	The default Linux container for your Runtime Stack is used. All you need to provide is your function app project code. Another option is to publish a custom Docker image .

SETTING	SUGGESTED VALUE	DESCRIPTION
Hosting plan	App Service plan	Hosting plan that defines how resources are allocated to your function app. When you run in an App Service plan, you can control the scaling of your function app .
App Service plan/Location	Create plan	Choose Create new and supply an App Service plan name. Choose a Location in a region near you or near other services your functions access. Choose your desired Pricing tier . You can't run both Linux and Windows function apps in the same App Service plan.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions. Python support is in preview at this time.
Storage	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 .
Application Insights	Enabled	Application Insights is disabled by default. We recommend enabling Application Insights integration now and choosing a hosting location near your App Service plan location. If you want to do this later, see Monitor Azure Functions .

3. Select **Create** to provision and deploy the function app.
4. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



5. Select **Go to resource** to view your new function app.

Next, you create a function in the new function app. Even after your function app is available, it may take a few minutes to be fully initialized.

Create an HTTP triggered 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. In your new function app, choose the **Overview** tab, and after it loads completely choose **+ New function**.

The screenshot shows the Azure Function App Overview page for a function app named 'myfunctionapplinux'. The 'Overview' tab is selected. On the left, there's a sidebar with 'Function Apps' and a list of functions: 'myfunctionapplinux' (selected), 'Functions', 'Proxies', and 'Slots (preview)'. Below the sidebar, there's a summary table with columns for Status, Subscription, Resource group, URL, Availability, Subscription ID, Location, and App Service plan / pricing tier. Underneath the table, there's a section titled 'Configured features' with links for 'Function app settings', 'Application settings', and 'Application Insights'. A message says 'You have created a function app!' and 'Now it is time to add your code...'. At the bottom right, there's a blue button with a plus sign and the text '+ New function', which is highlighted with a red box.

2. In the **Quickstart** tab, choose **In-portal**, and select **Continue**.

Home > myfunctionapplinux

myfunctionapplinux

Function Apps

Visual Studio Enterprise

Search

Overview Platform features Quickstart **Quickstart**

Azure Functions - getting started

Follow our Quickstart guidance to author and publish a function [Learn more](#)

1 CHOOSE A DEVELOPMENT ENVIRONMENT 2 CREATE A FUNCTION

VS Code
Use Visual Studio Code to author your functions

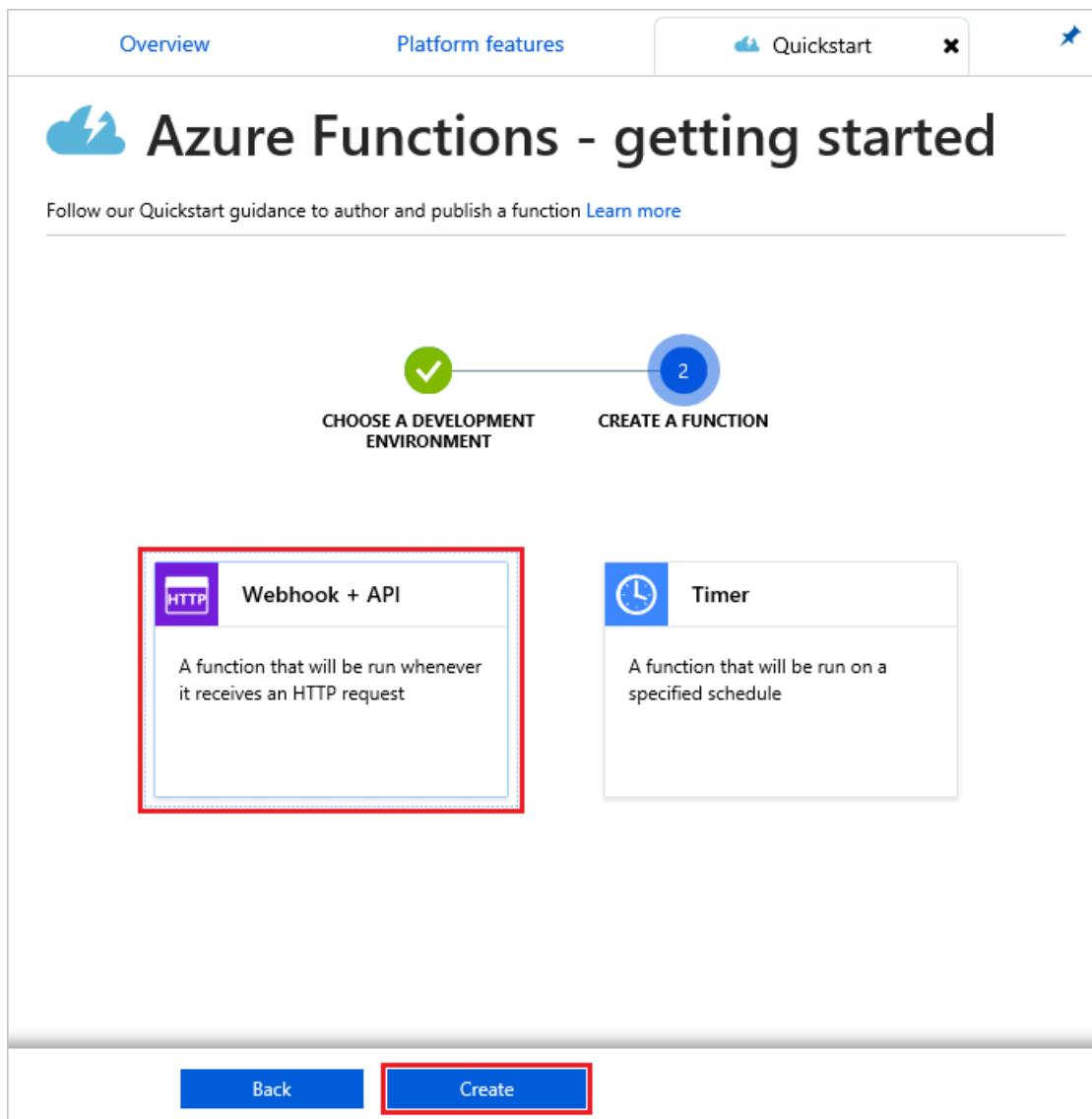
Any editor + Core Tools
Write functions using your favorite editor and the Azure Functions Core Tools

In-portal
Author functions quickly in the portal

Continue

The screenshot shows the Azure Functions Quickstart page. At the top right, there is a red box around the 'Quickstart' tab. Below it, the 'In-portal' section is also highlighted with a red box. A red box is also placed over the 'Continue' button at the bottom.

3. Choose **WebHook + API** and then select **Create**.

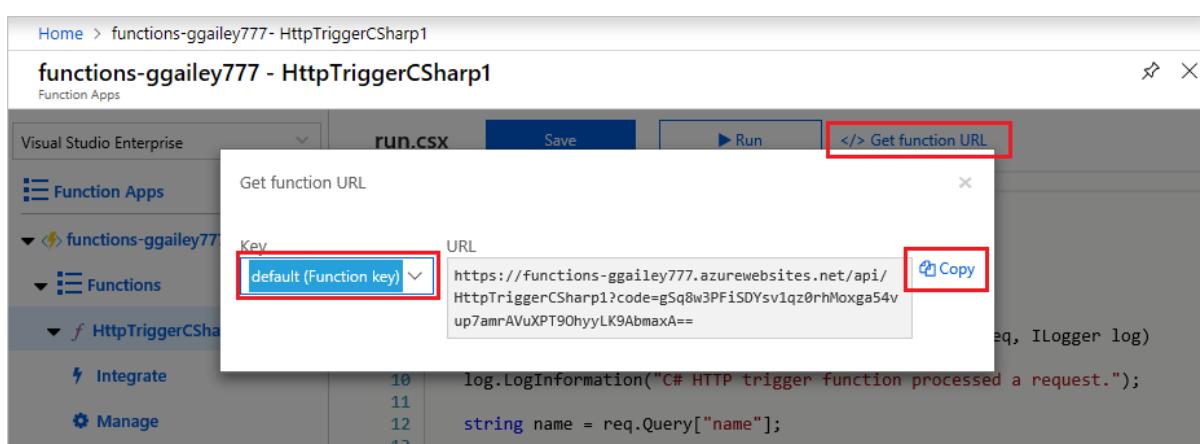


A function is created using a language-specific template for an HTTP triggered function.

Now, you can run the new function by sending an HTTP request.

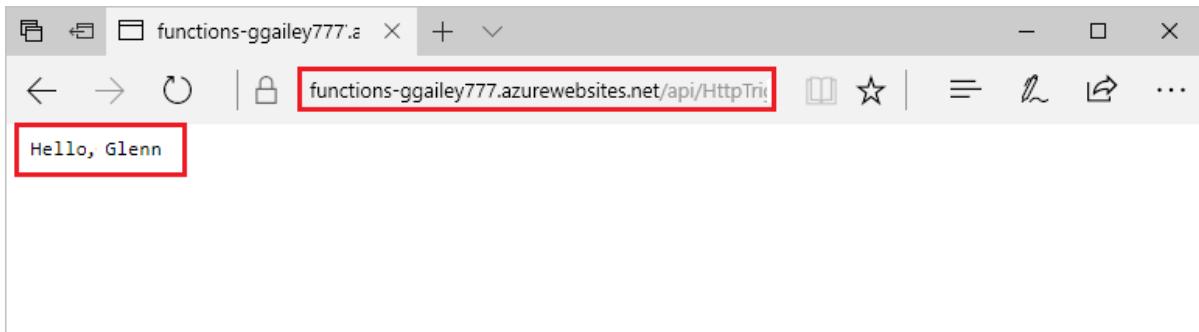
Test the function

1. In your new function, click **</> Get function URL** at the top right, select **default (Function key)**, and then click **Copy**.



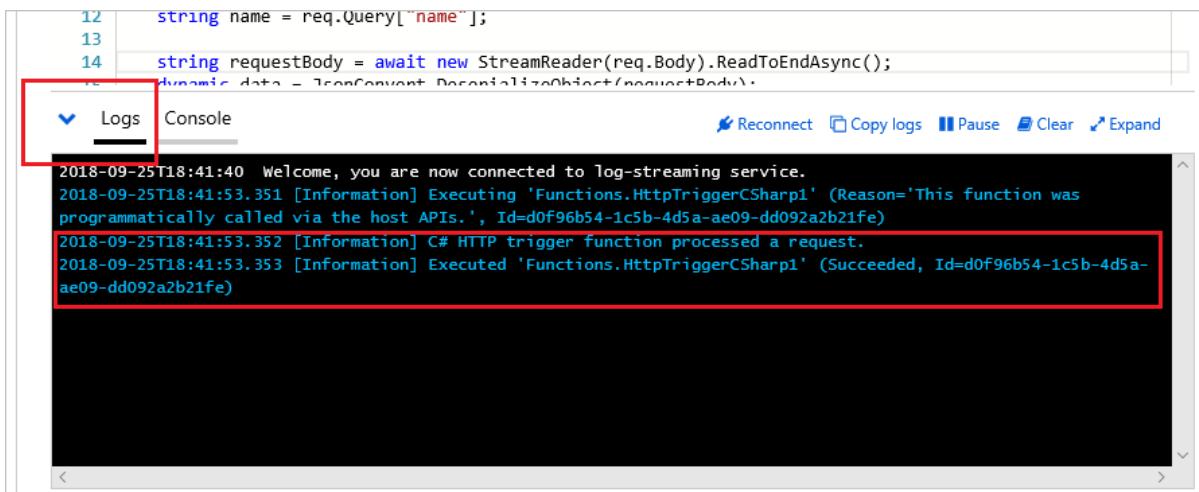
2. Paste the function URL into your browser's address bar. Add the query string value `&name=<yourusername>` to the end of this URL and press the **Enter** key on your keyboard to execute the request. You should see the response returned by the function displayed in the browser.

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.

3. When your function runs, trace information is written to the logs. To see the trace output from the previous execution, return to your function in the portal and click the arrow at the bottom of the screen to expand the **Logs**.



Clean up resources

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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 Microsoft Azure Functions overview page for a function app named 'functions-ggailey777'. The 'Overview' tab is selected. Key details shown include:

- Status: Running
- Subscription: Visual Studio Enterprise
- Resource group: functions-ggailey777 (highlighted with a red box)
- Location: South Central US
- URL: https://fun...
- App Service: SouthCent

The left sidebar shows 'Function Apps' with 'functions-ggailey777' expanded, showing 'Functions', 'Proxies (preview)', and 'Slots (preview)'.

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 are 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 triggered function.

Now that you have 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](#).

Debug PowerShell Azure Functions locally

7/9/2019 • 6 minutes to read • [Edit Online](#)

Azure Functions lets you develop your functions as PowerShell scripts.

NOTE

PowerShell for Azure Functions is currently in preview. To receive important updates, subscribe to the [Azure App Service announcements](#) repository on GitHub.

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.

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.

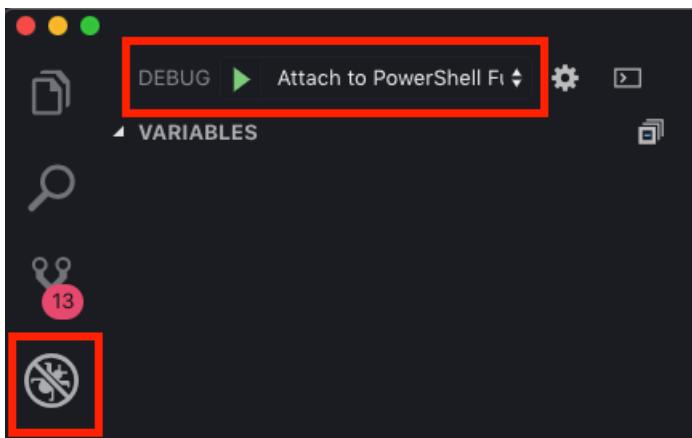
1. Press F1 to display the command pallet, then search for `Session`.
2. Choose **PowerShell: Show Session Menu**.
3. If your **Current session** isn't **PowerShell Core 6**, choose **Switch to: PowerShell Core 6**.

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.

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.

Next steps

To learn more about developing Functions using PowerShell, see [Azure Functions PowerShell developer guide](#).

Use dependency injection in .NET Azure Functions

7/22/2019 • 2 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.

Azure Functions builds on top of the ASP.NET Core Dependency Injection features. Being aware of services, lifetimes, and design patterns of [ASP.NET Core dependency injection](#) before using DI features in an Azure Functions app is recommended.

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
- Optional: [Microsoft.Extensions.Http](#) Only required for registering HttpClient at startup

Register services

To register services, you can 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. Also code is referencing a prerelease of [Microsoft.Azure.Cosmos](#) on Nuget.

```
using System;
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Http;
using Microsoft.Extensions.Logging;
using Microsoft.Azure.Cosmos;

[assembly: FunctionsStartup(typeof(MyNamespace.Startup))]

namespace MyNamespace
{
    public class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            builder.Services.AddHttpClient();
            builder.Services.AddSingleton(s => {
                return new CosmosClient(Environment.GetEnvironmentVariable("COSMOSDB_CONNECTIONSTRING"));
            });
            builder.Services.AddSingleton<ILoggerProvider, MyLoggerProvider>();
        }
    }
}
```

Use injected dependencies

ASP.NET Core uses constructor injection to make your dependencies available to your function. The following sample demonstrates how the `IMyService` and `HttpClient` dependencies are injected into an HTTP-triggered function.

```
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 MyNamespace
{
    public class HttpTrigger
    {
        private readonly IMyService _service;
        private readonly HttpClient _client;

        public HttpTrigger(IMyService service, IHttpClientFactory httpClientFactory)
        {
            _service = service;
            _client = httpClientFactory.CreateClient();
        }

        [FunctionName("GetPosts")]
        public async Task Get(
            [HttpTrigger(AuthorizationLevel.Function, "get", Route = "posts")] HttpRequest req,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");
            var res = await _client.GetAsync("https://microsoft.com");
            await _service.AddResponse(res);

            return new OkResult();
        }
    }
}
```

The use of constructor injection means that you should not use static functions if you want to take advantage of dependency injection.

Service lifetimes

Azure Functions apps provide the same service lifetimes as [ASP.NET Dependency Injection](#): transient, scoped, and singleton.

In a functions app, a 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. A 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 `SqlConnection`, `CloudBlobClient`, or `HttpClient` instances.

View or download a [sample of different service lifetimes](#) on GitHub.

Logging services

If you need your own logging provider, the recommended way is to register an `ILoggerProvider` instance. Application Insights is added by Azure Functions automatically.

WARNING

Do not add `AddApplicationInsightsTelemetry()` to the services collection as it registers services that conflict with services provided by the environment.

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>	Singleton	Runtime configuration
<code>Microsoft.Azure.WebJobs.Host.Executors.IJobHostEnvironment</code>	Singleton	Responsible for providing the ID of the host instance

If there are other services you want to take a dependency on, [create an issue and propose them on GitHub](#).

Overriding host services

Overriding services provided by the host is currently not supported. If there are services you want to override, [create an issue and propose them on GitHub](#).

Next steps

For more information, see the following resources:

- [How to monitor your function app](#)
- [Best practices for functions](#)

Manage connections in Azure Functions

5/17/2019 • 4 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
}

```

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 masterKey = process.env.COSMOS_API_KEY;
const { CosmosClient } = cosmos;

const client = new CosmosClient({ endpoint, auth: { masterKey } });
// All function invocations also reference the same database and container.
const container = client.database("MyDatabaseName").container("MyContainerName");

module.exports = async function (context) {
    const { result: itemArray } = await container.items.readAll().toArray();
    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

7/30/2019 • 2 minutes to read • [Edit Online](#)

This topic provides general guidance for handling errors that occur when your functions execute. It also provides links to the topics that describe binding-specific errors that may occur.

Handling errors in functions

Azure Functions [triggers and bindings](#) communicate with various Azure services. When integrating with these services, you may have errors raised that originate from the APIs of the underlying Azure services. Errors can also occur when you try to communicate with other services from your function code by using REST or client libraries. To avoid loss of data and ensure good behavior of your functions, it is important to handle errors from either source.

The following triggers have built-in retry support:

- [Azure Blob storage](#)
- [Azure Queue storage](#)
- [Azure Service Bus \(queue/topic\)](#)

By default, these triggers are retried up to five times. After the fifth retry, these triggers write a message to a special [poison queue](#).

For the other Functions triggers, there is no built-in retry when errors occur during function execution. To prevent loss of trigger information should an error occur in your function, we recommend that you use try-catch blocks in your function code to catch any errors. When an error occurs, write the information passed into the function by the trigger to a special "poison" message queue. This approach is the same one used by the [Blob storage trigger](#).

In this way, you can capture trigger events that could be lost due to errors and retry them at a later time using another function to process messages from the poison queue using the stored information.

Binding error codes

When integrating with Azure services, you may have errors raised that originate from the APIs of the underlying services. Links to the error code documentation for these services can be found in the **Exceptions and return codes** section of the following trigger and binding reference topics:

- [Azure Cosmos DB](#)
- [Blob storage](#)
- [Event Hubs](#)
- [Notification Hubs](#)
- [Queue storage](#)
- [Service Bus](#)
- [Table storage](#)

Manually run a non HTTP-triggered function

7/1/2019 • 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 to a way to send a request to Azure to run the function. The URL used to make this request takes a specific form.

`https://myfunctions.demos.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

Navigate to your function in the Azure portal and click on **Manage** and find the **Host Keys** section. Click on the **Copy** button in the `_master` row to copy the master key to your clipboard.

Microsoft Azure

Home > myfunctionsdemos - QueueTrigger

myfunctionsdemos - QueueTrigger

Function Apps

Search Microsoft Azure Internal Consumption

Function Apps

myfunctionsdemos

Functions

QueueTrigger

Integrate

Manage

Monitor

Proxies

Slots (preview)

Function State

Enabled Disabled Delete function

Host Keys (All functions)

NAME	VALUE	ACTIONS
_master	Click to show	Copy Renew
default	Click to show	Copy Renew Revoke

Add new host key

After copying the master key, click on the function name to return to the code file window. Next, click on the **Logs** tab. You'll see messages from the function logged here when you manually run the function from Postman.

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.

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. **Click on the Headers tab**.
4. Enter **x-functions-key** as the first **key** and paste the master key (from the clipboard) into the **value** box.
5. Enter **Content-Type** as the second **key** and enter **application/json** as the **value**.

The screenshot shows the Postman application interface. At the top, there's a toolbar with 'File', 'Edit', 'View', and 'Help' menus, and buttons for 'New', 'Import', 'Runner', and a plus sign. Below the toolbar, a header bar indicates a 'POST' method and the URL 'https://myfunctionsdemos.azurewebsites.net/admin/functions/QueueTrigger'. A red box highlights the URL field. To the right are buttons for 'Send' (highlighted) and 'Save'. Below the header, tabs for 'Params', 'Authorization', 'Headers (2)', 'Body', 'Pre-request Script', 'Tests', 'Cookies', and 'Code' are visible. The 'Headers (2)' tab is selected and highlighted with a red box. Under 'Headers', two entries are listed: 'x-functions-key' and 'Content-Type', both with checked checkboxes. A red box highlights these two rows. Below the headers, a section labeled 'Response' contains the placeholder text 'Hit the Send button to get a response.' At the bottom, there are various icons for file operations like 'New', 'Import', 'Run', and 'Help'.

6. Click on the **Body** tab.

7. Enter `{ "input": "test" }` as the body for the request.

This screenshot shows the same Postman interface as the previous one, but with the 'Body' tab selected, indicated by a red box around the tab label. Below the tab, there are options for 'form-data', 'x-www-form-urlencoded', 'raw' (which is selected), 'binary', and 'JSON (application/json)' (also selected). A red box highlights the JSON body entry field, which contains the JSON string `{ "input": "test" }`. The rest of the interface is identical to the first screenshot.

8. Click **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 '+' icon. Below the toolbar, the URL 'https://myfunctionsdemos.azurewebsites.net/admin/functions/QueueTrigger' is entered. The method dropdown shows 'POST'. On the right, there's a 'Send' button highlighted with a red box, and a 'Save' button. The 'Headers' tab is selected, showing two entries: 'x-functions-key' and 'Content-Type'. Below the headers, the status bar indicates 'Status: 202 Accepted'. The 'Headers' tab also lists several response headers: Cache-Control (no-cache), Pragma (no-cache), Expires (-1), Server (Microsoft-IIS/10.0), X-Powered-By (ASP.NET), Date (Wed, 05 Dec 2018 15:22:32 GMT), and Content-Length (0). At the bottom, there are icons for file operations and help.

Postman then reports a status of **202 Accepted**.

Next, return to your function in the Azure portal. Locate the *Logs* window and you'll see messages coming from the manual call to the function.

The screenshot shows the Azure Functions 'Logs' window. At the top, there's a 'run.csx' tab, a 'Save' button, and a 'Run' button. Below that is a code editor with the following C# code:

```
1 using System;
2
3 public static void Run(string myQueueItem, TraceWriter log)
4 {
5     log.Info($"C# Queue trigger function processed: {myQueueItem}");
6 }
```

Below the code editor, there are tabs for 'Logs', 'Errors and warnings', and 'Console'. The 'Logs' tab is selected. It displays the following log output:

```
2018-12-07T20:31:02 Welcome, you are now connected to log-streaming service.
2018-12-07T20:31:37.278 [Info] Function started (Id=9b80172f-a464-470c-bb85-a722dd67d563)
2018-12-07T20:31:37.294 [Info] C# Queue trigger function processed: test
2018-12-07T20:31:37.294 [Info] Function completed (Success, Id=9b80172f-a464-470c-bb85-a722dd67d563, Duration=10ms)
```

At the bottom of the window, there are buttons for 'Reconnect', 'Copy logs', 'Pause', 'Clear', and 'Expand'.

Next steps

- Strategies for testing your code in Azure Functions
- Azure Function Event Grid Trigger Local Debugging

Azure Function Event Grid Trigger Local Debugging

7/1/2019 • 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.

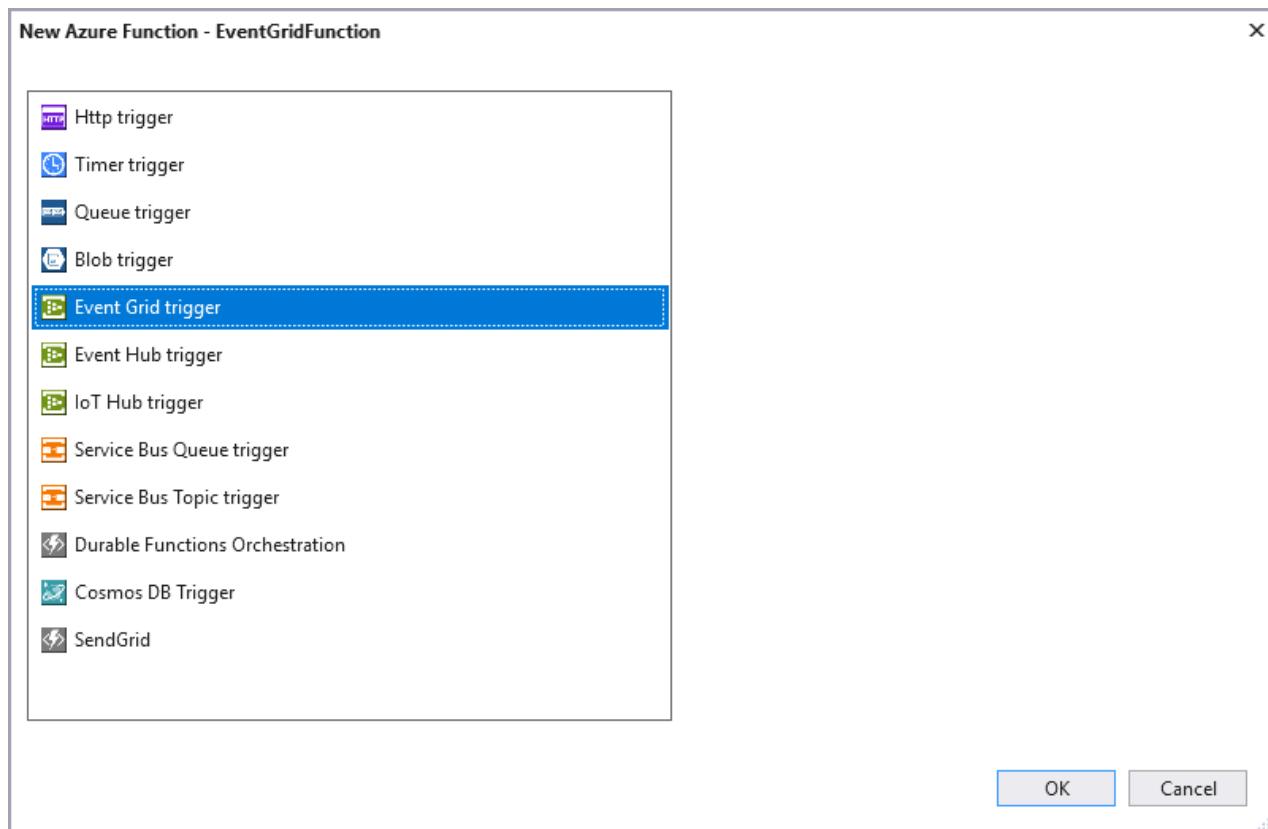
Prerequisites

- 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 URL 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`.



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:

```
cmd Select Command Prompt - ngrok http -host-headers=localhost 7071
ngrok by @inconshreveable                                         (Ctrl+C to quit)

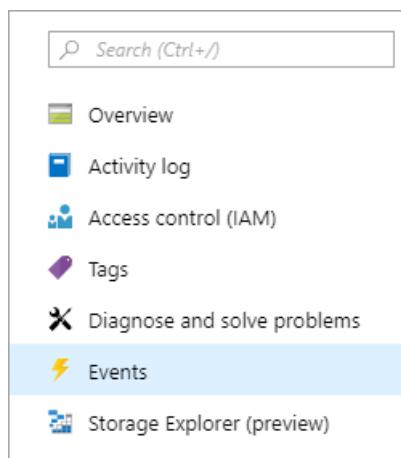
Session Status          online
Account                  (Plan: Free)
Version                 2.2.8
Region                  United States (us)
Web Interface           http://127.0.0.1:4040
Forwarding              http://9e14d0df.ngrok.io -> localhost:7071
Forwarding              https://9e14d0df.ngrok.io -> localhost:7071

Connections             ttl     opn      rt1      rt5      p50      p90
                        0       0       0.00    0.00    0.00    0.00
```

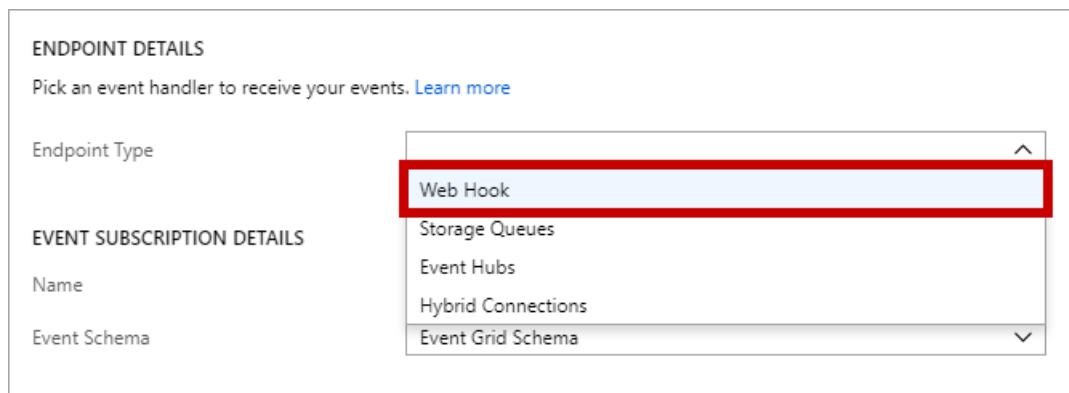
Copy the **HTTPS** URL generated when *ngrok* is run. This value is used when configuring the event grid event endpoint.

Add a storage event

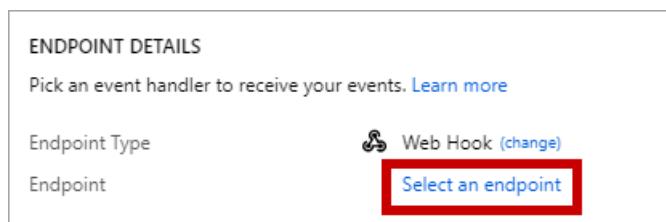
Open the Azure portal and navigate to a storage account and click on the **Events** option.



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**.

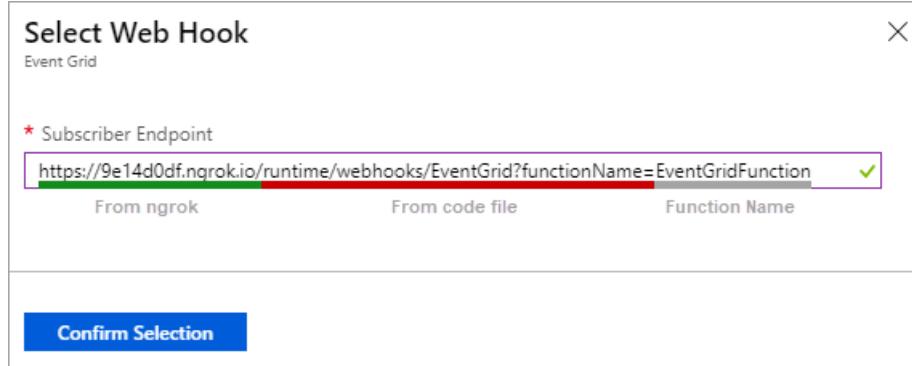


Once the endpoint type is configured, click on **Select an endpoint** to configure the endpoint value.



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](#)

Create a Function using Azure for Students Starter

5/6/2019 • 5 minutes to read • [Edit Online](#)

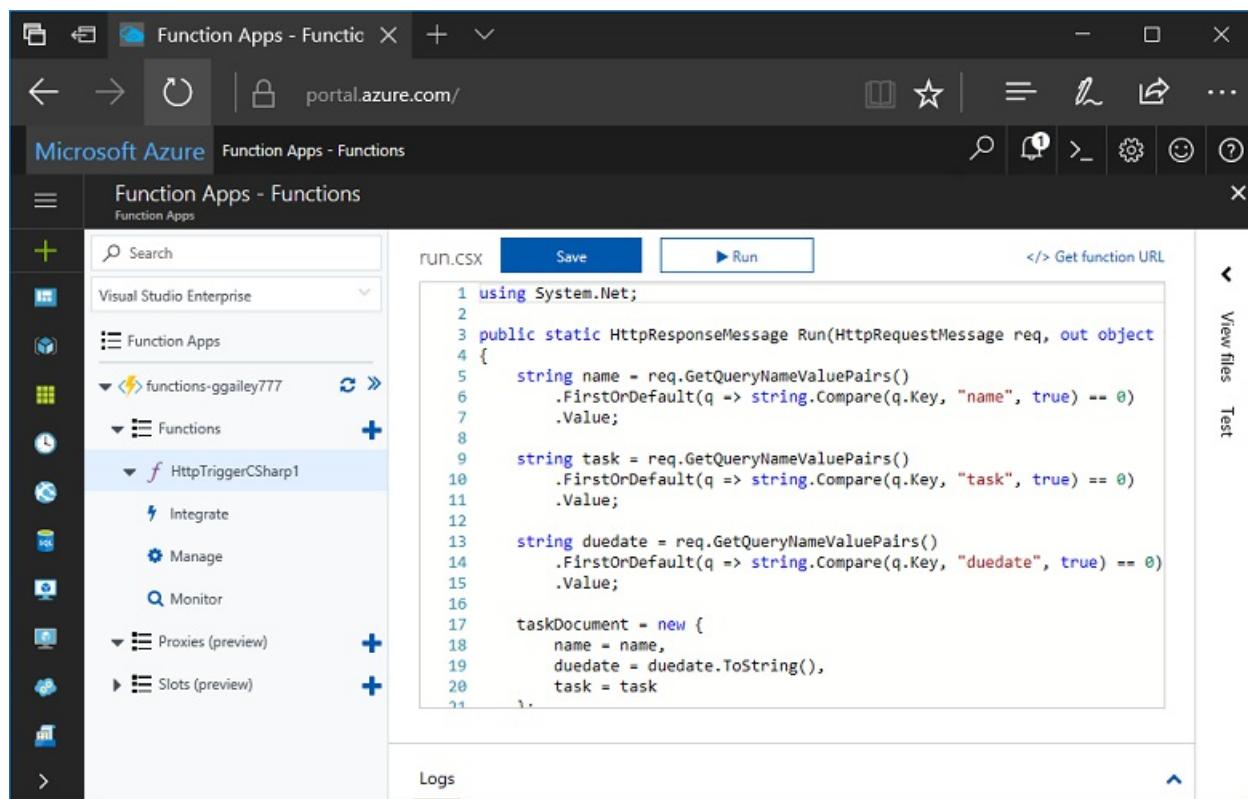
In this tutorial, we will create a hello world HTTP function in an Azure for Students Starter subscription. We'll also walk through what's available in Azure Functions in this subscription type.

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.](#)

Create a Function

In this topic, learn how to use Functions to create an HTTP triggered "hello world" function in the Azure portal.



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. A function app lets you group functions as a logic unit for easier management, deployment, and sharing of resources.

1. Select the **New** button found on the upper left-hand corner of the Azure portal, then select **Compute > Function App**.

Microsoft Azure

Home > New

Create a resource

All services

Favorites

- Dashboard
- All resources
- Resource groups
- App Services
- SQL databases
- SQL data warehouses
- Azure Cosmos DB
- Virtual machines
- Load balancers
- Storage accounts
- Virtual networks
- Azure Active Directory
- Monitor
- Advisor
- Security Center
- Cost Management + Billing

Search the Marketplace

Azure Marketplace

Get started

Recently created

Compute

Networking

Storage

Web + Mobile

Containers

Databases

Data + Analytics

AI + Cognitive Services

Internet of Things

Enterprise Integration

Security + Identity

Developer tools

Monitoring + Management

Add-ons

Blockchain

Featured

Windows Server 2016 Datacenter
Quickstart tutorial

Red Hat Enterprise Linux 7.2
Learn more

Ubuntu Server 16.04 LTS
Quickstart tutorial

SQL Server 2017 Enterprise Windows Server 2016
Learn more

Reserved VM Instances
Learn more

Service Fabric Cluster
Learn more

Web App for Containers
Learn more

Function App
Quickstart tutorial

Batch Service
Learn more

The screenshot shows the Microsoft Azure 'New' blade. On the left is a sidebar with various service icons and names. At the top, there's a red box around the 'Create a resource' button. Below it, another red box highlights the 'Compute' category under the 'Recently created' section of the Azure Marketplace. To the right of 'Compute', other categories like Networking, Storage, and Web + Mobile are listed. Further down, a list of featured services is shown, with 'Function App' highlighted by a red box.

2. Use the function app settings as specified in the table below the image.

Function App

Create

* App name
alex-function .azurewebsites.net

* Subscription
DreamSpark

* Resource Group i
 Create new Use existing
alex-function

* App Service plan/Location
ServicePland1a13fb3-a088(Centr...)

* Runtime Stack
.NET

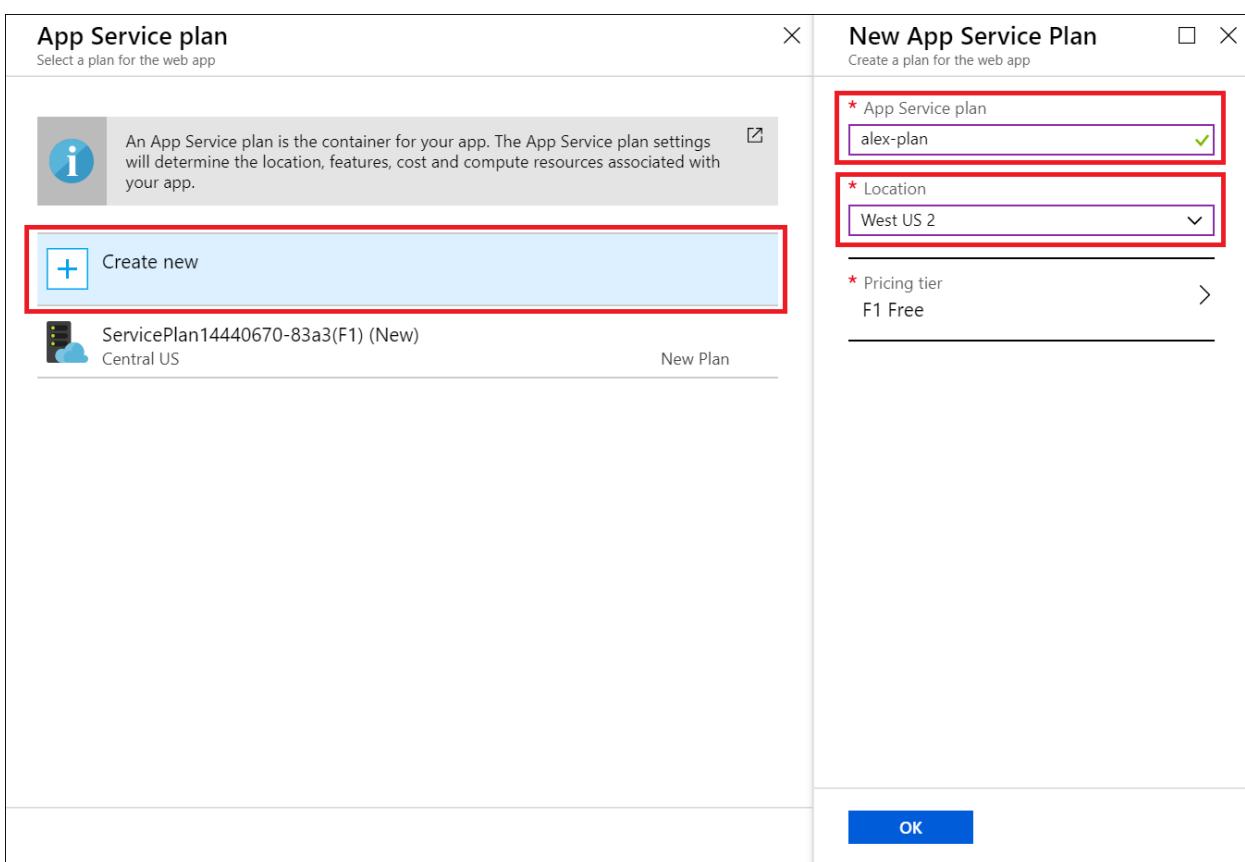
Application Insights
Disabled

Create [Automation options](#)

SETTING	SUGGESTED VALUE	DESCRIPTION
App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Your subscription	The subscription under which this new function app is created.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
App Service Plan/Location	New	The hosting plan that controls what region your function app is deployed to and the density of your resources. Multiple Function Apps deployed to the same plan will all share the same single free instance. This is a restriction of the Student Starter plan. The full hosting options are explained here .
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. Choose .NET for C# and F# functions.

SETTING	SUGGESTED VALUE	DESCRIPTION
Application Insights	Enabled	Application Insights is used to store and analyze your function app's logs. It is enabled by default if you choose a location that supports Application Insights. Application Insights can be enabled for any function by manually choosing a nearby region to deploy Application Insights. Without Application Insights, you will only be able to view live streaming logs.

3. Select **App Service plan/Location** above to choose a different location
4. Select **Create new** and then give your plan a unique name.
5. Select the location closest to you. [See a full map of Azure regions here.](#)



App Service plan
Select a plan for the web app

An App Service plan is the container for your app. The App Service plan settings will determine the location, features, cost and compute resources associated with your app.

Create new

ServicePlan14440670-83a3(F1) (New)
Central US

New Plan

New App Service Plan
Create a plan for the web app

*

App Service plan

✓

*

Location

✓

*

Pricing tier

F1 Free

OK

6. Select **Create** to provision and deploy the function app.

Function App

Create

* App name
alex-function .azurewebsites.net

* Subscription
DreamSpark

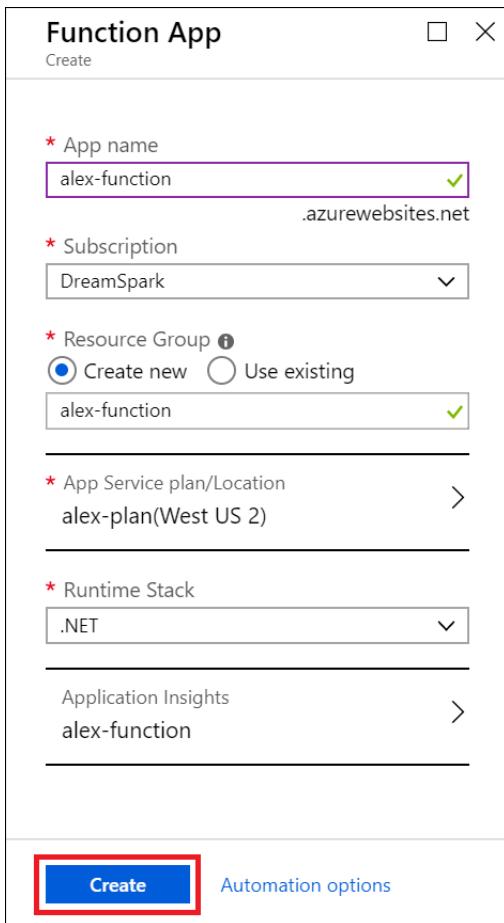
* Resource Group i
 Create new Use existing
alex-function

* App Service plan/Location
alex-plan(West US 2)

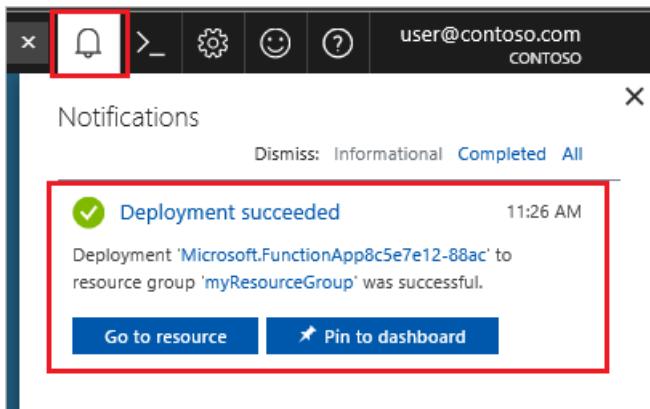
* Runtime Stack
.NET

Application Insights
alex-function

Create Automation options



7. Select the Notification icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.



8. Select **Go to resource** to view your new function app.

Next, you create a function in the new function app.

Create an HTTP triggered function

1. Expand your new function app, then select the + button next to **Functions**, choose **In-portal**, and select **Continue**.

Microsoft Azure

Home > functions-ggailey777

functions-ggailey777

Function Apps

Visual Studio Enterprise

Search

Function Apps

functions-ggailey777

+ Functions

+ Proxies

+ Slots (preview)

Overview Platform features Quickstart

Azure Functions - getting started

Follow our Quickstart guidance to author and publish a function [Learn more](#)

1 CHOOSE A DEVELOPMENT ENVIRONMENT

2 CREATE A FUNCTION

In-portal Author functions quickly in the portal

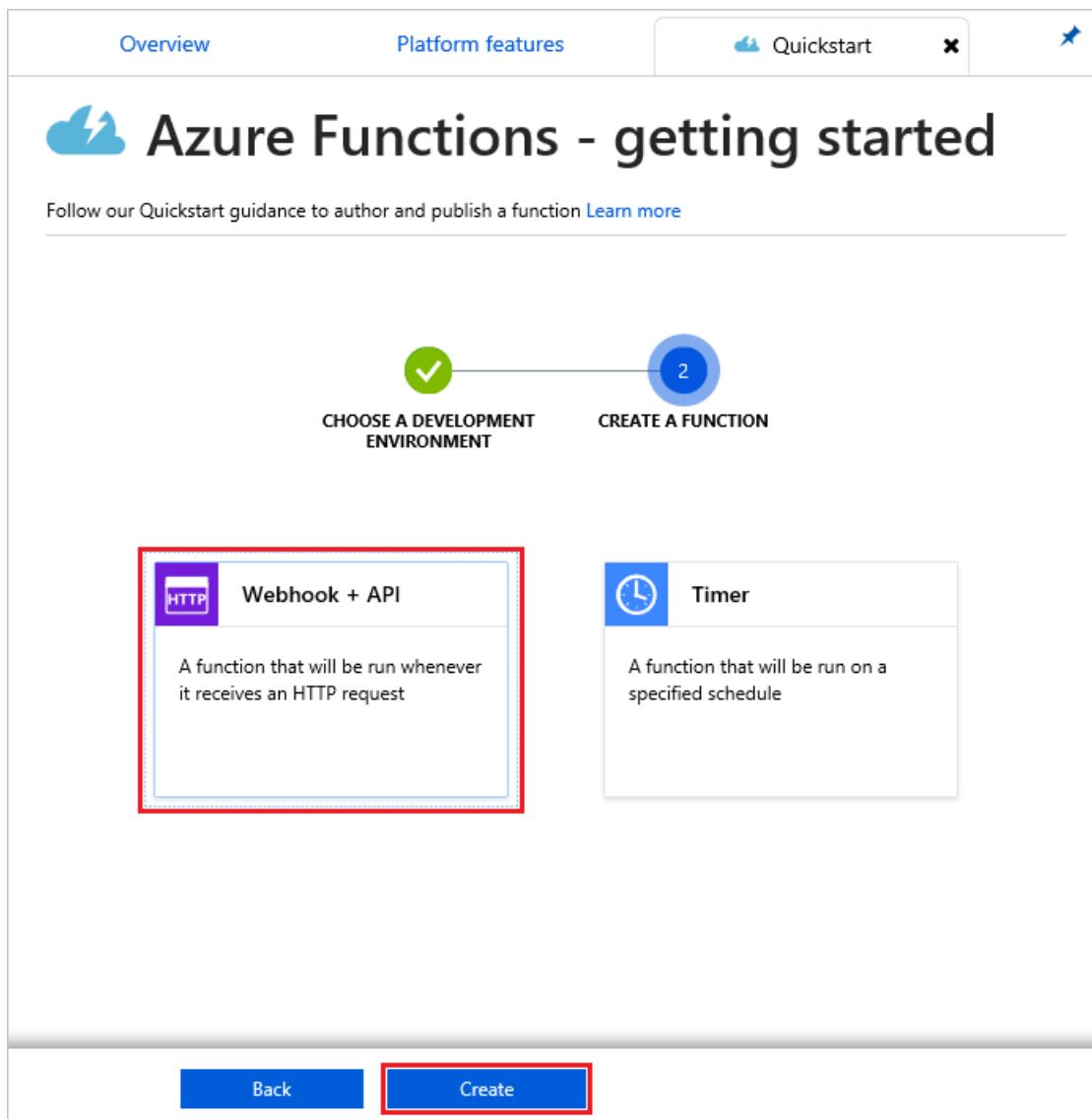
VS Code Use Visual Studio Code to author your functions

Any editor + Core Tools Write functions using your favorite editor and the Azure Functions Core Tools

Continue

The screenshot shows the Microsoft Azure Functions 'getting started' quickstart page. The left sidebar displays the 'functions-ggailey777' app under 'Function Apps'. The main content area is titled 'Azure Functions - getting started' and provides guidance for authoring and publishing functions. It includes two numbered steps: 'CHOOSE A DEVELOPMENT ENVIRONMENT' and 'CREATE A FUNCTION'. Step 1 offers three options: 'In-portal' (selected and highlighted with a red box), 'VS Code', and 'Any editor + Core Tools'. A red box also highlights the 'Continue' button at the bottom of the step 1 section.

- Choose **WebHook + API** and then select **Create**.
- Choose **WebHook + API** and then select **Create**.

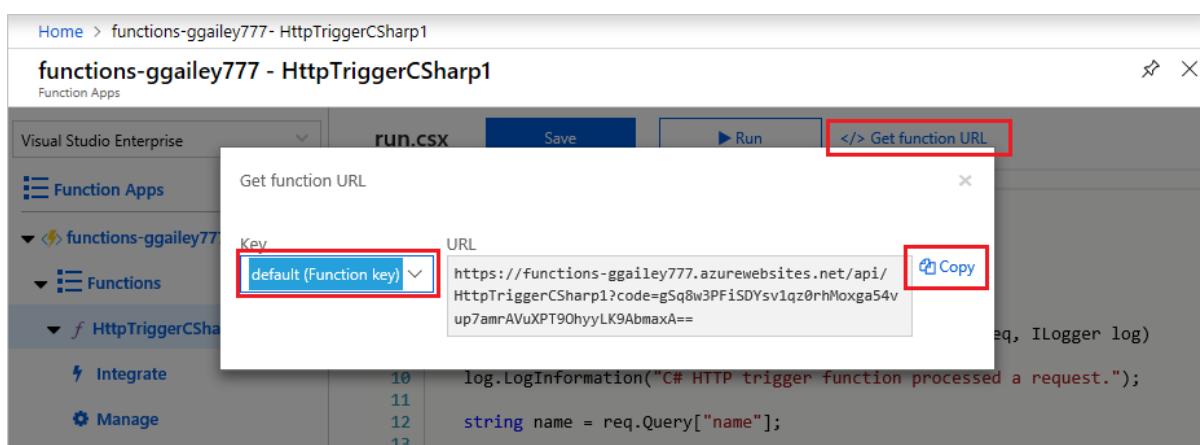


A function is created using a language-specific template for an HTTP triggered function.

Now, you can run the new function by sending an HTTP request.

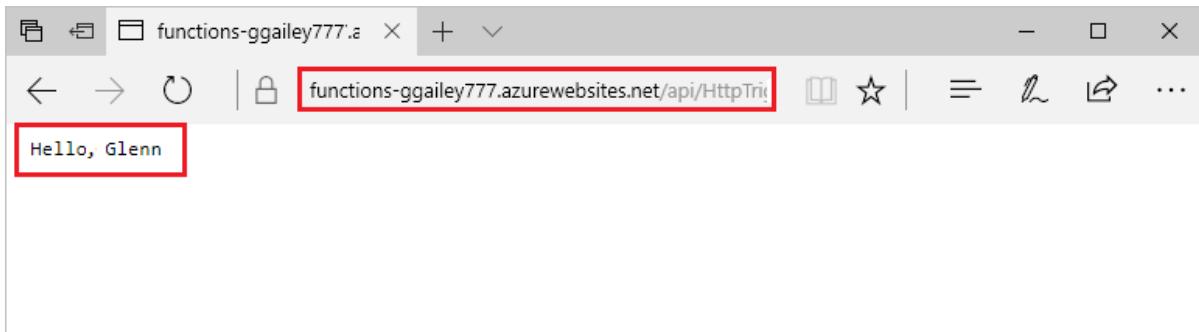
Test the function

1. In your new function, click **</> Get function URL** at the top right, select **default (Function key)**, and then click **Copy**.



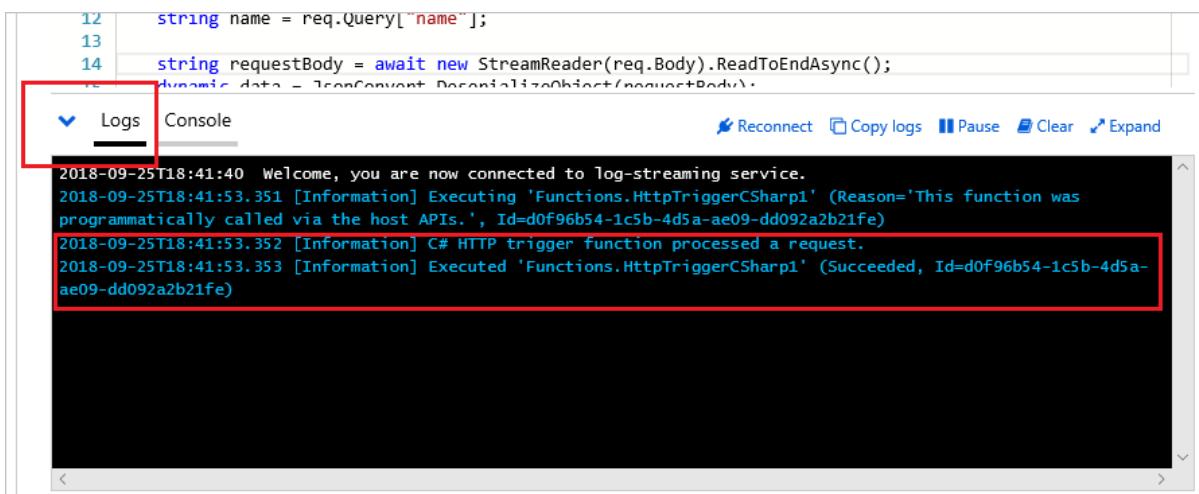
2. Paste the function URL into your browser's address bar. Add the query string value `&name=<yourusername>` to the end of this URL and press the **Enter** key on your keyboard to execute the request. You should see the response returned by the function displayed in the browser.

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.

3. When your function runs, trace information is written to the logs. To see the trace output from the previous execution, return to your function in the portal and click the arrow at the bottom of the screen to expand the **Logs**.



Clean up resources

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

Resources in Azure refers to function apps, functions, storage accounts, and so forth. They are 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 Microsoft Azure Functions overview page. At the top, there's a navigation bar with icons for search, notifications, and settings. Below that is a header with the text 'functions-ggailey777' and 'Function Apps'. On the left, there's a sidebar with various icons and a list of function apps, including 'functions-ggailey777'. The main area has tabs for 'Overview', 'Settings', 'Platform features', and 'API definition (preview)'. Under 'Overview', there are sections for 'Status' (Running), 'Subscription' (Visual Studio Enterprise), 'Resource group' (functions-ggailey777), 'Subscription ID', 'Location' (South Central US), and 'App Service' (SouthCent). At the bottom, there's a section titled 'Configured features' with a note: 'Quick links to your features will show up here after'. A red box highlights the 'Resource group' field, and another red box highlights the 'Functions' section under 'Configured features'.

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 are 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 Student 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 will serverlessly scale from 0 to 1 instance automatically as HTTP traffic is received, but no further.
- Only the [2.x 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 will 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 have created a function app with a simple HTTP triggered function! Now 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](#)

Continuous deployment for Azure Functions

7/5/2019 • 3 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 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](#)

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 **Platform features > Deployment Center**.

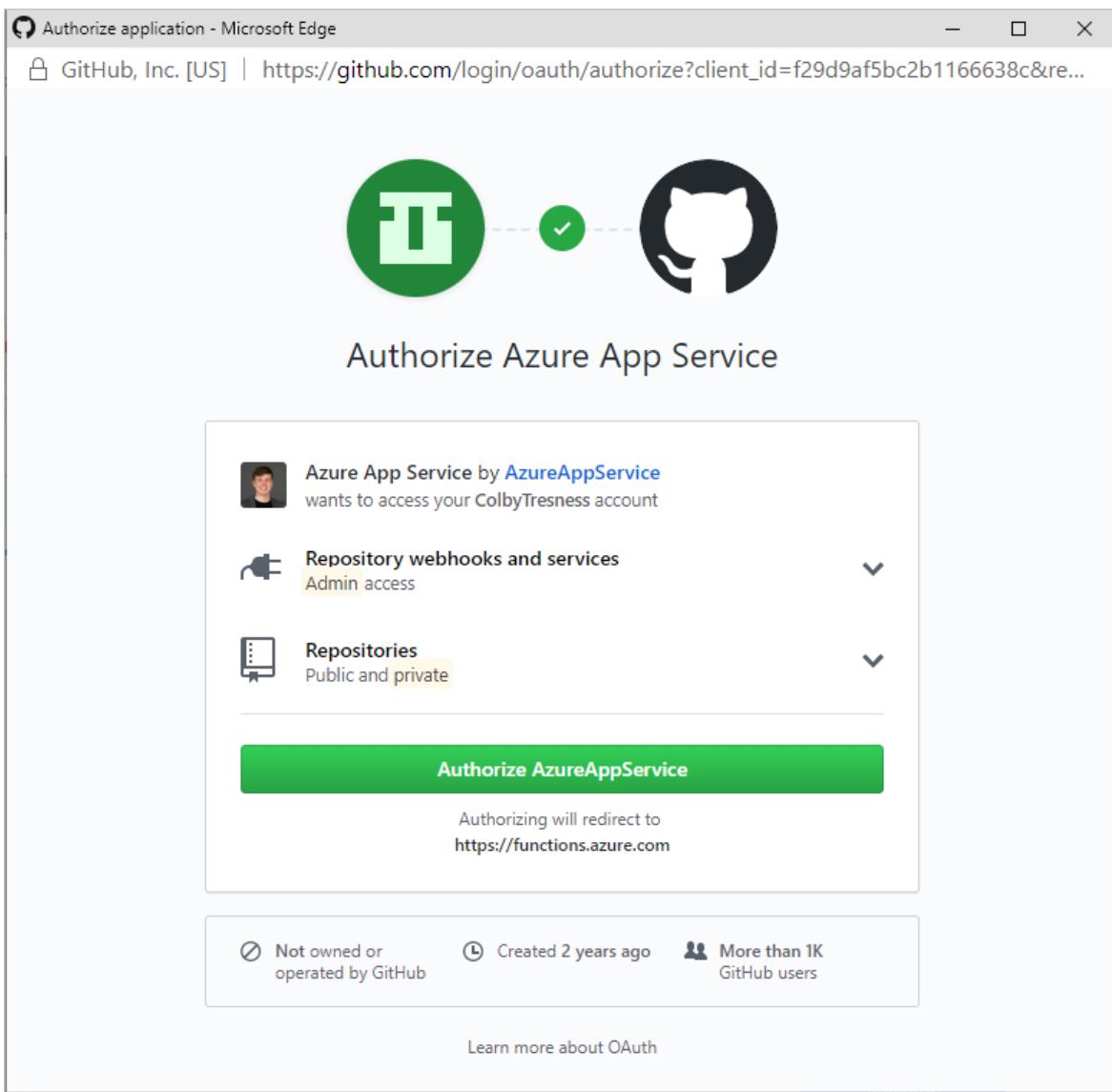
The screenshot shows the Azure portal interface for a function app named "colby-docs". The left sidebar shows "Function Apps" with "colby-docs" selected. The main content area has a red box around the "Platform features" tab, which is currently active. Below it, under "Code Deployment", the "Deployment Center" link is also highlighted with a red box.

2. In **Deployment Center**, select **GitHub**, and then select **Authorize**. If you've already authorized GitHub, select **Continue**.

The screenshot shows the "Deployment Center" page. At the top, there are four numbered steps: 1. SOURCE CONTROL, 2. BUILD PROVIDER, 3. CONFIGURE, and 4. SUMMARY. Step 1 is highlighted with a blue circle. Below the steps, there are eight options arranged in two rows of four. The "GitHub" option is highlighted with a red box. At the bottom of the page, there is a large red box highlighting the "Authorize" button.

Source Control Provider	Build Provider	Configure Options	Summary
Azure Repos	Github	Bitbucket	Local Git
OneDrive	Dropbox	External	FTP

3. In GitHub, select the **Authorize AzureAppService** button.



In **Deployment Center** in the Azure portal, select **Continue**.

4. 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.

A screenshot of the Azure Deployment Center interface. The top navigation bar shows "Overview", "Platform features", and "Deployment Center". The main area has a progress bar with four steps: "SOURCE CONTROL" (green checkmark), "BUILD PROVIDER" (blue circle with number 2), "CONFIGURE" (grey circle with number 3), and "SUMMARY" (grey circle with number 4). Below the progress bar, there are two options: "App Service build service" (selected) and "Azure Pipelines (Preview)". The "App Service build service" section says: "Use App Service as the build server. The App Service Kudu engine will automatically build your code during deployment when applicable with no additional configuration required." The "Azure Pipelines (Preview)" section says: "Configure a robust deployment pipeline for your application using Azure Pipelines, part of Azure DevOps Services (formerly known as VSTS). The pipeline builds, runs load tests and deploys to...". At the bottom, there are "Back" and "Continue" buttons.

5. 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**.

SOURCE CONTROL

BUILD PROVIDER

CONFIGURE

SUMMARY

Code

Organization: ColbyTresness

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

6. Review all details, and then select **Finish** to complete your deployment configuration.

SOURCE CONTROL

Repository: https://github.com/ColbyTresness/functions-typescript-intermediate

Branch: master

BUILD PROVIDER

Provider: App Service build service

Back Finish

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.

Deployment scenarios

Move existing functions to continuous deployment

If you've already written functions in the [Azure portal](#) and you want to download the contents of your app before you switch to continuous deployment, go to the **Overview** tab of your function app. Select the

Download app content button.

The screenshot shows the Azure Functions Overview page. On the left, there's a sidebar with 'Function Apps' and a list of functions: 'my-function-app' (Running), 'HttpTrigger1' (Integrate, Manage, Monitor), 'Proxies', and 'Slots (preview)'. The main area has tabs for 'Overview' (highlighted with a red box) and 'Platform features'. Below these are sections for 'Status' (Running), 'Subscription' (My Subscription), 'Resource group' (my-resource-group), and 'URL' (https://my-function-app.azurewebsites.net). There are also sections for 'Subscription ID', 'Location' (Central US), and 'App Service plan / pricing tier' (CentralUSPlan (Consumption)). At the bottom, there's a section for 'Configured features' with links to 'Function app settings', 'Configuration', and 'Application Insights'. A red box highlights the 'Download app content' button in the top right.

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](#)

Continuous delivery by using Azure DevOps

7/5/2019 • 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.

.NET

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: true  
    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'  
    name: 'drop'
```

JavaScript

You can use the following sample to create a YAML file to build a JavaScript app:

```

pool:
  vmImage: ubuntu-16.04 # Use 'VS2017-Win2016' if you have Windows native +Node modules
steps:
- bash: |
  if [ -f extensions.csproj ]
  then
    dotnet build extensions.csproj --output ./bin
  fi
  npm install
  npm run build --if-present
  npm prune --production
- task: ArchiveFiles@2
  displayName: "Archive files"
  inputs:
    rootFolderOrFile: "$(System.DefaultWorkingDirectory)"
    includeRootFolder: false
    archiveFile: "$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip"
- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish: '$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip'
    name: 'drop'

```

Python

You can use the following sample to create a YAML file to build a Python app. Python is supported only for Linux Azure Functions.

```

pool:
  vmImage: ubuntu-16.04
steps:
- task: UsePythonVersion@0
  displayName: "Setting python version to 3.6 as required by functions"
  inputs:
    versionSpec: '3.6'
    architecture: 'x64'
- bash: |
  if [ -f extensions.csproj ]
  then
    dotnet build extensions.csproj --output ./bin
  fi
  python3.6 -m venv worker_venv
  source worker_venv/bin/activate
  pip3.6 install setuptools
  pip3.6 install -r requirements.txt
- task: ArchiveFiles@2
  displayName: "Archive files"
  inputs:
    rootFolderOrFile: "$(System.DefaultWorkingDirectory)"
    includeRootFolder: false
    archiveFile: "$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip"
- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish: '$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip'
    name: 'drop'

```

PowerShell

You can use the following sample to create a YAML file to package a PowerShell app. PowerShell is supported only for Windows Azure Functions.

```

pool:
  vmImage: 'VS2017-Win2016'
steps:
- task: ArchiveFiles@2
  displayName: "Archive files"
  inputs:
    rootFolderOrFile: "$(System.DefaultWorkingDirectory)"
    includeRootFolder: false
    archiveFile: "$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip"
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(System.DefaultWorkingDirectory)/build$(Build.BuildId).zip'
    name: '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' (which contains 'Build extensions', 'Use Node version 10.14.1', 'Install Application Dependencies', 'Run 'build' script', 'Remove extraneous packages', and 'Archive files'), and 'Publish Artifact: drop'. On the right, the 'Archive files' task is being configured. It has fields for 'Root folder or file to archive' set to '\$(System.DefaultWorkingDirectory)', 'Archive type' set to 'zip', and 'Archive file to create' set to '\$(Build.ArtifactStagingDirectory)/\$(Build.BuildId).zip'. A checkbox labeled 'Prepend root folder name to archive paths' is checked and highlighted with a red box. Other options like 'Replace existing archive', 'Force verbose output', and 'Force quiet output' are also present.

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 with the same pipeline structure as the previous one. The 'Agent job 1' section is expanded, showing its internal tasks. The 'Agent pool' dropdown is open, displaying a list of available pools: 'Hosted', 'Hosted macOS', 'Hosted macOS High Sierra', 'Hosted Ubuntu 1604', 'Hosted VS2017' (which is highlighted with a red box), 'Hosted Windows 2019 with VS2019', and 'Hosted Windows Container'. There are also sections for 'Private' and 'Default (No agents)'.

Deploy your app

When you create a new release pipeline, search for the Azure Functions release template.

The screenshot shows the 'Select a template' dialog in Azure DevOps. At the top, there's a search bar with the text 'Azure Functions'. Below it, there are two main sections: 'Select a template' and 'Others'. In the 'Select a template' section, there's a card for 'Deploy a function app to Azure Functions' which includes a lightning bolt icon and the text 'Deploy your serverless application to Azure Function App.' This card is highlighted with a red box. In the 'Others' section, there's a link to 'Empty job'.

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](#).

Zip deployment for Azure Functions

3/15/2019 • 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 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 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](#)

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**.

Status	Subscription	Resource group	URL
Running	Visual Studio Enterprise	functions-ggailey777	https://functions-ggailey777.azurewebsites.net
	Subscription ID XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX	Location South Central US	App Service plan / pricing tier SouthCentralUSPlan (Consumption)

Configured features

- [Function app settings](#)
- [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.

```
az functionapp deployment source config-zip -g myResourceGroup -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 `<username>`, `<password>`, `<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 `Invoke-RestMethod` to send a request that contains the .zip file. Replace the placeholders `<deployment_user>`, `<deployment_password>`, `<zip_file_path>`, and `<app_name>`.

```
#PowerShell
$username = "<deployment_user>"
$password = "<deployment_password>"
$filePath = "<zip_file_path>"
$apiUrl = "https://<app_name>.scm.azurewebsites.net/api/zipdeploy"
$base64AuthInfo = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{}:{}" -f $username, $password)))
$userAgent = "powershell/1.0"
Invoke-RestMethod -Uri $apiUrl -Headers @{Authorization=("Basic {0}" -f $base64AuthInfo)} -UserAgent $userAgent -Method POST -InFile $filePath -ContentType "multipart/form-data"
```

This request triggers push deployment from the uploaded .zip file. To review the current and past deployments, run the following commands. Again, replace the <app_name> placeholder.

```
$apiUrl = "https://<app_name>.scm.azurewebsites.net/api/deployments"
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

7/24/2019 • 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.

IMPORTANT

When deploying your functions to a Linux function app in a [Premium plan](#), you should always run from the package file and [publish your app using the Azure Functions Core Tools](#).

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.

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. The function app is then run from the package after a restart, and `wwwroot` becomes read-only. For more information about zip deployment, see [Zip deployment for Azure Functions](#).

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`).

Next steps

[Continuous deployment for Azure Functions](#)

Automate resource deployment for your function app in Azure Functions

4/26/2019 • 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](#)

NOTE

The Premium plan for Azure Functions hosting is currently in preview. For more information, see [Azure Functions Premium 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](#) (in preview) 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": "2018-07-01",
  "location": "[resourceGroup().location]",
  "kind": "StorageV2",
  "properties": {
    "accountType": "[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'),'2015-05-01-preview').key1)]"
  },
  {
    "name": "AzureWebJobsDashboard",
    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'),
';AccountKey=', listKeys(variables('storageAccountid'),'2015-05-01-preview').key1)]"
  }
]
```

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('functionAppName')]"
  }
},
```

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](#) (in preview)
- [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	<code>~2</code>
FUNCTIONS_WORKER_RUNTIME	The language stack to be used for functions in this app	<code>dotnet</code> , <code>node</code> , <code>java</code> , or <code>python</code>
WEBSITE_NODE_DEFAULT_VERSION	Only needed if using the <code>node</code> language stack, specifies the version to use	<code>10.14.1</code>

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'),'2015-05-01-
preview').key1)]"
            },
            {
                "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 down 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": "2015-04-01",
    "name": "[variables('hostingPlanName')]",
    "location": "[resourceGroup().location]",
    "properties": {
        "name": "[variables('hostingPlanName')]",
        "computeMode": "Dynamic",
        "sku": "Dynamic"
    }
}
```

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'),'2015-05-01-  
preview').key1)]"  
                },  
                {  
                    "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",  
                    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',  
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'),'2015-05-01-  
preview').key1)]"  
                },  
                {  
                    "name": "WEBSITE_CONTENTSHARE",  
                    "value": "[toLowerCase(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'), '2015-05-01-preview').key1)]"
        },
        {
          "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 \(Preview\)](#).

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 `sku` property value.

```
{
  "type": "Microsoft.Web/serverfarms",
  "apiVersion": "2015-04-01",
  "name": "[variables('hostingPlanName')]",
  "location": "[resourceGroup().location]",
  "properties": {
    "name": "[variables('hostingPlanName')]",
    "sku": "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'), '2015-05-01-  
preview').key1)]"  
                },  
                {  
                    "name": "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING",  
                    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',  
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'), '2015-05-01-  
preview').key1)]"  
                },  
                {  
                    "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": "2015-04-01",  
    "name": "[variables('hostingPlanName')]",  
    "location": "[resourceGroup().location]",  
    "properties": {  
        "name": "[variables('hostingPlanName')]",  
        "sku": "[parameters('sku')]",  
        "workerSize": "[parameters('workerSize')]",  
        "hostingEnvironment": "",  
        "numberOfWorkers": 1  
    }  
}
```

To run your app on Linux, you must also set the `kind` to `Linux`:

```
{  
    "type": "Microsoft.Web/serverfarms",  
    "apiVersion": "2015-04-01",  
    "name": "[variables('hostingPlanName')]",  
    "location": "[resourceGroup().location]",  
    "kind": "Linux",  
    "properties": {  
        "name": "[variables('hostingPlanName')]",  
        "sku": "[parameters('sku')]",  
        "workerSize": "[parameters('workerSize')]",  
        "hostingEnvironment": "",  
        "numberOfWorkers": 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'), '2015-05-01-preview').key1)]"
                },
                {
                    "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 (Preview)	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'), '2015-05-01-preview').key1)]"
        },
        {
          "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'), '2015-05-01-preview').key1)]"
        },
        {
          "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'), '2015-05-01-preview').key1)]",
        "AzureWebJobsDashboard": "[concat('DefaultEndpointsProtocol=https;AccountName=', variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'), '2015-05-01-preview').key1)]",
        "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-azuredeploy-json>)
```

Here is an example that uses HTML:

```
<a href="https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-to-azuredeploy-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

4/12/2019 • 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 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

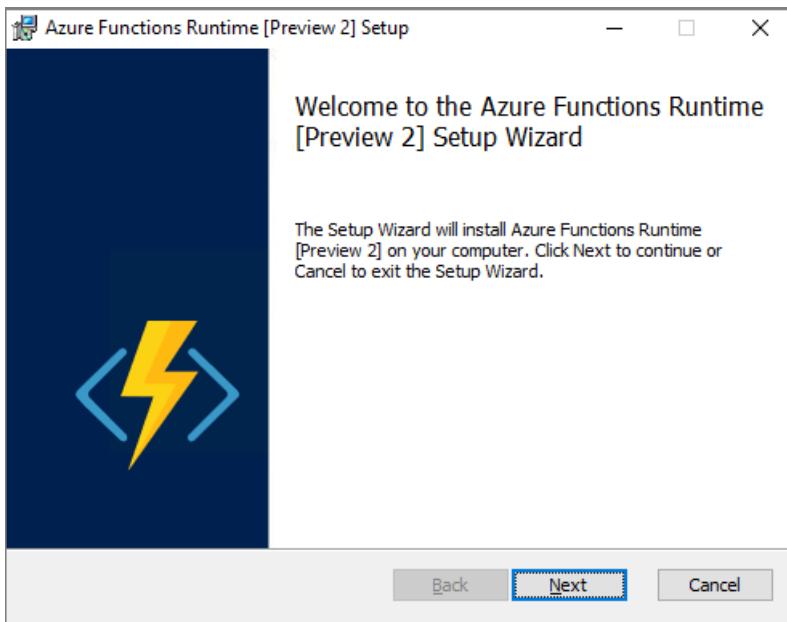
If you have previously installed the Azure Functions Runtime preview, you must uninstall before installing the latest release. Uninstall the Azure Functions Runtime preview by removing the program in Add/Remove Programs in Windows.

Install the Azure Functions Runtime Preview

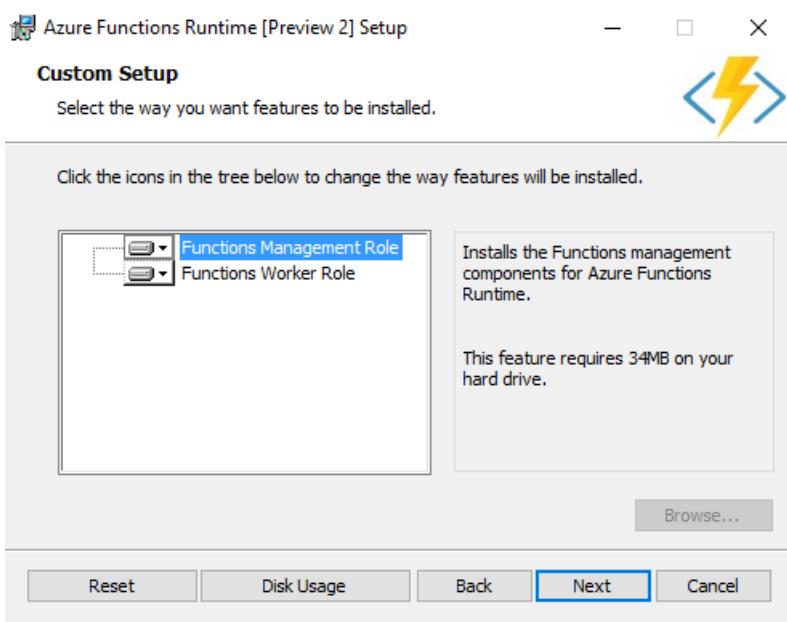
The Azure Functions Runtime Preview Installer guides you through the installation of the Azure Functions Runtime preview Management and Worker Roles. It is possible to install the Management and Worker role on the same machine. However, as you add more function apps, you must deploy more worker roles 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.



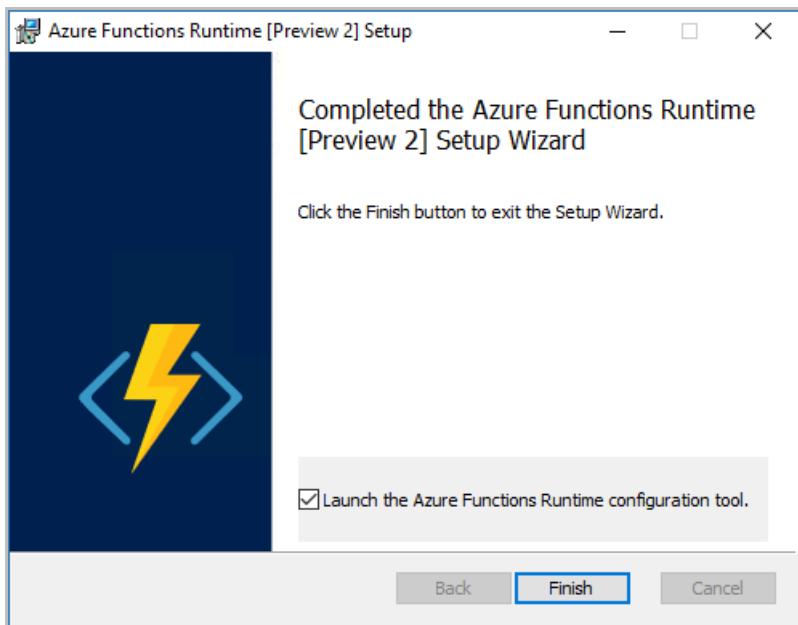
2. Click **Next**.
3. Once you have read the terms of the **EULA**, **check the box** to accept the terms and click **Next** to advance.
4. Select the roles you want to install on this machine **Functions Management Role** and/or **Functions Worker Role** and click **Next**.



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.



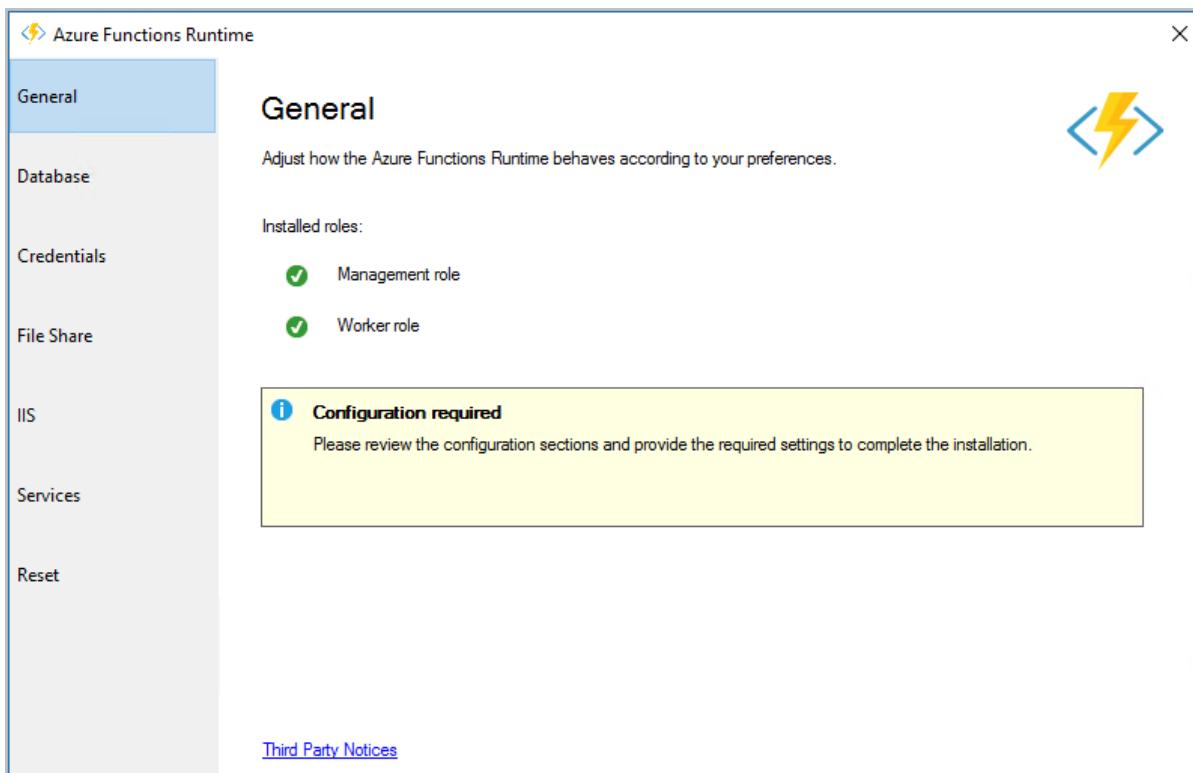
NOTE

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.

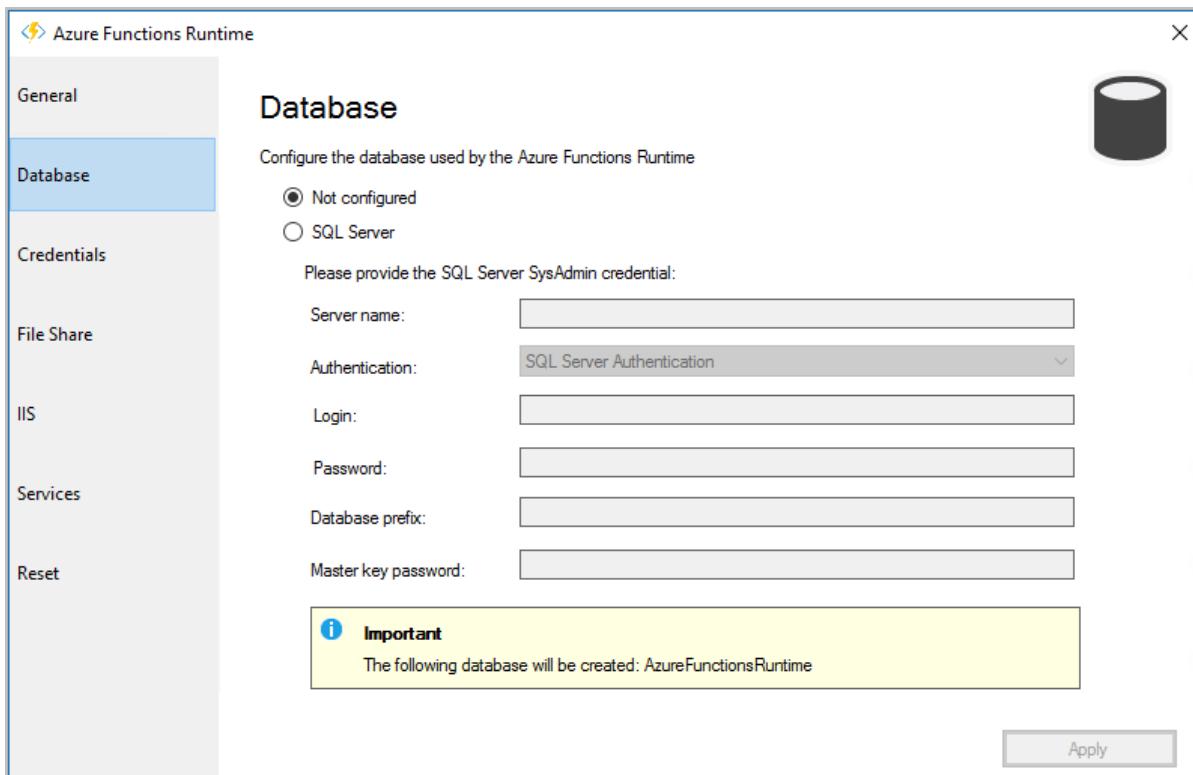
Configure the Azure Functions Runtime

To complete the Azure Functions Runtime installation, you must complete the configuration.

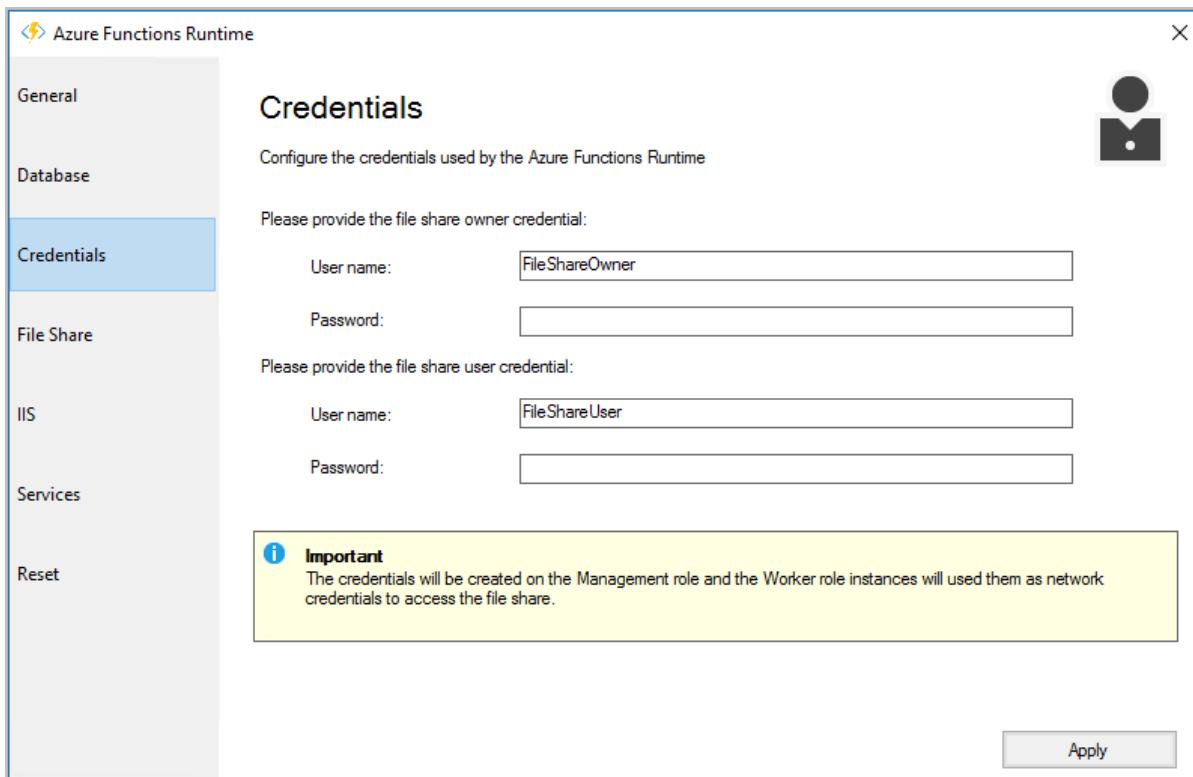
1. The **Azure Functions Runtime** configuration tool shows which roles are installed on your machine.



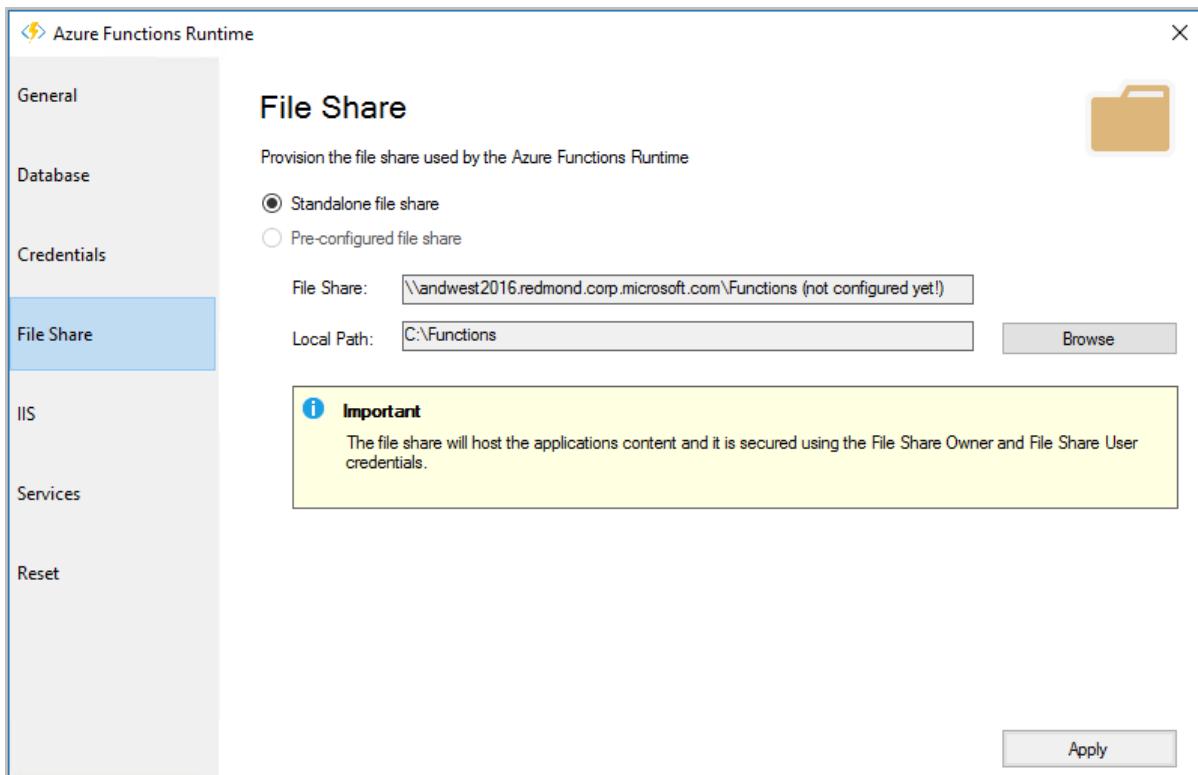
2. Click the **Database** tab, enter the connection details for your SQL Server instance, including specifying a **Database master key**, and click **Apply**. Connectivity to a SQL Server instance is required in order for the Azure Functions Runtime to create a database to support the Runtime.



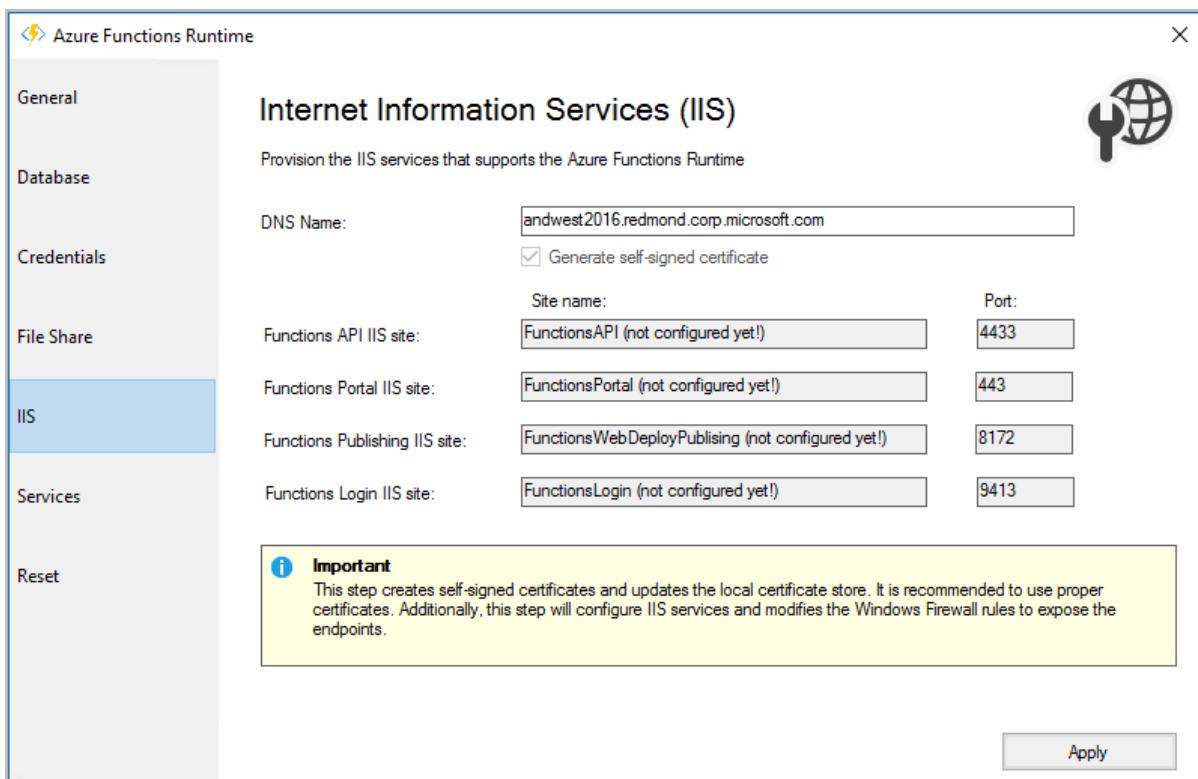
3. Click the **Credentials** tab. Here, you must create two new credentials for use with a file share for hosting all your function apps. Specify **User name** and **Password** combinations for the **file share owner** and for the **file share user**, then click **Apply**.



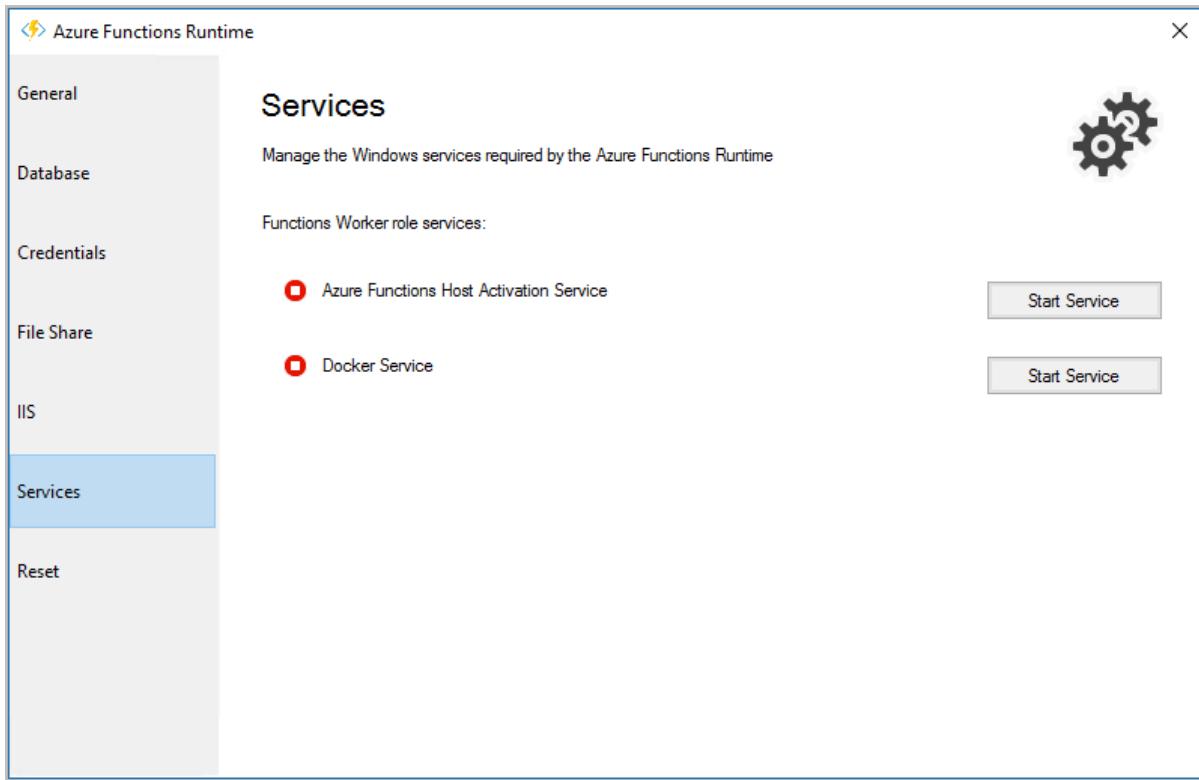
4. Click the **File Share** tab. Here you must specify the details of the file share location. The file share can be created for you or you can use an existing File Share and click **Apply**. If you select a new File Share location, you must specify a directory for use by the Azure Functions Runtime.



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>/>.

The screenshot shows the 'Function Apps' section of the Azure Functions Runtime portal. The left sidebar has 'Subscriptions' and 'Function Apps' selected. The main area displays a lightning bolt icon and the message 'No function apps to display'. Below this, there is descriptive text about Azure Functions being an event-based serverless compute experience. A link 'Learn more about azure functions' is present at the bottom.

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.

Azure Functions Runtime

Azure Functions is @ your fingertips!

- Process events with serverless code on-premises
- Rich and powerful services
- Effortless management experience

Sign in

Copyright © 2017. All Rights Reserved

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**.

Choose a plan below to create new subscription

DefaultPlan
A default plan for Azure Functions Runtime.

Provide a friendly subscription name
SampleSubscription

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

Subscription

Runtime image

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

Subscriptions

All subscriptions

Function Apps

TestFunction

Functions

TimerTriggerCSharp1

Integrate

Manage

TestFunctionv1

Functions

Choose a template below or [go to the quickstart](#)

Search by trigger, language, or description

Language: All Scenario: All

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:

Name:

Timer trigger

Schedule (*)

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

Subscriptions +

All subscriptions

Function Apps

Testfunction

Functions

TimerTriggerCSharp1

Integrate

Manage

run.csx

Save

```
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
```

Logs

Pause Clear Copy logs Expand

```
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-a7deddfcbb02, 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)
```

View files

How to manage a function app in the Azure portal

5/20/2019 • 3 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. This topic describes how to configure and manage your function apps in the Azure portal.

To begin, go to the [Azure portal](#) and sign in to your Azure account. In the search bar at the top of the portal, type the name of your function app and select it from the list. After selecting your function app, you see the following page:

The screenshot shows the Azure portal's Overview page for a function app named "myfunctionapp". The left sidebar shows a search bar with "myfunctionapp" and a dropdown for "Visual Studio Enterprise". Under "Function Apps", "myfunctionapp" is selected. Below the sidebar are three main sections: "Overview", "Platform features" (which is highlighted with a red box), and "Configured features". The "Overview" section displays basic information: Status (Running), Subscription (Visual Studio Enterprise), Resource group (myfunctionapp), URL (https://myfunctionapp.azurewebsites.net). The "Platform features" section has tabs for "Function app settings" and "Application settings" (which is also highlighted with a red box). A message says "You have created a function app! Now it is time to add your code...". A blue button at the bottom right says "+ New function".

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.

Dashboard > functions-ggailey777 > X

Save Discard

Application 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 (i) 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	(i) Hidden value. Click show values button abc	Delete Edit
AzureWebJobsStorage	(i) Hidden value. Click show values button abc	Delete Edit
FUNCTIONS_EXTENSION_VERSION	(i) Hidden value. Click show values button abc	Delete Edit
FUNCTIONS_WORKER_RUNTIME	(i) Hidden value. Click show values button abc	Delete Edit
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING	(i) Hidden value. Click show values button abc	Delete Edit
WEBSITE_CONTENTSHARE	(i) Hidden value. Click show values button abc	Delete Edit
WEBSITE_NODE_DEFAULT_VERSION	(i) Hidden value. Click show values button abc	Delete Edit
WEBSITE_TIME_ZONE	(i) Hidden value. Click show values button abc	Delete Edit

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)		

These settings are stored encrypted, and you must select **Show values** to see the values in the portal.

To add a setting, select **New application setting** and add the new key-value pair.

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](#)

When you develop a function app locally, these values are maintained in the local.settings.json project file.

Platform features

The screenshot shows the 'Platform features' tab selected in the Azure portal. The page is organized into several sections:

- General Settings:** Function app settings, Configuration, Properties, Backups, All settings.
- Code Deployment:** Deployment Center.
- Development tools:** Logic Apps, Console (CMD / PowerShell), Advanced tools (Kudu), App Service Editor, Resource Explorer, Site Extensions.
- Networking:** Networking, SSL, Custom domains, Authentication / Authorization, Identity, Push notifications.
- Monitoring:** Diagnostic logs, Log streaming, Process explorer, Metrics.
- API:** API Management, 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.

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 **Platform features** tab 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 topic 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](#)
- [API definition](#)

For more information about how to work with App Service settings, see [Configure Azure App Service Settings](#).

App Service Editor



The App Service editor is an advanced in-portal editor that you can use to modify JSON configuration files and code files alike. Choosing this option launches a separate browser tab with a basic editor. This 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 default function app blade.

The screenshot shows the App Service Editor interface. On the left is a sidebar titled 'EXPLORE' showing the project structure:

- WORKING FILES
- WWWROOT
 - GithubWebhookJS1
 - HttpTriggerCSharp1
 - HttpTriggerFSharp1
 - HttpTriggerJS1
 - function.json
 - index.js**
 - node_modules
 - QueueTriggerCSharp1
 - TimerTriggerCSharp1
 - .gitignore
 - host.json
 - iisnode.yml
 - index.js
 - LICENSE
 - package.json
 - process.json
 - README.md
 - web.config

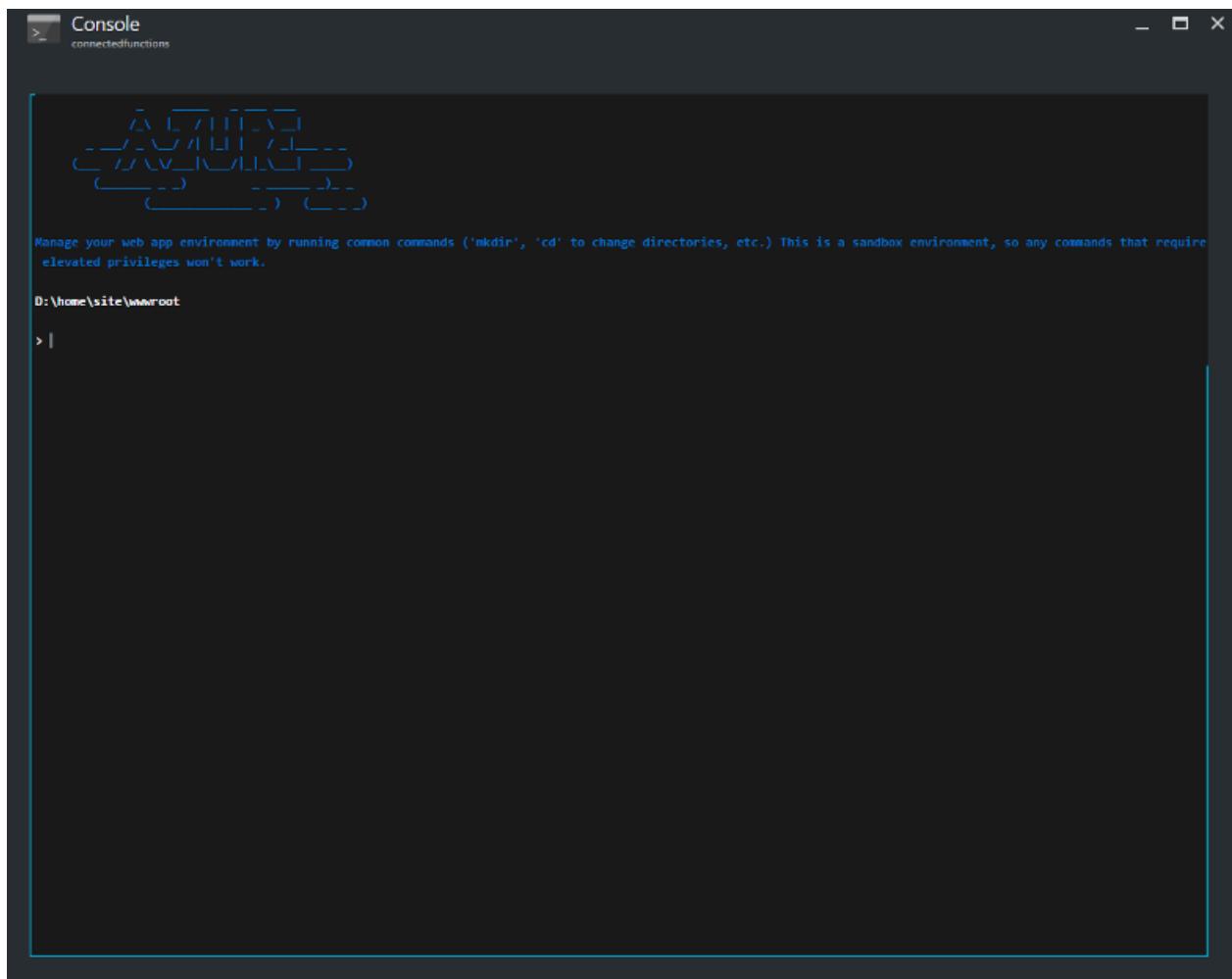
The main area displays the contents of 'index.js' (HttpTriggerJS1):

```
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 };
```

Console



The in-portal console is an ideal developer tool when you prefer to interact with your function app from the command line. Common commands include directory and file creation and navigation, as well as executing batch files and scripts.



The screenshot shows a terminal window titled "Console" with the URL "connectedfunctions". The window contains a command prompt with the path "D:\home\site\wwwroot". The command ">> |" is visible at the bottom, indicating where input can be entered. The background of the terminal is dark, and the text is white.

Advanced tools (Kudu)



The advanced tools for App Service (also known as Kudu) provide access to advanced administrative features of your function app. From Kudu, you manage system information, app settings, environment variables, site extensions, HTTP headers, and server variables. You can also launch **Kudu** by browsing to the SCM endpoint for your function app, like <https://<myfunctionapp>.scm.azurewebsites.net/>

/

+ | 3 items |



	Name	Modified	Size
	data	10/26/2016, 7:15:25 PM	
	LogFiles	10/26/2016, 7:15:17 PM	
	site	10/26/2016, 7:15:17 PM	



Use old console

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new CMD process.
Type 'cls' to clear the console

Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

D:\home>
```

Deployment options



Functions lets you develop your function code on your local machine. You can then upload your local function app project to Azure. In addition to traditional FTP upload, Functions lets you deploy your function app using popular continuous integration solutions, like GitHub, Azure DevOps, Dropbox, Bitbucket, and others. For more information, see [Continuous deployment for Azure Functions](#). To upload manually using FTP or local Git, you also must [configure your deployment credentials](#).

CORS



To prevent malicious code execution in your services, App Service blocks calls to your function apps from external sources. Functions supports cross-origin resource sharing (CORS) to let you define a "whitelist" of allowed origins from which your functions can accept remote requests.

CORS

ConnectedFunctions

Save Discard

ALLOWED ORIGINS

- https://functions.azure.com
- https://functions-staging.azure.com
- https://functions-next.azure.com

Authentication



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](#).

Authentication / Authorization

Save Discard

Authentication / Authorization

App Service Authentication

Off On

Action to take when request is not authenticated

Log in with Azure Active Directory

Authentication Providers

- Azure Active Directory
Not Configured
- Facebook
Not Configured
- Google
Not Configured
- Twitter
Not Configured
- Microsoft Account
Not Configured

Advanced Settings

Token Store Off On

Microsoft Account Authentication Settings

These settings allow users to sign in with Microsoft Account. Click here to learn more.

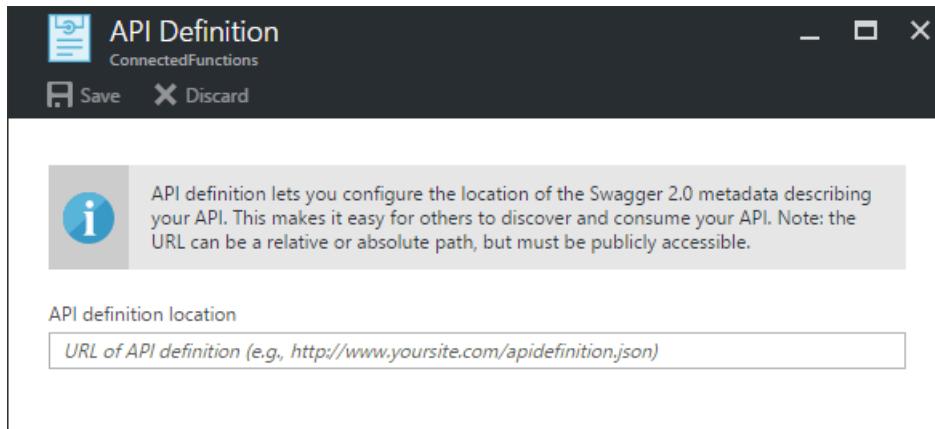
SCOPE	DESCRIPTION
wl.basic	Read access to a user's basic profile info. Also enables read access to...
wl.offline_access	The ability of an app to read and update a user's info at any time.
wl.signin	Single sign-in behavior.
wl.birthday	Read access to a user's birthday info including birth day, month, and...
wl.calendars	Read access to a user's calendars and events.
wl.calendars_update	Read and write access to a user's calendars and events.
wl.contacts_birthday	Read access to the birth day and birth month of a user's contacts.
wl.contacts_create	Creation of new contacts in the user's address book.
wl.contacts_calendars	Read access to a user's calendars and events.
wl.contacts_photos	Read access to a user's albums, photos, videos, and audio, and their ...
wl.contacts_skydrive	Read access to Microsoft OneDrive files that other users have share...
wl.emails	Read access to a user's personal, preferred, and business email addr...
wl.events_create	Read access to Microsoft OneDrive files that other users have share...
wl.imap	Read and write access to a user's email using IMAP, and send access...
wl.phone_numbers	Read access to a user's personal, business, and mobile phone num...

OK

API definition



Functions supports Swagger to allow clients to more easily consume your HTTP-triggered functions. For more information on creating API definitions with Swagger, visit [Host a RESTful API with CORS in Azure App Service](#). You can also use Functions Proxies to define a single API surface for multiple functions. For more information, see [Working with Azure Functions Proxies](#).



Next steps

- [Configure Azure App Service Settings](#)
- [Continuous deployment for Azure Functions](#)

How to target Azure Functions runtime versions

6/11/2019 • 4 minutes to read • [Edit Online](#)

A function app runs on a specific version of the Azure Functions runtime. There are two major versions: [1.x](#) and [2.x](#). By default, function apps that are created version 2.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 ("~2" for 2.x or "~1" for 1.x), the function app is automatically updated to new minor versions of the runtime when they become available. New minor versions do not 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.

NOTE

If you pin to a specific 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.

A change to the runtime version causes a function app to restart.

The values you can set in the `FUNCTIONS_EXTENSION_VERSION` app setting to enable automatic updates are currently "~1" for the 1.x runtime and "~2" for 2.x.

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 should only change the runtime version before you have created any functions in your function app. 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.

From the Azure portal

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 **Configured Features**, choose **Function app settings**.

The screenshot shows the Microsoft Azure Functions Overview page for the resource group 'functions-ggailey777'. In the left sidebar, under 'Function Apps', the 'functions-ggailey777' app is selected and highlighted with a red box. The main panel displays the following details:

- Status:** Running
- Subscription:** Visual Studio Enterprise
- Resource group:** functions-ggailey777
- Subscription ID:** <subscription ID>
- Location:** South Central US
- URL:** https://functions-ggailey777.azurewebsites.net
- App Service plan / pricing tier:** SouthCentralUSPlan (Consumption)

In the 'Configured features' section, 'Function app settings' is also highlighted with a red box.

3. In the **Function app settings** tab, locate the **Runtime version**. Note the specific runtime version and the requested major version. In the example below, the version is set to `~2`.

The screenshot shows the 'Function app settings' tab in the Azure portal. Under 'Runtime version', the dropdown menu is open, showing options `~1` and `~2`. The option `~2` is highlighted with a red box. Other sections visible include 'Daily Usage Quota (GB-Sec)', 'Application settings', and 'Function app edit mode'.

4. 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.
5. 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.

NOTE

Using the Azure portal, you can't change the runtime version for a function app that already contains functions.

From the Azure CLI

You can also view and set the `FUNCTIONS_EXTENSION_VERSION` from the Azure CLI.

NOTE

Because other settings may be impacted by the runtime version, you should change the version in the portal. The portal automatically makes the other needed updates, such as Node.js version and runtime stack, when you change runtime versions.

Using the Azure CLI, view the current runtime version with the [az functionapp config appsettings set](#) command.

```
az functionapp config appsettings list --name <function_app> \
--resource-group <my_resource_group>
```

In this code, replace `<function_app>` with the name of your function app. Also replace `<my_resource_group>` with the name of the resource group for your function app.

You see the `FUNCTIONS_EXTENSION_VERSION` in the following output, which has been truncated for clarity:

```
[
{
  "name": "FUNCTIONS_EXTENSION_VERSION",
  "slotSetting": false,
  "value": "~2"
},
{
  "name": "FUNCTIONS_WORKER_RUNTIME",
  "slotSetting": false,
  "value": "dotnet"
},
...
{
  "name": "WEBSITE_NODE_DEFAULT_VERSION",
  "slotSetting": false,
  "value": "8.11.1"
}
]
```

You can update the `FUNCTIONS_EXTENSION_VERSION` setting in the function app with the [az functionapp config appsettings set](#) command.

```
az functionapp config appsettings set --name <function_app> \
--resource-group <my_resource_group> \
--settings FUNCTIONS_EXTENSION_VERSION=<version>
```

Replace `<function_app>` with the name of your function app. Also replace `<my_resource_group>` with the name of the resource group for your function app. Also, replace `<version>` with a valid version of the 1.x runtime or `~2` for version 2.x.

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

[Target the 2.0 runtime in your local development environment](#)

[See Release notes for runtime versions](#)

Manually install or update Azure Functions binding extensions from the portal

2/22/2019 • 2 minutes to read • [Edit Online](#)

The Azure Functions version 2.x runtime uses binding extensions to implement code for triggers and bindings. Binding extensions are provided in NuGet packages. To register an extension, you essentially install a package. When developing functions, the way that you install binding extensions depends on the development environment. For more information, see [Register binding extensions](#) in the triggers and bindings article.

Sometimes you need to manually install or update your binding extensions in the Azure portal. For example, you may need to update a registered binding to a newer version. You may also need to register a supported binding that can't be installed in the **Integrate** tab in the portal.

Install a binding extension

Use the following steps to manually install or update extensions from the portal.

1. In the [Azure portal](#), locate your function app and select it. Choose the **Overview** tab and select **Stop**. Stopping the function app unlocks files so that changes can be made.
2. Choose the **Platform features** tab and under **Development tools** select **Advanced Tools (Kudu)**. The Kudu endpoint (https://<APP_NAME>.scm.azurewebsites.net/) is opened in a new window.
3. In the Kudu window, select **Debug console > CMD**.
4. In the command window, navigate to `D:\home\site\wwwroot` and choose the delete icon next to `bin` to delete the folder. Select **OK** to confirm the deletion.
5. Choose the edit icon next to the `extensions.csproj` file, which defines the binding extensions for the function app. The project file is opened in the online editor.
6. Make the required additions and updates of **PackageReference** items in the **ItemGroup**, then select **Save**. The current list of supported package versions can be found in the [What packages do I need?](#) wiki article. All three Azure Storage bindings require the Microsoft.Azure.WebJobs.Extensions.Storage package.
7. From the `wwwroot` folder, run the following command to rebuild the referenced assemblies in the `bin` folder.

```
dotnet build extensions.csproj -o bin --no-incremental --packages D:\home\.nuget
```
8. Back in the **Overview** tab in the portal, choose **Start** to restart the function app.

Next steps

[Learn more about Azure functions triggers and bindings](#)

How to disable functions in Azure Functions

9/7/2018 • 2 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 is defined for the function. The way you do that depends on the runtime version and the programming language:

- Functions 1.x
 - Scripting languages
 - C# class libraries
- Functions 2.x
 - One way for all languages
 - Optional way for C# class libraries

Functions 1.x - scripting languages

For scripting languages such as C# script and JavaScript, you use the `disabled` property of the `function.json` file to tell the runtime not to trigger a function. This 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.

You can edit the file in the Azure portal or use the **Function State** switch on the function's **Manage** tab. The portal switch works by changing the `function.json` file.

The screenshot shows the Microsoft Azure portal interface. In the top navigation bar, the URL is https://portal.azure.com. The left sidebar includes links for Apps, GitHub, Docs, and VSTS. The main content area displays the 'functionapp0510p - CosmosTriggerCSharp1' function app settings. The 'Function State' section has two buttons: 'Enabled' (which is highlighted with a red box) and 'Disabled'. Below this, there is a 'Delete function' button. On the left, the sidebar lists 'All subscriptions' and 'Function Apps'. Under 'Function Apps', it shows 'fs0629', 'fs0629b', and 'functionapp0510p'. Under 'functionapp0510p', it shows 'Functions' and 'CosmosTriggerCSharp1'. The 'Manage' button in the sidebar is also highlighted with a red box. At the bottom of the main content area, there is a blue button labeled 'Add new host key'.

Functions 1.x - C# class libraries

In a Functions 1.x class library, you use a `[Disable]` attribute to prevent a 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 2.x - all languages

In Functions 2.x you disable a function by using an app setting. For example, to disable a function named `QueueTrigger`, you create an app setting named `AzureWebJobs.QueueTrigger.Disabled`, and set it to `true`. To enable the function, set the app setting to `false`. You can also use the **Function State** switch on the function's **Manage** tab. The switch works by creating and deleting the `AzureWebJobs.<functionname>.Disabled` app setting.

The screenshot shows the Microsoft Azure portal interface for a function app named "functionapp0510p - CosmosTriggerCSharp1". In the top right corner, there is a user profile for "test@contoso.com" and a "MY AAD" link. The main content area displays the "Function State" section, which includes two buttons: "Enabled" (highlighted with a red box) and "Disabled". Below this, there is a "Delete function" button. On the left side, there is a sidebar with various icons and links, including "Search", "All subscriptions", "Function Apps", and a list of other function apps like "fs0629" and "fs0629b". Under "functionapp0510p", there is a "Functions" section containing "CosmosTriggerCSharp1". The "Manage" button in the sidebar is also highlighted with a red box.

Functions 2.x - C# class libraries

In a Functions 2.x class library, we recommend that you use the method that works for all languages. But if you prefer, you can [use the Disable attribute as in Functions 1.x](#).

Next steps

This article is about disabling automatic triggers. For more information about triggers, see [Triggers and bindings](#).

Monitor Azure Functions

7/29/2019 • 19 minutes to read • [Edit Online](#)

Azure Functions offers built-in integration with [Azure Application Insights](#) to monitor functions. This article shows you how to configure Azure Functions to send system-generated log files to Application Insights.

We recommend using Application Insights because it 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. It's designed to help you continuously improve performance and usability. You can even use Application Insights during local function app project development. For more information, see [What is Application Insights?](#).

As the required Application Insights instrumentation is built into Azure Functions, all you need is a valid instrumentation key to connect your function app to an Application Insights resource.

Application Insights pricing and limits

You can try out Application Insights integration with Function Apps for free. There's a daily limit to how much data can be processed for free. 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](#).

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**.

New function app in the portal

When you [create your function app in the Azure portal](#), 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 nearest region.

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.

The screenshot shows the Azure Function App creation interface. On the left, under 'Function App' settings, the 'App name' is set to 'functionapp0921'. The 'Subscription' is 'Visual Studio Enterprise'. 'Resource Group' is 'Create new' with 'functionapp0921'. 'OS' is 'Windows'. 'Hosting Plan' is 'Consumption Plan'. 'Location' is 'West Europe'. 'Runtime Stack' is '.NET'. 'Storage' is 'Create new' with 'functionapp09218f22'. Under 'Application Insights', the 'Create new resource' option is selected, with 'New resource name' as 'functionapp0921' and 'Location' as 'West Europe'. A red box highlights the 'Create new resource' section. At the bottom, there are 'Create' and 'Automation options' buttons, and a large 'Apply' button on the right.

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

When you create a function app using the [Azure CLI](#), [Visual Studio](#), or [Visual Studio Code](#), you must create the Application Insights resource. You can then add the instrumentation key from that resource as an application setting in your function app.

Functions makes it easy to add Application Insights integration to a function app from the [Azure portal](#).

1. In the [portal](#), select **All services > Function Apps**, select your function app, and then select the **Application Insights** banner at the top of the window

2. Create an Application Insights resource by using the settings specified in the table below the image.

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

3. Select **OK**. 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.
4. Back in your function app, select **Application settings**, and then scroll down to **Application settings**. If you see a setting named `APPINSIGHTS_INSTRUMENTATIONKEY`, Application Insights integration is enabled for your function app running in Azure.

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](#).

View 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 the **Monitor** tab.

The screenshot shows the Azure Functions portal interface. On the left, there's a sidebar with various icons and a search bar. Below the search bar, under 'Function Apps', it shows 'All subscriptions' and 'Function Apps'. Under 'functionapp0419p2', it lists 'Functions' and 'QueueTriggerCSharp1'. The 'Monitor' option under 'QueueTriggerCSharp1' is highlighted with a red box. On the right, the main content area shows a code editor with 'RUN.CSX' containing C# code for a queue trigger function. Below the code editor are tabs for 'Logs' and 'Errors and warnings'.

2. Select **Refresh** periodically, until the list of function invocations appears.

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.)

The screenshot shows the Azure Functions portal interface with the 'Monitor' tab selected. The left sidebar shows the same structure as the previous screenshot. The main content area now displays a table of function invocations. The table has columns for 'DATE', 'SUCCESS', and 'RESULT CODE'. The data shows several successful invocations from April 20, 2018, between 16:47:15 and 16:47:40. A 'Refresh' button is visible at the top of the table.

DATE	SUCCESS	RESULT CODE
2018-04-20T16:47:45.005Z	True	0
2018-04-20T16:47:40.016Z	True	0
2018-04-20T16:47:35.006Z	True	0
2018-04-20T16:47:30.006Z	True	0
2018-04-20T16:47:25.006Z	True	0
2018-04-20T16:47:20.002Z	True	0
2018-04-20T16:47:15.002Z	True	0

3. To see the logs for a particular function invocation, select the **Date** column link for that invocation.

Refresh	Run in Application Insights	
Application Insights Instance functionapp0420p2	Success count in last 30 days 130	Error count in last 30 days 0
DATE ▾	SUCCESS ▾	RESULT CODE ▾
2018-04-20T16:55:05.013Z	True	0
2018-04-20T16:55:00.006Z	True	0
2018-04-20T16:54:55.004Z	True	0
2018-04-20T16:54:50.018Z	True	0
2018-04-20T16:54:45.001Z	True	0
2018-04-20T16:54:40.012Z	True	0
2018-04-20T16:54:35.001Z	True	0
2018-04-20T16:54:30.001Z	True	0

The logging output for that invocation appears in a new page.

Invocation Details	
Run in Application Insights	
MESSAGE	LOG LEVEL
Function completed (Success, Id=c764367d-5710-43ae-9496-c0aa5bc45afe, Duration=0...)	Information
Function started (Id=c764367d-5710-43ae-9496-c0aa5bc45afe)	Information
C# Timer trigger function executed at: 4/20/2018 4:55:05 PM	Information

You can see that both pages have a **Run in Application Insights** link to the Application Insights Analytics query that retrieves the data.

Home > functions-ggailey777 - HttpTriggerCSharp1

functions-ggailey777 - HttpTriggerCSharp1

Function Apps

Search

Visual Studio Enterprise

Function Apps

- ▼ **functions-ggailey777**
- ▼ **Functions**
- ▶ [HttpTriggerCSharp1](#)
- ▶ [TimerTriggerCSharp1](#)
- ▶ **Proxies**
- ▶ **Slots (preview)**

Refresh

Application Insights Instance: functions-ggailey777 Success count in last 30 days: 14 Error count in last 30 days: 6

Query returned 20 items Run in Application Insights Troubleshoot your app Diagnose and solve problems

DATE (UTC) ▾	SUCCESS ▾	RESULT CODE ▾	DURATION (MS) ▾
2019-04-08 07:16:10.763	✓	0	6.4631
2019-04-08 07:15:19.398	✓	0	2.1411
2019-04-08 07:15:05.407	✓	0	10.5045
2019-04-08 07:05:19.357	✓	0	8.0953
2019-04-08 07:04:46.803	✓	0	8.1741
2019-04-08 07:04:40.637	✓	0	133.798
2019-04-08 06:55:59.997	✓	0	4.2124
2019-04-08 06:55:54.849	✓	0	2.4622
2019-04-08 06:55:44.079	✓	0	2.0653
2019-04-08 06:53:19.047	✓	0	1.8892
2019-04-08 06:53:00.107	✓	0	2.1717
2019-04-08 06:52:21.397	✓	0	6.7988
2019-04-07 17:56:33.415	✓	0	7.321
2019-04-07 17:49:56.440	✓	0	23.8193
2019-04-07 17:48:22.454	⚠	0	317.5018

The following query is displayed. You can see that the invocation list is limited to the last 30 days. The list shows no more than 20 rows (`where timestamp > ago(30d) | take 20`). The invocation details list is for the last 30 days with no limit.

Application Insights functions-ggailey777 > Analytics

New Query 1 + Run Time range: Set in query Save Copy link Export New alert rule Query explorer

functions-ggailey777

Schema Filter (preview) Collapsible all

Filter by name or type...

Completed 00:00:09.707 20 records Display time (UTC+00:00)

requests | project timestamp, id, operation_Name, success, resultCode, duration, operation_Id, cloud_RoleName | where timestamp > ago(30d) | where cloud_RoleName ~ 'functions-ggailey777' and operation_Name == 'HttpTriggerCSharp1' | order by timestamp desc | take 20

timestamp [UTC]	id	operation_Name	success	resultCode	duration	operation_Id
2019-04-08T07:16:10.763	f0hzN/V0p2c=.7fc2b698_	HttpTriggerCSharp1	True	0	6.4631	f0hzN/V0p2c=
2019-04-08T07:15:19.398	rFjoQUPKePU=.7fc2b67c_	HttpTriggerCSharp1	True	0	2.1411	rFjoQUPKePU=
2019-04-08T07:15:05.407	ucHluRhA9S0=.7fc2b675_	HttpTriggerCSharp1	True	0	10.5045	ucHluRhA9S0=
2019-04-08T07:05:19.357	IYZcQGz0TOY=.7fc2b558_	HttpTriggerCSharp1	True	0	8.0953	IYZcQGz0TOY=
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	QvSRNIp1CA=.5220de..._	HttpTriggerCSharp1	True	0	2.4622	QvSRNIp1CA=
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=.5220dd2..._	HttpTriggerCSharp1	True	0	2.1717	9Q7RJ2ISXzs=
2019-04-08T06:52:21.397	Trvet9BXy18=.5220ddbf_	HttpTriggerCSharp1	True	0	6.7988	Trvet9BXy18=

Page 1 of 1 items per page 50 1 - 20 of 20 it

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, go to the function app's **Overview** page. Under **Configured features**, select **Application Insights**.

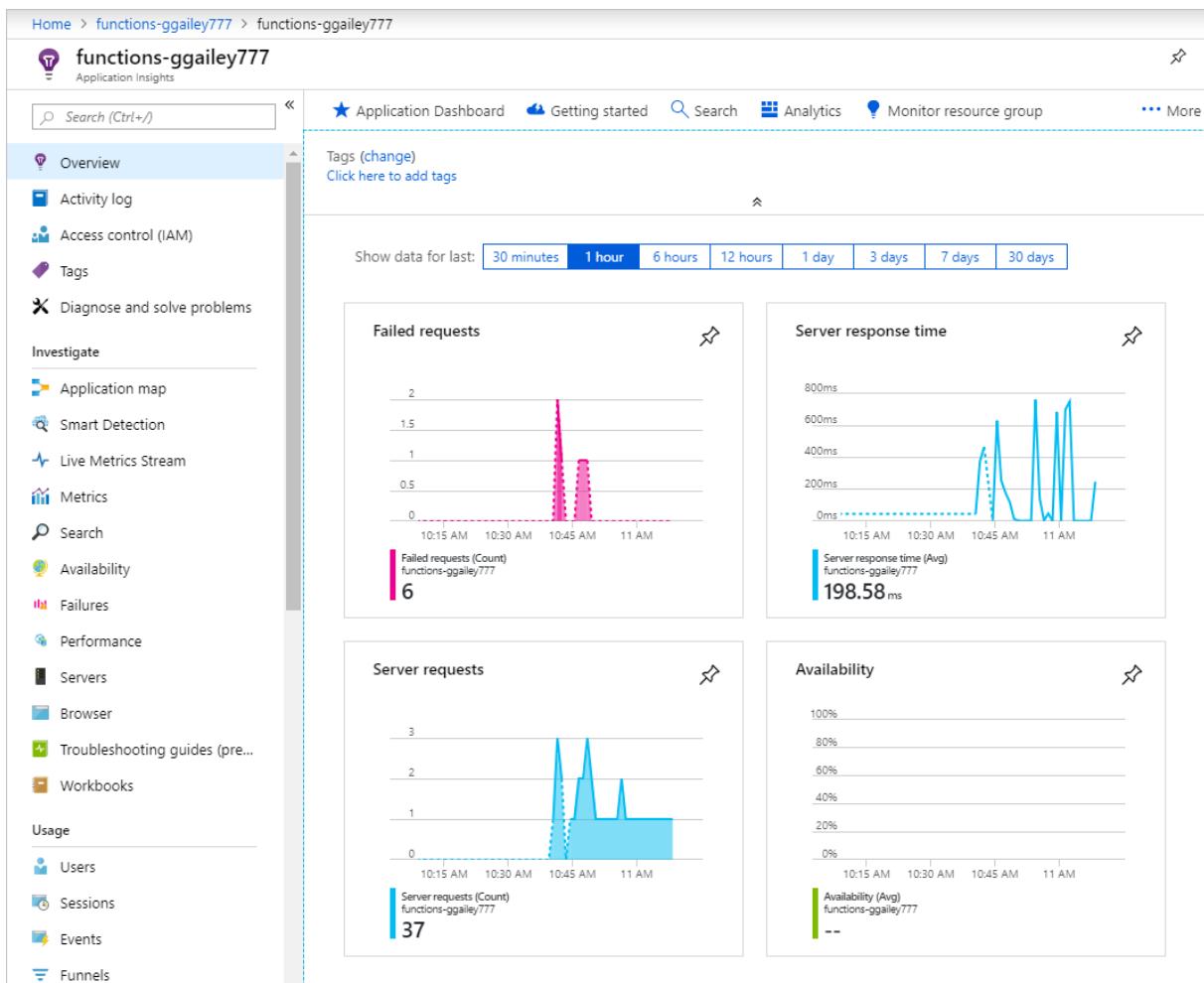
The screenshot shows the Azure portal interface for managing a Function App. On the left, there's a sidebar with various icons for different services like Storage, Queues, and Metrics. The main area shows the 'Overview' tab for the 'functionapp0425' function app. Key details shown include:

- Status:** Running
- Subscription:** Windows Azure MSDN - Visual Studio Ultimate
- Subscription ID:** {subscription ID}
- Resource group:** functionapp0425rg
- URL:** <https://functionapp0425.azurewebsites.net>
- Location:** West US 2
- App Service plan / pricing tier:** WestUS2Plan (Consumption)

Below the overview, the 'Configured features' section is expanded, listing:

- Function app settings
- Application settings
- Extensions (1 installed)
- Application Insights** (this item is highlighted with a red box)

For information about how to use Application Insights, see the [Application Insights documentation](#). This section shows some examples of how to view data in Application Insights. If you're already familiar with Application Insights, you can go directly to [the sections about how to configure and customize the telemetry data](#).



The following areas of Application Insights can be helpful when evaluating the behavior, performance, and errors in your functions:

TAB	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.
Servers	View resource utilization and throughput per server. This data can be useful for debugging scenarios where functions are bogging down your underlying resources. Servers are referred to as Cloud role instances .
Metrics	Create charts and alerts that are based on metrics. Metrics include the number of function invocations, execution time, and success rates.
Live Metrics Stream	View metrics data as it's created in 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.

The screenshot shows the Azure Application Insights Analytics blade for the functionapp0915 application. The top navigation bar includes 'Search (Ctrl+)', 'Metrics Explorer', 'Analytics' (which is highlighted with a red box), 'Time range', 'Refresh', and 'More'. The main content area has a header 'NEW - Analyze conversion rates within your app with the Funnels tool.' Below this is the 'Essentials' section, which displays various resource details:

- Resource group (change): functionapp0915
- Type: ASP.NET
- Location: East US
- Instrumentation Key: 302a48f0-e491-4068-9d4c-38365abc4c72
- Subscription name (change): Windows Azure MSDN - Visual Studio Ulti...
- Subscription ID: {subscription ID}

Below these details is a metrics summary card with the following data:

Alerts	Live Stream	Smart Detection	Availability	App map
0	1	0	--	
0	Servers	Users (1 day)	Detections (7d)	

The screenshot shows the Azure Application Insights Schema blade for the functionapp0915 application. The left sidebar is titled 'SCHEMA' and lists the following data sources under 'ACTIVE':

- functionapp0915
- APPLICATION INSIGHTS
 - traces
 - customEvents
 - pageViews
 - requests
 - dependencies
 - exceptions
 - availabilityResults
 - customMetrics
 - performanceCounters
 - browserTimings
- OTHER DATA SOURCES
- + Add new data source

The main pane shows the 'traces' table results. The title bar indicates the URL is analytics.applicationinsights.io/subscriptions/aeb4ae60-b7cb-4f3d-966d-f... and the page title is 'Analytics'. The table has columns: timestamp [UTC], message, and severityLevel. The results show log entries from September 18, 2017, such as:

timestamp [UTC]	message	severityLevel
2017-09-18T20:44:25.738	Reading host configuration file 'D:\home\site\wwwroo...' 1	
2017-09-18T20:44:25.738	Host configuration file read: { "logger": { "category..." 1	
2017-09-18T20:44:26.114	Function 'TimerFunction' is disabled 1	
2017-09-18T20:44:26.194	Loaded custom extension: EventGridExtensionConfig fr... 1	
2017-09-18T20:44:26.194	Loaded custom extension: SendGridConfiguration from... 1	
2017-09-18T20:44:26.194	Loaded custom extension: BotFrameworkConfiguration f... 1	
2017-09-18T20:44:27.097	Generating 3 job function(s) 1	
2017-09-18T20:44:27.146	Starting Host (HostId=functionapp0915, Version=1.0.1... 1	
2017-09-18T20:44:27.334	Found the following functions: functionapp0915.HttpT... 1	
2017-09-18T20:44:27.394	Host lock lease acquired by instance ID '78e1fb3ed5a... 1	

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 by 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. See [Structured logging](#) later in this article.

Configure categories and log levels

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. 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.

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.

The Functions runtime creates logs with a category that begin with "Host." In version 1.x, the `function started`, `function executed`, and `function completed` logs have the category `Host.Executor`. Starting in version 2.x, these logs have the category `Function.<YOUR_FUNCTION_NAME>`.

If you write logs in your function code, the category is `Function` in version 1.x of the Functions runtime. In version 2.x, the category is `Function.<YOUR_FUNCTION_NAME>.User`.

Log levels

The Azure Functions logger also includes a *log level* with every log. `LogLevel` is an enumeration, and the integer code indicates relative importance:

LOGLEVEL	CODE
Trace	0

LOGLEVEL	CODE
Debug	1
Information	2
Warning	3
Error	4
Critical	5
None	6

Log level `None` is explained in the next section.

Log configuration in host.json

The `host.json` file configures how much logging a function app sends to Application Insights. For each category, you indicate the minimum log level to send. There are two examples: the first example targets the [Functions version 2.x runtime](#) (.NET Core) and the second example is for the version 1.x runtime.

Version 2.x

The v2.x runtime uses the [.NET Core logging filter hierarchy](#).

```
{
  "logging": {
    "fileLoggingMode": "always",
    "logLevel": {
      "default": "Information",
      "Host.Results": "Error",
      "Function": "Error",
      "Host.Aggregator": "Trace"
    }
  }
}
```

Version 1.x

```
{
  "logger": {
    "categoryFilter": {
      "defaultLevel": "Information",
      "categoryLevels": {
        "Host.Results": "Error",
        "Function": "Error",
        "Host.Aggregator": "Trace"
      }
    }
  }
}
```

This example sets up the following rules:

- For logs with category `Host.Results` or `Function`, send only `Error` level and above to Application Insights. Logs for `Warning` level and below are ignored.
- For logs with category `Host.Aggregator`, send all logs to Application Insights. The `Trace` log level is the same as what some loggers call `Verbose`, but use `Trace` in the `host.json` file.

- For all other logs, send only `Information` level and above to Application Insights.

The category value in `host.json` controls logging for all categories that begin with the same value. `Host` in `host.json` controls logging for `Host.General`, `Host.Executor`, `Host.Results`, and so on.

If `host.json` includes multiple categories that start with the same string, the longer ones are matched first. Suppose you want everything from the runtime except `Host.Aggregator` to log at `Error` level, but you want `Host.Aggregator` to log at the `Information` level:

Version 2.x

```
{
  "logging": {
    "fileLoggingMode": "always",
    "logLevel": {
      "default": "Information",
      "Host": "Error",
      "Function": "Error",
      "Host.Aggregator": "Information"
    }
  }
}
```

Version 1.x

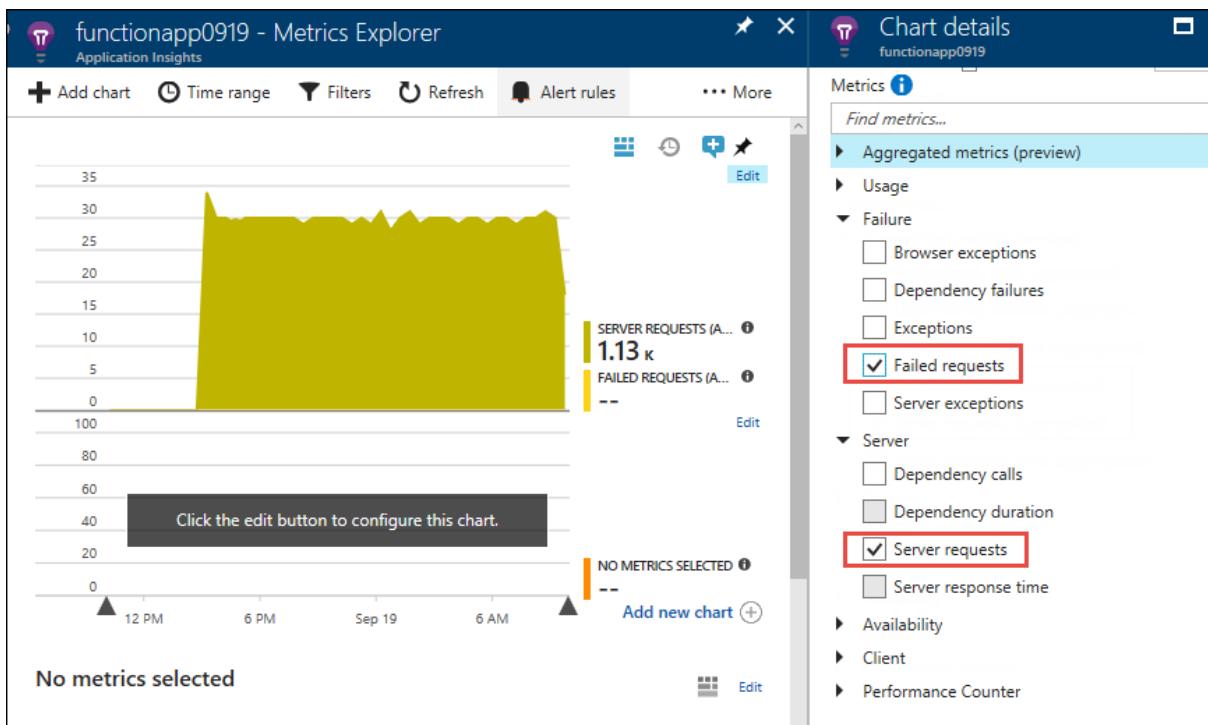
```
{
  "logger": {
    "categoryFilter": {
      "defaultLevel": "Information",
      "categoryLevels": {
        "Host": "Error",
        "Function": "Error",
        "Host.Aggregator": "Information"
      }
    }
  }
}
```

To suppress all logs for a category, you can use log level `None`. No logs are written with that category and there's no log level above it.

The following sections describe the main categories of logs that the runtime creates.

Category Host.Results

These logs show as "requests" in Application Insights. They 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 Host.Aggregator

These 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.

The logs are available in the **customMetrics** table in Application Insights. Examples are the number of runs, success rate, and duration.

The screenshot shows the Azure Analytics interface for the application 'functionapp0919'. On the left, the 'ACTIVE' sidebar shows categories like 'APPLICATION INSIGHTS' and 'customMetrics', with 'customMetrics' selected and highlighted with a red border. In the center, a query editor window titled 'New Query 1*' shows the results for the 'customMetrics' table. The table has columns: timestamp, name, value, valueCount, valueSum, valueMin, and valueMax. The data shows several rows for 'TimerFunction' with various metrics like Success Rate, Duration, Count, Successes, and Failures, all grouped by timestamp.

All of these logs are written at **Information** level. If you filter at **Warning** or above, you won't see any of this data.

Other categories

All logs for categories other than the ones already listed are available in the **traces** table in Application Insights.

timestamp [UTC]	message	severityLevel
2017-09-18T20:44:25.738	Reading host configuration file 'D:\home\site\wwwroot...' 1	
2017-09-18T20:44:25.738	Host configuration file read: { "logger": { "category...": 1 }	
2017-09-18T20:44:26.114	Function 'TimerFunction' is disabled 1	
2017-09-18T20:44:26.194	Loaded custom extension: EventGridExtensionConfig fr... 1	
2017-09-18T20:44:26.194	Loaded custom extension: SendGridConfiguration from... 1	
2017-09-18T20:44:26.194	Loaded custom extension: BotFrameworkConfiguration f... 1	
2017-09-18T20:44:27.097	Generating 3 job function(s) 1	
2017-09-18T20:44:27.146	Starting Host (HostId=functionapp0915, Version=1.0.1...) 1	
2017-09-18T20:44:27.334	Found the following functions: functionapp0915.HttpT... 1	
2017-09-18T20:44:27.394	Host lock lease acquired by instance ID '78e1fb3ed5a...' 1	

All logs with categories that begin with `Host` are written by the Functions runtime. The "Function started" and "Function completed" logs have category `Host.Executor`. For successful runs, these logs are `Information` level. Exceptions are logged at `Error` level. The runtime also creates `Warning` level logs, for example: queue messages sent to the poison queue.

Logs written by your function code have category `Function` and can be any log level.

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:

Version 2.x

```
{  
  "logging": {  
    "applicationInsights": {  
      "samplingSettings": {  
        "isEnabled": true,  
        "maxTelemetryItemsPerSecond" : 20  
      }  
    }  
  }  
}
```

Version 1.x

```
{  
  "applicationInsights": {  
    "sampling": {  
      "isEnabled": true,  
      "maxTelemetryItemsPerSecond" : 5  
    }  
  }  
}
```

NOTE

[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.

Write logs in C# functions

You can write logs in your function code that appear as traces in Application Insights.

ILogger

Use an [ILogger](#) parameter in your functions instead of a [TraceWriter](#) parameter. Logs created by using [TraceWriter](#) go to Application Insights, but [ILogger](#) lets you do [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."

```
public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, ILogger logger)  
{  
  logger.LogInformation("Request for item with key={itemKey}.", id);
```

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"  
    }  
}
```

Custom metrics logging

In C# script functions, 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.

Write logs in JavaScript functions

In Node.js functions, use `context.log` to write logs. Structured logging isn't enabled.

```
context.log('JavaScript HTTP trigger function processed a request.' + context.invocationId);
```

Custom metrics logging

When you're running on [version 1.x](#) of the Functions runtime, Node.js functions can use the `context.log.metric` method to create custom metrics in Application Insights. This method isn't currently supported in version 2.x. Here's a sample method call:

```
context.log.metric("TestMetric", 1234);
```

This code is an alternative to calling `trackMetric` by using the Node.js SDK for Application Insights.

Log custom telemetry in C# functions

You can use the [Microsoft.ApplicationInsights](#) NuGet package to send custom telemetry data to Application Insights. The following C# example uses the [custom telemetry API](#). The example is for a .NET class library, but the Application Insights code is the same for C# script.

Version 2.x

The version 2.x runtime uses 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.Linq;
using System.Threading.Tasks;
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;

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;

            // Track an Event
            var evt = new EventTelemetry("Function called");
            evt.Context.User.Id = name;
            this.telemetryClient.TrackEvent(evt);

            // Track a Metric
            var metric = new MetricTelemetry("Test Metric", DateTime.Now.Millisecond);
            metric.Context.User.Id = name;
            this.telemetryClient.TrackMetric(metric);

            // Track a Dependency
            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());
        }
    }
}

```

```

using System;
using System.Net;
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;
using Microsoft.ApplicationInsights.Extensibility;
using Microsoft.Azure.WebJobs;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;
using System.Linq;

namespace functionapp0915
{
    public static class HttpTrigger2
    {
        private static string key = TelemetryConfiguration.Active.InstrumentationKey =
            System.Environment.GetEnvironmentVariable(
                "APPINSIGHTS_INSTRUMENTATIONKEY", EnvironmentVariableTarget.Process);

        private static TelemetryClient telemetryClient =
            new TelemetryClient() { InstrumentationKey = key };

        [FunctionName("HttpTrigger2")]
        public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
            HttpRequestMessage req, ExecutionContext context, ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");
            DateTime start = DateTime.UtcNow;

            // Parse query parameter
            string name = req.GetQueryNameValuePairs()
                .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
                .Value;

            // Get request body
            dynamic data = await req.Content.ReadAsAsync<object>();

            // Set name to query string or body data
            name = name ?? data?.name;

            // Track an Event
            var evt = new EventTelemetry("Function called");
            UpdateTelemetryContext(evt.Context, context, name);
            telemetryClient.TrackEvent(evt);

            // Track a Metric
            var metric = new MetricTelemetry("Test Metric", DateTime.Now.Millisecond);
            UpdateTelemetryContext(metric.Context, context, name);
            telemetryClient.TrackMetric(metric);

            // Track a Dependency
            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
            };
            UpdateTelemetryContext(dependency.Context, context, name);
            telemetryClient.TrackDependency(dependency);
        }

        // Correlate all telemetry with the current Function invocation
        private static void UpdateTelemetryContext(TelemetryContext context, ExecutionContext

```

```

functionContext, string userName)
{
    context.Operation.Id = functionContext.InvocationId.ToString();
    context.Operation.ParentId = functionContext.InvocationId.ToString();
    context.Operation.Name = functionContext.FunctionName;
    context.User.Id = userName;
}
}
}

```

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()`). This method ensures that the telemetry has the correct correlation details for the current function invocation.

Log custom telemetry in JavaScript functions

Here is a sample code snippet that sends custom telemetry with the [Application Insights Node.js SDK](#):

```

const appInsights = require("applicationinsights");
appInsights.setup();
const client = appInsights.defaultClient;

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

    client.trackEvent({name: "my custom event", tagOverrides:{"ai.operation.id": context.invocationId},
    properties: {customProperty2: "custom property value"}});
    client.trackException({exception: new Error("handled exceptions can be logged with this method"),
    tagOverrides:{"ai.operation.id": context.invocationId}});
    client.trackMetric({name: "custom metric", value: 3, tagOverrides:{"ai.operation.id":
    context.invocationId}});
    client.trackTrace({message: "trace message", tagOverrides:{"ai.operation.id":
    context.invocationId}});
    client.trackDependency({target:"http://dbname", name:"select customers proc", data:"SELECT * FROM
    Customers", duration:231, resultCode:0, success: true, dependencyTypeName: "ZSQL", tagOverrides:
    {"ai.operation.id": context.invocationId}});
    client.trackRequest({name:"GET /customers", url:"http://myserver/customers", duration:309,
    resultCode:200, success:true, tagOverrides:{"ai.operation.id": context.invocationId}});

    context.done();
};

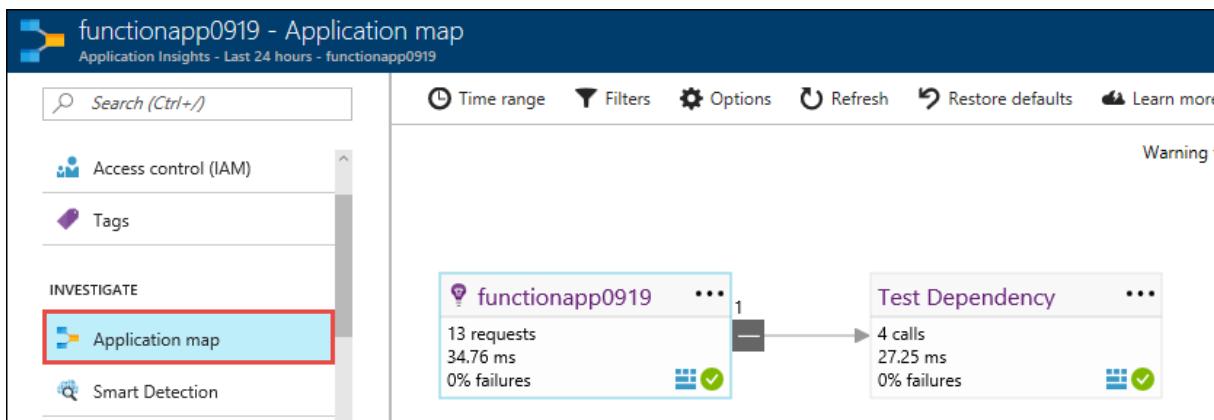
```

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.

Dependencies

Functions v2 automatically collects dependencies for HTTP requests, ServiceBus, and SQL.

You can write custom code to show the dependencies. For examples, see the sample code in the [C# custom telemetry section](#). The sample code results in an *application map* in Application Insights that looks like the following image:



Report issues

To report an issue with Application Insights integration in Functions, or to make a suggestion or request, [create an issue in GitHub](#).

Streaming Logs

While developing an application, it is often useful to see logging information in near-real time. You can view a stream of log files being generated by your functions either in the Azure portal or in a command-line session on your local computer.

This is equivalent to the output seen when you debug your functions during [local development](#). For more information, see [How to stream logs](#).

NOTE

Streaming logs support only a single instance of the Functions host. When your function is scaled to multiple instances, data from other instances are not shown in the log stream. The [Live Metrics Stream](#) in Application Insights does support multiple instances. While also in near real time, streaming analytics are also based on [sampled data](#).

Portal

To view streaming logs in the portal, select the **Platform features** tab in your function app. Then, under **Monitoring**, choose **Log streaming**.

The screenshot shows the Azure portal interface with the 'Platform features' tab selected, indicated by a red box around its title. The 'Log streaming' option under the Monitoring section is also highlighted with a red box. Other sections like General Settings, Networking, API, App Service plan, Resource management, and Development tools are visible but not selected.

General Settings	Networking	API
Function app settings	Networking	API definition
Application settings	SSL	CORS
Properties	Custom domains	
Backups	Authentication / Authorization	
All settings	Identity	
	Push notifications	
Code Deployment	Monitoring	
Deployment Center	Diagnostic logs	
Development tools	Log streaming	
Logic Apps	Process explorer	
Console (CMD / PowerShell)	Metrics	
Advanced tools (Kudu)		
App Service Editor		
Resource Explorer		
Site Extensions		

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 Log streaming interface. At the top, there are tabs for 'Overview', 'Platform features', and 'Log streaming'. Below these are buttons for 'Reconnect', 'Copy logs', 'Pause', and 'Clear'. A 'Log streaming' button with a stop icon is also present. The main area displays 'Application logs' with the following log 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'
```

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.

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 following commands to add your Azure account, choose your subscription, and stream log files:

```
Add-AzAccount  
Get-AzSubscription  
Get-AzSubscription -SubscriptionName "<subscription name>" | Select-AzSubscription  
Get-AzWebSiteLog -Name <FUNCTION_APP_NAME> -Tail
```

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

For more information, see the following resources:

- [Application Insights](#)
- [ASP.NET Core logging](#)

Buy and configure an SSL certificate for Azure App Service

7/7/2019 • 8 minutes to read • [Edit Online](#)

This tutorial shows you how to secure your [App Service app](#) or [function app](#) by creating (purchasing) an App Service certificate in [Azure Key Vault](#) and then bind it to an App Service app.

TIP

App Service Certificates can be used for any Azure or non-Azure Services and is not limited to App Services. To do so, you need to create a local PFX copy of an App Service certificate that you can use it anywhere you want. For more information, see [Creating a local PFX copy of an App Service Certificate](#).

Prerequisites

To follow this how-to guide:

- [Create an App Service app](#)
- [Map a domain name to your app](#) or [buy and configure it in Azure](#)

Prepare your web app

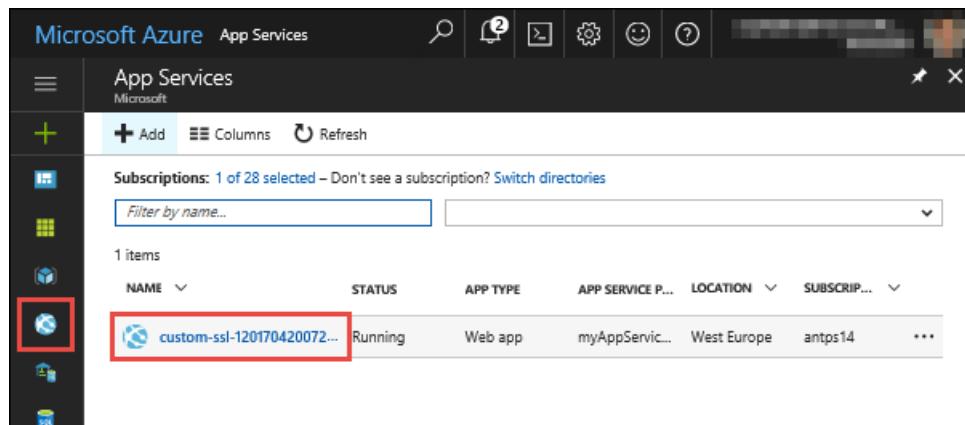
To bind a custom SSL certificate (a third-party certificate or App Service certificate) to your web app, your [App Service plan](#) must be in the **Basic**, **Standard**, **Premium**, or **Isolated** tier. In this step, you make sure that your web app is in the supported pricing tier.

Log in to Azure

Open the [Azure portal](#).

Navigate to your web app

From the left menu, click **App Services**, and then click the name of your web app.

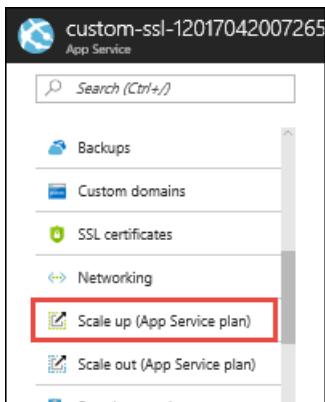


NAME	STATUS	APP TYPE	APP SERVICE P...	LOCATION	SUBSCRIP...
custom-ssl-120170420072...	Running	Web app	myAppService...	West Europe	antps14

You have landed in 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)**.



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.

A screenshot of the Azure App Service Plan scaling options page. It shows three categories: Dev / Test (F1), Production (B1, B2, B3, D1, D2, D3, D4), and Isolated. The F1 tier is highlighted with a blue box. Below the tiers, it says 'Recommended pricing tiers'. The F1 tier details are: Shared infrastructure, 1 GB memory, 60 minutes/day compute, and Free. The D1 tier details are: Shared infrastructure, 1 GB memory, 240 minutes/day compute, and 9.67 USD/Month (Estimated). The B1 tier details are: 1x cores, 1.75 GB memory, A-Series compute, and 55.80 USD/Month (Estimated). There is a 'See additional options' link. Below that, it lists 'Included hardware' with details for CPU, Memory, and Storage.

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 9.67 USD/Month (Estimated)
B1 1x cores 1.75 GB memory A-Series compute 55.80 USD/Month (Estimated)	

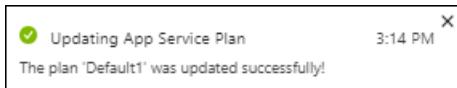
▼ See additional options

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	CPU Dedicated A-series compute resources used to run applications deployed in the...
Manual scale Up to 3 instances. Subject to availability.	Memory Memory per instance available to run applications deployed and running in th...
	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.



Start certificate order

Start an App Service certificate order in the [App Service Certificate create page](#).

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.
Naked Domain Host Name	If you specify the root domain here, you get a certificate that secures <i>both</i> the root domain and the <code>www</code> subdomain. To secure any subdomain only, specify the fully qualified domain name of the subdomain here (for example, <code>mysubdomain.contoso.com</code>).
Subscription	The datacenter where the web app is hosted.
Resource group	The resource group that contains 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.

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**.

The screenshot shows the Azure Key Vault Certificate Configuration page. On the left, there's a sidebar with various settings like Overview, Activity log, Access control (IAM), Tags, Auto Renew Settings, Timeline, Rekey and Sync, Export Certificate, Properties, Locks, Automation script, and Support + troubleshooting. The 'Certificate Configuration' section is highlighted with a red box. The main content area has three steps: Step 1: Store (highlighted with a red box), Step 2: Verify (with a green checkmark), and Step 3: Assign (with a red question mark icon). Each step has a description and a checkbox.

[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. See to create new Key Vault inside same subscription and resource group.

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 Grant several applications access to a key vault .
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 **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**.

The screenshot shows the Azure portal's left navigation bar with 'Certificate Configuration' selected. The main content area displays three steps: 'Step 1: Store' (key icon, checked), 'Step 2: Verify' (globe icon, unchecked, highlighted with a red box), and 'Step 3: Assign' (cloud icon, unchecked). A note at the top states: 'You must follow each of the steps below before you can use the certificate. Each step provides the instructions and will guide you through the necessary actions.'

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.

Bind certificate to app

In the [Azure portal](#), from the left menu, select **App Services > <your_app>**.

From the left navigation of your app, select **SSL settings > Private Certificates (.pfx) > Import App Service Certificate**.

Search (Ctrl+ /) Refresh FAQs

SSL Settings experience has been updated. Check our blog for more details →

Application settings
Authentication / Authoriza...
Application Insights
Managed service identity
Backups
Custom domains
SSL settings
Networking
Scale up (App Service plan)
Scale out (App Service plan)
WebJobs
Push
MySQL in App
Properties
Locks
Automation script

Bindings **Private Certificates (.pfx)** **Public Certificates (.cer)**

PFX Private Certificate

Private certificates (.pfx) can be used for 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 SSL. [Learn more](#)

Import App Service Certificate **Upload Certificate**

Private Certificates

Status Filter: All Healthy Warning Expired

NAME	HOSTNAME	EXPIRA...	THUMBPRINT
No private certificates available for app.			

Select the certificate that you just purchased.

Now that the certificate is imported, you need to bind it to a mapped domain name in your app. Select **Bindings** > **Add SSL Binding**.

Bindings **Private Certificates (.pfx)** **Public Certificates (.cer)**

Protocol Settings

Protocol settings are global and apply to all bindings defined by your app.

HTTPS Only: **Off** **On**
Minimum TLS Version: **1.0** **1.1** **1.2**
Incoming client certificates: **Off** **On**

SSL bindings

Bindings let you specify which certificate to use when responding to requests to a specific hostname over HTTPS. SSL Binding requires valid private certificate (.pfx) issued for the specific hostname. [Learn more](#)

Add SSL Binding

HOST NAME	PRIVATE CERTIFICATE THUMBPRINT	SSL TYPE
No SSL bindings configured for the app.		

Use the following table to help you configure the binding in the **SSL Bindings** dialog, then click **Add Binding**.

SETTING	DESCRIPTION
Hostname	The domain name to add SSL binding for.
Private Certificate Thumbprint	The certificate to bind.

SETTING	DESCRIPTION
SSL Type	<ul style="list-style-type: none"> SNI SSL - Multiple SNI-based SSL bindings may be added. This option allows multiple SSL certificates to secure multiple domains on the same IP address. Most modern browsers (including Internet Explorer, Chrome, Firefox, and Opera) support SNI (find more comprehensive browser support information at Server Name Indication). IP-based SSL - Only one IP-based SSL binding may be added. This option allows only one SSL certificate to secure a dedicated public IP address. After you configure the binding, follow the steps in Remap A record for IP SSL.

Verify HTTPS access

Visit your app using `HTTPS://<domain_name>` instead of `HTTP://<domain_name>` to verify that the certificate has been configured correctly.

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.

STATUS	LINKED PRIVATE CERTIFI...	RESOURCE GROUP	THUMBPRINT
Healthy	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.

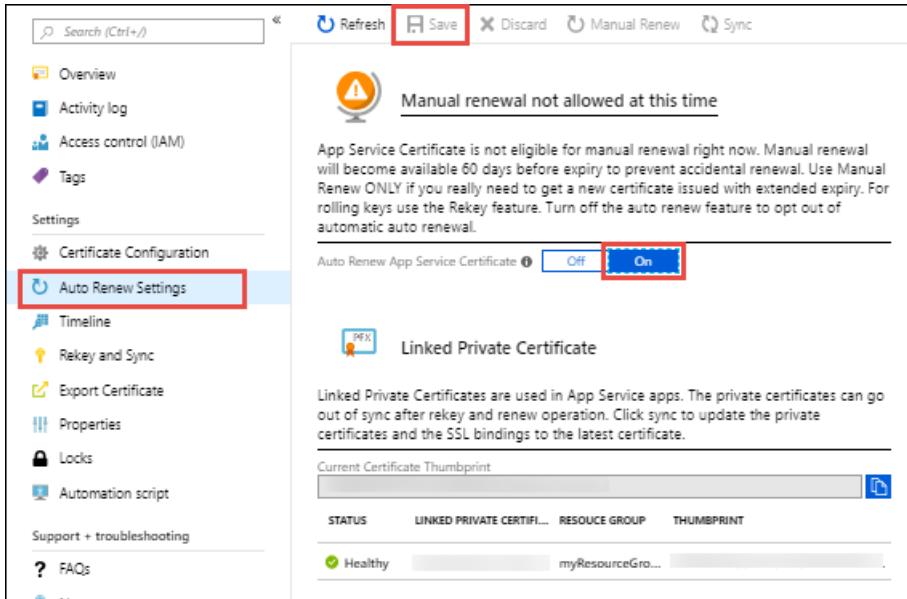
NOTE

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.

Select **On** and click **Save**. Certificates can start automatically renewing 60 days before expiration if you have automatic renewal turned 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.

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

- [Enforce HTTPS](#)
- [Enforce TLS 1.1/1.2](#)
- [Use an SSL certificate in your application code in Azure App Service](#)
- [FAQ : App Service Certificates](#)

Configure your App Service app to use Azure Active Directory sign-in

3/25/2019 • 5 minutes to read • [Edit Online](#)

NOTE

At this time, AAD V2 (including MSAL) is not supported for Azure App Services and Azure Functions. Please check back for updates.

This article shows you how to configure Azure App Services to use Azure Active Directory as an authentication provider.

Configure with express settings

1. In the [Azure portal](#), navigate to your App Service app. In the left navigation, select **Authentication / Authorization**.
2. If **Authentication / Authorization** is not enabled, select **On**.
3. Select **Azure Active Directory**, and then select **Express** under **Management Mode**.
4. Select **OK** to register the App Service app in Azure Active Directory. This creates a new app registration. If you want to choose an existing app registration instead, click **Select an existing app** and then search for the name of a previously created app registration within your tenant. Click the app registration to select it and click **OK**. Then click **OK** on the Azure Active Directory settings page. By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code.
5. (Optional) To restrict access to your site to only users authenticated by Azure Active Directory, set **Action to take when request is not authenticated** to **Log in with Azure Active Directory**. This requires that all requests be authenticated, and all unauthenticated requests are redirected to Azure Active Directory for authentication.
6. Click **Save**.

Configure with advanced settings

You can also provide configuration settings manually. This is the preferred solution if the Azure Active Directory tenant you wish to use is different from the tenant with which you sign into Azure. To complete the configuration, you must first create a registration in Azure Active Directory, and then you must provide some of the registration details to App Service.

Register your App Service app with Azure Active Directory

1. Sign in to the [Azure portal](#), and navigate to your App Service app. Copy your app **URL**. You will use this to configure your Azure Active Directory app registration.
2. Navigate to **Active Directory**, then select the **App registrations**, then click **New application registration** at the top to start a new app registration.
3. In the **Create** page, enter a **Name** for your app registration, select the **Web App / API** type, in the **Sign-on URL** box paste the application URL (from step 1). Then click to **Create**.
4. In a few seconds, you should see the new app registration you just created.
5. Once the app registration has been added, click on the app registration name, click on **Settings** at the top,

then click on **Properties**

6. In the **App ID URI** box, paste in the Application URL (from step 1), also in the **Home Page URL** paste in the Application URL (from step 1) as well, then click **Save**
7. Now click on the **Reply URLs**, edit the **Reply URL**, paste in the Application URL (from step 1), then append it to the end of the URL, `/auth/login/aad/callback` (For example, `https://contoso.azurewebsites.net/.auth/login/aad/callback`). Click **Save**.

NOTE

You can use the same app registration for multiple domains by adding additional **Reply URLs**. Make sure to model each App Service instance with its own registration, so it has its own permissions and consent. Also consider using separate app registrations for separate site slots. This is to avoid permissions being shared between environments, so that a bug in new code you are testing does not affect production.

8. At this point, copy the **Application ID** for the app. Keep it for later use. You will need it to configure your App Service app.
9. Close the **Registered app** page. On the **App registrations** page, click on the **Endpoints** button at the top, then copy the **WS-FEDERATION SIGN-ON ENDPOINT** URL but remove the `/wsfed` ending from the URL. The end result should look like `https://login.microsoftonline.com/00000000-0000-0000-0000-000000000000`. The domain name may be different for a sovereign cloud. This will serve as the Issuer URL for later.

Add Azure Active Directory information to your App Service app

1. Back in the [Azure portal](#), navigate to your App Service app. Click **Authentication/Authorization**. If the Authentication/Authorization feature is not enabled, turn the switch to **On**. Click on **Azure Active Directory**, under Authentication Providers, to configure your app.

(Optional) By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code. Set **Action to take when request is not authenticated** to **Log in with Azure Active Directory**. This option requires that all requests be authenticated, and all unauthenticated requests are redirected to Azure Active Directory for authentication.

2. In the Active Directory Authentication configuration, click **Advanced** under **Management Mode**. Paste the Application ID into the Client ID box (from step 8) and paste in the URL (from step 9) into the Issuer URL value. Then click **OK**.
3. On the Active Directory Authentication configuration page, click **Save**.

You are now ready to use Azure Active Directory for authentication in your App Service app.

Configure a native client application

You can register native clients, which provides greater control over permissions mapping. You need this if you wish to perform sign-ins using a client library such as the **Active Directory Authentication Library**.

1. Navigate to **Azure Active Directory** in the [Azure portal](#).
2. In the left navigation, select **App registrations**. Click **New app registration** at the top.
3. In the **Create** page, enter a **Name** for your app registration. Select **Native** in **Application type**.
4. In the **Redirect URI** box, enter your site's `/auth/login/done` endpoint, using the HTTPS scheme. This value should be similar to <https://contoso.azurewebsites.net/.auth/login/done>. If creating a Windows application, instead use the [package SID](#) as the URI.
5. Click **Create**.

6. Once the app registration has been added, select it to open it. Find the **Application ID** and make a note of this value.
7. Click **All settings > Required permissions > Add > Select an API**.
8. Type the name of the App Service app that you registered earlier to search for it, then select it and click **Select**.
9. Select **Access <app_name>**. Then click **Select**. Then click **Done**.

You have now configured a native client application that can access your App Service app.

Related Content

- [App Service Authentication / Authorization overview](#).
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

How to configure your App Service application to use Facebook login

6/6/2019 • 2 minutes to read • [Edit Online](#)

This topic shows you how to configure Azure App Service to use Facebook as an authentication provider.

To complete the procedure in this topic, you must have 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. Navigate to the [Facebook Developers](#) website and sign-in with your Facebook account credentials.
2. (Optional) If you don't have a Facebook for Developers account, click **Get Started** and follow the registration steps.
3. Click **My Apps > Add New App**.
4. In **Display Name**, type a unique name for your app. Also provide your **Contact Email**, and then click **Create App ID** and complete the security check. This takes you to the developer dashboard for your new Facebook app.
5. Click **Dashboard > Facebook Login > Set up > Web**.
6. In the left-hand navigation under **Facebook Login**, click **Settings**.
7. In **Valid OAuth redirect URIs**, type `https://<app-name>.azurewebsites.net/.auth/login/facebook/callback` and replace `<app-name>` with the name of your Azure App Service app. Click **Save Changes**.
8. In the left-hand navigation, click **Settings > Basic**. On the **App Secret** field, click **Show**. Copy the values of **App ID** and **App Secret**. You use these later to configure your App Service app in Azure.

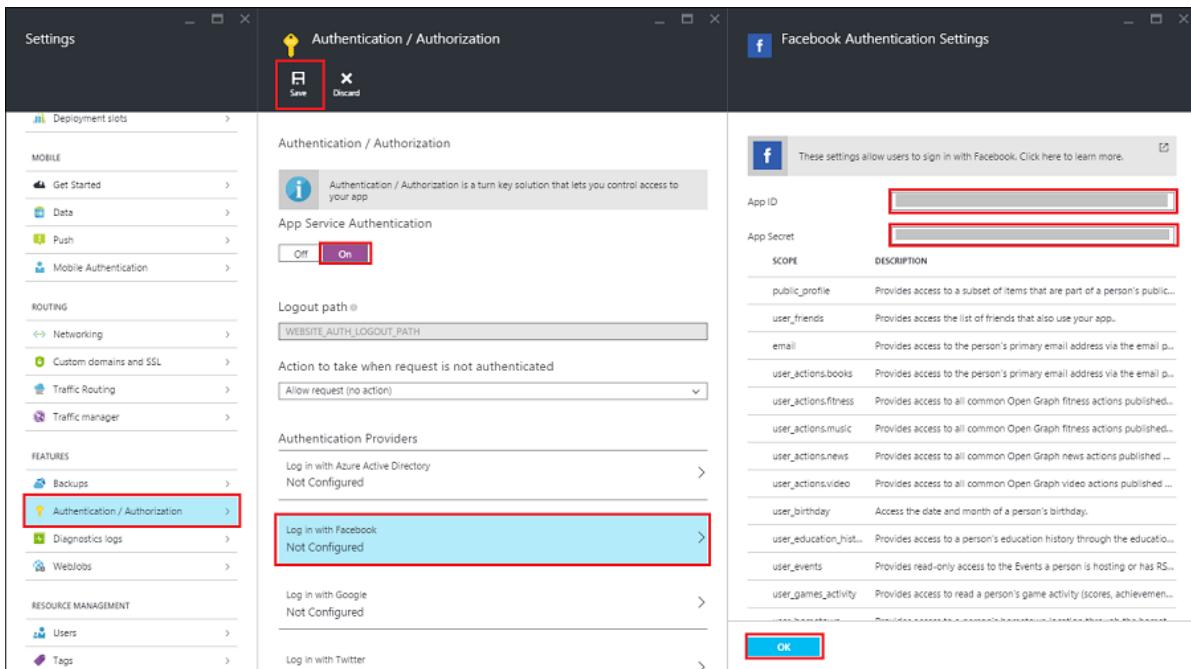
IMPORTANT

The app secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

9. The Facebook account which was used to register the application is an administrator of the app. At this point, only administrators can sign into this application. To authenticate other Facebook accounts, click **App Review** and enable **Make <your-app-name> public** to enable general public access using Facebook authentication.

Add Facebook information to your application

1. Sign in to the [Azure portal](#) and navigate to your App Service app. Click **Settings > Authentication / Authorization**, and make sure that **App Service Authentication** is **On**.
2. Click **Facebook**, paste in the App ID and App Secret values which you obtained previously, optionally enable any scopes needed by your application, then click **OK**.



By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code.

3. (Optional) To restrict access to your site to only users authenticated by Facebook, set **Action to take when request is not authenticated** to **Facebook**. This requires that all requests be authenticated, and all unauthenticated requests are redirected to Facebook for authentication.
4. When done configuring authentication, click **Save**.

You are now ready to use Facebook for authentication in your app.

Related Content

- [App Service Authentication / Authorization overview](#).
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

How to configure your App Service application to use Google login

12/14/2018 • 2 minutes to read • [Edit Online](#)

This topic shows you how to configure Azure App Service 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

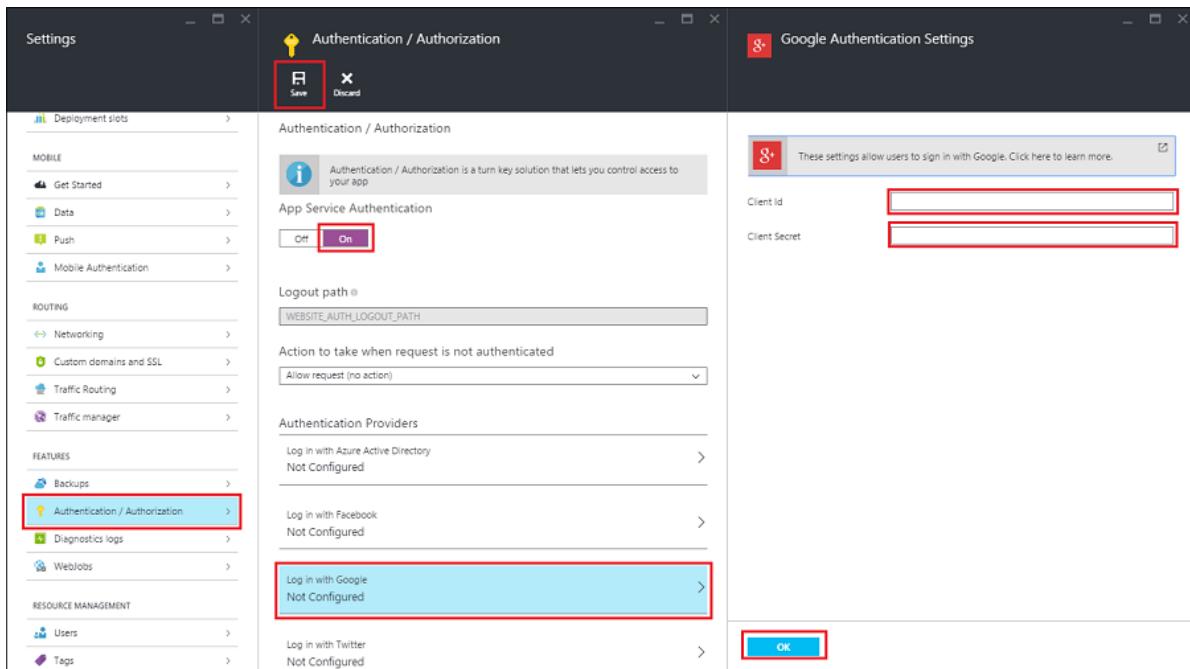
1. Log on to the [Azure portal](#), and navigate to your application. Copy your **URL**, which you use later to configure your Google app.
2. Navigate to the [Google apis](#) website, sign in with your Google account credentials, click **Create Project**, provide a **Project name**, then click **Create**.
3. Once the project is created, select it. From the project dashboard, click **Go to APIs overview**.
4. Select **Enable APIs and services**. Search for **Google+ API**, and select it. Then click **Enable**.
5. In the left navigation, **Credentials > OAuth consent screen**, then select your **Email address**, enter a **Product Name**, and click **Save**.
6. In the **Credentials** tab, click **Create credentials > OAuth client ID**.
7. On the "Create client ID" screen, select **Web application**.
8. Paste the App Service **URL** you copied earlier into **Authorized JavaScript Origins**, then paste your redirect URI into **Authorized Redirect URI**. The redirect URI is the URL of your application appended with the path, `/auth/login/google/callback`. For example,
`https://contoso.azurewebsites.net/.auth/login/google/callback`. Make sure that you are using the HTTPS scheme. Then click **Create**.
9. On the next screen, make a note of the values of the client ID and client secret.

IMPORTANT

The client 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. Back in the [Azure portal](#), navigate to your application. Click **Settings**, and then **Authentication / Authorization**.
2. If the Authentication / Authorization feature is not enabled, turn the switch to **On**.
3. Click **Google**. Paste in the App ID and App Secret values which you obtained previously, and optionally enable any scopes your application requires. Then click **OK**.



By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code.

4. (Optional) To restrict access to your site to only users authenticated by Google, set **Action to take when request is not authenticated** to **Google**. This requires that all requests be authenticated, and all unauthenticated requests are redirected to Google for authentication.

5. Click **Save**.

You are now ready to use Google for authentication in your app.

Related Content

- [App Service Authentication / Authorization overview](#).
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

How to configure your App Service application to use Microsoft Account login

7/26/2019 • 2 minutes to read • [Edit Online](#)

This topic shows you how to configure Azure App Service to use Microsoft Account as an authentication provider.

Register your app with Microsoft Account

1. Log on to the [Azure portal](#), and navigate to your application. Copy your **URL**, which later you use to configure your app with Microsoft Account.
2. Navigate to [App registrations](#), and sign in with your Microsoft account, if requested.
3. Click **Add an app**, then type an application name, and click **Create**.
4. Make a note of the **Application ID**, as you will need it later.
5. Under "Platforms," click **Add Platform** and select "Web".
6. Under "Redirect URIs" supply the endpoint for your application, then click **Save**.

NOTE

Your redirect URI is the URL of your application appended with the path, `/auth/login/microsoftaccount/callback`.

For example, `https://contoso.azurewebsites.net/.auth/login/microsoftaccount/callback`.

Make sure that you are using the HTTPS scheme.

7. Under "Application Secrets," click **Generate New Password**. Make note of the value that appears. Once you leave the page, it will not be displayed again.

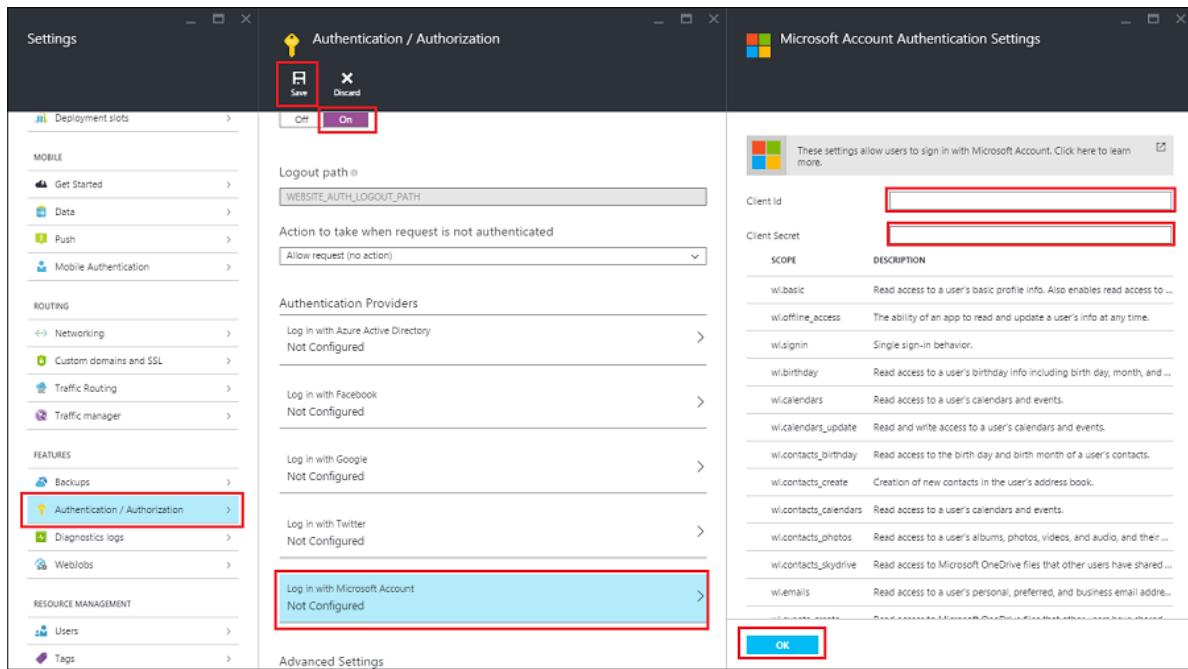
IMPORTANT

The password is an important security credential. Do not share the password with anyone or distribute it within a client application.

8. Click **Save**

Add Microsoft Account information to your App Service application

1. Back in the [Azure portal](#), navigate to your application, click **Settings > Authentication / Authorization**.
2. If the Authentication / Authorization feature is not enabled, switch it **On**.
3. Click **Microsoft Account**. Paste in the Application ID and Password values which you obtained previously, and optionally enable any scopes your application requires. Then click **OK**.



By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code.

4. (Optional) To restrict access to your site to only users authenticated by Microsoft account, set **Action to take when request is not authenticated** to **Microsoft Account**. This requires that all requests be authenticated, and all unauthenticated requests are redirected to Microsoft account for authentication.

5. Click **Save**.

You are now ready to use Microsoft Account for authentication in your app.

Related content

- [App Service Authentication / Authorization overview](#).
- Add authentication to your Mobile App: [iOS](#), [Android](#), [Windows Universal](#), [Xamarin.Android](#), [Xamarin.iOS](#), [Xamarin.Forms](#), [Cordova](#).

How to configure your App Service application to use Twitter login

12/14/2018 • 2 minutes to read • [Edit Online](#)

This topic shows you how to configure Azure App Service to use Twitter as an authentication provider.

To complete the procedure in this topic, you must have 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

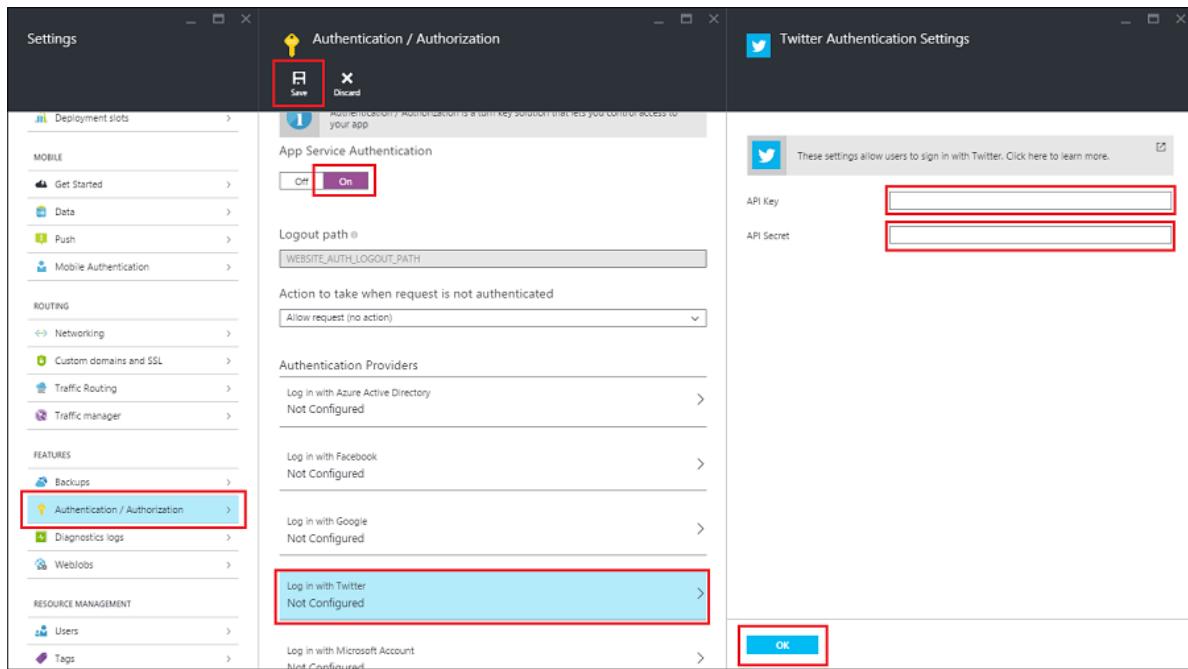
1. Log on to the [Azure portal](#), and navigate to your application. Copy your **URL**. You will use this to configure your Twitter app.
2. Navigate to the [Twitter Developers](#) website, sign in with your Twitter account credentials, and click **Create New App**.
3. Type in the **Name** and a **Description** for your new app. Paste in your application's **URL** for the **Website** value. Then, for the **Callback URL**, paste the **Callback URL** you copied earlier. This is your Mobile App gateway appended with the path, `/auth/login/twitter/callback`. For example,
`https://contoso.azurewebsites.net/.auth/login/twitter/callback`. Make sure that you are using the HTTPS scheme.
4. At the bottom of the page, read and accept the terms. Then click **Create your Twitter application**. This registers the app and displays the application details.
5. Click the **Settings** tab, check **Allow this application to be used to sign in with Twitter**, then click **Update Settings**.
6. Select the **Keys and Access Tokens** tab. Make a note of the values of **Consumer Key (API Key)** and **Consumer secret (API Secret)**.

NOTE

The consumer secret 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. Back in the [Azure portal](#), navigate to your application. Click **Settings**, and then **Authentication / Authorization**.
2. If the Authentication / Authorization feature is not enabled, turn the switch to **On**.
3. Click **Twitter**. Paste in the App ID and App Secret values which you obtained previously. Then click **OK**.



By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code.

4. (Optional) To restrict access to your site to only users authenticated by Twitter, set **Action to take when request is not authenticated** to **Twitter**. This requires that all requests be authenticated, and all unauthenticated requests are redirected to Twitter for authentication.

5. Click **Save**.

You are now ready to use Twitter for authentication in your app.

Related Content

- [App Service Authentication / Authorization overview](#).
- 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

7/12/2019 • 8 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 \(Windows\)](#)
- [Tutorial: Authenticate and authorize users end-to-end in Azure App Service for Linux](#)
- [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](#)

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>
```

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
```

You must 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.

Your application 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	<code>X-MS-TOKEN-AAD-ID-TOKEN</code> <code>X-MS-TOKEN-AAD-ACCESS-TOKEN</code> <code>X-MS-TOKEN-AAD-EXPIRES-ON</code> <code>X-MS-TOKEN-AAD-REFRESH-TOKEN</code>
Facebook Token	<code>X-MS-TOKEN-FACEBOOK-ACCESS-TOKEN</code> <code>X-MS-TOKEN-FACEBOOK-EXPIRES-ON</code>
Google	<code>X-MS-TOKEN-GOOGLE-ID-TOKEN</code> <code>X-MS-TOKEN-GOOGLE-ACCESS-TOKEN</code> <code>X-MS-TOKEN-GOOGLE-EXPIRES-ON</code> <code>X-MS-TOKEN-GOOGLE-REFRESH-TOKEN</code>
Microsoft Account	<code>X-MS-TOKEN-MICROSOFTACCOUNT-ACCESS-TOKEN</code> <code>X-MS-TOKEN-MICROSOFTACCOUNT-EXPIRES-ON</code> <code>X-MS-TOKEN-MICROSOFTACCOUNT-AUTHENTICATION-TOKEN</code> <code>X-MS-TOKEN-MICROSOFTACCOUNT-REFRESH-TOKEN</code>
Twitter	<code>X-MS-TOKEN-TWITTER-ACCESS-TOKEN</code> <code>X-MS-TOKEN-TWITTER-ACCESS-TOKEN-SECRET</code>

From your client code (such as a mobile app or in-browser JavaScript), send an HTTP `GET` request to `/auth/me`. 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 anytime, just call `/.auth/refresh` in any language. The following snippet uses jQuery to refresh your access tokens from a JavaScript client.

```
function refreshTokens() {
  let refreshToken = "/.auth/refresh";
  $.ajax(refreshToken) .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 Active Directory allows any number of custom domains for the sign-in accounts. This behavior may be undesirable for an internal app, which you don't want anyone with an *outlook.com* account to access. To limit 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>"]
```

Next steps

[Tutorial: Authenticate and authorize users end-to-end \(Windows\)](#) [Tutorial: Authenticate and authorize users end-to-end \(Linux\)](#)

Azure App Service Access Restrictions

7/9/2019 • 5 minutes to read • [Edit Online](#)

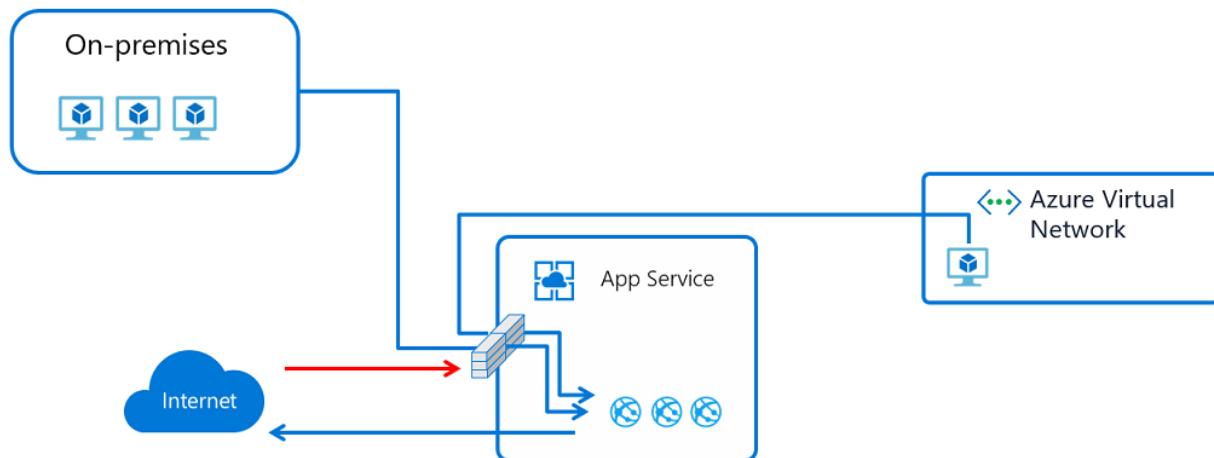
Access Restrictions enable you to define a priority ordered allow/deny list that controls network access to your app. The list can include IP addresses or Azure Virtual Network subnets. When there are one or more entries, there is then an implicit "deny all" that exists at the end of the list.

The Access Restrictions capability works with all App Service hosted work loads including; 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 restrictions list. If the FROM address is in a subnet that is configured with service endpoints to Microsoft.Web, then the source subnet is compared against the virtual network rules in your access restrictions list. If the address is not allowed access based on the rules in the list, the service replies with an [HTTP 403](#) status code.

The access restrictions 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 ACLs.

The ability to restrict access to your web app from an Azure Virtual Network (VNet) is called [service endpoints](#). Service endpoints enable you to restrict access to a multi-tenant service from selected subnets. It must be enabled on both the networking side as well as the service that it is being enabled with. It does not work to restrict traffic to apps that are hosted in an App Service Environment. If you are in an App Service Environment, you can control access to your app with IP address rules.



Adding and editing Access Restriction rules in the portal

To add an access restriction rule to your app, use the menu to open **Network>Access Restrictions** and click on **Configure Access Restrictions**

The screenshot shows the Azure App Service Networking blade for the 'vnet-integration-app'. The left sidebar has 'Networking' selected. The main area displays sections for VNet Integration, Hybrid connections, Azure CDN, and Access Restrictions. The 'Access Restrictions' section is expanded, showing a table of current rules.

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-s... Enabled		Allow
2147483647	Deny all	Any		Deny

From the Access Restrictions UI, you can review the list of access restriction rules defined for your app.

The screenshot shows the 'Access Restrictions' blade for the 'vnet-integration-app'. It includes a summary, a search bar, and a table of rules. The table columns are Priority, Name, Source, Endpoint Status, and Action.

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-s... Enabled		Allow
2147483647	Deny all	Any		Deny

The list will show all of the current restrictions that are on your app. If you have a VNet restriction on your app, the table will show if service endpoints are enabled for Microsoft.Web. When there are no defined restrictions on your app, your app will be accessible from anywhere.

Adding IP address rules

You can click on **[+] Add** to add a new access restriction rule. Once you add a rule, it will become effective immediately. Rules are enforced in priority order starting from the lowest number and going up. There is an implicit deny all that is in effect once you add even a single rule.

When creating a rule, you must select allow/deny and also the type of rule. You are also required to provide the priority value and what you are restricting access to. You can optionally add a name, and description to the rule.

Add IP Restriction

Name *

Action
 Allow Deny

* Priority

Description

Type

* Subscription

* Virtual Network

* Subnet

Add rule

To set an IP address based rule, select a type of IPv4 or IPv6. IP Address notation must be specified in CIDR notation for both IPv4 and IPv6 addresses. To specify an exact 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, you can read [Classless Inter-Domain Routing](#).

Service endpoints

Service endpoints enables you to restrict access to selected Azure virtual network subnets. To restrict access to a specific subnet, create a restriction rule with a type of Virtual Network. You can pick the subscription, VNet, and subnet you wish to allow or deny access with. If service endpoints are not already enabled with Microsoft.Web for the subnet that you selected, it will automatically be enabled for you unless you check the box asking not to do that. The situation where you would want to enable it on the app but not the subnet is largely related to if you have the permissions to enable service endpoints on the subnet or not. If you need to get somebody else to enable service endpoints on the subnet, you can check the box and have your app configured for service endpoints in anticipation of it being enabled later on the subnet.

Add IP Restriction

Name ⓘ

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

Service endpoints cannot be used 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 your app with IP access rules.

With service endpoints, you can configure your app with Application Gateways or other WAF devices. You can also configure multi-tier applications with secure backends. For more details on some of the possibilities, read [Networking features and App Service](#).

Managing access restriction rules

You can click on any row to edit an existing access restriction rule. Edits are effective immediately including changes in priority ordering.

Edit IP Restriction

Name

* Priority

Action
 Allow Deny

Description

* Subscription

* Virtual Network

* Subnet

When you edit a rule, you cannot change the type between an IP address rule and a Virtual Network rule.

Edit IP Restriction

Name

* Priority

Action
 Allow Deny

Description

* Subscription

* Virtual Network

* Subnet

To delete a rule, click the ... on your rule and then click **remove**.

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. [Learn more](#)

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

Blocking a single IP Address

When adding your first IP Restriction rule, the service will add an explicit **deny all** rule with a priority of 2147483647. In practice, the explicit **deny all** rule will be last rule executed and will block access to any IP address that is not explicitly allowed using an **Allow** rule.

For the scenario where users want to explicitly block a single IP address or IP address block, but allow everything else access, it is necessary to add an explicit **Allow All** rule.

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. [Learn more](#)

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

SCM site

In addition to being able to control access to your app, you can also restrict access to the scm site used by your app. The scm site is the web deploy endpoint and also the Kudu console. You can separately assign access restrictions to the scm site from the app or use the same set for both the app and the scm site. When you check the box to have the same restrictions as your app, everything is blanked out. If you uncheck the box, whatever settings you had earlier on the scm site are applied.

vnet-integration-app.azurewebsites.net vnet-integration-app.scm.azurewebsites.net

Same restrictions as vnet-integration-app.azurewebsites.net

[+ Add rule](#)

PRIORITY	NAME	SOURCE	ENDPOINT STATUS	ACTION
1	Allow all	Any		Allow

Programmatic manipulation of access restriction rules

There currently is no CLI or PowerShell for the new Access Restrictions capability but the values can be set manually with an [Azure REST API](#) PUT operation on the app configuration in Resource Manager. As an example, you can use resources.azure.com and edit the ipSecurityRestrictions block to add the required JSON.

The location for this information in 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`

The JSON syntax for the earlier example is:

```
{  
  "properties": {  
    "ipSecurityRestrictions": [  
      {  
        "ipAddress": "122.133.144.0/24",  
        "action": "Allow",  
        "tag": "Default",  
        "priority": 100,  
        "name": "IP example rule"  
      }  
    ]  
  }  
}
```

Function App IP Restrictions

IP restrictions are available for both Function Apps with the same functionality as App Service plans. Enabling IP restrictions will disable the portal code editor for any disallowed IPs.

[Learn more here](#)

How to use managed identities for App Service and Azure Functions

7/17/2019 • 12 minutes to read • [Edit Online](#)

NOTE

Managed identity support for App Service on Linux and Web App for Containers is currently in preview.

IMPORTANT

Managed identities for App Service and Azure Functions will not behave as expected if your app is migrated across subscriptions/tenants. The app will need to obtain a new identity, which can be done by disabling and re-enabling the feature. See [Removing an identity](#) below. Downstream resources will also need to have access policies updated to use the new identity.

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. A managed identity from Azure Active Directory allows your app to easily access other AAD-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 AAD, 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. System-assigned identity support is generally available for Windows apps.
- A **user-assigned identity** is a standalone Azure resource which can be assigned to your app. An app can have multiple user-assigned identities. User-assigned identity support is in preview for all app types.

Adding 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 **Managed identity**.
4. Within the **System assigned** tab, switch **Status** to **On**. Click **Save**.

A system assigned managed identity enables Azure resources to authenticate to cloud services (e.g. Azure Key lifecycle of this type of managed identity is tied to the lifecycle of this resource. Additionally, each resource (e.g.

System assigned User assigned (preview)

Save Discard Refresh

Status: On

Object ID: 7283a4ee-ac06-4f67-b8e7-513d24f010d1

This resource is registered with Azure Active Directory. You can control its access to services like Az

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 is 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 a web app and assigning it an identity using Azure PowerShell:

1. If needed, install the Azure PowerShell using the instruction 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 myResourceGroup -Location $location  
  
# Create an App Service plan in Free tier.  
New-AzAppServicePlan -Name $webappname -Location $location -ResourceGroupName myResourceGroup -Tier Free  
  
# Create a web app.  
New-AzWebApp -Name $webappname -Location $location -AppServicePlan $webappname -ResourceGroupName myResourceGroup
```

3. Run the `Set-AzWebApp -AssignIdentity` command to create the identity for this application:

```
Set-AzWebApp -AssignIdentity $true -Name $webappname -ResourceGroupName myResourceGroup
```

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>"
}
```

Where `<TENANTID>` and `<PRINCIPALID>` are replaced with GUIDs. The tenantId property identifies what AAD tenant the identity belongs to. The principalId is a unique identifier for the application's new identity. Within AAD, the service principal has the same name that you gave to your App Service or Azure Functions instance.

Adding a user-assigned identity (preview)

NOTE

User-assigned identities are currently in preview. Sovereign clouds are not yet supported.

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

NOTE

This portal experience is being deployed and may not yet be available in all regions.

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 **Managed identity**.

5. Within the **User assigned (preview)** 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 icons and a search bar. The main area is titled 'userassigned-windows - Identity' and shows the 'Identity' section. A note explains that user-assigned identities enable authentication without storing credentials. Below this, there are tabs for 'System assigned' and 'User assigned (preview)', with 'User assigned (preview)' being the active tab. There are buttons for 'Add', 'Remove', and 'Refresh'. A table lists the assigned identities, showing one entry: 'myUserAssignedIdentity' under the 'hendokvf' resource group.

NAME	RESOURCE GROUP
myUserAssignedIdentity	hendokvf

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 and a cotells 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": "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>"
        }
    }
}
```

Where `<PRINCIPALID>` and `<CLIENTID>` are replaced with GUIDs. The principalId is a unique identifier for the identity which is used for AAD administration. The clientId is a unique identifier for the application's new identity that is used for specifying which identity to use during runtime calls.

Obtaining tokens for Azure resources

An app can use its identity to get tokens to other resources protected by AAD, such as Azure Key Vault. These tokens represent the application accessing the resource, and not any specific user of the application.

IMPORTANT

You may need to configure the target resource to allow access from your application. For example, if you request a token to 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](#).

There is a simple REST protocol for obtaining a token in App Service and Azure Functions. For .NET applications, the Microsoft.Azure.Services.AppAuthentication library provides an abstraction over this protocol and supports a local development experience.

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. For more on local 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));
```

To learn more about Microsoft.Azure.Services.AppAuthentication 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);
```

Using the REST protocol

An app with a managed identity has two environment variables defined:

- **MSI_ENDPOINT** - the URL to the local token service.
- **MSI_SECRET** - a header used to help mitigate server-side request forgery (SSRF) attacks. The value is rotated by the platform.

The **MSI_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 AAD 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. "2017-09-01" is currently the only version supported.
secret	Header	The value of the MSI_SECRET environment variable. This header is used to help mitigate server-side request forgery (SSRF) attacks.
clientid	Query	(Optional) The ID of the user-assigned identity to be used. If omitted, the system-assigned identity is used.

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.
expires_on	The time when the access token expires. The date is represented as the number of seconds from 1970-01-01T0:0:0Z UTC until the expiration time. This value is used to determine the lifetime of cached tokens.
resource	The App ID URI of the receiving web service.
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 AAD service-to-service access token request](#).

NOTE

Environment variables are set up when the process first starts, so after enabling a managed identity for your application, you may need to restart your application, or redeploy its code, before `MSI_ENDPOINT` and `MSI_SECRET` are available to your code.

REST protocol examples

An example request might look like the following:

```
GET /MSI/token?resource=https://vault.azure.net&api-version=2017-09-01 HTTP/1.1
Host: localhost:4141
Secret: 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": "09/14/2017 00:00:00 PM +00:00",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer"
}
```

Code examples

To make this request in C#:

```
public static async Task<HttpResponseMessage> GetToken(string resource, string apiversion)  {
    HttpClient client = new HttpClient();
    client.DefaultRequestHeaders.Add("Secret", Environment.GetEnvironmentVariable("MSI_SECRET"));
    return await client.GetAsync(String.Format("{0}/?resource={1}&api-version={2}",
    Environment.GetEnvironmentVariable("MSI_ENDPOINT"), resource, apiversion));
}
```

TIP

For .NET languages, you can also use [Microsoft.Azure.Services.AppAuthentication](#) instead of crafting this request yourself.

In NodeJS:

```
const rp = require('request-promise');
const getToken = function(resource, apiver, cb) {
    let options = {
        uri: `${process.env["MSI_ENDPOINT"]}/?resource=${resource}&api-version=${apiver}`,
        headers: {
            'Secret': process.env["MSI_SECRET"]
        }
    };
    rp(options)
        .then(cb);
}
```

In PowerShell:

```
$apiVersion = "2017-09-01"
$resourceURI = "https://<AAD-resource-URI-for-resource-to-obtain-token>"
$tokenAuthURI = $env:MSI_ENDPOINT + "?resource=$resourceURI&api-version=$apiVersion"
$tokenResponse = Invoke-RestMethod -Method Get -Headers @{"Secret"="$env:MSI_SECRET"} -Uri $tokenAuthURI
$accessToken = $tokenResponse.access_token
```

Removing 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, in the REST/ARM template protocol, this is done by setting the type to "None":

```
"identity": {  
    "type": "None"  
}
```

Removing a system-assigned identity in this way will also delete it from AAD. System-assigned identities are also automatically removed from AAD when the app resource is deleted.

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 (preview)

7/9/2019 • 3 minutes to read • [Edit Online](#)

NOTE

Key Vault references are currently in preview.

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.

Granting access to an application identity in key vault is a onetime operation, and it will remain same for all Azure subscriptions. You can use it to deploy as many certificates as you want.

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

In the current preview, versions are 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 psuedo-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"  
            }  
        }  
    ]  
}
```

```

    ,
    //...
    "resources": [
        {
            "type": "config",
            "name": "appsettings",
            //...
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites', variables('functionAppName'))]",
                "[resourceId('Microsoft.KeyVault/vaults/', variables('keyVaultName'))]",
                "[resourceId('Microsoft.KeyVault/vaults/secrets', variables('keyVaultName'), variables('storageConnectionStringName'))]",
                "[resourceId('Microsoft.KeyVault/vaults/secrets', variables('keyVaultName'), variables('appInsightsKeyName'))]"
            ],
            "properties": {
                "AzureWebJobsStorage": "[concat('@Microsoft.KeyVault(SecretUri=',
reference(variables('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" ]
                    }
                }
            ]
        },
        "resources": [
            {
                "type": "secrets",
                "name": "[variables('storageConnectionStringName')]",
                //...
                "dependsOn": [
                    "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
                    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]"
                ],
                "properties": {
                    "value": "[concat('DefaultEndpointsProtocol=https://', AccountName, '.')

```

```
        value : "[concat(deploymentEndpointProtocolHttps,accountName) ,  
variables('storageAccountName'),';AccountKey=' , listKeys(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.

Use Azure Functions to connect to an Azure SQL Database

5/17/2019 • 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 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. 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](#).

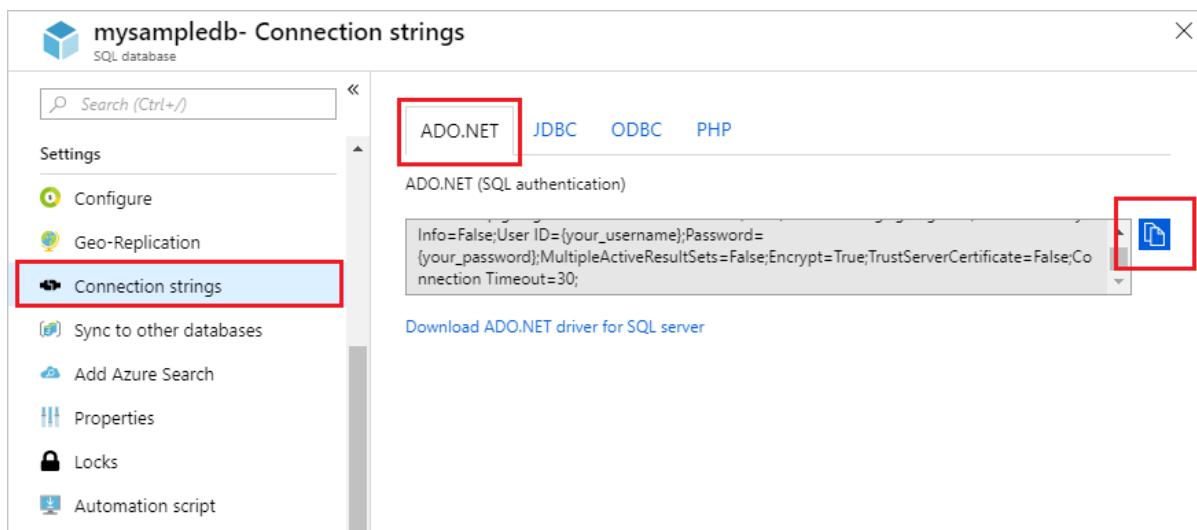
Prerequisites

- Complete the steps in the article [Create your first function using Visual Studio](#) to create a local function app that targets the version 2.x 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 an Azure SQL database in 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 an Azure SQL database in 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.

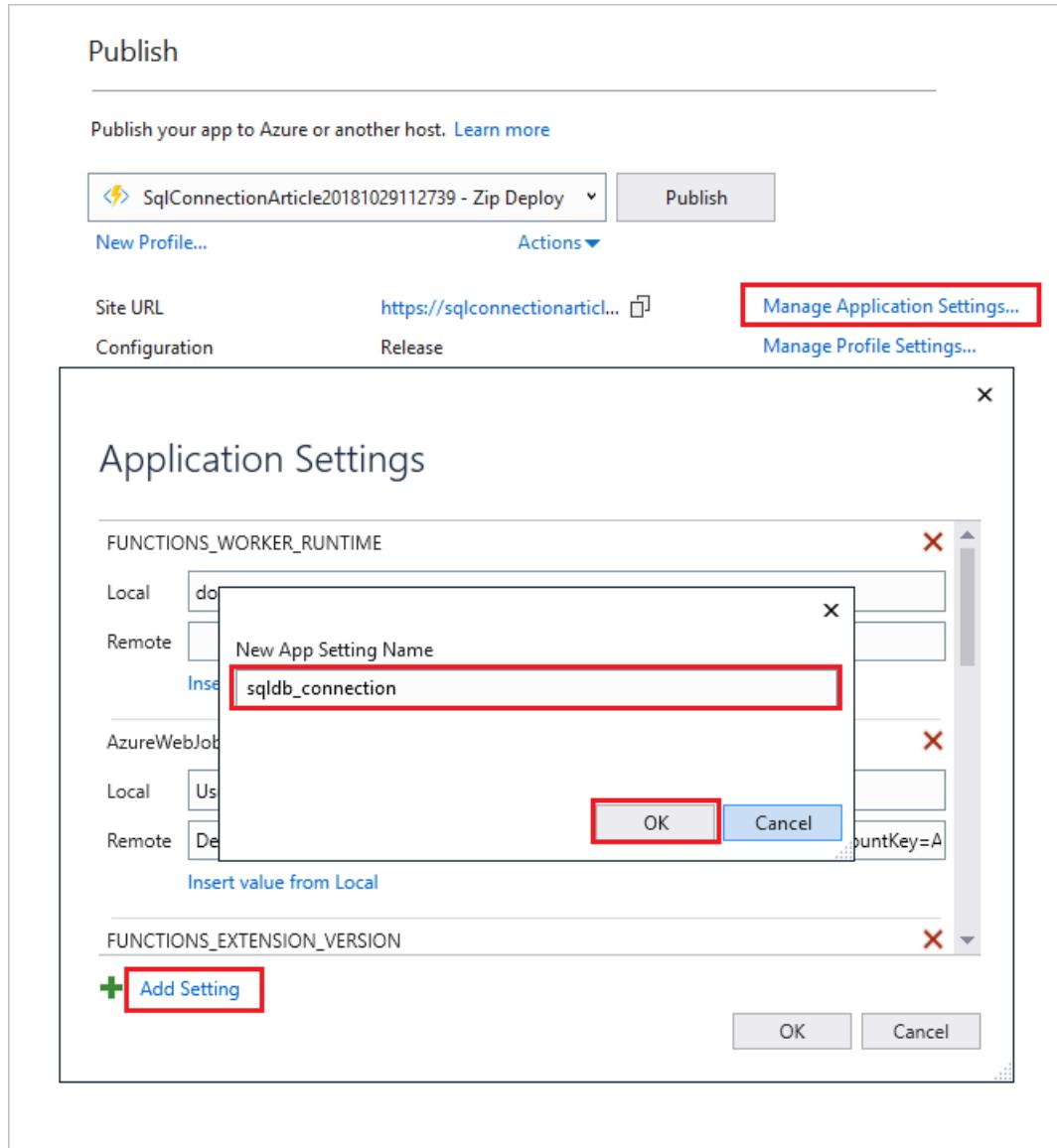


Set the connection string

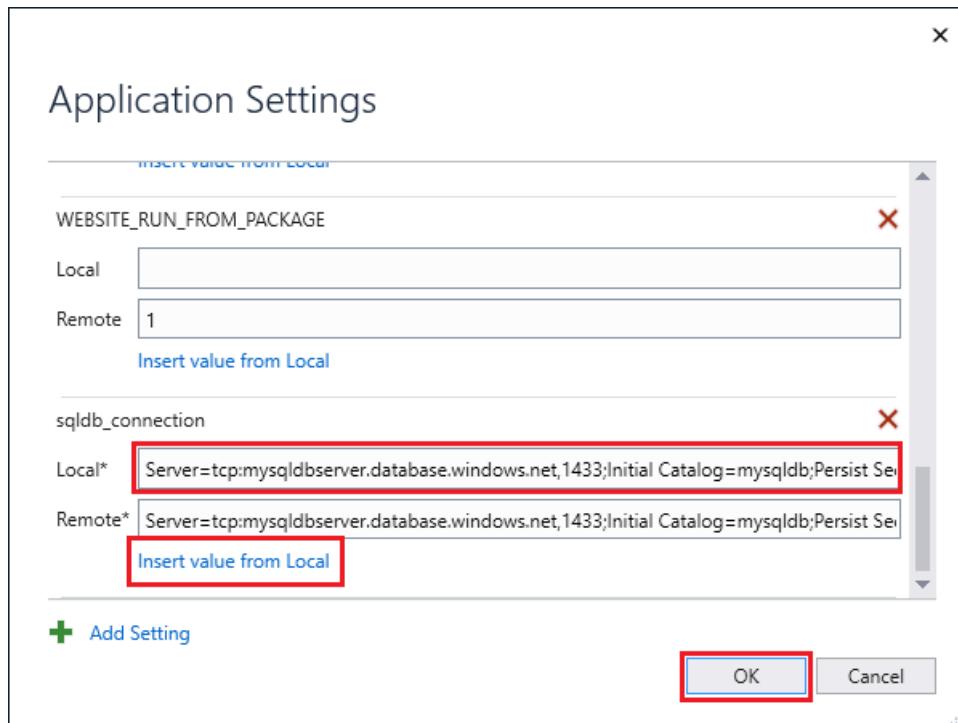
A function app hosts the execution of your functions in Azure. As a best security practice, store connection strings and other secrets in your function app settings. Using application settings prevents accidental disclosure of the connection string with your code. You can access app settings for your 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 > Manage application settings....** Select **Add setting**, in **New app setting name**, type `sqldb_connection`, and select **OK**.



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 a 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;
```

- 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](#).

- Press **F5** to start the function app. The [Azure Functions Core Tools](#) execution window opens behind Visual Studio.
- 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.

2 minutes to read

2 minutes to read

Exporting an Azure-hosted API to PowerApps and Microsoft Flow

1/18/2019 • 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. [Microsoft Flow](#) 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 topic 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.

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 Azure Functions or another App Service application.

If using Azure Functions, select your function app, choose **Platform features**, and then **API definition**.

The screenshot shows the Azure portal interface for a function app named "function-demo-energy". The left sidebar lists "Functions", "Proxies (preview)", and "Slots (preview)". The main area is titled "Platform features" and contains several sections: "GENERAL SETTINGS" (Function app settings, Application settings, Properties, Backups, All settings), "NETWORKING" (Networking, SSL, Custom domains, Authentication / Authorization, Push notifications), "API" (API definition, CORS), "APP SERVICE PLAN" (App Service plan, Quotas), "CODE DEPLOYMENT" (Deployment options, Deployment credentials), "MONITORING" (Diagnostic logs, Log streaming, Process explorer, Security scanning), "DEVELOPMENT TOOLS" (Console, Advanced tools (Kudu), App Service Editor, Resource Explorer, Extensions), and "RESOURCE MANAGEMENT" (Activity log, Access control (IAM), Tags, Locks, Automation script). The "API definition" item under the API section is highlighted with a red box.

If using Azure App Service, select **API definition** from the settings list.

The screenshot shows the Azure portal interface for an app service named "app-demo-energy". The left sidebar lists "Resource explorer", "Testing in production", and "Extensions". The main area is titled "API" and contains "API definition" and "CORS". The "API definition" item is highlighted with a red box.

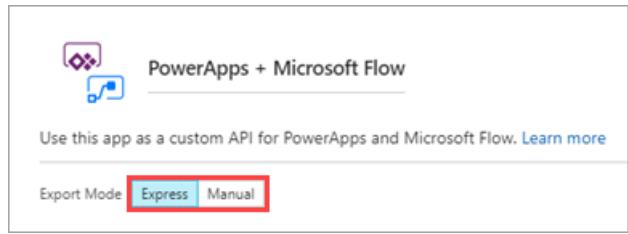
2. The **Export to PowerApps + Microsoft Flow** button should be available (if not, you must first create an OpenAPI definition). Click this button to begin the export process.

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 is the recommended approach 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 is the recommended approach 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.



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
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 required, 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 a three-step wizard for creating a custom API:

- Step 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
- Step 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)
- Step 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.

For examples of using **Express** mode with Azure Functions, see [Call a function from PowerApps](#) and [Call a function from Microsoft Flow](#).

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's a logo and the title 'PowerApps + Microsoft Flow'. Below it, a message says 'Use this app as a custom API for PowerApps and Microsoft Flow. [Learn more](#)'. There are three tabs: 'Export Mode' (selected), 'Express', and 'Manual'. Step 1, 'Prepare your metadata', is highlighted with a large number '1'. A note 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.' Below this is an 'API definition location' field containing a URL: 'https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=' followed by a red square icon. A 'Download' button is at the bottom of the section.

2. If your API definition includes any security definitions, these are called out in step #2. During import, PowerApps and Microsoft Flow detects these 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 step 2: 'Prepare security configuration'. It notes that 'Your metadata has security definitions. PowerApps and Microsoft Flow will ask for related configuration settings during import.' A red box highlights the 'DEFINITION NAME' and 'TYPE' columns of a table:

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 the 'Custom connectors' page with a gear icon in the top left. Below it, a red box highlights the '+ Create custom connector' button. A dropdown menu is open, showing 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

Upload an OpenAPI file

Open

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

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

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, this is typically `https://login.windows.net`.
- **Tenant ID** - the ID of the tenant to be used for the login. This 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. This 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

Call a function from PowerApps

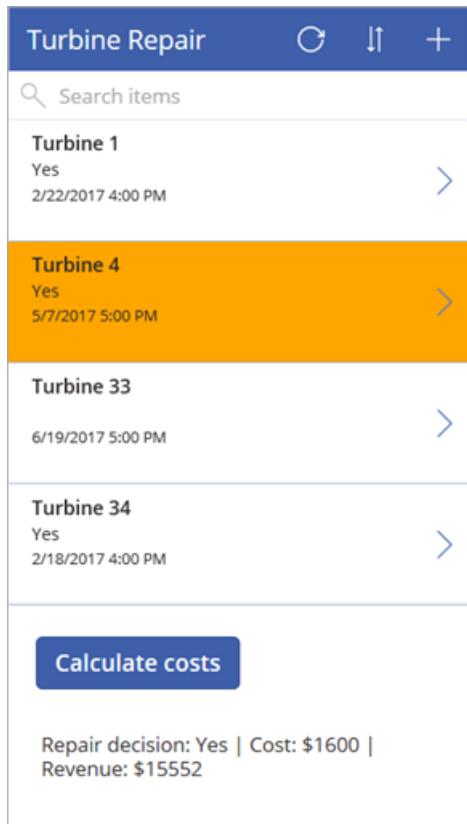
5/14/2019 • 7 minutes to read • [Edit Online](#)

The [PowerApps](#) platform is designed for business experts to build apps without traditional application code.

Professional developers can use Azure Functions to extend the capabilities of PowerApps, while shielding

PowerApps app builders from the technical details.

You build an app in this topic based on a maintenance scenario for wind turbines. This topic shows you how to call the function that you defined in [Create an OpenAPI definition for a function](#). The function determines if an emergency repair on a wind turbine is cost-effective.



For information on calling the same function from Microsoft Flow, see [Call a function from Microsoft Flow](#).

In this topic, you learn how to:

- Prepare sample data in Excel.
- Export an API definition.
- Add a connection to the API.
- Create an app and add data sources.
- Add controls to view data in the app.
- Add controls to call the function and display data.
- Run the app to determine whether a repair is cost-effective.

IMPORTANT

The OpenAPI feature is currently in preview and is only available for [version 1.x](#) of the Azure Functions runtime.

Prerequisites

- An active [PowerApps account](#) with the same sign in credentials as your Azure account.
- Excel and the [Excel sample file](#) that you will use as a data source for your app.
- Complete the tutorial [Create an OpenAPI definition for a function](#).

Export an API definition

You have an OpenAPI definition for your function, from [Create an OpenAPI definition for a function](#). The next step in this process is to export the API definition so that PowerApps and Microsoft Flow can use it in a custom API.

IMPORTANT

Remember that you must be signed in to Azure with the same credentials that you use for your PowerApps and Microsoft Flow tenants. This enables Azure to create the custom API and make it available for both PowerApps and Microsoft Flow.

1. In the [Azure portal](#), click your function app name (like **function-demo-energy**) > **Platform features** > **API definition**.

The screenshot shows the Azure portal interface for a function app named "function-demo-energy". The left sidebar lists "Visual Studio Enterprise" and "Function Apps". Under "Function Apps", "function-demo-energy" is selected, and its sub-options like "TurbineRepair", "Integrate", "Manage", and "Monitor" are visible. The main content area is titled "Platform features" and contains several sections: "GENERAL SETTINGS" (Function app settings, Application settings, Properties, Backups, All settings), "NETWORKING" (Networking, SSL, Custom domains, Authentication / Authorization, Push notifications), "API" (API definition, CORS), "APP SERVICE PLAN" (App Service plan, Quotas), "MONITORING" (Diagnostic logs, Log streaming, Process explorer, Security scanning), and "RESOURCE MANAGEMENT" (Activity log, Access control (IAM), Tags, Locks, Automation script). The "API definition" link in the API section is highlighted with a red box.

2. Click **Export to PowerApps + Flow**.

The screenshot shows the "Function API definition (Swagger)" page. It displays the URL [https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=\[REDACTED\]](https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=[REDACTED]). Below the URL is a "Copy" button. At the bottom of the page is a large blue button labeled "Export to Power Apps + Flow", which is also highlighted with a red box.

3. In the right pane, use the settings as specified in the table.

SETTING	DESCRIPTION
Export Mode	Select Express to automatically generate the custom API. Selecting Manual exports the API definition, but then you must import it into PowerApps and Microsoft Flow manually. For more information, see Export to PowerApps and Microsoft Flow .
Environment	Select the environment that the custom API should be saved to. For more information, see Environments overview (PowerApps) or Environments overview (Microsoft Flow) .
Custom API Name	Enter a name, like <code>Turbine Repair</code> .
API Key Name	Enter the name that app and flow builders should see in the custom API UI. Note that the example includes helpful information.

 PowerApps + Microsoft Flow

Use this app as a custom API for PowerApps and Microsoft Flow. [Learn more](#)

Export Mode **Express** Manual

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.

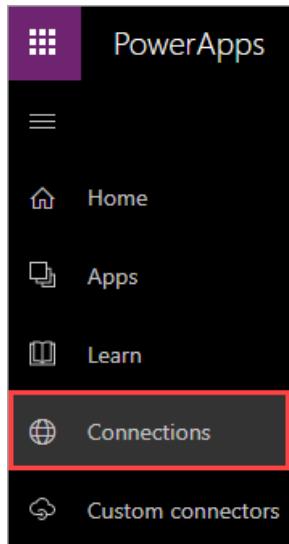
OK

4. Click **OK**. The custom API is now built and added to the environment you specified.

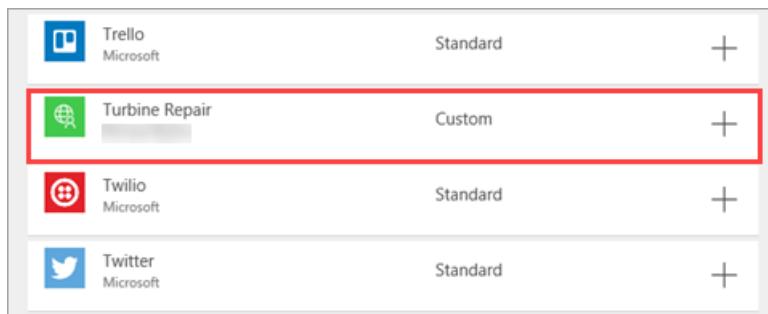
Add a connection to the API

The custom API (also known as a custom connector) is available in PowerApps, but you must make a connection to the API before you can use it in an app.

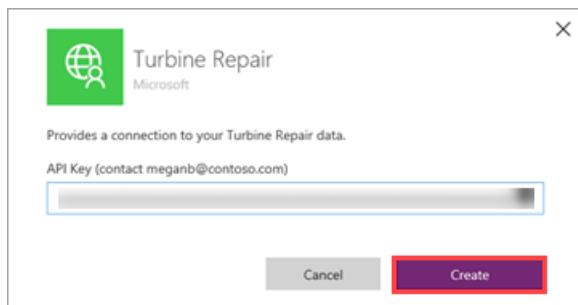
1. In web.powerapps.com, click **Connections**.



2. Click **New Connection**, scroll down to the **Turbine Repair** connector, and click it.



3. Enter the API Key, and click **Create**.



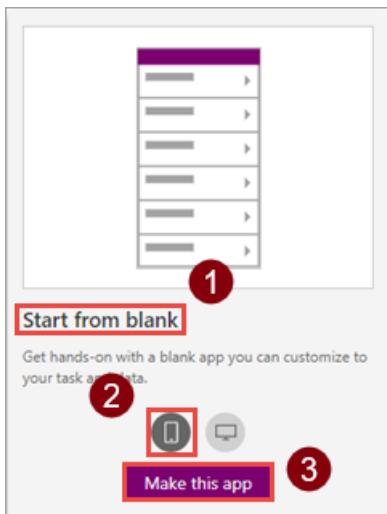
NOTE

If you share your app with others, each person who works on or uses the app must also enter the API key to connect to the API. This behavior might change in the future, and we will update this topic to reflect that.

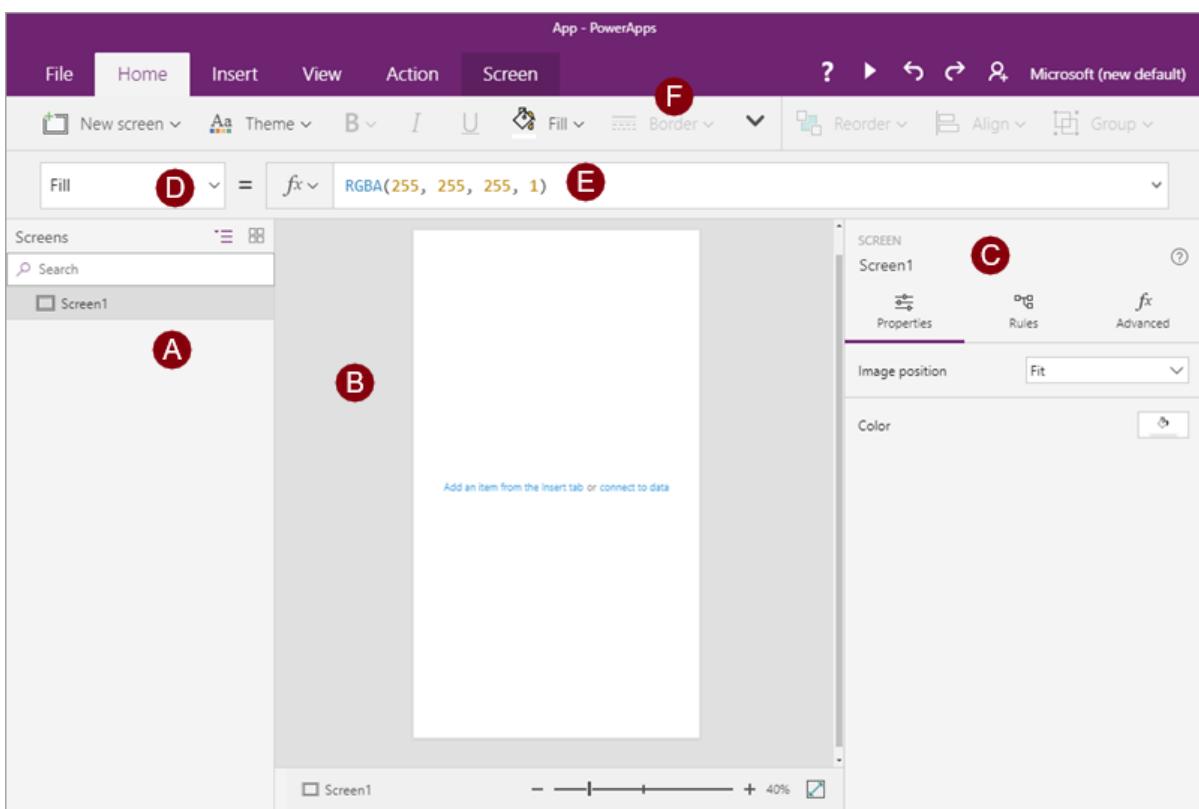
Create an app and add data sources

Now you're ready to create the app in PowerApps, and add the Excel data and the custom API as data sources for the app.

1. In web.powerapps.com, choose **Start from blank** > (phone) > **Make this app**.



The app opens in PowerApps Studio for web. The following image shows the different parts of PowerApps Studio.



(A) Left navigation bar, in which you see a hierarchical view of all the controls on each screen

(B) Middle pane, which shows the screen that you're working on

(C) Right pane, where you set options such as layout and data sources

(D) Property drop-down list, where you select the properties that formulas apply to

(E) Formula bar, where you add formulas (as in Excel) that define app behavior

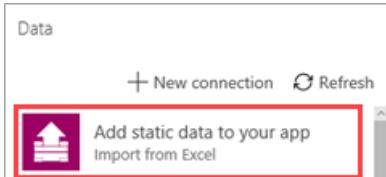
(F) Ribbon, where you add controls and customize design elements

2. Add the Excel file as a data source.

The data you will import looks like the following:

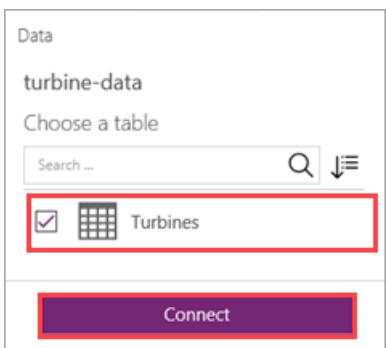
Title	Latitude	Longitude	LastServiceDate	MaxOutput	ServiceRequired	EstimatedEffort	InspectionNotes
Turbine 1	47.438401	-121.383767	2/23/2017	2850	Yes	6	This is the second issue this month.
Turbine 4	47.433385	-121.383767	5/8/2017	5400	Yes	6	
Turbine 33	47.428229	-121.404641	6/20/2017	2800			
Turbine 34	47.463637	-121.358824	2/19/2017	2800	Yes	7	
Turbine 46	47.471993	-121.298949	3/2/2017	1200			
Turbine 47	47.484059	-121.311171	8/2/2016	3350			
Turbine 55	47.438403	-121.383767	10/2/2016	2400	Yes	4	We have some parts coming in for this one.

- On the app canvas, choose **connect to data**.
- On the **Data** panel, click **Add static data to your app**.



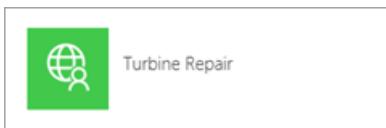
Normally you would read and write data from an external source, but you're adding the Excel data as static data because this is a sample.

- Navigate to the Excel file you saved, select the **Turbines** table, and click **Connect**.



- Add the custom API as a data source.

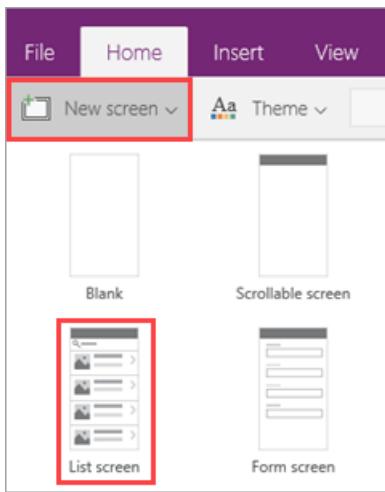
- On the **Data** panel, click **Add data source**.
- Click **Turbine Repair**.



Add controls to view data in the app

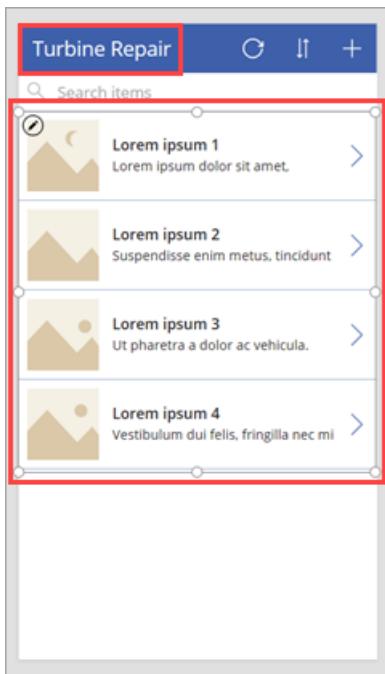
Now that the data sources are available in the app, you add a screen to your app so you can view the turbine data.

- On the **Home** tab, click **New screen > List screen**.

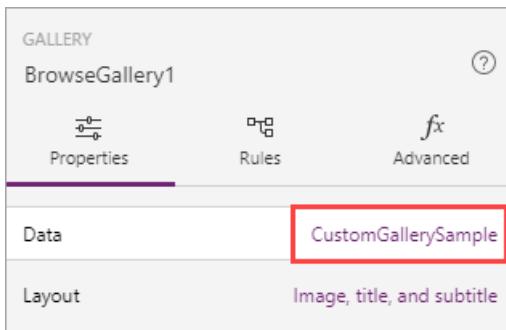


PowerApps adds a screen that contains a *gallery* to display items, and other controls that enable searching, sorting, and filtering.

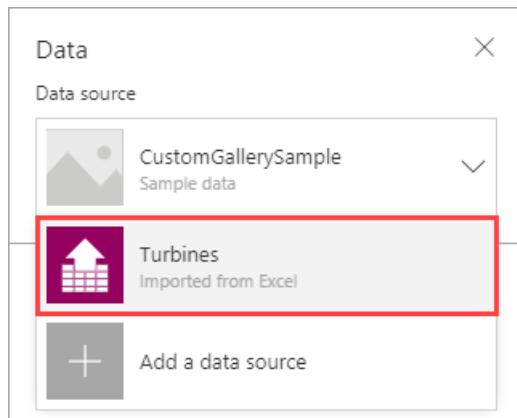
2. Change the title bar to `Turbine Repair`, and resize the gallery so there's room for more controls under it.



3. With the gallery selected, in the right pane, under **Properties**, click `CustomGallerySample`.

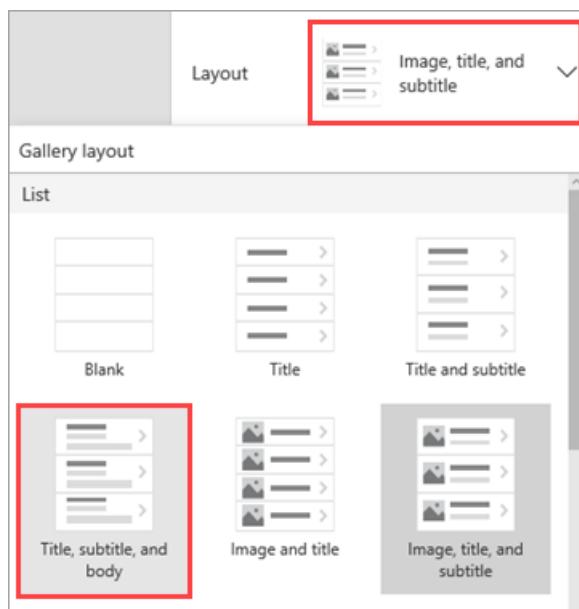


4. In the **Data** panel, select **Turbines** from the list.



The data set doesn't contain an image, so next you change the layout to better fit the data.

5. Still in the **Data** panel, change **Layout** to **Title, subtitle, and body**.



6. As the last step in the **Data** panel, change the fields that are displayed in the gallery.



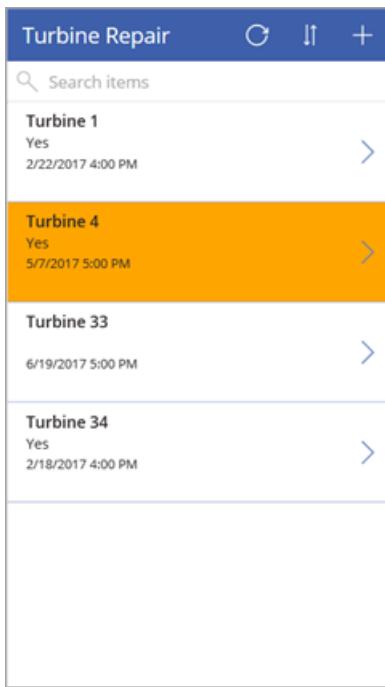
- **Body1** = LastServiceDate
- **Subtitle2** = ServiceRequired
- **Title2** = Title

7. With the gallery selected, set the **TemplateFill** property to the following formula:

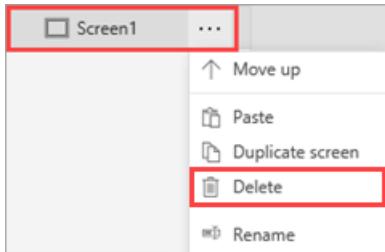
```
If(ThisItem.IsSelected, Orange, White)
```



Now it's easier to see which gallery item is selected.



8. You don't need the original screen in the app. In the left pane, hover over **Screen1**, click ..., and **Delete**.



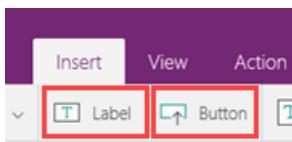
9. Click **File**, and name the app. Click **Save** on the left menu, then click **Save** in the bottom right corner.

There's a lot of other formatting you would typically do in a production app, but we'll move on to the important part for this scenario - calling the function.

Add controls to call the function and display data

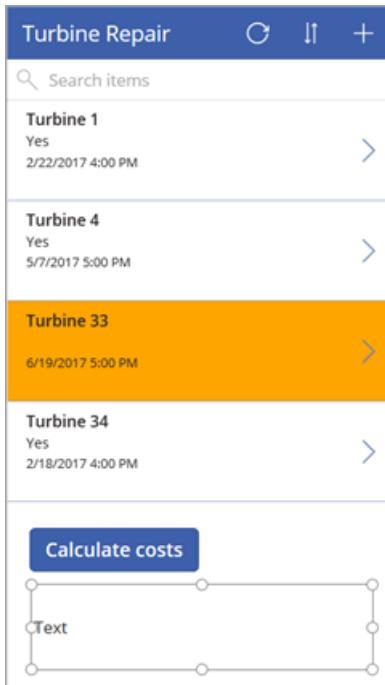
You have an app that displays summary data for each turbine, so now it's time to add controls that call the function you created, and display the data that is returned. You access the function based on the way you name it in the OpenAPI definition; in this case it's `TurbineRepair.CalculateCosts()`.

1. In the ribbon, on the **Insert** tab, click **Button**. Then on the same tab, click **Label**



2. Drag the button and the label below the gallery, and resize the label.

3. Select the button text, and change it to `Calculate costs`. The app should look like the following image.



4. Select the button, and enter the following formula for the button's **OnSelect** property.

```
If (BrowseGallery1.Selected.ServiceRequired="Yes", ClearCollect(DetermineRepair,
TurbineRepair.CalculateCosts({hours: BrowseGallery1.Selected.EstimatedEffort, capacity:
BrowseGallery1.Selected.MaxOutput})))
```

This formula executes when the button is clicked, and it does the following if the selected gallery item has a **ServiceRequired** value of **Yes**:

- Clears the collection `DetermineRepair` to remove data from previous calls. A collection is a tabular variable.
- Assigns to the collection the data returned by calling the function `TurbineRepair.CalculateCosts()`.

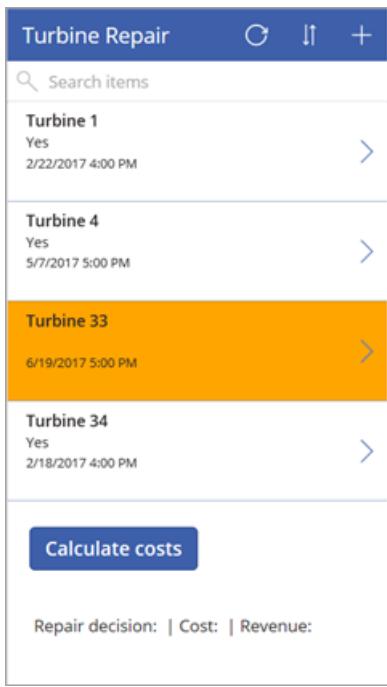
The values passed to the function come from the **EstimatedEffort** and **MaxOutput** fields for the item selected in the gallery. These fields aren't displayed in the gallery, but they're still available to use in formulas.

5. Select the label, and enter the following formula for the label's **Text** property.

```
"Repair decision: " & First(DetermineRepair).message & " | Cost: " & First(DetermineRepair).costToFix &
" | Revenue: " & First(DetermineRepair).revenueOpportunity
```

This formula uses the `First()` function to access the first (and only) row of the `DetermineRepair` collection. It then displays the three values that the function returns: `message`, `costToFix`, and `revenueOpportunity`. These values are blank before the app runs for the first time.

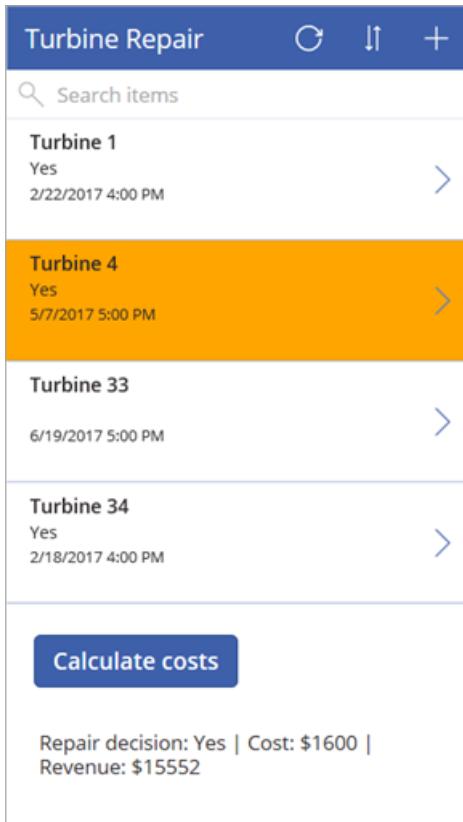
The completed app should look like the following image.



Run the app

You have a complete app! Now it's time to run it and see the function calls in action.

1. In the upper right corner of PowerApps Studio, click the run button:
2. Select a turbine with a value of **Yes** for **ServiceRequired**, then click the **Calculate costs** button. You should see a result like the following image.



3. Try the other turbines to see what's returned by the function each time.

Next steps

In this topic, you learned how to:

- Prepare sample data in Excel.
- Export an API definition.
- Add a connection to the API.
- Create an app and add data sources.
- Add controls to view data in the app.
- Add controls to call the function and display data
- Run the app to determine whether a repair is cost-effective.

To learn more about PowerApps, see [Introduction to PowerApps](#).

To learn about other interesting scenarios that use Azure Functions, see [Call a function from Microsoft Flow](#) and [Create a function that integrates with Azure Logic Apps](#).

Call a function from Microsoft Flow

5/14/2019 • 8 minutes to read • [Edit Online](#)

[Microsoft Flow](#) makes it easy to automate workflows and business processes between your favorite apps and services. Professional developers can use Azure Functions to extend the capabilities of Microsoft Flow, while shielding flow builders from the technical details.

You build a flow in this topic based on a maintenance scenario for wind turbines. This topic shows you how to call the function that you defined in [Create an OpenAPI definition for a function](#). The function determines if an emergency repair on a wind turbine is cost-effective. If it is cost-effective, the flow sends an email to recommend the repair.

For information on calling the same function from PowerApps, see [Call a function from PowerApps](#).

In this topic, you learn how to:

- Create a list in SharePoint.
- Export an API definition.
- Add a connection to the API.
- Create a flow to send email if a repair is cost-effective.
- Run the flow.

IMPORTANT

The OpenAPI feature is currently in preview and is only available for [version 1.x](#) of the Azure Functions runtime.

Prerequisites

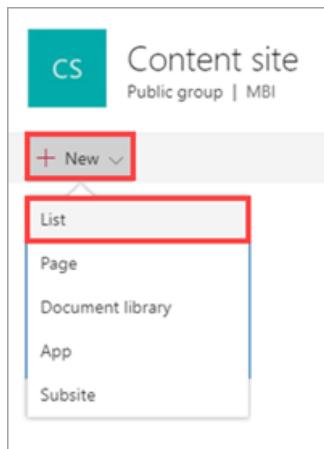
- An active [Microsoft Flow account](#) with the same sign in credentials as your Azure account.
- SharePoint, which you use as a data source for this flow. Sign up for [an Office 365 trial](#) if you don't already have SharePoint.
- Complete the tutorial [Create an OpenAPI definition for a function](#).

Create a SharePoint list

You start off by creating a list that you use as a data source for the flow. The list has the following columns.

LIST COLUMN	DATA TYPE	NOTES
Title	Single line of text	Name of the turbine
LastServiceDate	Date	
MaxOutput	Number	Output of the turbine, in KwH
ServiceRequired	Yes/No	
EstimatedEffort	Number	Estimated time for the repair, in hours

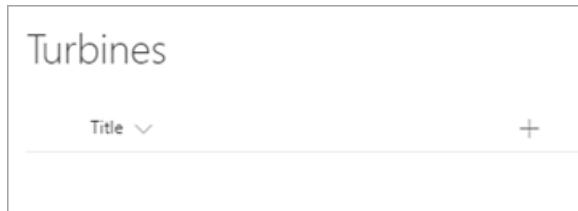
1. In your SharePoint site, click or tap **New**, then **List**.



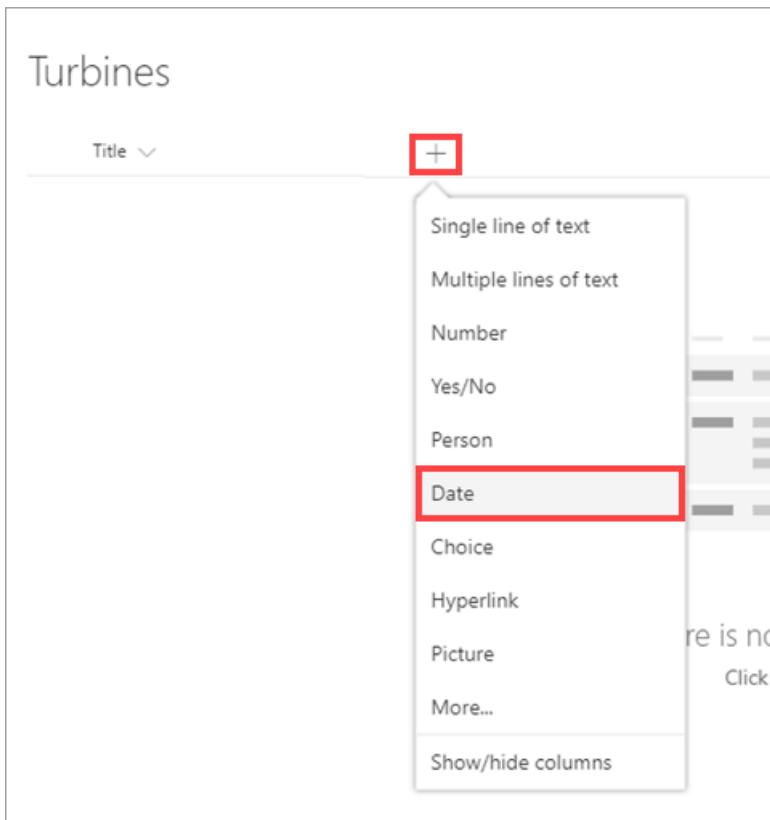
2. Enter the name **Turbines**, then click or tap **Create**.

A screenshot of the 'Create list' dialog box. It has fields for 'Name' (containing 'Turbines'), 'Description' (empty), and a checked 'Show in site navigation' checkbox. The 'Create' button is highlighted with a red box. Below it is a 'Cancel' button.

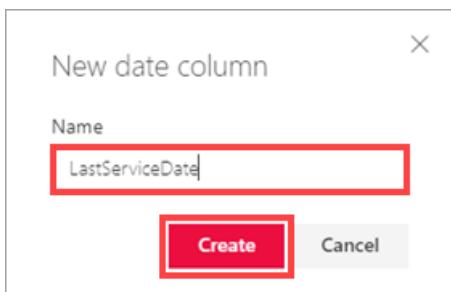
The **Turbines** list is created, with the default **Title** field.



3. Click or tap **+** then **Date**.



4. Enter the name `LastServiceDate`, then click or tap **Create**.



5. Repeat the last two steps for the other three columns in the list:

- a. **Number** > "MaxOutput"
- b. **Yes/No** > "ServiceRequired"
- c. **Number** > "EstimatedEffort"

That's it for now - you should have an empty list that looks like the following image. You add data to the list after you create the flow.

Title	LastServiceDate	MaxOutput	ServiceRequired	EstimatedEffort	+
Turbines					

Export an API definition

You have an OpenAPI definition for your function, from [Create an OpenAPI definition for a function](#). The next step in this process is to export the API definition so that PowerApps and Microsoft Flow can use it in a custom API.

IMPORTANT

Remember that you must be signed in to Azure with the same credentials that you use for your PowerApps and Microsoft Flow tenants. This enables Azure to create the custom API and make it available for both PowerApps and Microsoft Flow.

1. In the [Azure portal](#), click your function app name (like **function-demo-energy**) > **Platform features** > **API definition**.

The screenshot shows the Azure portal interface for a function app named 'function-demo-energy'. The left sidebar lists 'Visual Studio Enterprise' and 'Function Apps'. Under 'Function Apps', 'function-demo-energy' is selected, and its sub-options like 'TurbineRepair', 'Integrate', 'Manage', and 'Monitor' are visible. The main content area is titled 'Platform features' and contains several sections: 'GENERAL SETTINGS' (Function app settings, Application settings, Properties, Backups, All settings), 'NETWORKING' (Networking, SSL, Custom domains, Authentication / Authorization, Push notifications), 'API' (API definition, CORS), 'APP SERVICE PLAN' (App Service plan, Quotas), 'MONITORING' (Diagnostic logs, Log streaming, Process explorer, Security scanning), and 'RESOURCE MANAGEMENT' (Activity log, Access control (IAM), Tags, Locks, Automation script). A red box highlights the 'API definition' link under the API section.

2. Click **Export to PowerApps + Flow**.

The screenshot shows the 'Function API definition (Swagger)' page. It includes a description of what Swagger is and how to use it. A prominent blue button labeled 'Export to Power Apps + Flow' is highlighted with a red box. Below it, there's a 'Copy' link next to a URL field containing 'https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=' followed by a redacted code segment.

3. In the right pane, use the settings as specified in the table.

SETTING	DESCRIPTION
Export Mode	Select Express to automatically generate the custom API. Selecting Manual exports the API definition, but then you must import it into PowerApps and Microsoft Flow manually. For more information, see Export to PowerApps and Microsoft Flow .

SETTING	DESCRIPTION
Environment	Select the environment that the custom API should be saved to. For more information, see Environments overview (PowerApps) or Environments overview (Microsoft Flow) .
Custom API Name	Enter a name, like <code>Turbine Repair</code> .
API Key Name	Enter the name that app and flow builders should see in the custom API UI. Note that the example includes helpful information.

PowerApps + Microsoft Flow

Use this app as a custom API for PowerApps and Microsoft Flow. [Learn more](#)

Export Mode **Express** Manual

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.

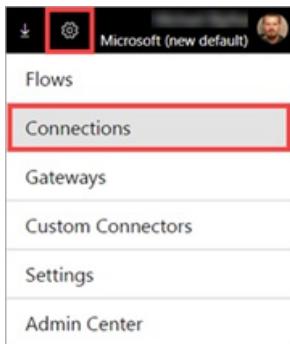
OK

4. Click **OK**. The custom API is now built and added to the environment you specified.

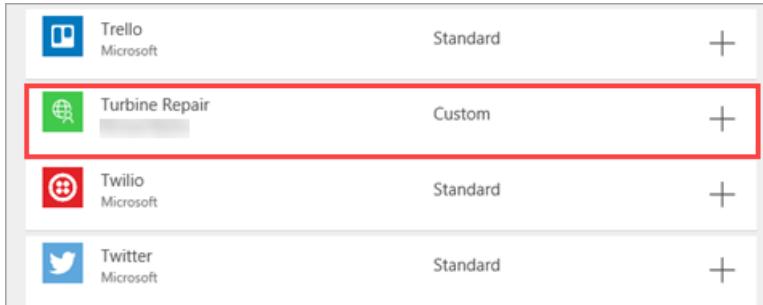
Add a connection to the API

The custom API (also known as a custom connector) is available in Microsoft Flow, but you must make a connection to the API before you can use it in a flow.

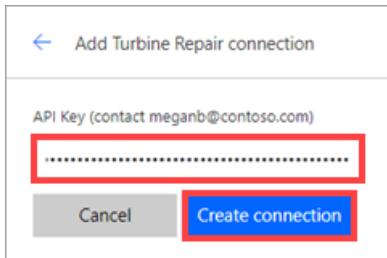
1. In flow.microsoft.com, click the gear icon (in the upper right), then click **Connections**.



2. Click **Create Connection**, scroll down to the **Turbine Repair** connector, and click it.



3. Enter the API Key, and click **Create connection**.



NOTE

If you share your flow with others, each person who works on or uses the flow must also enter the API key to connect to the API. This behavior might change in the future, and we will update this topic to reflect that.

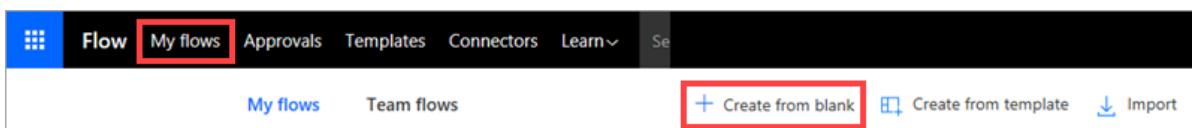
Create a flow

Now you're ready to create a flow that uses the custom connector and the SharePoint list you created.

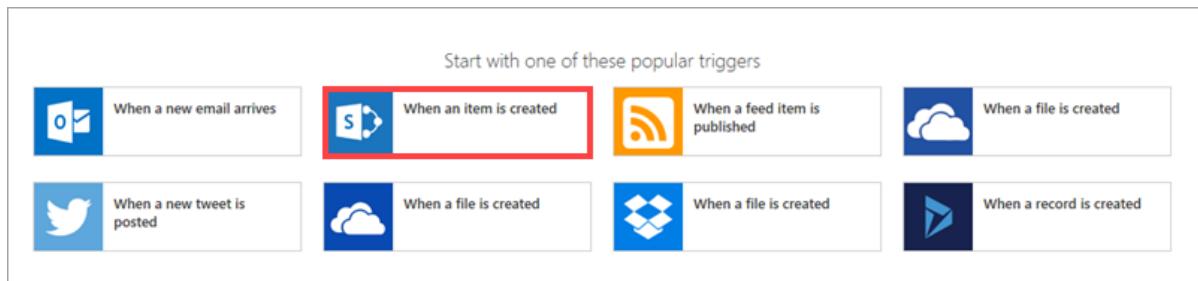
Add a trigger and specify a condition

You first create a flow from blank (without a template), and add a *trigger* that fires when an item is created in the SharePoint list. You then add a *condition* to determine what happens next.

1. In flow.microsoft.com, click **My Flows**, then **Create from blank**.

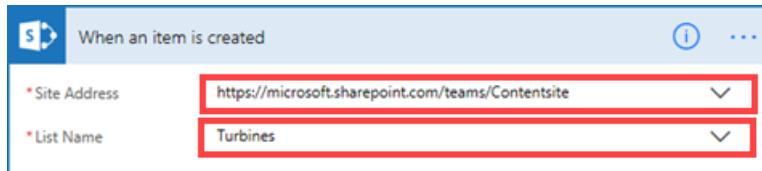


2. Click the SharePoint trigger **When an item is created**.

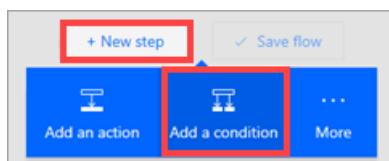


If you're not already signed into SharePoint, you will be prompted to do so.

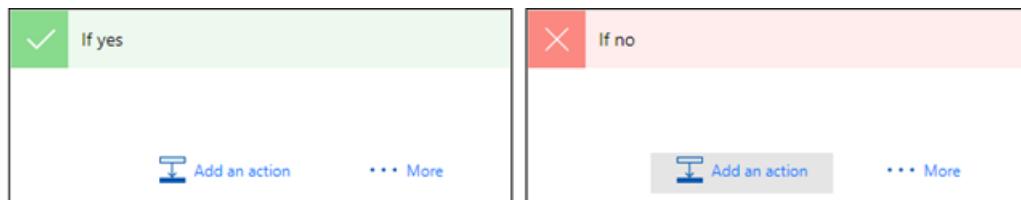
- For **Site Address**, enter your SharePoint site name, and for **List Name**, enter the list that contains the turbine data.



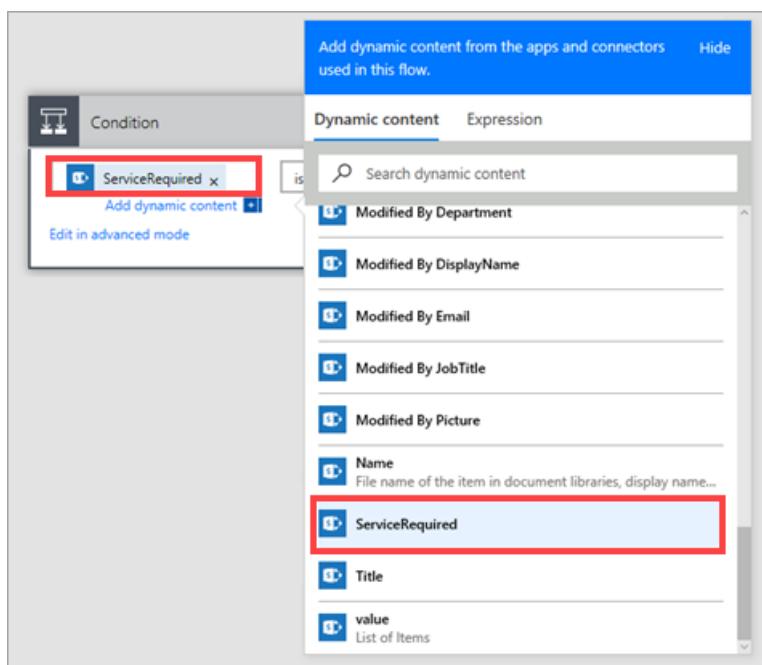
- Click **New step**, then **Add a condition**.



Microsoft Flow adds two branches to the flow: **If yes** and **If no**. You add steps to one or both branches after you define the condition that you want to match.



- On the **Condition** card, click the first box, then select **ServiceRequired** from the **Dynamic content** dialog box.



6. Enter a value of **True** for the condition.



The value is displayed as **Yes** or **No** in the SharePoint list, but it is stored as a *boolean*, either **True** or **False**.

7. Click **Create flow** at the top of the page. Be sure to click **Update Flow** periodically.

For any items created in the list, the flow checks if the **ServiceRequired** field is set to **Yes**, then goes to the **If yes** branch or the **If no** branch as appropriate. To save time, in this topic you only specify actions for the **If yes** branch.

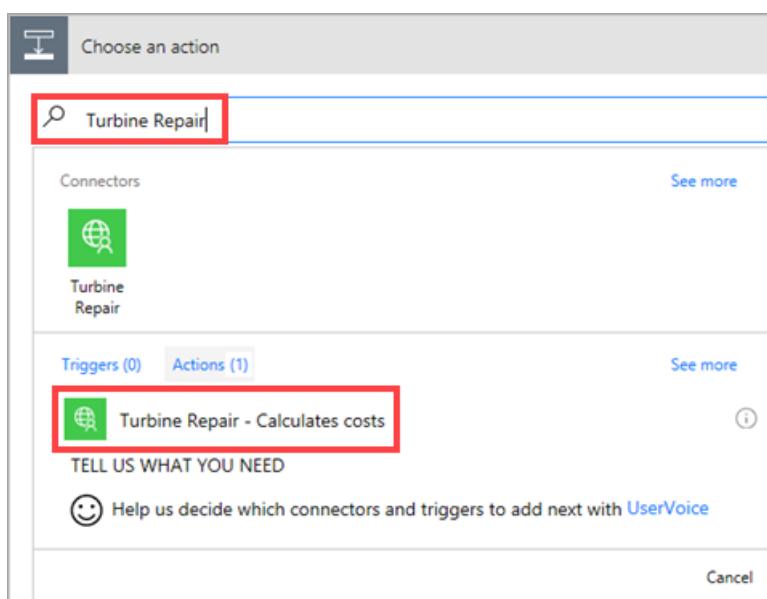
Add the custom connector

You now add the custom connector that calls the function in Azure. You add the custom connector to the flow just like a standard connector.

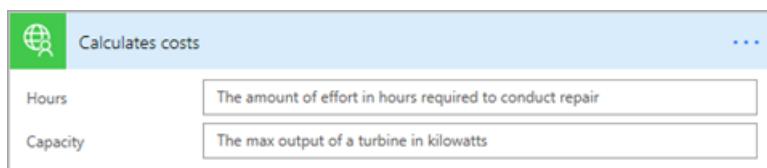
1. In the **If yes** branch, click **Add an action**.



2. In the **Choose an action** dialog box, search for **Turbine Repair**, then select the action **Turbine Repair - Calculates costs**.



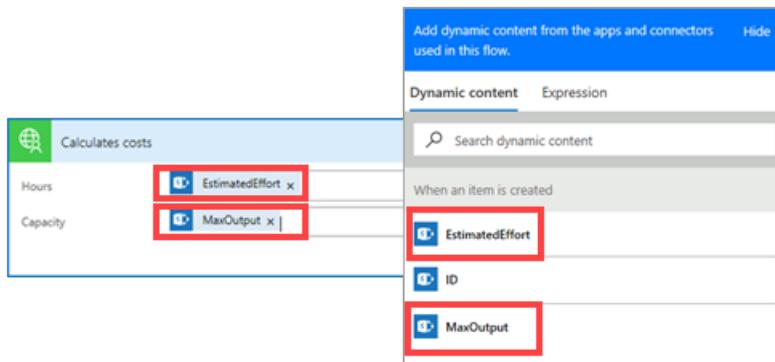
The following image shows the card that is added to the flow. The fields and descriptions come from the OpenAPI definition for the connector.



3. On the **Calculates costs** card, use the **Dynamic content** dialog box to select inputs for the function.

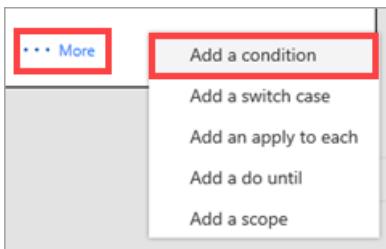
Microsoft Flow shows numeric fields but not the date field, because the OpenAPI definition specifies that **Hours** and **Capacity** are numeric.

For **Hours**, select **EstimatedEffort**, and for **Capacity**, select **MaxOutput**.

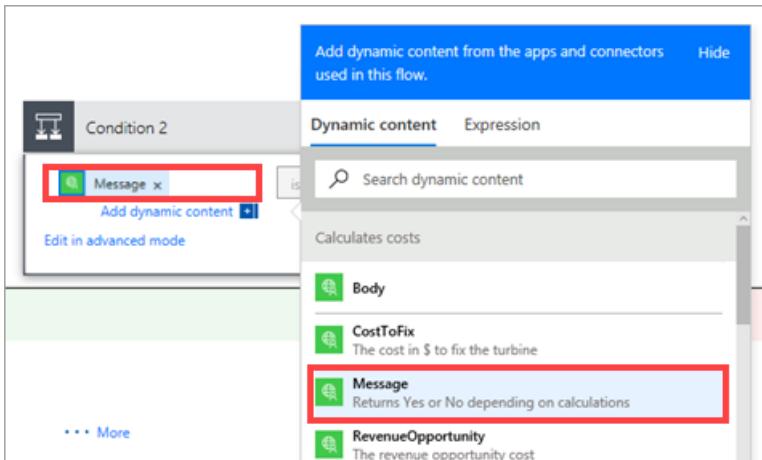


Now you add another condition based on the output of the function.

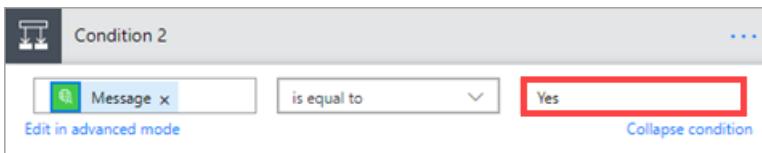
4. At the bottom of the **If yes** branch, click **More**, then **Add a condition**.



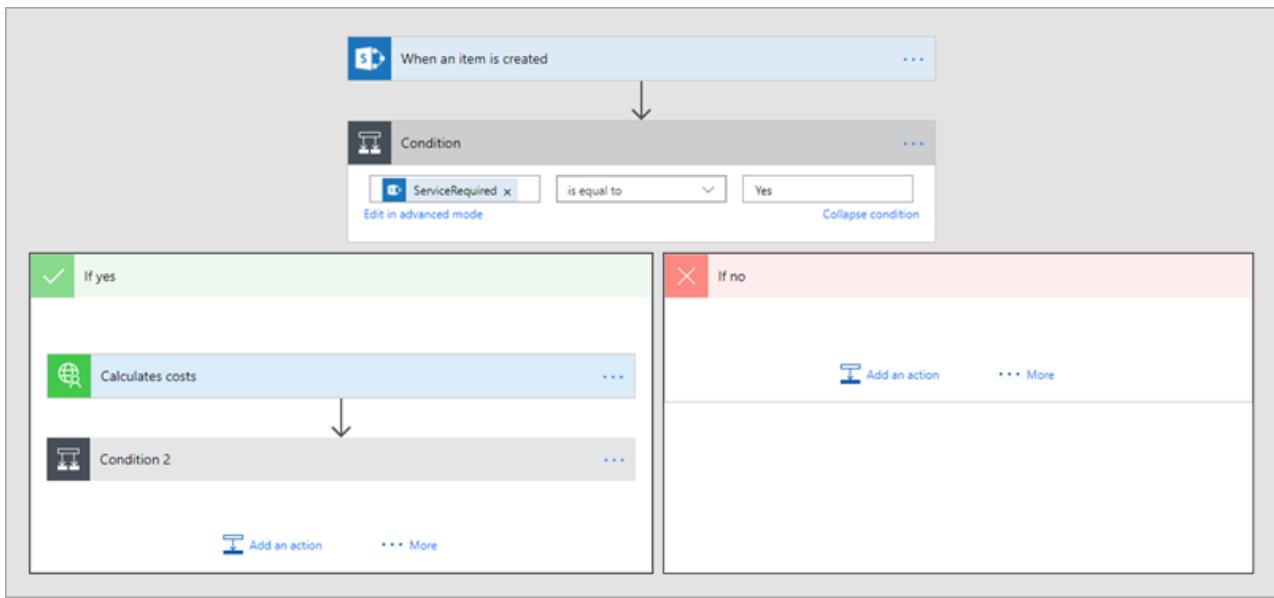
5. On the **Condition 2** card, click the first box, then select **Message** from the **Dynamic content** dialog box.



6. Enter a value of **Yes**. The flow goes to the next **If yes** branch or **If no** branch based on whether the message returned by the function is yes (make the repair) or no (don't make the repair).



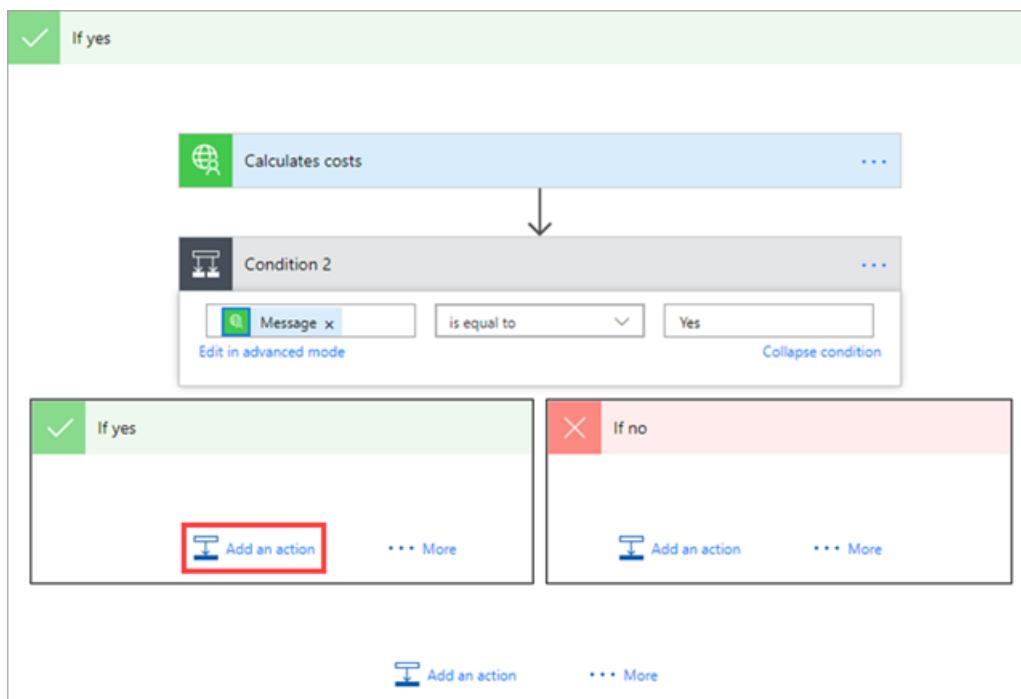
The flow should now look like the following image.



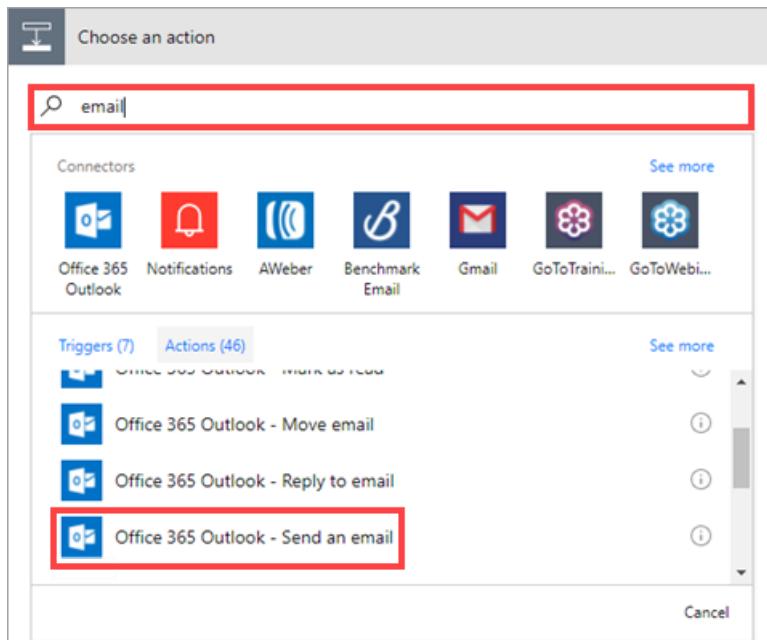
Send email based on function results

At this point in the flow, the function has returned a **Message** value of **Yes** or **No** from the function, as well as other information on costs and potential revenue. In the **If yes** branch of the second condition, you will send an email, but you could do any number of things, like writing back to the SharePoint list or starting an [approval process](#).

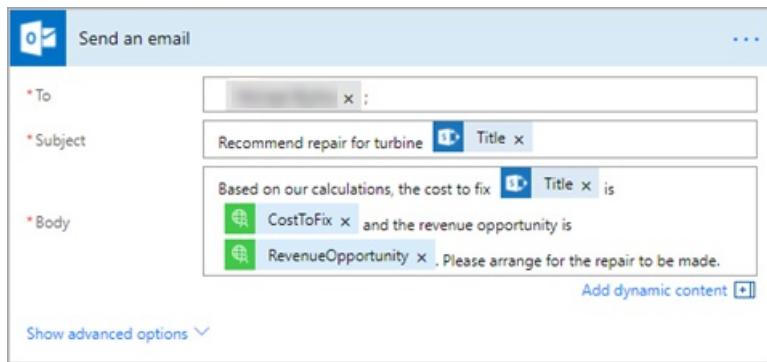
1. In the **If yes** branch of the second condition, click **Add an action**.



2. In the **Choose an action** dialog box, search for **email**, then select a send email action based on the email system you use (in this case Outlook).

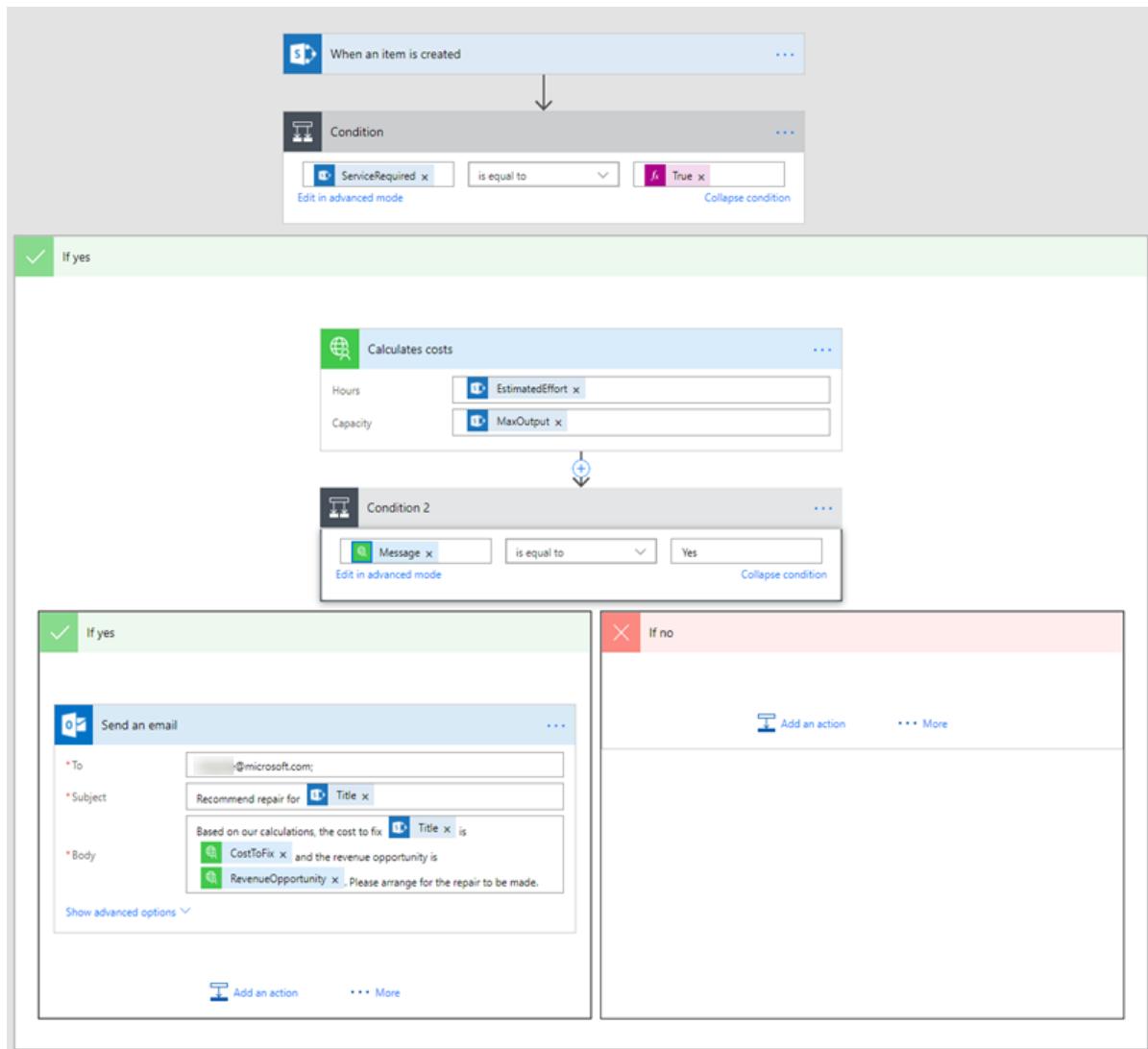


3. On the **Send an email** card, compose an email. Enter a valid name in your organization for the **To** field. In the image below you can see the other fields are a combination of text and tokens from the **Dynamic content** dialog box.



The **Title** token comes from the SharePoint list, and **CostToFix** and **RevenueOpportunity** are returned by the function.

The completed flow should look like the following image (we left out the first **If no** branch to save space).

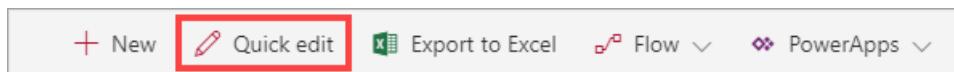


- Click **Update Flow** at the top of the page, then click **Done**.

Run the flow

Now that the flow is completed, you add a row to the SharePoint list and see how the flow responds.

- Go back to the SharePoint list, and click **Quick Edit**.



- Enter the following values in the edit grid.

LIST COLUMN	VALUE
Title	Turbine 60
LastServiceDate	08/04/2017
MaxOutput	2500
ServiceRequired	Yes
EstimatedEffort	10

- Click **Done**.

Title	LastServiceDate	MaxOutput	ServiceRequired	EstimatedEffort
Turbine 60 ✎	8/4/2017	2,500	Yes	10

When you add the item, it triggers the flow, which you take a look at next.

4. In [flow.microsoft.com](#), click **My Flows**, then click the flow you created.

Name	Last modified
Turbine Repair Approval	4 seconds ago

5. Under **RUN HISTORY**, click the flow run.

Succeeded	9 hours ago	7 seconds
-----------	-------------	-----------

If the run was successful, you can review the flow operations on the next page. If the run failed for any reason, the next page provides troubleshooting information.

6. Expand the cards to see what occurred during the flow. For example, expand the **Calculates costs** card to see the inputs to and outputs from the function.

INPUTS
Hours 10
Capacity 2500

OUTPUTS
Message Yes
RevenueOpportunity \$7200
CostToFix \$2600

7. Check the email account for the person you specified in the **To** field of the **Send an email** card. The email sent from the flow should look like the following image.



You can see how the tokens have been replaced with the correct values from the SharePoint list and the function.

Next steps

In this topic, you learned how to:

- Create a list in SharePoint.
- Export an API definition.
- Add a connection to the API.
- Create a flow to send email if a repair is cost-effective.
- Run the flow.

To learn more about Microsoft Flow, see [Get started with Microsoft Flow](#).

To learn about other interesting scenarios that use Azure Functions, see [Call a function from PowerApps](#) and [Create a function that integrates with Azure Logic Apps](#).

How to use managed identities for App Service and Azure Functions

7/17/2019 • 12 minutes to read • [Edit Online](#)

NOTE

Managed identity support for App Service on Linux and Web App for Containers is currently in preview.

IMPORTANT

Managed identities for App Service and Azure Functions will not behave as expected if your app is migrated across subscriptions/tenants. The app will need to obtain a new identity, which can be done by disabling and re-enabling the feature. See [Removing an identity](#) below. Downstream resources will also need to have access policies updated to use the new identity.

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. A managed identity from Azure Active Directory allows your app to easily access other AAD-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 AAD, 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. System-assigned identity support is generally available for Windows apps.
- A **user-assigned identity** is a standalone Azure resource which can be assigned to your app. An app can have multiple user-assigned identities. User-assigned identity support is in preview for all app types.

Adding 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 **Managed identity**.
4. Within the **System assigned** tab, switch **Status** to **On**. Click **Save**.

A system assigned managed identity enables Azure resources to authenticate to cloud services (e.g. Azure Key lifecycle of this type of managed identity is tied to the lifecycle of this resource. Additionally, each resource (e.g.

System assigned **User assigned (preview)**

Status **Off** **On**

Object ID 7283a4ee-ac06-4f67-b8e7-513d24f010d1

This resource is registered with Azure Active Directory. You can control its access to services like Az

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 is 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 a web app and assigning it an identity using Azure PowerShell:

1. If needed, install the Azure PowerShell using the instruction 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 myResourceGroup -Location $location  
  
# Create an App Service plan in Free tier.  
New-AzAppServicePlan -Name $webappname -Location $location -ResourceGroupName myResourceGroup -Tier Free  
  
# Create a web app.  
New-AzWebApp -Name $webappname -Location $location -AppServicePlan $webappname -ResourceGroupName myResourceGroup
```

3. Run the `Set-AzWebApp -AssignIdentity` command to create the identity for this application:

```
Set-AzWebApp -AssignIdentity $true -Name $webappname -ResourceGroupName myResourceGroup
```

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>"
}
```

Where `<TENANTID>` and `<PRINCIPALID>` are replaced with GUIDs. The tenantId property identifies what AAD tenant the identity belongs to. The principalId is a unique identifier for the application's new identity. Within AAD, the service principal has the same name that you gave to your App Service or Azure Functions instance.

Adding a user-assigned identity (preview)

NOTE

User-assigned identities are currently in preview. Sovereign clouds are not yet supported.

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

NOTE

This portal experience is being deployed and may not yet be available in all regions.

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 **Managed identity**.

5. Within the **User assigned (preview)** tab, click **Add**.

6. Search for the identity you created earlier and select it. Click **Add**.

The screenshot shows the Azure portal's 'Identity' blade for an App Service named 'userassigned-windows'. The 'User assigned (preview)' tab is active. A single identity named 'myUserAssignedIdentity' is listed under the 'NAME' column, with 'hendokvf' in the 'RESOURCE GROUP' column. The blade also includes a sidebar with links for Application Insights, Identity, Backups, Custom domains, SSL settings, Networking, Scale up (App Service plan), Scale out (App Service plan), and WebJobs.

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 and a cotells 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": "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>"
        }
    }
}
```

Where `<PRINCIPALID>` and `<CLIENTID>` are replaced with GUIDs. The principalId is a unique identifier for the identity which is used for AAD administration. The clientId is a unique identifier for the application's new identity that is used for specifying which identity to use during runtime calls.

Obtaining tokens for Azure resources

An app can use its identity to get tokens to other resources protected by AAD, such as Azure Key Vault. These tokens represent the application accessing the resource, and not any specific user of the application.

IMPORTANT

You may need to configure the target resource to allow access from your application. For example, if you request a token to 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](#).

There is a simple REST protocol for obtaining a token in App Service and Azure Functions. For .NET applications, the Microsoft.Azure.Services.AppAuthentication library provides an abstraction over this protocol and supports a local development experience.

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. For more on local 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));
```

To learn more about Microsoft.Azure.Services.AppAuthentication 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);
```

Using the REST protocol

An app with a managed identity has two environment variables defined:

- **MSI_ENDPOINT** - the URL to the local token service.
- **MSI_SECRET** - a header used to help mitigate server-side request forgery (SSRF) attacks. The value is rotated by the platform.

The **MSI_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 AAD 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. "2017-09-01" is currently the only version supported.
secret	Header	The value of the MSI_SECRET environment variable. This header is used to help mitigate server-side request forgery (SSRF) attacks.
clientid	Query	(Optional) The ID of the user-assigned identity to be used. If omitted, the system-assigned identity is used.

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.
expires_on	The time when the access token expires. The date is represented as the number of seconds from 1970-01-01T0:0:0Z UTC until the expiration time. This value is used to determine the lifetime of cached tokens.
resource	The App ID URI of the receiving web service.
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 AAD service-to-service access token request](#).

NOTE

Environment variables are set up when the process first starts, so after enabling a managed identity for your application, you may need to restart your application, or redeploy its code, before `MSI_ENDPOINT` and `MSI_SECRET` are available to your code.

REST protocol examples

An example request might look like the following:

```
GET /MSI/token?resource=https://vault.azure.net&api-version=2017-09-01 HTTP/1.1
Host: localhost:4141
Secret: 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": "09/14/2017 00:00:00 PM +00:00",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer"
}
```

Code examples

To make this request in C#:

```
public static async Task<HttpResponseMessage> GetToken(string resource, string apiversion) {
    HttpClient client = new HttpClient();
    client.DefaultRequestHeaders.Add("Secret", Environment.GetEnvironmentVariable("MSI_SECRET"));
    return await client.GetAsync(String.Format("{0}/?resource={1}&api-version={2}",
        Environment.GetEnvironmentVariable("MSI_ENDPOINT"), resource, apiversion));
}
```

TIP

For .NET languages, you can also use [Microsoft.Azure.Services.AppAuthentication](#) instead of crafting this request yourself.

In Node.js:

```
const rp = require('request-promise');
const getToken = function(resource, apiver, cb) {
    let options = {
        uri: `${process.env["MSI_ENDPOINT"]}/?resource=${resource}&api-version=${apiver}`,
        headers: {
            'Secret': process.env["MSI_SECRET"]
        }
    };
    rp(options)
        .then(cb);
}
```

In PowerShell:

```
$apiVersion = "2017-09-01"
$resourceURI = "https://<AAD-resource-URI-for-resource-to-obtain-token>"
$tokenAuthURI = $env:MSI_ENDPOINT + "?resource=$resourceURI&api-version=$apiVersion"
$tokenResponse = Invoke-RestMethod -Method Get -Headers @{"Secret"="$env:MSI_SECRET"} -Uri $tokenAuthURI
$accessToken = $tokenResponse.access_token
```

Removing 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, in the REST/ARM template protocol, this is done by setting the type to "None":

```
"identity": {  
    "type": "None"  
}
```

Removing a system-assigned identity in this way will also delete it from AAD. System-assigned identities are also automatically removed from AAD when the app resource is deleted.

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](#)

How to troubleshoot "functions runtime is unreachable"

3/27/2019 • 3 minutes to read • [Edit Online](#)

Error text

This doc is intended to troubleshoot the following error when displayed in the Functions portal.

Error: Azure Functions Runtime is unreachable. Click here for details on storage configuration

Summary

This issue occurs when the Azure Functions Runtime cannot start. The most common reason for this error to occur is the function app losing access to its storage account. [Read more about the storage account requirements here](#)

Troubleshooting

We'll walk through the four most common error cases, how to identify, and how to resolve each case.

1. Storage Account deleted
2. Storage Account application settings deleted
3. Storage Account credentials invalid
4. Storage Account Inaccessible
5. Daily Execution Quota Full

Storage account deleted

Every function app requires a storage account to operate. If that account is deleted your Function will not work.

How to find your storage account

Start by looking up your storage account name in your Application Settings. Either `AzureWebJobsStorage` or `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` will contain the name of your storage account wrapped up in a connection string. Read more specifics at the [application setting reference here](#)

Search for your storage account in the Azure portal to see if it still exists. If it has been deleted, you will need to recreate a storage account and replace your storage connection strings. Your function code is lost and you will need to redeploy it again.

Storage account application settings deleted

In the previous step, if you did not have a storage account connection string they were likely deleted or overwritten. Deleting app settings is most commonly done when using deployment slots or Azure Resource Manager scripts to set application settings.

Required application settings

- Required
 - `AzureWebJobsStorage`
- Required for Consumption Plan Functions
 - `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING`
 - `WEBSITE_CONTENTSHARE`

[Read about these application settings here](#)

Guidance

- Do not check "slot setting" for any of these settings. When you swap deployment slots the Function will break.
- Do not modify these settings as part of automated deployments.
- These settings must be provided and valid at creation time. An automated deployment that does not contain these settings will result in a non-functional App, even if the settings are added after the fact.

Storage account credentials invalid

The above Storage Account connection strings must be updated if you regenerate storage keys. [Read more about storage key management here](#)

Storage account inaccessible

Your Function App must be able to access the storage account. Common issues that block a Functions access to a storage account are:

- Function Apps deployed to App Service Environments 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. [Read more about storage account firewall configuration here](#)

Daily Execution Quota Full

If you have a Daily Execution Quota configured, your Function App will be temporarily disabled and many of the portal controls will become unavailable.

- To verify, check open Platform Features > Function App Settings in the portal. You will see the following message if you are over quota
 - The Function App has reached daily usage quota and has been stopped until the next 24 hours time frame.
- Remove the quota and restart your app to resolve the issue.

Next Steps

Now that your Function App is back and operational take a look at our quickstarts and developer references to get up and running again!

- [Create your first Azure Function](#)
Jump right in and create your first function using the Azure Functions quickstart.
- [Azure Functions developer reference](#)
Provides more technical information about the Azure Functions runtime and a reference for coding functions and defining triggers and bindings.
- [Testing Azure Functions](#)
Describes various tools and techniques for testing your functions.
- [How to scale Azure Functions](#)
Discusses service plans available with Azure Functions, including the Consumption hosting plan, and how to choose the right plan.
- [Learn more about Azure App Service](#)
Azure Functions leverages Azure App Service for core functionality like deployments, environment variables, and diagnostics.

App settings reference for Azure Functions

7/30/2019 • 5 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.

APPINSIGHTS_INSTRUMENTATIONKEY

The Application Insights instrumentation key if you're using Application Insights. See [Monitor Azure Functions](#).

KEY	SAMPLE VALUE
APPINSIGHTS_INSTRUMENTATIONKEY	5dbdd5e9-af77-484b-9032-64f83bb83bb

AZURE_FUNCTIONS_ENVIRONMENT

In version 2.x 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.

AzureWebJobsDashboard

Optional storage account connection string for storing logs and displaying them in the **Monitor** tab in the portal. The storage account must be a general-purpose one that supports blobs, queues, and tables. See [Storage account](#) and [Storage account requirements](#).

KEY	SAMPLE VALUE
AzureWebJobsDashboard	DefaultEndpointsProtocol=https;AccountName=[name];AccountKey=[key]

TIP

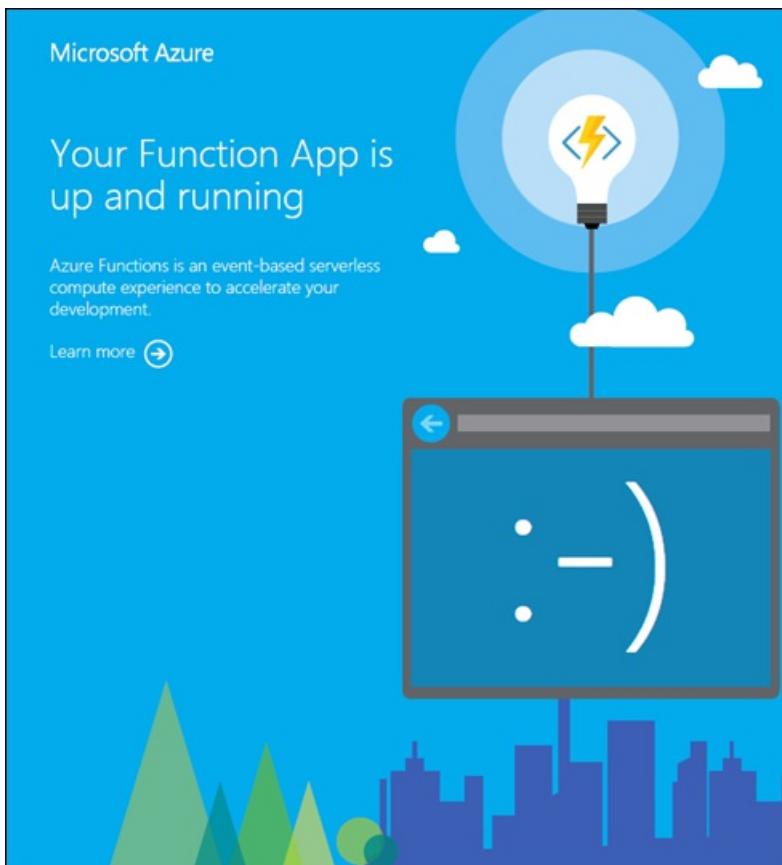
For performance and experience, it is recommended to use APPINSIGHTS_INSTRUMENTATIONKEY and App Insights for monitoring instead of AzureWebJobsDashboard

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
AzureWebJobsDisableHomepage	true

When this app setting is omitted or set to `false`, a page similar to the following example is displayed in response to the URL `<functionappname>.azurewebsites.net`.



AzureWebJobsDotNetReleaseCompilation

`true` means use Release mode when compiling .NET code; `false` means use Debug mode. Default is `true`.

KEY	SAMPLE VALUE
AzureWebJobsDotNetReleaseCompilation	true

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
AzureWebJobsFeatureFlags	feature1,feature2

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
AzureWebJobsSecretStorageType	Files

AzureWebJobsStorage

The Azure Functions runtime uses this storage account connection string for all functions except for HTTP triggered functions. 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_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

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
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING	DefaultEndpointsProtocol=https;AccountName=[name];AccountKey=[key]

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.

NOTE

This setting is a preview feature - and only reliable if set to a value <= 5

KEY	SAMPLE VALUE
WEBSITE_MAX_DYNAMIC_APPLICATION_SCALE_OUT	5

WEBSITE_NODE_DEFAULT_VERSION

Default is "8.11.1".

KEY	SAMPLE VALUE
WEBSITE_NODE_DEFAULT_VERSION	8.11.1

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](#).

AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL

By default Functions proxies will utilize a shortcut to send API calls from proxies directly to functions in the same Function App, rather than creating a new HTTP request. This setting allows you to disable that 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 will no longer be sent directly to the function, and will instead be directed back to the HTTP front end for the Function App
AZURE_FUNCTION_PROXY_DISABLE_LOCAL_CALL	false	This is the default value. Calls with a backend url pointing to a function in the local Function App will be forwarded directly to that Function

AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES

This setting controls whether `%2F` is 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 will have them decoded. <code>example.com/api%2ftest</code> will become <code>example.com/api/test</code>
AZURE_FUNCTION_PROXY_BACKEND_URL_DECODE_SLASHES	false	This is the default behavior. All route parameters will be passed along unchanged

Example

Here is an example `proxies.json` in a function app at the URL `myfunction.com`

```
{
    "$schema": "http://json.schemastore.org/proxies",
    "proxies": {
        "root": {
            "matchCondition": {
                "route": "/{*all}"
            },
            "backendUri": "example.com/{all}"
        }
    }
}
```

URL DECODING	INPUT	OUTPUT
true	myfunction.com/test%2fapi	example.com/test/api
false	myfunction.com/test%2fapi	example.com/test%2fapi

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

7/26/2019 • 27 minutes to read • [Edit Online](#)

This article explains how to work with Azure Blob storage bindings in Azure Functions. Azure Functions supports trigger, input, and output bindings for blobs. The article includes a section for each binding:

- [Blob trigger](#)
- [Blob input binding](#)
- [Blob output binding](#)

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

Use the Event Grid trigger instead of the Blob storage trigger for blob-only storage accounts, for high scale, or to reduce latency. For more information, see the [Trigger](#) section.

Packages - Functions 1.x

The Blob 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](#).

Packages - Functions 2.x

The Blob 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.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Trigger

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 [Event Grid trigger](#) has built-in support for [blob events](#) and can also be used to start a function when a new or updated blob is detected. For an example, see the [Image resize with Event Grid](#) tutorial.

Use Event Grid instead of the Blob storage trigger for the following scenarios:

- Blob storage accounts
- High scale
- Minimizing latency

Blob storage accounts

[Blob storage accounts](#) are supported for blob input and output bindings but not for blob triggers. Blob storage triggers require a general-purpose storage account.

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.

Latency issues

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](#).

Queue storage trigger

Besides Event Grid, another alternative for processing blobs is the Queue storage trigger, but it has no built-in support for blob events. You would have to create queue messages when creating or updating blobs. For an example that assumes you've done that, see the [blob input binding example later in this article](#).

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Trigger - C# example

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 [Trigger - attributes](#).

Trigger - C# script example

The following example shows a blob trigger binding in a `function.json` file and [Python code](#) that uses the binding. The function writes a log when a blob is added or updated in the `samples-workitems` container.

Here's the binding data in the `function.json` file:

```
{
    "disabled": false,
    "bindings": [
        {
            "name": "myBlob",
            "type": "blobTrigger",
            "direction": "in",
            "path": "samples-workitems/{name}",
            "connection": "MyStorageAccountAppSetting"
        }
    ]
}
```

The string `{name}` in the blob trigger path `samples-workitems/{name}` creates a [binding expression](#) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](#) later in this article.

For more information about `function.json` file properties, see the [Configuration](#) section explains these properties.

Here's C# script code that binds to a `Stream`:

```
public static void Run(Stream myBlob, string name, ILogger log)
{
    log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");
}
```

Here's C# script code that binds to a `CloudBlockBlob`:

```
#r "Microsoft.WindowsAzure.Storage"

using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name, ILogger log)
{
    log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name}\nURI:{myBlob.StorageUri}");
}
```

Trigger - JavaScript example

The following example shows a blob trigger binding in a `function.json` file and [JavaScript code](#) that uses the binding. The function writes a log when a blob is added or updated in the `samples-workitems` container.

Here's the `function.json` file:

```
{
  "disabled": false,
  "bindings": [
    {
      "name": "myBlob",
      "type": "blobTrigger",
      "direction": "in",
      "path": "samples-workitems/{name}",
      "connection": "MyStorageAccountAppSetting"
    }
  ]
}
```

The string `{name}` in the blob trigger path `samples-workitems/{name}` creates a [binding expression](#) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](#) later in this article.

For more information about `function.json` file properties, see the [Configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function(context) {
  context.log('Node.js Blob trigger function processed', context.bindings.myBlob);
  context.done();
};
```

Trigger - Python example

The following example shows a blob trigger binding in a `function.json` file and [Python code](#) that uses the binding. The function writes a log when a blob is added or updated in the `samples-workitems` container.

Here's the `function.json` file:

```
{
  "scriptFile": "__init__.py",
  "disabled": false,
  "bindings": [
    {
      "name": "myblob",
      "type": "blobTrigger",
      "direction": "in",
      "path": "samples-workitems/{name}",
      "connection": "MyStorageAccountAppSetting"
    }
  ]
}
```

The string `{name}` in the blob trigger path `samples-workitems/{name}` creates a [binding expression](#) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](#) later in this article.

For more information about `function.json` file properties, see the [Configuration](#) section explains these properties.

Here's the Python code:

```
import logging
import azure.functions as func

def main(myblob: func.InputStream):
    logging.info('Python Blob trigger function processed %s', myblob.name)
```

Trigger - Java example

The following example shows a blob trigger binding in a `function.json` file and [Java code](#) that uses the binding. The function writes a log when a blob is added or updated in the `myblob` container.

Here's the `function.json` file:

```
{
  "disabled": false,
  "bindings": [
    {
      "name": "file",
      "type": "blobTrigger",
      "direction": "in",
      "path": "myblob/{name}",
      "connection": "MyStorageAccountAppSetting"
    }
  ]
}
```

Here's the Java code:

```

@FunctionName("blobprocessor")
public void run(
    @BlobTrigger(name = "file",
        dataType = "binary",
        path = "myblob/{name}",
        connection = "MyStorageAccountAppSetting") byte[] content,
    @BindingName("name") String filename,
    final ExecutionContext context
) {
    context.getLogger().info("Name: " + filename + " Size: " + content.length + " bytes");
}

```

Trigger - attributes

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 - 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("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).

Trigger - 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 .
<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 storage account.</p>

When you're developing locally, app settings go into the [local.settings.json file](#).

Trigger - usage

In C# and C# script, you can use the following parameter types for the triggering blob:

- `Stream`

- `TextReader`
- `string`
- `Byte[]`
- A POCO serializable as JSON
 - `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`, `Byte[]`, or POCO 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.

In JavaScript, access the input blob data using `context.bindings.<name from function.json>`.

Trigger - 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-Blob 1.txt*, the values of the `blobname` and `blobextension` variables in function code are *original-Blob 1* 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-Blob 1.txt*, the value of the `name` variable in function code is `Blob1`.

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`.

Trigger - metadata

The blob trigger provides several metadata properties. These properties can be used as part of binding expressions in other bindings or as parameters in your code. These values have the same semantics as the [CloudBlob](#) type.

PROPERTY	TYPE	DESCRIPTION
<code>BlobTrigger</code>	<code>string</code>	The path to the triggering blob.
<code>Uri</code>	<code>System.Uri</code>	The blob's URI for the primary location.
<code>Properties</code>	<code>BlobProperties</code>	The blob's system properties.
<code>Metadata</code>	<code>IDictionary<string, string></code>	The user-defined metadata for the blob.

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}");
}

module.exports = function (context, myBlob) {
    context.log("Full blob path:", context.bindingData.blobTrigger);
    context.done();
};
```

Trigger - 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.

Trigger - 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 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")

Trigger - 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 * 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 $3 \times 24 = 72$ function invocations.

JavaScript and Java functions load the entire blob into memory, and C# functions do that if you bind to `string`, `Byte[]`, or POCO.

Trigger - polling

If the blob container being monitored contains more than 10,000 blobs (across all containers), the Functions runtime scans log files to watch for new or changed blobs. This process can result in delays. A function might not get triggered until several minutes or longer after the blob is created.

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](#).

Input

Use a Blob storage input binding to read blobs.

Input - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)

- Python

Input - C# example

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");
}
```

Input - C# script example

The following example shows blob input and output bindings in a *function.json* file and [C# script \(.csx\)](#) code that uses the bindings. The function makes a copy of a text blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named *{originalblobname}-Copy*.

In the *function.json* file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
public static void Run(string myQueueItem, string myInputBlob, out string myOutputBlob, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
    myOutputBlob = myInputBlob;
}
```

Input - JavaScript example

The following example shows blob input and output bindings in a *function.json* file and [JavaScript code](#) that uses the bindings. The function makes a copy of a blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named *{originalblobname}-Copy*.

In the *function.json* file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function(context) {
  context.log('Node.js Queue trigger function processed', context.bindings.myQueueItem);
  context.bindings.myOutputBlob = context.bindings.myInputBlob;
  context.done();
};
```

Input - Python example

The following example shows blob input and output bindings in a *function.json* file and [Python code](#) that uses the bindings. The function makes a copy of a blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named *{originalblobname}-Copy*.

In the *function.json* file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "queuemsg",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "inputblob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "$return",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false,
  "scriptFile": "__init__.py"
}
```

The [configuration](#) section explains these properties.

Here's the Python code:

```
import logging
import azure.functions as func

def main(queuemsg: func.QueueMessage, inputblob: func.InputStream) -> func.InputStream:
    logging.info('Python Queue trigger function processed %s', inputblob.name)
    return inputblob
```

Input - Java examples

This section contains the following examples:

- [HTTP trigger, look up blob name from query string](#)
- [Queue trigger, receive blob name from queue message](#)

HTTP trigger, look up blob name from query string (Java)

The following example shows a Java function that uses the `HttpTrigger` annotation to receive a parameter containing the name of a file in a blob storage container. The `BlobInput` annotation then reads the file and passes its contents to the function as a `byte[]`.

```

@FunctionName("getBlobSizeHttp")
@StorageAccount("Storage_Account_Connection_String")
public HttpResponseMessage blobSize(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @BlobInput(
        name = "file",
        dataType = "binary",
        path = "samples-workitems/{Query.file}")
    byte[] content,
    final ExecutionContext context) {
    // build HTTP response with size of requested blob
    return request.createResponseBuilder(HttpStatus.OK)
        .body("The size of '" + request.getQueryParameters().get("file") + "' is: " + content.length + " bytes")
        .build();
}

```

Queue trigger, receive blob name from queue message (Java)

The following example shows a Java function that uses the `QueueTrigger` annotation to receive a message containing the name of a file in a blob storage container. The `BlobInput` annotation then reads the file and passes its contents to the function as a `byte[]`.

```

@FunctionName("getBlobSize")
@StorageAccount("Storage_Account_Connection_String")
public void blobSize(
    @QueueTrigger(
        name = "filename",
        queueName = "myqueue-items-sample")
    String filename,
    @BlobInput(
        name = "file",
        dataType = "binary",
        path = "samples-workitems/{queueTrigger}")
    byte[] content,
    final ExecutionContext context) {
    context.getLogger().info("The size of '" + filename + "' is: " + content.length + " bytes");
}

```

In the [Java functions runtime library](#), use the `@BlobInput` annotation on parameters whose value would come from a blob. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

Input - attributes

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](#).

Input - 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
type	n/a	Must be set to <code>blob</code> .
direction	n/a	Must be set to <code>in</code> . Exceptions are noted in the usage section.
name	n/a	The name of the variable that represents the blob in function code.
path	BlobPath	The path to the blob.
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](#).

Input - usage

In C# and C# script, 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.

In JavaScript, access the blob data using `context.bindings.<name from function.json>`.

Output

Use Blob storage output bindings to write blobs.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Output - C# example

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;

[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(image, out format))
    {
        ResizeImage(input, imageSmall, ImageSize.Small, format);
    }

    image.Position = 0;
    using (Image<Rgba32> input = Image.Load(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) }
};

```

Output - C# script example

The following example shows blob input and output bindings in a `function.json` file and [C# script \(.csx\)](#) code that uses the bindings. The function makes a copy of a text blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named `{originalblobname}-Copy`.

In the `function.json` file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
public static void Run(string myQueueItem, string myInputBlob, out string myOutputBlob, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
    myOutputBlob = myInputBlob;
}
```

Output - JavaScript example

The following example shows blob input and output bindings in a *function.json* file and [JavaScript code](#) that uses the bindings. The function makes a copy of a blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named *{originalblobname}-Copy*.

In the *function.json* file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function(context) {
  context.log('Node.js Queue trigger function processed', context.bindings.myQueueItem);
  context.bindings.myOutputBlob = context.bindings.myInputBlob;
  context.done();
};
```

Output - Python example

The following example shows blob input and output bindings in a *function.json* file and [Python code](#) that uses the bindings. The function makes a copy of a blob. The function is triggered by a queue message that contains the name of the blob to copy. The new blob is named *{originalblobname}-Copy*.

In the *function.json* file, the `queueTrigger` metadata property is used to specify the blob name in the `path` properties:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "queuemsg",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "inputblob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    },
    {
      "name": "outputblob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "out"
    }
  ],
  "disabled": false,
  "scriptFile": "__init__.py"
}
```

The [configuration](#) section explains these properties.

Here's the Python code:

```
import logging
import azure.functions as func

def main(queuemsg: func.QueueMessage, inputblob: func.InputStream,
        outputblob: func.Out(func.InputStream)):
    logging.info('Python Queue trigger function processed %s', inputblob.name)
    outputblob.set(inputblob)
```

Output - Java examples

This section contains the following examples:

- [HTTP trigger, using OutputBinding](#)
- [Queue trigger, using function return value](#)

HTTP trigger, using OutputBinding (Java)

The following example shows a Java function that uses the `HttpTrigger` annotation to receive a parameter containing the name of a file in a blob storage container. The `BlobInput` annotation then reads the file and passes its contents to the function as a `byte[]`. The `BlobOutput` annotation binds to `OutputBinding outputItem`, which is then used by the function to write the contents of the input blob to the configured storage container.

```

@FunctionName("copyBlobHttp")
@StorageAccount("Storage_Account_Connection_String")
public HttpResponseMessage copyBlobHttp(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @BlobInput(
        name = "file",
        dataType = "binary",
        path = "samples-workitems/{Query.file}")
    byte[] content,
    @BlobOutput(
        name = "target",
        path = "myblob/{Query.file}-CopyViaHttp")
    OutputBinding<String> outputItem,
    final ExecutionContext context) {
    // Save blob to outputItem
    outputItem.setValue(new String(content, StandardCharsets.UTF_8));

    // build HTTP response with size of requested blob
    return request.createResponseBuilder(HttpStatus.OK)
        .body("The size of '" + request.getQueryParameters().get("file") + "' is: " + content.length + " bytes")
        .build();
}

```

Queue trigger, using function return value (Java)

The following example shows a Java function that uses the `QueueTrigger` annotation to receive a message containing the name of a file in a blob storage container. The `BlobInput` annotation then reads the file and passes its contents to the function as a `byte[]`. The `BlobOutput` annotation binds to the function return value, which is then used by the runtime to write the contents of the input blob to the configured storage container.

```

@FunctionName("copyBlobQueueTrigger")
@StorageAccount("Storage_Account_Connection_String")
@BlobOutput(
    name = "target",
    path = "myblob/{queueTrigger}-Copy")
public String copyBlobQueue(
    @QueueTrigger(
        name = "filename",
        dataType = "string",
        queueName = "myqueue-items")
    String filename,
    @BlobInput(
        name = "file",
        path = "samples-workitems/{queueTrigger}")
    String content,
    final ExecutionContext context) {
    context.getLogger().info("The content of '" + filename + "' is: " + content);
    return content;
}

```

In the [Java functions runtime library](#), use the `@BlobOutput` annotation on function parameters whose value would be written to an object in blob storage. The parameter type should be `OutputBinding<T>`, where T is any native Java type or a POJO.

Output - attributes

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 - C# example](#).

You can use the `StorageAccount` attribute to specify the storage account at class, method, or parameter level. For more information, see [Trigger - attributes](#).

Output - 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
type	n/a	Must be set to <code>blob</code> .
direction	n/a	Must be set to <code>out</code> for an output binding. Exceptions are noted in the usage section.
name	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.
path	BlobPath	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](#).

Output - usage

In C# and C# script, 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](#).

In async functions, use the return value or `IAsyncCollector` instead of an `out` parameter.

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.

In JavaScript, access the blob data using `context.bindings.<name from function.json>`.

Exceptions and return codes

BINDING	REFERENCE
Blob	Blob Error Codes
Blob, Table, Queue	Storage Error Codes
Blob, Table, Queue	Troubleshooting

Next steps

- [Learn more about Azure functions triggers and bindings](#)

Azure Cosmos DB bindings for Azure Functions 1.x

7/1/2019 • 28 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, 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

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

[Skip trigger examples](#)

Trigger - C# example

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}");
            }
        }
    }
}
```

[Skip trigger examples](#)

Trigger - C# script example

The following example shows a Cosmos DB trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function writes log messages when Cosmos DB records are modified.

Here's the binding data in the *function.json* file:

```
{
    "type": "cosmosDBTrigger",
    "name": "documents",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "<connection-app-setting>",
    "databaseName": "Tasks",
    "collectionName": "Items",
    "createLeaseCollectionIfNotExists": true
}
```

Here's the C# script code:

```
#r "Microsoft.Azure.Documents.Client"

using System;
using Microsoft.Azure.Documents;
using System.Collections.Generic;

public static void Run(IReadOnlyList<Document> documents, TraceWriter log)
{
    log.Info("Documents modified " + documents.Count);
    log.Info("First document Id " + documents[0].Id);
}
```

[Skip trigger examples](#)

Trigger - JavaScript example

The following example shows a Cosmos DB trigger binding in a *function.json* file and a [JavaScript](#) function that uses the binding. The function writes log messages when Cosmos DB records are modified.

Here's the binding data in the *function.json* file:

```
{
    "type": "cosmosDBTrigger",
    "name": "documents",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "<connection-app-setting>",
    "databaseName": "Tasks",
    "collectionName": "Items",
    "createLeaseCollectionIfNotExists": true
}
```

Here's the JavaScript code:

```
module.exports = function (context, documents) {
    context.log('First document Id modified : ', documents[0].id);

    context.done();
}
```

Trigger - attributes

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
type		Must be set to <code>cosmosDBTrigger</code> .
direction		Must be set to <code>in</code> . This parameter is set automatically when you create the trigger in the Azure portal.
name		The variable name used in function code that represents the list of documents with changes.
connectionStringSetting	ConnectionStringSetting	The name of an app setting that contains the connection string used to connect to the Azure Cosmos DB account being monitored.
databaseName	DatabaseName	The name of the Azure Cosmos DB database with the collection being monitored.
collectionName	CollectionName	The name of the collection being monitored.
leaseConnectionStringSetting	LeaseConnectionStringSetting	(Optional) The name of an app setting that contains the connection string to the service which holds the lease collection. When not set, the <code>connectionStringSetting</code> value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases collection must have write permissions.
leaseDatabaseName	LeaseDatabaseName	(Optional) The name of the database that holds the collection used to store leases. When not set, the value of the <code>databaseName</code> setting is used. This parameter is automatically set when the binding is created in the portal.
leaseCollectionName	LeaseCollectionName	(Optional) The name of the collection used to store leases. When not set, the value <code>leases</code> is used.
createLeaseCollectionIfNotExists	CreateLeaseCollectionIfNotExists	(Optional) When set to <code>true</code> , the leases collection is automatically 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 - examples

See the language-specific examples that read a single document by specifying an ID value:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)
- [F#](#)

[Skip input examples](#)

Input - C# examples

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; }
    }
}
```

Skip input examples

Queue trigger, look up ID from JSON (C#)

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}");
            }
        }
    }
}
```

Skip input examples

HTTP trigger, look up ID from query string (C#)

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 = "CosmosDBCConnection",
                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);
        }
    }
}
```

Skip input examples

HTTP trigger, look up ID from route data (C#)

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 (C#)

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 (C#)

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);
        }
    }
}

```

Skip input examples

Input - C# script examples

This section contains the following examples:

- [Queue trigger, look up ID from string](#)
- [Queue trigger, get multiple docs, using SqlQuery](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [HTTP trigger, get multiple docs, using SqlQuery](#)

- [HTTP trigger, get multiple docs, using DocumentClient](#)

The HTTP trigger examples refer to a simple `ToDoItem` type:

```
namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Skip input examples

Queue trigger, look up ID from string (C# script)

The following example shows a Cosmos DB input binding in a `function.json` file and a [C# script function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the `function.json` file:

```
{
    "name": "inputDocument",
    "type": "documentDB",
    "databaseName": "MyDatabase",
    "collectionName": "MyCollection",
    "id" : "{queueTrigger}",
    "partitionKey": "{partition key value}",
    "connection": "MyAccount_COSMOSDB",
    "direction": "in"
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
using System;

// Change input document contents using Azure Cosmos DB input binding
public static void Run(string myQueueItem, dynamic inputDocument)
{
    inputDocument.text = "This has changed.";
}
```

Skip input examples

Queue trigger, get multiple docs, using SqlQuery (C# script)

The following example shows an Azure Cosmos DB input binding in a `function.json` file and a [C# script function](#) that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the `function.json` file:

```
{  
    "name": "documents",  
    "type": "documentdb",  
    "direction": "in",  
    "databaseName": "MyDb",  
    "collectionName": "MyCollection",  
    "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",  
    "connection": "CosmosDBConnection"  
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
public static void Run(QueuePayload myQueueItem, IEnumerable<dynamic> documents)  
{  
    foreach (var doc in documents)  
    {  
        // operate on each document  
    }  
}  
  
public class QueuePayload  
{  
    public string departmentId { get; set; }  
}
```

Skip input examples

HTTP trigger, look up ID from query string (C# script)

The following example shows a [C# script function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a [ToDoItem](#) document from the specified database and collection.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "Id": "{Query.id}"
    }
  ],
  "disabled": true
}
```

Here's the C# script code:

```
using System.Net;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem ToDoItem, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.Info($"ToDo item not found");
    }
    else
    {
        log.Info($"Found ToDo item, Description={ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

Skip input examples

HTTP trigger, look up ID from route data (C# script)

The following example shows a [C# script function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ],
      "route": "todoitems/{id}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "Id": "{id}"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
using System.Net;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem ToDoItem, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.Info($"ToDo item not found");
    }
    else
    {
        log.Info($"Found ToDo item, Description={ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

Skip input examples

HTTP trigger, get multiple docs, using SqlQuery (C# script)

The following example shows a [C# script function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "ToDoItems",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "sqlQuery": "SELECT top 2 * FROM c order by c._ts desc"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
using System.Net;

public static HttpResponseMessage Run(HttpRequestMessage req, IEnumerable<ToDoItem> ToDoItems, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    foreach (ToDoItem ToDoItem in ToDoItems)
    {
        log.Info(ToDoItem.Description);
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

Skip input examples

HTTP trigger, get multiple docs, using DocumentClient (C# script)

The following example shows a [C# script function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

Here's the `function.json` file:

```
{  
  "bindings": [  
    {  
      "authLevel": "anonymous",  
      "name": "req",  
      "type": "httpTrigger",  
      "direction": "in",  
      "methods": [  
        "get",  
        "post"  
      ]  
    },  
    {  
      "name": "$return",  
      "type": "http",  
      "direction": "out"  
    },  
    {  
      "type": "documentDB",  
      "name": "client",  
      "databaseName": "ToDoItems",  
      "collectionName": "Items",  
      "connection": "CosmosDBConnection",  
      "direction": "inout"  
    }  
  ],  
  "disabled": false  
}
```

Here's the C# script code:

```

#r "Microsoft.Azure.Documents.Client"

using System.Net;
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, DocumentClient client, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    Uri collectionUri = UriFactory.CreateDocumentCollectionUri("ToDoItems", "Items");
    string searchterm = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "searchterm", true) == 0)
        .Value;

    if (searchterm == null)
    {
        return req.CreateResponse(HttpStatusCode.NotFound);
    }

    log.Info($"Searching for word: {searchterm} using Uri: {collectionUri.ToString()}");
    IDocumentQuery<ToDoItem> query = client.CreateDocumentQuery<ToDoItem>(collectionUri)
        .Where(p => p.Description.Contains(searchterm))
        .AsDocumentQuery();

    while (query.HasMoreResults)
    {
        foreach (ToDoItem result in await query.ExecuteNextAsync())
        {
            log.Info(result.Description);
        }
    }
    return req.CreateResponse(HttpStatusCode.OK);
}

```

Skip input examples

Input - JavaScript examples

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](#)
- [Queue trigger, get multiple docs, using SqlQuery](#)

Skip input examples

Queue trigger, look up ID from JSON (JavaScript)

The following example shows a Cosmos DB input binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

```
{
  "name": "inputDocumentIn",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger_payload_property}",
  "partitionKey": "{queueTrigger_payload_property}",
  "connection": "MyAccount_COSMOSDB",
  "direction": "in"
},
{
  "name": "inputDocumentOut",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "createIfNotExists": false,
  "partitionKey": "{queueTrigger_payload_property}",
  "connection": "MyAccount_COSMOSDB",
  "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
// Change input document contents using Azure Cosmos DB input binding, using
context.bindings.inputDocumentOut
module.exports = function (context) {
  context.bindings.inputDocumentOut = context.bindings.inputDocumentIn;
  context.bindings.inputDocumentOut.text = "This was updated!";
  context.done();
};
```

Skip input examples

HTTP trigger, look up ID from query string (JavaScript)

The following example shows a [JavaScript function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a [ToDoItem](#) document from the specified database and collection.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "Id": "{Query.id}"
    }
  ],
  "disabled": true
}
```

Here's the JavaScript code:

```
module.exports = function (context, req, ToDoItem) {
  context.log('JavaScript queue trigger function processed work item');
  if (!ToDoItem)
  {
    context.log("ToDo item not found");
  }
  else
  {
    context.log("Found ToDo item, Description=" + ToDoItem.Description);
  }

  context.done();
};
```

Skip input examples

HTTP trigger, look up ID from route data (JavaScript)

The following example shows a [JavaScript function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ],
      "route": "todoitems/{id}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "documentDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "Id": "{id}"
    }
  ],
  "disabled": false
}
```

Here's the JavaScript code:

```
module.exports = function (context, req, ToDoItem) {
  context.log('JavaScript queue trigger function processed work item');
  if (!ToDoItem)
  {
    context.log("ToDo item not found");
  }
  else
  {
    context.log("Found ToDo item, Description=" + ToDoItem.Description);
  }

  context.done();
};
```

Skip input examples

Queue trigger, get multiple docs, using SqlQuery (JavaScript)

The following example shows an Azure Cosmos DB input binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the `function.json` file:

```
{
  "name": "documents",
  "type": "documentdb",
  "direction": "in",
  "databaseName": "MyDb",
  "collectionName": "MyCollection",
  "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",
  "connection": "CosmosDBConnection"
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function (context, input) {
  var documents = context.bindings.documents;
  for (var i = 0; i < documents.length; i++) {
    var document = documents[i];
    // operate on each document
  }
  context.done();
};
```

[Skip input examples](#)

Input - F# examples

The following example shows a Cosmos DB input binding in a *function.json* file and a [F# function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

```
{
  "name": "inputDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger}",
  "connection": "MyAccount_COSMOSDB",
  "direction": "in"
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```
(* Change input document contents using Azure Cosmos DB input binding *)
open FSharp.Interop.Dynamic
let Run(myQueueItem: string, inputDocument: obj) =
  inputDocument?text <- "This has changed."
```

This example requires a `project.json` file that specifies the `FSharp.Interop.Dynamic` and `Dynamitey` NuGet dependencies:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

To add a `project.json` file, see [F# package management](#).

Input - attributes

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
type		Must be set to <code>documentdb</code> .
direction		Must be set to <code>in</code> .
name		Name of the binding parameter that represents the document in the function.
databaseName	DatabaseName	The database containing the document.
collectionName	CollectionName	The name of the collection that contains the document.
id	Id	The ID of the document to retrieve. This property supports binding expressions . Don't set both the id and sqlQuery properties. If you don't set either one, the entire collection is retrieved.
sqlQuery	SqlQuery	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: <code>SELECT * FROM c WHERE c.departmentId = {departmentId}</code> . Don't set both the id and sqlQuery properties. If you don't set either one, the entire collection is retrieved.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
connection	ConnectionStringSetting	The name of the app setting containing your Azure Cosmos DB connection string.
partitionKey	PartitionKey	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

In C# and F# functions, when the function exits successfully, any changes made to the input document via named input parameters are automatically persisted.

In JavaScript functions, updates are not made automatically upon function exit. Instead, use

`context.bindings.<documentName>In` and `context.bindings.<documentName>Out` to make updates. See the [JavaScript example](#).

Output

The Azure Cosmos DB output binding lets you write a new document to an Azure Cosmos DB database using the SQL API.

Output - examples

See the language-specific examples:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)
- [F#](#)

See also the [input example](#) that uses `DocumentClient`.

[Skip output examples](#)

Output - C# examples

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; }
    }
}
```

[Skip output examples](#)

Queue trigger, write one doc (C#)

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}");
        }
    }
}
```

Skip output examples

Queue trigger, write docs using IAsyncCollector (C#)

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);
            }
        }
    }
}
```

Skip output examples

Output - C# script examples

This section contains the following examples:

- Queue trigger, write one doc
- Queue trigger, write docs using IAsyncCollector

Skip output examples

Queue trigger, write one doc (C# script)

The following example shows an Azure Cosmos DB output binding in a *function.json* file and a [C# script function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

```
{  
    "name": "John Henry",  
    "employeeId": "123456",  
    "address": "A town nearby"  
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

```
{  
    "id": "John Henry-123456",  
    "name": "John Henry",  
    "employeeId": "123456",  
    "address": "A town nearby"  
}
```

Here's the binding data in the *function.json* file:

```
{  
    "name": "employeeDocument",  
    "type": "documentDB",  
    "databaseName": "MyDatabase",  
    "collectionName": "MyCollection",  
    "createIfNotExists": true,  
    "connection": "MyAccount_COSMOSDB",  
    "direction": "out"  
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```

#r "Newtonsoft.Json"

using Microsoft.Azure.WebJobs.Host;
using Newtonsoft.Json.Linq;

public static void Run(string myQueueItem, out object employeeDocument, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    dynamic employee = JObject.Parse(myQueueItem);

    employeeDocument = new {
        id = employee.name + "-" + employee.employeeId,
        name = employee.name,
        employeeId = employee.employeeId,
        address = employee.address
    };
}

```

Queue trigger, write docs using IAsyncCollector

To create multiple documents, you can bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

This example refers to a simple `ToDoItem` type:

```

namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}

```

Here's the `function.json` file:

```

{
    "bindings": [
        {
            "name": "ToDoItemsIn",
            "type": "queueTrigger",
            "direction": "in",
            "queueName": "todoqueueforwritemulti",
            "connection": "AzureWebJobsStorage"
        },
        {
            "type": "documentDB",
            "name": "ToDoItemsOut",
            "databaseName": "ToDoItems",
            "collectionName": "Items",
            "connection": "CosmosDBConnection",
            "direction": "out"
        }
    ],
    "disabled": false
}

```

Here's the C# script code:

```
using System;

public static async Task Run(ToDoItem[] ToDoItemsIn, IAsyncCollector<ToDoItem> ToDoItemsOut, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed {ToDoItemsIn?.Length} items");

    foreach (ToDoItem ToDoItem in ToDoItemsIn)
    {
        log.Info($"Description={ToDoItem.Description}");
        await ToDoItemsOut.AddAsync(ToDoItem);
    }
}
```

Skip output examples

Output - JavaScript examples

The following example shows an Azure Cosmos DB output binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

```
{
    "name": "John Henry",
    "employeeId": "123456",
    "address": "A town nearby"
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

```
{
    "id": "John Henry-123456",
    "name": "John Henry",
    "employeeId": "123456",
    "address": "A town nearby"
}
```

Here's the binding data in the *function.json* file:

```
{
    "name": "employeeDocument",
    "type": "documentDB",
    "databaseName": "MyDatabase",
    "collectionName": "MyCollection",
    "createIfNotExists": true,
    "connection": "MyAccount_COSMOSDB",
    "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```

module.exports = function (context) {

    context.bindings.employeeDocument = JSON.stringify({
        id: context.bindings.myQueueItem.name + "-" + context.bindings.myQueueItem.employeeId,
        name: context.bindings.myQueueItem.name,
        employeeId: context.bindings.myQueueItem.employeeId,
        address: context.bindings.myQueueItem.address
    });

    context.done();
};

```

[Skip output examples](#)

Output - F# examples

The following example shows an Azure Cosmos DB output binding in a *function.json* file and an [F# function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

```
{
    "name": "John Henry",
    "employeeId": "123456",
    "address": "A town nearby"
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

```
{
    "id": "John Henry-123456",
    "name": "John Henry",
    "employeeId": "123456",
    "address": "A town nearby"
}
```

Here's the binding data in the *function.json* file:

```
{
    "name": "employeeDocument",
    "type": "documentDB",
    "databaseName": "MyDatabase",
    "collectionName": "MyCollection",
    "createIfNotExists": true,
    "connection": "MyAccount_COSMOSDB",
    "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```

open FSharp.Interop.Dynamic
open Newtonsoft.Json

type Employee = {
    id: string
    name: string
    employeeId: string
    address: string
}

let Run(myQueueItem: string, employeeDocument: byref<obj>, log: TraceWriter) =
    log.Info(sprintf "F# Queue trigger function processed: %s" myQueueItem)
    let employee = JObject.Parse(myQueueItem)
    employeeDocument <-
        { id = sprintf "%s-%s" employee?name employee?employeeId
          name = employee?name
          employeeId = employee?employeeId
          address = employee?address }

```

This example requires a `project.json` file that specifies the `FSharp.Interop.Dynamic` and `Dynamitey` NuGet dependencies:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

To add a `project.json` file, see [F# package management](#).

Output - attributes

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 - C# example](#).

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		Must be set to <code>documentdb</code> .
direction		Must be set to <code>out</code> .
name		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 <i>false</i> 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 bindings for Azure Functions 2.x

7/26/2019 • 42 minutes to read • [Edit Online](#)

This article explains how to work with [Azure Cosmos DB](#) bindings in Azure Functions 2.x. Azure Functions supports trigger, input, and output bindings for Azure Cosmos DB.

NOTE

This article is for [Azure Functions version 2.x](#). 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, the trigger, bindings, and package are all named Cosmos DB.

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](#).

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](#).

Packages - Functions 2.x

The Azure Cosmos DB bindings for Functions version 2.x are provided in the [Microsoft.Azure.WebJobs.Extensions.CosmosDB](#) NuGet package, version 3.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 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

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

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Skip trigger examples

Trigger - C# example

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}");
            }
        }
    }
}
```

Skip trigger examples

Trigger - C# script example

The following example shows a Cosmos DB trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function writes log messages when Cosmos DB records are modified.

Here's the binding data in the *function.json* file:

```
{
  "type": "cosmosDBTrigger",
  "name": "documents",
  "direction": "in",
  "leaseCollectionName": "leases",
  "connectionStringSetting": "<connection-app-setting>",
  "databaseName": "Tasks",
  "collectionName": "Items",
  "createLeaseCollectionIfNotExists": true
}
```

Here's the C# script code:

```
#r "Microsoft.Azure.DocumentDB.Core"

using System;
using Microsoft.Azure.Documents;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

public static void Run(IReadOnlyList<Document> documents, ILogger log)
{
    log.LogInformation("Documents modified " + documents.Count);
    log.LogInformation("First document Id " + documents[0].Id);
}
```

Skip trigger examples

Trigger - JavaScript example

The following example shows a Cosmos DB trigger binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function writes log messages when Cosmos DB records are modified.

Here's the binding data in the *function.json* file:

```
{
  "type": "cosmosDBTrigger",
  "name": "documents",
  "direction": "in",
  "leaseCollectionName": "leases",
  "connectionStringSetting": "<connection-app-setting>",
  "databaseName": "Tasks",
  "collectionName": "Items",
  "createLeaseCollectionIfNotExists": true
}
```

Here's the JavaScript code:

```
module.exports = function (context, documents) {
    context.log('First document Id modified : ', documents[0].id);

    context.done();
}
```

Trigger - Java example

The following example shows a Cosmos DB trigger binding in *function.json* file and a [Java function](#) that uses the binding. The function is involved when there are inserts or updates in the specified database and collection.

```
{
    "type": "cosmosDBTrigger",
    "name": "items",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "AzureCosmosDBConnection",
    "databaseName": "ToDoList",
    "collectionName": "Items",
    "createLeaseCollectionIfNotExists": false
}
```

Here's the Java code:

```
@FunctionName("cosmosDBMonitor")
public void cosmosDbProcessor(
    @CosmosDBTrigger(name = "items",
        databaseName = "ToDoList",
        collectionName = "Items",
        leaseCollectionName = "leases",
        createLeaseCollectionIfNotExists = true,
        connectionStringSetting = "AzureCosmosDBConnection") String[] items,
    final ExecutionContext context) {
    context.getLogger().info(items.length + "item(s) is/are changed.");
}
```

In the [Java functions runtime library](#), use the `@CosmosDBTrigger` annotation on parameters whose value would come from Cosmos DB. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

Skip trigger examples

Trigger - Python example

The following example shows a Cosmos DB trigger binding in a `function.json` file and a [Python function](#) that uses the binding. The function writes log messages when Cosmos DB records are modified.

Here's the binding data in the `function.json` file:

```
{
    "name": "documents",
    "type": "cosmosDBTrigger",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "<connection-app-setting>",
    "databaseName": "Tasks",
    "collectionName": "Items",
    "createLeaseCollectionIfNotExists": true
}
```

Here's the Python code:

```
import logging
import azure.functions as func

def main(documents: func.DocumentList) -> str:
    if documents:
        logging.info('First document Id modified: %s', documents[0]['id'])
```

Trigger - C# attributes

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")]
    IReadableList<Document> documents,
    ILogger 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
type		Must be set to <code>cosmosDBTrigger</code> .
direction		Must be set to <code>in</code> . This parameter is set automatically when you create the trigger in the Azure portal.
name		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.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
leaseConnectionStringSetting	LeaseConnectionStringSetting	(Optional) The name of an app setting that contains the connection string to the service which holds the lease collection. When not set, the <code>connectionStringSetting</code> value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases collection must have write permissions.
leaseDatabaseName	LeaseDatabaseName	(Optional) The name of the database that holds the collection used to store leases. When not set, the value of the <code>databaseName</code> setting is used. This parameter is automatically set when the binding is created in the portal.
leaseCollectionName	LeaseCollectionName	(Optional) The name of the collection used to store leases. When not set, the value <code>leases</code> is used.
createLeaseCollectionIfNotExists	CreateLeaseCollectionIfNotExists	(Optional) When set to <code>true</code> , the leases collection is automatically created when it doesn't already exist. The default value is <code>false</code> .
leasesCollectionThroughput	LeasesCollectionThroughput	(Optional) Defines the amount of Request Units to assign when the leases collection is created. This setting is only used When <code>createLeaseCollectionIfNotExists</code> is set to <code>true</code> . This parameter is automatically set when the binding is created using the portal.
leaseCollectionPrefix	LeaseCollectionPrefix	(Optional) When set, it adds a prefix to the leases created in the Lease collection for this Function, effectively allowing two separate Azure Functions to share the same Lease collection by using different prefixes.
feedPollDelay	FeedPollDelay	(Optional) When set, it defines, in milliseconds, the delay in between polling a partition for new changes on the feed, after all current changes are drained. Default is 5000 (5 seconds).
leaseAcquireInterval	LeaseAcquireInterval	(Optional) When set, it defines, in milliseconds, the interval to kick off a task to compute if partitions are distributed evenly among known host instances. Default is 13000 (13 seconds).

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
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.
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 - examples

See the language-specific examples that read a single document by specifying an ID value:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

[Skip input examples](#)

Input - C# examples

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
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

[Skip input examples](#)

Queue trigger, look up ID from JSON (C#)

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 CosmosDBSamplesV2
{
    public class ToDoItemLookup
    {
        public string ToDoItemId { 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}")][ToDoItem] ToDoItem ToDoItem,
            ILogger log)
        {
            log.LogInformation($"C# Queue trigger function processed Id={ToDoItemLookup?.ToDoItemId}");

            if (ToDoItem == null)
            {
                log.LogInformation($"ToDo item not found");
            }
            else
            {
                log.LogInformation($"Found ToDo item, Description={ToDoItem.Description}");
            }
        }
    }
}

```

Skip input examples

HTTP trigger, look up ID from query string (C#)

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.

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}")] 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();
        }
    }
}

```

Skip input examples

HTTP trigger, look up ID from route data (C#)

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.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/{id}")]HttpRequest req,
            [CosmosDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{id}")] 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();
        }
    }
}

```

Skip input examples

HTTP trigger, look up ID from route data, using `SqlQuery` (C#)

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](#).

```

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();
        }
    }
}

```

Skip input examples

HTTP trigger, get multiple docs, using SqlQuery (C#)

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();
        }
    }
}

```

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.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();
        }
    }
}

```

Skip input examples

Input - C# script examples

This section contains the following examples:

- [Queue trigger, look up ID from string](#)
- [Queue trigger, get multiple docs, using SqlQuery](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [HTTP trigger, get multiple docs, using SqlQuery](#)

- **HTTP trigger, get multiple docs, using DocumentClient**

The HTTP trigger examples refer to a simple `ToDoItem` type:

```
namespace CosmosDBSamplesV2
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Skip input examples

Queue trigger, look up ID from string (C# script)

The following example shows a Cosmos DB input binding in a `function.json` file and a [C# script function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the `function.json` file:

```
{
    "name": "inputDocument",
    "type": "cosmosDB",
    "databaseName": "MyDatabase",
    "collectionName": "MyCollection",
    "id" : "{queueTrigger}",
    "partitionKey": "{partition key value}",
    "connectionStringSetting": "MyAccount_COSMOSDB",
    "direction": "in"
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
using System;

// Change input document contents using Azure Cosmos DB input binding
public static void Run(string myQueueItem, dynamic inputDocument)
{
    inputDocument.text = "This has changed.";
}
```

Skip input examples

Queue trigger, get multiple docs, using SqlQuery (C# script)

The following example shows an Azure Cosmos DB input binding in a `function.json` file and a [C# script function](#) that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the `function.json` file:

```
{  
    "name": "documents",  
    "type": "cosmosDB",  
    "direction": "in",  
    "databaseName": "MyDb",  
    "collectionName": "MyCollection",  
    "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",  
    "connectionStringSetting": "CosmosDBConnection"  
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
public static void Run(QueuePayload myQueueItem, IEnumerable<dynamic> documents)  
{  
    foreach (var doc in documents)  
    {  
        // operate on each document  
    }  
}  
  
public class QueuePayload  
{  
    public string departmentId { get; set; }  
}
```

Skip input examples

HTTP trigger, look up ID from query string (C# script)

The following example shows a [C# script function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a [ToDoItem](#) document from the specified database and collection.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "Id": "{Query.id}"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem ToDoItem, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.LogInformation($"ToDo item not found");
    }
    else
    {
        log.LogInformation($"Found ToDo item, Description={ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

Skip input examples

HTTP trigger, look up ID from route data (C# script)

The following example shows a [C# script function](#) that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ],
      "route": "todoitems/{id}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "Id": "{id}"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage Run(HttpRequestMessage req, ToDoItem ToDoItem, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    if (ToDoItem == null)
    {
        log.LogInformation($"ToDo item not found");
    }
    else
    {
        log.LogInformation($"Found ToDo item, Description={ToDoItem.Description}");
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

Skip input examples

HTTP trigger, get multiple docs, using SqlQuery (C# script)

The following example shows a [C# script function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItems",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "sqlQuery": "SELECT top 2 * FROM c order by c._ts desc"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage Run(HttpRequestMessage req, IEnumerable<ToDoItem> ToDoItems, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    foreach (ToDoItem toItem in ToDoItems)
    {
        log.LogInformation(toItem.Description);
    }
    return req.CreateResponse(HttpStatusCode.OK);
}
```

Skip input examples

HTTP trigger, get multiple docs, using DocumentClient (C# script)

The following example shows a [C# script function](#) that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

Here's the `function.json` file:

```
{  
  "bindings": [  
    {  
      "authLevel": "anonymous",  
      "name": "req",  
      "type": "httpTrigger",  
      "direction": "in",  
      "methods": [  
        "get",  
        "post"  
      ]  
    },  
    {  
      "name": "$return",  
      "type": "http",  
      "direction": "out"  
    },  
    {  
      "type": "cosmosDB",  
      "name": "client",  
      "databaseName": "ToDoItems",  
      "collectionName": "Items",  
      "connectionStringSetting": "CosmosDBConnection",  
      "direction": "inout"  
    }  
  ],  
  "disabled": false  
}
```

Here's the C# script code:

```

#r "Microsoft.Azure.Documents.Client"

using System.Net;
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using Microsoft.Extensions.Logging;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, DocumentClient client, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    Uri collectionUri = UriFactory.CreateDocumentCollectionUri("ToDoItems", "Items");
    string searchterm = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "searchterm", true) == 0)
        .Value;

    if (searchterm == null)
    {
        return req.CreateResponse(HttpStatusCode.NotFound);
    }

    log.LogInformation($"Searching for word: {searchterm} using Uri: {collectionUri.ToString()}");
    IDocumentQuery<ToDoItem> query = client.CreateDocumentQuery<ToDoItem>(collectionUri)
        .Where(p => p.Description.Contains(searchterm))
        .AsDocumentQuery();

    while (query.HasMoreResults)
    {
        foreach (ToDoItem result in await query.ExecuteNextAsync())
        {
            log.LogInformation(result.Description);
        }
    }
    return req.CreateResponse(HttpStatusCode.OK);
}

```

Skip input examples

Input - JavaScript examples

This section contains the following examples that read a single document by specifying an ID value from various sources:

- [Queue trigger, look up ID from JSON](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [Queue trigger, get multiple docs, using SQLQuery](#)

Skip input examples

Queue trigger, look up ID from JSON (JavaScript)

The following example shows a Cosmos DB input binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

```
{
  "name": "inputDocumentIn",
  "type": "cosmosDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger_payload_property}",
  "partitionKey": "{queueTrigger_payload_property}",
  "connectionStringSetting": "MyAccount_COSMOSDB",
  "direction": "in"
},
{
  "name": "inputDocumentOut",
  "type": "cosmosDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "createIfNotExists": false,
  "partitionKey": "{queueTrigger_payload_property}",
  "connectionStringSetting": "MyAccount_COSMOSDB",
  "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
// Change input document contents using Azure Cosmos DB input binding, using
context.bindings.inputDocumentOut
module.exports = function (context) {
  context.bindings.inputDocumentOut = context.bindings.inputDocumentIn;
  context.bindings.inputDocumentOut.text = "This was updated!";
  context.done();
};
```

Skip input examples

HTTP trigger, look up ID from query string (JavaScript)

The following example shows a [JavaScript function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a [ToDoItem](#) document from the specified database and collection.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "Id": "{Query.id}"
    }
  ],
  "disabled": false
}
```

Here's the JavaScript code:

```
module.exports = function (context, req, ToDoItem) {
  context.log('JavaScript queue trigger function processed work item');
  if (!ToDoItem)
  {
    context.log("ToDo item not found");
  }
  else
  {
    context.log("Found ToDo item, Description=" + ToDoItem.Description);
  }

  context.done();
};
```

Skip input examples

HTTP trigger, look up ID from route data (JavaScript)

The following example shows a [JavaScript function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ],
      "route": "todoitems/{id}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItem",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "Id": "{id}"
    }
  ],
  "disabled": false
}
```

Here's the JavaScript code:

```
module.exports = function (context, req, ToDoItem) {
  context.log('JavaScript queue trigger function processed work item');
  if (!ToDoItem)
  {
    context.log("ToDo item not found");
  }
  else
  {
    context.log("Found ToDo item, Description=" + ToDoItem.Description);
  }

  context.done();
};
```

Skip input examples

Queue trigger, get multiple docs, using SqlQuery (JavaScript)

The following example shows an Azure Cosmos DB input binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the `function.json` file:

```
{  
    "name": "documents",  
    "type": "cosmosDB",  
    "direction": "in",  
    "databaseName": "MyDb",  
    "collectionName": "MyCollection",  
    "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",  
    "connectionStringSetting": "CosmosDBConnection"  
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function (context, input) {  
    var documents = context.bindings.documents;  
    for (var i = 0; i < documents.length; i++) {  
        var document = documents[i];  
        // operate on each document  
    }  
    context.done();  
};
```

[Skip input examples](#)

Input - Python examples

This section contains the following examples that read a single document by specifying an ID value from various sources:

- [Queue trigger, look up ID from JSON](#)
- [HTTP trigger, look up ID from query string](#)
- [HTTP trigger, look up ID from route data](#)
- [Queue trigger, get multiple docs, using SqlQuery](#)

[Skip input examples](#)

Queue trigger, look up ID from JSON (Python)

The following example shows a Cosmos DB input binding in a *function.json* file and a [Python function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

```
{  
    "name": "documents",  
    "type": "cosmosDB",  
    "databaseName": "MyDatabase",  
    "collectionName": "MyCollection",  
    "id" : "{queueTrigger_payload_property}",  
    "partitionKey": "{queueTrigger_payload_property}",  
    "connectionStringSetting": "MyAccount_COSMOSDB",  
    "direction": "in"  
},  
{  
    "name": "$return",  
    "type": "cosmosDB",  
    "databaseName": "MyDatabase",  
    "collectionName": "MyCollection",  
    "createIfNotExists": false,  
    "partitionKey": "{queueTrigger_payload_property}",  
    "connectionStringSetting": "MyAccount_COSMOSDB",  
    "direction": "out"  
}
```

The [configuration](#) section explains these properties.

Here's the Python code:

```
import azure.functions as func  
  
def main(queuemsg: func.QueueMessage, documents: func.DocumentList) -> func.Document:  
    if documents:  
        document = documents[0]  
        document['text'] = 'This was updated!'  
        return document
```

Skip input examples

HTTP trigger, look up ID from query string (Python)

The following example shows a [Python function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a [ToDoItem](#) document from the specified database and collection.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "todoitems",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "in",
      "Id": "{Query.id}"
    }
  ],
  "disabled": true,
  "scriptFile": "__init__.py"
}
```

Here's the Python code:

```
import logging
import azure.functions as func

def main(req: func.HttpRequest, todoitems: func.DocumentList) -> str:
    if not todoitems:
        logging.warning("ToDo item not found")
    else:
        logging.info("Found ToDo item, Description=%s",
                    todoitems[0]['description'])

    return 'OK'
```

Skip input examples

HTTP trigger, look up ID from route data (Python)

The following example shows a [Python function](#) that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a [ToDoItem](#) document from the specified database and collection.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ],
      "route": "todoitems/{id}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "type": "cosmosDB",
      "name": "todoitems",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connection": "CosmosDBConnection",
      "direction": "in",
      "Id": "{id}"
    }
  ],
  "disabled": false,
  "scriptFile": "__init__.py"
}
```

Here's the Python code:

```
import logging
import azure.functions as func

def main(req: func.HttpRequest, todoitems: func.DocumentList) -> str:
    if not todoitems:
        logging.warning("ToDo item not found")
    else:
        logging.info("Found ToDo item, Description=%s",
                    todoitems[0]['description'])
    return 'OK'
```

Skip input examples

Queue trigger, get multiple docs, using SqlQuery (Python)

The following example shows an Azure Cosmos DB input binding in a `function.json` file and a [Python function](#) that uses the binding. The function retrieves multiple documents specified by a SQL query, using a queue trigger to customize the query parameters.

The queue trigger provides a parameter `departmentId`. A queue message of `{ "departmentId" : "Finance" }` would return all records for the finance department.

Here's the binding data in the `function.json` file:

```
{
  "name": "documents",
  "type": "cosmosDB",
  "direction": "in",
  "databaseName": "MyDb",
  "collectionName": "MyCollection",
  "sqlQuery": "SELECT * from c where c.departmentId = {departmentId}",
  "connectionStringSetting": "CosmosDBConnection"
}
```

The [configuration](#) section explains these properties.

Here's the Python code:

```
import azure.functions as func

def main(queuemsg: func.QueueMessage, documents: func.DocumentList):
    for document in documents:
        # operate on each document
```

[Skip input examples](#)

Input - F# examples

The following example shows a Cosmos DB input binding in a *function.json* file and a [F# function](#) that uses the binding. The function reads a single document and updates the document's text value.

Here's the binding data in the *function.json* file:

```
{
  "name": "inputDocument",
  "type": "cosmosDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger}",
  "connectionStringSetting": "MyAccount_COSMOSDB",
  "direction": "in"
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```
(* Change input document contents using Azure Cosmos DB input binding *)
open FSharp.Interop.Dynamic
let Run(myQueueItem: string, inputDocument: obj) =
    inputDocument?text <- "This has changed."
```

This example requires a `project.json` file that specifies the `FSharp.Interop.Dynamic` and `Dynamitey` NuGet dependencies:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

To add a `project.json` file, see [F# package management](#).

Input - Java examples

This section contains the following examples:

- [HTTP trigger, look up ID from query string - String parameter](#)
- [HTTP trigger, look up ID from query string - POJO parameter](#)
- [HTTP trigger, look up ID from route data](#)
- [HTTP trigger, look up ID from route data, using SqlCommand](#)
- [HTTP trigger, get multiple docs from route data, using SqlCommand](#)

The examples refer to a simple `ToDoItem` type:

```
public class ToDoItem {

  private String id;
  private String description;

  public String getId() {
    return id;
  }

  public String getDescription() {
    return description;
  }

  @Override
  public String toString() {
    return "ToDoItem={id=" + id + ",description=" + description + "}";
  }
}
```

HTTP trigger, look up ID from query string - String parameter (Java)

The following example shows a Java function that retrieves a single document. The function is triggered by a HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a document from the specified database and collection, in String form.

```

public class DocByIdFromQueryString {

    @FunctionName("DocByIdFromQueryString")
    public HttpResponseMessage run(
        @HttpTrigger(name = "req",
            methods = {HttpMethod.GET, HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<String>> request,
        @CosmosDBInput(name = "database",
            databaseName = "ToDoList",
            collectionName = "Items",
            id = "{Query.id}",
            partitionKey = "{Query.id}",
            connectionStringSetting = "Cosmos_DB_Connection_String")
        Optional<String> item,
        final ExecutionContext context) {

        // Item list
        context.getLogger().info("Parameters are: " + request.getQueryParameters());
        context.getLogger().info("String from the database is " + (item.isPresent() ? item.get() : null));

        // Convert and display
        if (!item.isPresent()) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
                .body("Document not found.")
                .build();
        }
        else {
            // return JSON from Cosmos. Alternatively, we can parse the JSON string
            // and return an enriched JSON object.
            return request.createResponseBuilder(HttpStatus.OK)
                .header("Content-Type", "application/json")
                .body(item.get())
                .build();
        }
    }
}

```

In the [Java functions runtime library](#), use the `@CosmosDBInput` annotation on function parameters whose value would come from Cosmos DB. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

HTTP trigger, look up ID from query string - POJO parameter (Java)

The following example shows a Java function that retrieves a single document. The function is triggered by a HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a document from the specified database and collection. The document is then converted to an instance of the `ToDoItem` POJO previously created, and passed as an argument to the function.

```

public class DocByIdFromQueryStringPojo {

    @FunctionName("DocByIdFromQueryStringPojo")
    public HttpResponseMessage run(
        @HttpTrigger(name = "req",
            methods = {HttpMethod.GET, HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<String>> request,
        @CosmosDBInput(name = "database",
            databaseName = "ToDoList",
            collectionName = "Items",
            id = "{Query.id}",
            partitionKey = "{Query.id}",
            connectionStringSetting = "Cosmos_DB_Connection_String")
        ToDoItem item,
        final ExecutionContext context) {

        // Item list
        context.getLogger().info("Parameters are: " + request.getQueryParameters());
        context.getLogger().info("Item from the database is " + item);

        // Convert and display
        if (item == null) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
                .body("Document not found.")
                .build();
        }
        else {
            return request.createResponseBuilder(HttpStatus.OK)
                .header("Content-Type", "application/json")
                .body(item)
                .build();
        }
    }
}

```

HTTP trigger, look up ID from route data (Java)

The following example shows a Java function that retrieves a single document. The function is triggered by a HTTP request that uses a route parameter to specify the ID to look up. That ID is used to retrieve a document from the specified database and collection, returning it as an `Optional<String>`.

```

public class DocByIdFromRoute {

    @FunctionName("DocByIdFromRoute")
    public HttpResponseMessage run(
        @HttpTrigger(name = "req",
            methods = {HttpMethod.GET, HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS,
            route = "todoitems/{id}")
        HttpRequestMessage<Optional<String>> request,
        @CosmosDBInput(name = "database",
            databaseName = "ToDoList",
            collectionName = "Items",
            id = "{id}",
            partitionKey = "{id}",
            connectionStringSetting = "Cosmos_DB_Connection_String")
        Optional<String> item,
        final ExecutionContext context) {

        // Item list
        context.getLogger().info("Parameters are: " + request.getQueryParameters());
        context.getLogger().info("String from the database is " + (item.isPresent() ? item.get() : null));

        // Convert and display
        if (!item.isPresent()) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
                .body("Document not found.")
                .build();
        }
        else {
            // return JSON from Cosmos. Alternatively, we can parse the JSON string
            // and return an enriched JSON object.
            return request.createResponseBuilder(HttpStatus.OK)
                .header("Content-Type", "application/json")
                .body(item.get())
                .build();
        }
    }
}

```

HTTP trigger, look up ID from route data, using SqlDataReader (Java)

The following example shows a Java function that retrieves a single document. The function is triggered by a HTTP request that uses a route parameter to specify the ID to look up. That ID is used to retrieve a document from the specified database and collection, converting the result set to a `ToDoItem[]`, since many documents may be returned, depending on the query criteria.

```

public class DocByIdFromRouteSqlQuery {

    @FunctionName("DocByIdFromRouteSqlQuery")
    public HttpResponseMessage run(
        @HttpTrigger(name = "req",
            methods = {HttpMethod.GET, HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS,
            route = "todoitems2/{id}")
        HttpRequestMessage<Optional<String>> request,
        @CosmosDBInput(name = "database",
            databaseName = "ToDoList",
            collectionName = "Items",
            sqlQuery = "select * from Items r where r.id = {id}",
            connectionStringSetting = "Cosmos_DB_Connection_String")
        ToDoItem[] item,
        final ExecutionContext context) {

        // Item list
        context.getLogger().info("Parameters are: " + request.getQueryParameters());
        context.getLogger().info("Items from the database are " + item);

        // Convert and display
        if (item == null) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
                .body("Document not found.")
                .build();
        }
        else {
            return request.createResponseBuilder(HttpStatus.OK)
                .header("Content-Type", "application/json")
                .body(item)
                .build();
        }
    }
}

```

HTTP trigger, get multiple docs from route data, using SqlQuery (Java)

The following example shows a Java function that multiple documents. The function is triggered by a HTTP request that uses a route parameter `desc` to specify the string to search for in the `description` field. The search term is used to retrieve a collection of documents from the specified database and collection, converting the result set to a `ToDoItem[]` and passing it as an argument to the function.

```

public class DocsFromRouteSqlQuery {

    @FunctionName("DocsFromRouteSqlQuery")
    public HttpResponseMessage run(
        @HttpTrigger(name = "req",
            methods = {HttpMethod.GET},
            authLevel = AuthorizationLevel.ANONYMOUS,
            route = "todoitems3/{desc}")
        HttpRequestMessage<Optional<String>> request,
        @CosmosDBInput(name = "database",
            databaseName = "ToDoList",
            collectionName = "Items",
            sqlQuery = "select * from Items r where contains(r.description, {desc})",
            connectionStringSetting = "Cosmos_DB_Connection_String")
        ToDoItem[] items,
        final ExecutionContext context) {

        // Item list
        context.getLogger().info("Parameters are: " + request.getQueryParameters());
        context.getLogger().info("Number of items from the database is " + (items == null ? 0 : items.length));

        // Convert and display
        if (items == null) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
                .body("No documents found.")
                .build();
        }
        else {
            return request.createResponseBuilder(HttpStatus.OK)
                .header("Content-Type", "application/json")
                .body(items)
                .build();
        }
    }
}

```

Input - attributes

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](#).

Input - 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
type		Must be set to <code>cosmosDB</code> .
direction		Must be set to <code>in</code> .
name		Name of the binding parameter that represents the document in the function.
databaseName	DatabaseName	The database containing the document.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
collectionName	CollectionName	The name of the collection that contains the document.
id	Id	The ID of the document to retrieve. This property supports binding expressions . Don't set both the id and sqlQuery properties. If you don't set either one, the entire collection is retrieved.
sqlQuery	SqlQuery	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: <div style="border: 1px solid black; padding: 5px;"> <pre>SELECT * FROM c WHERE c.departmentId = {departmentId}</pre> </div> . Don't set both the id and sqlQuery properties. If you don't set either one, the entire collection is retrieved.
connectionStringSetting	ConnectionStringSetting	The name of the app setting containing your Azure Cosmos DB connection string.
partitionKey	PartitionKey	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

In C# and F# functions, when the function exits successfully, any changes made to the input document via named input parameters are automatically persisted.

In JavaScript functions, updates are not made automatically upon function exit. Instead, use

`context.bindings.<documentName>In` and `context.bindings.<documentName>Out` to make updates. See the [JavaScript example](#).

Output

The Azure Cosmos DB output binding lets you write a new document to an Azure Cosmos DB database using the SQL API.

Output - examples

See the language-specific examples:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)

See also the [input example](#) that uses `DocumentClient`.

[Skip output examples](#)

Output - C# examples

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; }
    }
}
```

[Skip output examples](#)

Queue trigger, write one doc (C#)

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}");
        }
    }
}
```

[Skip output examples](#)

Queue trigger, write docs using `IAsyncCollector` (C#)

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);
            }
        }
    }
}

```

Skip output examples

Output - C# script examples

This section contains the following examples:

- Queue trigger, write one doc
- Queue trigger, write docs using IAsyncCollector

Skip output examples

Queue trigger, write one doc (C# script)

The following example shows an Azure Cosmos DB output binding in a *function.json* file and a [C# script function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

```
{
    "name": "John Henry",
    "employeeId": "123456",
    "address": "A town nearby"
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

```
{
    "id": "John Henry-123456",
    "name": "John Henry",
    "employeeId": "123456",
    "address": "A town nearby"
}
```

Here's the binding data in the *function.json* file:

```
{
    "name": "employeeDocument",
    "type": "cosmosDB",
    "databaseName": "MyDatabase",
    "collectionName": "MyCollection",
    "createIfNotExists": true,
    "connectionStringSetting": "MyAccount_COSMOSDB",
    "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
#r "Newtonsoft.Json"

using Microsoft.Azure.WebJobs.Host;
using Newtonsoft.Json.Linq;
using Microsoft.Extensions.Logging;

public static void Run(string myQueueItem, out object employeeDocument, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");

    dynamic employee = JObject.Parse(myQueueItem);

    employeeDocument = new {
        id = employee.name + "-" + employee.employeeId,
        name = employee.name,
        employeeId = employee.employeeId,
        address = employee.address
    };
}
```

Queue trigger, write docs using `IAsyncCollector`

To create multiple documents, you can bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

This example refers to a simple `ToDoItem` type:

```
namespace CosmosDBSamplesV2
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Here's the function.json file:

```
{
  "bindings": [
    {
      "name": "ToDoItemsIn",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "todoqueueforwritemulti",
      "connectionStringSetting": "AzureWebJobsStorage"
    },
    {
      "type": "cosmosDB",
      "name": "ToDoItemsOut",
      "databaseName": "ToDoItems",
      "collectionName": "Items",
      "connectionStringSetting": "CosmosDBConnection",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
using System;
using Microsoft.Extensions.Logging;

public static async Task Run(ToDoItem[] ToDoItemsIn, IAsyncCollector<ToDoItem> ToDoItemsOut, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed {ToDoItemsIn?.Length} items");

    foreach (ToDoItem ToDoItem in ToDoItemsIn)
    {
        log.LogInformation($"Description={ToDoItem.Description}");
        await ToDoItemsOut.AddAsync(ToDoItem);
    }
}
```

[Skip output examples](#)

Output - JavaScript examples

The following example shows an Azure Cosmos DB output binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

```
{
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

```
{
  "id": "John Henry-123456",
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

Here's the binding data in the *function.json* file:

```
{
  "name": "employeeDocument",
  "type": "cosmosDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "createIfNotExists": true,
  "connectionStringSetting": "MyAccount_COSMOSDB",
  "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function (context) {

  context.bindings.employeeDocument = JSON.stringify({
    id: context.bindings.myQueueItem.name + "-" + context.bindings.myQueueItem.employeeId,
    name: context.bindings.myQueueItem.name,
    employeeId: context.bindings.myQueueItem.employeeId,
    address: context.bindings.myQueueItem.address
  });

  context.done();
};
```

Skip output examples

Output - F# examples

The following example shows an Azure Cosmos DB output binding in a *function.json* file and an [F# function](#) that uses the binding. The function uses a queue input binding for a queue that receives JSON in the following format:

```
{
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

The function creates Azure Cosmos DB documents in the following format for each record:

```
{
  "id": "John Henry-123456",
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

Here's the binding data in the *function.json* file:

```
{
  "name": "employeeDocument",
  "type": "cosmosDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "createIfNotExists": true,
  "connectionStringSetting": "MyAccount_COSMOSDB",
  "direction": "out"
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```
open FSharp.Interop.Dynamic
open Newtonsoft.Json
open Microsoft.Extensions.Logging

type Employee = {
    id: string
    name: string
    employeeId: string
    address: string
}

let Run(myQueueItem: string, employeeDocument: byref<obj>, log: ILogger) =
    log.LogInformation(sprintf "F# Queue trigger function processed: %s" myQueueItem)
    let employee = JObject.Parse(myQueueItem)
    employeeDocument <-
        { id = sprintf "%s-%s" employee?name employee?employeeId
          name = employee?name
          employeeId = employee?employeeId
          address = employee?address }
```

This example requires a `project.json` file that specifies the `FSharp.Interop.Dynamic` and `Dynamitey` NuGet dependencies:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

To add a `project.json` file, see [F# package management](#).

Output - Java examples

- [Queue trigger, save message to database via return value](#)
- [HTTP trigger, save one document to database via return value](#)
- [HTTP trigger, save one document to database via OutputBinding](#)
- [HTTP trigger, save multiple documents to database via OutputBinding](#)

Queue trigger, save message to database via return value (Java)

The following example shows a Java function that adds a document to a database with data from a message in Queue storage.

```

@FunctionName("getItem")
@CosmosDBOutput(name = "database",
    databaseName = "ToDoList",
    collectionName = "Items",
    connectionStringSetting = "AzureCosmosDBConnection")
public String cosmosDbQueryById(
    @QueueTrigger(name = "msg",
        queueName = "myqueue-items",
        connection = "AzureWebJobsStorage")
    String message,
    final ExecutionContext context) {
    return "{ id: \"\" + System.currentTimeMillis() + "\", Description: " + message + " }";
}

```

HTTP trigger, save one document to database via return value (Java)

The following example shows a Java function whose signature is annotated with `@CosmosDBOutput` and has return value of type `String`. The JSON document returned by the function will be automatically written to the corresponding CosmosDB collection.

```

@FunctionName("WriteOneDoc")
@CosmosDBOutput(name = "database",
    databaseName = "ToDoList",
    collectionName = "Items",
    connectionStringSetting = "Cosmos_DB_Connection_String")
public String run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET, HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context) {

    // Item list
    context.getLogger().info("Parameters are: " + request.getQueryParameters());

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

    // Generate random ID
    final int id = Math.abs(new Random().nextInt());

    // Generate document
    final String jsonDocument = "{\"id\":\"" + id + "\", " +
        "\"description\": \"\" + name + "\"}";

    context.getLogger().info("Document to be saved: " + jsonDocument);

    return jsonDocument;
}

```

HTTP trigger, save one document to database via OutputBinding (Java)

The following example shows a Java function that writes a document to CosmosDB via an `OutputBinding<T>` output parameter. Note that, in this setup, it is the `outputItem` parameter that needs to be annotated with `@CosmosDBOutput`, not the function signature. Using `OutputBinding<T>` lets your function take advantage of the binding to write the document to CosmosDB while also allowing returning a different value to the function caller, such as a JSON or XML document.

```

@FunctionName("WriteOneDocOutputBinding")
public HttpResponseMessage run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET, HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @CosmosDBOutput(name = "database",
        databaseName = "ToDoList",
        collectionName = "Items",
        connectionStringSetting = "Cosmos_DB_Connection_String")
    OutputBinding<String> outputItem,
    final ExecutionContext context) {

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

    // Item list
    context.getLogger().info("Parameters are: " + request.getQueryParameters());

    // Generate random ID
    final int id = Math.abs(new Random().nextInt());

    // Generate document
    final String jsonDocument = "{\"id\":\"" + id + "\", " +
        "\"description\": \"\" + name + "\"}";

    context.getLogger().info("Document to be saved: " + jsonDocument);

    // Set outputItem's value to the JSON document to be saved
    outputItem.setValue(jsonDocument);

    // return a different document to the browser or calling client.
    return request.createResponseBuilder(HttpStatus.OK)
        .body("Document created successfully.")
        .build();
}

```

HTTP trigger, save multiple documents to database via OutputBinding (Java)

The following example shows a Java function that writes multiple documents to CosmosDB via an `OutputBinding<T>` output parameter. Note that, in this setup, it is the `outputItem` parameter that needs to be annotated with `@CosmosDBOutput`, not the function signature. The output parameter, `outputItem` has a list of `ToDoItem` objects as its template parameter type. Using `OutputBinding<T>` lets your function take advantage of the binding to write the documents to CosmosDB while also allowing returning a different value to the function caller, such as a JSON or XML document.

```

@FunctionName("WriteMultipleDocsOutputBinding")
public HttpResponseMessage run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET, HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    @CosmosDBOutput(name = "database",
        databaseName = "ToDoList",
        collectionName = "Items",
        connectionStringSetting = "Cosmos_DB_Connection_String")
    OutputBinding<List<ToDoItem>> outputItem,
    final ExecutionContext context) {

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

    // Item list
    context.getLogger().info("Parameters are: " + request.getQueryParameters());

    // Generate documents
    List<ToDoItem> items = new ArrayList<>();

    for (int i = 0; i < 5; i++) {
        // Generate random ID
        final int id = Math.abs(new Random().nextInt());

        // Create ToDoItem
        ToDoItem item = new ToDoItem(String.valueOf(id), name);

        items.add(item);
    }

    // Set outputItem's value to the list of POJOs to be saved
    outputItem.setValue(items);
    context.getLogger().info("Document to be saved: " + items);

    // return a different document to the browser or calling client.
    return request.createResponseBuilder(HttpStatus.OK)
        .body("Documents created successfully.")
        .build();
}

```

In the [Java functions runtime library](#), use the `@CosmosDBOutput` annotation on parameters that will be written to Cosmos DB. The annotation parameter type should be `OutputBinding<T>`, where T is either a native Java type or a POJO.

Output - attributes

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)
{
    ...
}
```

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 `CosmosDB` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
type		Must be set to <code>cosmosDB</code> .
direction		Must be set to <code>out</code> .
name		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.
connectionStringSetting	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

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 <code>Direct</code> and <code>Gateway</code>
Protocol	Https	The connection protocol used by the function when connection to the Azure Cosmos DB service. Read here for an explanation of both modes
leasePrefix	n/a	Lease prefix to use across all functions in an app.

Next steps

- [Learn more about serverless database computing with Cosmos DB](#)
- [Learn more about Azure functions triggers and bindings](#)

Event Grid trigger for Azure Functions

7/1/2019 • 17 minutes to read • [Edit Online](#)

This article explains how to handle [Event Grid](#) events in Azure Functions.

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 article, you learn how to use an Event Grid trigger to invoke a function when an event is received from Event Grid.

If you prefer, you can use an HTTP trigger to handle Event Grid Events; see [Use an HTTP trigger as an Event Grid trigger](#) later in this article. 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.

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

The Event Grid trigger is provided in the [Microsoft.Azure.WebJobs.Extensions.EventGrid](#) NuGet package, version 2.x. Source code for the package is in the [azure-functions-eventgrid-extension](#) GitHub repository.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Packages - Functions 1.x

The Event Grid trigger is provided in the [Microsoft.Azure.WebJobs.Extensions.EventGrid](#) NuGet package, version 1.x. Source code for the package is in the [azure-functions-eventgrid-extension](#) 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

Example

See the language-specific example for an Event Grid trigger:

- [C#](#)
- [C# script \(.csx\)](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

For an HTTP trigger example, see [How to use HTTP trigger](#) later in this article.

C# (2.x)

The following example shows a Functions 2.x [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](#).

C# (Version 1.x)

The following example shows a Functions 1.x [C# function](#) that binds to `IObject`:

```

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));
        }
    }
}

```

C# script example

The following example shows a trigger binding in a *function.json* file and a [C# script function](#) that uses the binding.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "type": "eventGridTrigger",
      "name": "eventGridEvent",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

C# script (Version 2.x)

Here's Functions 2.x C# script code that binds to `EventGridEvent`:

```

#r "Microsoft.Azure.EventGrid"
using Microsoft.Azure.EventGrid.Models;
using Microsoft.Extensions.Logging;

public static void Run(EventGridEvent eventGridEvent, ILogger log)
{
    log.LogInformation(eventGridEvent.Data.ToString());
}

```

For more information, see [Packages](#), [Attributes](#), [Configuration](#), and [Usage](#).

C# script (Version 1.x)

Here's Functions 1.x C# script code that binds to `JObject`:

```
#r "Newtonsoft.Json"

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

public static void Run(JObject eventGridEvent, TraceWriter log)
{
    log.Info(eventGridEvent.ToString(Formatting.Indented));
}
```

JavaScript example

The following example shows a trigger binding in a *function.json* file and a [JavaScript function](#) that uses the binding.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "type": "eventGridTrigger",
      "name": "eventGridEvent",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here's the JavaScript code:

```
module.exports = function (context, eventGridEvent) {
    context.log("JavaScript Event Grid function processed a request.");
    context.log("Subject: " + eventGridEvent.subject);
    context.log("Time: " + eventGridEvent.eventTime);
    context.log("Data: " + JSON.stringify(eventGridEvent.data));
    context.done();
};
```

Python example

The following example shows a trigger binding in a *function.json* file and a [Python function](#) that uses the binding.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "type": "eventGridTrigger",
      "name": "event",
      "direction": "in"
    }
  ],
  "disabled": false,
  "scriptFile": "__init__.py"
}
```

Here's the Python code:

```

import logging
import azure.functions as func


def main(event: func.EventGridEvent):
    logging.info("Python Event Grid function processed a request.")
    logging.info(" Subject: %s", event.subject)
    logging.info(" Time: %s", event.event_time)
    logging.info(" Data: %s", event.get_json())

```

Trigger - Java examples

This section contains the following examples:

- [Event Grid trigger, String parameter](#)
- [Event Grid trigger, POJO parameter](#)

The following examples show trigger binding in a `function.json` file and [Java functions](#) that use the binding and print out an event, first receiving the event as `String` and second as a POJO.

```
{
  "bindings": [
    {
      "type": "eventGridTrigger",
      "name": "eventGridEvent",
      "direction": "in"
    }
  ]
}
```

Event Grid trigger, String parameter (Java)

```

@FunctionName("eventGridMonitorString")
public void logEvent(
    @EventGridTrigger(
        name = "event"
    )
    String content,
    final ExecutionContext context) {
    // log
    context.getLogger().info("Event content: " + content);
}

```

Event Grid trigger, POJO parameter (Java)

This example uses the following POJO, representing the top-level properties of an Event Grid event:

```

import java.util.Date;
import java.util.Map;

public class EventSchema {

    public String topic;
    public String subject;
    public String eventType;
    public Date eventTime;
    public String id;
    public String dataVersion;
    public String metadataVersion;
    public Map<String, Object> data;

}

```

Upon arrival, the event's JSON payload is de-serialized into the `EventSchema` POJO for use by the function. This allows the function to access the event's properties in an object-oriented way.

```
@FunctionName("eventGridMonitor")
public void logEvent(
    @EventGridTrigger(
        name = "event"
    )
    EventSchema event,
    final ExecutionContext context) {
    // log
    context.getLogger().info("Event content: ");
    context.getLogger().info("Subject: " + event.subject);
    context.getLogger().info("Time: " + event.eventTime); // automatically converted to Date by the
    runtime
    context.getLogger().info("Id: " + event.id);
    context.getLogger().info("Data: " + event.data);
}
```

In the [Java functions runtime library](#), use the `EventGridTrigger` annotation on parameters whose value would come from EventGrid. Parameters with these annotations cause the function to run when an event arrives. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

Attributes

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
type	Required - must be set to <code>eventGridTrigger</code> .
direction	Required - must be set to <code>in</code> .
name	Required - the variable name used in function code for the parameter that receives the event data.

Usage

For C# and F# functions in Azure Functions 1.x, you can use the following parameter types for the Event Grid trigger:

- `JObject`
- `string`

For C# and F# functions in Azure Functions 2.x, 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`. For information about how to reference NuGet packages in a C# script function, see [Using NuGet packages](#)

For JavaScript functions, the parameter named by the `function.json` `name` property has a reference to the event object.

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": "0x8D5628310444DD0",
    "contentType": "application/octet-stream",
    "contentLength": 2248,
    "blobType": "BlockBlob",
    "url": "https://egblobstore.blob.core.windows.net/{containername}/blobname.jpg",
    "sequencer": "00000000000272D000000000003D60F",
    "storageDiagnostics": {
      "batchId": "{guid}"
    },
    "dataVersion": "",
    "metadataVersion": "1"
  }
}]
```

The example shown is an array of one element. Event Grid always sends an array and may send more than one event in the array. The runtime invokes your function once for each array element.

The top-level properties in the event JSON data are the same among all event types, while the contents of the `data` property are specific to each event type. The example shown is for a blob storage event.

For explanations of the common and event-specific properties, see [Event properties](#) in the Event Grid documentation.

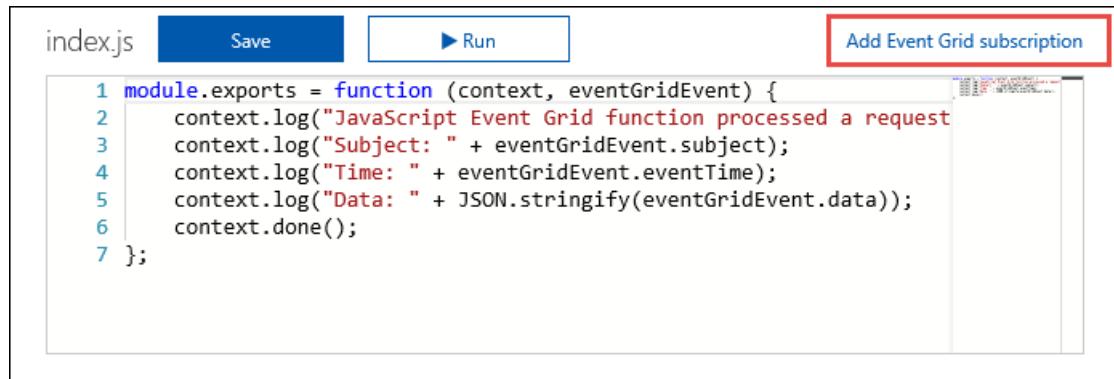
The `EventGridEvent` type defines only the top-level properties; the `Data` property is a `JObject`.

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 **Add Event Grid subscription**.



When you select this link, the portal opens the **Create Event Subscription** page with the endpoint URL prefilled.

Create Event Subscription

Event Grid

* Name

Topic Type
Storage Accounts

Subscription
Windows Azure MSDN - Visual Studio Ultimate

Resource group
Use existing
eg0122

Instance ⓘ

Subscribe to all event types

Subscriber Type
Web Hook

* Subscriber Endpoint

Prefix Filter
 Optional

Suffix Filter
 Optional

Filter Case Sensitive

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 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 runtime

```
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

```
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 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

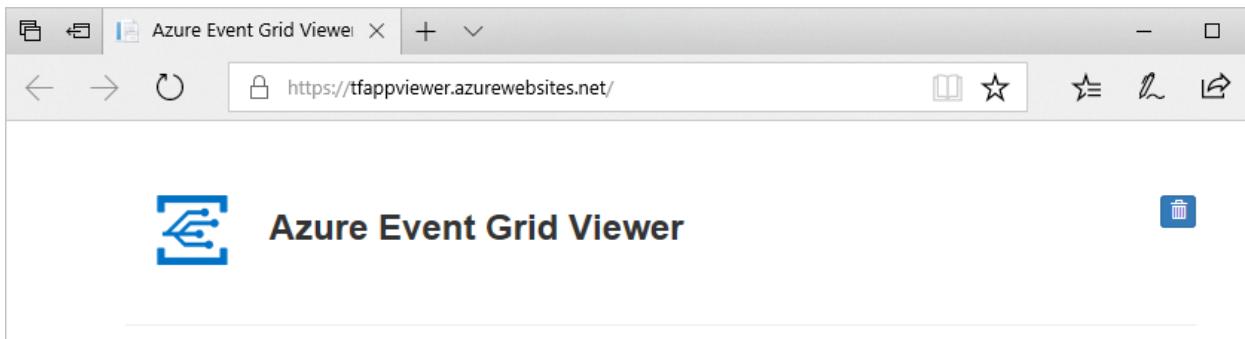
To simplify capturing event messages, you can deploy a [pre-built web app](#) that displays the event messages. The deployed solution includes an App Service plan, an App Service web app, and source code from GitHub.

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 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

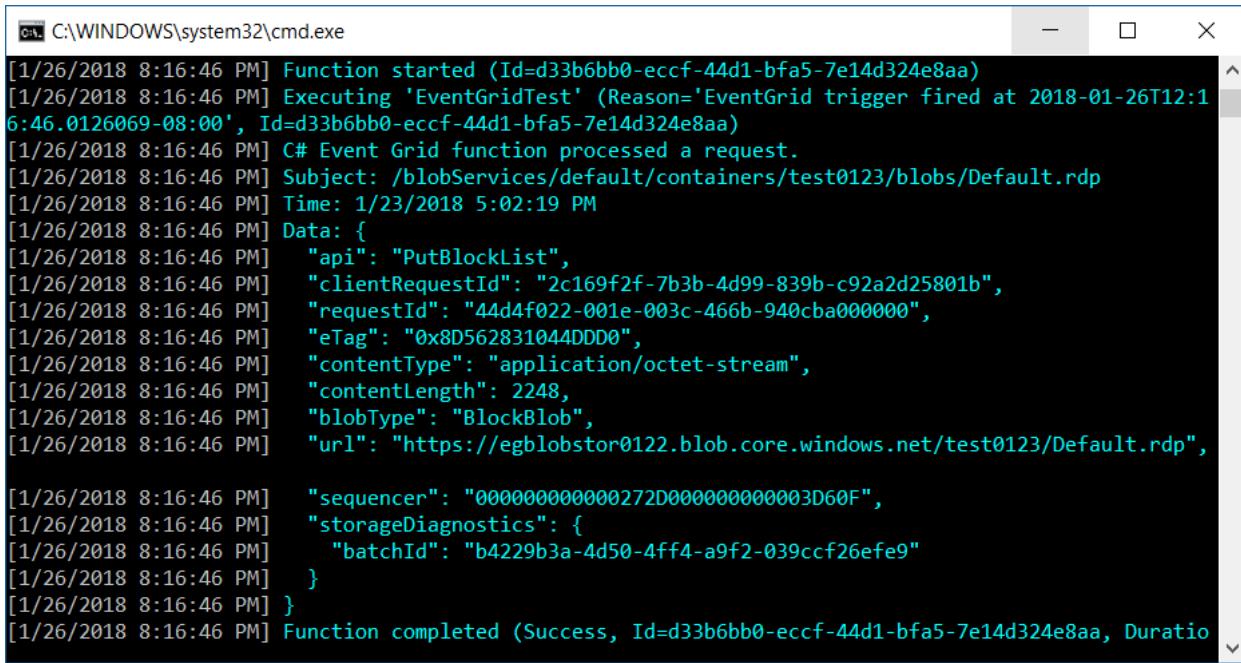
POST http://localhost:7071/admin/extensions/EventGridExtensionConfig?functionNa...

form-data	x-www-form-urlencoded	raw	binary	JSON (application/json) ▾
-----------	-----------------------	-----	--------	---------------------------

```

1 [
2   {
3     "topic": "/subscriptions/{subscriptionid}/resourceGroups/eg0122/providers/Microsoft_BlobStorage/accounts/egblobstor0122",
4     "subject": "/blobServices/default/containers/test0123/blobs/Default.rdp",
5     "eventType": "Microsoft.Storage.BlobCreated",
6     "eventTime": "2018-01-23T17:02:19.6069787Z",
7     "id": "44d4f022-001e-003c-466b-940cba0612e5",
8     "data": {
9       "api": "PutBlockList",
10      "clientRequestId": "2c169f2f-7b3b-4d99-839b-c92a2d25801b",
11      "requestId": "44d4f022-001e-003c-466b-940cba000000",
12      "eTag": "0x8D562831044DD0",
13      "contentType": "application/octet-stream",
14      "contentLength": 2248,
15      "blobType": "BlockBlob",
16      "url": "https://egblobstor0122.blob.core.windows.net/test0123/Default.rdp",
17      "sequencer": "000000000000272D000000000003D60F",
18      "storageDiagnostics": {
19        "batchId": "b4229b3a-4d50-4ff4-a9f2-039ccf26efe9"
20      }
21    },
22    "dataVersion": "",
23    "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 2018-01-26T12:16:46.0126069-08:00', Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa)
[1/26/2018 8:16:46 PM] C# Event Grid function processed a request.
[1/26/2018 8:16:46 PM] Subject: /blobServices/default/containers/test0123/blobs/Default.rdp
[1/26/2018 8:16:46 PM] Time: 1/23/2018 5:02:19 PM
[1/26/2018 8:16:46 PM] Data: {
[1/26/2018 8:16:46 PM]     "api": "PutBlockList",
[1/26/2018 8:16:46 PM]     "clientRequestId": "2c169f2f-7b3b-4d99-839b-c92a2d25801b",
[1/26/2018 8:16:46 PM]     "requestId": "44d4f022-001e-003c-466b-940cba000000",
[1/26/2018 8:16:46 PM]     "eTag": "0x8D562831044DD0",
[1/26/2018 8:16:46 PM]     "contentType": "application/octet-stream",
[1/26/2018 8:16:46 PM]     "contentLength": 2248,
[1/26/2018 8:16:46 PM]     "blobType": "BlockBlob",
[1/26/2018 8:16:46 PM]     "url": "https://egblobstor0122.blob.core.windows.net/test0123/Default.rdp",
[1/26/2018 8:16:46 PM]     "sequencer": "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, Duration: 00:00:00.000)
```

Local testing with ngrok

Another way to test an Event Grid trigger locally is to automate the HTTP connection between the Internet and your development computer. You can do that with a tool like [ngrok](#):

1. [Create an ngrok endpoint](#).
2. [Run the Event Grid trigger function](#).
3. [Create an Event Grid subscription](#) that sends events to the ngrok endpoint.
4. [Trigger an event](#).

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 an ngrok endpoint

Download `ngrok.exe` from [ngrok](#), and run with the following command:

```
ngrok http -host-header=localhost 7071
```

The `-host-header` parameter is needed because the functions runtime expects requests from localhost when it runs on localhost. 7071 is the default port number when the runtime runs locally.

The command creates output like the following:

```
Session Status          online
Version                2.2.8
Region                 United States (us)
Web Interface          http://127.0.0.1:4040
Forwarding             http://263db807.ngrok.io -> localhost:7071
Forwarding             https://263db807.ngrok.io -> localhost:7071

Connections            ttl     opn     rt1     rt5     p50     p90
                        0       0       0.00   0.00   0.00   0.00
```

You'll use the `https://[subdomain].ngrok.io` URL for your Event Grid subscription.

Run the Event Grid trigger function

The ngrok URL doesn't get special handling by Event Grid, so your function must be running locally when the

subscription is created. If it isn't, the validation response doesn't get sent and the subscription creation fails.

Create a subscription

Create an Event Grid subscription of the type you want to test, and give it your ngrok endpoint.

Use this endpoint pattern for Functions 2.x:

```
https://{{SUBDOMAIN}}.ngrok.io/runtime/webhooks/eventgrid?functionName={{FUNCTION_NAME}}
```

Use this endpoint pattern for Functions 1.x:

```
https://{{SUBDOMAIN}}.ngrok.io/admin/extensions/EventGridExtensionConfig?functionName={{FUNCTION_NAME}}
```

The `{FUNCTION_NAME}` parameter must be the name specified in the `FunctionName` attribute.

Here's an example using the Azure CLI:

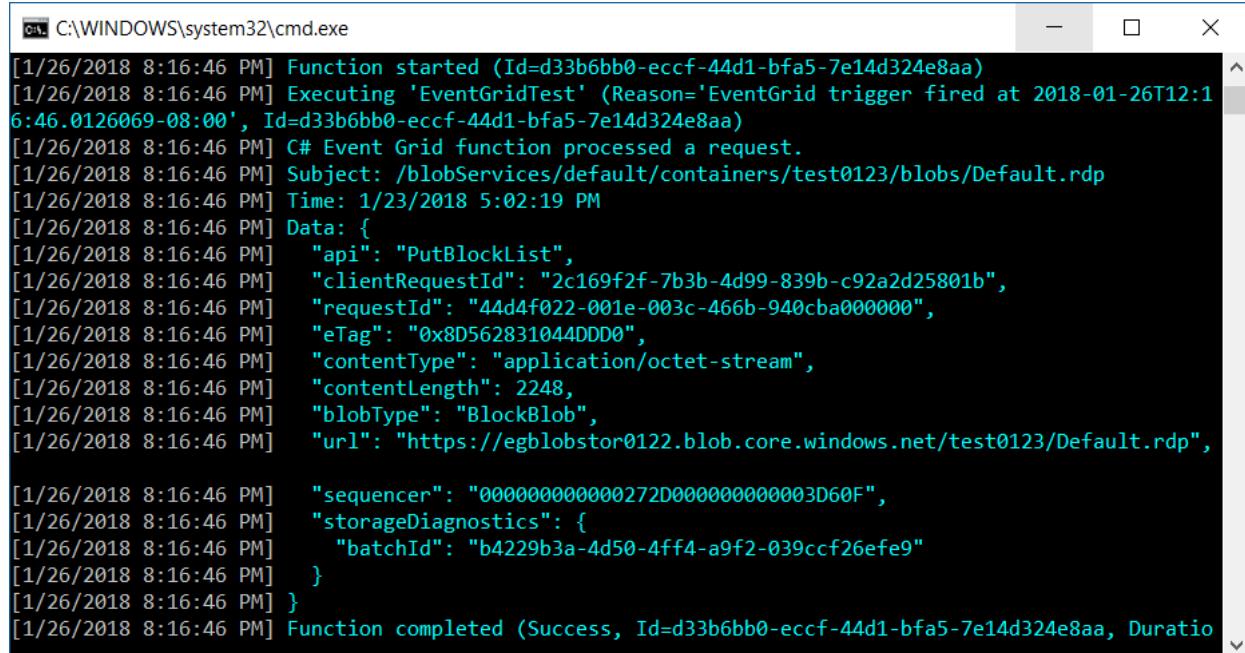
```
az eventgrid event-subscription create --resource-id /subscriptions/aeb4b7cb-b7cb-b7cb-b7cb-b7cbb6607f30/resourceGroups/eg0122/providers/Microsoft.Storage/storageAccounts/egblobstor0122 --name egblobsub0126 --endpoint https://263db807.ngrok.io/runtime/webhooks/eventgrid?functionName=EventGridTrigger
```

For information about how to create a subscription, see [Create a subscription](#) earlier in this article.

Trigger an event

Trigger an event that will generate HTTP traffic to your ngrok endpoint. For example, if you created a blob storage subscription, upload or delete a blob.

The Event Grid trigger function executes and shows logs similar to the following example:



```
C:\WINDOWS\system32\cmd.exe
[1/26/2018 8:16:46 PM] Function started (Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa)
[1/26/2018 8:16:46 PM] Executing 'EventGridTest' (Reason='EventGrid trigger fired at 2018-01-26T12:16:46.0126069-08:00', Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa)
[1/26/2018 8:16:46 PM] C# Event Grid function processed a request.
[1/26/2018 8:16:46 PM] Subject: /blobServices/default/containers/test0123/blobs/Default.rdp
[1/26/2018 8:16:46 PM] Time: 1/23/2018 5:02:19 PM
[1/26/2018 8:16:46 PM] Data: {
[1/26/2018 8:16:46 PM]   "api": "PutBlockList",
[1/26/2018 8:16:46 PM]   "clientRequestId": "2c169f2f-7b3b-4d99-839b-c92a2d25801b",
[1/26/2018 8:16:46 PM]   "requestId": "44d4f022-001e-003c-466b-940cba000000",
[1/26/2018 8:16:46 PM]   "eTag": "0x8D562831044DD0",
[1/26/2018 8:16:46 PM]   "contentType": "application/octet-stream",
[1/26/2018 8:16:46 PM]   "contentLength": 2248,
[1/26/2018 8:16:46 PM]   "blobType": "BlockBlob",
[1/26/2018 8:16:46 PM]   "url": "https://egblobstor0122.blob.core.windows.net/test0123/Default.rdp",
[1/26/2018 8:16:46 PM] 
[1/26/2018 8:16:46 PM]   "sequencer": "000000000000272D000000000003D60F",
[1/26/2018 8:16:46 PM]   "storageDiagnostics": {
[1/26/2018 8:16:46 PM]     "batchId": "b4229b3a-4d50-4ff4-a9f2-039ccf26efe9"
[1/26/2018 8:16:46 PM]   }
[1/26/2018 8:16:46 PM] }
[1/26/2018 8:16:46 PM] Function completed (Success, Id=d33b6bb0-eccf-44d1-bfa5-7e14d324e8aa, Duration: 00:00:00.000000)
```

Use an HTTP trigger as an Event Grid trigger

Event Grid events are received as HTTP requests, so you can handle events by using an HTTP trigger instead of an Event Grid trigger. One possible reason for doing that is to get more control over the endpoint URL that invokes the function. Another reason is when you need to receive events in the [CloudEvents schema](#). Currently, the Event Grid trigger doesn't support the CloudEvents schema. The examples in this section show solutions for both Event Grid schema and CloudEvents schema.

If you use an HTTP trigger, you have to write code for what the Event Grid trigger does automatically:

- Sends a validation response to a [subscription validation request](#).
- Invokes the function once per element of the event array contained in the request body.

For information about the URL to use for invoking the function locally or when it runs in Azure, see the [HTTP trigger binding reference documentation](#)

Event Grid schema

The following sample C# code for an HTTP trigger simulates Event Grid trigger behavior. Use this example for events delivered in the Event Grid schema.

```
[FunctionName("HttpTrigger")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]HttpRequestMessage req,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    var messages = await req.Content.ReadAsAsync<JArray>();

    // If the request is for subscription validation, send back the validation code.
    if (messages.Count > 0 && string.Equals((string)messages[0]["eventType"],
        "Microsoft.EventGrid.SubscriptionValidationEvent",
        System.StringComparison.OrdinalIgnoreCase))
    {
        log.LogInformation("Validate request received");
        return req.CreateResponse<object>(new
        {
            validationResponse = messages[0]["data"]["validationCode"]
        });
    }

    // The request is not for subscription validation, so it's for one or more events.
    foreach ( JObject message in messages)
    {
        // Handle one event.
        EventGridEvent eventGridEvent = message.ToObject<EventGridEvent>();
        log.LogInformation($"Subject: {eventGridEvent.Subject}");
        log.LogInformation($"Time: {eventGridEvent.EventTime}");
        log.LogInformation($"Event data: {eventGridEvent.Data.ToString()}");
    }

    return req.CreateResponse(HttpStatusCode.OK);
}
```

The following sample JavaScript code for an HTTP trigger simulates Event Grid trigger behavior. Use this example for events delivered in the Event Grid schema.

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

    var messages = req.body;
    // If the request is for subscription validation, send back the validation code.
    if (messages.length > 0 && messages[0].eventType == "Microsoft.EventGrid.SubscriptionValidationEvent") {
        context.log('Validate request received');
        var code = messages[0].data.validationCode;
        context.res = { status: 200, body: { "ValidationResponse": code } };
    }
    else {
        // The request is not for subscription validation, so it's for one or more events.
        // Event Grid schema delivers events in an array.
        for (var i = 0; i < messages.length; i++) {
            // Handle one event.
            var message = messages[i];
            context.log('Subject: ' + message.subject);
            context.log('Time: ' + message.eventTime);
            context.log('Data: ' + JSON.stringify(message.data));
        }
    }
    context.done();
};
```

Your event-handling code goes inside the loop through the `messages` array.

CloudEvents schema

The following sample C# code for an HTTP trigger simulates Event Grid trigger behavior. Use this example for events delivered in the CloudEvents schema.

```

[FunctionName("HttpTrigger")]
public static async Task<HttpResponseMessage> Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
Route = null)]HttpRequestMessage req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    var requestmessage = await req.Content.ReadAsStringAsync();
    var message = JToken.Parse(requestmessage);

    if (message.Type == JTokenType.Array)
    {
        // If the request is for subscription validation, send back the validation code.
        if (string.Equals((string)message[0]["eventType"],
        "Microsoft.EventGrid.SubscriptionValidationEvent",
        StringComparison.OrdinalIgnoreCase))
        {
            log.LogInformation("Validate request received");
            return req.CreateResponse<object>(new
            {
                validationResponse = message[0]["data"]["validationCode"]
            });
        }
    }
    else
    {
        // The request is not for subscription validation, so it's for an event.
        // CloudEvents schema delivers one event at a time.
        log.LogInformation($"Source: {message["source"]}");
        log.LogInformation($"Time: {message["eventTime"]}");
        log.LogInformation($"Event data: {message["data"].ToString()}");
    }

    return req.CreateResponse(HttpStatusCode.OK);
}

```

The following sample JavaScript code for an HTTP trigger simulates Event Grid trigger behavior. Use this example for events delivered in the CloudEvents schema.

```

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

    var message = req.body;
    // If the request is for subscription validation, send back the validation code.
    if (message.length > 0 && message[0].eventType == "Microsoft.EventGrid.SubscriptionValidationEvent") {
        context.log('Validate request received');
        var code = message[0].data.validationCode;
        context.res = { status: 200, body: { "ValidationResponse": code } };
    }
    else {
        // The request is not for subscription validation, so it's for an event.
        // CloudEvents schema delivers one event at a time.
        var event = JSON.parse(message);
        context.log('Source: ' + event.source);
        context.log('Time: ' + event.eventTime);
        context.log('Data: ' + JSON.stringify(event.data));
    }
    context.done();
};

```

Next steps

[Learn more about Azure functions triggers and bindings](#)

[Learn more about Event Grid](#)

Azure Event Hubs bindings for Azure Functions

7/1/2019 • 17 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.

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

For Azure Functions version 1.x, the Event Hubs bindings are provided in the [Microsoft.Azure.WebJobs.ServiceBus](#) NuGet package, version 2.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

For Functions 2.x, use the [Microsoft.Azure.WebJobs.Extensions.EventHubs](#) package, version 3.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 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Trigger

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.

Trigger - 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 its 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`.

When Functions scales, `N` instances is a number greater than the number of event hub partitions. This is done 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.

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Trigger - C# example

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}");
    }
}
```

Trigger - C# script example

The following example shows an event hub trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function logs the message body of the event hub trigger.

The following examples show Event Hubs binding data in the `function.json` file.

Version 2.x

```
{  
    "type": "eventHubTrigger",  
    "name": "myEventHubMessage",  
    "direction": "in",  
    "eventHubName": "MyEventHub",  
    "connection": "myEventHubReadConnectionAppSetting"  
}
```

Version 1.x

```
{  
    "type": "eventHubTrigger",  
    "name": "myEventHubMessage",  
    "direction": "in",  
    "path": "MyEventHub",  
    "connection": "myEventHubReadConnectionAppSetting"  
}
```

Here's the C# script code:

```
using System;  
  
public static void Run(string myEventHubMessage, TraceWriter log)  
{  
    log.Info($"C# function triggered to process a message: {myEventHubMessage}");  
}
```

To get access to [event metadata](#) in function code, bind to an [EventData](#) object (requires a using statement for `#r "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:

```
#r "Microsoft.Azure.EventHubs"  
  
using System.Text;  
using System;  
using Microsoft.ServiceBus.Messaging;  
using Microsoft.Azure.EventHubs;  
  
public static void Run(EventData myEventHubMessage,  
    DateTime enqueuedTimeUtc,  
    Int64 sequenceNumber,  
    string offset,  
    TraceWriter log)  
{  
    log.Info($"Event: {Encoding.UTF8.GetString(myEventHubMessage.Body)}");  
    log.Info($"EnqueuedTimeUtc={myEventHubMessage.SystemProperties.EnqueuedTimeUtc}");  
    log.Info($"SequenceNumber={myEventHubMessage.SystemProperties.SequenceNumber}");  
    log.Info($"Offset={myEventHubMessage.SystemProperties.Offset}");  
  
    // Metadata accessed by using binding expressions  
    log.Info($"EnqueuedTimeUtc={enqueuedTimeUtc}");  
    log.Info($"SequenceNumber={sequenceNumber}");  
    log.Info($"Offset={offset}");  
}
```

To receive events in a batch, make `string` or `EventData` an array:

```

public static void Run(string[] eventHubMessages, TraceWriter log)
{
    foreach (var message in eventHubMessages)
    {
        log.Info($"C# function triggered to process a message: {message}");
    }
}

```

Trigger - F# example

The following example shows an event hub trigger binding in a *function.json* file and an [F# function](#) that uses the binding. The function logs the message body of the event hub trigger.

The following examples show Event Hubs binding data in the *function.json* file.

Version 2.x

```
{
    "type": "eventHubTrigger",
    "name": "myEventHubMessage",
    "direction": "in",
    "eventHubName": "MyEventHub",
    "connection": "myEventHubReadConnectionAppSetting"
}
```

Version 1.x

```
{
    "type": "eventHubTrigger",
    "name": "myEventHubMessage",
    "direction": "in",
    "path": "MyEventHub",
    "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the F# code:

```

let Run(myEventHubMessage: string, log: TraceWriter) =
    log.Log(sprintf "F# eventhub trigger function processed work item: %s" myEventHubMessage)

```

Trigger - JavaScript example

The following example shows an event hub trigger binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function reads [event metadata](#) and logs the message.

The following examples show Event Hubs binding data in the *function.json* file.

Version 2.x

```
{
    "type": "eventHubTrigger",
    "name": "myEventHubMessage",
    "direction": "in",
    "eventHubName": "MyEventHub",
    "connection": "myEventHubReadConnectionAppSetting"
}
```

Version 1.x

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the JavaScript code:

```
module.exports = function (context, myEventHubMessage) {
    context.log('Function triggered to process a message: ', myEventHubMessage);
    context.log('EnqueuedTimeUtc =', context.bindingData.enqueuedTimeUtc);
    context.log('SequenceNumber =', context.bindingData.sequenceNumber);
    context.log('Offset =', context.bindingData.offset);

    context.done();
};
```

To receive events in a batch, set `cardinality` to `many` in the `function.json` file, as shown in the following examples.

Version 2.x

```
{
  "type": "eventHubTrigger",
  "name": "eventHubMessages",
  "direction": "in",
  "eventHubName": "MyEventHub",
  "cardinality": "many",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Version 1.x

```
{
  "type": "eventHubTrigger",
  "name": "eventHubMessages",
  "direction": "in",
  "path": "MyEventHub",
  "cardinality": "many",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the JavaScript code:

```
module.exports = function (context, eventHubMessages) {
    context.log(`JavaScript eventhub trigger function called for message array ${eventHubMessages}`);

    eventHubMessages.forEach((message, index) => {
        context.log(`Processed message ${message}`);
        context.log(`EnqueuedTimeUtc = ${context.bindingData.enqueuedTimeUtcArray[index]}`);
        context.log(`SequenceNumber = ${context.bindingData.sequenceNumberArray[index]}`);
        context.log(`Offset = ${context.bindingData.offsetArray[index]}`);
    });

    context.done();
};
```

Trigger - Python example

The following example shows an event hub trigger binding in a *function.json* file and a [Python function](#) that uses the binding. The function reads [event metadata](#) and logs the message.

The following examples show Event Hubs binding data in the *function.json* file.

```
{  
  "type": "eventHubTrigger",  
  "name": "event",  
  "direction": "in",  
  "eventHubName": "MyEventHub",  
  "connection": "myEventHubReadConnectionAppSetting"  
}
```

Here's the Python code:

```
import logging  
import azure.functions as func  
  
def main(event: func.EventHubEvent):  
    logging.info('Function triggered to process a message: ', event.get_body())  
    logging.info(' EnqueuedTimeUtc =', event.enqueued_time)  
    logging.info(' SequenceNumber =', event.sequence_number)  
    logging.info(' Offset =', event.offset)
```

Trigger - Java example

The following example shows an Event Hub trigger binding in a *function.json* file and an [Java function](#) that uses the binding. The function logs the message body of the Event Hub trigger.

```
{  
  "type": "eventHubTrigger",  
  "name": "msg",  
  "direction": "in",  
  "eventHubName": "myeventhubname",  
  "connection": "myEventHubReadConnectionAppSetting"  
}
```

```
@FunctionName("ehprocessor")  
public void eventHubProcessor(  
    @EventHubTrigger(name = "msg",  
        eventHubName = "myeventhubname",  
        connection = "myconnvarname") String message,  
    final ExecutionContext context )  
{  
    context.getLogger().info(message);  
}
```

In the [Java functions runtime library](#), use the `EventHubTrigger` annotation on parameters whose value would come from Event Hub. Parameters with these annotations cause the function to run when an event arrives. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

Trigger - attributes

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](#).

Trigger - 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
type	n/a	Must be set to <code>eventHubTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	n/a	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 represents the event item in function code.
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 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.
consumerGroup	ConsumerGroup	An optional property that sets the <code>consumer group#event-consumers</code> used to subscribe to events in the hub. If omitted, the <code>\$Default</code> consumer group is used.
cardinality	n/a	For Javascript. Set to <code>many</code> in order to enable batching. If omitted or set to <code>one</code> , single message passed to function.

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 namespace#create-an-event-hubs-namespace , not the event hub itself. This connection string must have at least read permissions to activate the trigger.
path	EventHubName	The name of the event hub. Can be referenced via app settings %eventHubName%

When you're developing locally, app settings go into the [local.settings.json file](#).

Trigger - event metadata

The Event Hubs 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 are properties of the [EventData](#) class.

PROPERTY	TYPE	DESCRIPTION
<code>PartitionContext</code>	<code>PartitionContext</code>	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.

Trigger - host.json properties

The `host.json` file contains settings that control Event Hubs trigger behavior.

```
{
    "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 PrefetchCount that will be used by the underlying EventProcessorHost.
batchCheckpointFrequency	1	The number of event batches to process before creating an EventHub cursor checkpoint.

Output

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.

Ensure the required package references are in place: Functions 1.x or Functions 2.x

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Output - C# example

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 sample 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));
    }
}

```

Output - C# script example

The following example shows an event hub trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file. The first example is for Functions 2.x, and the second one is for Functions 1.x.

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

Here's C# script code that creates one message:

```

using System;
using Microsoft.Extensions.Logging;

public static void Run(TimerInfo myTimer, out string outputEventHubMessage, ILogger log)
{
    String msg = $"TimerTriggerCSharp1 executed at: {DateTime.Now}";
    log.LogInformation(msg);
    outputEventHubMessage = msg;
}

```

Here's C# script code that creates multiple messages:

```

public static void Run(TimerInfo myTimer, ICollector<string> outputEventHubMessage, ILogger log)
{
    string message = $"Message created at: {DateTime.Now}";
    log.LogInformation(message);
    outputEventHubMessage.Add("1 " + message);
    outputEventHubMessage.Add("2 " + message);
}

```

Output - F# example

The following example shows an event hub trigger binding in a *function.json* file and an [F# function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file. The first example is for Functions 2.x, and the second one is for Functions 1.x.

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

Here's the F# code:

```

let Run(myTimer: TimerInfo, outputEventHubMessage: byref<string>, log: ILogger) =
    let msg = sprintf "TimerTriggerFSharp1 executed at: %s" DateTime.Now.ToString()
    log.LogInformation(msg);
    outputEventHubMessage <- msg;

```

Output - JavaScript example

The following example shows an event hub trigger binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file. The first example is for Functions 2.x, and the second one is for Functions 1.x.

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

Here's JavaScript code that sends a single message:

```
module.exports = function (context, myTimer) {
    var timeStamp = new Date().toISOString();
    context.log('Message created at: ', timeStamp);
    context.bindings.outputEventHubMessage = "Message created at: " + timeStamp;
    context.done();
};
```

Here's JavaScript code that sends multiple messages:

```
module.exports = function(context) {
    var timeStamp = new Date().toISOString();
    var message = 'Message created at: ' + timeStamp;

    context.bindings.outputEventHubMessage = [];

    context.bindings.outputEventHubMessage.push("1 " + message);
    context.bindings.outputEventHubMessage.push("2 " + message);
    context.done();
};
```

Output - Python example

The following example shows an event hub trigger binding in a *function.json* file and a [Python function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file.

```
{
    "type": "eventHub",
    "name": "$return",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

Here's Python code that sends a single message:

```
import datetime
import logging
import azure.functions as func

def main(timer: func.TimerRequest) -> str:
    timestamp = datetime.datetime.utcnow()
    logging.info('Message created at: %s', timestamp)
    return 'Message created at: {}'.format(timestamp)
```

Output - Java example

The following example shows a Java function that writes a message containing the current time to an Event

Hub.

```
@FunctionName("sendTime")
@EventHubOutput(name = "event", eventHubName = "samples-workitems", connection = "AzureEventHubConnection")
public String sendTime()
{
    @TimerTrigger(name = "sendTimeTrigger", schedule = "0 */* * * *") String timerInfo) {
        return LocalDateTime.now().toString();
}
```

In the [Java functions runtime library](#), use the `@EventHubOutput` annotation on parameters whose value would be published to Event Hub. The parameter should be of type `OutputBinding<T>`, where T is a POJO or any native Java type.

Output - attributes

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](#).

Output - 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.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
eventHubName	EventHubName	Functions 2.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.
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.

Output - usage

In C# and C# script, 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`.

In JavaScript, access the output event by using `context.bindings.<name>`. `<name>` is the value specified in the `name` property of *function.json*.

Exceptions and return codes

BINDING	REFERENCE
Event Hub	Operations Guide

host.json settings

This section describes the global configuration settings available for this binding in version 2.x. The example host.json file below contains only the version 2.x settings for this binding. For more information about global configuration settings in version 2.x, see [host.json reference for Azure Functions version 2.x](#).

NOTE

For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

```
{
    "version": "2.0",
    "extensions": {
        "eventHubs": {
            "batchCheckpointFrequency": 5,
            "eventProcessorOptions": {
                "maxBatchSize": 256,
                "prefetchCount": 512
            }
        }
    }
}
```

PROPERTY	DEFAULT	DESCRIPTION
maxBatchSize	64	The maximum event count received per receive loop.
prefetchCount	n/a	The default PrefetchCount that will be used by the underlying EventProcessorHost.
batchCheckpointFrequency	1	The number of event batches to process before creating an EventHub cursor checkpoint.

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure IoT Hub bindings for Azure Functions

7/1/2019 • 17 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](#).

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

For Azure Functions version 1.x, the IoT Hub bindings are provided in the [Microsoft.Azure.WebJobs.ServiceBus](#) NuGet package, version 2.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

For Functions 2.x, use the [Microsoft.Azure.WebJobs.Extensions.EventHubs](#) package, version 3.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 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

IMPORTANT

While the following code samples use the Event Hub API, the given syntax is applicable for IoT Hub functions.

Trigger

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.

Trigger - 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 its 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`.

When Functions scales, `N` instances is a number greater than the number of event hub partitions. This is done 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.

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)

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

Trigger - C# example

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}");
    }
}
```

Trigger - C# script example

The following example shows an event hub trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function logs the message body of the event hub trigger.

The following examples show Event Hubs binding data in the `function.json` file.

Version 2.x

```
{  
  "type": "eventHubTrigger",  
  "name": "myEventHubMessage",  
  "direction": "in",  
  "eventHubName": "MyEventHub",  
  "connection": "myEventHubReadConnectionAppSetting"  
}
```

Version 1.x

```
{  
  "type": "eventHubTrigger",  
  "name": "myEventHubMessage",  
  "direction": "in",  
  "path": "MyEventHub",  
  "connection": "myEventHubReadConnectionAppSetting"  
}
```

Here's the C# script code:

```
using System;  
  
public static void Run(string myEventHubMessage, TraceWriter log)  
{  
    log.Info($"C# function triggered to process a message: {myEventHubMessage}");  
}
```

To get access to [event metadata](#) in function code, bind to an `EventData` object (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:

```
#r "Microsoft.Azure.EventHubs"  
  
using System.Text;  
using System;  
using Microsoft.ServiceBus.Messaging;  
using Microsoft.Azure.EventHubs;  
  
public static void Run(EventData myEventHubMessage,  
                      DateTime enqueuedTimeUtc,  
                      Int64 sequenceNumber,  
                      string offset,  
                      TraceWriter log)  
{  
    log.Info($"Event: {Encoding.UTF8.GetString(myEventHubMessage.Body)}");  
    log.Info($"EnqueuedTimeUtc={myEventHubMessage.SystemProperties.EnqueuedTimeUtc}");  
    log.Info($"SequenceNumber={myEventHubMessage.SystemProperties.SequenceNumber}");  
    log.Info($"Offset={myEventHubMessage.SystemProperties.Offset}");  
  
    // Metadata accessed by using binding expressions  
    log.Info($"EnqueuedTimeUtc={enqueuedTimeUtc}");  
    log.Info($"SequenceNumber={sequenceNumber}");  
    log.Info($"Offset={offset}");  
}
```

To receive events in a batch, make `string` or `EventData` an array:

```
public static void Run(string[] eventHubMessages, TraceWriter log)
{
    foreach (var message in eventHubMessages)
    {
        log.Info($"C# function triggered to process a message: {message}");
    }
}
```

Trigger - F# example

The following example shows an event hub trigger binding in a `function.json` file and an [F# function](#) that uses the binding. The function logs the message body of the event hub trigger.

The following examples show Event Hubs binding data in the `function.json` file.

Version 2.x

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "eventHubName": "MyEventHub",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Version 1.x

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the F# code:

```
let Run(myEventHubMessage: string, log: TraceWriter) =
    log.Log(sprintf "F# eventhub trigger function processed work item: %s" myEventHubMessage)
```

Trigger - JavaScript example

The following example shows an event hub trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function reads `event metadata` and logs the message.

The following examples show Event Hubs binding data in the `function.json` file.

Version 2.x

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "eventHubName": "MyEventHub",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Version 1.x

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the JavaScript code:

```
module.exports = function (context, myEventHubMessage) {
    context.log('Function triggered to process a message: ', myEventHubMessage);
    context.log('EnqueuedTimeUtc =', context.bindingData.enqueuedTimeUtc);
    context.log('SequenceNumber =', context.bindingData.sequenceNumber);
    context.log('Offset =', context.bindingData.offset);

    context.done();
};
```

To receive events in a batch, set `cardinality` to `many` in the `function.json` file, as shown in the following examples.

Version 2.x

```
{
  "type": "eventHubTrigger",
  "name": "eventHubMessages",
  "direction": "in",
  "eventHubName": "MyEventHub",
  "cardinality": "many",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Version 1.x

```
{
  "type": "eventHubTrigger",
  "name": "eventHubMessages",
  "direction": "in",
  "path": "MyEventHub",
  "cardinality": "many",
  "connection": "myEventHubReadConnectionAppSetting"
}
```

Here's the JavaScript code:

```
module.exports = function (context, eventHubMessages) {
    context.log(`JavaScript eventhub trigger function called for message array ${eventHubMessages}`);

    eventHubMessages.forEach((message, index) => {
        context.log(`Processed message ${message}`);
        context.log(`EnqueuedTimeUtc = ${context.bindingData.enqueuedTimeUtcArray[index]}`);
        context.log(`SequenceNumber = ${context.bindingData.sequenceNumberArray[index]}`);
        context.log(`Offset = ${context.bindingData.offsetArray[index]}`);
    });

    context.done();
};
```

Trigger - Python example

The following example shows an event hub trigger binding in a `function.json` file and a [Python function](#) that uses

the binding. The function reads [event metadata](#) and logs the message.

The following examples show Event Hubs binding data in the `function.json` file.

```
{  
  "type": "eventHubTrigger",  
  "name": "event",  
  "direction": "in",  
  "eventHubName": "MyEventHub",  
  "connection": "myEventHubReadConnectionAppSetting"  
}
```

Here's the Python code:

```
import logging  
import azure.functions as func  
  
def main(event: func.EventHubEvent):  
    logging.info('Function triggered to process a message: ', event.get_body())  
    logging.info(' EnqueuedTimeUtc =', event.enqueued_time)  
    logging.info(' SequenceNumber =', event.sequence_number)  
    logging.info(' Offset =', event.offset)
```

Trigger - Java example

The following example shows an Event Hub trigger binding in a `function.json` file and an [Java function](#) that uses the binding. The function logs the message body of the Event Hub trigger.

```
{  
  "type": "eventHubTrigger",  
  "name": "msg",  
  "direction": "in",  
  "eventHubName": "myeventhubname",  
  "connection": "myEventHubReadConnectionAppSetting"  
}
```

```
@FunctionName("ehprocessor")  
public void eventHubProcessor(  
    @EventHubTrigger(name = "msg",  
        eventHubName = "myeventhubname",  
        connection = "myconnvarname") String message,  
    final ExecutionContext context )  
{  
    context.getLogger().info(message);  
}
```

In the [Java functions runtime library](#), use the `EventHubTrigger` annotation on parameters whose value would come from Event Hub. Parameters with these annotations cause the function to run when an event arrives. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

Trigger - attributes

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](#).

Trigger - 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
type	n/a	Must be set to <code>eventHubTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	n/a	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 represents the event item in function code.
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 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.
consumerGroup	ConsumerGroup	An optional property that sets the <code>consumer group#event-consumers</code> used to subscribe to events in the hub. If omitted, the <code>\$Default</code> consumer group is used.
cardinality	n/a	For Javascript. Set to <code>many</code> in order to enable batching. If omitted or set to <code>one</code> , single message passed to function.

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 namespace#create-an-event-hubs-namespace , not the event hub itself. This connection string must have at least read permissions to activate the trigger.
path	EventHubName	The name of the event hub. Can be referenced via app settings %eventHubName%

When you're developing locally, app settings go into the [local.settings.json file](#).

Trigger - event metadata

The Event Hubs 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 are properties of the [EventData](#) class.

PROPERTY	TYPE	DESCRIPTION
<code>PartitionContext</code>	<code>PartitionContext</code>	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.

Trigger - host.json properties

The [host.json](#) file contains settings that control Event Hubs trigger behavior.

```
{
    "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 PrefetchCount that will be used by the underlying EventProcessorHost.
batchCheckpointFrequency	1	The number of event batches to process before creating an EventHub cursor checkpoint.

Output

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.

Ensure the required package references are in place: Functions 1.x or Functions 2.x

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Output - C# example

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 sample 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));
    }
}
```

Output - C# script example

The following example shows an event hub trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file. The first example is for Functions 2.x, and the second one is for Functions 1.x.

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

Here's C# script code that creates one message:

```
using System;
using Microsoft.Extensions.Logging;

public static void Run(TimerInfo myTimer, out string outputEventHubMessage, ILogger log)
{
    String msg = $"TimerTriggerCSharp1 executed at: {DateTime.Now}";
    log.LogInformation(msg);
    outputEventHubMessage = msg;
}
```

Here's C# script code that creates multiple messages:

```

public static void Run(TimerInfo myTimer, ICollector<string> outputEventHubMessage, ILogger log)
{
    string message = $"Message created at: {DateTime.Now}";
    log.LogInformation(message);
    outputEventHubMessage.Add("1 " + message);
    outputEventHubMessage.Add("2 " + message);
}

```

Output - F# example

The following example shows an event hub trigger binding in a *function.json* file and an [F# function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file. The first example is for Functions 2.x, and the second one is for Functions 1.x.

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

Here's the F# code:

```

let Run(myTimer: TimerInfo, outputEventHubMessage: byref<string>, log: ILogger) =
    let msg = sprintf "TimerTriggerFSharp1 executed at: %s" DateTime.Now.ToString()
    log.LogInformation(msg);
    outputEventHubMessage <- msg;

```

Output - JavaScript example

The following example shows an event hub trigger binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file. The first example is for Functions 2.x, and the second one is for Functions 1.x.

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "eventHubName": "myeventhub",
    "connection": "MyEventHubSendAppSetting",
    "direction": "out"
}
```

```
{
  "type": "eventHub",
  "name": "outputEventHubMessage",
  "path": "myeventhub",
  "connection": "MyEventHubSendAppSetting",
  "direction": "out"
}
```

Here's JavaScript code that sends a single message:

```
module.exports = function (context, myTimer) {
  var timeStamp = new Date().toISOString();
  context.log('Message created at: ', timeStamp);
  context.bindings.outputEventHubMessage = "Message created at: " + timeStamp;
  context.done();
};
```

Here's JavaScript code that sends multiple messages:

```
module.exports = function(context) {
  var timeStamp = new Date().toISOString();
  var message = 'Message created at: ' + timeStamp;

  context.bindings.outputEventHubMessage = [];

  context.bindings.outputEventHubMessage.push("1 " + message);
  context.bindings.outputEventHubMessage.push("2 " + message);
  context.done();
};
```

Output - Python example

The following example shows an event hub trigger binding in a *function.json* file and a [Python function](#) that uses the binding. The function writes a message to an event hub.

The following examples show Event Hubs binding data in the *function.json* file.

```
{
  "type": "eventHub",
  "name": "$return",
  "eventHubName": "myeventhub",
  "connection": "MyEventHubSendAppSetting",
  "direction": "out"
}
```

Here's Python code that sends a single message:

```
import datetime
import logging
import azure.functions as func

def main(timer: func.TimerRequest) -> str:
    timestamp = datetime.datetime.utcnow()
    logging.info('Message created at: %s', timestamp)
    return 'Message created at: {}'.format(timestamp)
```

Output - Java example

The following example shows a Java function that writes a message containing the current time to an Event Hub.

```

@FunctionName("sendTime")
@EventHubOutput(name = "event", eventHubName = "samples-workitems", connection = "AzureEventHubConnection")
public String sendTime(
    @TimerTrigger(name = "sendTimeTrigger", schedule = "0 * * * *") String timerInfo) {
    return LocalDateTime.now().toString();
}

```

In the [Java functions runtime library](#), use the `@EventHubOutput` annotation on parameters whose value would be published to Event Hub. The parameter should be of type `OutputBinding<T>`, where T is a POJO or any native Java type.

Output - attributes

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](#).

Output - 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 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.

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](#).

Output - usage

In C# and C# script, 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`.

In JavaScript, access the output event by using `context.bindings.<name>`. `<name>` is the value specified in the `name` property of *function.json*.

Exceptions and return codes

BINDING	REFERENCE
Event Hub	Operations Guide

host.json settings

This section describes the global configuration settings available for this binding in version 2.x. The example host.json file below contains only the version 2.x settings for this binding. For more information about global configuration settings in version 2.x, see [host.json reference for Azure Functions version 2.x](#).

NOTE

For a reference of host.json in Functions 1.x, see [host.json reference for Azure Functions 1.x](#).

```
{
  "version": "2.0",
  "extensions": {
    "eventHubs": {
      "batchCheckpointFrequency": 5,
      "eventProcessorOptions": {
        "maxBatchSize": 256,
        "prefetchCount": 512
      }
    }
  }
}
```

PROPERTY	DEFAULT	DESCRIPTION
maxBatchSize	64	The maximum event count received per receive loop.
prefetchCount	n/a	The default PrefetchCount that will be used by the underlying EventProcessorHost.
batchCheckpointFrequency	1	The number of event batches to process before creating an EventHub cursor checkpoint.

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Functions HTTP triggers and bindings

7/1/2019 • 21 minutes to read • [Edit Online](#)

This article explains how to work with HTTP triggers and output bindings in Azure Functions.

An HTTP trigger can be customized to respond to [webhooks](#).

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

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](#).

The code in this article defaults to Functions 2.x syntax which uses .NET Core. For information on the 1.x syntax, see the [1.x functions templates](#).

Packages - Functions 1.x

The HTTP bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.Http](#) NuGet package, version 1.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 2.x

The HTTP bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.Http](#) 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.

Trigger

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.

By default, an HTTP trigger returns HTTP 200 OK with an empty body in Functions 1.x, or HTTP 204 No Content with an empty body in Functions 2.x. To modify the response, configure an [HTTP output binding](#).

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

Trigger - C# example

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` 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");
}
```

Trigger - C# script example

The following example shows a trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function looks for a `name` parameter either in the query string or the body of the HTTP request.

Here's the `function.json` file:

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

The [configuration](#) section explains these properties.

Here's C# script code that binds to `HttpRequest`:

```

#r "Newtonsoft.Json"

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

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

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

```

You can bind to a custom object instead of `HttpRequest`. This object is created from the body of the request and parsed as JSON. Similarly, a type can be passed to the HTTP response output binding and returned as the response body, along with a 200 status code.

```

using System.Net;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

public static string Run(Person person, ILogger log)
{
    return person.Name != null
        ? (ActionResult)new OkObjectResult($"Hello, {person.Name}")
        : new BadRequestObjectResult("Please pass an instance of Person.");
}

public class Person {
    public string Name {get; set;}
}

```

Trigger - F# example

The following example shows a trigger binding in a `function.json` file and an [F# function](#) that uses the binding. The function looks for a `name` parameter either in the query string or the body of the HTTP request.

Here's the `function.json` file:

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

The [configuration](#) section explains these properties.

Here's the F# code:

```
open System.Net
open System.Net.Http
open FSharp.Interop.Dynamic

let Run(req: HttpRequestMessage) =
    async {
        let q =
            req.GetQueryNameValuePairs()
            |> Seq.tryFind (fun kv -> kv.Key = "name")
        match q with
        | Some kv ->
            return req.CreateResponse(HttpStatusCode.OK, "Hello " + kv.Value)
        | None ->
            let! data = Async.AwaitTask(req.Content.ReadAsAsync<obj>())
            try
                return req.CreateResponse(HttpStatusCode.OK, "Hello " + data?name)
            with e ->
                return req.CreateErrorResponse(HttpStatusCode.BadRequest, "Please pass a name on the
query string or in the request body")
    } |> Async.StartAsTask
```

You need a `project.json` file that uses NuGet to reference the `FSharp.Interop.Dynamic` and `Dynamitey` assemblies, as shown in the following example:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

Trigger - JavaScript example

The following example shows a trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function looks for a `name` parameter either in the query string or the body of the HTTP request.

Here's the `function.json` file:

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

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function(context, req) {
  context.log('Node.js HTTP trigger function processed a request. RequestUri=%s', req.originalUrl);

  if (req.query.name || (req.body && req.body.name)) {
    context.res = {
      // status 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"
    };
  }
  context.done();
};
```

Trigger - Python example

The following example shows a trigger binding in a *function.json* file and a [Python function](#) that uses the binding. The function looks for a `name` parameter either in the query string or the body of the HTTP request.

Here's the *function.json* file:

```
{
  "scriptFile": "__init__.py",
  "disabled": false,
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```

The [configuration](#) section explains these properties.

Here's the Python code:

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

Trigger - Java examples

- [Read parameter from the query string](#)
- [Read body from a POST request](#)
- [Read parameter from a route](#)
- [Read POJO body from a POST request](#)

The following examples show the HTTP trigger binding in a *function.json* file and the respective [Java functions](#) that use the binding.

Here's the *function.json* file:

```
{
  "disabled": false,
  "bindings": [
    {
      "authLevel": "anonymous",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```

Read parameter from the query string (Java)

This example reads a parameter, named `id`, from the query string, and uses it to build a JSON document returned to the client, with content type `application/json`.

```

@FunctionName("TriggerStringGet")
public HttpResponseMessage run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context) {

    // Item list
    context.getLogger().info("GET parameters are: " + request.getQueryParameters());

    // Get named parameter
    String id = request.getQueryParameters().getOrDefault("id", "");

    // Convert and display
    if (id.isEmpty()) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Document not found.")
            .build();
    }
    else {
        // return JSON from to the client
        // Generate document
        final String name = "fake_name";
        final String jsonDocument = "{\"id\":\"" + id + "\", \""
            + "\"description\": \"\" + name + "\"}";
        return request.createResponseBuilder(HttpStatus.OK)
            .header("Content-Type", "application/json")
            .body(jsonDocument)
            .build();
    }
}

```

Read body from a POST request (Java)

This example reads the body of a POST request, as a `String`, and uses it to build a JSON document returned to the client, with content type `application/json`.

```

@FunctionName("TriggerStringPost")
public HttpResponseMessage run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context) {

    // Item list
    context.getLogger().info("Request body is: " + request.getBody().orElse(""));

    // Check request body
    if (!request.getBody().isPresent()) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Document not found.")
            .build();
    }
    else {
        // return JSON from to the client
        // Generate document
        final String body = request.getBody().get();
        final String jsonDocument = "{\"id\":\"123456\", " +
            "\"description\": \"\" + body + "\"}";
        return request.createResponseBuilder(HttpStatus.OK)
            .header("Content-Type", "application/json")
            .body(jsonDocument)
            .build();
    }
}

```

Read parameter from a route (Java)

This example reads a mandatory parameter, named `id`, and an optional parameter `name` from the route path, and uses them to build a JSON document returned to the client, with content type `application/json`. T

```

@FunctionName("TriggerStringRoute")
public HttpResponseMessage run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.ANONYMOUS,
        route = "trigger/{id}/{name=EMPTY}") // name is optional and defaults to EMPTY
    HttpRequestMessage<Optional<String>> request,
    @BindingName("id") String id,
    @BindingName("name") String name,
    final ExecutionContext context) {

    // Item list
    context.getLogger().info("Route parameters are: " + id);

    // Convert and display
    if (id == null) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Document not found.")
            .build();
    }
    else {
        // return JSON from to the client
        // Generate document
        final String jsonDocument = "{\"id\":\"" + id + "\", " +
            "\"description\": \"\" + name + "\"}";
        return request.createResponseBuilder(HttpStatus.OK)
            .header("Content-Type", "application/json")
            .body(jsonDocument)
            .build();
    }
}

```

Read POJO body from a POST request (Java)

Here is the code for the `ToDoItem` class, referenced in this example:

```

public class ToDoItem {

    private String id;
    private String description;

    public ToDoItem(String id, String description) {
        this.id = id;
        this.description = description;
    }

    public String getId() {
        return id;
    }

    public String getDescription() {
        return description;
    }

    @Override
    public String toString() {
        return "ToDoItem{id=" + id + ",description=" + description + "}";
    }
}

```

This example reads the body of a POST request. The request body gets automatically de-serialized into a `ToDoItem` object, and is returned to the client, with content type `application/json`. The `ToDoItem` parameter is serialized by the Functions runtime as it is assigned to the `body` property of the

`HttpMessageResponse.Builder` class.

```
@FunctionName("TriggerPojoPost")
public HttpResponseMessage run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<ToDoItem>> request,
    final ExecutionContext context) {

    // Item list
    context.getLogger().info("Request body is: " + request.getBody().orElse(null));

    // Check request body
    if (!request.getBody().isPresent()) {
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
            .body("Document not found.")
            .build();
    }
    else {
        // return JSON from to the client
        // Generate document
        final ToDoItem body = request.getBody().get();
        return request.createResponseBuilder(HttpStatus.OK)
            .header("Content-Type", "application/json")
            .body(body)
            .build();
    }
}
```

Trigger - attributes

In C# class libraries, use the `HttpTrigger` attribute.

You can set the authorization level and allowable HTTP methods in attribute constructor parameters, and there are properties for webhook type and route template. For more information about these settings, see [Trigger - configuration](#). Here's an `HttpTrigger` attribute in a method signature:

```
[FunctionName("HttpTriggerCSharp")]
public static Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequest req
{
    ...
}
```

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 `HttpTrigger` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
type	n/a	Required - must be set to <code>httpTrigger</code> .
direction	n/a	Required - must be set to <code>in</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name	n/a	Required - the variable name used in function code for the request or request body.
authLevel	AuthLevel	<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>
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 <code><functionname></code> . For more information, see customize the http endpoint .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
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.

Trigger - usage

For C# and F# functions, you can declare the type of your trigger input to be 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.

For JavaScript functions, the Functions runtime provides the request body instead of the request object. For more information, see the [JavaScript trigger example](#).

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://<yourapp>.azurewebsites.net/api/<funcname>
```

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://<yourapp>.azurewebsites.net/api/products/electronics/357
```

This allows the function code to support two parameters in the address, *category* and *id*. You can use any [Web API Route Constraint](#) with your parameters. The following C# function code makes use of both parameters.

```
public static Task<IActionResult> Run(HttpContext req, string category, int? id, ILogger log)
{
    if (id == null)
    {
        return (ActionResult)new OkObjectResult($"All {category} items were requested.");
    }
    else
    {
        return (ActionResult)new OkObjectResult($"{category} item with id = {id} has been requested.");
    }

    // -----
    log.LogInformation($"C# HTTP trigger function processed a request. RequestUri={req.RequestUri}");
}
```

Here is Nodejs function code that uses the same route parameters.

```

module.exports = function (context, req) {

    var category = context.bindingData.category;
    var id = context.bindingData.id;

    if (!id) {
        context.res = {
            // status defaults to 200 */
            body: "All " + category + " items were requested."
        };
    }
    else {
        context.res = {
            // status defaults to 200 */
            body: category + " item with id = " + id + " was requested."
        };
    }

    context.done();
}

```

By default, all function routes are prefixed with *api*. You can also customize or remove the prefix using the `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.

```
{
    "http": {
        "routePrefix": ""
    }
}
```

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 2.x runtime. It is also currently only available for .NET languages.

In .NET languages, this information 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:

```
#r "Newtonsoft.Json"

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;
}
```

Authorization keys

Functions lets you use keys to make it harder to access your HTTP function endpoints during development. A standard HTTP trigger may require such an API key be present in the request.

IMPORTANT

While keys may help obfuscate your HTTP endpoints during development, they are not intended as a way to secure an HTTP trigger in production. To learn more, see [Secure an HTTP endpoint in production](#).

NOTE

In the Functions 1.x runtime, webhook providers may use keys to authorize requests in a variety of ways, depending on what the provider supports. This is covered in [Webhooks and keys](#). The version 2.x runtime does not include built-in support for webhook providers.

There are two types of keys:

- **Host keys:** These keys are shared by all functions within the function app. When used as an API key, these allow access to any function within the function app.
- **Function keys:** 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.

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.

Each function app also has a special **master key**. This key is a host key named `_master`, which provides administrative access to the runtime APIs. This key cannot be revoked. When you set an authorization level of `admin`, requests must use the master key; any other key results in authorization 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 authorization 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 a function app. On the left, there's a navigation sidebar with a search bar and sections for Visual Studio Enterprise, Function Apps, myfunctionapp, Functions, and HttpTriggerCSharp1. A red box highlights the 'Manage' button under the function. Below the sidebar, the main content area shows the 'Function Keys' section. It has a table with columns for NAME, VALUE, and ACTIONS. Two rows are listed: 'default' (Value: Click to show) and 'namedKey' (Value: Click to show). Each row has 'Copy', 'Renew', and 'Revoke' actions. A blue 'Add new function key' button is at the bottom. Below this, another table for 'Host Keys (All functions)' is shown, with rows for '_master' and 'default' (both Value: Click to show) and similar actions. A blue 'Add new host key' button is also present.

There is no supported API for programmatically obtaining function keys.

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 be 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 authentication level setting. After publishing to Azure, the `authLevel` setting in your trigger is enforced.

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:

- Turn on App Service Authentication / Authorization for your function app. The App Service platform lets use Azure Active Directory (AAD) and several third-party identity providers to authenticate clients. You can use this 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 to an Azure App Service Environment (ASE). 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](#).

When using one of these function app-level security methods, you should set the HTTP-triggered function authentication level to `anonymous`.

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.

In version 1.x, webhook templates provide additional validation for webhook payloads. In version 2.x, 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

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.

Trigger - 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 about 2.5 minutes, the gateway 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](#).

Trigger - host.json properties

The `host.json` file contains settings that control HTTP trigger behavior.

```
{  
    "http": {  
        "routePrefix": "api",  
        "maxOutstandingRequests": 200,  
        "maxConcurrentRequests": 100,  
        "dynamicThrottlesEnabled": true  
    }  
}
```

PROPERTY	DEFAULT	DESCRIPTION
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.
maxOutstandingRequests	200*	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows 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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 200. The default for version 2.x in a dedicated plan is unbounded (-1).

PROPERTY	DEFAULT	DESCRIPTION
maxConcurrentRequests	100*	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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 100. The default for version 2.x in a dedicated plan is unbounded (-1).
dynamicThrottlesEnabled	true*	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels. *The default for version 1.x is false. The default for version 2.x in a consumption plan is true. The default for version 2.x in a dedicated plan is false.

Output

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. If an HTTP output binding is not provided, an HTTP trigger returns HTTP 200 OK with an empty body in Functions 1.x, or HTTP 204 No Content with an empty body in Functions 2.x.

Output - 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
type	Must be set to <code>http</code> .
direction	Must be set to <code>out</code> .
name	The variable name used in function code for the response, or <code>\$return</code> to use the return value.

Output - 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](#).

Next steps

[Learn more about Azure functions triggers and bindings](#)

Microsoft Graph bindings for Azure Functions

7/1/2019 • 30 minutes to read • [Edit Online](#)

This article explains how to configure and work with Microsoft Graph triggers and bindings in Azure Functions. With these, you can use Azure Functions to work with data, insights, and events from the [Microsoft Graph](#).

The Microsoft Graph extension provides the following bindings:

- An [auth token input binding](#) allows you to interact with any Microsoft Graph API.
- An [Excel table input binding](#) allows you to read data from Excel.
- An [Excel table output binding](#) allows you to modify Excel data.
- A [OneDrive file input binding](#) allows you to read files from OneDrive.
- A [OneDrive file output binding](#) allows you to write to files in OneDrive.
- An [Outlook message output binding](#) allows you to send email through Outlook.
- A collection of [Microsoft Graph webhook triggers and bindings](#) allows you to react to events from the Microsoft Graph.

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

Microsoft Graph bindings are currently in preview for Azure Functions version 2.x. They are not supported in Functions version 1.x.

Packages

The auth token input binding is provided in the [Microsoft.Azure.WebJobs.Extensions.AuthTokens](#) NuGet package. The other Microsoft Graph bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#) package. Source code for the packages is in the [azure-functions-microsoftgraph-extension](#) GitHub repository.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Setting up the extensions

Microsoft Graph bindings are available through *binding extensions*. Binding extensions are optional components to the Azure Functions runtime. This section shows how to set up the Microsoft Graph and auth token extensions.

Enabling Functions 2.0 preview

Binding extensions are available only for Azure Functions 2.0 preview.

For information about how to set a function app to use the preview 2.0 version of the Functions runtime, see [How to target Azure Functions runtime versions](#).

Installing the extension

To install an extension from the Azure portal, navigate to either a template or binding that references it. Create a new function, and while in the template selection screen, choose the "Microsoft Graph" scenario. Select one of the templates from this scenario. Alternatively, you can navigate to the "Integrate" tab of an existing function and select one of the bindings covered in this article.

In both cases, a warning will appear which specifies the extension to be installed. Click **Install** to obtain the extension. Each extension only needs to be installed once per function app.

NOTE

The in-portal installation process can take up to 10 minutes on a consumption plan.

If you are using Visual Studio, you can get the extensions by installing the [NuGet packages that are listed earlier in this article](#).

Configuring Authentication / Authorization

The bindings outlined in this article require an identity to be used. This allows the Microsoft Graph to enforce permissions and audit interactions. The identity can be a user accessing your application or the application itself. To configure this identity, set up [App Service Authentication / Authorization](#) with Azure Active Directory. You will also need to request any resource permissions your functions require.

NOTE

The Microsoft Graph extension only supports Azure AD authentication. Users need to log in with a work or school account.

If you're using the Azure portal, you'll see a warning below the prompt to install the extension. The warning prompts you to configure App Service Authentication / Authorization and request any permissions the template or binding requires. Click **Configure Azure AD now** or **Add permissions now** as appropriate.

Auth token

The auth token input binding gets an Azure AD token for a given resource and provides it to your code as a

string. The resource can be any for which the application has permissions.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

Auth token - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Auth token - C# script example

The following example gets user profile information.

The *function.json* file defines an HTTP trigger with a token input binding:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "token",
      "direction": "in",
      "name": "graphToken",
      "resource": "https://graph.microsoft.com",
      "identity": "userFromRequest"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The C# script code uses the token to make an HTTP call to the Microsoft Graph and returns the result:

```
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using Microsoft.Extensions.Logging;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, string graphToken, ILogger log)
{
    HttpClient client = new HttpClient();
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", graphToken);
    return await client.GetAsync("https://graph.microsoft.com/v1.0/me/");
}
```

Auth token - JavaScript example

The following example gets user profile information.

The `function.json` file defines an HTTP trigger with a token input binding:

```
{  
  "bindings": [  
    {  
      "name": "req",  
      "type": "httpTrigger",  
      "direction": "in"  
    },  
    {  
      "type": "token",  
      "direction": "in",  
      "name": "graphToken",  
      "resource": "https://graph.microsoft.com",  
      "identity": "userFromRequest"  
    },  
    {  
      "name": "res",  
      "type": "http",  
      "direction": "out"  
    }  
  "disabled": false  
}
```

The JavaScript code uses the token to make an HTTP call to the Microsoft Graph and returns the result.

```
const rp = require('request-promise');  
  
module.exports = function (context, req) {  
  let token = "Bearer " + context.bindings.graphToken;  
  

```

Auth token - attributes

In [C# class libraries](#), use the `Token` attribute.

Auth token - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Token` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the auth token. See Using an auth token input binding from code .
type		Required - must be set to <code>token</code> .
direction		Required - must be set to <code>in</code> .
identity	Identity	Required - The identity that will be used to perform the action. Can be one of the following values: <ul style="list-style-type: none">• <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user.• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.• <code>clientCredentials</code> - Uses the identity of the function app.
userId	UserId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
Resource	resource	Required - An Azure AD resource URL for which the token is being requested.

Auth token - usage

The binding itself does not require any Azure AD permissions, but depending on how the token is used, you may need to request additional permissions. Check the requirements of the resource you intend to access with the token.

The token is always presented to code as a string.

NOTE

When developing locally with either of `userFromId`, `userFromToken` or `userFromRequest` options, required token can be [obtained manually](#) and specified in `X-MS-TOKEN-AAD-ID-TOKEN` request header from a calling client application.

Excel input

The Excel table input binding reads the contents of an Excel table stored in OneDrive.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

Excel input - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Excel input - C# script example

The following *function.json* file defines an HTTP trigger with an Excel input binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "excel",
      "direction": "in",
      "name": "excelTableData",
      "path": "{query.workbook}",
      "identity": "UserFromRequest",
      "tableName": "{query.table}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The following C# script code reads the contents of the specified table and returns them to the user:

```
using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Microsoft.Extensions.Logging;

public static IActionResult Run(HttpContext req, string[][] excelTableData, ILogger log)
{
    return new OkObjectResult(excelTableData);
}
```

Excel input - JavaScript example

The following *function.json* file defines an HTTP trigger with an Excel input binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "excel",
      "direction": "in",
      "name": "excelTableData",
      "path": "{query.workbook}",
      "identity": "UserFromRequest",
      "tableName": "{query.table}"
    },
    {
      "name": "res",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The following JavaScript code reads the contents of the specified table and returns them to the user.

```
module.exports = function (context, req) {
  context.res = {
    body: context.bindings.excelTableData
  };
  context.done();
};
```

Excel input - attributes

In C# class libraries, use the [Excel](#) attribute.

Excel input - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the [Excel](#) attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the Excel table. See Using an Excel table input binding from code .
type		Required - must be set to <code>excel</code> .
direction		Required - must be set to <code>in</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
identity	Identity	<p>Required - The identity that will be used to perform the action. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user. • <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property. • <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property. • <code>clientCredentials</code> - Uses the identity of the function app.
userId	UserId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
path	Path	Required - the path in OneDrive to the Excel workbook.
worksheetName	WorksheetName	The worksheet in which the table is found.
tableName	TableName	The name of the table. If not specified, the contents of the worksheet will be used.

Excel input - usage

This binding requires the following Azure AD permissions:

RESOURCE	PERMISSION
Microsoft Graph	Read user files

The binding exposes the following types to .NET functions:

- `string[][]`
- `Microsoft.Graph.WorkbookTable`
- Custom object types (using structural model binding)

Excel output

The Excel output binding modifies the contents of an Excel table stored in OneDrive.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

Excel output - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Excel output - C# script example

The following example adds rows to an Excel table.

The *function.json* file defines an HTTP trigger with an Excel output binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "newExcelRow",
      "type": "excel",
      "direction": "out",
      "identity": "userFromRequest",
      "updateType": "append",
      "path": "{query.workbook}",
      "tableName": "{query.table}"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The C# script code adds a new row to the table (assumed to be single-column) based on input from the query string:

```

using System.Net;
using System.Text;
using Microsoft.Extensions.Logging;

public static async Task Run(HttpContext req, IAsyncCollector<object> newExcelRow, ILogger log)
{
    string input = req.Query
        .FirstOrDefault(q => string.Compare(q.Key, "text", true) == 0)
        .Value;
    await newExcelRow.AddAsync(new {
        Text = input
        // Add other properties for additional columns here
    });
    return;
}

```

Excel output - JavaScript example

The following example adds rows to an Excel table.

The `function.json` file defines an HTTP trigger with an Excel output binding:

```

{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "newExcelRow",
      "type": "excel",
      "direction": "out",
      "identity": "userFromRequest",
      "updateType": "append",
      "path": "{query.workbook}",
      "tableName": "{query.table}"
    },
    {
      "name": "res",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}

```

The following JavaScript code adds a new row to the table (assumed to be single-column) based on input from the query string.

```

module.exports = function (context, req) {
    context.bindings.newExcelRow = {
        text: req.query.text
        // Add other properties for additional columns here
    }
    context.done();
};

```

Excel output - attributes

In [C# class libraries](#), use the `Excel` attribute.

Excel output - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `Excel` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the auth token. See Using an Excel table output binding from code .
type		Required - must be set to <code>excel</code> .
direction		Required - must be set to <code>out</code> .
identity	Identity	Required - The identity that will be used to perform the action. Can be one of the following values: <ul style="list-style-type: none"> • <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user. • <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property. • <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property. • <code>clientCredentials</code> - Uses the identity of the function app.
UserId	userId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
path	Path	Required - the path in OneDrive to the Excel workbook.
worksheetName	WorksheetName	The worksheet in which the table is found.
tableName	TableName	The name of the table. If not specified, the contents of the worksheet will be used.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
updateType	UpdateType	<p>Required - The type of change to make to the table. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>update</code> - Replaces the contents of the table in OneDrive. • <code>append</code> - Adds the payload to the end of the table in OneDrive by creating new rows.

Excel output - usage

This binding requires the following Azure AD permissions:

RESOURCE	PERMISSION
Microsoft Graph	Have full access to user files

The binding exposes the following types to .NET functions:

- `string[][]`
- `Newtonsoft.Json.Linq JObject`
- `Microsoft.Graph.WorkbookTable`
- Custom object types (using structural model binding)

File input

The OneDrive File input binding reads the contents of a file stored in OneDrive.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

File input - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

File input - C# script example

The following example reads a file that is stored in OneDrive.

The `function.json` file defines an HTTP trigger with a OneDrive file input binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "myOneDriveFile",
      "type": "onedrive",
      "direction": "in",
      "path": "{query.filename}",
      "identity": "userFromRequest"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The C# script code reads the file specified in the query string and logs its length:

```
using System.Net;
using Microsoft.Extensions.Logging;

public static void Run(HttpRequestMessage req, Stream myOneDriveFile, ILogger log)
{
    log.LogInformation(myOneDriveFile.Length.ToString());
}
```

File input - JavaScript example

The following example reads a file that is stored in OneDrive.

The *function.json* file defines an HTTP trigger with a OneDrive file input binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "myOneDriveFile",
      "type": "onedrive",
      "direction": "in",
      "path": "{query.filename}",
      "identity": "userFromRequest"
    },
    {
      "name": "res",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The following JavaScript code reads the file specified in the query string and returns its length.

```
module.exports = function (context, req) {
    context.res = {
        body: context.bindings.myOneDriveFile.length
    };
    context.done();
};
```

File input - attributes

In C# class libraries, use the [OneDrive](#) attribute.

File input - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the `OneDrive` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the file. See Using a OneDrive file input binding from code .
type		Required - must be set to <code>onedrive</code> .
direction		Required - must be set to <code>in</code> .
identity	Identity	Required - The identity that will be used to perform the action. Can be one of the following values: <ul style="list-style-type: none">• <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user.• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.• <code>clientCredentials</code> - Uses the identity of the function app.
userId	UserId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
path	Path	Required - the path in OneDrive to the file.

File input - usage

This binding requires the following Azure AD permissions:

RESOURCE	PERMISSION
Microsoft Graph	Read user files

The binding exposes the following types to .NET functions:

- byte[]
- Stream
- string
- Microsoft.Graph.DriveItem

File output

The OneDrive file output binding modifies the contents of a file stored in OneDrive.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

File output - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

File output - C# script example

The following example writes to a file that is stored in OneDrive.

The *function.json* file defines an HTTP trigger with a OneDrive output binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "myOneDriveFile",
      "type": "onedrive",
      "direction": "out",
      "path": "FunctionsTest.txt",
      "identity": "userFromRequest"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The C# script code gets text from the query string and writes it to a text file (FunctionsTest.txt as defined in the preceding example) at the root of the caller's OneDrive:

```
using System.Net;
using System.Text;
using Microsoft.Extensions.Logging;

public static async Task Run(HttpRequest req, ILogger log, Stream myOneDriveFile)
{
    string data = req.Query
        .FirstOrDefault(q => string.Compare(q.Key, "text", true) == 0)
        .Value;
    await myOneDriveFile.WriteAsync(Encoding.UTF8.GetBytes(data), 0, data.Length);
    myOneDriveFile.Close();
    return;
}
```

File output - JavaScript example

The following example writes to a file that is stored in OneDrive.

The *function.json* file defines an HTTP trigger with a OneDrive output binding:

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "myOneDriveFile",
      "type": "onedrive",
      "direction": "out",
      "path": "FunctionsTest.txt",
      "identity": "userFromRequest"
    },
    {
      "name": "res",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The JavaScript code gets text from the query string and writes it to a text file (FunctionsTest.txt as defined in the config above) at the root of the caller's OneDrive.

```
module.exports = function (context, req) {
  context.bindings.myOneDriveFile = req.query.text;
  context.done();
};
```

File output - attributes

In [C# class libraries](#), use the [OneDrive](#) attribute.

File output - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the [OneDrive](#) attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for file. See Using a OneDrive file output binding from code .
type		Required - must be set to onedrive .
direction		Required - must be set to out .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
identity	Identity	<p>Required - The identity that will be used to perform the action. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user. • <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property. • <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property. • <code>clientCredentials</code> - Uses the identity of the function app.
UserId	userId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
path	Path	Required - the path in OneDrive to the file.

File output - usage

This binding requires the following Azure AD permissions:

RESOURCE	PERMISSION
Microsoft Graph	Have full access to user files

The binding exposes the following types to .NET functions:

- `byte[]`
- `Stream`
- `string`
- `Microsoft.Graph.DriveItem`

Outlook output

The Outlook message output binding sends a mail message through Outlook.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)

- [Usage](#)

Outlook output - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Outlook output - C# script example

The following example sends an email through Outlook.

The `function.json` file defines an HTTP trigger with an Outlook message output binding:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "message",
      "type": "outlook",
      "direction": "out",
      "identity": "userFromRequest"
    }
  ],
  "disabled": false
}
```

The C# script code sends a mail from the caller to a recipient specified in the query string:

```
using System.Net;
using Microsoft.Extensions.Logging;

public static void Run(HttpRequest req, out Message message, ILogger log)
{
    string emailAddress = req.Query["to"];
    message = new Message(){
        subject = "Greetings",
        body = "Sent from Azure Functions",
        recipient = new Recipient() {
            address = emailAddress
        }
    };
}

public class Message {
    public String subject {get; set;}
    public String body {get; set;}
    public Recipient recipient {get; set;}
}

public class Recipient {
    public String address {get; set;}
    public String name {get; set;}
}
```

Outlook output - JavaScript example

The following example sends an email through Outlook.

The `function.json` file defines an HTTP trigger with an Outlook message output binding:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "name": "message",
      "type": "outlook",
      "direction": "out",
      "identity": "userFromRequest"
    }
  ],
  "disabled": false
}
```

The JavaScript code sends a mail from the caller to a recipient specified in the query string:

```
module.exports = function (context, req) {
  context.bindings.message = {
    subject: "Greetings",
    body: "Sent from Azure Functions with JavaScript",
    recipient: {
      address: req.query.to
    }
  };
  context.done();
};
```

Outlook output - attributes

In [C# class libraries](#), use the [Outlook](#) attribute.

Outlook output - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the `outlook` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the mail message. See Using an Outlook message output binding from code .
type		Required - must be set to <code>outlook</code> .
direction		Required - must be set to <code>out</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
identity	Identity	<p>Required - The identity that will be used to perform the action. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user. • <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property. • <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property. • <code>clientCredentials</code> - Uses the identity of the function app.
userId	UserId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.

Outlook output - usage

This binding requires the following Azure AD permissions:

RESOURCE	PERMISSION
Microsoft Graph	Send mail as user

The binding exposes the following types to .NET functions:

- Microsoft.Graph.Message
- Newtonsoft.Json.Linq JObject
- string
- Custom object types (using structural model binding)

Webhooks

Webhooks allow you to react to events in the Microsoft Graph. To support webhooks, functions are needed to create, refresh, and react to *webhook subscriptions*. A complete webhook solution requires a combination of the following bindings:

- A [Microsoft Graph webhook trigger](#) allows you to react to an incoming webhook.
- A [Microsoft Graph webhook subscription input binding](#) allows you to list existing subscriptions and optionally refresh them.
- A [Microsoft Graph webhook subscription output binding](#) allows you to create or delete webhook subscriptions.

The bindings themselves do not require any Azure AD permissions, but you need to request permissions relevant to the resource type you wish to react to. For a list of which permissions are needed for each resource type, see [subscription permissions](#).

For more information about webhooks, see [Working with webhooks in Microsoft Graph](#).

Webhook trigger

The Microsoft Graph webhook trigger allows a function to react to an incoming webhook from the Microsoft Graph. Each instance of this trigger can react to one Microsoft Graph resource type.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

Webhook trigger - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Webhook trigger - C# script example

The following example handles webhooks for incoming Outlook messages. To use a webhook trigger you [create a subscription](#), and you can [refresh the subscription](#) to prevent it from expiring.

The *function.json* file defines a webhook trigger:

```
{
  "bindings": [
    {
      "name": "msg",
      "type": "GraphWebhookTrigger",
      "direction": "in",
      "resourceType": "#Microsoft.Graph.Message"
    }
  ],
  "disabled": false
}
```

The C# script code reacts to incoming mail messages and logs the body of those sent by the recipient and containing "Azure Functions" in the subject:

```
#r "Microsoft.Graph"
using Microsoft.Graph;
using System.Net;
using Microsoft.Extensions.Logging;

public static async Task Run(Message msg, ILogger log)
{
    log.LogInformation("Microsoft Graph webhook trigger function processed a request.");

    // Testable by sending oneself an email with the subject "Azure Functions" and some text body
    if (msg.Subject.Contains("Azure Functions") && msg.From.Equals(msg.Sender)) {
        log.LogInformation($"Processed email: {msg.BodyPreview}");
    }
}
```

Webhook trigger - JavaScript example

The following example handles webhooks for incoming Outlook messages. To use a webhook trigger you [create a subscription](#), and you can [refresh the subscription](#) to prevent it from expiring.

The `function.json` file defines a webhook trigger:

```
{  
  "bindings": [  
    {  
      "name": "msg",  
      "type": "GraphWebhookTrigger",  
      "direction": "in",  
      "resourceType": "#Microsoft.Graph.Message"  
    }  
  ],  
  "disabled": false  
}
```

The JavaScript code reacts to incoming mail messages and logs the body of those sent by the recipient and containing "Azure Functions" in the subject:

```
module.exports = function (context) {  
  context.log("Microsoft Graph webhook trigger function processed a request.");  
  const msg = context.bindings.msg  
  // Testable by sending oneself an email with the subject "Azure Functions" and some text body  
  if((msg.subject.indexOf("Azure Functions") > -1) && (msg.from === msg.sender) ) {  
    context.log(`Processed email: ${msg.bodyPreview}`);  
  }  
  context.done();  
};
```

Webhook trigger - attributes

In [C# class libraries](#), use the `GraphWebHookTrigger` attribute.

Webhook trigger - configuration

The following table explains the binding configuration properties that you set in the `function.json` file and the `GraphWebHookTrigger` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the mail message. See Using an Outlook message output binding from code .
type		Required - must be set to <code>graphWebhook</code> .
direction		Required - must be set to <code>trigger</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
resourceType	ResourceType	<p>Required - the graph resource for which this function should respond to webhooks. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>#Microsoft.Graph.Message</code> - changes made to Outlook messages. • <code>#Microsoft.Graph.DriveItem</code> - changes made to OneDrive root items. • <code>#Microsoft.Graph.Contact</code> - changes made to personal contacts in Outlook. • <code>#Microsoft.Graph.Event</code> - changes made to Outlook calendar items.

NOTE

A function app can only have one function that is registered against a given `resourceType` value.

Webhook trigger - usage

The binding exposes the following types to .NET functions:

- Microsoft Graph SDK types relevant to the resource type, such as `Microsoft.Graph.Message` or `Microsoft.Graph.DriveItem`.
- Custom object types (using structural model binding)

Webhook input

The Microsoft Graph webhook input binding allows you to retrieve the list of subscriptions managed by this function app. The binding reads from function app storage, so it does not reflect other subscriptions created from outside the app.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

Webhook input - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Webhook input - C# script example

The following example gets all subscriptions for the calling user and deletes them.

The `function.json` file defines an HTTP trigger with a subscription input binding and a subscription output binding that uses the delete action:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "existingSubscriptions",
      "direction": "in",
      "filter": "userFromRequest"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "subscriptionsToDelete",
      "direction": "out",
      "action": "delete",
      "identity": "userFromRequest"
    },
    {
      "type": "http",
      "name": "res",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The C# script code gets the subscriptions and deletes them:

```
using System.Net;
using Microsoft.Extensions.Logging;

public static async Task Run(HttpContext req, string[] existingSubscriptions, IAsyncCollector<string> subscriptionsToDelete, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");
    foreach (var subscription in existingSubscriptions)
    {
        log.LogInformation($"Deleting subscription {subscription}");
        await subscriptionsToDelete.AddAsync(subscription);
    }
}
```

Webhook input - JavaScript example

The following example gets all subscriptions for the calling user and deletes them.

The *function.json* file defines an HTTP trigger with a subscription input binding and a subscription output binding that uses the delete action:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "existingSubscriptions",
      "direction": "in",
      "filter": "userFromRequest"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "subscriptionsToDelete",
      "direction": "out",
      "action": "delete",
      "identity": "userFromRequest"
    },
    {
      "type": "http",
      "name": "res",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The JavaScript code gets the subscriptions and deletes them:

```
module.exports = function (context, req) {
  const existing = context.bindings.existingSubscriptions;
  var toDelete = [];
  for (var i = 0; i < existing.length; i++) {
    context.log(`Deleting subscription ${existing[i]}`);
    toDelete.push(existing[i]);
  }
  context.bindings.subscriptionsToDelete = toDelete;
  context.done();
};
```

Webhook input - attributes

In [C# class libraries](#), use the [GraphWebHookSubscription](#) attribute.

Webhook input - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the [GraphWebHookSubscription](#) attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the mail message. See Using an Outlook message output binding from code .
type		Required - must be set to graphWebhookSubscription .
direction		Required - must be set to in .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
filter	Filter	If set to <code>userFromRequest</code> , then the binding will only retrieve subscriptions owned by the calling user (valid only with HTTP trigger).

Webhook input - usage

The binding exposes the following types to .NET functions:

- `string[]`
- Custom object type arrays
- `Newtonsoft.Json.Linq JObject[]`
- `Microsoft.Graph.Subscription[]`

Webhook output

The webhook subscription output binding allows you to create, delete, and refresh webhook subscriptions in the Microsoft Graph.

This section contains the following subsections:

- [Example](#)
- [Attributes](#)
- [Configuration](#)
- [Usage](#)

Webhook output - example

See the language-specific example:

- [C# script \(.csx\)](#)
- [JavaScript](#)

Webhook output - C# script example

The following example creates a subscription. You can [refresh the subscription](#) to prevent it from expiring.

The `function.json` file defines an HTTP trigger with a subscription output binding using the create action:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "clientState",
      "direction": "out",
      "action": "create",
      "subscriptionResource": "me/mailFolders('Inbox')/messages",
      "changeTypes": [
        "created"
      ],
      "identity": "userFromRequest"
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The C# script code registers a webhook that will notify this function app when the calling user receives an Outlook message:

```
using System;
using System.Net;
using Microsoft.Extensions.Logging;

public static HttpResponseMessage run(HttpRequestMessage req, out string clientState, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");
    clientState = Guid.NewGuid().ToString();
    return new HttpResponseMessage(HttpStatusCode.OK);
}
```

Webhook output - JavaScript example

The following example creates a subscription. You can [refresh the subscription](#) to prevent it from expiring.

The *function.json* file defines an HTTP trigger with a subscription output binding using the create action:

```
{
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "clientState",
      "direction": "out",
      "action": "create",
      "subscriptionResource": "me/mailFolders('Inbox')/messages",
      "changeTypes": [
        "created"
      ],
      "identity": "userFromRequest"
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The JavaScript code registers a webhook that will notify this function app when the calling user receives an Outlook message:

```
const uuidv4 = require('uuid/v4');

module.exports = function (context, req) {
  context.bindings.clientState = uuidv4();
  context.done();
};
```

Webhook output - attributes

In [C# class libraries](#), use the [GraphWebHookSubscription](#) attribute.

Webhook output - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the [GraphWebHookSubscription](#) attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		Required - the variable name used in function code for the mail message. See Using an Outlook message output binding from code .
type		Required - must be set to graphWebhookSubscription .
direction		Required - must be set to out .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
identity	Identity	<p>Required - The identity that will be used to perform the action. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>userFromRequest</code> - Only valid with HTTP trigger. Uses the identity of the calling user. • <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property. • <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property. • <code>clientCredentials</code> - Uses the identity of the function app.
userId	UserId	Needed if and only if <i>identity</i> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
userToken	UserToken	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
action	Action	<p>Required - specifies the action the binding should perform. Can be one of the following values:</p> <ul style="list-style-type: none"> • <code>create</code> - Registers a new subscription. • <code>delete</code> - Deletes a specified subscription. • <code>refresh</code> - Refreshes a specified subscription to keep it from expiring.
subscriptionResource	SubscriptionResource	Needed if and only if the <i>action</i> is set to <code>create</code> . Specifies the Microsoft Graph resource that will be monitored for changes. See Working with webhooks in Microsoft Graph .
changeType	ChangeType	Needed if and only if the <i>action</i> is set to <code>create</code> . Indicates the type of change in the subscribed resource that will raise a notification. The supported values are: <code>created</code> , <code>updated</code> , <code>deleted</code> . Multiple values can be combined using a comma-separated list.

Webhook output - usage

The binding exposes the following types to .NET functions:

- string
- Microsoft.Graph.Subscription

Webhook subscription refresh

There are two approaches to refreshing subscriptions:

- Use the application identity to deal with all subscriptions. This will require consent from an Azure Active Directory admin. This can be used by all languages supported by Azure Functions.
- Use the identity associated with each subscription by manually binding each user ID. This will require some custom code to perform the binding. This can only be used by .NET functions.

This section contains an example for each of these approaches:

- [App identity example](#)
- [User identity example](#)

Webhook Subscription refresh - app identity example

See the language-specific example:

- [C# script \(.csx\)](#)
- JavaScript

App identity refresh - C# script example

The following example uses the application identity to refresh a subscription.

The *function.json* defines a timer trigger with a subscription input binding and a subscription output binding:

```
{
  "bindings": [
    {
      "name": "myTimer",
      "type": "timerTrigger",
      "direction": "in",
      "schedule": "0 * * */2 * *"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "existingSubscriptions",
      "direction": "in"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "subscriptionsToRefresh",
      "direction": "out",
      "action": "refresh",
      "identity": "clientCredentials"
    }
  ],
  "disabled": false
}
```

The C# script code refreshes the subscriptions:

```

using System;
using Microsoft.Extensions.Logging;

public static void Run(TimerInfo myTimer, string[] existingSubscriptions, ICollector<string>
subscriptionsToRefresh, ILogger log)
{
    // This template uses application permissions and requires consent from an Azure Active Directory
    admin.
    // See https://go.microsoft.com/fwlink/?linkid=858780
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
    foreach (var subscription in existingSubscriptions)
    {
        log.LogInformation($"Refreshing subscription {subscription}");
        subscriptionsToRefresh.Add(subscription);
    }
}

```

App identity refresh - C# script example

The following example uses the application identity to refresh a subscription.

The *function.json* defines a timer trigger with a subscription input binding and a subscription output binding:

```

{
  "bindings": [
    {
      "name": "myTimer",
      "type": "timerTrigger",
      "direction": "in",
      "schedule": "0 * * */2 * *"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "existingSubscriptions",
      "direction": "in"
    },
    {
      "type": "graphWebhookSubscription",
      "name": "subscriptionsToRefresh",
      "direction": "out",
      "action": "refresh",
      "identity": "clientCredentials"
    }
  ],
  "disabled": false
}

```

The JavaScript code refreshes the subscriptions:

```

// This template uses application permissions and requires consent from an Azure Active Directory
admin.

// See https://go.microsoft.com/fwlink/?linkid=858780

module.exports = function (context) {
    const existing = context.bindings.existingSubscriptions;
    var toRefresh = [];
    for (var i = 0; i < existing.length; i++) {
        context.log(`Refreshing subscription ${existing[i]}`);
        toRefresh.push(existing[i]);
    }
    context.bindings.subscriptionsToRefresh = toRefresh;
    context.done();
};


```

Webhook Subscription refresh - user identity example

The following example uses the user identity to refresh a subscription.

The *function.json* file defines a timer trigger and defers the subscription input binding to the function code:

```

{
    "bindings": [
        {
            "name": "myTimer",
            "type": "timerTrigger",
            "direction": "in",
            "schedule": "0 * * */2 * *"
        },
        {
            "type": "graphWebhookSubscription",
            "name": "existingSubscriptions",
            "direction": "in"
        }
    ],
    "disabled": false
}

```

The C# script code refreshes the subscriptions and creates the output binding in code, using each user's identity:

```
using System;
using Microsoft.Extensions.Logging;

public static async Task Run(TimerInfo myTimer, UserSubscription[] existingSubscriptions, IBinder
binder, ILogger log)
{
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
    foreach (var subscription in existingSubscriptions)
    {
        // binding in code to allow dynamic identity
        using (var subscriptionsToRefresh = await binder.BindAsync<IAsyncCollector<string>>(
            new GraphWebhookSubscriptionAttribute() {
                Action = "refresh",
                Identity = "userFromId",
                UserId = subscription.UserId
            }
        ))
        {
            log.LogInformation($"Refreshing subscription {subscription}");
            await subscriptionsToRefresh.AddAsync(subscription);
        }
    }
}

public class UserSubscription {
    public string UserId {get; set;}
    public string Id {get; set;}
}
```

Next steps

[Learn more about Azure functions triggers and bindings](#)

Mobile Apps bindings for Azure Functions

7/1/2019 • 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.

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		Must be set to "mobileTable"
direction		Must be set to "in"
name		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 + "'"  
  }  
  
  context.done();  
};
```

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>		Must be set to "mobileTable"
<code>direction</code>		Must be set to "out"
<code>name</code>		Name of output parameter in function signature.
<code>tableName</code>	TableName	Name of the mobile app's data table
<code>connection</code>	MobileAppUriSetting	The name of an app setting that has the mobile app's URL. The function uses this URL to construct the required REST operations against your mobile app. Create an app setting in your function app that contains the mobile app's URL, then specify the name of the app setting in the <code>connection</code> property in your input binding. The URL looks like <code>http://<appname>.azurewebsites.net</code>
<code>apiKey</code>	ApiKeySetting	The name of an app setting that has your mobile app's API key. Provide the API key if you implement an API key in your Node.js mobile app backend , or implement an API key in your .NET mobile app backend . To provide the key, create an app setting in your function app that contains the API key, then add the <code>apiKey</code> property in your input binding with the name of the app setting.

When you're developing locally, 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

7/1/2019 • 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

This binding is not available in Functions 2.x.

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
type	n/a	Must be set to "notificationHub".
direction	n/a	Must be set to "out".
name	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.

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 bindings for Azure Functions

7/1/2019 • 15 minutes to read • [Edit Online](#)

This article explains how to work with Azure Queue storage bindings in Azure Functions. Azure Functions supports trigger and output bindings for queues.

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 Queue 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](#).

Packages - Functions 2.x

The Queue 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.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

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.

Trigger

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.

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)
- [Java](#)

Trigger - C# example

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}");
    }
}
```

Trigger - C# script example

The following example shows a queue trigger binding in a *function.json* file and [C# script \(.csx\)](#) code that uses the binding. The function polls the `myqueue-items` queue and writes a log each time a queue item is processed.

Here's the *function.json* file:

```
{
  "disabled": false,
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "myQueueItem",
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting"
    }
  ]
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```

#r "Microsoft.WindowsAzure.Storage"

using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.Queue;
using System;

public static void Run(CloudQueueMessage myQueueItem,
    DateTimeOffset expirationTime,
    DateTimeOffset insertionTime,
    DateTimeOffset nextVisibleTime,
    string queueTrigger,
    string id,
    string popReceipt,
    int dequeueCount,
    ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem.AsString}\n" +
        $"queueTrigger={queueTrigger}\n" +
        $"expirationTime={expirationTime}\n" +
        $"insertionTime={insertionTime}\n" +
        $"nextVisibleTime={nextVisibleTime}\n" +
        $"id={id}\n" +
        $"popReceipt={popReceipt}\n" +
        $"dequeueCount={dequeueCount}");
}

```

The [usage](#) section explains `myQueueItem`, which is named by the `name` property in `function.json`. The [message metadata section](#) explains all of the other variables shown.

Trigger - JavaScript example

The following example shows a queue trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function polls the `myqueue-items` queue and writes a log each time a queue item is processed.

Here's the `function.json` file:

```
{
  "disabled": false,
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "myQueueItem",
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting"
    }
  ]
}
```

The [configuration](#) section explains these properties.

NOTE

The name parameter reflects as `context.bindings.<name>` in the JavaScript code which contains the queue item payload. This payload is also passed as the second parameter to the function.

Here's the JavaScript code:

```

module.exports = async function (context, message) {
    context.log('Node.js queue trigger function processed work item', message);
    // OR access using context.bindings.<name>
    // context.log('Node.js queue trigger function processed work item', context.bindings.myQueueItem);
    context.log('expirationTime =', context.bindingData.expirationTime);
    context.log('insertionTime =', context.bindingData.insertionTime);
    context.log('nextVisibleTime =', context.bindingData.nextVisibleTime);
    context.log('id =', context.bindingData.id);
    context.log('popReceipt =', context.bindingData.popReceipt);
    context.log('dequeueCount =', context.bindingData.dequeueCount);
    context.done();
};

```

The [usage](#) section explains `myQueueItem`, which is named by the `name` property in `function.json`. The [message metadata section](#) explains all of the other variables shown.

Trigger - Java example

The following Java example shows a storage queue trigger functions which logs the triggered message placed into queue `myqueuename`.

```

@FunctionName("queueprocessor")
public void run(
    @QueueTrigger(name = "msg",
        queueName = "myqueuename",
        connection = "myconnvarname") String message,
    final ExecutionContext context
) {
    context.getLogger().info(message);
}

```

Trigger - attributes

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 storage account 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 [Trigger - 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("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.

Trigger - 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
type	n/a	Must be set to <code>queueTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
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.

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 "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](#).

Trigger - usage

In C# and C# script, access the message data by using a method parameter such as `string paramName`. In C# script, `paramName` is the value specified in the `name` property of `function.json`. 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](#).

In JavaScript, use `context.bindings.<name>` to access the queue item payload. If the payload is JSON, it's deserialized into an object.

Trigger - 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. These are properties of the [CloudQueueMessage](#) class.

PROPERTY	TYPE	DESCRIPTION
<code>QueueTrigger</code>	<code>string</code>	Queue payload (if a valid string). If the queue message payload as a string, <code>QueueTrigger</code> has the same value as the variable named by the <code>name</code> property in <code>function.json</code> .
<code>DequeueCount</code>	<code>int</code>	The number of times this message has been dequeued.
<code>ExpirationTime</code>	<code>DatetimeOffset</code>	The time that the message expires.

PROPERTY	TYPE	DESCRIPTION
<code>Id</code>	<code>string</code>	Queue message ID.
<code>InsertionTime</code>	<code>DateTimeOffset</code>	The time that the message was added to the queue.
<code>NextVisibleTime</code>	<code>DateTimeOffset</code>	The time that the message will next be visible.
<code>PopReceipt</code>	<code>string</code>	The message's pop receipt.

Trigger - poison messages

When a queue trigger function fails, Azure Functions retries the function up to five times for a given queue message, including the first try. If all five attempts fail, the functions runtime adds a message to a queue named `<originalqueueusername>-poison`. You can write a function to process messages from the poison queue by logging them or sending a notification that manual attention is needed.

To handle poison messages manually, check the [dequeueCount](#) of the queue message.

Trigger - polling algorithm

The queue trigger implements a random exponential back-off algorithm to reduce the effect of idle-queue polling on storage transaction costs. 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](#).

Trigger - 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; functions do not have to be written to be idempotent.

Trigger - 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.

Output

Use the Azure Queue storage output binding to write messages to a queue.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)
- [Java](#)

Output - C# example

The following example shows a [C# function](#) that creates a queue message for each HTTP request received.

```
[StorageAccount("AzureWebJobsStorage")]
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;
    }
}
```

Output - C# script example

The following example shows an HTTP trigger binding in a *function.json* file and [C# script \(.csx\)](#) code that uses the binding. The function creates a queue item with a **CustomQueueMessage** object payload for each HTTP request received.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "function",
      "name": "input"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "return"
    },
    {
      "type": "queue",
      "direction": "out",
      "name": "$return",
      "queueName": "outqueue",
      "connection": "MyStorageConnectionAppSetting"
    }
  ]
}
```

The [configuration](#) section explains these properties.

Here's C# script code that creates a single queue message:

```

public class CustomQueueMessage
{
    public string PersonName { get; set; }
    public string Title { get; set; }
}

public static CustomQueueMessage Run(CustomQueueMessage input, ILogger log)
{
    return input;
}

```

You can send multiple messages at once by using an `ICollector` or `IAsyncCollector` parameter. Here's C# script code that sends multiple messages, one with the HTTP request data and one with hard-coded values:

```

public static void Run(
    CustomQueueMessage input,
    ICollector<CustomQueueMessage> myQueueItems,
    ILogger log)
{
    myQueueItems.Add(input);
    myQueueItems.Add(new CustomQueueMessage { PersonName = "You", Title = "None" });
}

```

Output - JavaScript example

The following example shows an HTTP trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function creates a queue item for each HTTP request received.

Here's the `function.json` file:

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "function",
      "name": "input"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "return"
    },
    {
      "type": "queue",
      "direction": "out",
      "name": "$return",
      "queueName": "outqueue",
      "connection": "MyStorageConnectionAppSetting"
    }
  ]
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```

module.exports = function (context, input) {
    context.done(null, input.body);
};

```

You can send multiple messages at once by defining a message array for the `myQueueItem` output binding. The following JavaScript code sends two queue messages with hard-coded values for each HTTP request received.

```
module.exports = function(context) {
    context.bindings.myQueueItem = ["message 1", "message 2"];
    context.done();
};
```

Output - Java example

The following example shows a Java function that creates a queue message for when triggered by a HTTP request.

```
@FunctionName("httpToQueue")
@QueueOutput(name = "item", queueName = "myqueue-items", connection = "AzureWebJobsStorage")
public String pushToQueue(
    @HttpTrigger(name = "request", methods = {HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
    final String message,
    @HttpOutput(name = "response") final OutputBinding<String> result) {
    result.setValue(message + " has been added.");
    return message;
}
```

In the [Java functions runtime library](#), use the `@QueueOutput` annotation on parameters whose value would be written to Queue storage. The parameter type should be `OutputBinding<T>`, where T is any native Java type or a POJO.

Output - attributes

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 - C# example](#).

You can use the `StorageAccount` attribute to specify the storage account at class, method, or parameter level.

For more information, see [Trigger - attributes](#).

Output - 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
type	n/a	Must be set to <code>queue</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	n/a	Must be set to <code>out</code> . 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 in function code. Set to <code>\$return</code> to reference the function return value.
queueName	QueueName	The name of the queue.
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 "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](#).

Output - usage

In C# and C# script, write a single queue message by using a method parameter such as `out T paramName`. In C# script, `paramName` is the value specified in the `name` property of `function.json`. 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`

In JavaScript functions, use `context.bindings.<name>` to access the output queue message. You can use a string or a JSON-serializable object for the queue item payload.

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 version 2.x. The example host.json file below contains only the version 2.x settings for this binding. For more information about global configuration settings in version 2.x, see [host.json reference for Azure Functions version 2.x](#).

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).
visibilityTimeout	00:00:00	The time interval between retries when processing of a message fails.

PROPERTY	DEFAULT	DESCRIPTION
batchSize	16	<p>The number of queue messages that the Functions runtime retrieves simultaneously and processes in parallel. When the number being processed gets down to the <code>newBatchThreshold</code>, the runtime gets another batch and starts processing those messages. So the maximum number of concurrent messages being processed per function is <code>batchsize</code> plus <code>newBatchThreshold</code>. This limit applies separately to each queue-triggered function.</p> <p>If you want to avoid parallel execution for messages received on one queue, you can set <code>batchsize</code> to 1. However, this setting eliminates concurrency only so long as your function app runs on a single virtual machine (VM). If the function app scales out to multiple VMs, each VM could run one instance of each queue-triggered function.</p> <p>The maximum <code>batchSize</code> is 32.</p>
maxDequeueCount	5	The number of times to try processing a message before moving it to the poison queue.
newBatchThreshold	batchSize/2	Whenever the number of messages being processed concurrently gets down to this number, the runtime retrieves another batch.

Next steps

- [Learn more about Azure functions triggers and bindings](#)

[Go to a tutorial that uses a Queue storage output binding](#)

Azure Functions SendGrid bindings

7/1/2019 • 5 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

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.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)
- [Java](#)

C# example

The following example shows a [C# function](#) that uses a Service Bus queue trigger and a SendGrid output binding.

Synchronous C# example:

```
[FunctionName("SendEmail")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")] Message email,
    [SendGrid(ApiKey = "CustomSendGridKeyAppSettingName")] out SendGridMessage message)
{
    var emailObject = JsonConvert.DeserializeObject<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 C# example:

```
[FunctionName("SendEmail")]
public static async void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")] Message email,
    [SendGrid(ApiKey = "CustomSendGridKeyAppSettingName")] IAsyncCollector<SendGridMessage> messageCollector)
{
    var emailObject = JsonConvert.DeserializeObject<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".

C# script example

The following example shows a SendGrid output binding in a *function.json* file and a [C# script function](#) that uses the binding.

Here's the binding data in the *function.json* file:

```
{  
  "bindings": [  
    {  
      "type": "queueTrigger",  
      "name": "mymsg",  
      "queueName": "myqueue",  
      "connection": "AzureWebJobsStorage",  
      "direction": "in"  
    },  
    {  
      "type": "sendGrid",  
      "name": "$return",  
      "direction": "out",  
      "apiKey": "SendGridAPIKeyAsAppSetting",  
      "from": "{FromEmail}",  
      "to": "{ToEmail}"  
    }  
  ]  
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
#r "SendGrid"  
  
using System;  
using SendGrid.Helpers.Mail;  
using Microsoft.Azure.WebJobs.Host;  
  
public static SendGridMessage Run(Message mymsg, ILogger log)  
{  
    SendGridMessage message = new SendGridMessage()  
    {  
        Subject = $"{mymsg.Subject}"  
    };  
  
    message.AddContent("text/plain", $"{mymsg.Content}");  
  
    return message;  
}  
public class Message  
{  
    public string ToEmail { get; set; }  
    public string FromEmail { get; set; }  
    public string Subject { get; set; }  
    public string Content { get; set; }  
}
```

Java example

The following example uses the `@SendGridOutput` annotation from the [Java functions runtime library](#) to send an email using the SendGrid output binding.

```

@FunctionName("SendEmail")
public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET, HttpMethod.POST}, authLevel =
AuthorizationLevel.FUNCTION) HttpRequestMessage<Optional<String>> request,
    @SendGridOutput(
        name = "email", dataType = "String", apiKey = "SendGridConnection", to = "test@example.com",
from = "test@example.com",
        subject= "Sending with SendGrid", text = "Hello from Azure Functions"
    ) OutputBinding<String> email
)
{
    String name = request.getBody().orElse("World");

    final String emailBody = "{\"personalizations\"::" +
        "[{\"to\":[{\"email\":\"test@example.com\"}],\" +
        "\"subject\":\"Sending with SendGrid\"}],\" +
        \"from\":{\"email\":\"test@example.com\"},\" +
        \"content\":[{\"type\":\"text/plain\",\"value\": \"Hello\" + name +
    "\"}]}";
    email.setValue(emailBody);
    return request.createResponseBuilder(HttpStatus.OK).body("Hello, " + name).build();
}

```

JavaScript example

The following example shows a SendGrid output binding in a `function.json` file and a [JavaScript function](#) that uses the binding.

Here's the binding data in the `function.json` file:

```
{
  "bindings": [
    {
      "name": "$return",
      "type": "sendGrid",
      "direction": "out",
      "apiKey" : "MySendGridKey",
      "to": "{ToEmail}",
      "from": "{FromEmail}",
      "subject": "SendGrid output bindings"
    }
  ]
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```

module.exports = function (context, input) {
    var message = {
        "personalizations": [ { "to": [ { "email": "sample@sample.com" } ] } ],
        "from": { email: "sender@contoso.com" },
        "subject": "Azure news",
        "content": [
            {
                type: 'text/plain',
                value: input
            }
        ]
    };
    context.done(null, message);
};

```

Attributes

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 explains the binding configuration properties that you set in the `function.json` file and the `SendGrid` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
type		Required - must be set to <code>SendGrid</code> .
direction		Required - must be set to <code>out</code> .
name		Required - 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.
apiKey	ApiKey	The name of an app setting that contains your API key. If not set, the default app setting name is "AzureWebJobsSendGridApiKey".
to	To	the recipient's email address.
from	From	the sender's email address.
subject	Subject	the subject of the email.
text	Text	the email content.

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 version 2.x. The example `host.json` file below contains only the version 2.x settings for this binding. For more information about global configuration settings in version 2.x, see [host.json reference for Azure Functions version 2.x](#).

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

7/16/2019 • 17 minutes to read • [Edit Online](#)

This article explains how to work with Azure Service Bus bindings in Azure Functions. Azure Functions supports trigger and output bindings for Service Bus queues and topics.

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 Service Bus bindings are provided in the [Microsoft.Azure.WebJobs.ServiceBus](#) NuGet package, version 2.x.

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

The Service Bus bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.ServiceBus](#) NuGet package, version 3.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 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Trigger

Use the Service Bus trigger to respond to messages from a Service Bus queue or topic.

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)

Trigger - C# example

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", AccessRights.Manage, 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}");
}
```

This example is for Azure Functions version 1.x. To make this code work for 2.x:

- [omit the access rights parameter](#)
- change the type of the log parameter from `TraceWriter` to `ILogger`
- change `log.Info` to `log.LogInformation`

Trigger - C# script example

The following example shows a Service Bus trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function reads [message metadata](#) and logs a Service Bus queue message.

Here's the binding data in the `function.json` file:

```
{
  "bindings": [
    {
      "queueName": "testqueue",
      "connection": "MyServiceBusConnection",
      "name": "myQueueItem",
      "type": "serviceBusTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```

using System;

public static void Run(string myQueueItem,
    Int32 deliveryCount,
    DateTime enqueuedTimeUtc,
    string messageId,
    TraceWriter log)
{
    log.Info($"C# ServiceBus queue trigger function processed message: {myQueueItem}");

    log.Info($"EnqueuedTimeUtc={enqueuedTimeUtc}");
    log.Info($"DeliveryCount={deliveryCount}");
    log.Info($"MessageId={messageId}");
}

```

Trigger - F# example

The following example shows a Service Bus trigger binding in a `function.json` file and an [F# function](#) that uses the binding. The function logs a Service Bus queue message.

Here's the binding data in the `function.json` file:

```
{
  "bindings": [
    {
      "queueName": "testqueue",
      "connection": "MyServiceBusConnection",
      "name": "myQueueItem",
      "type": "serviceBusTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here's the F# script code:

```
let Run(myQueueItem: string, log: ILogger) =
    log.LogInformation(sprintf "F# ServiceBus queue trigger function processed message: %s" myQueueItem)
```

Trigger - Java example

The following Java function uses the `@ServiceBusQueueTrigger` annotation from the [Java functions runtime library](#) to describe the configuration for a Service Bus queue trigger. The function grabs the message placed on the queue and adds it to the logs.

```

@FunctionName("sbprocessor")
public void serviceBusProcess(
    @ServiceBusQueueTrigger(name = "msg",
        queueName = "myqueuename",
        connection = "myconnvarname") String message,
    final ExecutionContext context
) {
    context.getLogger().info(message);
}
```

Java functions can also be triggered when a message is added to a Service Bus topic. The following example uses the `@ServiceBusTopicTrigger` annotation to describe the trigger configuration.

```

@FunctionName("sbtopicprocessor")
public void run(
    @ServiceBusTopicTrigger(
        name = "message",
        topicName = "mytopicname",
        subscriptionName = "mysubscription",
        connection = "ServiceBusConnection"
    ) String message,
    final ExecutionContext context
) {
    context.getLogger().info(message);
}

```

Trigger - JavaScript example

The following example shows a Service Bus trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function reads [message metadata](#) and logs a Service Bus queue message.

Here's the binding data in the `function.json` file:

```
{
"bindings": [
    {
        "queueName": "testqueue",
        "connection": "MyServiceBusConnection",
        "name": "myQueueItem",
        "type": "serviceBusTrigger",
        "direction": "in"
    }
],
"disabled": false
}
```

Here's the JavaScript script code:

```

module.exports = function(context, myQueueItem) {
    context.log('Node.js ServiceBus queue trigger function processed message', myQueueItem);
    context.log('EnqueuedTimeUtc =', context.bindingData.enqueuedTimeUtc);
    context.log('DeliveryCount =', context.bindingData.deliveryCount);
    context.log('MessageId =', context.bindingData.messageId);
    context.done();
};

```

Trigger - attributes

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)
{
    ...
}
```

You can set the `Connection` property to specify the Service Bus account 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 - C# 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.

Trigger - 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. Set to "\$return" to reference the function return value.
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
<code>accessRights</code>	<code>Access</code>	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, this property is not available because the latest version of the Storage SDK doesn't support manage operations.

When you're developing locally, app settings go into the [local.settings.json file](#).

Trigger - usage

In C# and C# script, you can use the following parameter types 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.

These parameters are for Azure Functions version 1.x; for 2.x, use `Message` instead of `BrokeredMessage`.

In JavaScript, access the queue or topic message by using `context.bindings.<name from function.json>`. The Service Bus message is passed into the function as either a string or JSON object.

Trigger - poison messages

Poison message handling can't be controlled or configured in Azure Functions. Service Bus handles poison messages itself.

Trigger - 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.

Trigger - 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 are properties of the `BrokeredMessage` class.

PROPERTY	TYPE	DESCRIPTION
DeliveryCount	Int32	The number of deliveries.
DeadLetterSource	string	The dead letter source.
ExpiresAtUtc	DateTime	The expiration time in UTC.
EnqueuedTimeUtc	DateTime	The enqueued time in UTC.
MessageId	string	A user-defined value that Service Bus can use to identify duplicate messages, if enabled.
ContentType	string	A content type identifier utilized by the sender and receiver for application specific logic.
ReplyTo	string	The reply to queue address.
SequenceNumber	Int64	The unique number assigned to a message by the Service Bus.
To	string	The send to address.
Label	string	The application specific label.
CorrelationId	string	The correlation ID.

NOTE

Currently, Service bus trigger that works with session enabled queues and subscriptions is in preview. Please track [this item](#) for any further updates regarding this.

See [code examples](#) that use these properties earlier in this article.

Trigger - host.json properties

The `host.json` file contains settings that control Service Bus trigger behavior.

```
{
  "serviceBus": {
    "maxConcurrentCalls": 16,
    "prefetchCount": 100,
    "maxAutoRenewDuration": "00:05:00"
  }
}
```

PROPERTY	DEFAULT	DESCRIPTION

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.
maxAutoRenewDuration	00:05:00	The maximum duration within which the message lock will be renewed automatically.

Output

Use Azure Service Bus output binding to send queue or topic messages.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)

Output - C# example

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;
}
```

Output - C# script example

The following example shows a Service Bus output binding in a *function.json* file and a [C# script function](#) that uses the binding. The function uses a timer trigger to send a queue message every 15 seconds.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "schedule": "0/15 * * * *",
      "name": "myTimer",
      "runsOnStartup": true,
      "type": "timerTrigger",
      "direction": "in"
    },
    {
      "name": "outputSbQueue",
      "type": "serviceBus",
      "queueName": "testqueue",
      "connection": "MyServiceBusConnection",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's C# script code that creates a single message:

```
public static void Run(TimerInfo myTimer, ILogger log, out string outputSbQueue)
{
    string message = $"Service Bus queue message created at: {DateTime.Now}";
    log.LogInformation(message);
    outputSbQueue = message;
}
```

Here's C# script code that creates multiple messages:

```
public static void Run(TimerInfo myTimer, ILogger log, ICollector<string> outputSbQueue)
{
    string message = $"Service Bus queue messages created at: {DateTime.Now}";
    log.LogInformation(message);
    outputSbQueue.Add("1 " + message);
    outputSbQueue.Add("2 " + message);
}
```

Output - F# example

The following example shows a Service Bus output binding in a *function.json* file and an [F# script function](#) that uses the binding. The function uses a timer trigger to send a queue message every 15 seconds.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "schedule": "0/15 * * * *",
      "name": "myTimer",
      "runsOnStartup": true,
      "type": "timerTrigger",
      "direction": "in"
    },
    {
      "name": "outputSbQueue",
      "type": "serviceBus",
      "queueName": "testqueue",
      "connection": "MyServiceBusConnection",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's F# script code that creates a single message:

```
let Run(myTimer: TimerInfo, log: ILogger, outputSbQueue: byref<string>) =
    let message = sprintf "Service Bus queue message created at: %s" (DateTime.Now.ToString())
    log.LogInformation(message)
    outputSbQueue = message
```

Output - Java example

The following example shows a Java function that sends a message to a Service Bus queue `myqueue` when triggered by a HTTP request.

```
@FunctionName("httpToServiceBusQueue")
@ServiceBusQueueOutput(name = "message", queueName = "myqueue", connection = "AzureServiceBusConnection")
public String pushToQueue(
    @HttpTrigger(name = "request", methods = {HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
    final String message,
    @HttpOutput(name = "response") final OutputBinding<T> result ) {
    result.setValue(message + " has been sent.");
    return message;
}
```

In the [Java functions runtime library](#), use the `@QueueOutput` annotation on function parameters whose value would be written to a Service Bus queue. The parameter type should be `OutputBinding<T>`, where T is any native Java type of a POJO.

Java functions can also write to a Service Bus topic. The following example uses the `@ServiceBusTopicOutput` annotation to describe the configuration for the output binding.

```

@FunctionName("sbtopicsend")
public HttpResponseMessage run(
    @HttpTrigger(name = "req", methods = {HttpMethod.GET, HttpMethod.POST}, authLevel =
AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> request,
    @ServiceBusTopicOutput(name = "message", topicName = "mytopicname", subscriptionName =
"mysubscription", connection = "ServiceBusConnection") OutputBinding<String> message,
    final ExecutionContext context) {

    String name = request.getBody().orElse("Azure Functions");

    message.setValue(name);
    return request.createResponseBuilder(HttpStatus.OK).body("Hello, " + name).build();

}

```

Output - JavaScript example

The following example shows a Service Bus output binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function uses a timer trigger to send a queue message every 15 seconds.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "schedule": "0/15 * * * *",
      "name": "myTimer",
      "runsOnStartup": true,
      "type": "timerTrigger",
      "direction": "in"
    },
    {
      "name": "outputSbQueue",
      "type": "serviceBus",
      "queueName": "testqueue",
      "connection": "MyServiceBusConnection",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's JavaScript script code that creates a single message:

```

module.exports = function (context, myTimer) {
    var message = 'Service Bus queue message created at ' + timeStamp;
    context.log(message);
    context.bindings.outputSbQueue = message;
    context.done();
};

```

Here's JavaScript script code that creates multiple messages:

```

module.exports = function (context, myTimer) {
    var message = 'Service Bus queue message created at ' + timeStamp;
    context.log(message);
    context.bindings.outputSbQueue = [];
    context.bindings.outputSbQueue.push("1 " + message);
    context.bindings.outputSbQueue.push("2 " + message);
    context.done();
};

```

Output - attributes

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 Service Bus account 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 - C# 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](#).

Output - 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
type	n/a	Must be set to "serviceBus". This property is set automatically when you create the trigger in the Azure portal.
direction	n/a	Must be set to "out". 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 in function code. Set to "\$return" to reference the function return value.
queueName	QueueName	Name of the queue. Set only if sending queue messages, not for a topic.
topicName	TopicName	Name of the topic to monitor. Set only if sending topic messages, not for a queue.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
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>
accessRights	Access	<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, this property is not available because the latest version of the Storage SDK doesn't support manage operations.</p>

When you're developing locally, app settings go into the [local.settings.json file](#).

Output - 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, the queue or topic must already exist; if you specify a queue or topic that doesn't exist, the function will fail.

In C# and C# script, you can 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.
- `ICollector<T>` or `IAsyncCollector<T>` - For creating multiple messages. A message is created when you call the `Add` method.

In async functions, use the return value or `IAsyncCollector` instead of an `out` parameter.

These parameters are for Azure Functions version 1.x; for 2.x, use `Message` instead of `BrokeredMessage`.

In JavaScript, access the queue or topic by using `context.bindings.<name from function.json>`. You can assign a string, a byte array, or a Javascript object (deserialized into JSON) to `context.binding.<name>`.

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 version 2.x. The example host.json file below contains only the version 2.x settings for this binding. For more information about global configuration settings in version 2.x, see [host.json reference for Azure Functions version 2.x](#).

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": false,
        "maxConcurrentCalls": 32,
        "maxAutoRenewDuration": "00:55:00"
      }
    }
  }
}
```

PROPERTY	DEFAULT	DESCRIPTION
maxAutoRenewDuration	00:05:00	The maximum duration within which the message lock will be renewed automatically.
autoComplete	true	Whether the trigger should immediately mark as complete (autocomplete) or wait for processing to call complete.

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.

Next steps

[Learn more about Azure functions triggers and bindings](#)

SignalR Service bindings for Azure Functions

7/1/2019 • 12 minutes to read • [Edit Online](#)

This article 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.

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

The SignalR Service bindings are provided in the [Microsoft.Azure.WebJobs.Extensions.SignalRService](#) NuGet package, version 1.*. Source code for the package is in the [azure-functions-signalrservice-extension](#) GitHub repository.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#	Register the extension
Portal development	Register the extension

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Java annotations

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.

```
<dependency>
  <groupId>com.microsoft.azure.functions</groupId>
  <artifactId>azure-functions-java-library-signalr</artifactId>
  <version>1.0.0</version>
</dependency>
```

NOTE

To use the SignalR Service bindings in Java, make sure you are using version 2.4.419 or higher of the Azure Functions Core Tools (host version 2.0.12332).

Using SignalR Service with Azure Functions

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](#).

SignalR connection info input binding

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.

See the language-specific example:

- [2.x C#](#)
- [2.x JavaScript](#)
- [2.x Java](#)

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.

2.x C# input examples

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. 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;
}
```

2.x JavaScript input examples

The following example shows a SignalR connection info input binding in a `function.json` file and a [JavaScript function](#) that uses the binding to return the connection information.

Here's binding data in the `function.json` file:

Example `function.json`:

```
{  
    "type": "signalRConnectionInfo",  
    "name": "connectionInfo",  
    "hubName": "chat",  
    "connectionStringSetting": "<name of setting containing SignalR Service connection string>",  
    "direction": "in"  
}
```

Here's the JavaScript code:

```
module.exports = async function (context, req, connectionInfo) {  
    context.res.body = 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. You can set the `userId` property of the binding to the value from either header using a [binding expression](#):

```
{headers.x-ms-client-principal-id} or {headers.x-ms-client-principal-name}.
```

Example `function.json`:

```
{  
    "type": "signalRConnectionInfo",  
    "name": "connectionInfo",  
    "hubName": "chat",  
    "userId": "{headers.x-ms-client-principal-id}",  
    "connectionStringSetting": "<name of setting containing SignalR Service connection string>",  
    "direction": "in"  
}
```

Here's the JavaScript code:

```
module.exports = async function (context, req, connectionInfo) {  
    // connectionInfo contains an access key token with a name identifier  
    // claim set to the authenticated user  
    context.res.body = connectionInfo;  
};
```

2.x Java input examples

The following example shows a [Java function](#) that acquires SignalR connection information using the input binding and returns it over HTTP.

```

@FunctionName("negotiate")
public SignalRConnectionInfo negotiate(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> req,
    @SignalRConnectionInfoInput(
        name = "connectionInfo",
        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. 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 SignalRConnectionInfo negotiate(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> req,
    @SignalRConnectionInfoInput(
        name = "connectionInfo",
        hubName = "chat",
        userId = "{headers.x-ms-client-principal-id}") SignalRConnectionInfo connectionInfo) {
    return connectionInfo;
}

```

SignalR output binding

Use the *SignalR* output binding to send one or more messages using Azure SignalR Service. You can broadcast a message to all connected clients, or you can broadcast it only to connected clients that have been authenticated to a given user.

You can also use it to manage the groups that a user belongs to.

See the language-specific example:

- [2.x C#](#)
- [2.x JavaScript](#)
- [2.x Java](#)

2.x C# send message output examples

Broadcast to all clients

The following example shows a [C# 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.

```
[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 `UserId` property of the SignalR message.

```
[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 `GroupName` property of the SignalR message.

```
[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 }
        });
}
```

2.x C# group management output examples

SignalR Service allows users to be added to groups. Messages can then be sent to a group. You can use the `SignalRGroupAction` class with the `signalR` output binding to manage a user's group membership.

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,
    string userId,
    [SignalR(HubName = "chat")]
    IAsyncCollector<SignalRGroupAction> signalRGroupActions)
{
    return signalRGroupActions.AddAsync(
        new SignalRGroupAction
        {
            UserId = userId,
            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,
    string userId,
    [SignalR(HubName = "chat")]
    IAsyncCollector<SignalRGroupAction> signalRGroupActions)
{
    return signalRGroupActions.AddAsync(
        new SignalRGroupAction
        {
            UserId = userId,
            GroupName = "myGroup",
            Action = GroupAction.Remove
        });
}
```

2.x JavaScript send message output examples

Broadcast to all clients

The following example shows a SignalR output binding in a *function.json* file and a [JavaScript function](#) that uses the binding to send a message with Azure SignalR Service. Set the output binding to an array of one or more SignalR messages. A SignalR message consists of a `target` property that specifies the name of the method to invoke on each client, and an `arguments` property that is an array of objects to pass to the client method as arguments.

Here's binding data in the *function.json* file:

Example *function.json*:

```
{
    "type": "signalR",
    "name": "signalRMessages",
    "hubName": "<hub_name>",
    "connectionStringSetting": "<name of setting containing SignalR Service connection string>",
    "direction": "out"
}
```

Here's the JavaScript code:

```
module.exports = async function (context, req) {
    context.bindings.signalRMessages = [
        {
            "target": "newMessage",
            "arguments": [ req.body ]
        }
    ];
};
```

Send to a user

You can send a message only to connections that have been authenticated to a user by setting the `userId` property of the SignalR message.

function.json stays the same. Here's the JavaScript code:

```
module.exports = async function (context, req) {
    context.bindings.signalRMessages = [
        // message will only be sent to this user ID
        {
            "userId": "userId1",
            "target": "newMessage",
            "arguments": [ req.body ]
        }
    ];
};
```

Send to a group

You can send a message only to connections that have been added to a group by setting the `groupName` property of the SignalR message.

function.json stays the same. Here's the JavaScript code:

```
module.exports = async function (context, req) {
    context.bindings.signalRMessages = [
        // message will only be sent to this group
        {
            "groupName": "myGroup",
            "target": "newMessage",
            "arguments": [ req.body ]
        }
    ];
};
```

2.x JavaScript group management output examples

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.

Add user to a group

The following example adds a user to a group.

function.json

```
{  
  "disabled": false,  
  "bindings": [  
    {  
      "authLevel": "anonymous",  
      "type": "httpTrigger",  
      "direction": "in",  
      "name": "req",  
      "methods": [  
        "post"  
      ]  
    },  
    {  
      "type": "http",  
      "direction": "out",  
      "name": "res"  
    },  
    {  
      "type": "signalR",  
      "name": "signalRGroupActions",  
      "connectionStringSetting": "<name of setting containing SignalR Service connection string>",  
      "hubName": "chat",  
      "direction": "out"  
    }  
  ]  
}
```

index.js

```
module.exports = async function (context, req) {  
  context.bindings.signalRGroupActions = [{  
    "userId": req.query.userId,  
    "groupName": "myGroup",  
    "action": "add"  
  }];  
};
```

Remove user from a group

The following example removes a user from a group.

function.json

```
{
  "disabled": false,
  "bindings": [
    {
      "authLevel": "anonymous",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    },
    {
      "type": "signalR",
      "name": "signalRGroupActions",
      "connectionStringSetting": "<name of setting containing SignalR Service connection string>",
      "hubName": "chat",
      "direction": "out"
    }
  ]
}
```

index.js

```
module.exports = async function (context, req) {
  context.bindings.signalRGroupActions = [
    {
      "userId": req.query.userId,
      "groupName": "myGroup",
      "action": "remove"
    }
  ];
};
```

2.x Java send message output examples

Broadcast to all clients

The following example shows a [Java 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.

```
@FunctionName("sendMessage")
@SignalROutput(name = "$return", hubName = "chat")
public SignalRMessage sendMessage(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Object> req) {

    SignalRMessage message = new SignalRMessage();
    message.target = "newMessage";
    message.arguments.add(req.getBody());
    return message;
}
```

Send to a user

You can send a message only to connections that have been authenticated to a user by setting the `userId` property of the SignalR message.

```

@FunctionName("sendMessage")
@SignalROutput(name = "$return", hubName = "chat")
public SignalRMessage sendMessage(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Object> req) {

    SignalRMessage message = new SignalRMessage();
    message.userId = "userId1";
    message.target = "newMessage";
    message.arguments.add(req.getBody());
    return message;
}

```

Send to a group

You can send a message only to connections that have been added to a group by setting the `groupName` property of the SignalR message.

```

@FunctionName("sendMessage")
@SignalROutput(name = "$return", hubName = "chat")
public SignalRMessage sendMessage(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Object> req) {

    SignalRMessage message = new SignalRMessage();
    message.groupName = "myGroup";
    message.target = "newMessage";
    message.arguments.add(req.getBody());
    return message;
}

```

2.x Java group management output examples

SignalR Service allows users to be added to groups. Messages can then be sent to a group. You can use the `SignalRGroupAction` class with the `signalROutput` output binding to manage a user's group membership.

Add user to a group

The following example adds a user to a group.

```

@FunctionName("addToGroup")
@SignalROutput(name = "$return", hubName = "chat")
public SignalRGroupAction addToGroup(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Object> req,
    @BindingName("userId") String userId) {

    SignalRGroupAction groupAction = new SignalRGroupAction();
    groupAction.action = "add";
    groupAction.userId = userId;
    groupAction.groupName = "myGroup";
    return action;
}

```

Remove user from a group

The following example removes a user from a group.

```

@FunctionName("removeFromGroup")
@SignalROutput(name = "$return", hubName = "chat")
public SignalRGroupAction removeFromGroup(
    @HttpTrigger(
        name = "req",
        methods = { HttpMethod.POST },
        authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Object> req,
    @BindingName("userId") String userId) {

    SignalRGroupAction groupAction = new SignalRGroupAction();
    groupAction.action = "remove";
    groupAction.userId = userId;
    groupAction.groupName = "myGroup";
    return action;
}

```

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
type		Must be set to <code>signalRConnectionInfo</code> .
direction		Must be set to <code>in</code> .
name		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.
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		Must be set to <code>signalR</code> .
direction		Must be set to <code>out</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
name		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

[Learn more about Azure functions triggers and bindings](#)

[Azure Functions development and configuration with Azure SignalR Service](#)

Azure Table storage bindings for Azure Functions

7/16/2019 • 17 minutes to read • [Edit Online](#)

This article explains how to work with Azure Table storage bindings in Azure Functions. Azure Functions supports input and output bindings for Azure Table storage.

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 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](#).

Packages - Functions 2.x

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.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Input

Use the Azure Table storage input binding to read a table in an Azure Storage account.

Input - example

See the language-specific example:

- [C# read one entity](#)
- [C# bind to IQueryable](#)
- [C# bind to CloudTable](#)
- [C# script read one entity](#)
- [C# script bind to IQueryable](#)
- [C# script bind to CloudTable](#)
- [F#](#)
- [JavaScript](#)
- [Java](#)

Input - C# example - one entity

The following example shows a [C# function](#) that reads a single table row.

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}");
    }
}
```

Input - C# example - IQueryable

The following example shows a [C# function](#) that reads multiple table rows. Note that 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")] IQueryable<MyPoco> pocos,
        ILogger log)
    {
        foreach (MyPoco poco in pocos)
        {
            log.LogInformation($"PK={poco.PartitionKey}, RK={poco.RowKey}, Text={poco.Text}");
        }
    }
}
```

Input - C# example - CloudTable

`IQueryable` isn't supported in the [Functions v2 runtime](#). An alternative is to use a `CloudTable` method parameter to read the table by using the Azure Storage SDK. Here's an example of a 2.x function that queries an Azure Functions log table:

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.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](#).

Input - C# script example - one entity

The following example shows a table input binding in a `function.json` file and C# script code that uses the binding. The function uses a queue trigger to read a single table row.

The `function.json` file specifies a `partitionKey` and a `rowKey`. The `rowKey` value "{queueTrigger}" indicates that the row key comes from the queue message string.

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "personEntity",
      "type": "table",
      "tableName": "Person",
      "partitionKey": "Test",
      "rowKey": "{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
public static void Run(string myQueueItem, Person personEntity, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
    log.LogInformation($"Name in Person entity: {personEntity.Name}");
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
}
```

Input - C# script example - IQuerybable

The following example shows a table input binding in a *function.json* file and C# script code that uses the binding. The function reads entities for a partition key that is specified in a queue message.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "tableBinding",
      "type": "table",
      "connection": "MyStorageConnectionAppSetting",
      "tableName": "Person",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

The C# script code adds a reference to the Azure Storage SDK so that the entity type can derive from

`TableEntity`:

```
#r "Microsoft.WindowsAzure.Storage"
using Microsoft.WindowsAzure.Storage.Table;
using Microsoft.Extensions.Logging;

public static void Run(string myQueueItem, IQueryable<Person> tableBinding, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
    foreach (Person person in tableBinding.Where(p => p.PartitionKey == myQueueItem).ToList())
    {
        log.LogInformation($"Name: {person.Name}");
    }
}

public class Person : TableEntity
{
    public string Name { get; set; }
}
```

Input - C# script example - CloudTable

`IQueryable` isn't supported in the [Functions v2 runtime](#). An alternative is to use a `CloudTable` method parameter to read the table by using the Azure Storage SDK. Here's an example of a 2.x function that queries an Azure Functions log table:

```
{
  "bindings": [
    {
      "name": "myTimer",
      "type": "timerTrigger",
      "direction": "in",
      "schedule": "0 */1 * * *"
    },
    {
      "name": "cloudTable",
      "type": "table",
      "connection": "AzureWebJobsStorage",
      "tableName": "AzureWebJobsHostLogscommon",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

```
#r "Microsoft.WindowsAzure.Storage"
using Microsoft.WindowsAzure.Storage.Table;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

public static async Task Run(TimerInfo myTimer, 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,
                "a")));

    // 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}");
    }
}

public class LogEntity : TableEntity
{
    public string OriginalName { get; set; }
}
```

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](#).

Input - F# example

The following example shows a table input binding in a `function.json` file and [F# script](#) code that uses the binding. The function uses a queue trigger to read a single table row.

The `function.json` file specifies a `partitionKey` and a `rowKey`. The `rowKey` value "`{queueTrigger}`" indicates that the row key comes from the queue message string.

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "personEntity",
      "type": "table",
      "tableName": "Person",
      "partitionKey": "Test",
      "rowKey": "{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```
[<CLIMutable>]
type Person = {
  PartitionKey: string
  RowKey: string
  Name: string
}

let Run(myQueueItem: string, personEntity: Person) =
  log.LogInformation(sprintf "F# Queue trigger function processed: %s" myQueueItem)
  log.LogInformation(sprintf "Name in Person entity: %s" personEntity.Name)
```

Input - JavaScript example

The following example shows a table input binding in a *function.json* file and [JavaScript code](#) that uses the binding. The function uses a queue trigger to read a single table row.

The *function.json* file specifies a `partitionKey` and a `rowKey`. The `rowKey` value "`{queueTrigger}`" indicates that the row key comes from the queue message string.

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "personEntity",
      "type": "table",
      "tableName": "Person",
      "partitionKey": "Test",
      "rowKey": "{queueTrigger}",
      "connection": "MyStorageConnectionAppSetting",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function (context, myQueueItem) {
  context.log('Node.js queue trigger function processed work item', myQueueItem);
  context.log('Person entity name: ' + context.bindings.personEntity.Name);
  context.done();
};
```

Input - Java example

The following example shows an HTTP triggered function which returns the total count of the items in a specified partition in Table storage.

```
@FunctionName("getallcount")
public int run(
    @HttpTrigger(name = "req",
        methods = {HttpMethod.GET},
        authLevel = AuthorizationLevel.ANONYMOUS) Object dummyShouldNotBeUsed,
    @TableInput(name = "items",
        tableName = "mytablename", partitionKey = "myparkey",
        connection = "myconnvarname") MyItem[] items
) {
    return items.length;
}
```

Input - attributes

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. It 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 [Input - 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).

Input - Java annotations

In the [Java functions runtime library](#), use the `@TableInput` annotation on parameters whose value would come from Table storage. This annotation can be used with native Java types, POJOs, or nullable values using `Optional<T>`.

Input - 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
type	n/a	Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal.
direction	n/a	Must be set to <code>in</code> . This property is set automatically when you create the binding in the Azure portal.
name	n/a	The name of the variable that represents the table or entity in function code.
tableName	TableName	The name of the table.
partitionKey	PartitionKey	Optional. The partition key of the table entity to read. See the usage section for guidance on how to use this property.
rowKey	RowKey	Optional. The row key of the table entity to read. See the usage section for guidance on how to use this property.
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. 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> .

When you're developing locally, app settings go into the `local.settings.json` file.

Input - usage

The Table storage input binding supports the following scenarios:

- **Read one row in C# or C# script**

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 in C# or C# script**

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](#).

- **Read one or more rows in JavaScript**

Set the `filter` and `take` properties. Don't set `partitionKey` or `rowKey`. Access the input table entity (or entities) using `context.bindings.<name>`. The deserialized objects have `RowKey` and `PartitionKey` properties.

Output

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.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

Output - C# example

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 };
    }
}

```

Output - C# script example

The following example shows a table output binding in a *function.json* file and [C# script](#) code that uses the binding. The function writes multiple table entities.

Here's the *function.json* file:

```
{
    "bindings": [
        {
            "name": "input",
            "type": "manualTrigger",
            "direction": "in"
        },
        {
            "tableName": "Person",
            "connection": "MyStorageConnectionAppSetting",
            "name": "tableBinding",
            "type": "table",
            "direction": "out"
        }
    ],
    "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```

public static void Run(string input, ICollector<Person> tableBinding, ILogger log)
{
    for (int i = 1; i < 10; i++)
    {
        log.LogInformation($"Adding Person entity {i}");
        tableBinding.Add(
            new Person() {
                PartitionKey = "Test",
                RowKey = i.ToString(),
                Name = "Name" + i.ToString() }
        );
    }
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
}

```

Output - F# example

The following example shows a table output binding in a *function.json* file and [F# script](#) code that uses the binding. The function writes multiple table entities.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "name": "input",
      "type": "manualTrigger",
      "direction": "in"
    },
    {
      "tableName": "Person",
      "connection": "MyStorageConnectionAppSetting",
      "name": "tableBinding",
      "type": "table",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```
[<CLIMutable>]
type Person = {
    PartitionKey: string
    RowKey: string
    Name: string
}

let Run(input: string, tableBinding: ICollector<Person>, log: ILogger) =
    for i = 1 to 10 do
        log.LogInformation(sprintf "Adding Person entity %d" i)
        tableBinding.Add(
            { PartitionKey = "Test"
              RowKey = i.ToString()
              Name = "Name" + i.ToString() })
```

Output - JavaScript example

The following example shows a table output binding in a *function.json* file and a [JavaScript function](#) that uses the binding. The function writes multiple table entities.

Here's the *function.json* file:

```
{
    "bindings": [
        {
            "name": "input",
            "type": "manualTrigger",
            "direction": "in"
        },
        {
            "tableName": "Person",
            "connection": "MyStorageConnectionAppSetting",
            "name": "tableBinding",
            "type": "table",
            "direction": "out"
        }
    ],
    "disabled": false
}
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
module.exports = function (context) {

    context.bindings.tableBinding = [];

    for (var i = 1; i < 10; i++) {
        context.bindings.tableBinding.push({
            PartitionKey: "Test",
            RowKey: i.toString(),
            Name: "Name " + i
        });
    }

    context.done();
};
```

Output - attributes

In [C# class libraries](#), use the [TableAttribute](#).

The attribute's constructor takes the table name. It 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 [Output - 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](#).

Output - 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
type	n/a	Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal.
direction	n/a	Must be set to <code>out</code> . This property 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 table or entity. Set to <code>\$return</code> to reference the function return value.
tableName	TableName	The name of the table.
partitionKey	PartitionKey	The partition key of the table entity to write. See the usage section for guidance on how to use this property.
rowKey	RowKey	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 "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.

Output - usage

The Table storage output binding supports the following scenarios:

- **Write one row in any language**

In C# and C# script, access the output table entity by using a method parameter such as `out T paramName` or the function return value. In C# script, `paramName` is the value specified in the `name` property of `function.json`. `T` can be any serializable type if the partition key and row key are provided by the `function.json` file or the `Table` attribute. Otherwise, `T` must be a type that includes `PartitionKey` and `RowKey` properties. In this scenario, `T` typically implements `ITableEntity` or derives from `TableEntity`, but it doesn't have to.

- **Write one or more rows in C# or C# script**

In C# and C# script, access the output table entity by using a method parameter `ICollector<T> paramName` or `IAsyncCollector<T> paramName`. In C# script, `paramName` is the value specified in the `name` property of `function.json`. `T` specifies the schema of the entities you want to add. Typically, `T` derives from `TableEntity` or implements `ITableEntity`, but it doesn't have to. The partition key and row key values in `function.json` or the `Table` attribute constructor are not used in this scenario.

An alternative is to 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](#). For an example of code that binds to `CloudTable`, see the input binding examples for [C#](#) or [C# script](#) earlier in this article.

- **Write one or more rows in JavaScript**

In JavaScript functions, access the table output using `context.bindings.<name>`.

Exceptions and return codes

BINDING	REFERENCE
Table	Table Error Codes

BINDING	REFERENCE
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

7/22/2019 • 10 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](#).

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.

Packages - Functions 2.x

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.

Example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [Java](#)
- [JavaScript](#)
- [Python](#)

C# example

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}");
}
```

C# script example

The following example shows a timer trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function writes a log indicating whether this function invocation is due to a missed schedule occurrence. The `TimerInfo` object is passed into the function.

Here's the binding data in the `function.json` file:

```
{
    "schedule": "0 */5 * * *",
    "name": "myTimer",
    "type": "timerTrigger",
    "direction": "in"
}
```

Here's the C# script code:

```
public static void Run(TimerInfo myTimer, ILogger log)
{
    if (myTimer.IsPastDue)
    {
        log.LogInformation("Timer is running late!");
    }
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now} ");
}
```

F# example

The following example shows a timer trigger binding in a `function.json` file and an [F# script function](#) that uses the binding. The function writes a log indicating whether this function invocation is due to a missed schedule occurrence. The `TimerInfo` object is passed into the function.

Here's the binding data in the `function.json` file:

```
{
    "schedule": "0 */5 * * *",
    "name": "myTimer",
    "type": "timerTrigger",
    "direction": "in"
}
```

Here's the F# script code:

```
let Run(myTimer: TimerInfo, log: ILogger ) =
    if (myTimer.IsPastDue) then
        log.LogInformation("F# function is running late.")
    let now = DateTime.Now.ToString()
    log.LogInformation(sprintf "F# function executed at %s!" now)
```

Java example

The following example function triggers and executes every five minutes. The `@TimerTrigger` annotation on the function defines the schedule using the same string format as [CRON expressions](#).

```
@FunctionName("keepAlive")
public void keepAlive(
    @TimerTrigger(name = "keepAliveTrigger", schedule = "0 */5 * * *") String timerInfo,
    ExecutionContext context
) {
    // timeInfo is a JSON string, you can deserialize it to an object using your favorite JSON library
    context.getLogger().info("Timer is triggered: " + timerInfo);
}
```

JavaScript example

The following example shows a timer trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function writes a log indicating whether this function invocation is due to a missed schedule occurrence. A `timer object` is passed into the function.

Here's the binding data in the `function.json` file:

```
{
    "schedule": "0 */5 * * *",
    "name": "myTimer",
    "type": "timerTrigger",
    "direction": "in"
}
```

Here's the JavaScript code:

```
module.exports = function (context, myTimer) {
    var timeStamp = new Date().toISOString();

    if (myTimer.IsPastDue)
    {
        context.log('Node is running late!');
    }
    context.log('Node timer trigger function ran!', timeStamp);

    context.done();
};
```

Python example

The following example uses a timer trigger binding whose configuration is described in the `function.json` file. The actual [Python function](#) that uses the binding is described in the `init.py` file. The object passed into the function is of type [azure.functions.TimerRequest object](#). The function logic writes to the logs indicating whether the current invocation is due to a missed schedule occurrence.

Here's the binding data in the `function.json` file:

```
{
    "name": "mytimer",
    "type": "timerTrigger",
    "direction": "in",
    "schedule": "0 */5 * * *"
}
```

Here's the Python code:

```

import datetime
import logging

import azure.functions as func


def main(mytimer: func.TimerRequest) -> None:
    utc_timestamp = datetime.datetime.utcnow().replace(
        tzinfo=datetime.timezone.utc).isoformat()

    if mytimer.past_due:
        logging.info('The timer is past due!')

    logging.info('Python timer trigger function ran at %s', utc_timestamp)

```

Attributes

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. 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
type	n/a	Must be set to "timerTrigger". 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 timer object in function code.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
schedule	ScheduleExpression	A CRON expression or a <code>TimeSpan</code> 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 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.

CRON expressions

Azure Functions uses the [NCronTab](#) library to interpret CRON expressions. A CRON expression includes six fields:

```
{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	"0 5 * * *"	at hh:05:00 where hh is every hour (once an hour)
All values (<code>*</code>)	"0 * 5 * * *"	at 5:mm:00 every day, where mm is every minute of the hour (60 times a day)
A range (<code>-</code> operator)	"5-7 * * * *"	at hh:mm:05, hh:mm:06, and hh:mm:07 where hh:mm is every minute of every hour (3 times a minute)
A set of values (<code>,</code> operator)	"5,8,10 * * * *"	at hh:mm:05, hh:mm:08, and hh:mm:10 where hh:mm is every minute of every hour (3 times a minute)
An interval value (<code>/</code> operator)	"0 */5 * * *"	at hh:05:00, hh:10:00, hh:15:00, and so on through hh:55:00 where hh is every hour (12 times an hour)

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`.

CRON examples

Here are some examples of CRON expressions you can use for the timer trigger in Azure Functions.

EXAMPLE	WHEN TRIGGERED
"0 */5 * * *"	once every five minutes
"0 0 * * * *"	once at the top of every hour
"0 0 */2 * * *"	once every two hours
"0 0 9-17 * * *"	once every hour from 9 AM to 5 PM
"0 30 9 * * *"	at 9:30 AM every day

EXAMPLE	WHEN TRIGGERED
"0 30 9 * * 1-5"	at 9:30 AM every weekday
"0 30 9 * Jan Mon"	at 9:30 AM every Monday in January

NOTE

You can find CRON expression examples online, but many of them omit the `{second}` field. If you copy from one of them, add the missing `{second}` field. Usually you'll want a zero in that field, not an asterisk.

CRON 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`. Set the value to the name of the desired time zone as shown in the [Microsoft Time Zone Index](#).

For example, *Eastern Standard Time* is UTC-05:00. To have your timer trigger fire at 10:00 AM EST every day, use the following CRON expression that accounts for UTC time zone:

```
"schedule": "0 0 15 * * *"
```

Or create an app setting for your function app named `WEBSITE_TIME_ZONE` and set the value to **Eastern Standard Time**. Then uses the following CRON expression:

```
"schedule": "0 0 10 * * *"
```

When you use `WEBSITE_TIME_ZONE`, the time is adjusted for time changes in the specific timezone, such as daylight savings 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

EXAMPLE	WHEN TRIGGERED
"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 share a Storage account across multiple function apps, make sure that each function app has a different `id` in `host.json`. 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.

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

7/1/2019 • 9 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

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.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 2.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#, Java and Python	Register the extension
Portal development	Install when adding output binding

To learn how to update existing binding extensions in the portal without having to republish your function app project, see [Update your extensions](#).

Example - Functions 1.x

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

C# example

The following example shows a [C# function](#) that sends a text message when triggered by a queue message.

```
[FunctionName("QueueTwilio")]
[return: TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting = "TwilioAuthToken", From =
"+1425XXXXXXX")]
public static SMSMessage Run(
    [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] JObject order,
    TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {order}");

    var message = new SMSMessage()
    {
        Body = $"Hello {order["name"]}, thanks for your order!",
        To = order["mobileNumber"].ToString()
    };

    return message;
}
```

This example uses the `TwilioSms` attribute with the method return value. An alternative is to use the attribute with an `out SMSMessage` parameter or an `ICollector<SMSMessage>` or `IAsyncCollector<SMSMessage>` parameter.

C# script example

The following example shows a Twilio output binding in a `function.json` file and a [C# script function](#) that uses the binding. The function uses an `out` parameter to send a text message.

Here's binding data in the `function.json` file:

Example `function.json`:

```
{
    "type": "twilioSms",
    "name": "message",
    "accountSid": "TwilioAccountSid",
    "authToken": "TwilioAuthToken",
    "to": "+1704XXXXXXX",
    "from": "+1425XXXXXXX",
    "direction": "out",
    "body": "Azure Functions Testing"
}
```

Here's C# script code:

```

#r "Newtonsoft.Json"
#r "Twilio.Api"

using System;
using Newtonsoft.Json;
using Twilio;

public static void Run(string myQueueItem, out SMSMessage message, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");

    // In this example the queue item is a JSON string representing an order that contains the name of a
    // customer and a mobile number to send text updates to.
    dynamic order = JsonConvert.DeserializeObject(myQueueItem);
    string msg = "Hello " + order.name + ", thank you for your order./";

    // Even if you want to use a hard coded message and number in the binding, you must at least
    // initialize the SMSMessage variable.
    message = new SMSMessage();

    // A dynamic message can be set instead of the body in the output binding. In this example, we use
    // the order information to personalize a text message to the mobile number provided for
    // order status updates.
    message.Body = msg;
    message.To = order.mobileNumber;
}

```

You can't use out parameters in synchronous code. Here's an asynchronous C# script code example:

```

#r "Newtonsoft.Json"
#r "Twilio.Api"

using System;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Twilio;

public static async Task Run(string myQueueItem, IAsyncCollector<SMSMessage> message, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");

    // In this example the queue item is a JSON string representing an order that contains the name of a
    // customer and a mobile number to send text updates to.
    dynamic order = JsonConvert.DeserializeObject(myQueueItem);
    string msg = "Hello " + order.name + ", thank you for your order./";

    // Even if you want to use a hard coded message and number in the binding, you must at least
    // initialize the SMSMessage variable.
    SMSMessage smsText = new SMSMessage();

    // A dynamic message can be set instead of the body in the output binding. In this example, we use
    // the order information to personalize a text message to the mobile number provided for
    // order status updates.
    smsText.Body = msg;
    smsText.To = order.mobileNumber;

    await message.AddAsync(smsText);
}

```

JavaScript example

The following example shows a Twilio output binding in a *function.json* file and a [JavaScript function](#) that uses the binding.

Here's binding data in the *function.json* file:

Example function.json:

```
{  
  "type": "twilioSms",  
  "name": "message",  
  "accountSid": "TwilioAccountSid",  
  "authToken": "TwilioAuthToken",  
  "to": "+1704XXXXXXX",  
  "from": "+1425XXXXXXX",  
  "direction": "out",  
  "body": "Azure Functions Testing"  
}
```

Here's the JavaScript code:

```
module.exports = function (context, myQueueItem) {  
  context.log('Node.js queue trigger function processed work item', myQueueItem);  
  
  // In this example the queue item is a JSON string representing an order that contains the name of a  
  // customer and a mobile number to send text updates to.  
  var msg = "Hello " + myQueueItem.name + ", thank you for your order.";  
  
  // Even if you want to use a hard coded message and number in the binding, you must at least  
  // initialize the message binding.  
  context.bindings.message = {};  
  
  // A dynamic message can be set instead of the body in the output binding. In this example, we use  
  // the order information to personalize a text message to the mobile number provided for  
  // order status updates.  
  context.bindings.message = {  
    body : msg,  
    to : myQueueItem.mobileNumber  
  };  
  
  context.done();  
};
```

Example – Functions 2.x

See the language-specific example:

- [2.x C#](#)
- [2.x C# script \(.csx\)](#)
- [2.x JavaScript](#)

2.x C# example

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 = "+1425XXXXXX")]
        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.

2.x C# script example

The following example shows a Twilio output binding in a `function.json` file and a [C# script function](#) that uses the binding. The function uses an `out` parameter to send a text message.

Here's binding data in the `function.json` file:

Example `function.json`:

```
{
    "type": "twilioSms",
    "name": "message",
    "accountSidSetting": "TwilioAccountSid",
    "authTokenSetting": "TwilioAuthToken",
    "from": "+1425XXXXXX",
    "direction": "out",
    "body": "Azure Functions Testing"
}
```

Here's C# script code:

```

#r "Newtonsoft.Json"
#r "Twilio"
#r "Microsoft.Azure.WebJobs.Extensions.Twilio"

using System;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Microsoft.Azure.WebJobs.Extensions.Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

public static void Run(string myQueueItem, out CreateMessageOptions message, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");

    // In this example the queue item is a JSON string representing an order that contains the name of a
    // customer and a mobile number to send text updates to.
    dynamic order = JsonConvert.DeserializeObject(myQueueItem);
    string msg = "Hello " + order.name + ", thank you for your order./";

    // You must initialize the CreateMessageOptions variable with the "To" phone number.
    message = new CreateMessageOptions(new PhoneNumber("+1704XXXXXX"));

    // A dynamic message can be set instead of the body in the output binding. In this example, we use
    // the order information to personalize a text message.
    message.Body = msg;
}

```

You can't use out parameters in synchronous code. Here's an asynchronous C# script code example:

```

#r "Newtonsoft.Json"
#r "Twilio"
#r "Microsoft.Azure.WebJobs.Extensions.Twilio"

using System;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Microsoft.Azure.WebJobs.Extensions.Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

public static async Task Run(string myQueueItem, IAsyncCollector<CreateMessageOptions> message, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");

    // In this example the queue item is a JSON string representing an order that contains the name of a
    // customer and a mobile number to send text updates to.
    dynamic order = JsonConvert.DeserializeObject(myQueueItem);
    string msg = "Hello " + order.name + ", thank you for your order./";

    // You must initialize the CreateMessageOptions variable with the "To" phone number.
    CreateMessageOptions smsText = new CreateMessageOptions(new PhoneNumber("+1704XXXXXX"));

    // A dynamic message can be set instead of the body in the output binding. In this example, we use
    // the order information to personalize a text message.
    smsText.Body = msg;

    await message.AddAsync(smsText);
}

```

2.x JavaScript example

The following example shows a Twilio output binding in a *function.json* file and a [JavaScript function](#) that uses the binding.

Here's binding data in the `function.json` file:

Example `function.json`:

```
{  
  "type": "twilioSms",  
  "name": "message",  
  "accountSidSetting": "TwilioAccountSid",  
  "authTokenSetting": "TwilioAuthToken",  
  "from": "+1425XXXXXXX",  
  "direction": "out",  
  "body": "Azure Functions Testing"  
}
```

Here's the JavaScript code:

```
module.exports = function (context, myQueueItem) {  
    context.log('Node.js queue trigger function processed work item', myQueueItem);  
  
    // In this example the queue item is a JSON string representing an order that contains the name of a  
    // customer and a mobile number to send text updates to.  
    var msg = "Hello " + myQueueItem.name + ", thank you for your order.";  
  
    // Even if you want to use a hard coded message in the binding, you must at least  
    // initialize the message binding.  
    context.bindings.message = {};  
  
    // A dynamic message can be set instead of the body in the output binding. The "To" number  
    // must be specified in code.  
    context.bindings.message = {  
        body : msg,  
        to : myQueueItem.mobileNumber  
    };  
  
    context.done();  
};
```

Attributes

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 =  
"+1425XXXXXXX")]  
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
type	type	must be set to <code>twilioSms</code> .	
direction	direction	must be set to <code>out</code> .	
name	name	Variable name used in function code for the Twilio SMS text message.	
accountSid	accountSidSetting	AccountSidSetting	This value must be set to the name of an app setting that holds your Twilio Account Sid e.g. <code>TwilioAccountSid</code> . If not set, the default app setting name is <code>"AzureWebJobsTwilioAccountSid"</code> .
authToken	authTokenSetting	AuthTokenSetting	This value must be set to the name of an app setting that holds your Twilio authentication token e.g. <code>TwilioAccountAuthToken</code> . If not set, the default app setting name is <code>"AzureWebJobsTwilioAuthToKen"</code> .
to	N/A - specify in code	To	This value is set to the phone number that the SMS text is sent to.
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](#)

host.json reference for Azure Functions 2.x

7/26/2019 • 6 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 v2 runtime.

NOTE

This article is for Azure Functions 2.x. 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](#).

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.

```
{
    "version": "2.0",
    "aggregator": {
        "batchSize": 1000,
        "flushTimeout": "00:00:30"
    },
    "extensions": {
        "cosmosDb": {},
        "durableTask": {},
        "eventHubs": {},
        "http": {},
        "queues": {},
        "sendGrid": {},
        "serviceBus": {}
    },
    "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" : 5
            }
        }
    },
    "singleton": {
        "lockPeriod": "00:00:15",
        "listenerLockPeriod": "00:01:00",
        "listenerLockRecoveryPollingInterval": "00:01:00",
        "lockAcquisitionTimeout": "00:01:00",
        "lockAcquisitionPollingInterval": "00:00:03"
    },
    "watchDirectories": [ "Shared", "Test" ],
    "managedDependency": {
        "enabled": true
    }
}
```

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 the [sampling feature in Application Insights](#).

```
{
    "applicationInsights": {
        "samplingSettings": {
            "isEnabled": true,
            "maxTelemetryItemsPerSecond" : 5
        }
    }
}
```

NOTE

Log sampling may cause some executions to not show up in the Application Insights monitor blade.

PROPERTY	DEFAULT	DESCRIPTION
isEnabled	true	Enables or disables sampling.
maxTelemetryItemsPerSecond	5	The threshold at which sampling begins.

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](#).

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. In a serverless Consumption plan, the valid range is from 1 second to 10 minutes, and the default value is 5 minutes. In a Dedicated (App Service) plan, there is no overall limit, and the default depends on the runtime version:

- Version 1.x: the default is *null*, which indicates no timeout.
- Version 2.x: the default value is 30 minutes. A value of `-1` indicates unbounded execution.

```
{  
  "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.

PROPERTY	DEFAULT	DESCRIPTION
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](#).

```
{
  "http": {
    "routePrefix": "api",
    "maxOutstandingRequests": 200,
    "maxConcurrentRequests": 100,
    "dynamicThrottlesEnabled": true
  }
}
```

PROPERTY	DEFAULT	DESCRIPTION
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.
maxOutstandingRequests	200*	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows 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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 200. The default for version 2.x in a dedicated plan is unbounded (-1).

PROPERTY	DEFAULT	DESCRIPTION
maxConcurrentRequests	100*	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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 100. The default for version 2.x in a dedicated plan is unbounded (-1).
dynamicThrottlesEnabled	true*	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels. *The default for version 1.x is false. The default for version 2.x in a consumption plan is true. The default for version 2.x in a dedicated plan is false.

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
----------	---------	-------------

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. Version 2.x follows the ASP.NET Core layout for log category filtering. This lets you filter logging for specific functions. For more information, see Log filtering in the ASP.NET Core documentation.
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.

queues

Configuration settings can be found in [Storage queue triggers and bindings](#).

sendGrid

Configuration setting can be found in [SendGrid triggers and bindings](#).

serviceBus

Configuration setting can be found in [Service Bus triggers and bindings](#).

singleton

Configuration settings for Singleton lock behavior. For more information, see [GitHub issue about singleton support](#).

```
{
  "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.

version

The version string `"version": "2.0"` is required for a function app that targets the v2 runtime.

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" ]
}
```

managedDependency

Managed dependency is a preview 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 will be processed. Dependencies will be updated when any minor versions are released.

```
{
  "managedDependency": {
    "enabled": true
  }
}
```

Next steps

[Learn how to update the host.json file](#)

[See global settings in environment variables](#)

host.json reference for Azure Functions 1.x

3/25/2019 • 11 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, 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
    }
  },
  "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",
        "Request": "Information"
      }
    }
  }
}
```

```

        "Host.Aggregator": "Information"
    }
},
"queues": {
    "maxPollingInterval": 2000,
    "visibilityTimeout" : "00:00:30",
    "batchSize": 16,
    "maxDequeueCount": 5,
    "newBatchThreshold": 8
},
"serviceBus": {
    "maxConcurrentCalls": 16,
    "prefetchCount": 100,
    "autoRenewTimeout": "00:05:00"
},
"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.

durableTask

Configuration settings for [Durable Functions](#).

```
{
  "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.

PROPERTY	DEFAULT	DESCRIPTION
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.
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>azureStorageConnectionStringName</code> connection is used.
trackingStoreNamePrefix		The prefix to use for the History and Instances tables when <code>trackingStoreConnectionStringName</code> is specified. If not set, the default prefix value will be <code>DurableTask</code> . If <code>trackingStoreConnectionStringName</code> is not specified, then the History and Instances tables will use the <code>hubName</code> value as their prefix, and any setting for <code>trackingStoreNamePrefix</code> will be ignored.

PROPERTY	DEFAULT	DESCRIPTION
traceInputsAndOutputs	false	A value indicating whether to trace the inputs and outputs of function calls. The default behavior when tracing function execution events is to include the number of bytes in the serialized inputs and outputs for function calls. This behavior provides minimal information about what the inputs and outputs look like without bloating the logs or inadvertently exposing sensitive information. Setting this property to true causes the default function logging to log the entire contents of function inputs and outputs.
logReplayEvents	false	A value indicating whether to write orchestration replay events to Application Insights.
eventGridTopicEndpoint		The URL of an Azure Event Grid custom topic endpoint. When this property is set, orchestration life-cycle notification events are published to this endpoint. This property supports App Settings resolution.
eventGridKeySettingName		The name of the app setting containing the key used for authenticating with the Azure Event Grid custom topic at <code>EventGridTopicEndpoint</code> .
eventGridPublishRetryCount	0	The number of times to retry if publishing to the Event Grid Topic fails.
eventGridPublishRetryInterval	5 minutes	The Event Grid publishes retry interval in the <i>hh:mm:ss</i> format.
eventGridPublishEventTypes		A list of event types to publish to Event Grid. If not specified, all event types will be published. Allowed values include <code>Started</code> , <code>Completed</code> , <code>Failed</code> , <code>Terminated</code> .

Many of these settings are for optimizing performance. For more information, see [Performance and scale](#).

eventHub

Configuration settings for [Event Hub triggers and bindings](#).

```
{
  "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 PrefetchCount that will be used by the underlying EventProcessorHost.
batchCheckpointFrequency	1	The number of event batches to process before creating an EventHub cursor checkpoint.

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 depends on the runtime version.

```
{
  "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.

PROPERTY	DEFAULT	DESCRIPTION
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
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.
maxOutstandingRequests	200*	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows 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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 200. The default for version 2.x in a dedicated plan is unbounded (-1).

PROPERTY	DEFAULT	DESCRIPTION
maxConcurrentRequests	100*	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. *The default for version 1.x is unbounded (-1). The default for version 2.x in a consumption plan is 100. The default for version 2.x in a dedicated plan is unbounded (-1).
dynamicThrottlesEnabled	true*	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels. *The default for version 1.x is false. The default for version 2.x in a consumption plan is true. The default for version 2.x in a dedicated plan is false.

id

Version 1.x only.

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.

PROPERTY	DEFAULT	DESCRIPTION
batchSize	16	<p>The number of queue messages that the Functions runtime retrieves simultaneously and processes in parallel. When the number being processed gets down to the <code>newBatchThreshold</code>, the runtime gets another batch and starts processing those messages. So the maximum number of concurrent messages being processed per function is <code>batchSize</code> plus <code>newBatchThreshold</code>. This limit applies separately to each queue-triggered function.</p> <p>If you want to avoid parallel execution for messages received on one queue, you can set <code>batchsize</code> to 1. However, this setting eliminates concurrency only so long as your function app runs on a single virtual machine (VM). If the function app scales out to multiple VMs, each VM could run one instance of each queue-triggered function.</p> <p>The maximum <code>batchSize</code> is 32.</p>
maxDequeueCount	5	The number of times to try processing a message before moving it to the poison queue.
newBatchThreshold	<code>batchSize/2</code>	Whenever the number of messages being processed concurrently gets down to this number, the runtime retrieves another batch.

serviceBus

Configuration setting for Service Bus triggers and bindings.

```
{
  "serviceBus": {
    "maxConcurrentCalls": 16,
    "prefetchCount": 100,
    "autoRenewTimeout": "00:05:00"
  }
}
```

PROPERTY	DEFAULT	DESCRIPTION
----------	---------	-------------

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.

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. See [C# Logging](#) and [Node.js 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

7/9/2019 • 3 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 whitelisted. 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.

The only way to totally restrict a function such that all traffic flows through a virtual network is to use an internally load-balanced App Service Environment. This option deploys your site on a dedicated infrastructure inside a virtual network and sends all triggers and traffic through the virtual network.

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 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 (currently in preview), you can access service-endpoint-secured resources from a running function. For more information, see [Preview 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](#).

You can also trigger a function from a resource in a virtual network by deploying your function app to an App Service Environment. For details on using an App Service Environment, see [Create and use an internal load balancer with an App Service Environment](#).

The Premium and App Service plan support HTTP triggers from a virtual network, but only an App Service environment support all other function trigger types through a virtual network.

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)

7/8/2019 • 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 scale and hosting

7/24/2019 • 11 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 hosting plans available for Azure Functions: [Consumption plan](#), [Premium plan](#), and [App Service plan](#).

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 VNET 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 down 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 down to zero. (Requires that [Always on](#) is enabled.)

NOTE

You can switch between Consumption and Premium plans by changing the plan property of the function app resource.

Hosting plan support

Feature support falls into the following two categories:

- *Generally available (GA)*: fully supported and approved for production use.
- *Preview*: not yet fully supported and approved for production use.

The following table indicates the current level of support for the three hosting plans, when running on either Windows or Linux:

	CONSUMPTION PLAN	PREMIUM PLAN	DEDICATED PLAN
Windows	GA	preview	GA
Linux	preview	preview	GA

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. Billing 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.

Premium plan (preview)

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
- Premium instance sizes (one core, two core, and four core instances)
- More predictable pricing
- High-density app allocation for plans with multiple function apps

Information on how you can configure these options can be found in the [Azure Functions premium plan document](#).

Instead of billing per execution and memory consumed, billing for the Premium plan is based on the number of core seconds, execution time, and memory used across needed and reserved instances. At least one instance must be warm at all times. This means that there is a fixed monthly cost per active plan, regardless of the number of executions.

Consider the Azure Functions premium plan in the following situations:

- Your function apps run continuously, or nearly continuously.
- 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 VNET/VPN connectivity.

When running JavaScript functions on a Premium plan, you should choose an instance that has fewer vCPUs. For more information, see the [Choose single-core Premium plans](#).

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](#).

With an App Service plan, you can manually scale out by adding more VM instances. You can also enable autoscale. 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](#).

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 in both runtime versions:

PLAN	RUNTIME VERSION	DEFAULT	MAXIMUM
Consumption	1.x	5	10
Consumption	2.x	5	10
App Service	1.x	Unlimited	Unlimited
App Service	2.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 / pricing tier** in the **Overview** tab for the function app in the [Azure portal](#). For App Service plans, the pricing tier is also indicated.

The screenshot shows the Azure portal's 'Function Apps' section. A specific function app, 'myfunctionapp', is selected. The 'Overview' tab is active. In the 'Configured features' section, there are links for 'Function app settings', 'Application settings', 'Deployment options configured with VSTS/ARM', and 'Application Insights'. The 'App Service plan / pricing tier' section is highlighted with a red box, showing 'CentralUSPlan (Consumption)'. Other details shown include the status as 'Running', subscription as 'Visual Studio Enterprise', resource group as 'myresourcegroup', and URL as 'https://myfunctionapp.azurewebsites.net'.

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 Functions rely on Azure Storage for operations such as managing triggers and logging function executions, but some storage accounts do not 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.

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

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.

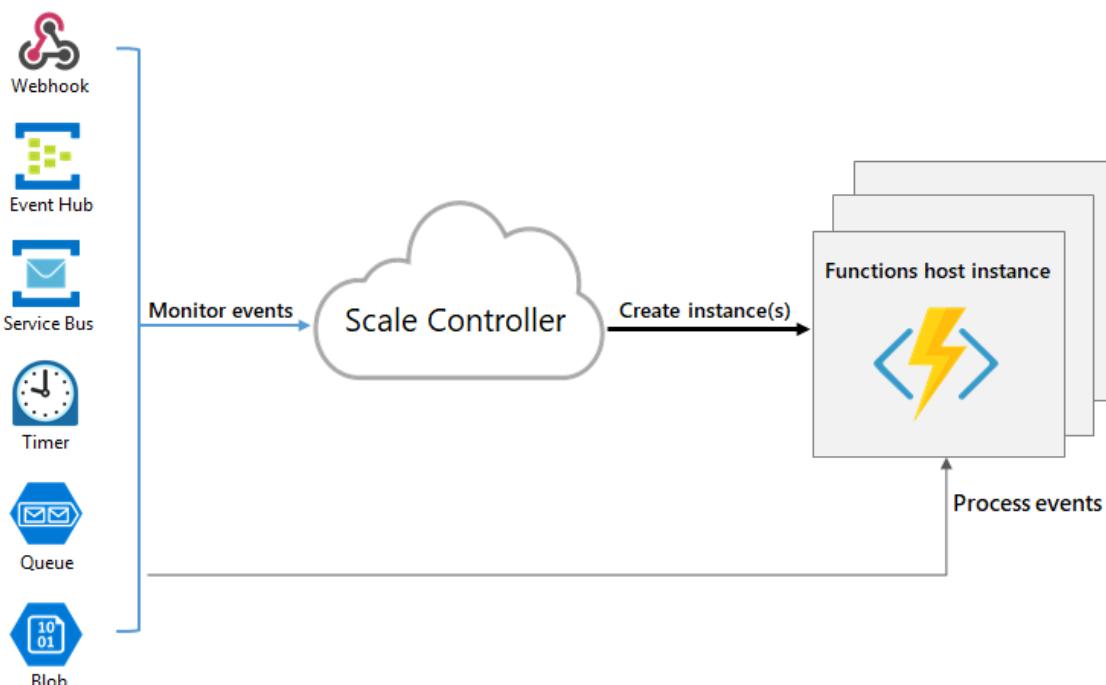
NOTE

When you're using a blob trigger on a Consumption plan, there can be up to a 10-minute delay in processing new blobs. This delay occurs when a function app has gone idle. After the function app is running, blobs are processed immediately. To avoid this cold-start delay, use the Premium plan, or use the [Event Grid trigger](#). For more information, see [the blob trigger binding reference article](#).

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 down to zero when no functions are running within a function app.



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 up 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.
- For HTTP triggers, new instances will only be allocated at most once every 1 second.
- For non-HTTP triggers, new instances will only be allocated at most once every 30 seconds.

Different triggers may also have different scaling limits as well as documented below:

- [Event Hub](#)

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](#).

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](#).

Service limits

The following table indicates the limits that apply to function apps when running in the various hosting plans:

RESOURCE	CONSUMPTION PLAN	PREMIUM PLAN	APP SERVICE PLAN ¹
Scale out	Event driven	Event driven	Manual/autoscale
Max instances	200	20	10-20
Default time out duration (min)	5	30	30 ²
Max time out duration (min)	10	unbounded	unbounded ³
Max outbound connections (per instance)	600 active (1200 total)	unbounded	unbounded
Max request size (MB) ⁴	100	100	100
Max query string length ⁴	4096	4096	4096
Max request URL length ⁴	8192	8192	8192
ACU per instance	100	210-840	100-840
Max memory (GB per instance)	1.5	3.5-14	1.75-14
Function apps per plan	100	100	unbounded ⁵
App Service plans	100 per region	100 per resource group	100 per resource group
Storage ⁶	1 GB	250 GB	50-1000 GB
Custom domains per app	500 ⁷	500	500

RESOURCE	CONSUMPTION PLAN	PREMIUM PLAN	APP SERVICE PLAN
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

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

² By default, the timeout for the Functions 1.x runtime in an App Service plan is unbounded.

³ Requires the App Service plan be set to [Always On](#). Pay at standard [rates](#).

⁴ These limits are [set in the host](#).

⁵ 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.

⁶ 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.

⁷ When your function app is hosted in a [Consumption plan](#), only the CNAME option is supported. For function apps in a [Premium plan](#) or an [App Service plan](#), you can map a custom domain using either a CNAME or an A record.