

Table of Contents

[Functions Documentation](#)

[Overview](#)

[About Functions](#)

[Serverless comparison](#)

[Quickstarts](#)

[Create function - portal](#)

[Create function - Azure CLI](#)

[Create function - Visual Studio](#)

[Create function - Java/Maven](#)

[Create function - Linux](#)

[Triggers](#)

[Blob storage](#)

[Azure Cosmos DB](#)

[Timer](#)

[Generic webhook](#)

[GitHub webhook](#)

[Queue storage](#)

[Integrate](#)

[Storage](#)

[Azure Cosmos DB](#)

[Tutorials](#)

[Functions with Logic Apps](#)

[Create a serverless API](#)

[Create an OpenAPI definition](#)

[Image resize with Event Grid](#)

[Functions on Linux](#)

[Custom image](#)

[IoT Edge device](#)

[Samples](#)

[Code samples](#)

[Azure CLI](#)

[Concepts](#)

[Scale and hosting](#)

[Triggers and bindings](#)

[Languages](#)

[Supported languages](#)

[C# \(class library\)](#)

[C# script \(.csx\)](#)

[F#](#)

[JavaScript](#)

[Java](#)

[Performance considerations](#)

[Functions Proxies](#)

[On-premises functions](#)

[Durable Functions](#)

[Overview](#)

[Bindings](#)

[Checkpoint and replay](#)

[Instance management](#)

[HTTP API](#)

[Error handling](#)

[Diagnostics](#)

[Timers](#)

[External events](#)

[Eternal orchestrations](#)

[Singleton orchestrations](#)

[Sub-orchestrations](#)

[Task hubs](#)

[Versioning](#)

[Performance and scale](#)

[How-to guides](#)

Develop

- [Developer guide](#)
- [Testing functions](#)
- [Develop and debug locally](#)
- [Visual Studio development](#)

Deploy

- [Continuous deployment](#)
- [Zip push deployment](#)
- [Automate resource deployment](#)
- [On-premises functions](#)

Configure

- [Manage a function app](#)
- [Set the runtime version](#)

Monitor

Integrate

- [Connect to SQL Database](#)
- [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 Managed Service Identity](#)

Durable Functions

- [Install](#)
- [Chaining](#)
- [Fan-out/fan-in](#)
- [Stateful singleton](#)
- [Human interaction](#)

Reference

- [Bindings](#)
- [Blob storage](#)
- [Azure Cosmos DB](#)
- [Event Hubs](#)

- [External file](#)
- [External table](#)
- [HTTP and webhooks](#)
- [Microsoft Graph](#)
- [Mobile Apps](#)
- [Notification Hubs](#)
- [Queue storage](#)
- [SendGrid](#)
- [Service Bus](#)
- [Table storage](#)
- [Timer](#)
- [Twilio](#)

[host.json reference](#)

[App settings reference](#)

[OpenAPI reference](#)

Resources

- [Azure Roadmap](#)
- [Pricing](#)
- [Pricing calculator](#)
- [Regional availability](#)
- [Videos](#)
- [MSDN forum](#)
- [Stack Overflow](#)
- [Twitter](#)

[Azure Functions GitHub repository](#)

[Service updates](#)

An introduction to Azure Functions

11/16/2017 • 4 min 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#, F#, Node.js, Java, 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#, F#, or Javascript. See [Supported languages](#) for other options.
- **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](#), [Visual Studio Team 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](#).
- **GitHub webhook** - Respond to events that occur in your GitHub repositories. For an example, see [Create a function triggered by a GitHub webhook](#).
- **Generic webhook** - Process webhook HTTP requests from any service that supports webhooks. For an example, see [Create a function triggered by a generic webhook](#).
- **CosmosDBTrigger** - Process Azure Cosmos DB documents when they are added or updated in collections in a NoSQL database. For an example, see [Create a function triggered by Azure Cosmos DB](#).
- **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 an example, see [Create a function triggered by Azure Queue storage](#).
- **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 Mobile Apps (tables)
- Azure Notification Hubs
- Azure Service Bus (queues and topics)
- Azure Storage (blob, queues, and tables)
- GitHub (webhooks)
- 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, mobile, and API 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.

Choose between Flow, Logic Apps, Functions, and WebJobs

1/19/2018 • 4 min to read • [Edit Online](#)

This article compares and contrasts the following services in the Microsoft cloud, which can all solve integration problems and automate business processes:

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

All these services are useful when "gluing" together disparate systems. They can all define input, actions, conditions, and output. You can run each of them on a schedule or trigger. However, each service has unique advantages, and comparing them is not a question of "Which service is the best?" but one of "Which service is best suited for this situation?" Often, a combination of these services is the best way to rapidly build a scalable, full-featured integration solution.

Flow vs. Logic Apps

We can discuss Microsoft Flow and Azure Logic Apps together because they are both *configuration-first* integration services. They make it easy to build processes and workflows and integrate with various SaaS and enterprise applications.

- Flow is built on top of Logic Apps
- They have the same workflow designer
- [Connectors](#) that work in one can also work in the other

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. On the other hand, Logic Apps can enable advanced integrations (for example, B2B processes) where enterprise-level DevOps and security practices are required. It is typical for a business workflow to grow in complexity over time. Accordingly, you can start with a flow at first, then convert it to a logic app as needed.

The following table helps you determine whether Flow or Logic Apps is best for a given integration.

	FLOW	LOGIC APPS
Audience	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.	DevOps: source control, testing, support, automation and manageability in Azure Resource Management

	FLOW	LOGIC APPS
Admin Experience	Manage Flow Environments and Data Loss Prevention (DLP) policies, track licensing https://admin.flow.microsoft.com	Manage Resource Groups, Connections, Access Management and Logging https://portal.azure.com
Security	Office 365 Security and Compliance audit logs, Data Loss Prevention (DLP), encryption at rest for sensitive data, etc.	Security assurance of Azure: Azure Security , Security Center , audit logs , and more.

Functions vs. WebJobs

We can discuss Azure Functions and Azure App Service WebJobs together because they are both *code-first* integration services and designed for developers. They enable you to run a script or a piece of code in response to various events, such as [new Storage Blobs](#) or a [WebHook request](#). Here are their similarities:

- Both are built on [Azure App Service](#) and enjoy features such as [source control](#), [authentication](#), and [monitoring](#).
- Both are developer-focused services.
- Both support standard scripting and programming languages.
- Both have NuGet and NPM support.

Functions is the natural evolution of WebJobs in that it takes the best things about WebJobs and improves upon them. The improvements include:

- [Serverless](#) app model.
- Streamlined dev, test, and run of code, directly in the browser.
- Built-in integration with more Azure services and 3rd-party services like [GitHub WebHooks](#).
- Pay-per-use, no need to pay for an [App Service plan](#).
- Automatic, [dynamic scaling](#).
- For existing customers of App Service, running on App Service plan still possible (to take advantage of under-utilized resources).
- Integration with Logic Apps.

The following table summarizes the differences between Functions and WebJobs:

	FUNCTIONS	WEBJOBS
Scaling	Configurationless scaling	Scale with App Service plan
Pricing	Pay-per-use or part of App Service plan	Part of App Service plan
Run-type	Triggered, scheduled (by timer trigger)	Triggered, continuous, scheduled
Trigger events	Timer , Azure Cosmos DB , Azure Event Hubs , HTTP/WebHook (GitHub, Slack) , Azure App Service Mobile Apps , Azure Event Hubs , Azure Storage queues and blobs , Azure Service Bus queues and topics	Azure Storage queues and blobs , Azure Service Bus queues and topics
In-browser development	Supported	Not Supported
C#	Supported	Supported

	FUNCTIONS	WEBJOBS
F#	Supported	Not Supported
JavaScript	Supported	Supported
Java	Preview	Not supported
Bash	Experimental	Supported
Windows scripting (.cmd, .bat)	Experimental	Supported
PowerShell	Experimental	Supported
PHP	Experimental	Supported
Python	Experimental	Supported
TypeScript	Experimental	Not Supported

Whether to use Functions or WebJobs ultimately depends on what you're already doing with App Service. If you have an App Service app for which you want to run code snippets, and you want to manage them together in the same DevOps environment, use WebJobs. In the following scenarios, use Functions.

- You want to run code snippets for other Azure services or 3rd-party apps.
- You want to manage your integration code separately from your App Service apps.
- You want to call your code snippets from a Logic app.

Flow, Logic Apps, and Functions together

As previously mentioned, which service is best suited to you depends on your situation.

- For simple business optimization, use Flow.
- If your integration scenario is too advanced for Flow, or you need DevOps capabilities, then use Logic Apps.
- If a step in your integration scenario requires highly custom transformation or specialized code, then write a function and trigger the function as an action in your logic app.

You can call a logic app in a flow. You can also call a function in a logic app, and a logic app in a function. The integration between Flow, Logic Apps, and Functions continues to improve over time. You can build something in one service and use it in the other services. Therefore, any investment you make in these three technologies is worthwhile.

Next steps

Get started with each of the services by creating your first flow, logic app, function app, or WebJob. Click any of the following links:

- [Get started with Microsoft Flow](#)
- [Create a logic app](#)
- [Create your first Azure Function](#)
- [Deploy WebJobs using Visual Studio](#)

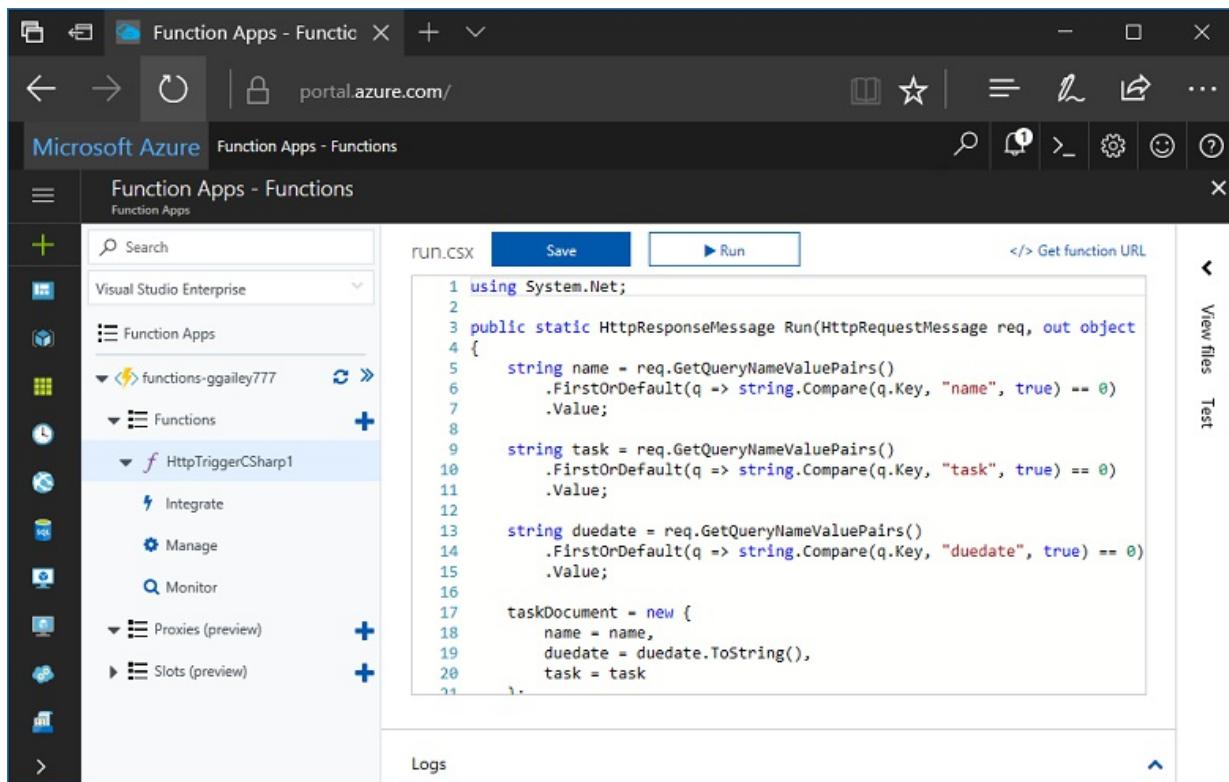
Or, get more information on these integration services with the following links:

- [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](#)

Create your first function in the Azure portal

1/12/2018 • 5 min 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 topic, 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.

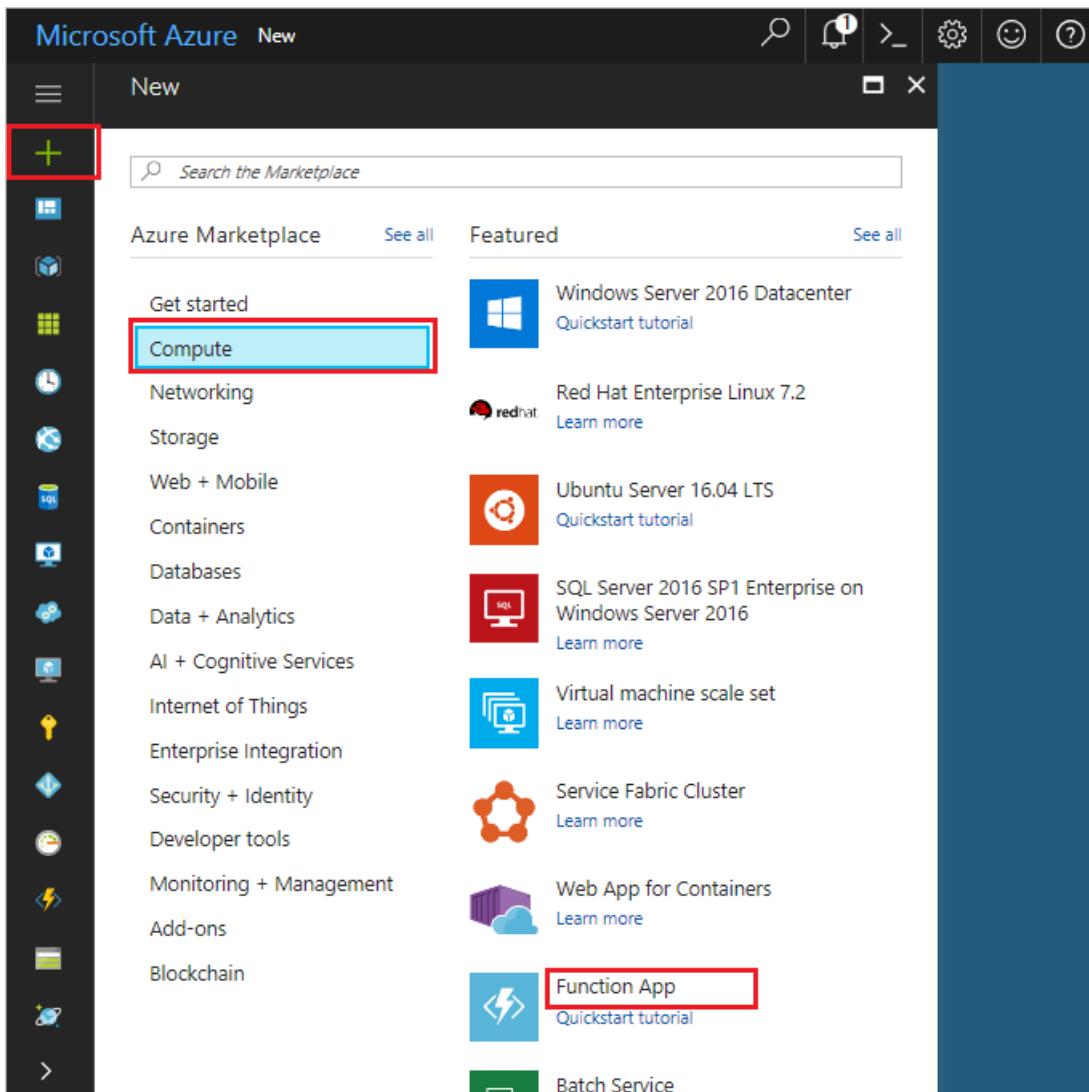
Sign in to Azure

Open the Azure portal. To do this, sign in to the [Azure portal](#) with your Azure account.

Create a function app

You must have a function app to host the execution of your functions. A function app lets you group functions as a logic unit for easier management, deployment, and sharing of resources.

1. Click the **New** 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
functions-ggailey777.azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
functions-ggailey777

* OS Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
West Europe

* Storage ⓘ
 Create new Use existing
functions-ggaile87e8

Application Insights ⓘ On Off

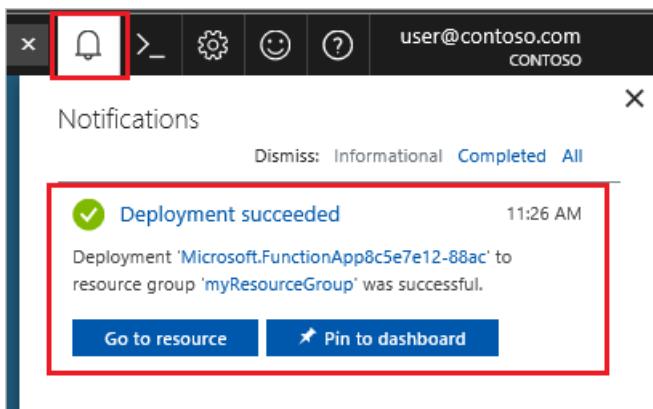
Pin to dashboard

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.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.

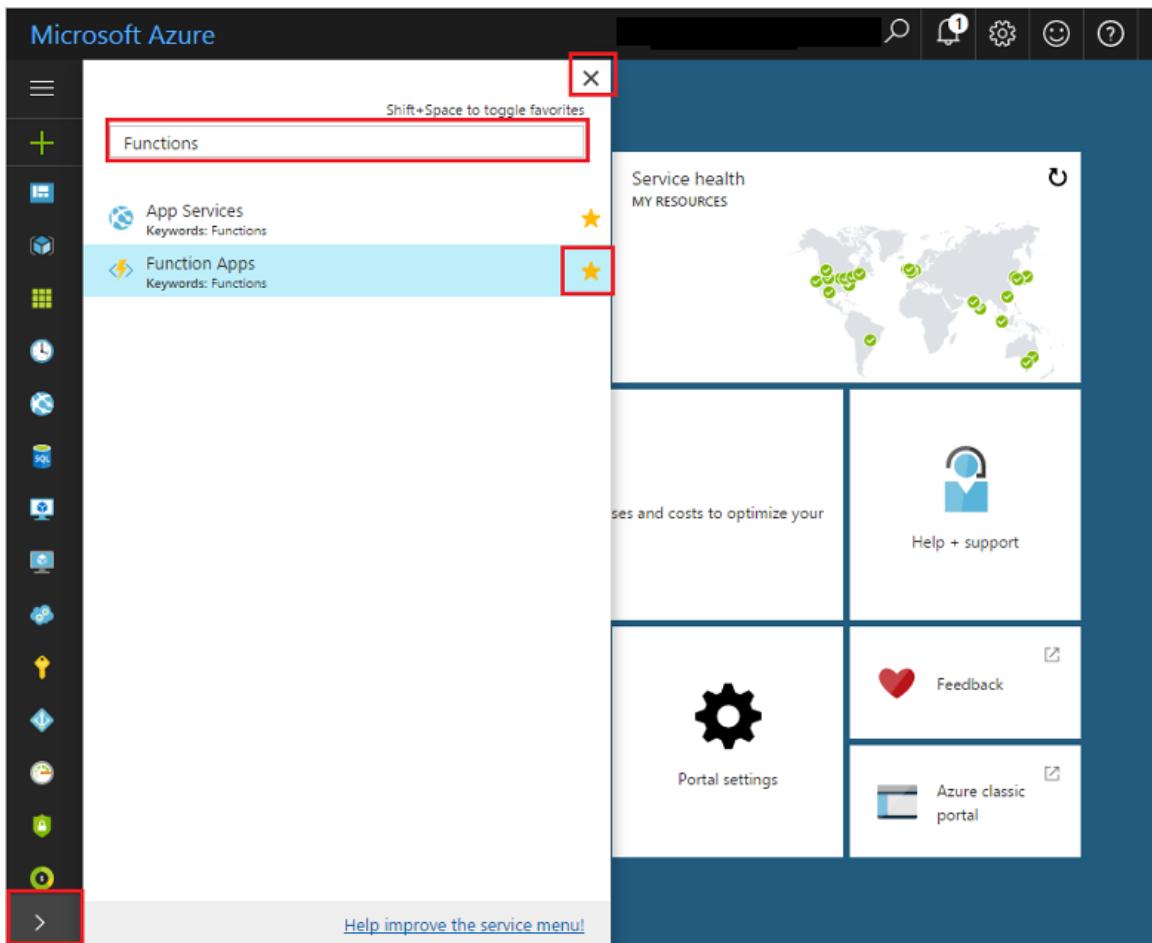


Clicking **Go to resource** takes you to your new function app.

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

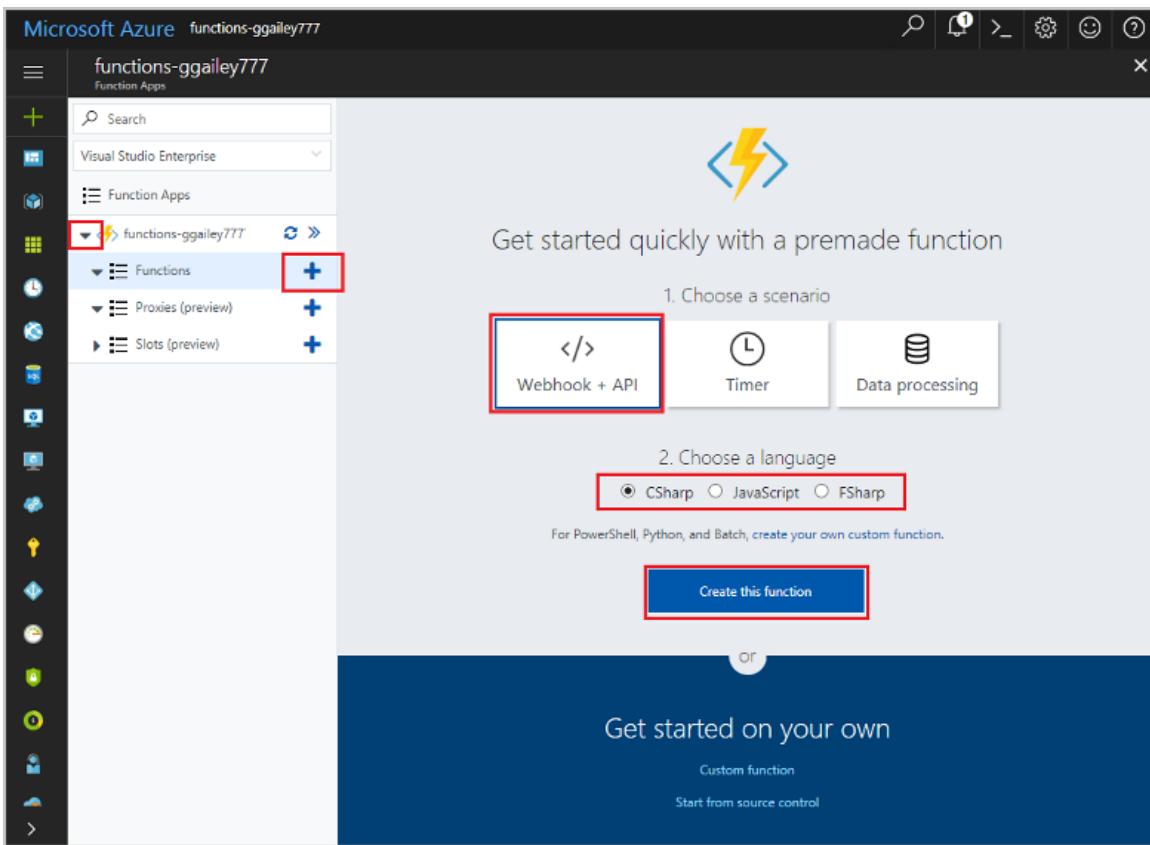
3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

The screenshot shows the 'Function Apps' blade in the Microsoft Azure portal. On the left, there is a sidebar with a 'Function Apps' icon highlighted by a red box. The main content area shows a table of function apps. The table has columns for NAME, SUBSCRIPTION ID, RESOURCE GROUP, and LOCATION. One row is visible, showing 'functions-ggailey777' under NAME, 'Visual Studio Enterprise' under SUBSCRIPTION ID, 'functions-ggailey777' under RESOURCE GROUP, and 'southcentralus' under LOCATION. The table header includes 'NAME ▾', 'SUBSCRIPTION ID ▾', 'RESOURCE GROUP', and 'LOCATION'. At the bottom of the sidebar, there is a red box highlighting the 'Functions' icon again.

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

Create an HTTP triggered function

1. Expand your new function app, then click the + button next to **Functions**.
2. In the **Get started quickly** page, select **WebHook + API**, **Choose a language** for your function, and click **Create this function**.

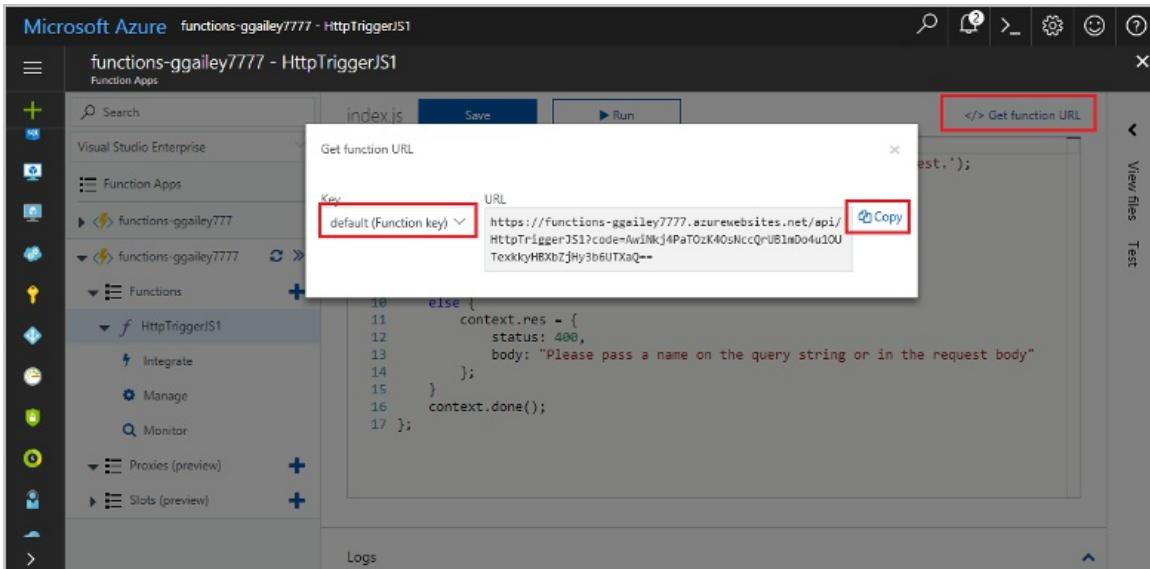


A function is created in your chosen language using the template for an HTTP triggered function. This topic shows a C# script function in the portal, but you can create a function in any [supported language](#).

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

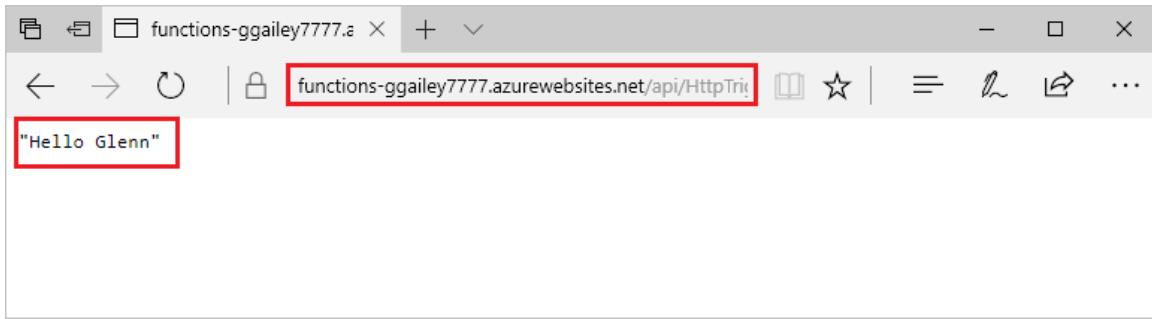
Test the function

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



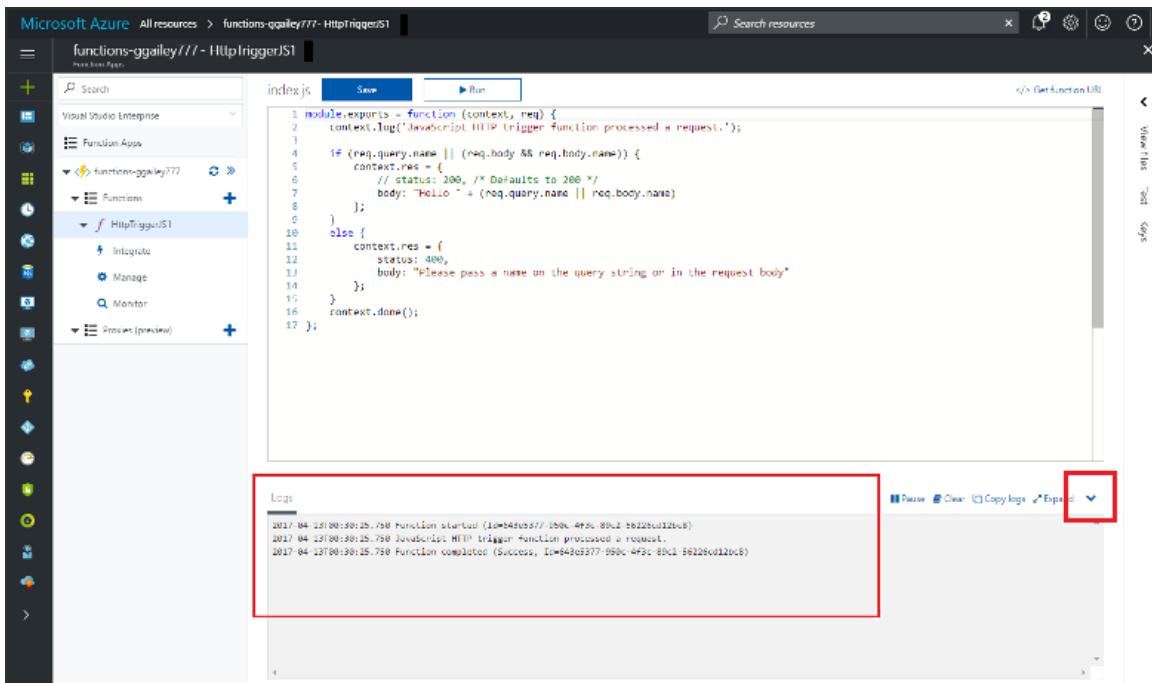
2. Paste the function URL into your browser's address bar. Add the query string value `&name=<yourname>` to the end of this URL and press the `Enter` key on your keyboard to execute the request. You should see the response returned by the function displayed in the browser.

The following is an example of the response in the Edge browser (other browsers may include displayed XML):



The request URL includes a key that is required, by default, to access your function over HTTP.

- When your function runs, trace information is written to the logs. To see the trace output 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:

- 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 resource group named 'functions-ggailey777'. The 'Overview' tab is selected. Key details shown include:

- Status: Running
- Subscription: Visual Studio Enterprise
- Resource group: functions-ggailey777
- Location: South Central US
- URL: https://functions-ggailey777.azurewebsites.net

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.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information, see [Azure Functions HTTP and webhook bindings](#).

Create your first function using the Azure CLI

11/15/2017 • 4 min to read • [Edit Online](#)

This quickstart topic walks you through how to use Azure Functions to create your first function. You use the Azure CLI to create a function app, which is the [serverless](#) infrastructure that hosts your function. The function code itself is deployed from a GitHub sample repository.

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

Prerequisites

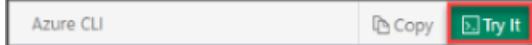
Before running this sample, you must have the following:

- An active [GitHub](#) account.
- An active Azure subscription.

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

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account. Just click the **Copy** to copy the code, paste it into the Cloud Shell, and then press enter to run it. There are a few ways to launch the Cloud Shell:

Click Try It in the upper right corner of a code block.	
Open Cloud Shell in your browser.	
Click the Cloud Shell button on the menu in the upper right of the Azure portal .	

If you choose to install and use the CLI locally, this topic requires the Azure CLI version 2.0 or later. Run `az --version` to find the version you have. If you need to install or upgrade, see [Install Azure CLI 2.0](#).

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. To see all supported locations for App Service plans, run the [az appservice list-locations](#) command.

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
```

After the storage account has been created, the Azure CLI shows information similar to the following example:

```
{
  "creationTime": "2017-04-15T17:14:39.320307+00:00",
  "id": "/subscriptions/bbbef702-e769-477b-9f16-bc4d3aa97387/resourceGroups/myresourcegroup/...",
  "kind": "Storage",
  "location": "westerurope",
  "name": "myfunctionappstorage",
  "primaryEndpoints": {
    "blob": "https://myfunctionappstorage.blob.core.windows.net/",
    "file": "https://myfunctionappstorage.file.core.windows.net/",
    "queue": "https://myfunctionappstorage.queue.core.windows.net/",
    "table": "https://myfunctionappstorage.table.core.windows.net/"
  },
  ...
  // Remaining output has been truncated for readability.
}
```

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. The `deployment-source-url` parameter is a sample repository in GitHub that contains a "Hello World" HTTP triggered function.

```
az functionapp create --name <app_name> --storage-account <storage_name> --resource-group myResourceGroup \
--consumption-plan-location westerurope --deployment-source-url https://github.com/Azure-Samples/functions-quickstart
```

Setting the `consumption-plan-location` parameter means that the function app is hosted in a Consumption hosting plan. In this 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](#).

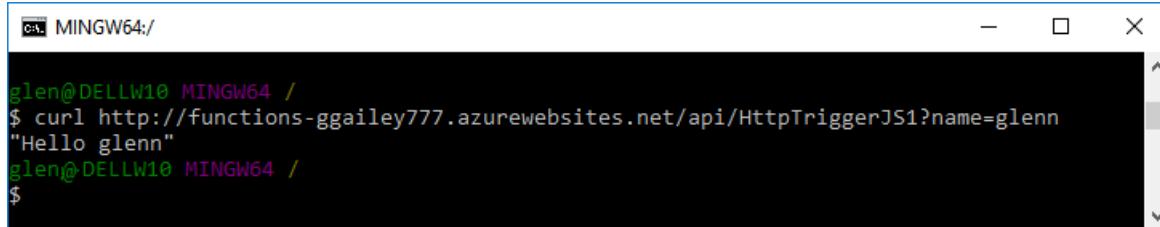
After the function app has been created, the Azure CLI shows information similar to the following example:

```
{  
    "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.  
}
```

Test the function

Use cURL to test the deployed function on a Mac or Linux computer or using Bash on Windows. Execute the following cURL command, replacing the <app_name> placeholder with the name of your function app. Append the query string &name=<yourname> to the URL.

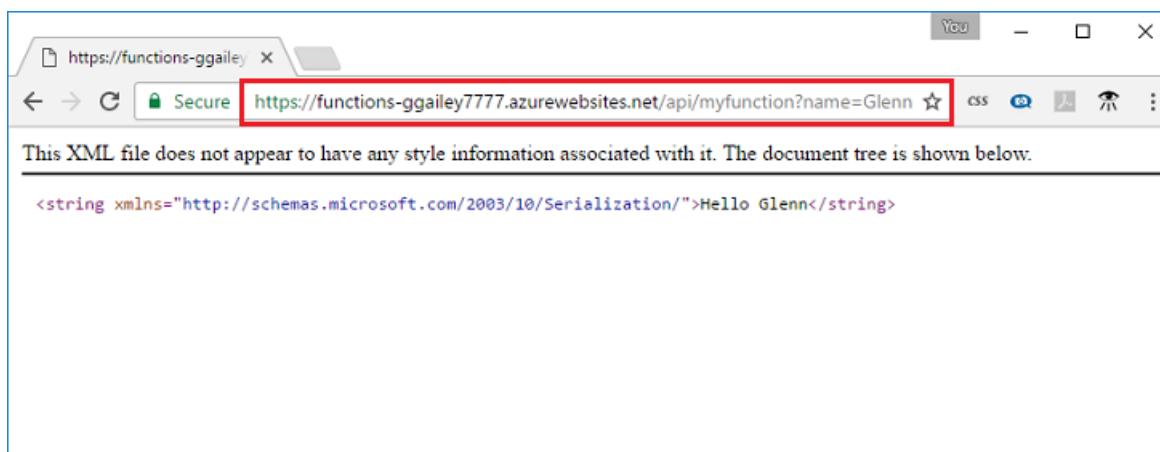
```
curl http://<app_name>.azurewebsites.net/api/HttpTriggerJS1?name=<yourname>
```



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

If you don't have cURL available in your command line, enter the same URL in the address of your web browser. Again, replace the <app_name> placeholder with the name of your function app, and append the query string &name=<yourname> to the URL and execute the request.

```
http://<app_name>.azurewebsites.net/api/HttpTriggerJS1?name=<yourname>
```



Clean up resources

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

continue, use the following command to delete all resources created by this quickstart:

```
az group delete --name myResourceGroup
```

Type `y` when prompted.

Next steps

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

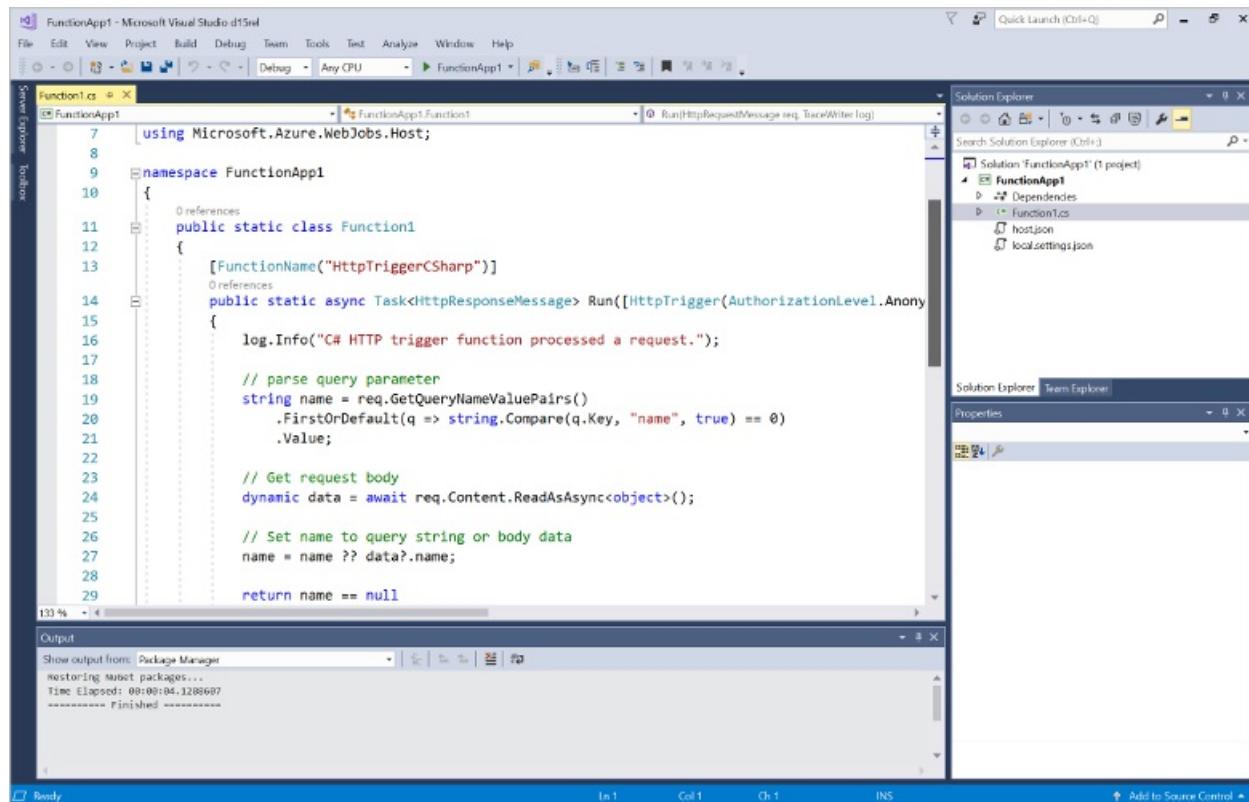
[Code and test Azure Functions locally](#)

Create your first function using Visual Studio

1/19/2018 • 5 min 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 Visual Studio 2017 tools for Azure Functions to locally create and test a "hello world" function. You then publish the function code to Azure. These tools are available as part of the Azure development workload in Visual Studio 2017.

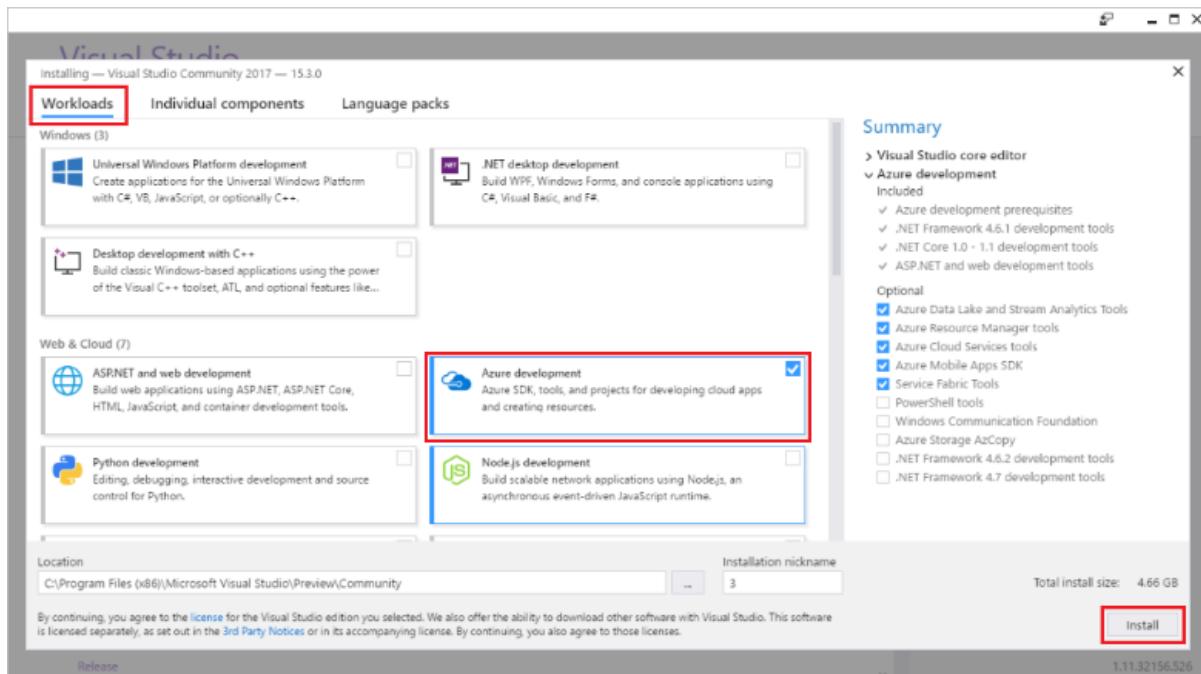


This topic includes a [video](#) that demonstrates the same basic steps.

Prerequisites

To complete this tutorial:

- Install [Visual Studio 2017 version 15.4](#) or a later version, including the **Azure development** workload.



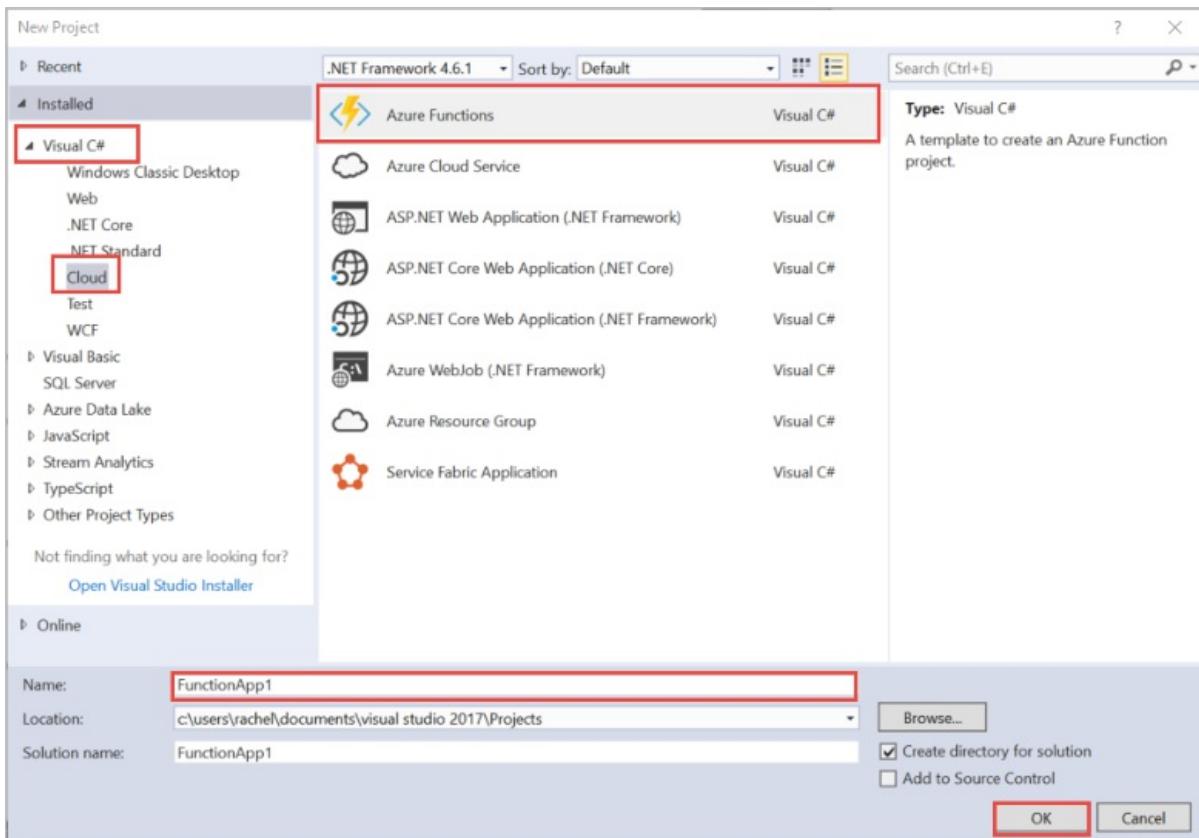
- Make sure you have updated to the most recent version of Azure Functions and WebJobs Tools. Do this under **Updates > Visual Studio Marketplace** in **Extensions and Updates**.

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

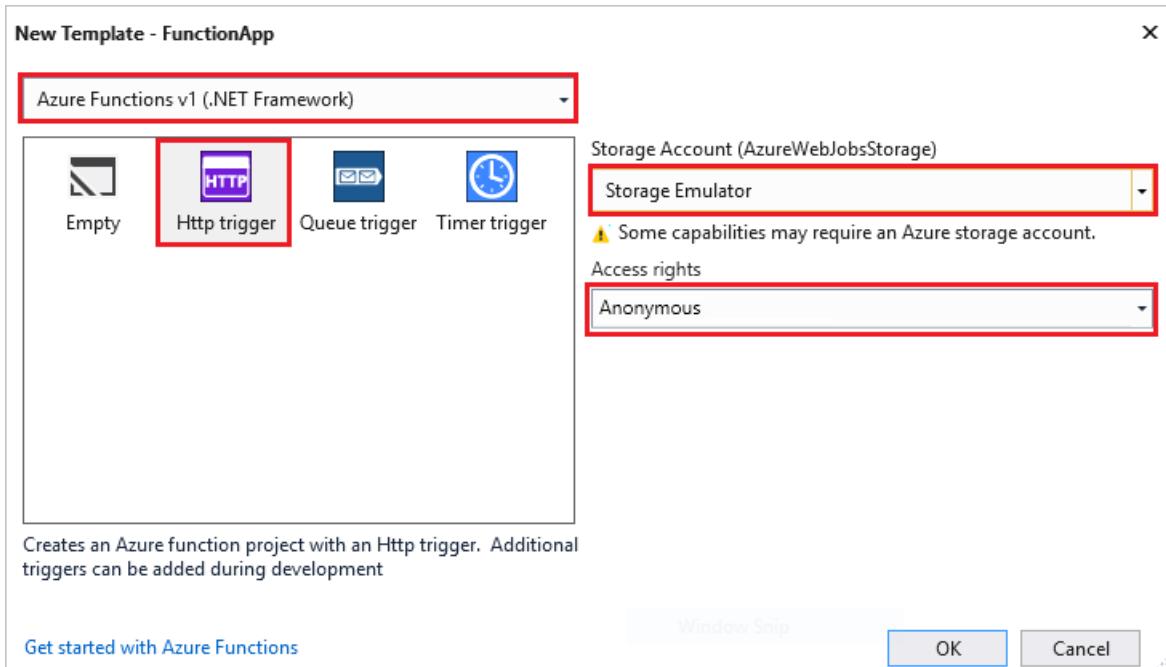
Create a function app project

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

1. In Visual Studio, select **New > Project** from the **File** menu.
2. In the **New Project** dialog, select **Installed**, expand **Visual C# > Cloud**, select **Azure Functions**, type a **Name** for your project, and click **OK**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other nonalphanumeric characters.



3. Use the settings specified in the table that follows the image.

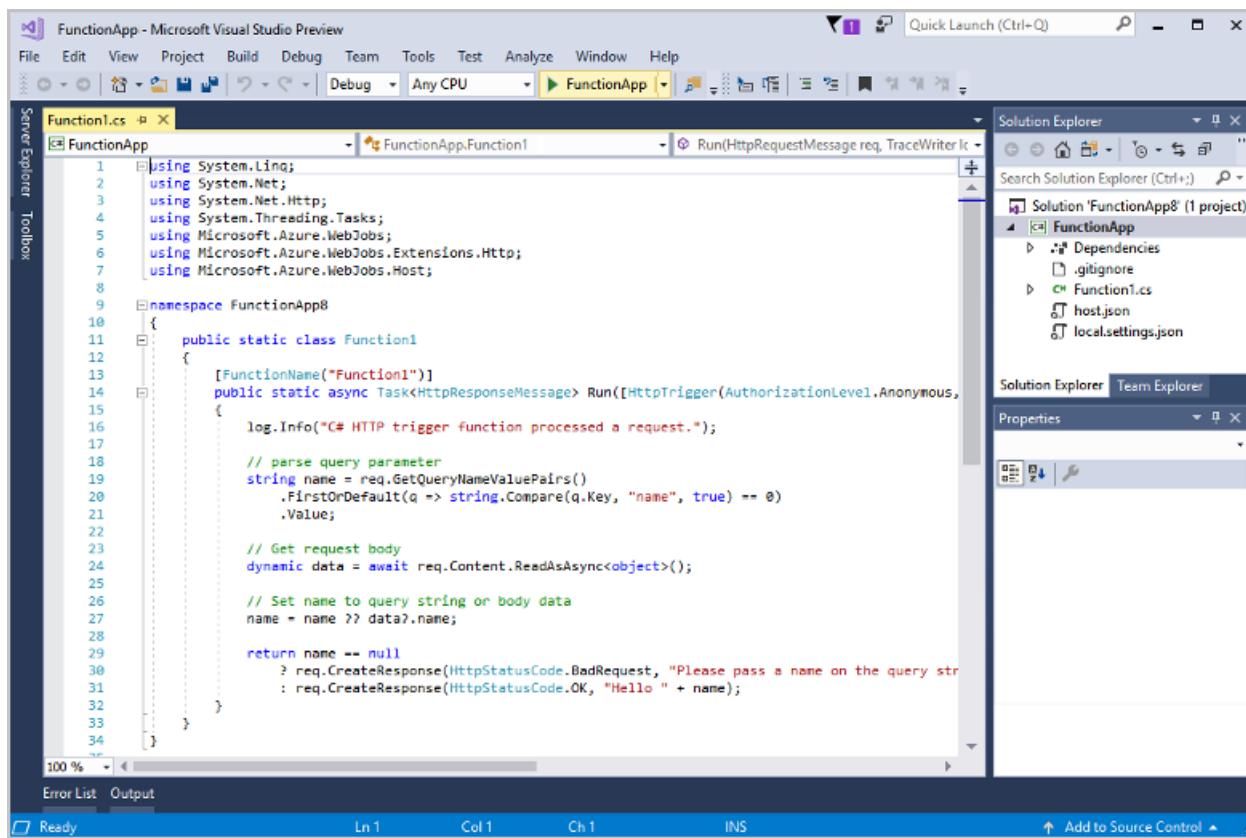


SETTING	SUGGESTED VALUE	DESCRIPTION
Version	Azure Functions v1 (.NET Framework)	This creates a function project that uses the version 1 runtime of Azure Functions. The version 2 runtime, which supports .NET Core, is currently in preview. For more information, see How to target Azure Functions runtime version .
Template	HTTP trigger	This creates a function triggered by an HTTP request.

SETTING	SUGGESTED VALUE	DESCRIPTION
Storage account	Storage Emulator	An HTTP trigger doesn't use the Storage account connection. All other trigger types require a valid Storage account connection string.
Access rights	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 .

- Click **OK** to create the function project and HTTP triggered function.

Visual Studio creates a project and in it a class that contains boilerplate code for the chosen function type. The **FunctionName** attribute on the method sets the name of the function. 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 add input and output bindings to a function 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 have created your function project and an HTTP-triggered function, you can test it on your local computer.

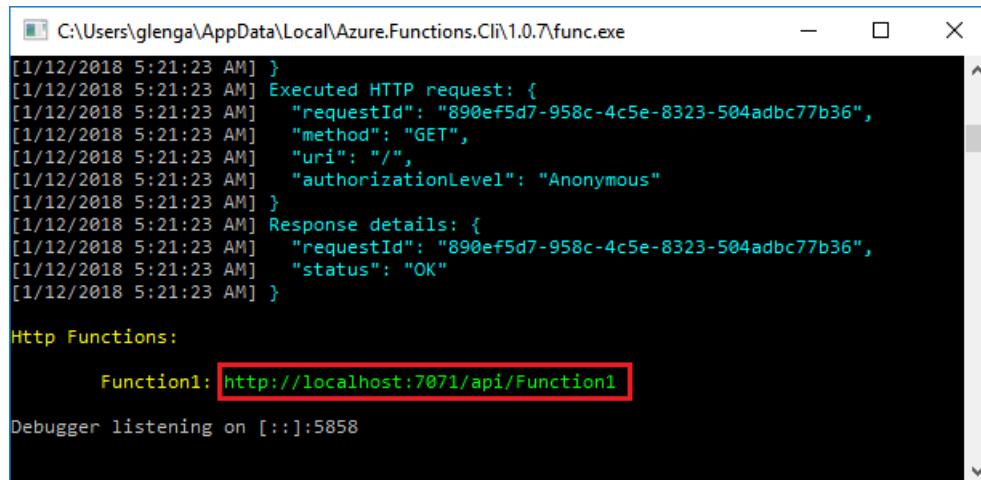
Test the function locally

Azure Functions Core Tools lets you run an 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.

2. Copy the URL of your function from the Azure Functions runtime output.



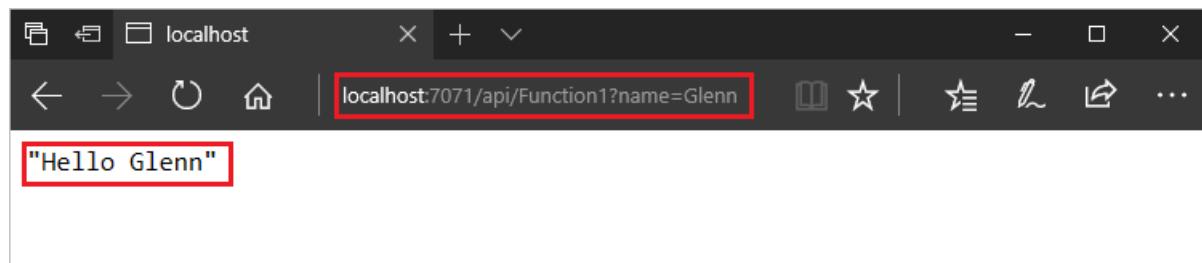
```
C:\Users\glenga\AppData\Local\Azure.Functions.Cli\1.0.7\func.exe
[1/12/2018 5:21:23 AM] }
[1/12/2018 5:21:23 AM] Executed HTTP request: {
[1/12/2018 5:21:23 AM]   "requestId": "890ef5d7-958c-4c5e-8323-504adbc77b36",
[1/12/2018 5:21:23 AM]   "method": "GET",
[1/12/2018 5:21:23 AM]   "uri": "/",
[1/12/2018 5:21:23 AM]   "authorizationLevel": "Anonymous"
[1/12/2018 5:21:23 AM] }
[1/12/2018 5:21:23 AM] Response details: {
[1/12/2018 5:21:23 AM]   "requestId": "890ef5d7-958c-4c5e-8323-504adbc77b36",
[1/12/2018 5:21:23 AM]   "status": "OK"
[1/12/2018 5:21:23 AM] }

Http Functions:

    Function1: http://localhost:7071/api/Function1

Debugger listening on [::]:5858
```

3. Paste the URL for the HTTP request 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 local GET request returned by the function:



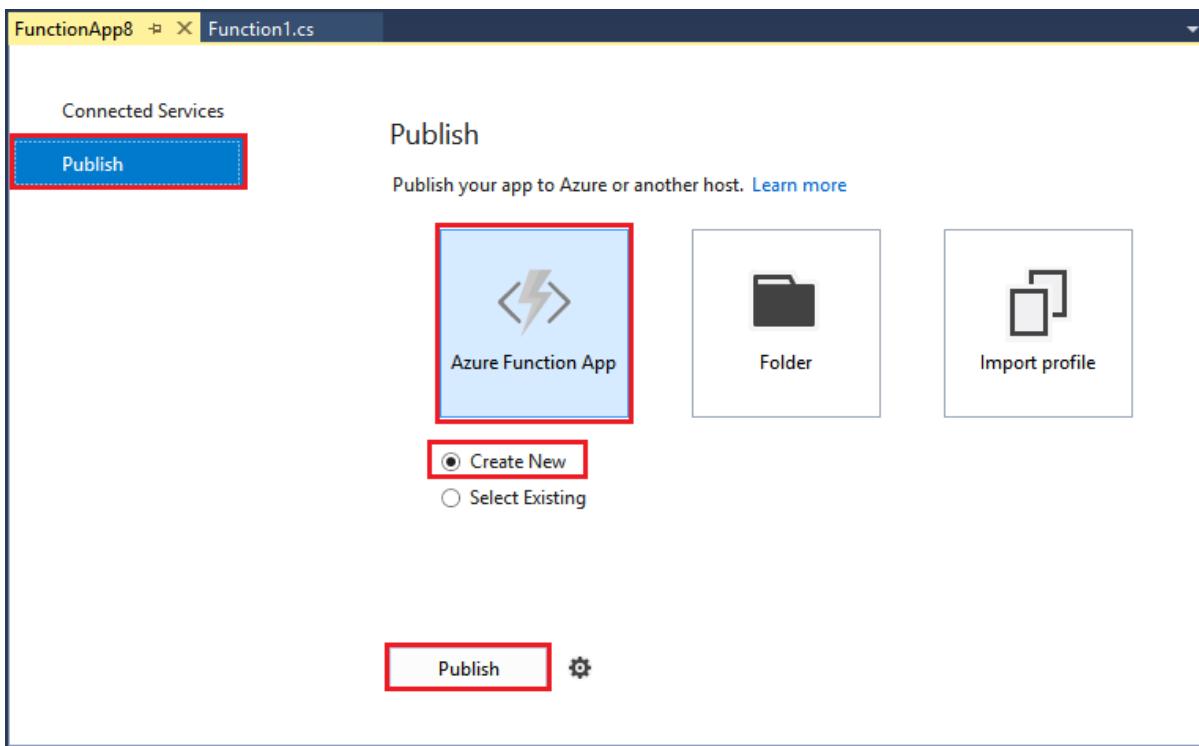
4. To stop debugging, click the **Stop** button on the Visual Studio toolbar.

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. You can create a function app right from Visual Studio.

1. In **Solution Explorer**, right-click the project and select **Publish**. Choose **Create New** and then **Publish**.



2. If you haven't already connected Visual Studio to your Azure account, select **Add an account...**
3. In the **Create App Service** dialog, use the **Hosting** settings as specified in the following table:

Create App Service
Host your web and mobile applications, REST APIs, and more in Azure

Hosting **Services**

App Name	FunctionApp20180114125726
Subscription	Visual Studio Enterprise
Resource Group	myResourceGroup (southcentralus)
App Service Plan	FunctionApp20180114125726Plan*

Add an account...

Clicking the Create button will create the following Azure resources
[Explore additional Azure services](#)

App Service - FunctionApp20180114125726
App Service Plan - FunctionApp20180114125726Plan

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... **Create** **Cancel**

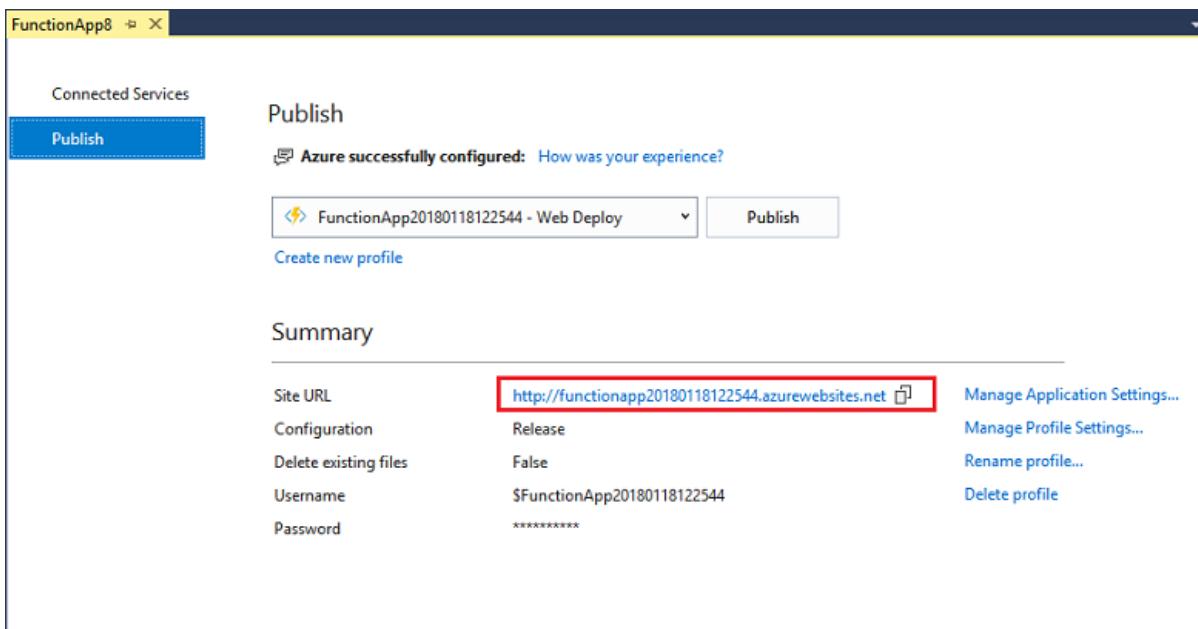
SETTING	SUGGESTED VALUE	DESCRIPTION
App Name	Globally unique name	Name that uniquely identifies your new function app.
Subscription	Choose your subscription	The Azure subscription to use.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource Group	myResourceGroup	Name of the resource group in which to create your function app. Choose New to create a new resource group.
App Service Plan	Consumption plan	Make sure to choose the Consumption under Size after you click New to create a new plan. Also, choose a Location in a region near you or near other services your functions access.

NOTE

An Azure storage account is required by the Functions runtime. Because of this, a new Azure Storage account is created for you when you create a function app.

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



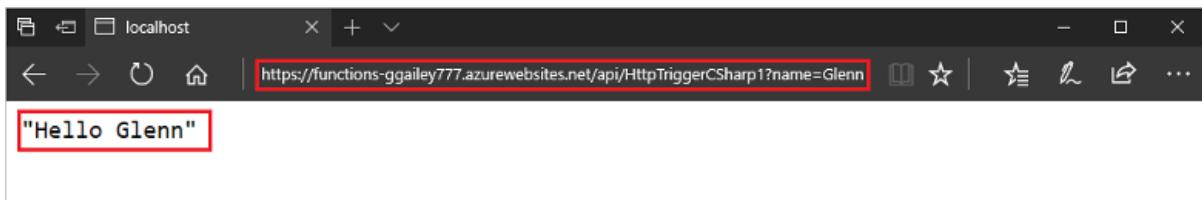
Test your function in Azure

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=<yourname>` to 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:



Watch the video

Next steps

You have used Visual Studio to create a C# function app with a simple HTTP triggered function.

- To learn how to configure your project to support other types of triggers and bindings, see the [Configure the project for local development](#) section in [Azure Functions Tools for Visual Studio](#).
- To learn more about local testing and debugging using the Azure Functions Core Tools, see [Code and test Azure Functions locally](#).
- To learn more about developing functions as .NET class libraries, see [Using .NET class libraries with Azure Functions](#).

Create your first function with Java and Maven (Preview)

1/4/2018 • 3 min to read • [Edit Online](#)

NOTE

Java for Azure Functions is currently in preview.

This quickstart guides through creating a [serverless](#) function project with Maven, testing it locally, and deploying it to Azure Functions. When you're done, you have a HTTP-triggered function app running in Azure.



```
[INFO] Deploying Function App with package...
[INFO] Successfully deployed Function App with package.
[INFO] Deleting deployment package from Azure Storage...
[INFO] Successfully deleted deployment package fabrikam-function-20170920120101928.20170928171008689.zip
[INFO] Successfully deployed Function App at https://fabrikam-function-20170920120101928.azurewebsites.net
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 55.794 s
[INFO] Finished at: 2017-09-28T17:10:25-07:00
[INFO] Final Memory: 41M/493M
[INFO] -----
$ curl -w '\n' -d HelloAzure https://fabrikam-function-20170920120101928.azurewebsites.net/api/hello
Hello, HelloAzure!
$
```

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

Prerequisites

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

- [.NET Core](#), latest version.
- [Java Developer Kit](#), version 8.
- [Azure CLI](#)
- [Apache Maven](#), version 3.0 or above.
- [Node.js](#), version 8.6 or higher.

IMPORTANT

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

Install the Azure Functions Core Tools

The [Azure Functions Core Tools 2.0](#) provide a local development environment for writing, running, and debugging Azure Functions. Install the tools with [npm](#), included with [Node.js](#).

```
npm install -g azure-functions-core-tools@core
```

NOTE

If you have trouble installing Azure Functions Core Tools version 2.0, see [Version 2.x runtime](#).

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
```

Windows (CMD)

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

Maven prompts 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.

```
Define value for property 'groupId': com.fabrikam.functions
Define value for property 'artifactId' : 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:
Confirm properties configuration: Y
```

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 request:

```

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<String> hello(
        @HttpTrigger(name = "req", methods = {"get", "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.createResponse(400, "Please pass a name on the query string or in the request body");
        } else {
            return request.createResponse(200, "Hello, " + name);
        }
    }
}

```

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:

```

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:

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

```
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. [Log in with the Azure CLI](#) and then deploy your code into a new Function app using the `azure-functions:deploy` Maven target.

```
az login  
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:

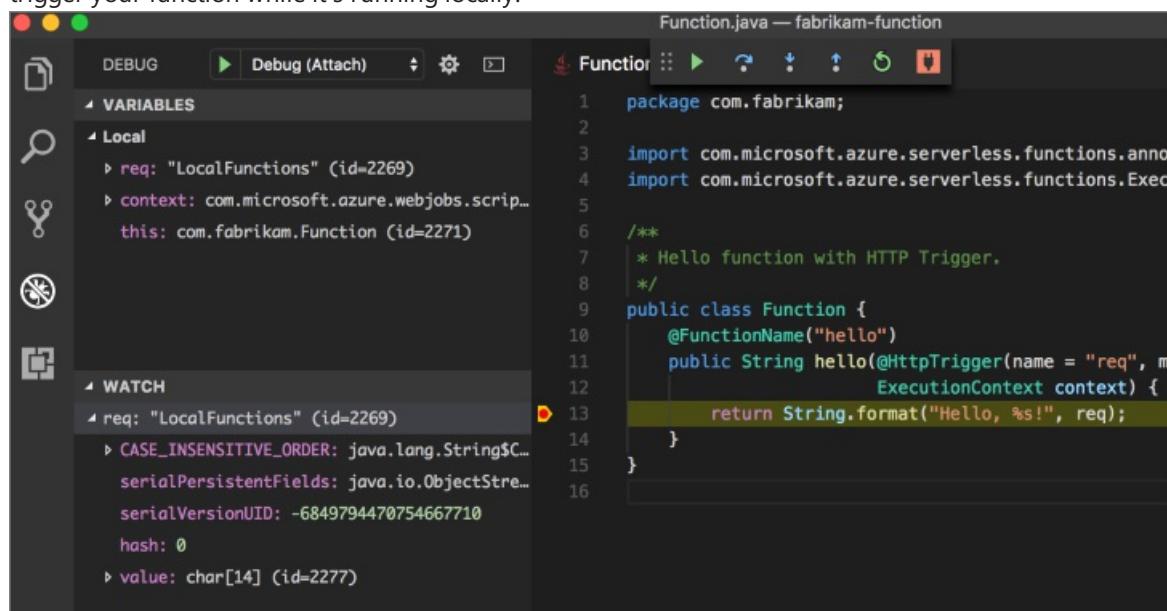
```
curl -w '\n' https://fabrikam-function-20170920120101928.azurewebsites.net/api/hello -d AzureFunctions
```

```
Hello AzureFunctions!
```

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.
- Debug functions locally with Visual Studio Code. With the [Java extension pack](#) installed and with your Functions project open in Visual Studio Code, [attach the debugger](#) to port 5005. Then set a breakpoint in the editor and trigger your function while it's running locally:



Create your first function running on Linux using the Azure CLI (preview)

11/16/2017 • 5 min to read • [Edit Online](#)

Azure Functions lets you host your functions on Linux in a default Azure App Service container. This functionality is currently in preview. You can also [bring your own custom container](#).

This quickstart topic walks you through how to use Azure Functions with the Azure CLI to create your first function app on Linux hosted on the default App Service container. The function code itself is deployed to the image from a GitHub sample repository.

The following steps are supported on a Mac, Windows, or Linux computer.

Prerequisites

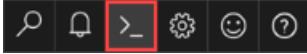
To complete this quickstart, you need:

- An active Azure subscription.

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

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account. Just click the **Copy** to copy the code, paste it into the Cloud Shell, and then press enter to run it. There are a few ways to launch the Cloud Shell:

Click Try It in the upper right corner of a code block.	
Open Cloud Shell in your browser.	
Click the Cloud Shell button on the menu in the upper right of the Azure portal .	

If you choose to install and use the CLI locally, this topic requires the Azure CLI version 2.0.21 or later. Run `az --version` to find the version you have. If you need to install or upgrade, see [Install Azure CLI 2.0](#).

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. To see all supported locations for App Service plans, run the [az appservice list-locations](#) command.

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
```

After the storage account has been created, the Azure CLI shows information similar to the following example:

```
{
  "creationTime": "2017-04-15T17:14:39.320307+00:00",
  "id": "/subscriptions/bbbef702-e769-477b-9f16-bc4d3aa97387/resourceGroups/myresourcegroup/...",
  "kind": "Storage",
  "location": "westerurope",
  "name": "myfunctionappstorage",
  "primaryEndpoints": {
    "blob": "https://myfunctionappstorage.blob.core.windows.net/",
    "file": "https://myfunctionappstorage.file.core.windows.net/",
    "queue": "https://myfunctionappstorage.queue.core.windows.net/",
    "table": "https://myfunctionappstorage.table.core.windows.net/"
  },
  ....
  // Remaining output has been truncated for readability.
}
```

Create a Linux App Service plan

Linux hosting for Functions is currently only supported on an App Service plan. Consumption plan hosting is not yet supported. To learn more about hosting, see [Azure Functions hosting plans comparison](#).

In the Cloud Shell, create an App Service plan in the resource group with the [az appservice plan create](#) command.

The following example creates an App Service plan named `myAppServicePlan` in the **Standard** pricing tier (`--sku S1`) and in a Linux container (`--is-linux`).

```
az appservice plan create --name myAppServicePlan --resource-group myResourceGroup --sku S1 --is-linux
```

When the App Service plan has been created, the Azure CLI shows information similar to the following example:

```
{  
    "adminSiteName": null,  
    "appServicePlanName": "myAppServicePlan",  
    "geoRegion": "West Europe",  
    "hostingEnvironmentProfile": null,  
    "id": "/subscriptions/0000-  
0000/resourceGroups/myResourceGroup/providers/Microsoft.Web/serverfarms/myAppServicePlan",  
    "kind": "linux",  
    "location": "West Europe",  
    "maximumNumberOfWorkers": 1,  
    "name": "myAppServicePlan",  
    < JSON data removed for brevity. >  
    "targetWorkerSizeId": 0,  
    "type": "Microsoft.Web/serverfarms",  
    "workerTierName": null  
}
```

Create a function app on Linux

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. Create a function app by using the [az functionapp create](#) command with a Linux App Service plan.

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. The `deployment-source-url` parameter is a sample repository in GitHub that contains a "Hello World" HTTP triggered function.

```
az functionapp create --name <app_name> --storage-account <storage_name> --resource-group myResourceGroup \  
--plan myAppServicePlan --deployment-source-url https://github.com/Azure-Samples/functions-quickstart-linux
```

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

```
{  
    "availabilityState": "Normal",  
    "clientAffinityEnabled": true,  
    "clientCertEnabled": false,  
    "cloningInfo": null,  
    "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.  
}
```

Because `myAppServicePlan` is a Linux plan, the built-in docker image is used to create the container that runs the function app on Linux.

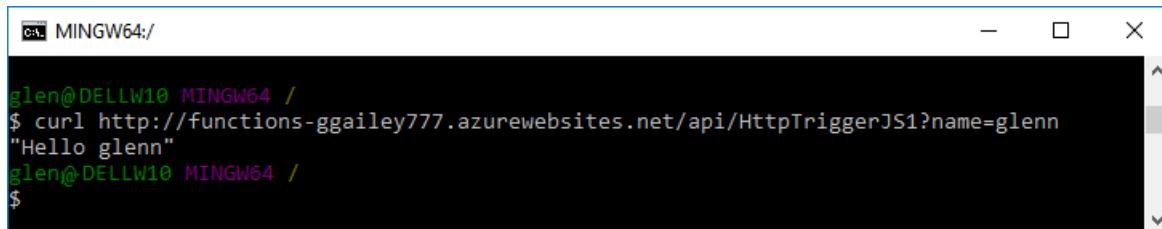
NOTE

The sample repository currently includes two scripting files, `deploy.sh` and `.deployment`. The `.deployment` file tells the deployment process to use `deploy.sh` as the [custom deployment script](#). In the current preview release, scripts are required to deploy the function app on a Linux image.

Test the function

Use cURL to test the deployed function on a Mac or Linux computer or using Bash on Windows. Execute the following cURL command, replacing the `<app_name>` placeholder with the name of your function app. Append the query string `&name=<yourname>` to the URL.

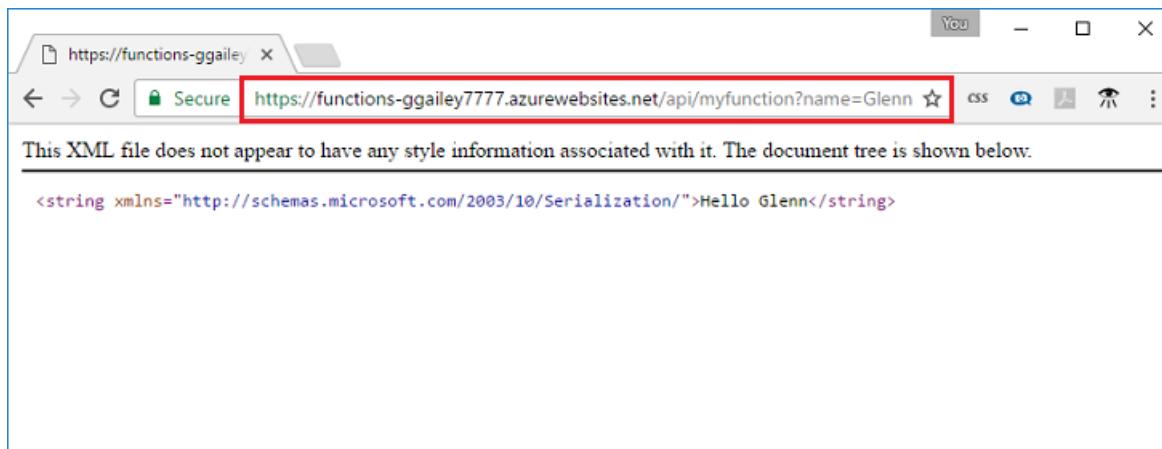
```
curl http://<app_name>.azurewebsites.net/api/HttpTriggerJS1?name=<yourname>
```



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

If you don't have cURL available in your command line, enter the same URL in the address of your web browser. Again, replace the `<app_name>` placeholder with the name of your function app, and append the query string `&name=<yourname>` to the URL and execute the request.

```
http://<app_name>.azurewebsites.net/api/HttpTriggerJS1?name=<yourname>
```



Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to continue on to work with subsequent quickstarts or with the tutorials, do not clean up the resources created in this quickstart. If you do not plan to continue, use the following command to delete all resources created by this quickstart:

```
az group delete --name myResourceGroup
```

Type `y` when prompted.

Next steps

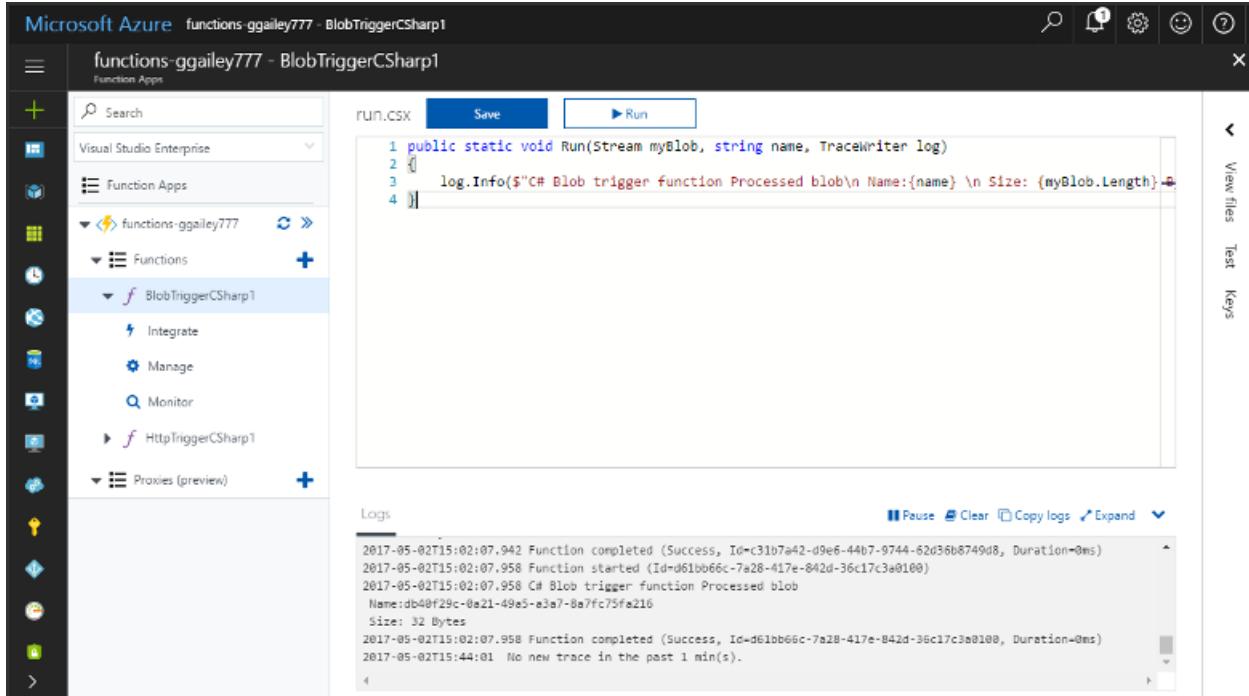
Learn more about developing Azure Functions locally using the Azure Functions Core Tools.

[Code and test Azure Functions locally](#)

Create a function triggered by Azure Blob storage

12/13/2017 • 5 min to read • [Edit Online](#)

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



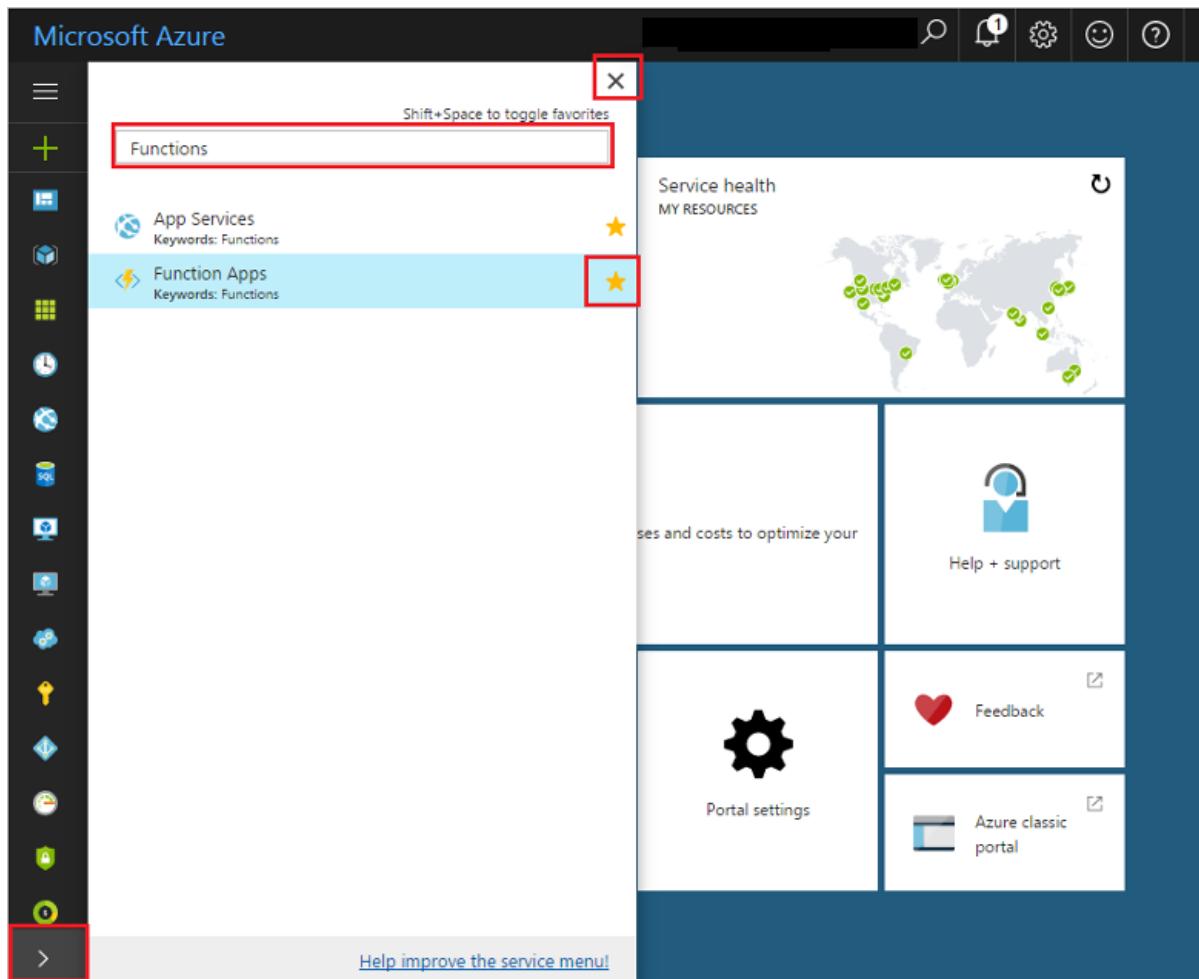
Prerequisites

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

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

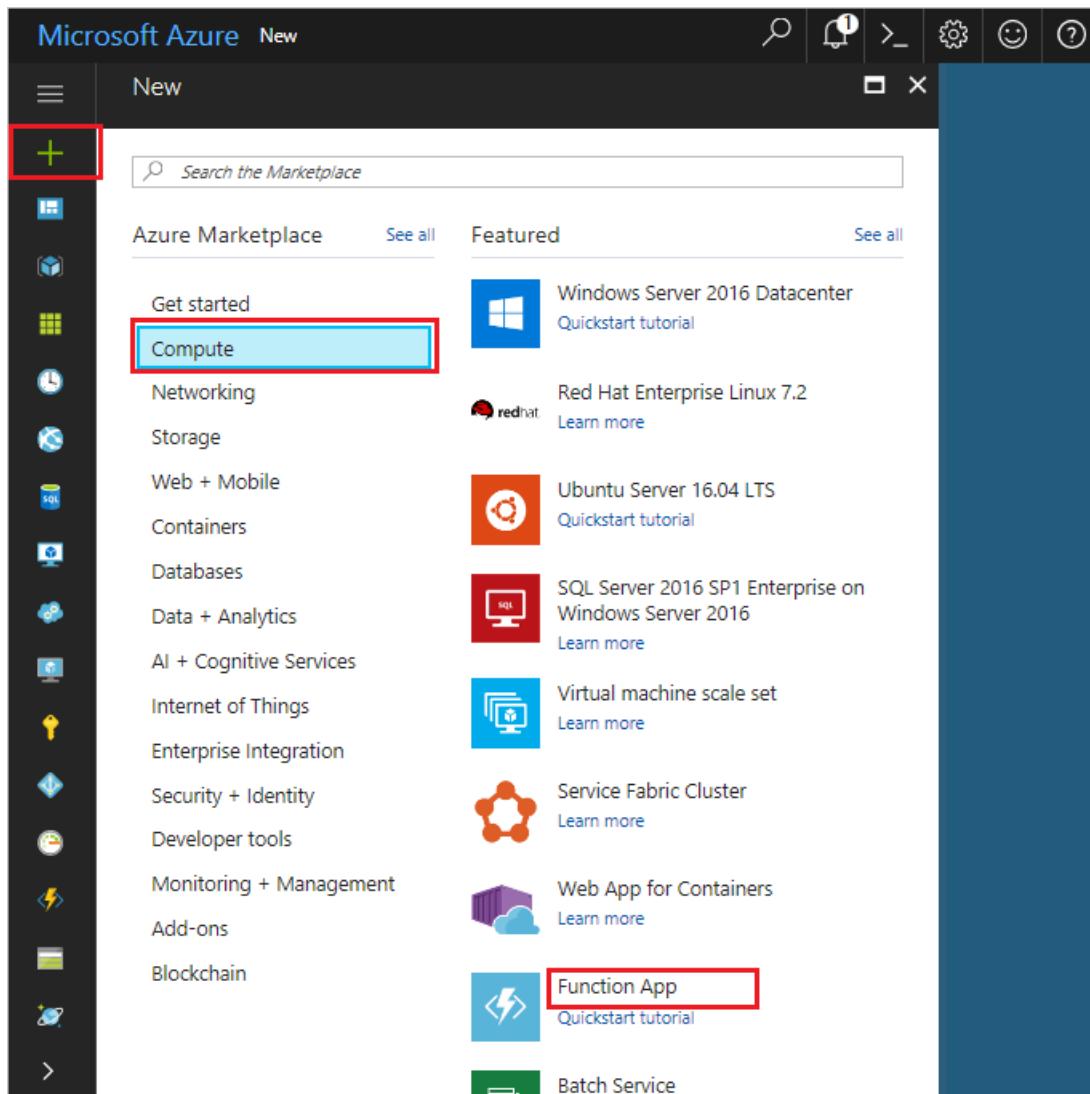
3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

The screenshot shows the 'Function Apps' section of the Azure portal. The left sidebar has a red box around the 'Function Apps' icon. The main area displays a table titled 'Function Apps' with one row. The table columns are 'NAME', 'SUBSCRIPTION ID', 'RESOURCE GROUP', and 'LOCATION'. The single entry is 'functions-ggailey777', 'Visual Studio Enterprise', 'functions-ggailey777', and 'southcentralus' respectively.

NAME	SUBSCRIPTION ID	RESOURCE GROUP	LOCATION
functions-ggailey777	Visual Studio Enterprise	functions-ggailey777	southcentralus

Create an Azure Function app

1. Click the **New** 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
functions-ggailey777 ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group i
 Create new Use existing
functions-ggailey777 ✓

* OS Windows Linux

* Hosting Plan i
Consumption Plan

* Location
West Europe

* Storage i
 Create new Use existing
functionsggaile87e8 ✓

Application Insights i On Off

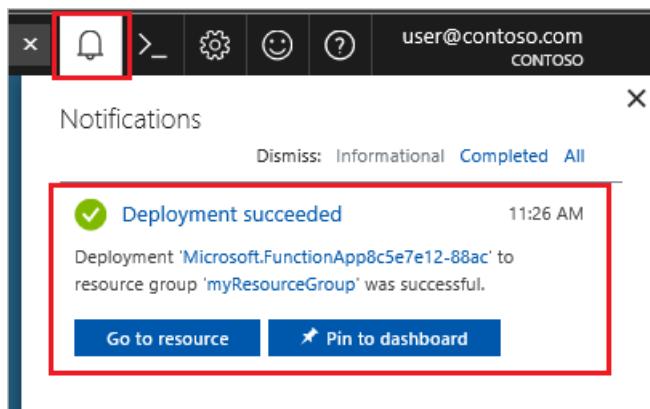
Pin to dashboard

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.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.



Clicking **Go to resource** takes you to your new function app.

The screenshot shows the Azure portal's overview page for the function app "functions-ggaley777". The left sidebar shows the app's name and a list of resources like Visual Studio Enterprise and Function Apps. 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_ggaley777), and URL (https://functions-ggaley777.azurewebsites.net). Below this is a section titled "Configured features" with a note about quick links to features after configuration.

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 **Custom function**. This displays the complete set of function templates.

Microsoft Azure functions-ggailey777

functions-ggailey777

Function Apps

Search

Visual Studio Enterprise

Function Apps

functions-ggailey777

+

+

+

Get started quickly with a premade function

1. Choose a scenario

</> Webhook + API

Timer

Data processing

2. Choose a language

CSharp

JavaScript

FSharp

Java

For PowerShell, Python, and Batch, [create your own custom function](#).

Create this function

Or

Get started on your own

Custom function

Start from source control

2. In the search field, type `blob` and then choose your desired language for the Blob storage trigger template.

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

blob

Language: All Scenario: All

Blob trigger

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

Batch C# F# JavaScript TypeScript

Image resizer

A function that creates resized images whenever a blob is added to a specified container

C# F#

SAS Token Generator

A function that generates a SAS token for a given Azure Storage container and blob name

C#

SAS token generator

A function that generates a SAS token for a given Azure Storage container and blob name

F# JavaScript

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

Blob trigger

New Function

Language:

Name:

Azure Blob Storage trigger

Path ?

Storage account connection ?

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.

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

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

Create the container

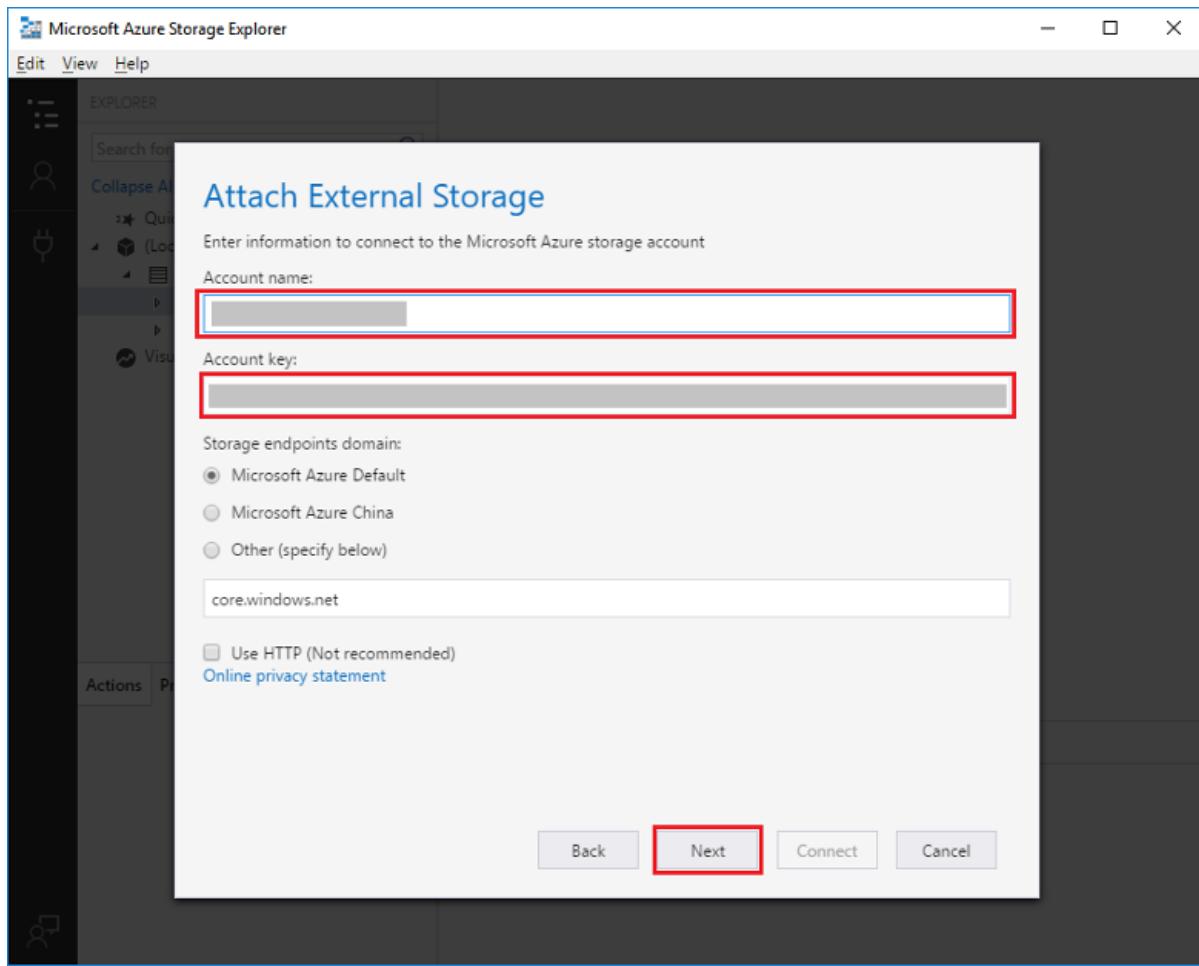
- In your function, click **Integrate**, expand **Documentation**, and copy both **Account name** and **Account key**. You use these credentials to connect to the storage account. If you have already connected your storage account, skip to step 4.

The screenshot shows the Microsoft Azure portal interface for a function app named 'functions-ggailey777 - BlobTriggerCSharp1'. On the left sidebar, there's a 'Triggers' section with 'Azure Blob Storage (myBlob)' selected. Below it, there's an 'Integrate' button which is highlighted with a red box. The main panel shows the configuration for the 'Azure Blob Storage trigger (myBlob)'. It includes fields for 'Blob parameter name' (set to 'myBlob'), 'Path' (set to '/samples-workitems/{name}'), 'Storage account connection' (set to 'AzureWebJobsStorage'), and a 'Cancel' button. At the bottom, there's a 'Documentation' link and a note about connecting to your storage account. The 'Account Name' field contains 'funcosama101' and the 'Account Key' field contains a masked value.

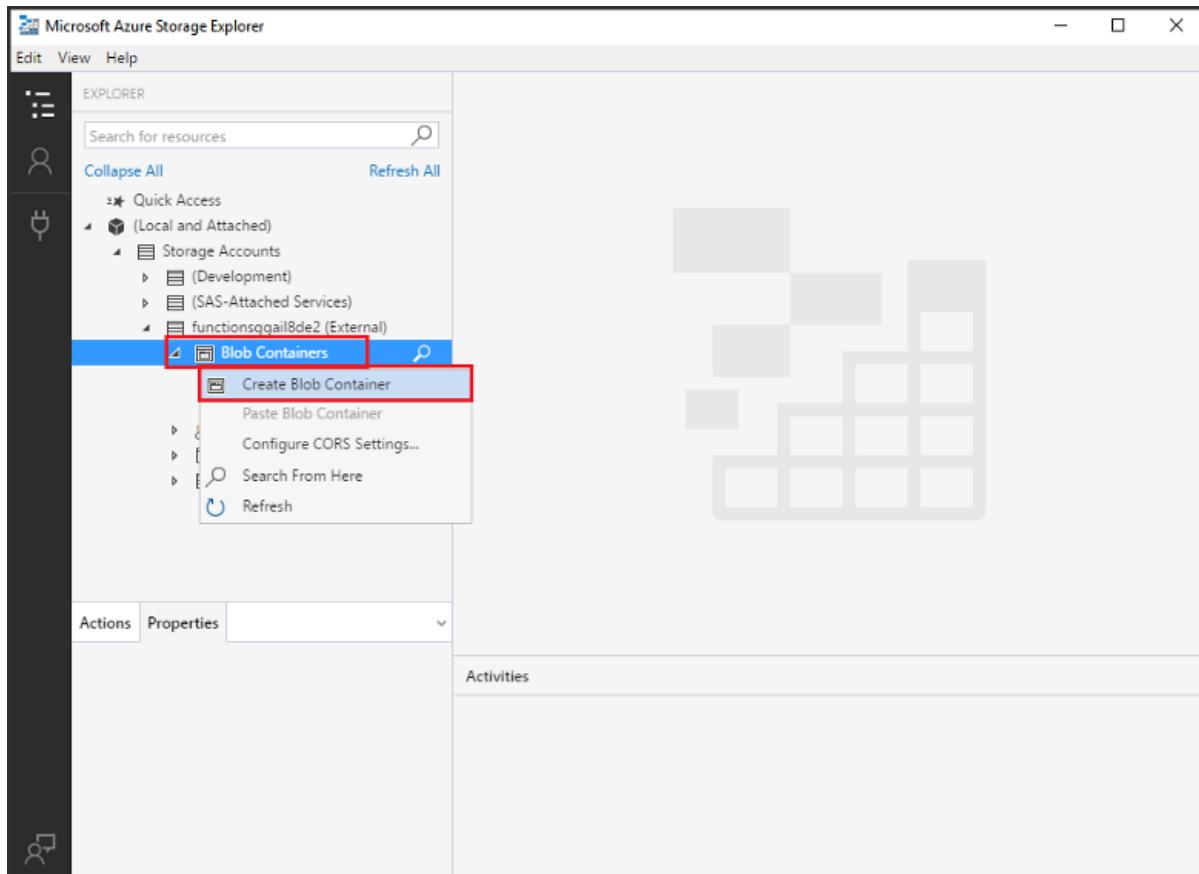
- Run the [Microsoft Azure Storage Explorer](#) tool, click the connect icon on the left, choose **Use a storage account name and key**, and click **Next**.

The screenshot shows the 'Microsoft Azure Storage Explorer' application. On the left, there's a toolbar with a 'Connect' icon, which is highlighted with a red box. The main window displays a 'Connect to Azure Storage' dialog. It asks how to connect to the storage account, with two options: 'Add an Azure Account' and 'Use a shared access signature (SAS) URI or connection string'. Below these, there's a checked radio button for 'Use a storage account name and key', which is also highlighted with a red box. At the bottom of the dialog are 'Next', 'Connect', and 'Cancel' buttons, with 'Next' being highlighted with a red box.

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



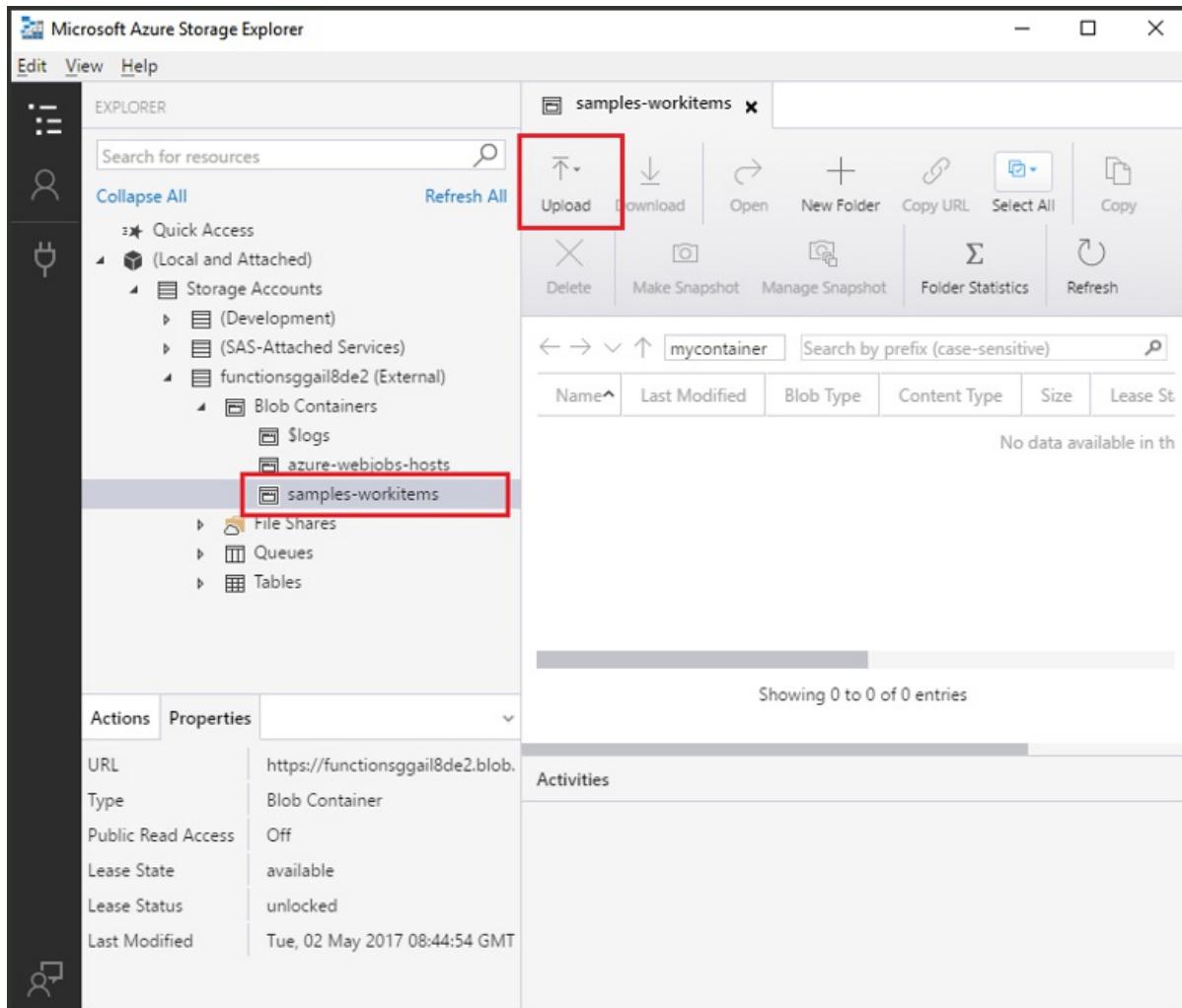
4. Expand the attached storage account, right-click **Blob containers**, click **Create blob container**, type `samples-workitems`, and then press enter.



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

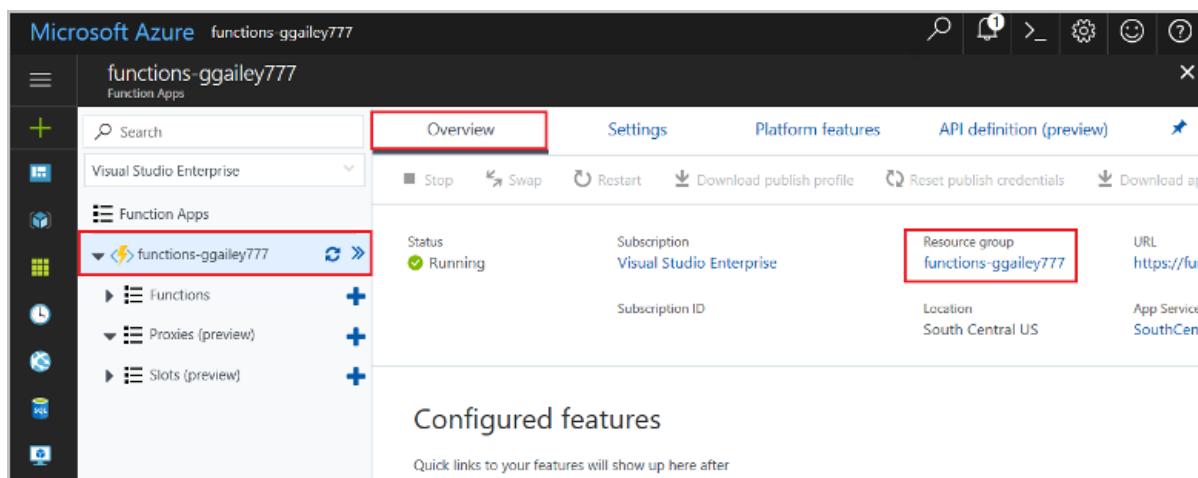
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

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' with the value 'functions-ggailey777', which is also highlighted with a red box. The left sidebar shows a tree view with 'functions-ggailey777' expanded, revealing 'Functions', 'Proxies (preview)', and 'Slots (preview)' under it, each with a plus sign icon.

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.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)

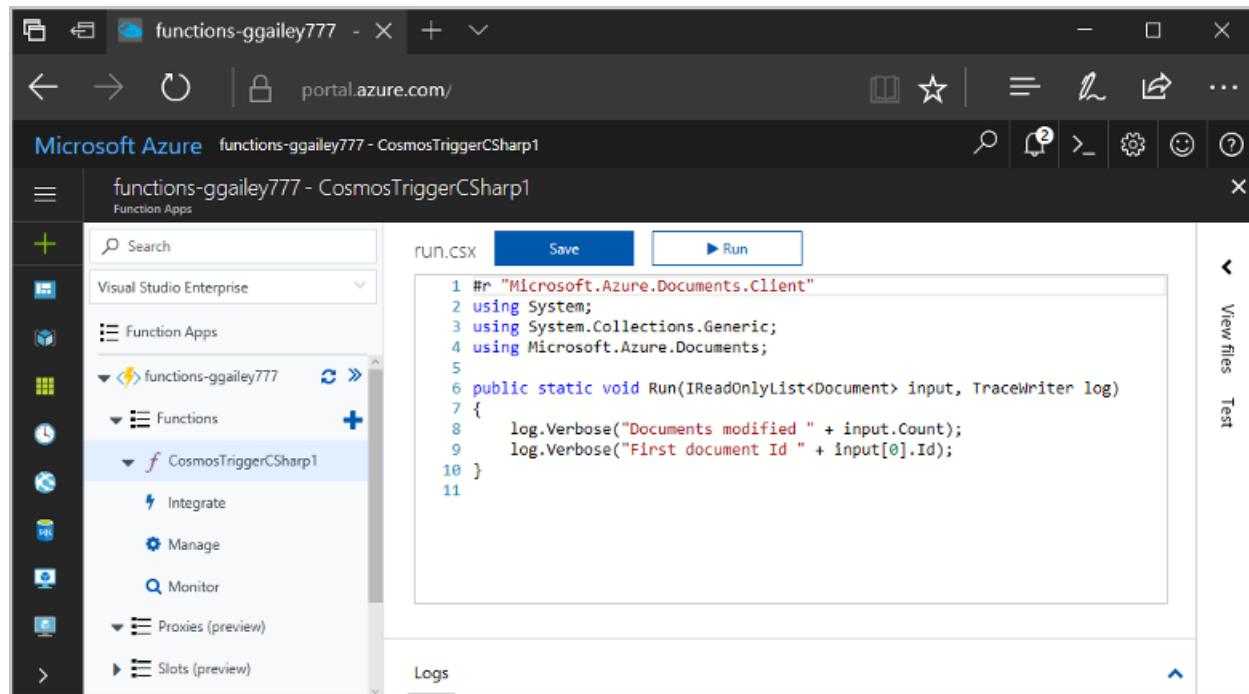
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information about Blob storage triggers, see [Azure Functions Blob storage bindings](#).

Create a function triggered by Azure Cosmos DB

1/10/2018 • 7 min 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

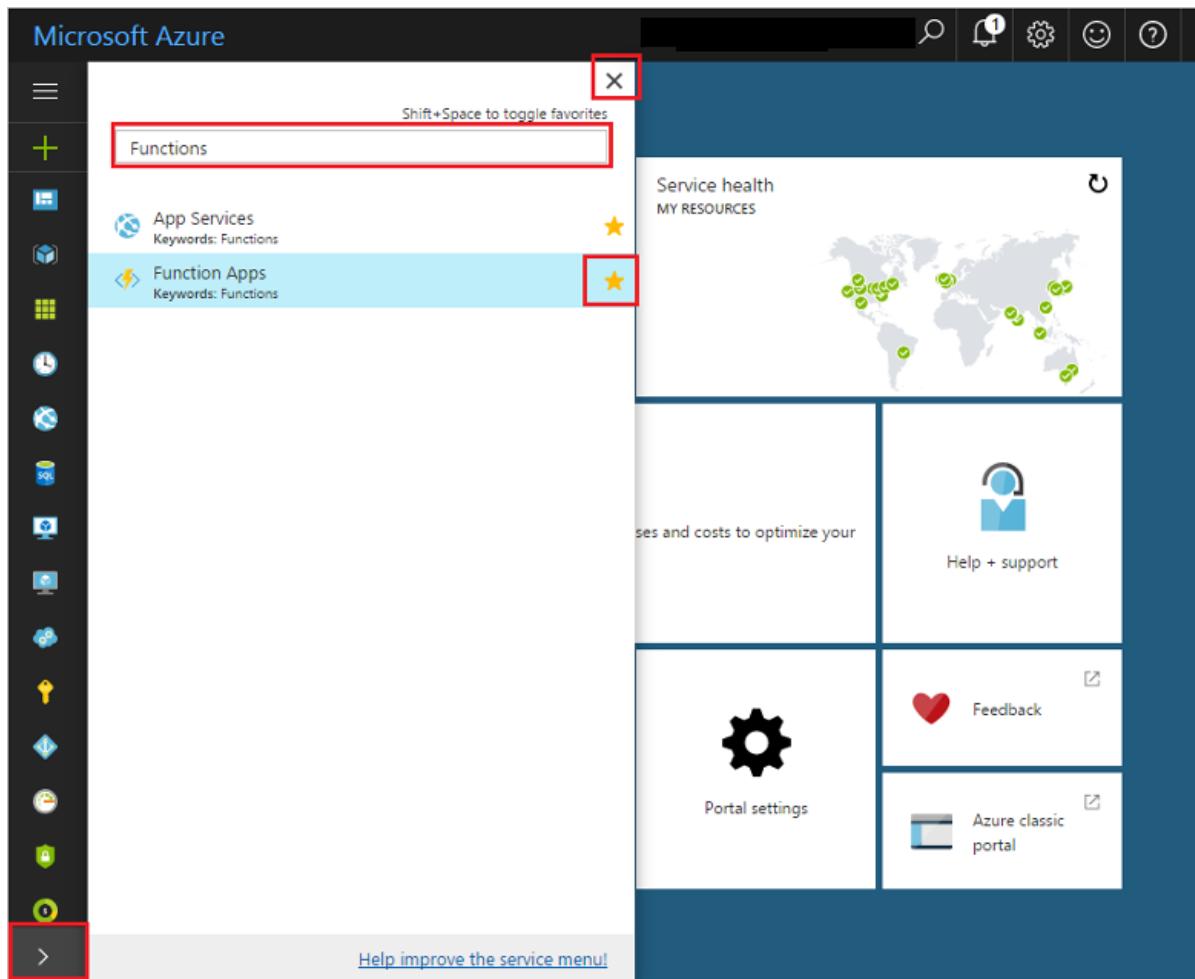
To complete this tutorial:

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

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

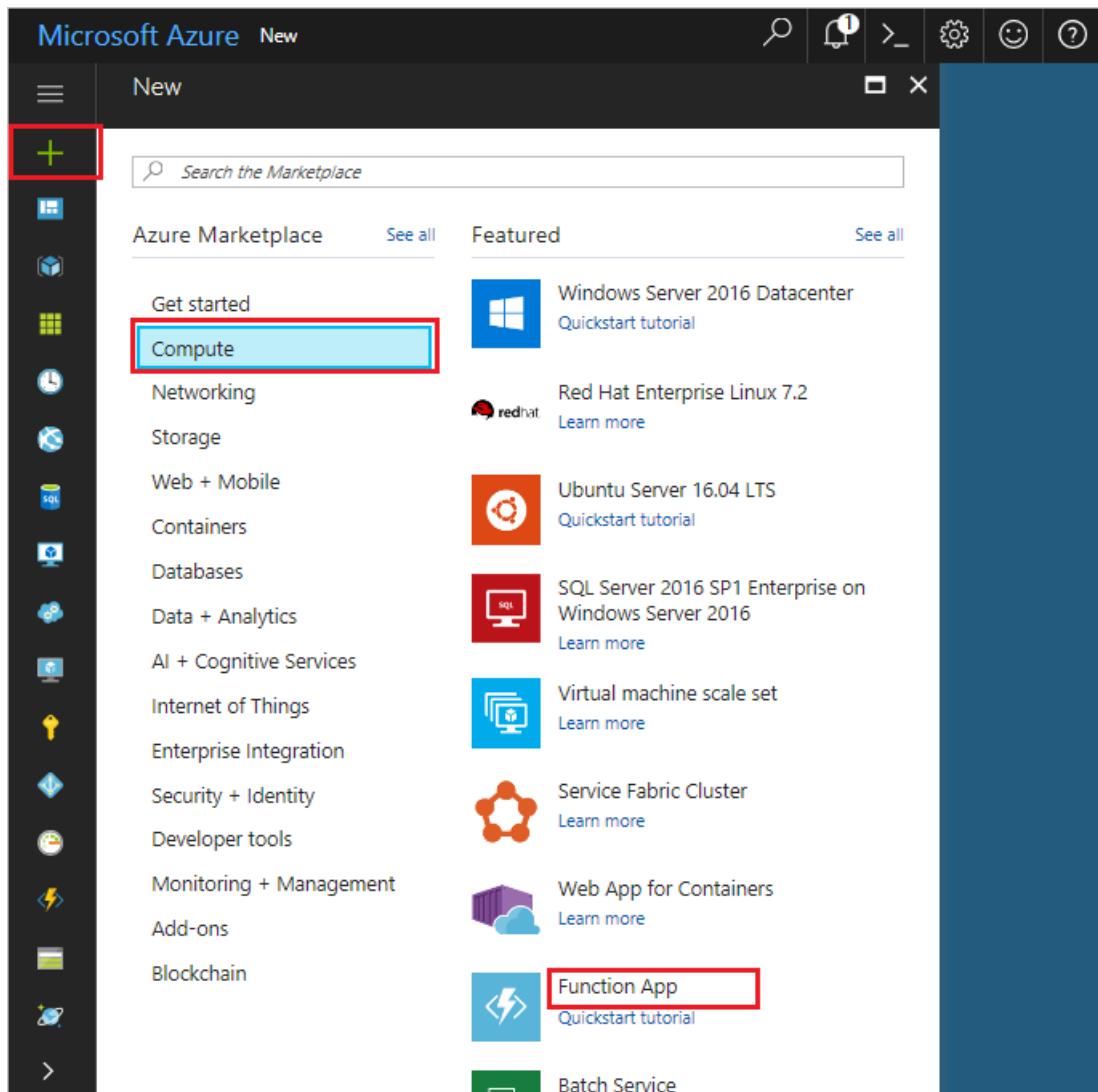
3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

The screenshot shows the 'Function Apps' blade in the Microsoft Azure portal. The left sidebar has a red box around the 'Function Apps' icon. The main area displays a table of function apps:

NAME	SUBSCRIPTION ID	RESOURCE GROUP	LOCATION
functions-ggailey777	Visual Studio Enterprise	functions-ggailey777	southcentralus

Create an Azure Function app

1. Click the **New** 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
functions-ggailey777 .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
functions-ggailey777

* OS Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
West Europe

* Storage ⓘ
 Create new Use existing
functionsggaile87e8

Application Insights ⓘ On Off

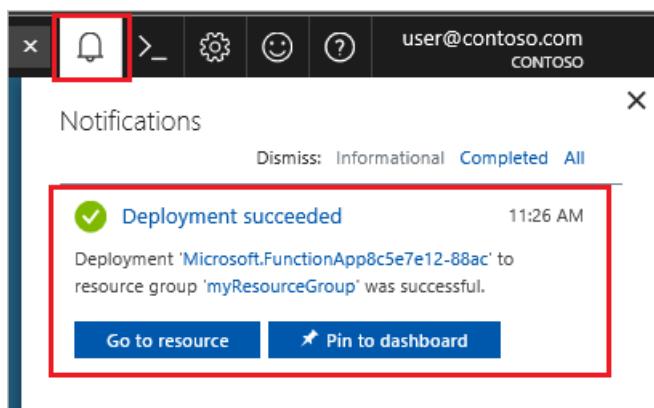
Pin to dashboard

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.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.

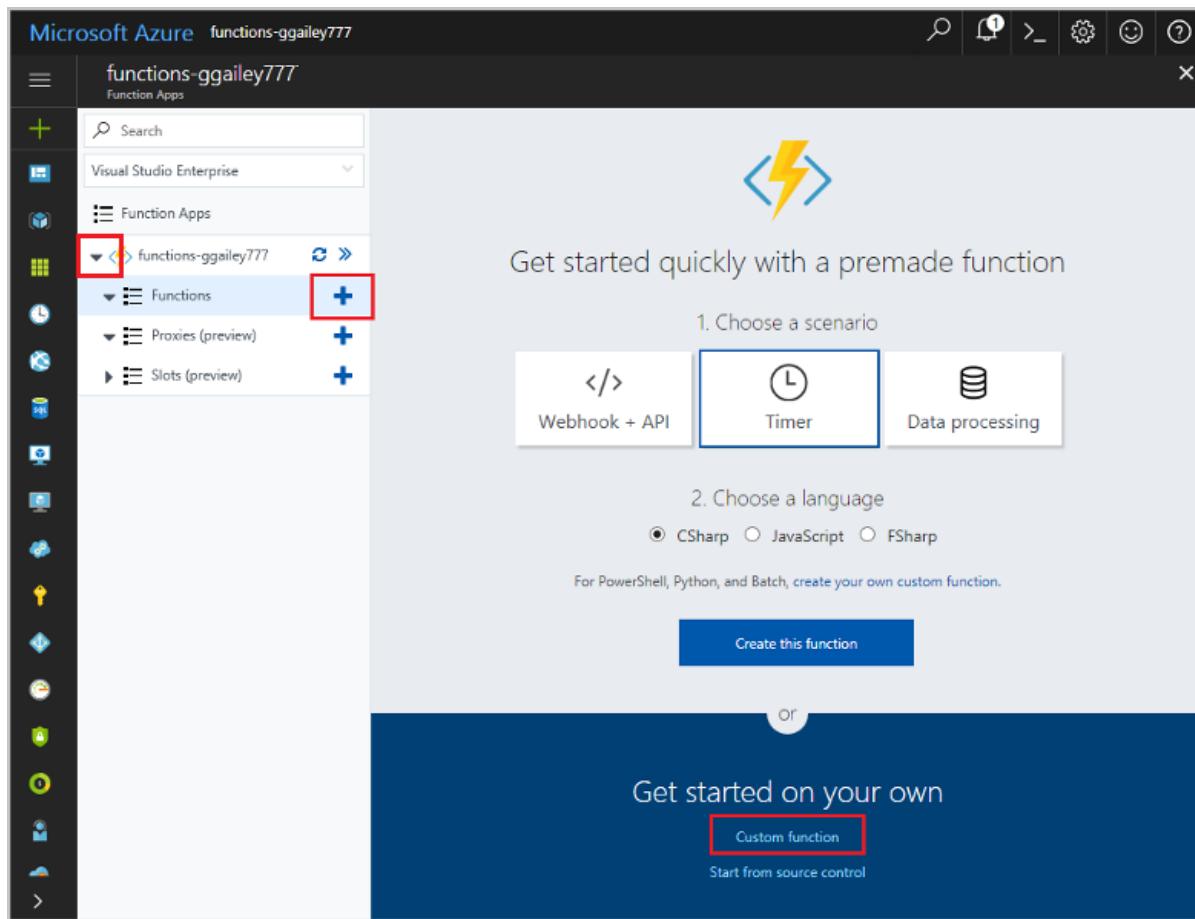


Clicking **Go to resource** takes you to your new function app.

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 **Custom function**. This displays the complete set of function templates.



2. In the search field, type `cosmos` and then choose your desired language for the Azure Cosmos DB trigger template.

Choose a template below or go to the quickstart

Language: All Scenario: All

Cosmos DB trigger

A function that will be run whenever documents change in a document collection

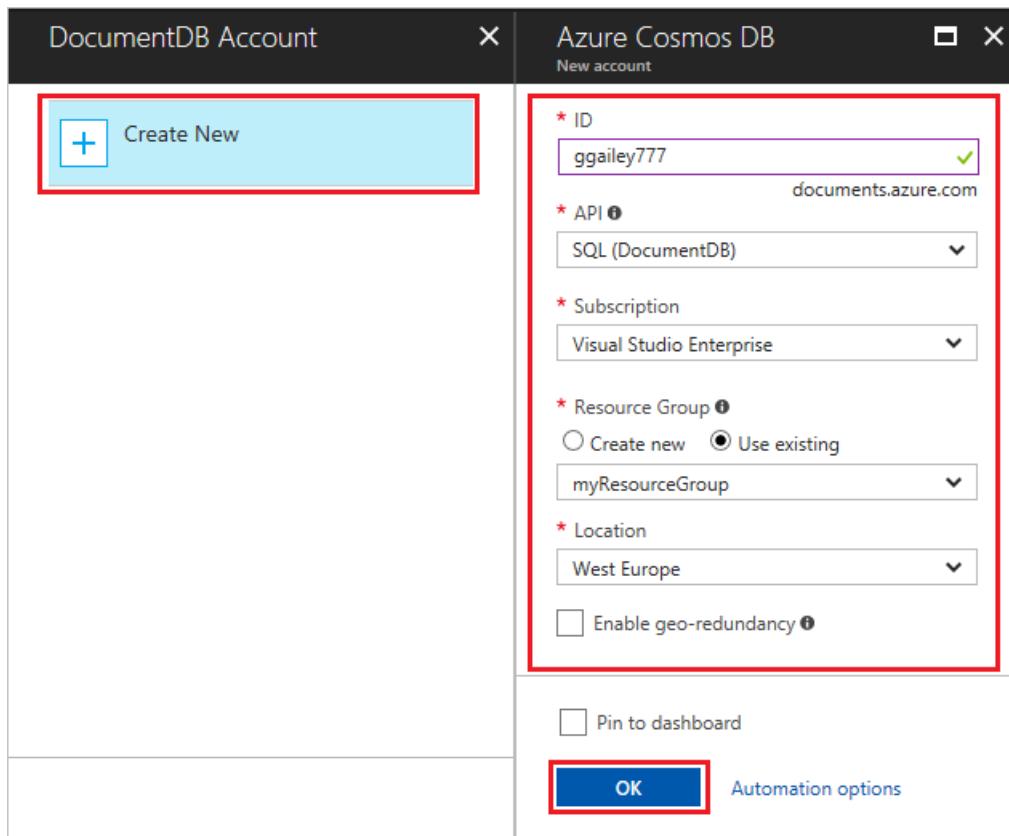
C# JavaScript

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

The screenshot shows the 'New Function' dialog for creating a function with a 'Cosmos DB trigger'. The 'Language' is set to 'JavaScript'. The 'Name' field contains 'CosmosTriggerJavascript1'. A red box highlights the configuration section for the 'Azure Cosmos DB trigger': 'Azure Cosmos DB account connection' (set to 'ggailey777-cosmosdb_DOCUMENTDB'), 'Collection name' (set to 'Items'), and 'Create lease collection if it does not exist' (checked). Another red box highlights the 'Database name' field (set to 'Tasks'). Below these fields is the 'Collection name for leases' field (set to 'leases'). At the bottom are 'Create' and 'Cancel' buttons, with 'Create' also highlighted by a red box.

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Default	Use the default function name suggested by the template.
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.

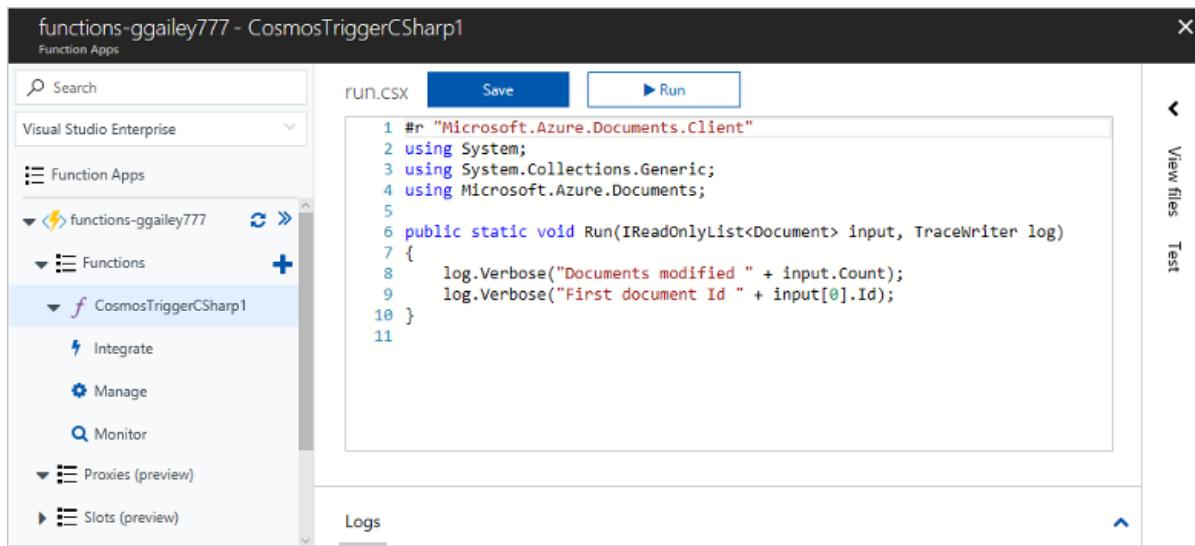
- Select **New** next to the **Azure Cosmos DB account connection** label, and choose an existing Cosmos DB account or **+ Create new**.



5. When creating a new Cosmos DB account, use the **New account** settings as specified in the table.

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	Name of database	Unique ID for the Azure Cosmos DB database
API	SQL (DocumentDB)	This topic uses the document database API.
Subscription	Azure Subscription	Azure Subscription
Resource Group	myResourceGroup	Use the existing resource group that contains your function app.
Location	WestEurope	Select a location near to either your function app or to other apps that use the stored documents.

6. Click **OK** to create the database. It may take a few minutes to create the database. After the database is created, the database connection string is stored as a function app setting. The name of this app setting is inserted in **Azure Cosmos DB account connection**.
7. Click **Create** to create your Azure Cosmos DB triggered function. After the function is created, the template-based function code is displayed.



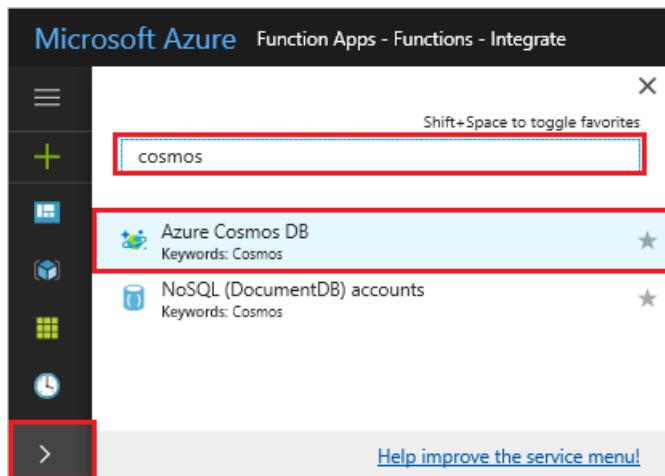
```
1 #r "Microsoft.Azure.Documents.Client"
2 using System;
3 using System.Collections.Generic;
4 using Microsoft.Azure.Documents;
5
6 public static void Run(IReadOnlyList<Document> input, TraceWriter log)
7 {
8     log.Verbose("Documents modified " + input.Count);
9     log.Verbose("First document Id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

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 **Tasks** collection in the 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**.

Microsoft Azure <> ggailey777 - Data Explorer (Preview)

ggailey777 - Data Explorer (Preview)
Azure Cosmos DB account

+ New Collection Delete Database Feedback

COLLECTIONS

taskDatabase leases

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Quick start Data Explorer (Preview)

SETTINGS

Duplicate data globally

A screenshot of the Microsoft Azure Data Explorer (Preview) interface. The top navigation bar shows 'Microsoft Azure' and the current view as 'ggailey777 - Data Explorer (Preview)'. Below the title, it says 'Azure Cosmos DB account'. On the left, there's a sidebar with various icons and links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, and Data Explorer (Preview). The 'Data Explorer (Preview)' link is highlighted with a red box. The main area is titled 'COLLECTIONS' and shows a database named 'taskDatabase' with a single collection called 'leases'. At the top right of the main area, there are buttons for 'New Collection', 'Delete Database', and 'Feedback'. A search bar at the top left contains the placeholder 'Search (Ctrl+Shift+F)'.

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

Add Collection

* Database id Tasks

* Collection Id Items

* Storage capacity Fixed (10 GB) Unlimited

* Throughput (400 - 10,000 RU/s) 400

Partition key /category

OK

A screenshot of the 'Add Collection' dialog box. It has a black header bar with the text 'Add Collection' and a close button. Below the header are four input fields, each with a red border around it. The first field is labeled 'Database id' and contains the value 'Tasks'. The second field is labeled 'Collection Id' and contains the value 'Items'. The third field is labeled 'Storage capacity' and has two options: 'Fixed (10 GB)' (which is selected) and 'Unlimited'. The fourth field is labeled 'Throughput (400 - 10,000 RU/s)' and contains the value '400'. Below these fields is a 'Partition key' input field containing the value '/category'. At the bottom of the dialog is a blue 'OK' button.

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

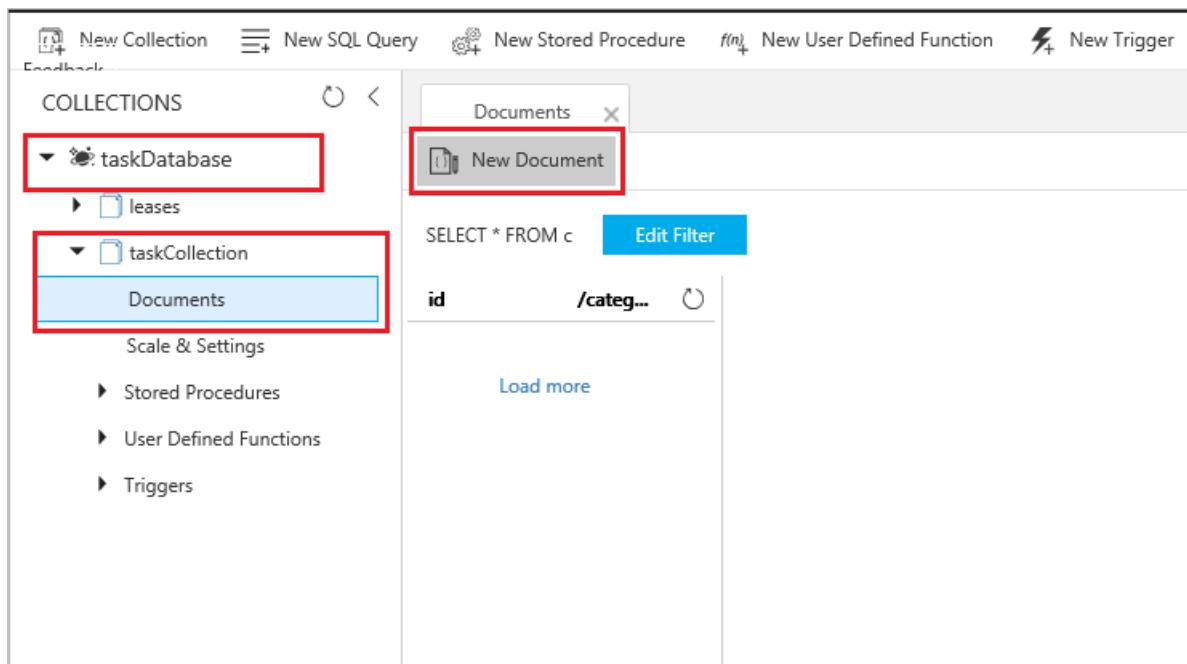
SETTING	SUGGESTED VALUE	DESCRIPTION
Collection ID	Items	The name for the new collection. This must match the name defined in your function binding.
Storage capacity	Fixed (10 GB)	Use the default value. This value is the storage capacity of the database.
Throughput	400 RU	Use the default value. If you 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.

6. Click **OK** to create the **Tasks** 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

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

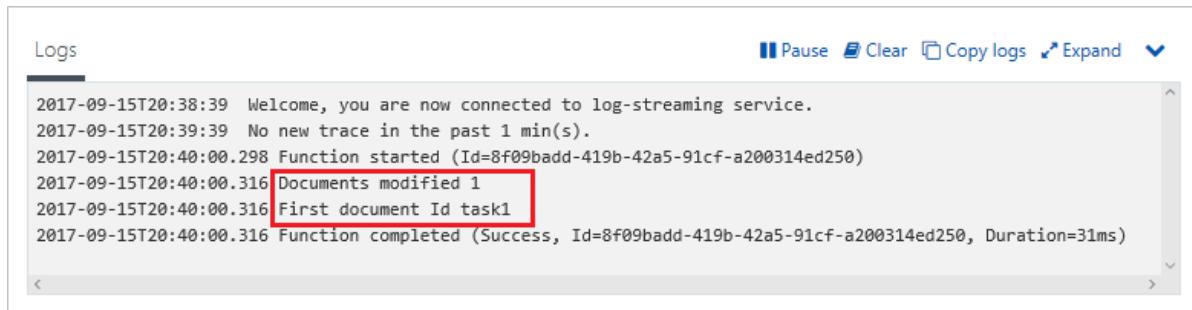


2. Replace the contents of the new document with the following content, then choose **Save**.

```
{
  "id": "task1",
  "category": "general",
  "description": "some task"
}
```

3. Switch to the first browser tab that contains your function in the portal. Expand the function logs and verify that the new document has triggered the function. See that the `task1` document ID value is written to the

logs.



```
Logs
```

2017-09-15T20:38:39 Welcome, you are now connected to log-streaming service.
2017-09-15T20:39:39 No new trace in the past 1 min(s).
2017-09-15T20:40:00.298 Function started (Id=8f09badd-419b-42a5-91cf-a200314ed250)
2017-09-15T20:40:00.316 Documents modified 1
2017-09-15T20:40:00.316 First document Id task1
2017-09-15T20:40:00.316 Function completed (Success, Id=8f09badd-419b-42a5-91cf-a200314ed250, Duration=31ms)

4. (Optional) Go back to your document, make a change, and click **Update**. Then, go back to the function logs and verify that the update has also triggered the function.

Clean up resources

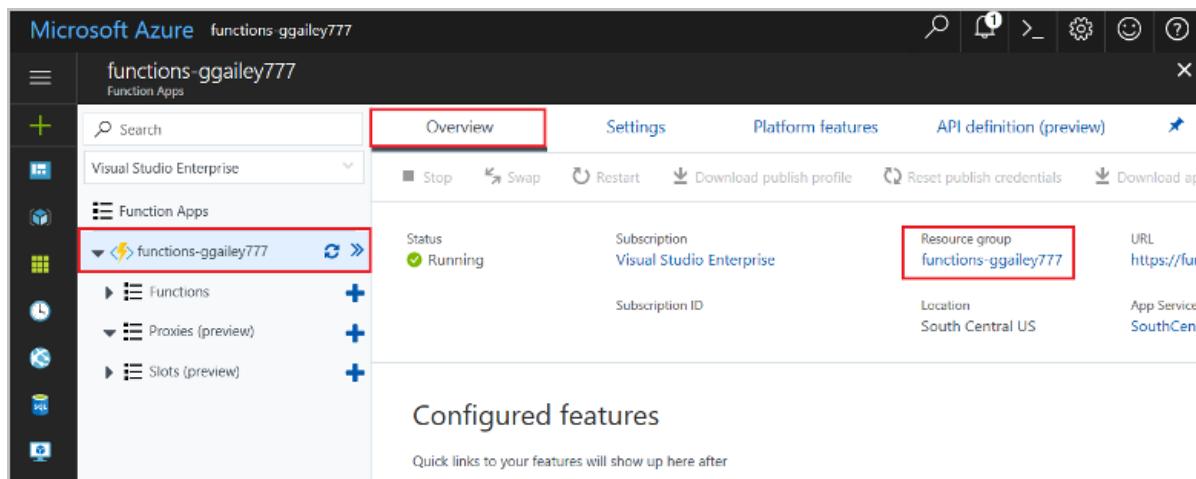
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

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**.



Microsoft Azure functions-ggailey777

functions-ggailey777

Function Apps

Visual Studio Enterprise

Overview

Status: Running

Subscription: Visual Studio Enterprise

Resource group: functions-ggailey777

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.

Next steps

You have created a function that runs when a document is added or modified in your Azure Cosmos DB.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

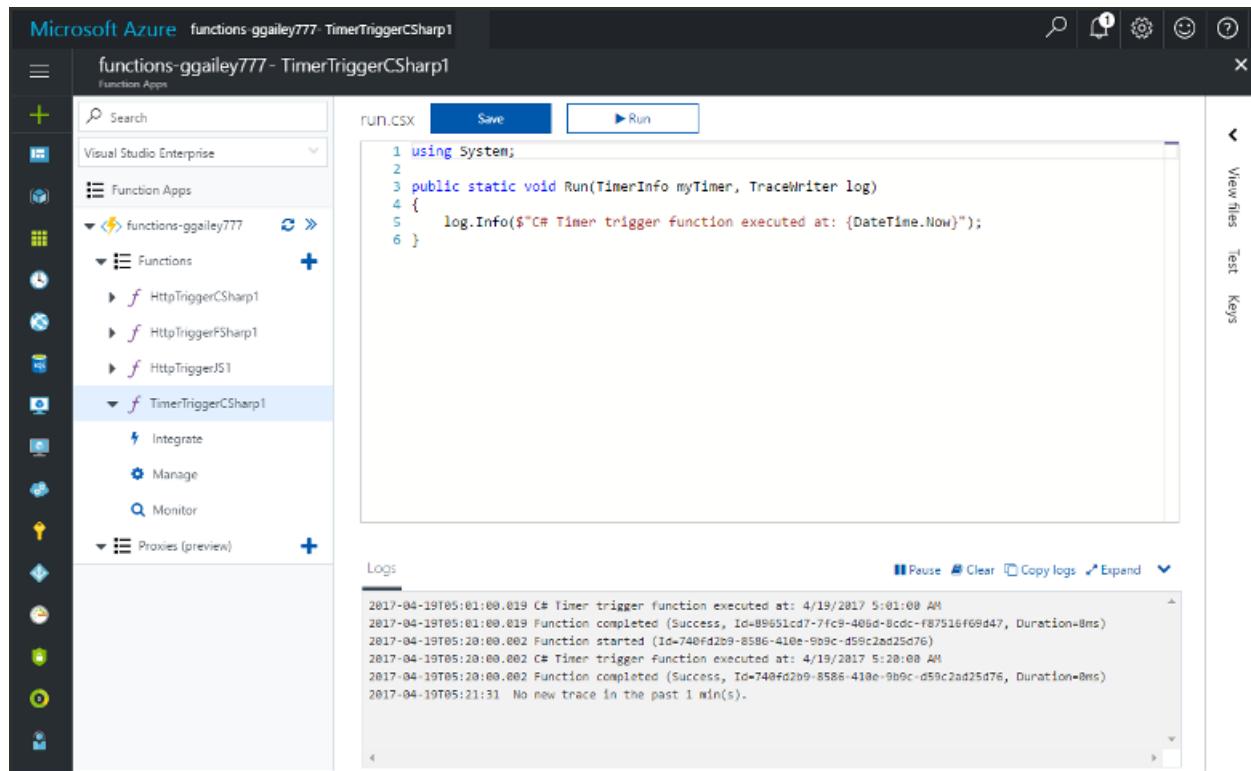
- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information about Azure Cosmos DB triggers, see [Azure Cosmos DB bindings for Azure Functions](#).

Create a function in Azure that is triggered by a timer

1/4/2018 • 4 min to read • [Edit Online](#)

Learn how to use Azure Functions to create a **serverless** function that runs based a schedule that you define.



Prerequisites

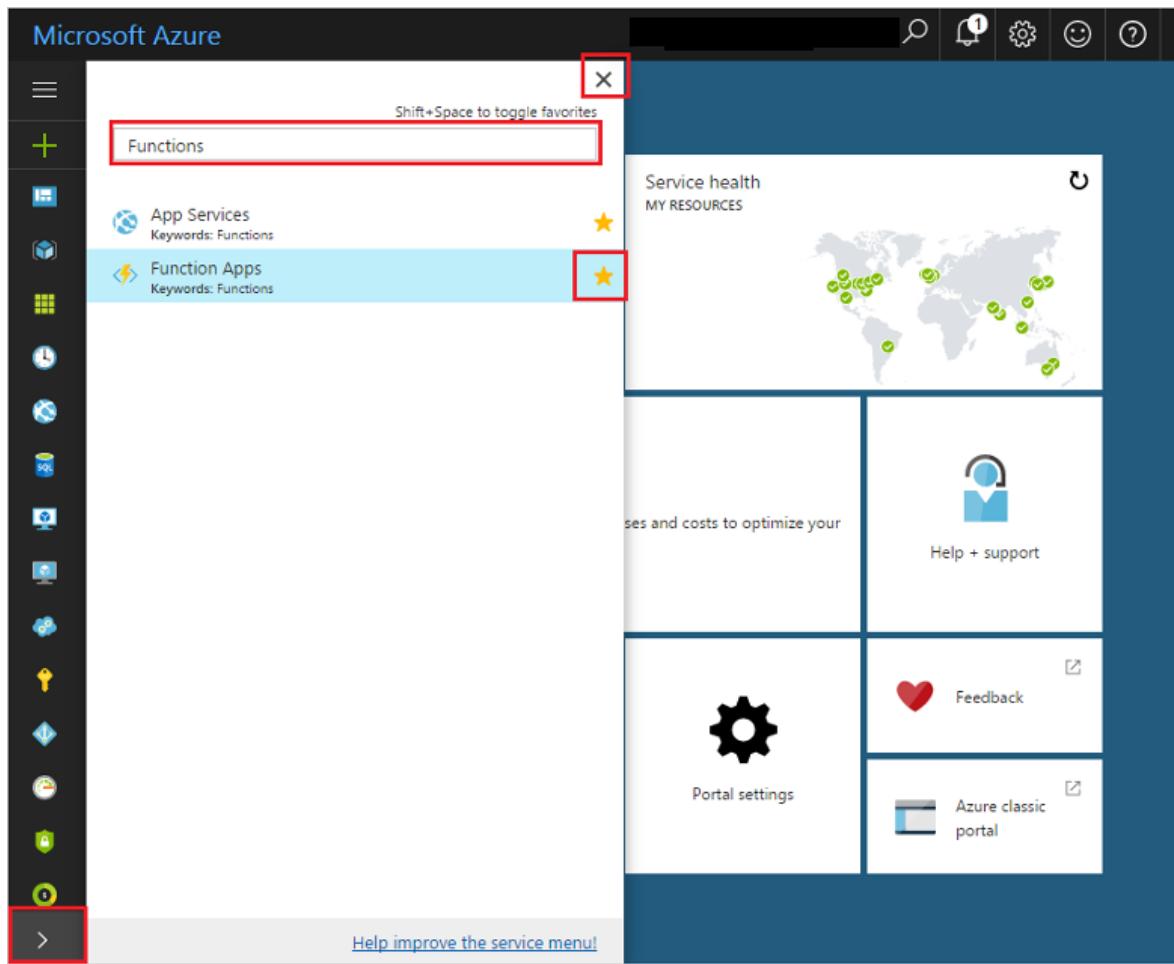
To complete this tutorial:

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

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



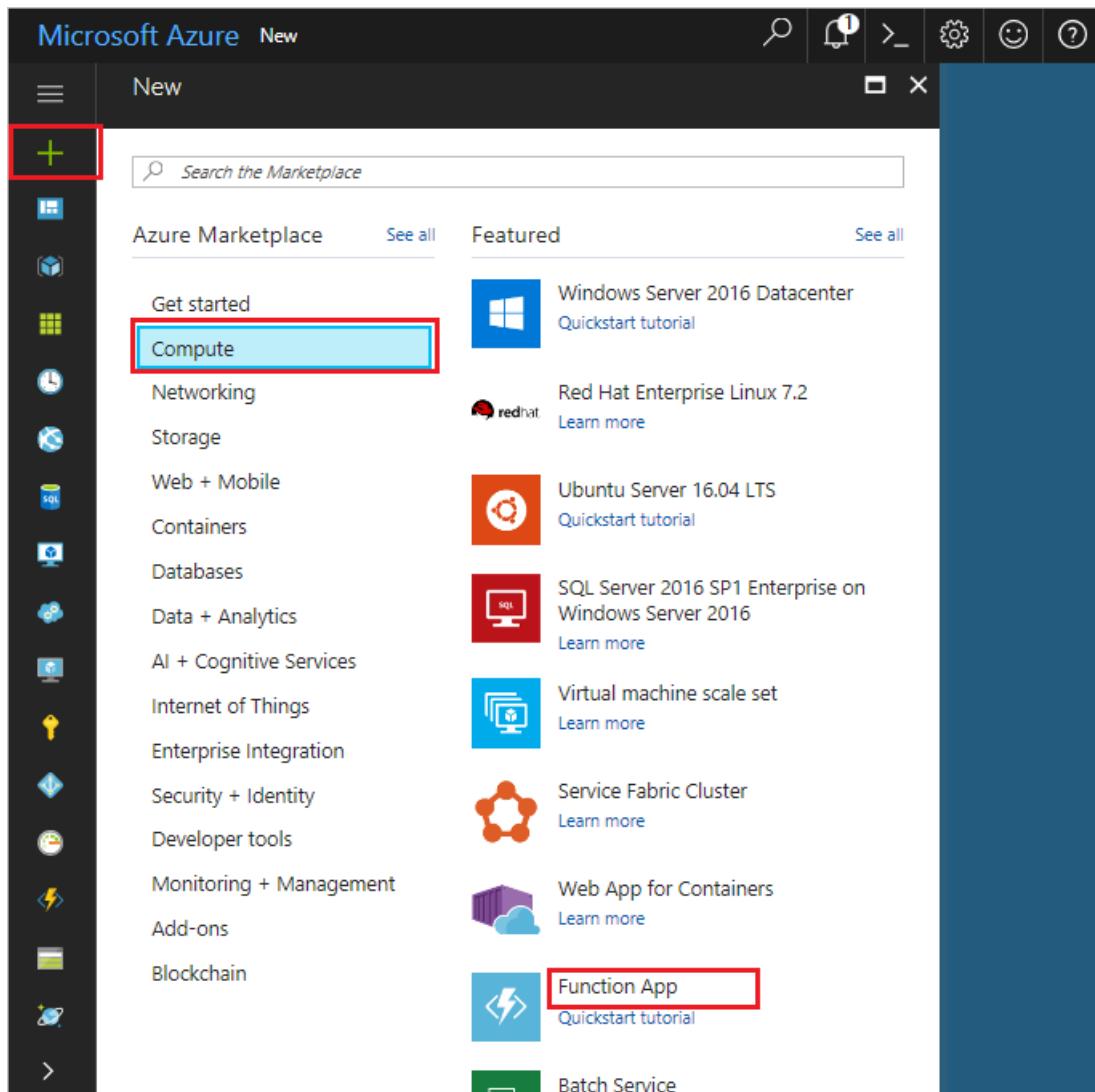
This adds the Functions icon to the menu on the left of the portal.

3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

A screenshot of the Microsoft Azure Function Apps blade. The left sidebar shows a tree view with 'Function Apps' selected, and a red box highlights the 'Function Apps' icon. The main content area is titled 'Function Apps' and displays a table of function apps. The table has columns for NAME, SUBSCRIPTION ID, RESOURCE GROUP, and LOCATION. One row is shown, with the values: 'functions-ggailey777', 'Visual Studio Enterprise', 'functions-ggailey777', and 'southcentralus'. The top right corner of the blade has a red box highlighting the 'X' button.

Create an Azure Function app

1. Click the **New** 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 X

Create

* App name
functions-ggailey777 ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group i
 Create new Use existing
functions-ggailey777 ✓

* OS Windows Linux

* Hosting Plan i
Consumption Plan

* Location
West Europe

* Storage i
 Create new Use existing
functionsggaile87e8 ✓

Application Insights i On Off

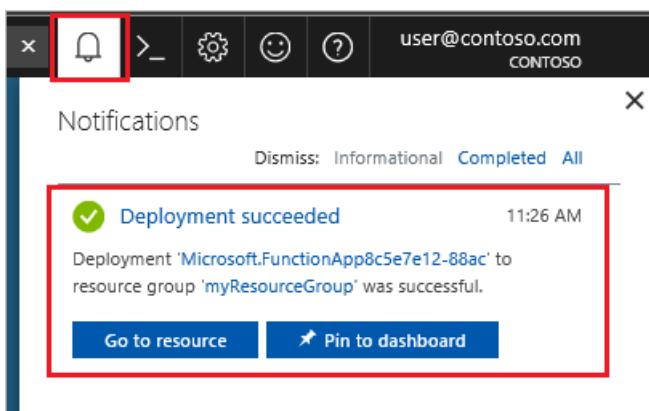
Pin to dashboard

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.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.



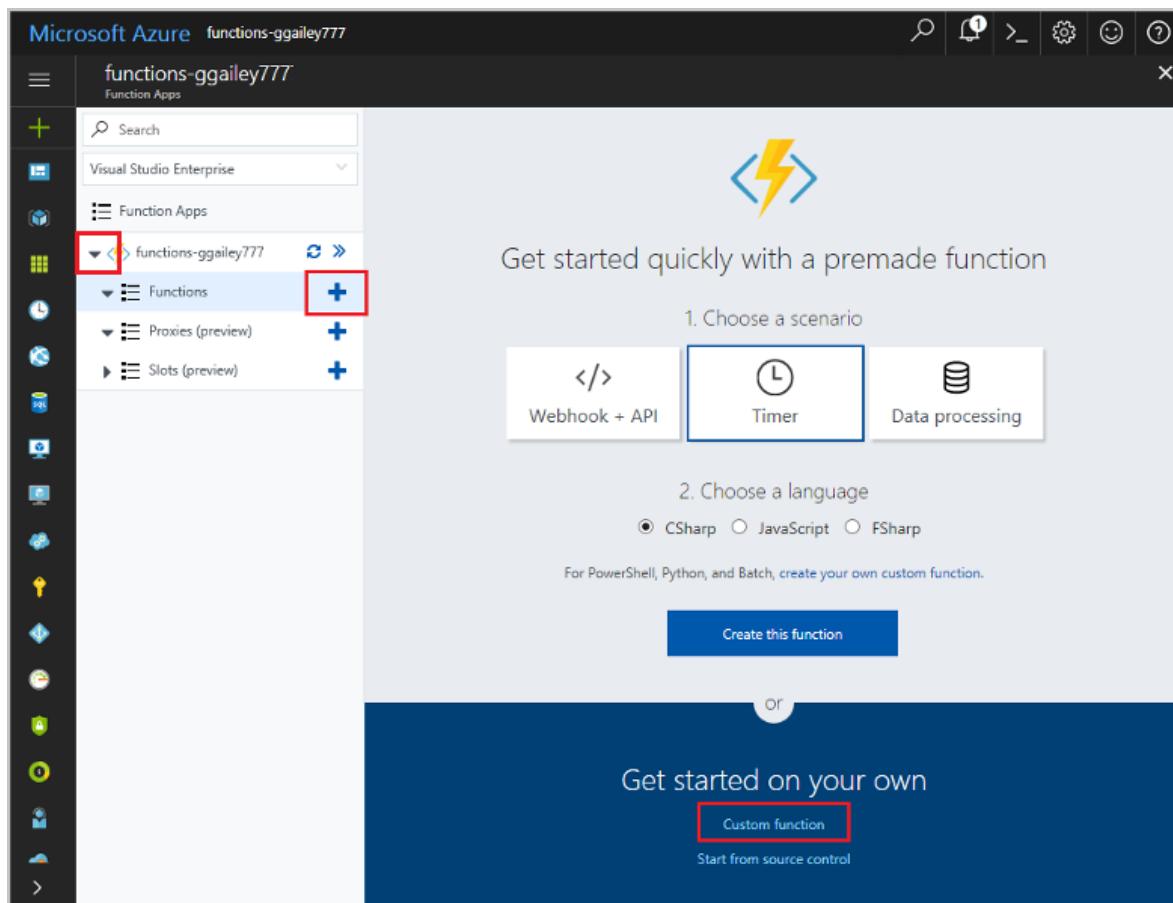
Clicking **Go to resource** takes you to your new function app.

The screenshot shows the Azure portal's overview page for the function app "functions-ggaley777". The left sidebar shows the app's name and a list of features like Functions and Proxies. The main pane displays the "Overview" tab with details such as Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggaley777), URL (https://functions-ggaley777.azurewebsites.net), and Location (South Central US). Below this, a section titled "Configured features" provides quick links to platform features.

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 **Custom function**. This displays the complete set of function templates.



2. In the search field, type `timer` and then choose your desired language for the timer trigger template.

This screenshot shows a search interface for function templates. At the top, a search bar contains the text 'timer' with a red box around it. Below the search bar are dropdown menus for 'Language: All' and 'Scenario: All'. The main area features a card for the 'Timer trigger' template, which includes a clock icon, the name 'Timer trigger', and a description: 'A function that will be run on a specified schedule'. At the bottom of this card, there's a horizontal bar with language options: 'C#' (highlighted with a red box), 'F#', 'JavaScript', 'PowerShell', and 'TypeScript'.

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

Timer trigger

New Function

Language:

Name:

Timer trigger

Schedule ?

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.

The screenshot shows the 'Logs' panel with the following log entries:

```

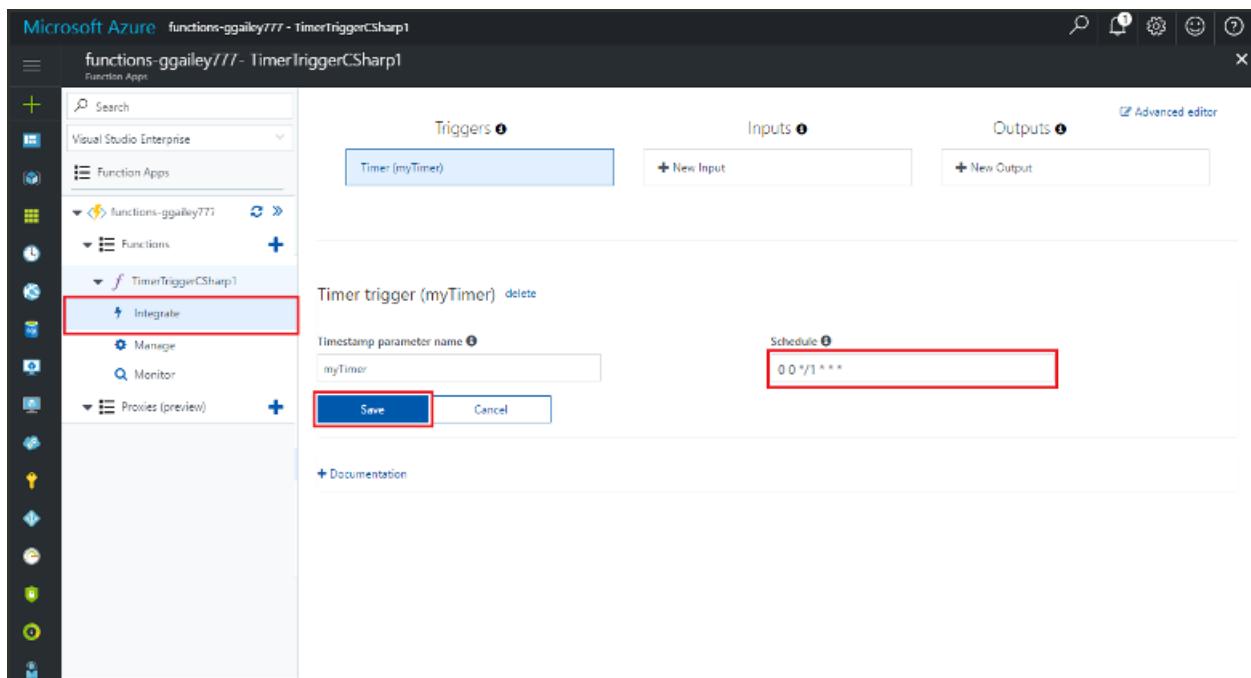
Logs
Pause Clear Copy logs Expand ▾
2017-04-27T17:51:00.189 JavaScript timer trigger function ran! 2017-04-27T17:51:00.189Z
2017-04-27T17:51:00.204 Function completed (Success, Id=a2830e1b-7d62-4ad0-81ce-9e4f857bf175, Duration=185ms)
2017-04-27T17:52:00.688 Function started (Id=c3491caf-28af-4db0-af2e-7698e49da6dc)
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)

```

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**.



You now have a function that runs once every hour.

Clean up resources

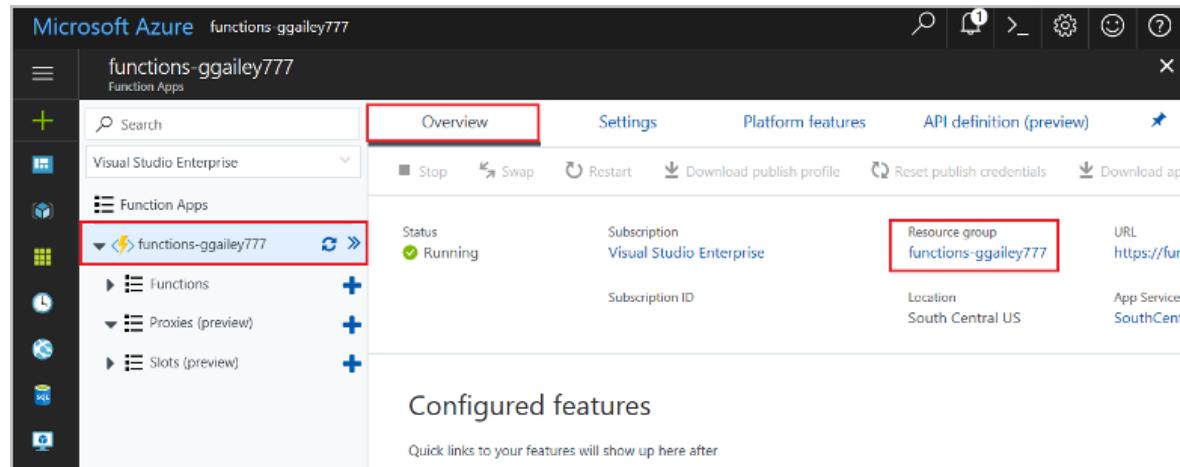
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

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**.



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.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

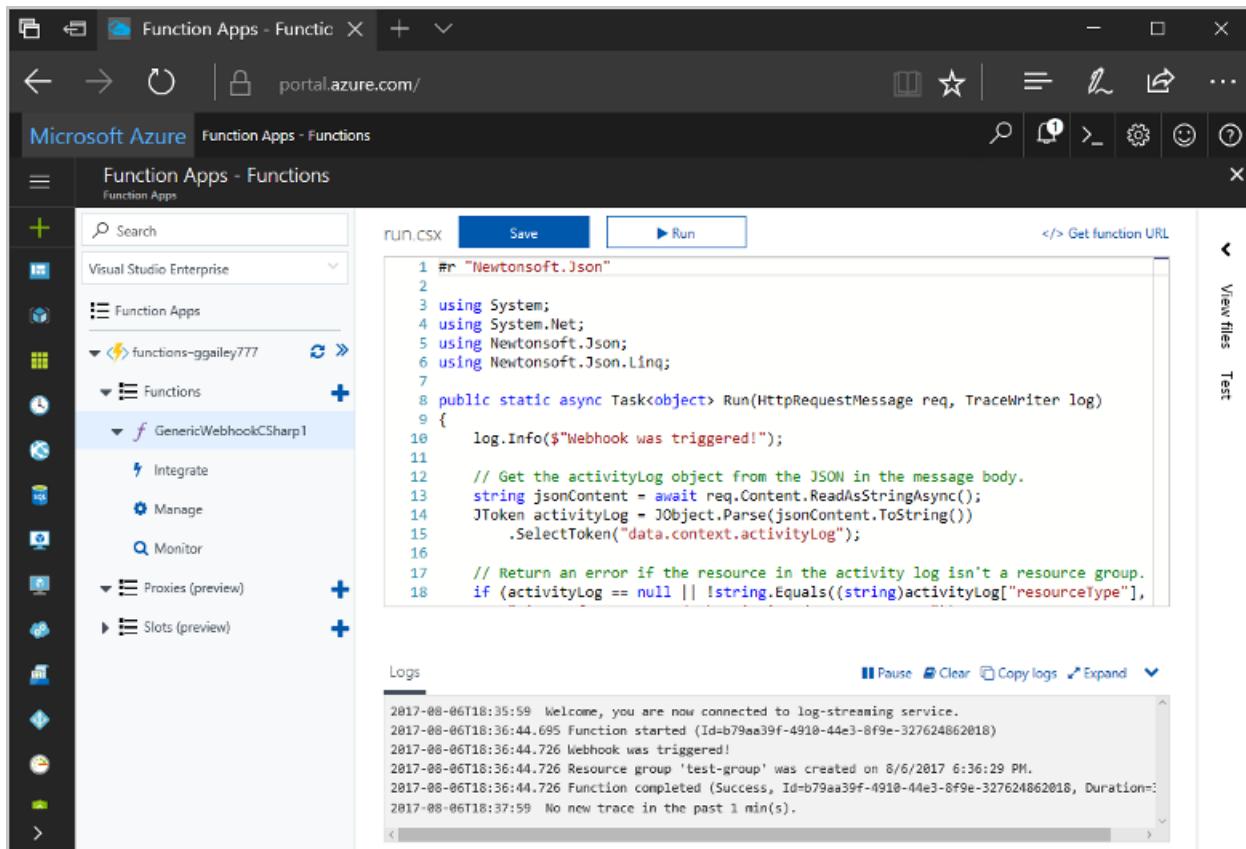
- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information timer triggers, see [Schedule code execution with Azure Functions](#).

Create a function triggered by a generic webhook

12/13/2017 • 7 min 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. For example, you can configure a function to be triggered by an alert raised by Azure Monitor. This topic shows you how to execute C# code when a resource group is added to your subscription.



Prerequisites

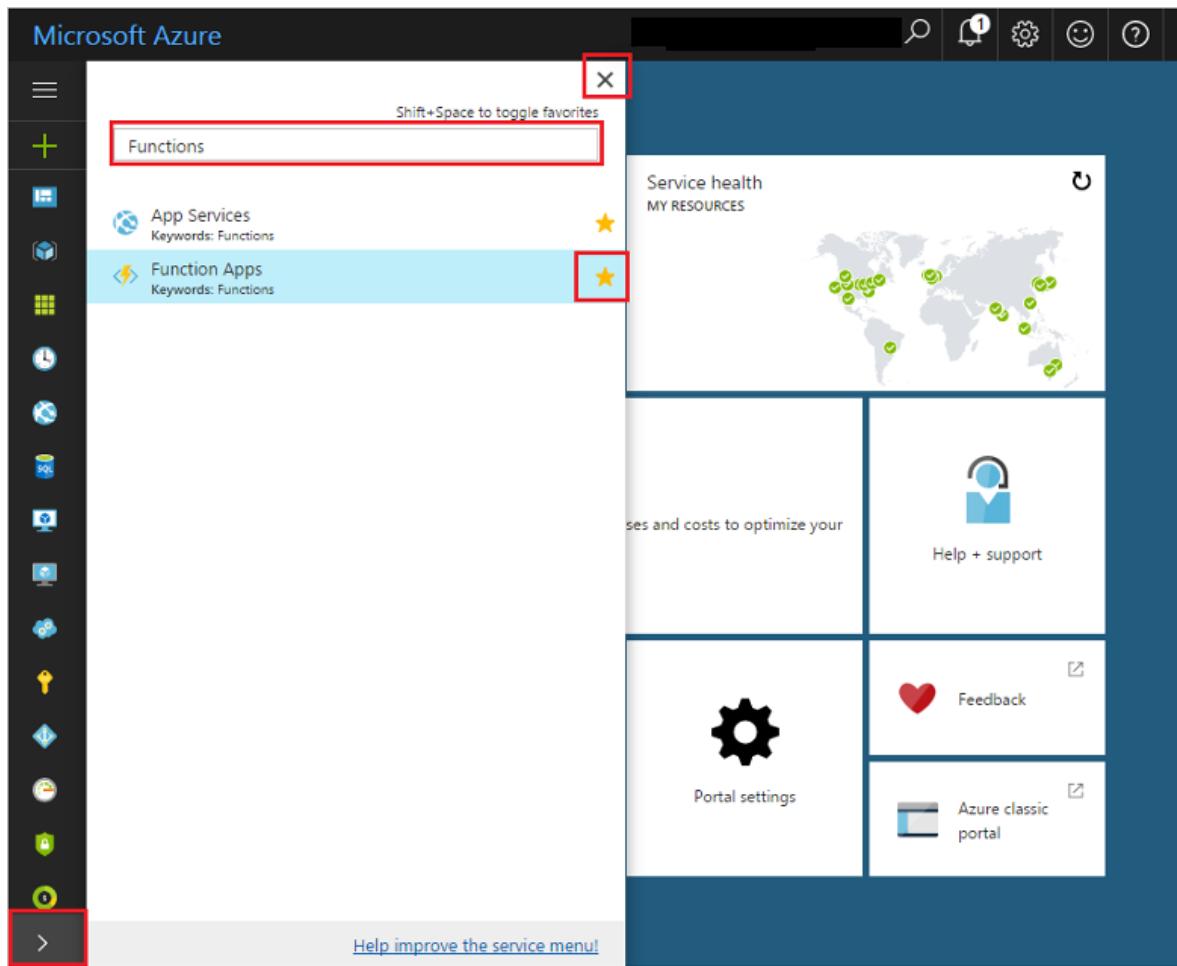
To complete this tutorial:

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

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



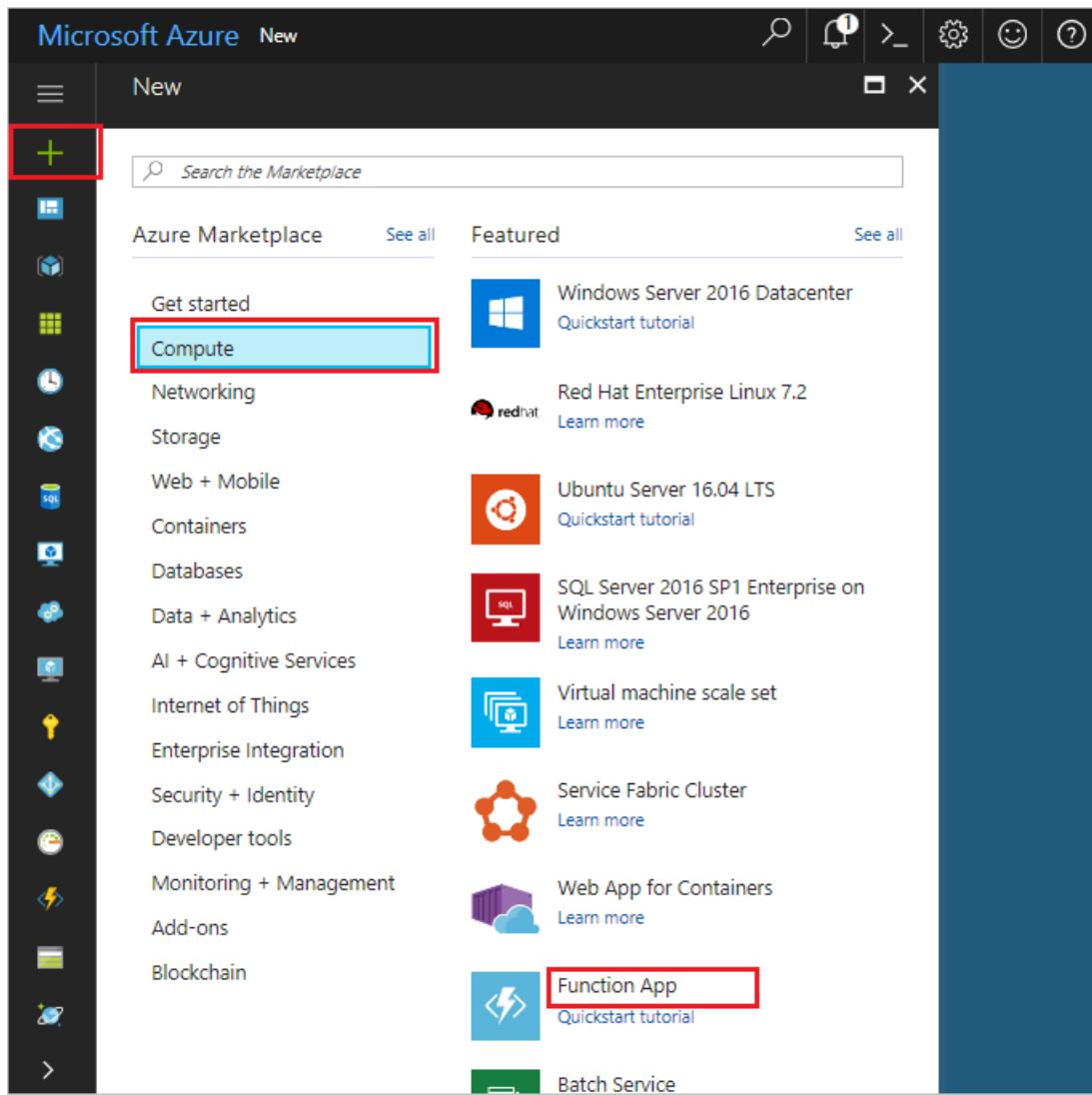
This adds the Functions icon to the menu on the left of the portal.

3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

A screenshot of the Microsoft Azure portal showing the 'Function Apps' blade. The left sidebar has the 'Function Apps' icon highlighted with a red box. The main content area shows a table titled 'Function Apps' with one row. The table has columns for NAME, SUBSCRIPTION ID, RESOURCE GROUP, and LOCATION. The single row contains the values: 'functions-ggailey777', 'Visual Studio Enterprise', 'functions-ggailey777', and 'southcentralus'. The entire screenshot is framed by a red border.

Create an Azure Function app

1. Click the **New** 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 X

Create

* App name
functions-ggailey777 ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group i
 Create new Use existing
functions-ggailey777 ✓

* OS Windows Linux

* Hosting Plan i
Consumption Plan

* Location
West Europe

* Storage i
 Create new Use existing
functions-ggaile87e8 ✓

Application Insights i On Off

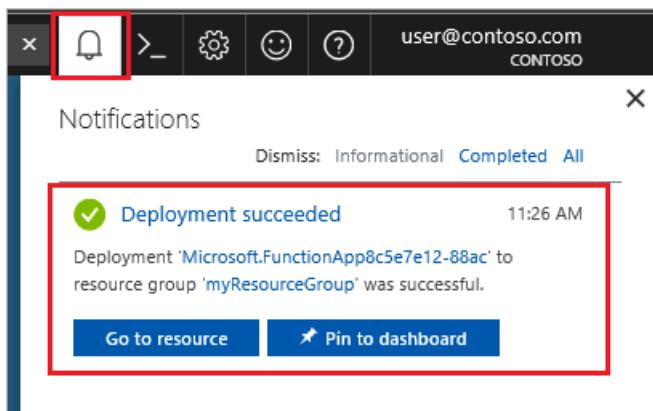
Pin to dashboard

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.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.

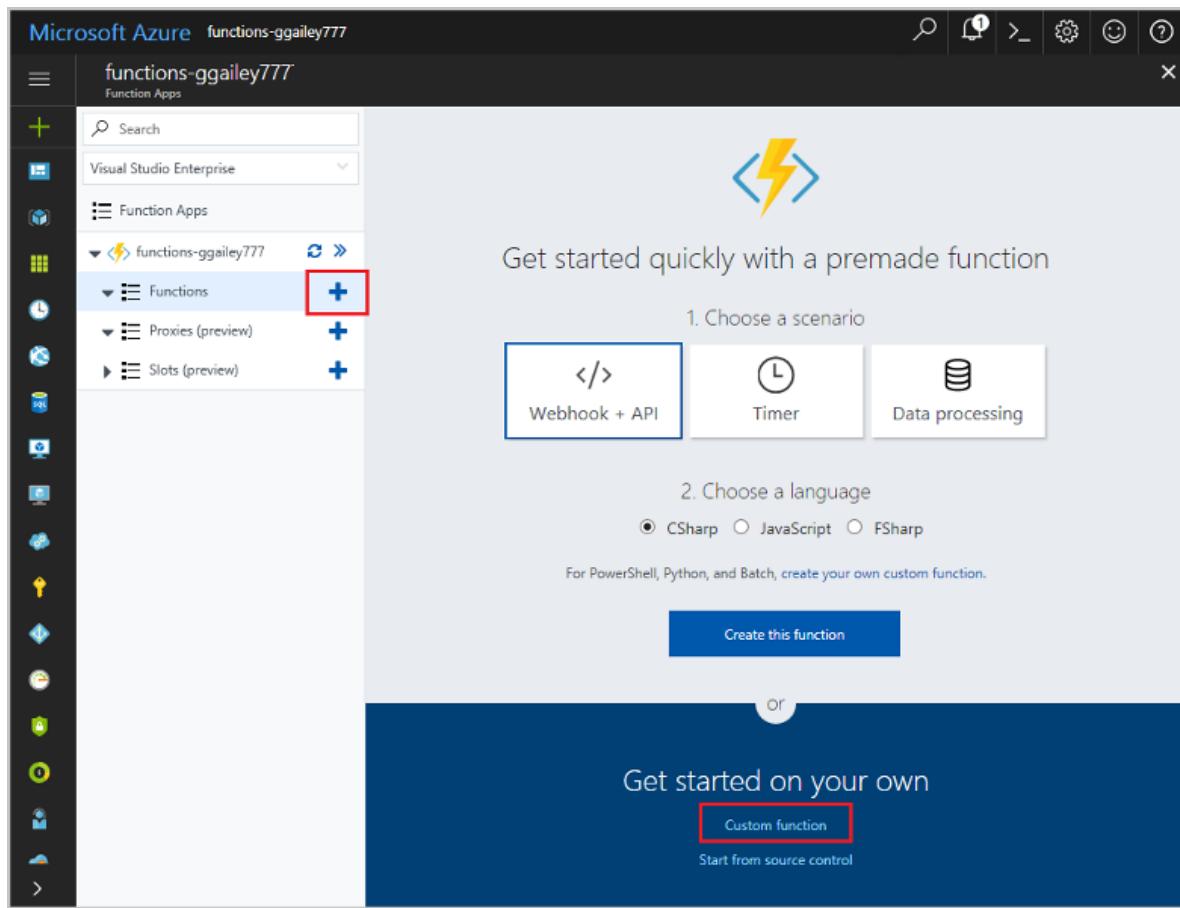


Clicking **Go to resource** takes you to your new function app.

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

Create a generic webhook triggered function

1. Expand your function app and click the + button next to **Functions**. If this function is the first one in your function app, select **Custom function**. This displays the complete set of function templates.



2. In the search field, type `generic` and then choose your desired language for the generic webhook trigger template. This topic uses a C# function.

Choose a template below or go to the quickstart

generic

Language: All Scenario: All

Generic webhook

A function that will be run whenever it receives a webhook

C# F# JavaScript TypeScript

3. Type a **Name** for your function, then select **Create**.



Generic webhook

New Function

Language:

C#

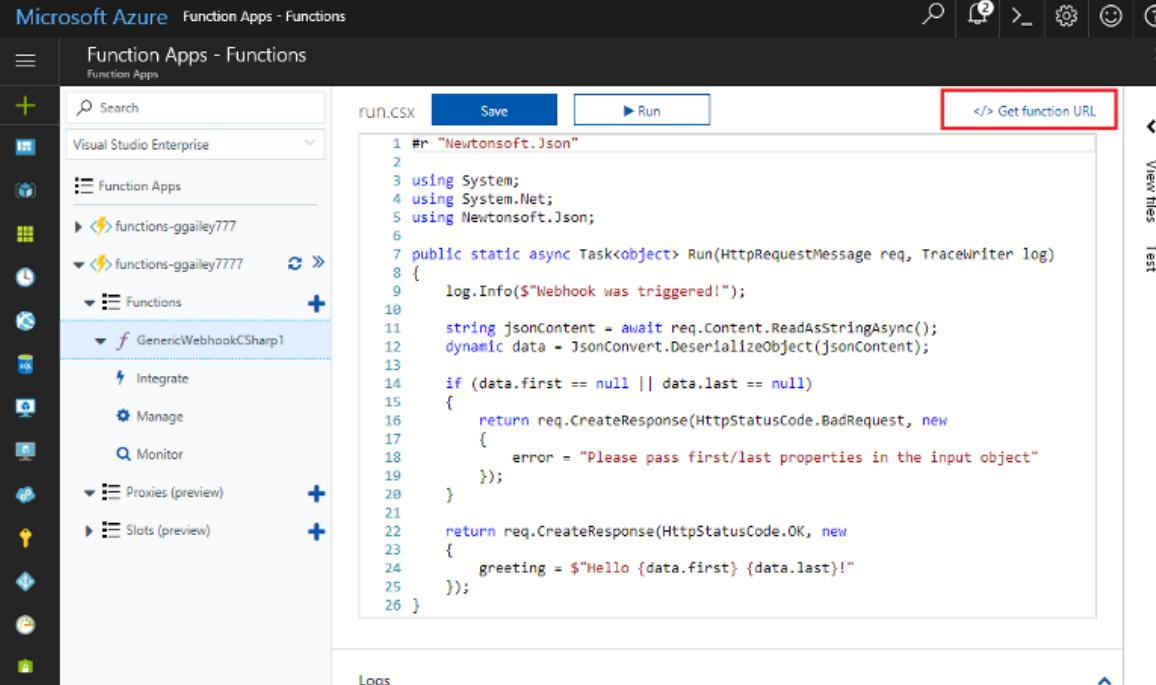
Name:

GenericWebhookCSharp1

Create

Cancel

- In your new function, click **</> Get function URL**, then copy and save the value. You use this value to configure the webhook.



Microsoft Azure Function Apps - Functions

Function Apps - Functions

Visual Studio Enterprise

Function Apps

functions-ggailey777

functions-ggailey777

Functions

GenericWebhookCSharp1

Integrate

Manage

Monitor

Proxies (preview)

Slots (preview)

RUN.csx

Save

Run

</> Get function URL

```

1 #r "Newtonsoft.Json"
2
3 using System;
4 using System.Net;
5 using Newtonsoft.Json;
6
7 public static async Task<object> Run(HttpRequestMessage req, TraceWriter log)
8 {
9     log.Info($"Webhook was triggered!");
10
11    string jsonContent = await req.Content.ReadAsStringAsync();
12    dynamic data = JsonConvert.DeserializeObject(jsonContent);
13
14    if (data.first == null || data.last == null)
15    {
16        return req.CreateResponse(HttpStatusCode.BadRequest, new
17        {
18            error = "Please pass first/last properties in the input object"
19        });
20    }
21
22    return req.CreateResponse(HttpStatusCode.OK, new
23    {
24        greeting = $"Hello {data.first} {data.last}!"
25    });
26 }
```

View file

Test

Logs

Next, you create a webhook endpoint in an activity log alert in Azure Monitor.

Create an activity log alert

- In the Azure portal, navigate to the **Monitor** service, select **Alerts**, and click **Add activity log alert**.

Microsoft Azure Monitor - Alerts

New
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Azure Active Directory
Monitor
Security Center
Billing
Help + support
Advisor
Function Apps

Monitor - Alerts Microsoft

Search (Ctrl+ /)

Columns Add metric alert **Add activity log alert**

* Subscription Visual Studio Enterprise Source All sources Resource group Type to start filter

Filter alerts...

NAME	STATUS	CONDITION	RESOURCE G...
test-function-al...	Active	category equal...	functions-ggail...

2. Use the settings as specified in the table:

Add activity log alert

* Activity log alert name **resource-group-create-alert**

Description

* Subscription **Visual Studio Enterprise**

* Resource group **myResourceGroup**

Source

Input **Activity log**

Criteria

* Event category **Administrative**

Resource type **Resource groups (Microsoft.Resources/subscriptions/resourceGroups)**

Resource group **All**

Resource **All**

Operation name **Create Resource Group (subscriptions/resourceGroups)**

Level **Informational**

Status **Succeeded**

Event initiated by

Alert via

Action group **New** Existing

* Action group name **function-webhook**

* Short name **funcwebhook**

Actions

SETTING	SUGGESTED VALUE	DESCRIPTION
Activity log alert name	resource-group-create-alert	Name of the activity log alert.
Subscription	Your subscription	The subscription you are using for this tutorial.
Resource Group	myResourceGroup	The resource group that the alert resources are deployed to. Using the same resource group as your function app makes it easier to clean up after you complete the tutorial.
Event category	Administrative	This category includes changes made to Azure resources.
Resource type	Resource groups	Filters alerts to resource group activities.
Resource Group and Resource	All	Monitor all resources.
Operation name	Create Resource Group	Filters alerts to create operations.
Level	Informational	Include informational level alerts.
Status	Succeeded	Filters alerts to actions that have completed successfully.
Action group	New	Create a new action group, which defines the action takes when an alert is raised.
Action group name	function-webhook	A name to identify the action group.
Short name	funcwebhook	A short name for the action group.

3. In **Actions**, add an action using the settings as specified in the table:

Actions

NAME	ACTION TYPE	DETAILS
CallFunctionWebhook ✓	Webhook ▾	https://functions-glenga... The name of the receiver ...
	SMS ▾	Depending on the action. ...

Note that only the country code '1' is currently supported for SMS.
It can take up to 5 minutes for an Activity log alert to become active.

OK

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	CallFunctionWebhook	A name for the action.
Action type	Webhook	The response to the alert is that a Webhook URL is called.
Details	Function URL	Paste in the webhook URL of the function that you copied earlier.

- Click **OK** to create the alert and action group.

The webhook is now called when a resource group is created in your subscription. Next, you update the code in your function to handle the JSON log data in the body of the request.

Update the function code

- Navigate back to your function app in the portal, and expand your function.
- Replace the C# script code in the function in the portal with the following code:

```
#r "Newtonsoft.Json"

using System;
using System.Net;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

public static async Task<object> Run(HttpRequestMessage req, TraceWriter log)
{
    log.Info($"Webhook was triggered!");

    // Get the activityLog object from the JSON in the message body.
    string jsonContent = await req.Content.ReadAsStringAsync();
    JToken activityLog = JObject.Parse(jsonContent.ToString())
        .SelectToken("data.context.activityLog");

    // Return an error if the resource in the activity log isn't a resource group.
    if (activityLog == null || !string.Equals((string)activityLog["resourceType"],
        "Microsoft.Resources/subscriptions/resourcegroups"))
    {
        log.Error("An error occurred");
        return req.CreateResponse(HttpStatusCode.BadRequest, new
        {
            error = "Unexpected message payload or wrong alert received."
        });
    }

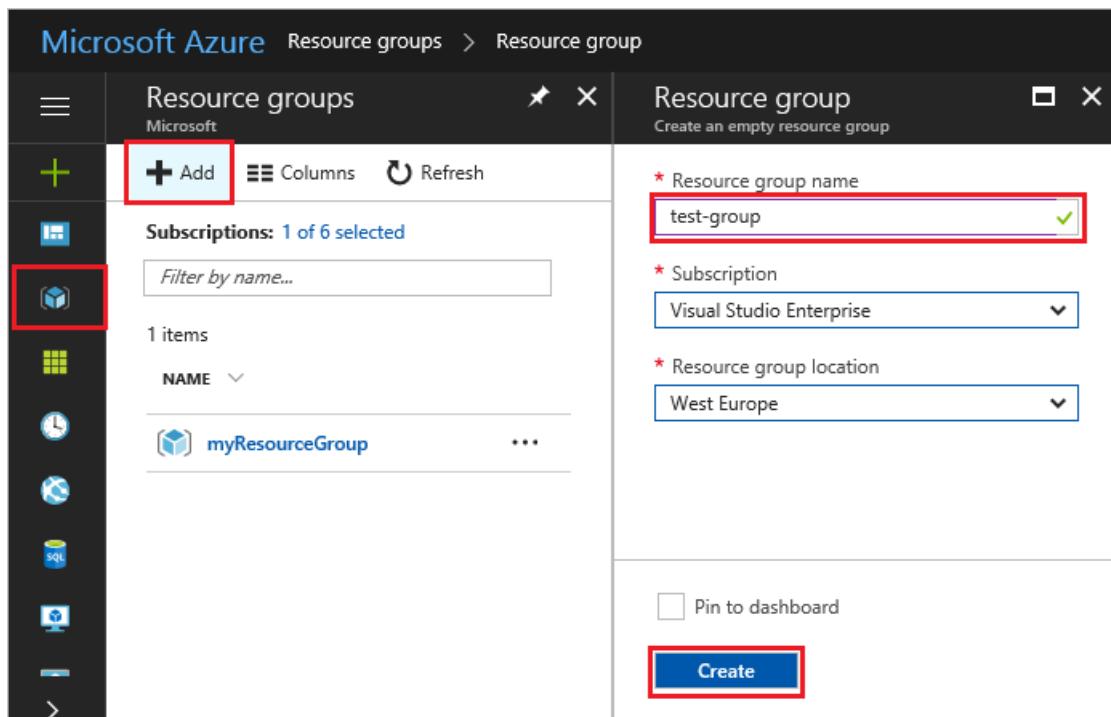
    // Write information about the created resource group to the streaming log.
    log.Info(string.Format("Resource group '{0}' was {1} on {2}.",
        (string)activityLog["resourceGroupName"],
        ((string)activityLog["subStatus"]).ToLower(),
        (DateTime)activityLog["submissionTimestamp"]));

    return req.CreateResponse(HttpStatusCode.OK);
}
```

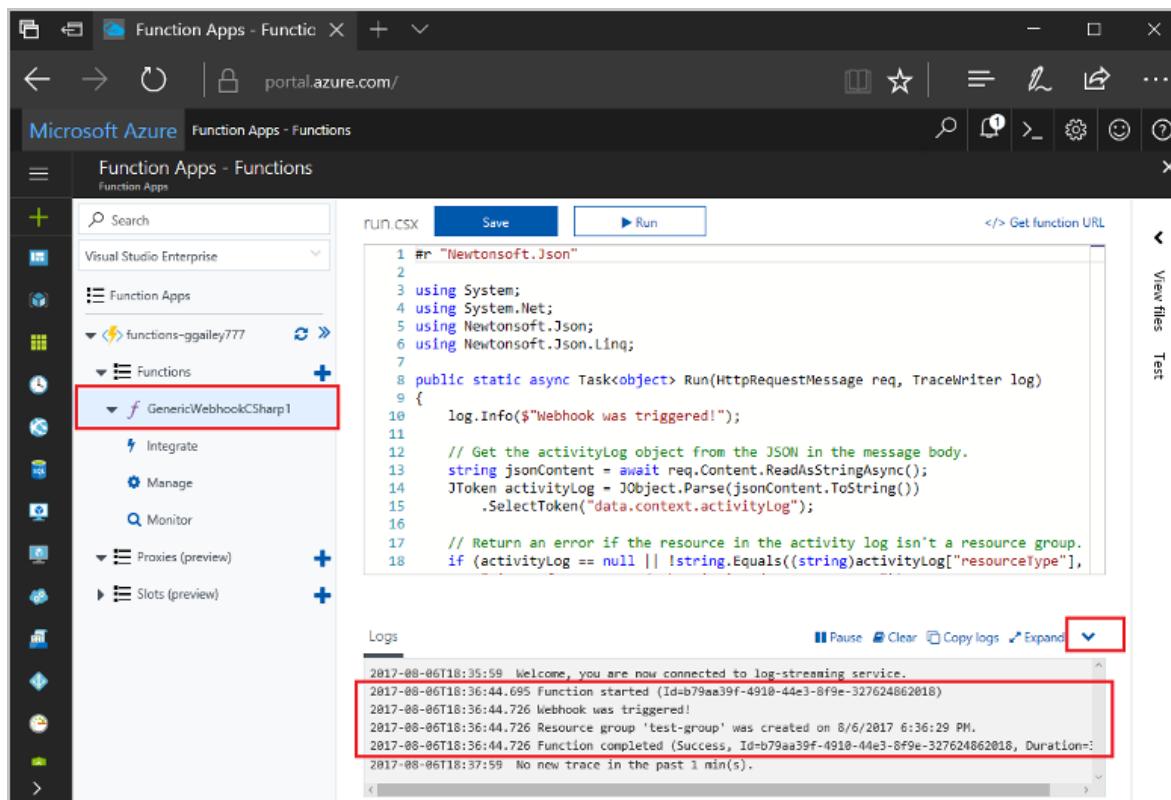
Now you can test the function by creating a new resource group in your subscription.

Test the function

- Click the resource group icon in the left of the Azure portal, select **+ Add**, type a **Resource group name**, and select **Create** to create an empty resource group.



- Go back to your function and expand the **Logs** window. After the resource group is created, the activity log alert triggers the webhook and the function executes. You see the name of the new resource group written to the logs.



- (Optional) Go back and delete the resource group that you created. Note that this activity doesn't trigger the function. This is because delete operations are filtered out by the alert.

Clean up resources

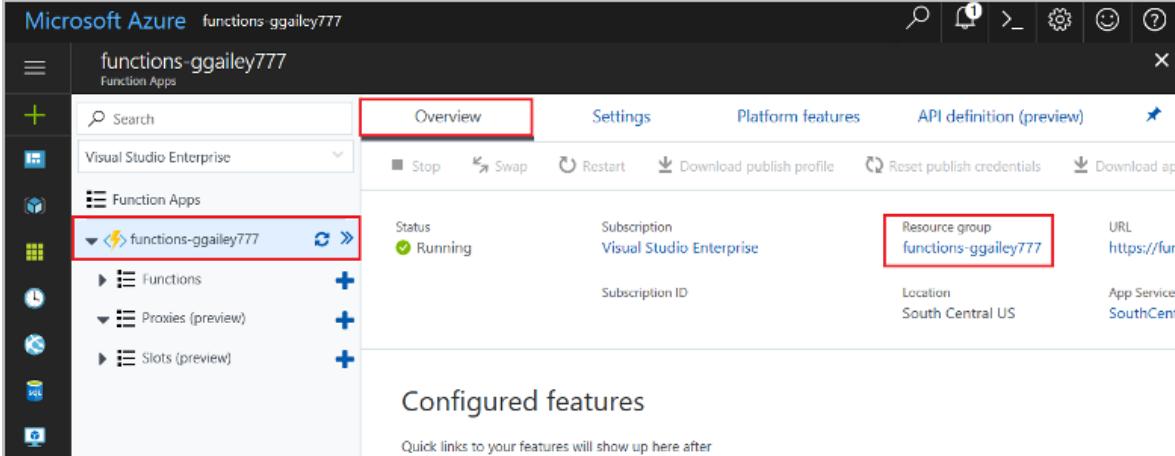
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

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' with the value 'functions-ggailey777', which is also highlighted with a red box. The left sidebar shows a tree view with 'functions-ggailey777' expanded, revealing 'Functions', 'Proxies (preview)', and 'Slots (preview)' under 'Function Apps'. The main content area displays the function app's status as 'Running', its subscription as 'Visual Studio Enterprise', and its location as 'South Central US'. It also shows the URL '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 when a request is received from a generic webhook.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

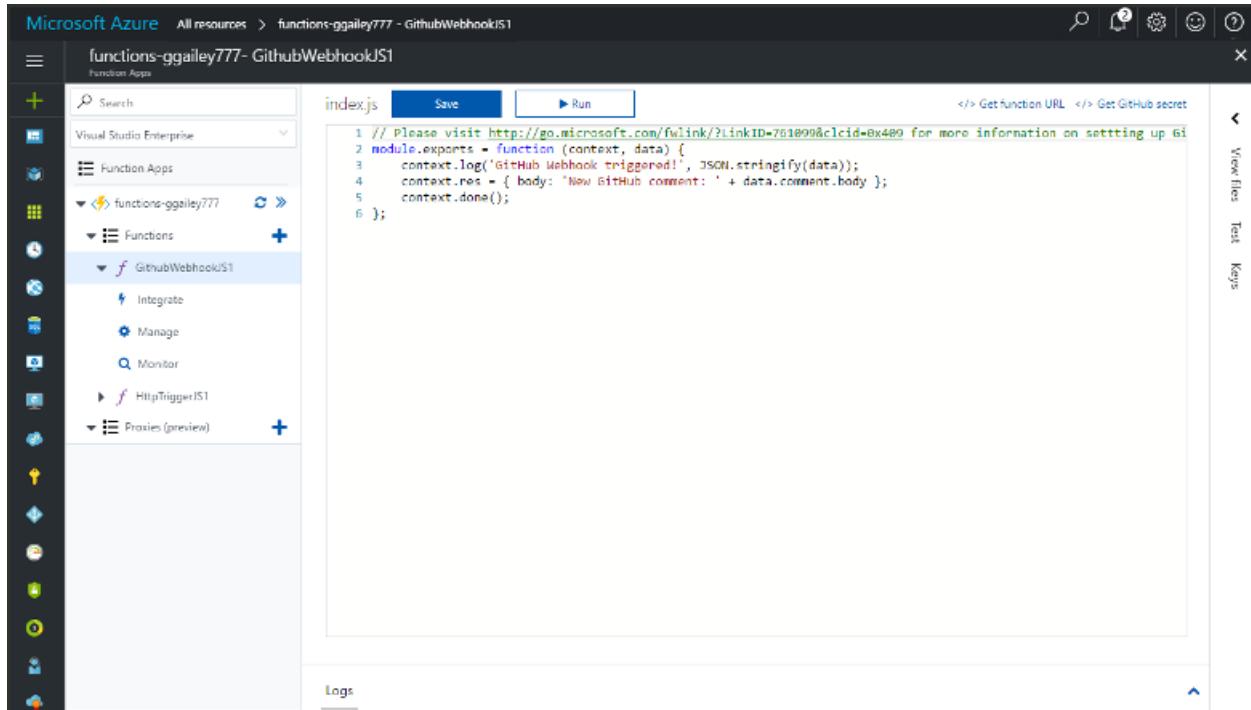
- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information about webhook triggers, see [Azure Functions HTTP and webhook bindings](#). To learn more about developing functions in C#, see [Azure Functions C# script developer reference](#).

Create a function triggered by a GitHub webhook

12/13/2017 • 5 min to read • [Edit Online](#)

Learn how to create a function that is triggered by an HTTP webhook request with a GitHub-specific payload.



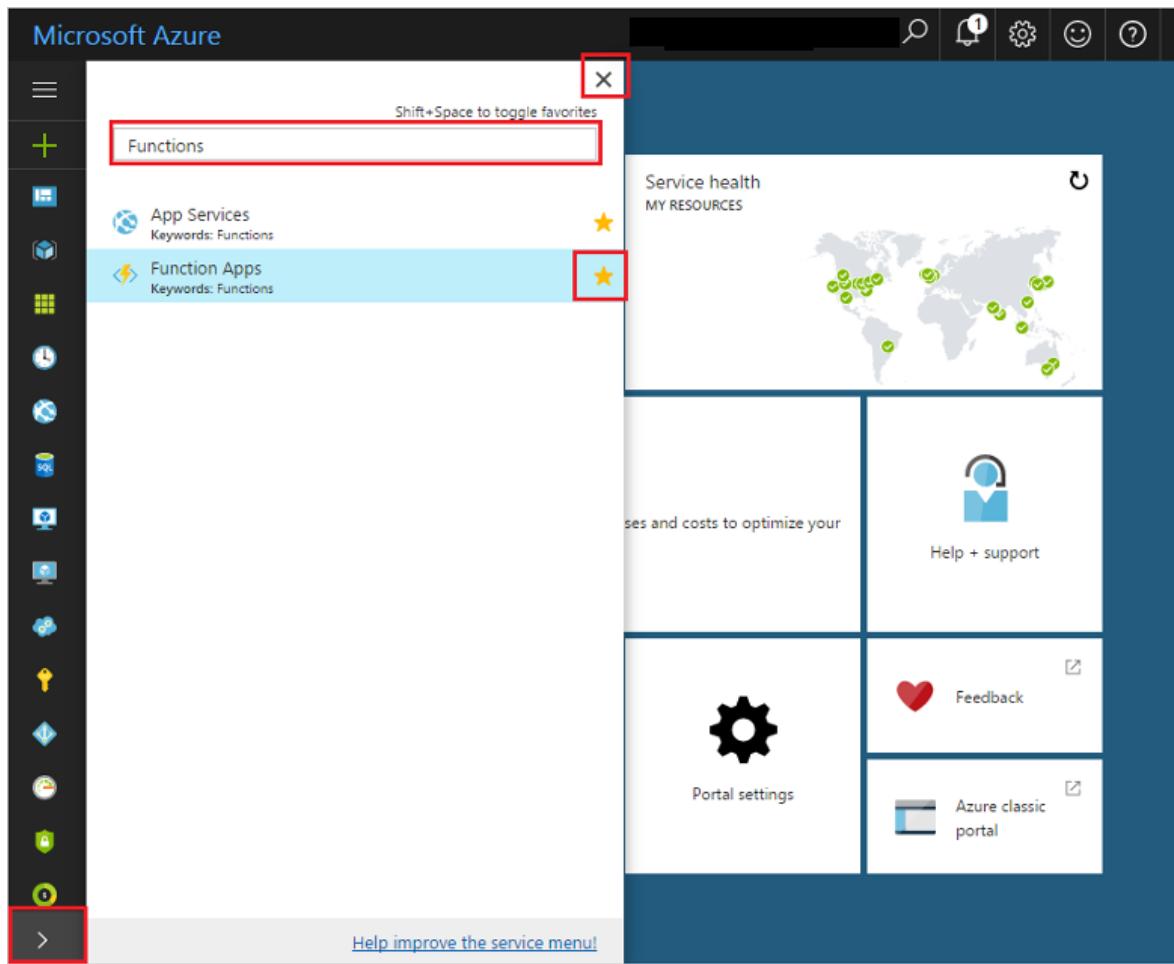
Prerequisites

- A GitHub account with at least one project.
- An Azure subscription. If you don't have one, create a [free account](#) before you begin.

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

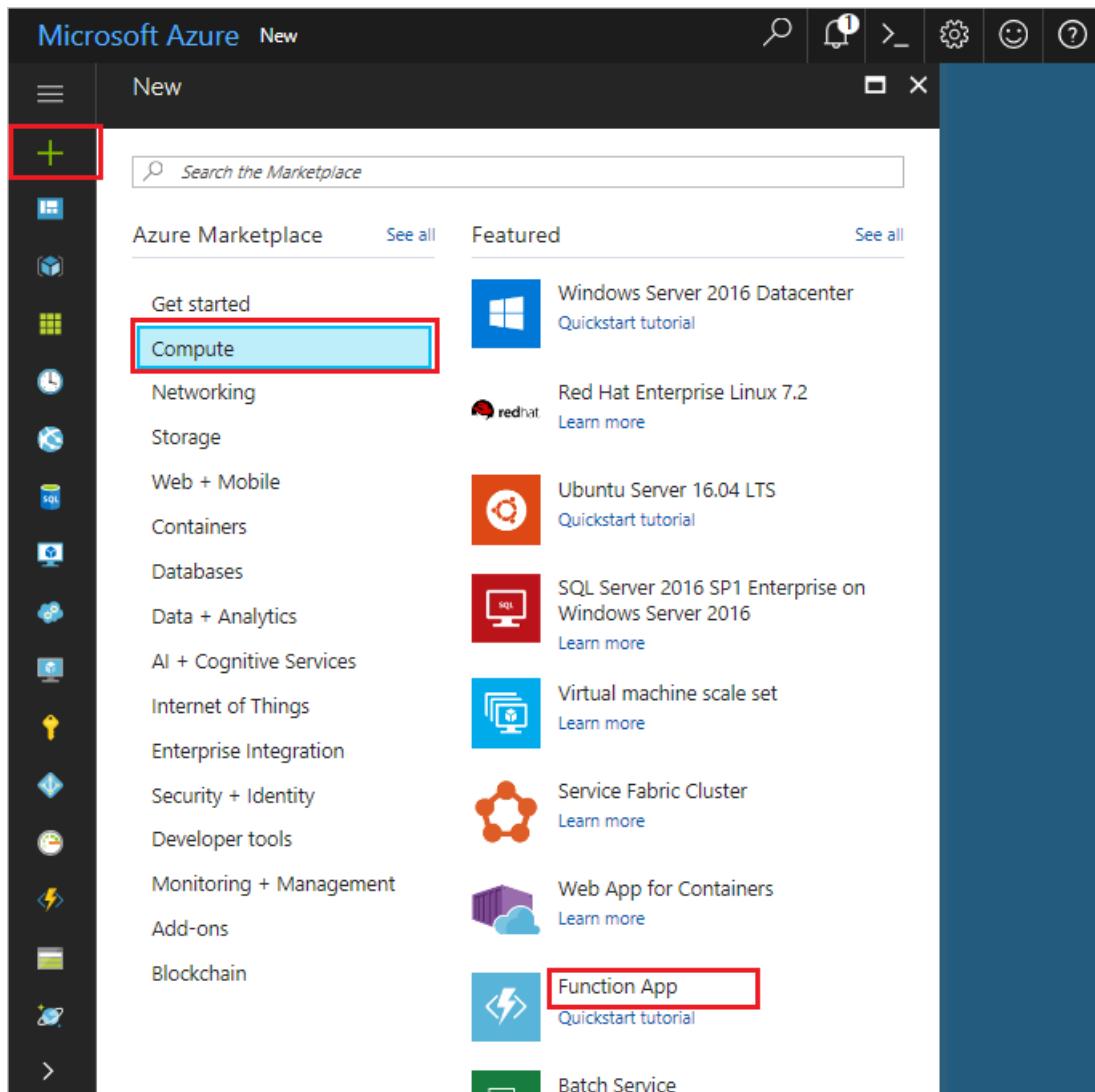
3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

A screenshot of the Microsoft Azure Function Apps blade. The top navigation bar shows 'Function Apps'. The left sidebar has a 'Function Apps' icon highlighted with a red box. The main content area displays a table titled 'Function Apps' with one row:

NAME	SUBSCRIPTION ID	RESOURCE GROUP	LOCATION
functions-ggailey777	Visual Studio Enterprise	functions-ggailey777	southcentralus

Create an Azure Function app

1. Click the **New** 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 X

Create

* App name
functions-ggailey777 ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group i
 Create new Use existing
functions-ggailey777 ✓

* OS Windows Linux

* Hosting Plan i
Consumption Plan

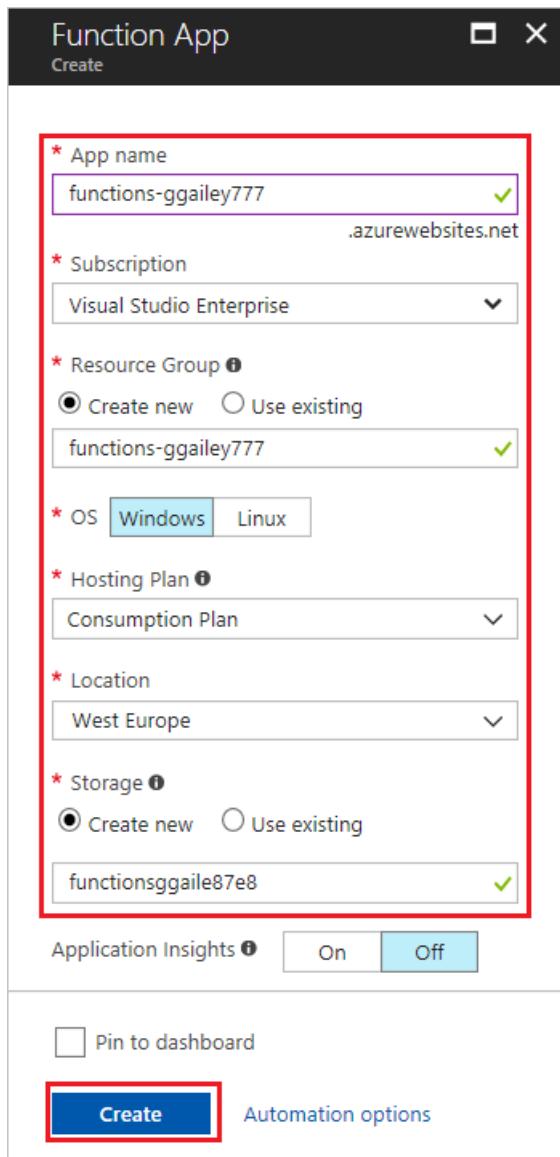
* Location
West Europe

* Storage i
 Create new Use existing
functionsggaile87e8 ✓

Application Insights i On Off

Pin to dashboard

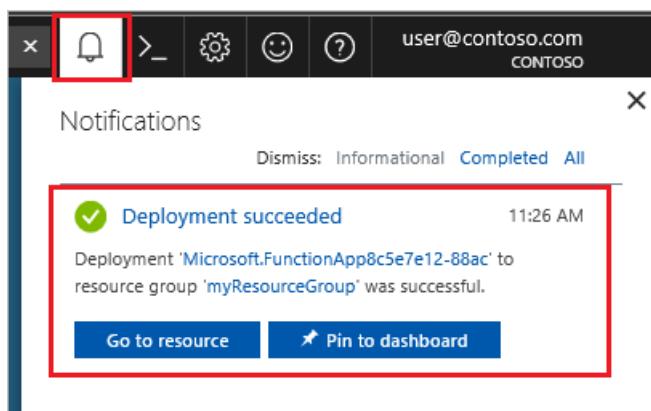
Create Automation options



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.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.

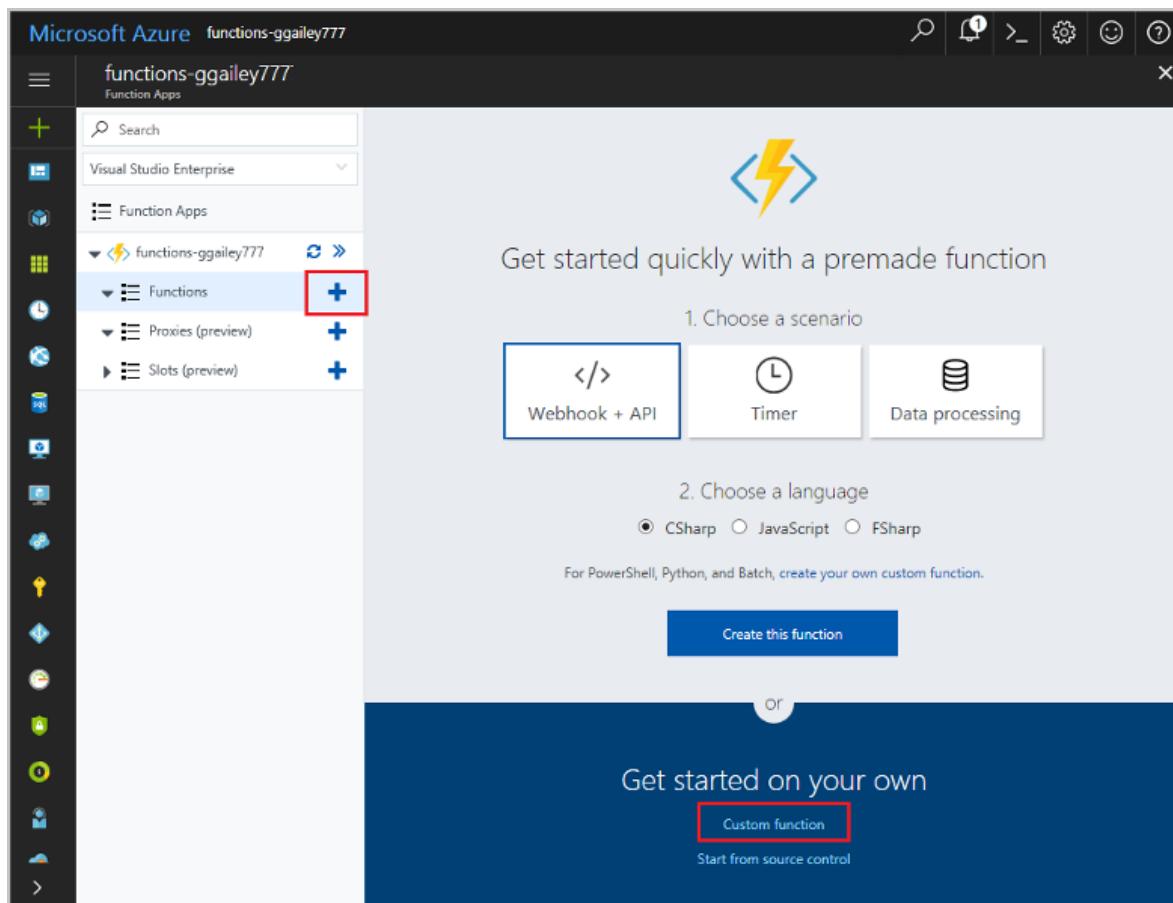


Clicking **Go to resource** takes you to your new function app.

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

Create a GitHub webhook 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 **Custom function**. This displays the complete set of function templates.



2. In the search field, type `github` and then choose your desired language for the GitHub webhook trigger template.

Choose a template below or go to the quickstart

×

Language: All

Scenario: All

 GitHub commenter

A function that will be run whenever it receives a GitHub webhook for an issue or pull request and adds a comment

C# F# JavaScript

 GitHub webhook

A function that will be run whenever it receives a GitHub webhook

C# F# JavaScript TypeScript

3. Type a **Name** for your function, then select **Create**.

 GitHub webhook

New Function

Language:

Name:

4. In your new function, click **</> Get function URL**, then copy and save the values. Do the same thing for **</> Get GitHub secret**. You use these values to configure the webhook in GitHub.

```

1 // Please visit http://go.microsoft.com/fwlink/?LinkId=761099&clcid=0x409 for more information
2 module.exports = function (context, data) {
3     context.log('GitHub Webhook triggered!', data.comment.body);
4     context.res = { body: 'New GitHub comment: ' + data.comment.body };
5     context.done();
6 };

```

Logs

- 2017-04-19T16:29:48 Welcome, you are now connected to log-streaming service.
- 2017-04-19T16:30:48 No new trace in the past 1 min(s).
- 2017-04-19T16:31:48 No new trace in the past 2 min(s).
- 2017-04-19T16:32:48 No new trace in the past 3 min(s).

Next, you create a webhook in your GitHub repository.

Configure the webhook

1. In GitHub, navigate to a repository that you own. You can also use any repository that you have forked. If you need to fork a repository, use <https://github.com/Azure-Samples/functions-quickstart>.
2. Click **Settings**, then click **Webhooks**, and **Add webhook**.

This repository Search Pull requests Issues Gist

ggailey777 / try-stuff

Code Issues 2 Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

Webhooks

Add webhook

Webhooks allow external services to be notified when certain events happen within your repository. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#).

Event	URL	Content type
Push event	http://functions-ggailey777.azurewebsites.net/api/GithubWebhookJS1	application/json

Options Collaborators Branches Webhooks Integrations & services Deploy keys

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

3. Use settings as specified in the table, then click **Add webhook**.

The screenshot shows the GitHub repository settings page for 'ggailey777 / try-stuff'. The 'Webhooks' tab is selected. A new webhook is being configured with the following details:

- Payload URL ***: <https://functions-ggailey777.azurewebsites.net/api/GithubWebhook>
- Content type**: application/json
- Secret**: (redacted)
- SSL verification**: By default, we verify SSL certificates when delivering payloads. (Disable SSL verification)
- Event triggers**:
 - Let me select individual events.
 - Commit comment
Commit or diff commented on.
 - Create
Branch or tag created.
 - Delete
Branch or tag deleted.
 - Deployment
Repository deployed.
 - Deployment status
Deployment status updated from the API.
 - Fork
Repository forked.
 - Issue comment
Issue comment created, edited, or deleted.

SETTING	SUGGESTED VALUE	DESCRIPTION
Payload URL	Copied value	Use the value returned by </> Get function URL .
Secret	Copied value	Use the value returned by </> Get GitHub secret .
Content type	application/json	The function expects a JSON payload.
Event triggers	Let me select individual events	We only want to trigger on issue comment events.
	Issue comment	

Now, the webhook is configured to trigger your function when a new issue comment is added.

Test the function

- In your GitHub repository, open the **Issues** tab in a new browser window.
- In the new window, click **New Issue**, type a title, and then click **Submit new issue**.
- In the issue, type a comment and click **Comment**.

The screenshot shows a GitHub issue page for a repository named 'try-stuff'. The issue is titled 'Test issue #20'. A comment from 'ggailey777' is visible, stating 'This is an issue to test the webhook.' Below this, there is a comment input field with the placeholder 'This is an issue comment.' A red box highlights this input field. To the right of the input field are several configuration options: Assignees ('No one—assign yourself'), Labels ('None yet'), Projects ('None yet'), Milestone ('No milestone'), and Notifications ('Unsubscribe'). A note at the bottom right says 'You're receiving notifications because you authored the thread.'

4. Go back to the portal and view the logs. You should see a trace entry with the new comment text.

The screenshot shows the Azure portal's log viewer. The logs section displays a trace entry with the following text:
2017-04-27T21:32:09 Welcome, you are now connected to log-streaming service.
2017-04-27T21:36:09 No new trace in the past 1 min(s).
2017-04-27T21:36:20.688 Function started (Id=5eba490f-1d34-4adf-bc7b-ce73fb42170c)
2017-04-27T21:36:21.751 GitHub Webhook triggered! This is an issue comment.
2017-04-27T21:36:21.782 Function completed (Success, Id=5eba490f-1d34-4adf-bc7b-ce73fb42170c, Duration=1084ms)

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 project name 'functions-ggailey777' and a 'Function Apps' dropdown set to 'Visual Studio Enterprise'. The main area has tabs for 'Overview' (which is selected and 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 a tree view showing 'functions-ggailey777' expanded, with 'Functions', 'Proxies (preview)', and 'Slots (preview)' listed. Below the sidebar is a section titled 'Configured features' with a note: 'Quick links to your features will show up here after'. The entire screenshot is framed by a red border.

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 request is received from a GitHub webhook.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

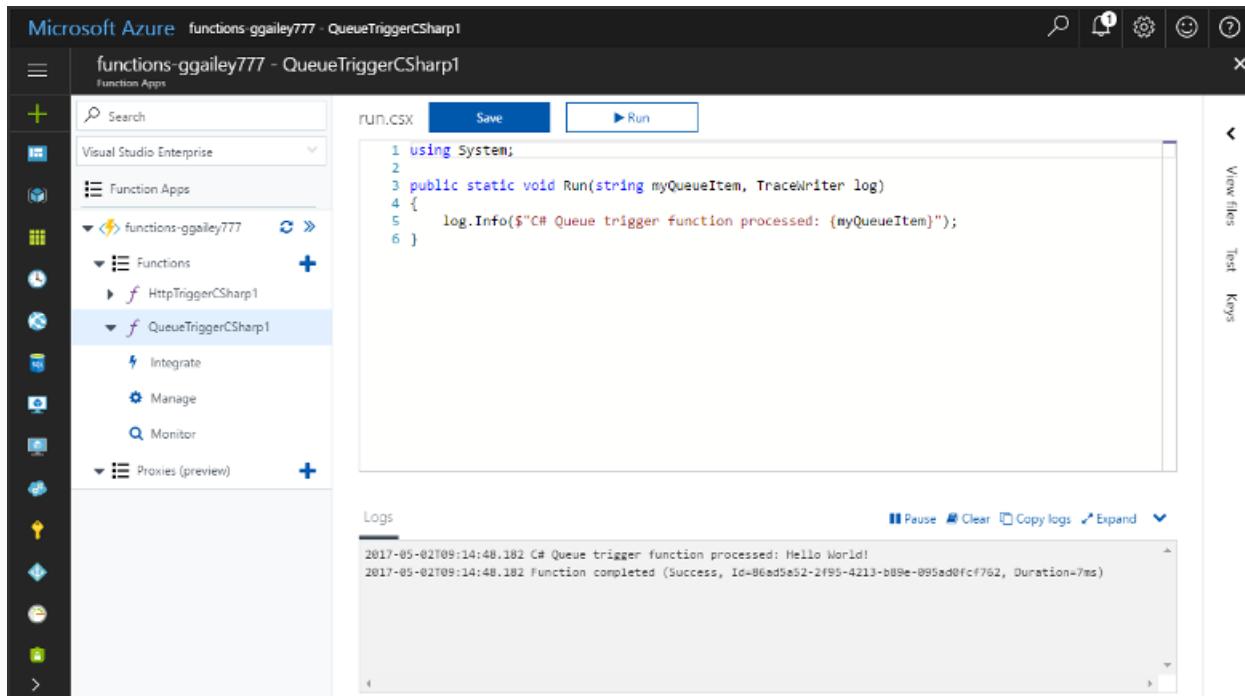
- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information about webhook triggers, see [Azure Functions HTTP and webhook bindings](#).

Create a function triggered by Azure Queue storage

12/13/2017 • 5 min to read • [Edit Online](#)

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



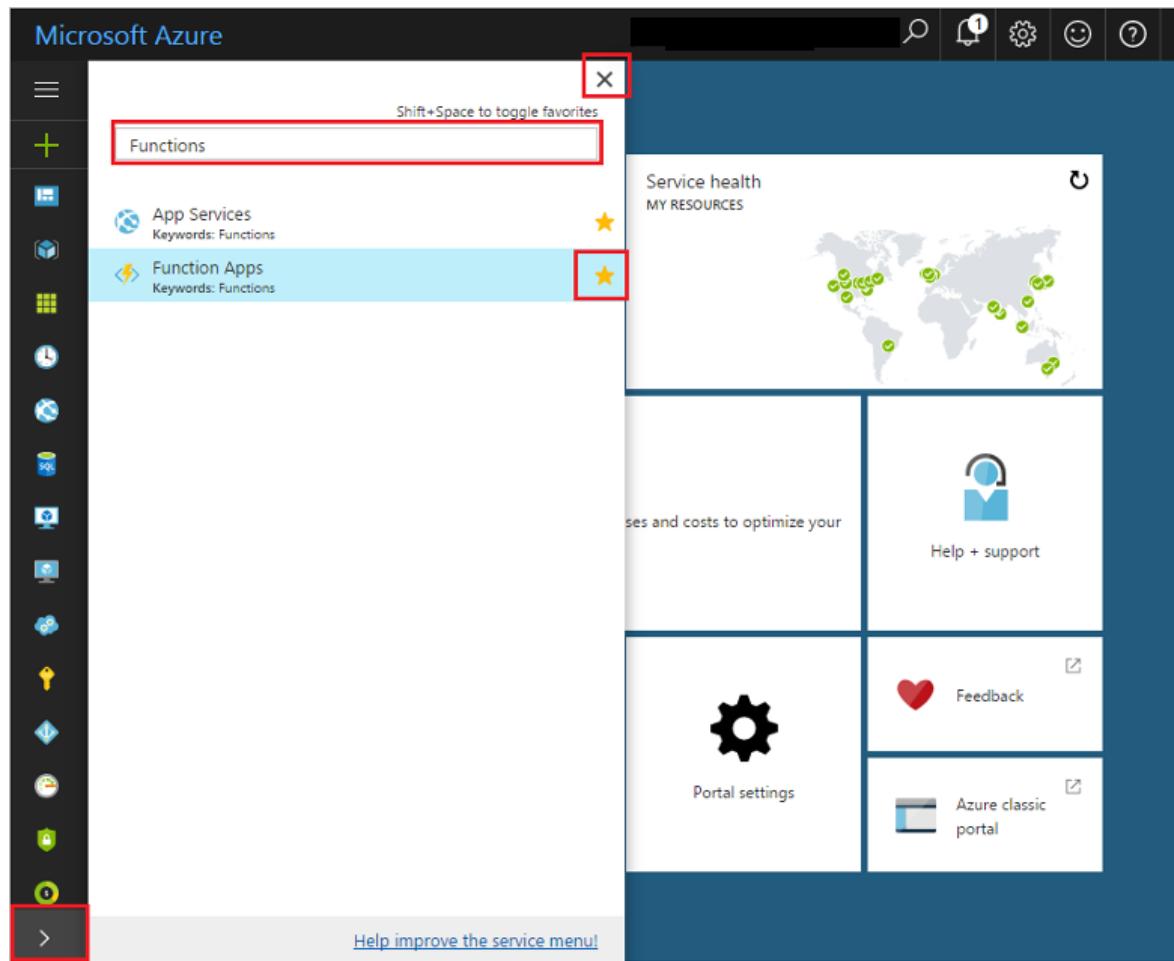
Prerequisites

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

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

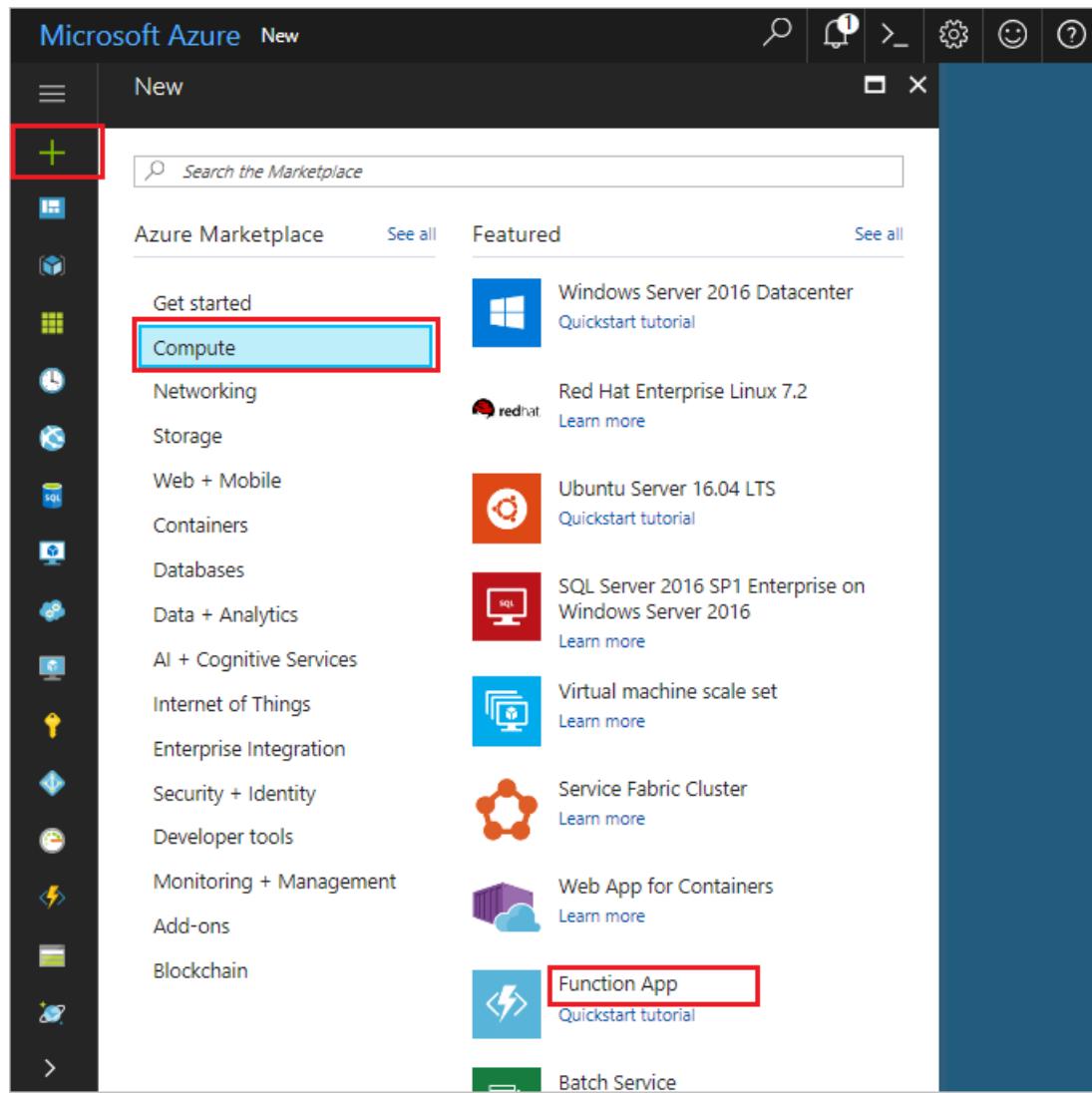
3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

The screenshot shows the Microsoft Azure portal with the 'Function Apps' blade open. The left sidebar has a red box around the 'Function Apps' icon. The main content area is titled 'Function Apps' and contains a table with one row of data. The table columns are 'NAME', 'SUBSCRIPTION ID', 'RESOURCE GROUP', and 'LOCATION'. The single row shows 'functions-ggailey777', 'Visual Studio Enterprise', 'functions-ggailey777', and 'southcentralus'. The top right corner of the screen has standard Azure portal navigation icons.

NAME	SUBSCRIPTION ID	RESOURCE GROUP	LOCATION
functions-ggailey777	Visual Studio Enterprise	functions-ggailey777	southcentralus

Create an Azure Function app

1. Click the **New** 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 X

Create

* App name
functions-ggailey777 ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise ▼

* Resource Group i
 Create new Use existing
functions-ggailey777 ✓

* OS Windows Linux

* Hosting Plan i
Consumption Plan ▼

* Location
West Europe ▼

* Storage i
 Create new Use existing
functions-ggaile87e8 ✓

Application Insights i On Off

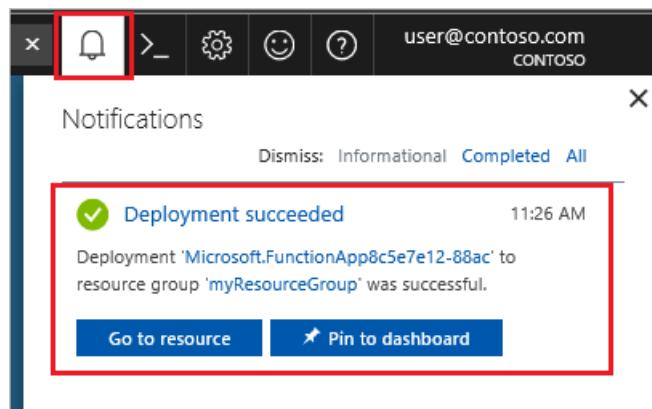
Pin to dashboard

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.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.



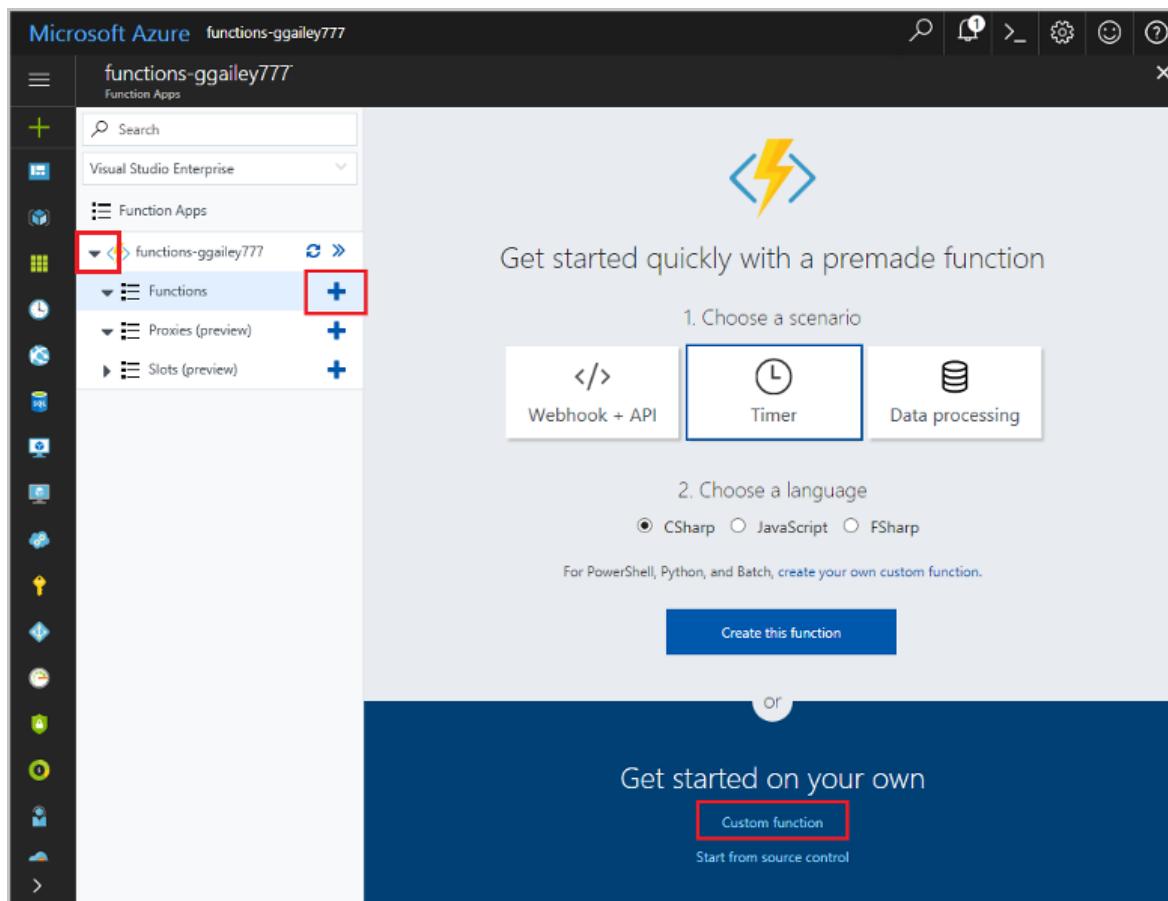
Clicking **Go to resource** takes you to your new function app.

The screenshot shows the Azure portal's "Overview" page for the function app "functions-ggailey777". The left sidebar shows the resource tree with "functions-ggailey777" selected under "Function Apps". The main pane displays the app's status as "Running", subscription information (Visual Studio Enterprise), resource group ("functions-ggailey777"), and URL ("https://functions-ggailey777.azurewebsites.net"). 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

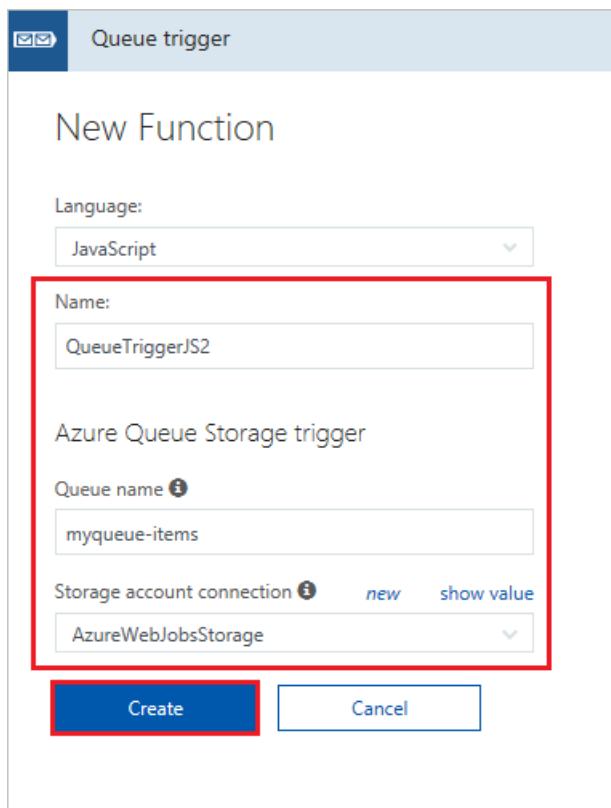
1. Expand your function app and click the + button next to **Functions**. If this is the first function in your function app, select **Custom function**. This displays the complete set of function templates.



2. In the search field, type `queue` and then choose your desired language for the Queue storage trigger template.

The screenshot shows the Azure portal search results for 'queue'. A red box highlights the search bar containing 'queue'. Below the search bar, there are filters for 'Language: All' and 'Scenario: All'. The results section shows the 'Queue trigger' template card, which has a red box around its language selection area ('Bash', 'Batch', 'C#', 'F#', 'JavaScript', 'PHP', 'PowerShell', 'Python', 'TypeScript'). Below this, there's another card for 'Service Bus Queue trigger'.

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



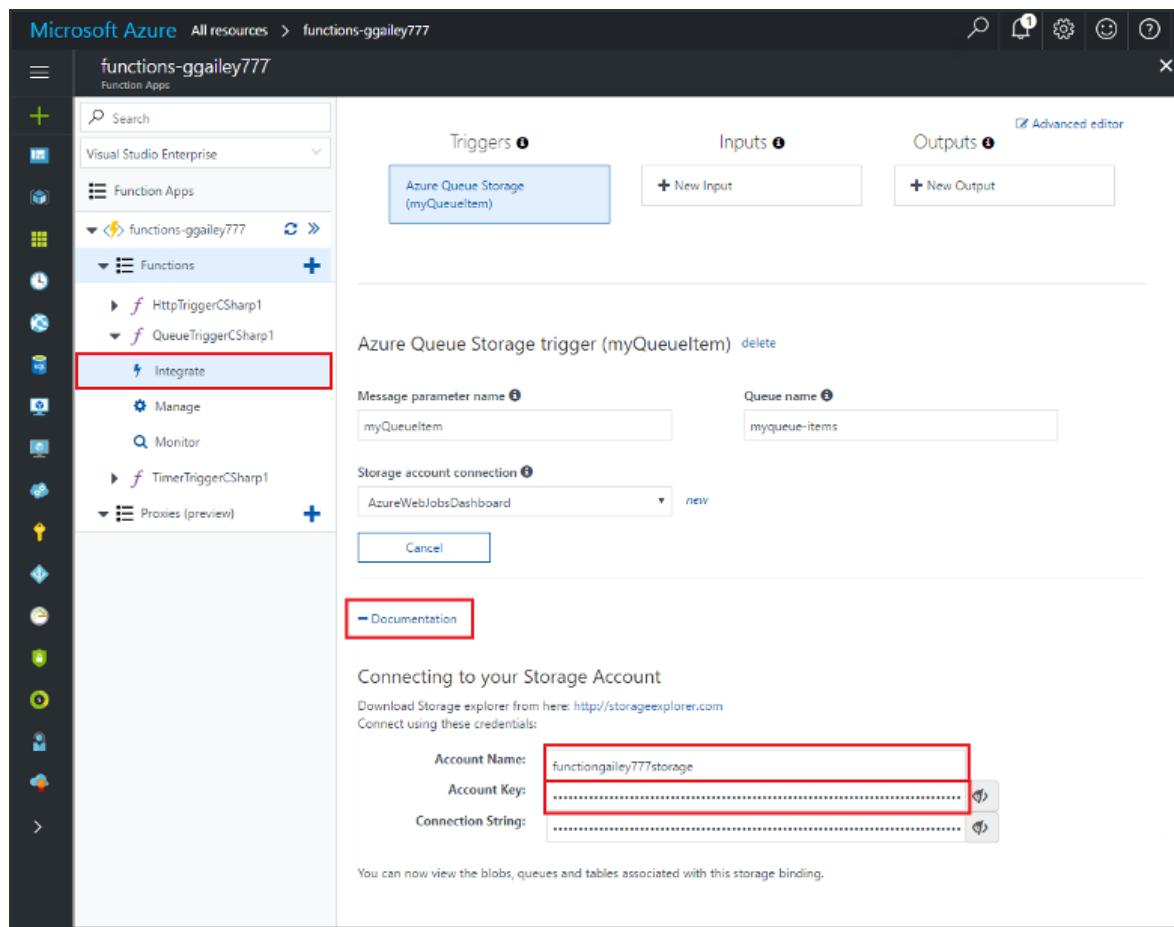
SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Unique in your function app	Name of this queue triggered function.
Queue name	myqueue-items	Name of the queue to connect to in your Storage account.
Storage account connection	AzureWebJobStorage	You can use the storage account connection already being used by your function app, or create a new one.

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

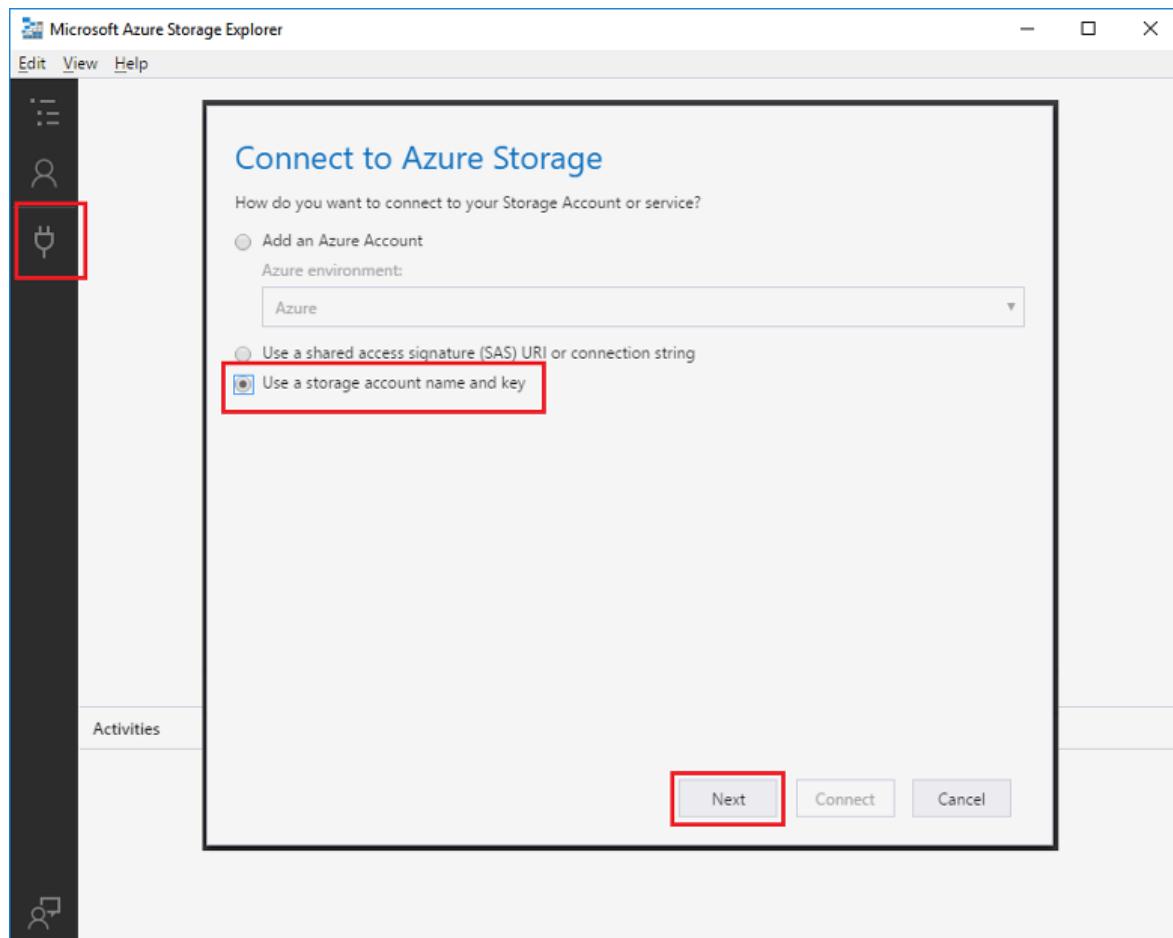
Next, you connect to your Azure Storage account and create the **myqueue-items** storage queue.

Create the queue

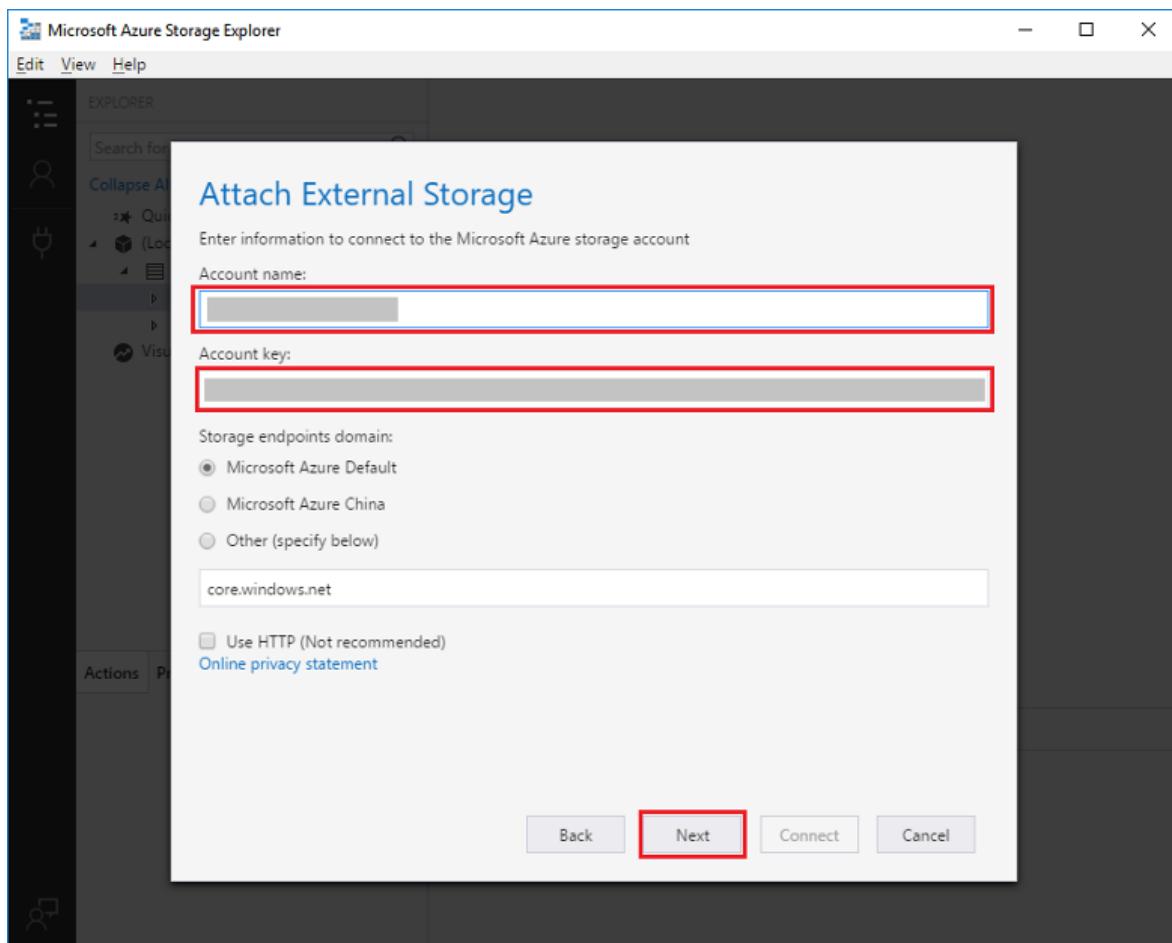
1. In your function, click **Integrate**, expand **Documentation**, and copy both **Account name** and **Account key**. You use these credentials to connect to the storage account in Azure Storage Explorer. If you have already connected your storage account, skip to step 4.



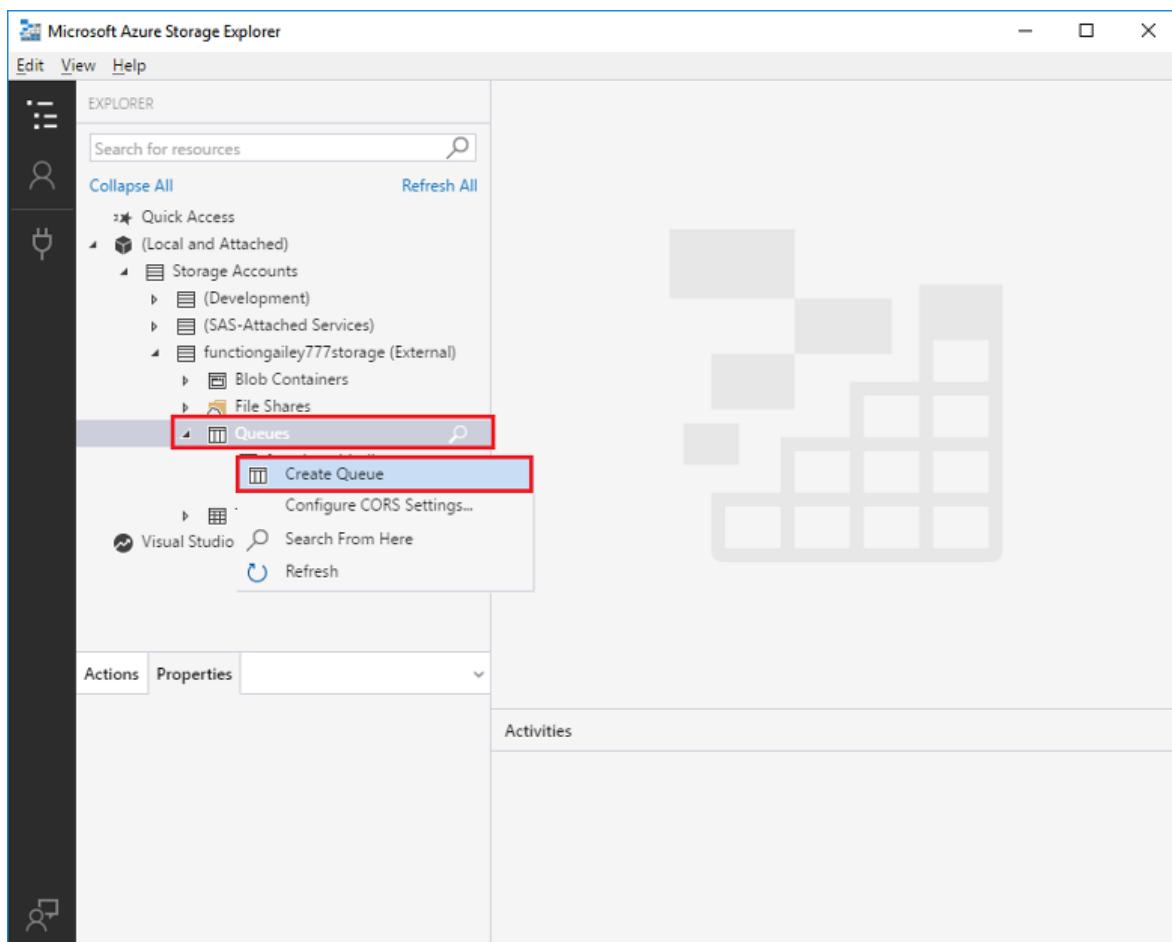
- Run the [Microsoft Azure Storage Explorer](#) tool, click the connect icon on the left, choose **Use a storage account name and key**, and click **Next**.



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



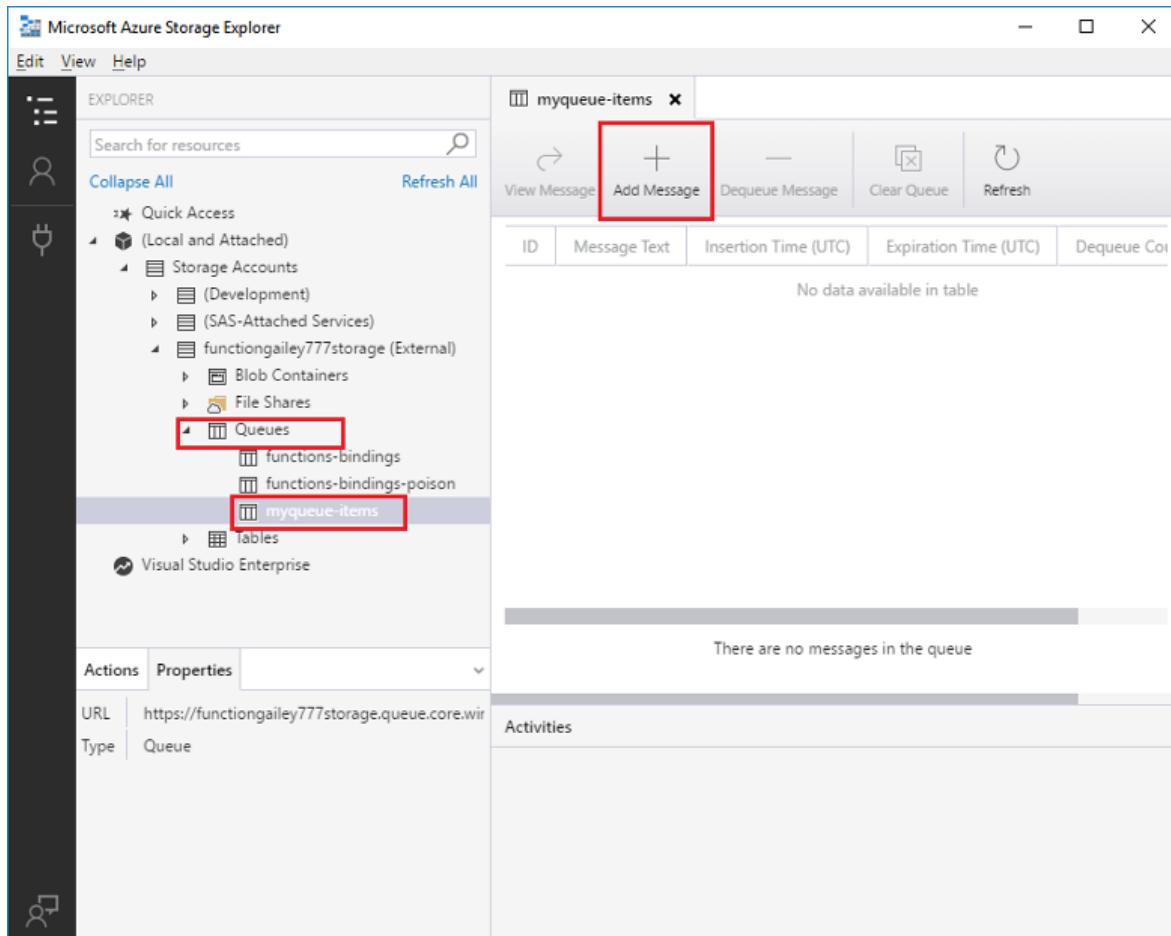
4. Expand the attached storage account, right-click **Queues**, click **Create Queue**, type `myqueue-items`, and then press enter.



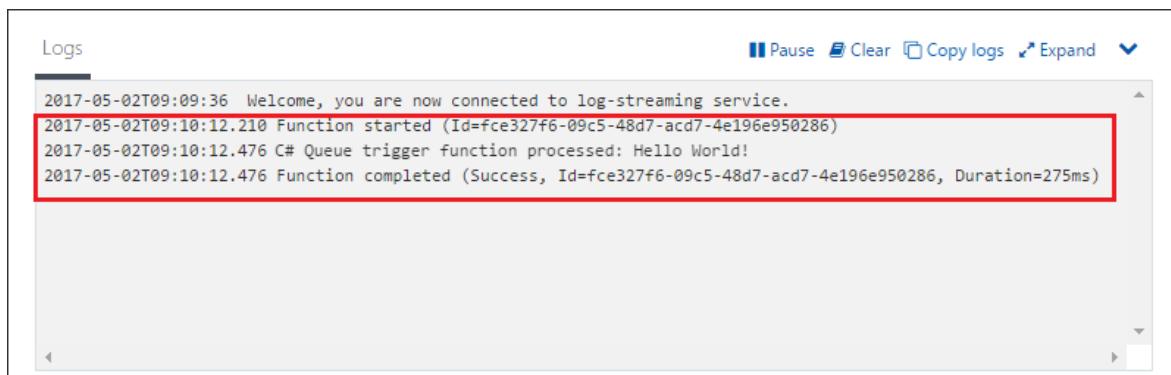
Now that you have a storage queue, you can test the function by adding a message to the queue.

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

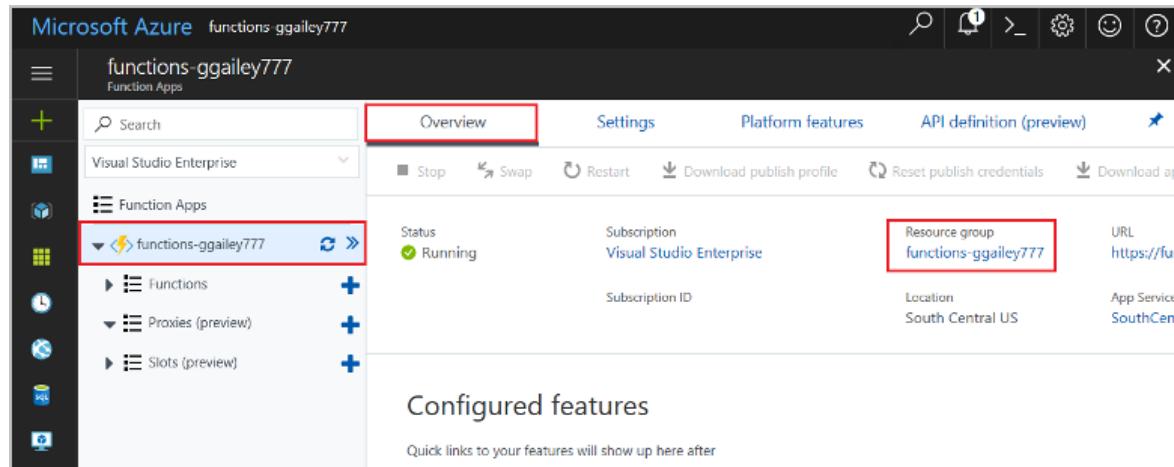
Other quick starts in this collection build upon this quick start. If you plan to work with subsequent quick starts, tutorials, or with any of the services you have created in this quick start, do not clean up the resources.

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 left sidebar lists 'Function Apps' with one item expanded, showing 'functions-ggailey777'. The top navigation bar has tabs for 'Overview', 'Settings', 'Platform features', and 'API definition (preview)'. Below the tabs are buttons for Stop, Swap, Restart, Download publish profile, Reset publish credentials, and Download ap. The main content area displays the app's status as 'Running', its subscription as 'Visual Studio Enterprise', and its resource group as 'functions-ggailey777'. It also shows location as 'South Central US' and URL as 'https://fun...'. A section titled 'Configured features' is present with a note: 'Quick links to your features will show up here after'. A red box highlights the 'Resource group' link in the top right corner of the main content area.

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.

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

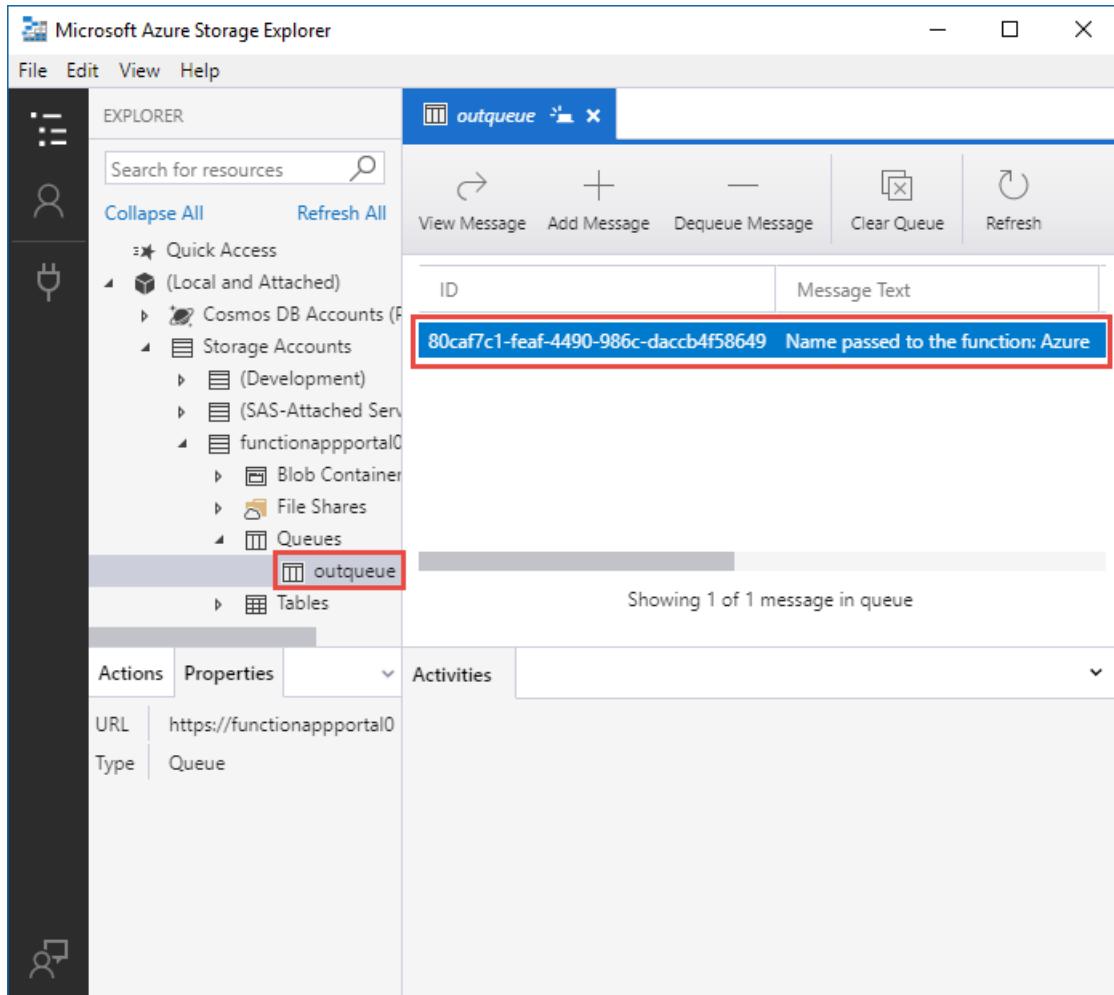
- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information about Queue storage triggers, see [Azure Functions Storage queue bindings](#).

Add messages to an Azure Storage queue using Functions

1/11/2018 • 6 min 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:

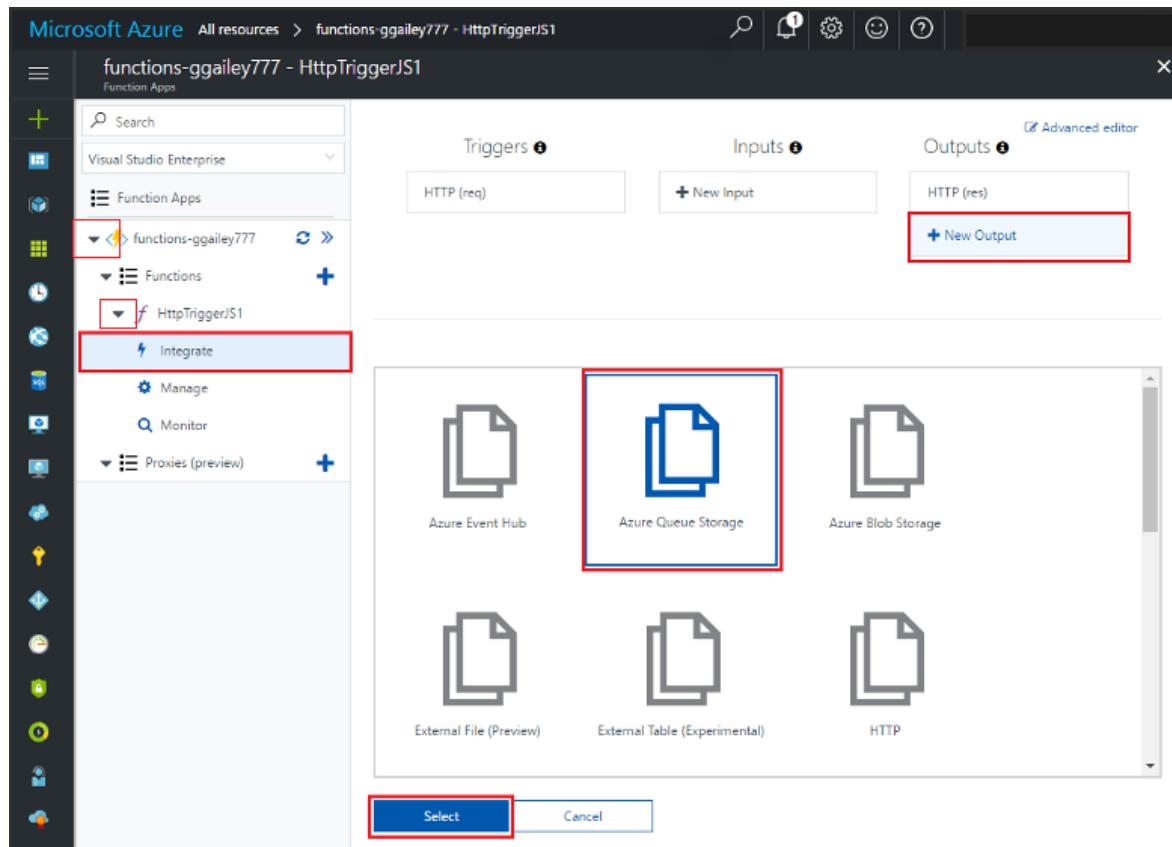
- Follow the directions in [Create your first function from the Azure portal](#) and don't do the **Clean up resources** step. That quickstart creates the function app and function that you use here.
- Install [Microsoft Azure Storage Explorer](#). This is a tool you'll use to examine queue messages 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 **More 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. Under **Azure Queue Storage output**, use the settings as specified in the table that follows this screenshot:

Azure Queue Storage output

Message parameter name i

Use function return value

Storage account connection i show value new

Queue name i

Save Cancel

The form has several input fields and buttons. The 'Message parameter name' field contains 'outputQueueItem'. The 'Storage account connection' dropdown is set to 'AzureWebJobsStorage' and has a 'new' link next to it. The 'Queue name' field contains 'outqueue'. At the bottom, there are 'Save' and 'Cancel' buttons, with 'Save' being highlighted with a red box.

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.

6. 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. For a C# function, add a method parameter for the binding and write code to use it:

Add an **outputQueueItem** parameter to the method signature as shown in the following example. The parameter name is the same as what you entered for **Message parameter name** when you created the binding.

```
public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,
    ICollector<string> outputQueueItem, TraceWriter log)
{
    ...
}
```

In the body of the C# 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. For a JavaScript function, add code that uses the output binding on the `context.bindings` object to create a queue message. Add this code before the `context.done` statement.

```
context.bindings.outputQueueItem = "Name passed to the function: " +
    (req.query.name || req.body.name);
```

4. Select **Save** to save changes.

Test the function

1. After the code changes are saved, select **Run**.

The screenshot shows the Azure Functions developer portal interface. On the left, the navigation bar includes 'Visual Studio Enterprise', 'Function Apps', 'functions-ggailey777', 'Functions', and 'Proxies (preview)'. The main area displays the code editor for 'index.js' with the following content:

```

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        context.bindings.outQueueItem = "Name passed to the function: " +
11            (req.query.name || req.body.name);
12    }
13    else {
14        context.res = {
15            status: 400,
16            body: "Please pass a name on the query string or in the request body"
17        };
18    }
19    context.done();
20 };

```

Below the code editor is the 'Logs' panel showing three log entries:

- 2017-04-20T17:45:48.408 Function started (Id=8b6cc21e-e419-4f83-8787-cd82fa9cbc24)
- 2017-04-20T17:45:48.424 JavaScript HTTP trigger function processed a request.
- 2017-04-20T17:45:48.596 Function completed (Success, Id=8b6cc21e-e419-4f83-8787-cd82fa9cbc24)

To the right of the code editor is the 'Test' interface. It shows the 'HTTP method' set to 'POST', the 'Request body' containing a JSON object with a single key 'name' and value 'Glenn', and the 'Output' pane displaying the response 'Hello Glenn*' and a red-bordered status message 'Status: 200 OK'.

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](#).

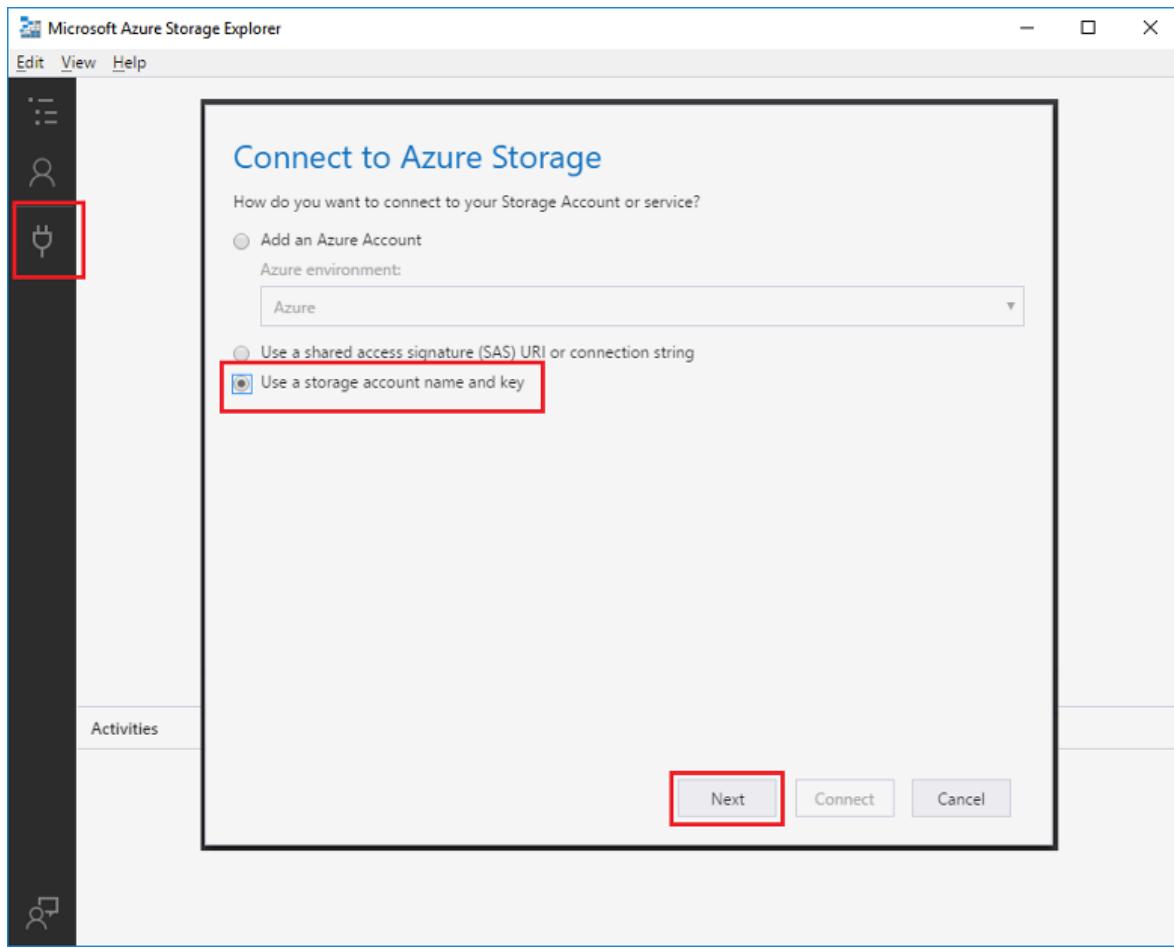
2. Check the logs to make sure that the function succeeded.

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.

Azure Queue Storage output (outputQueueItem) [delete](#)

Message parameter name [i](#): outputQueueItem

Queue name [i](#): outqueue

Use function return value

Storage account connection [i](#): AzureWebJobsDashboard [new](#)

Cancel

[Documentation](#)

Connecting to your Storage Account

Download Storage explorer from here: <http://storageexplorer.com>

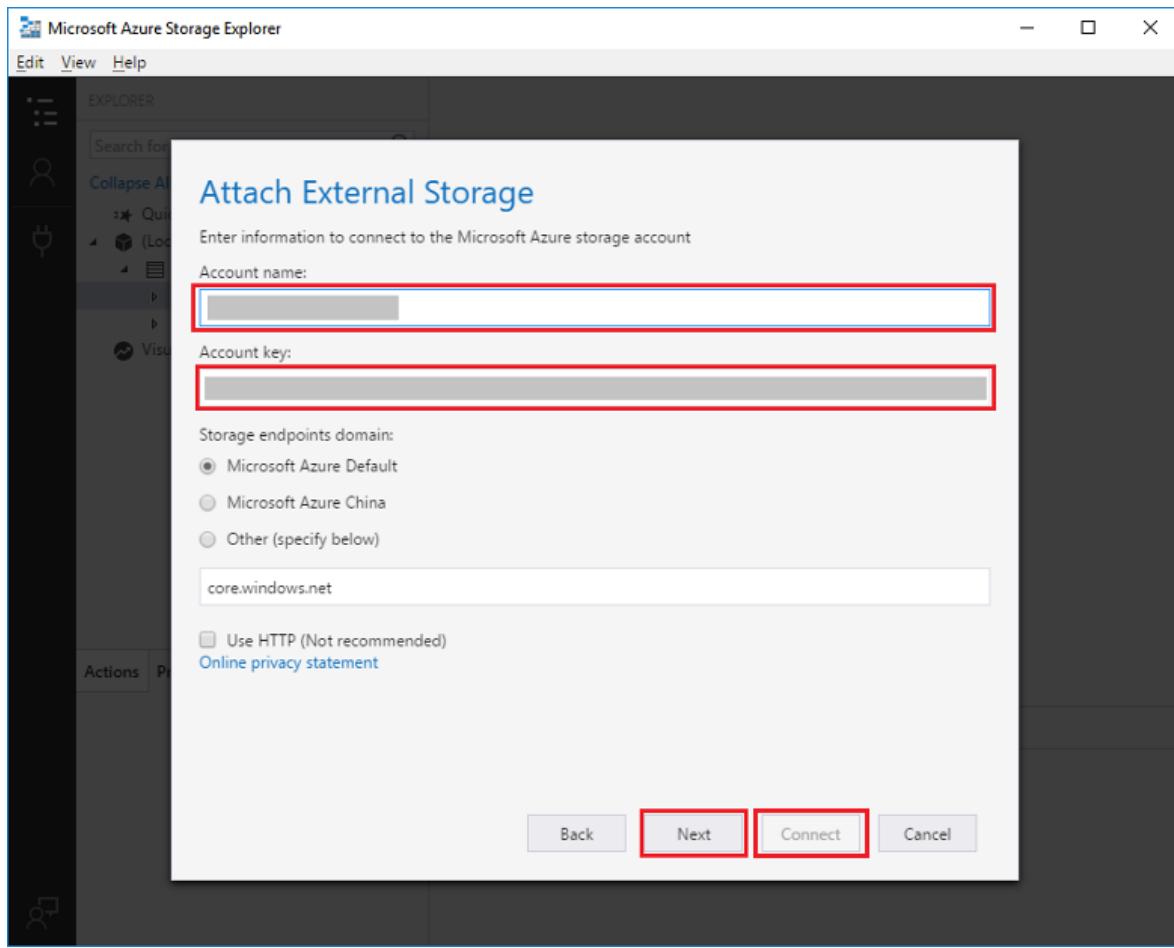
Connect using these credentials:

Account Name: [functions-ggailey777](#)

Account Key: [XXXXXXXXXXXXXX](#) [show](#) [hide](#)

Connection String: [XXXXXXXXXXXXXX](#) [show](#) [hide](#)

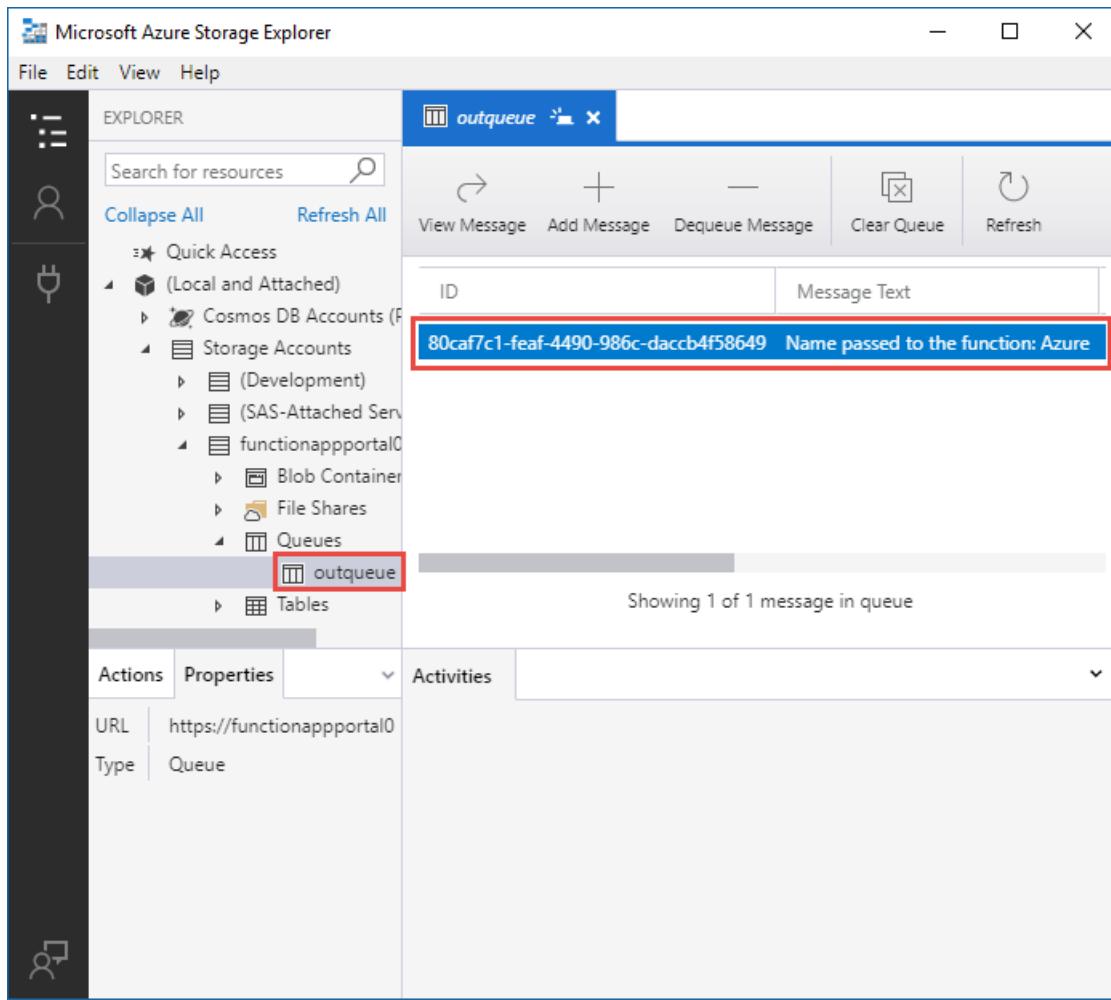
5. Copy the **Account Name** value from the portal and paste it in the **Account name** box in Storage Explorer.
6. Click the show/hide icon next to **Account Key** to display the value, and then copy the **Account Key** value and paste it in the **Account key** box in Storage Explorer.
7. Select **Next > 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. At the top, there's a navigation bar with icons for search, notifications, and settings. Below that is a header with the project name 'functions-ggailey777' and a 'Function Apps' dropdown set to 'Visual Studio Enterprise'. The main area has tabs for 'Overview' (which is selected and 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 a tree view showing 'functions-ggailey777' expanded, with 'Functions', 'Proxies (preview)', and 'Slots (preview)' listed. There are also '+' buttons to add more items. Below the sidebar is a section titled 'Configured features' with a note: 'Quick links to your features will show up here after'. The entire screenshot is framed by a red border.

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](#).

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

Store unstructured data using Azure Functions and Azure Cosmos DB

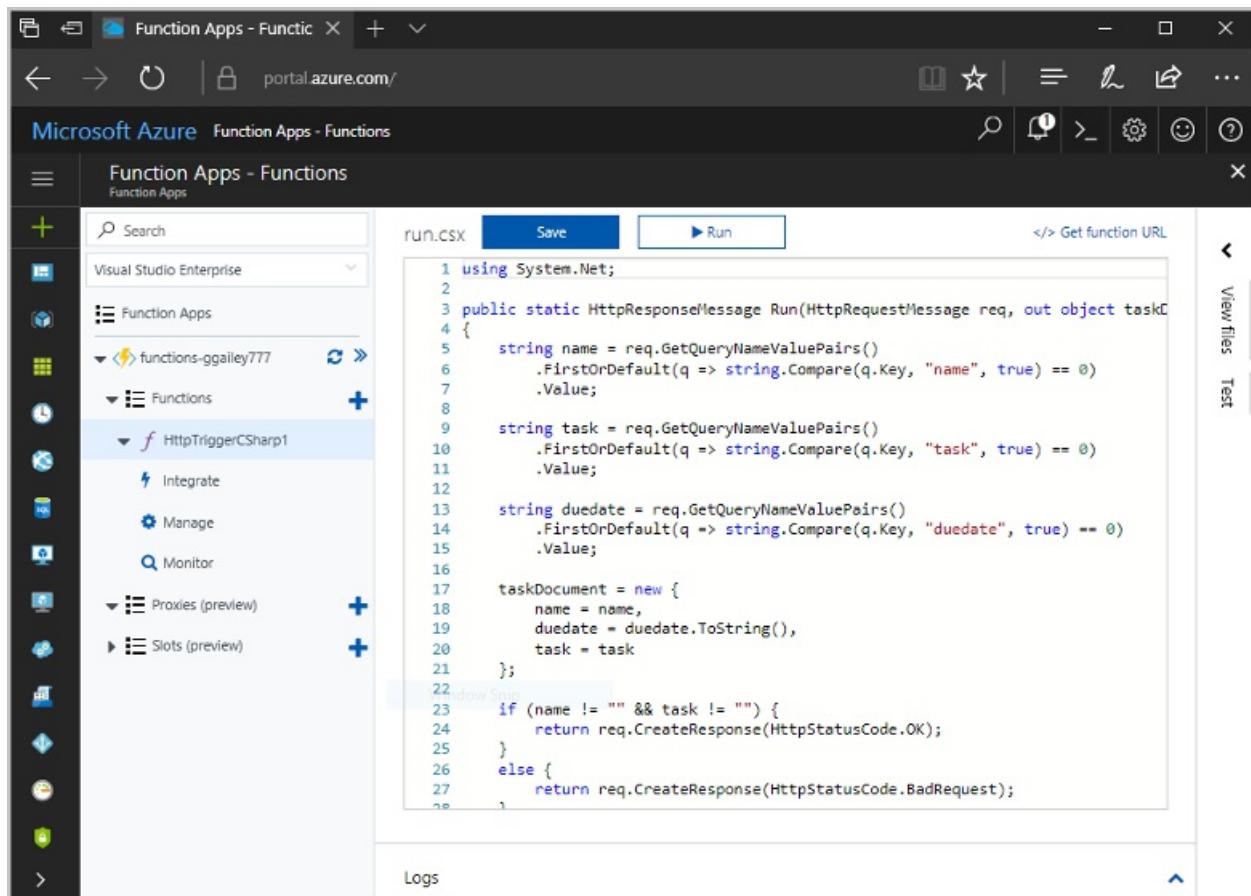
1/10/2018 • 4 min to read • [Edit Online](#)

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 topic, learn how to update an existing C# function to add an output binding that stores unstructured data in a Cosmos DB document.



The screenshot shows the Microsoft Azure Function Apps portal. On the left, the sidebar lists 'Function Apps - Functions' under 'Visual Studio Enterprise'. A sub-menu for 'functions-ggalley777' shows 'Functions' selected, with 'HttpTriggerCSharp1' highlighted. The main area is a code editor for 'run.csx' with the following C# code:

```
1 using System.Net;
2
3 public static HttpResponseMessage Run(HttpRequestMessage req, out object task)
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,
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     }
29 }
```

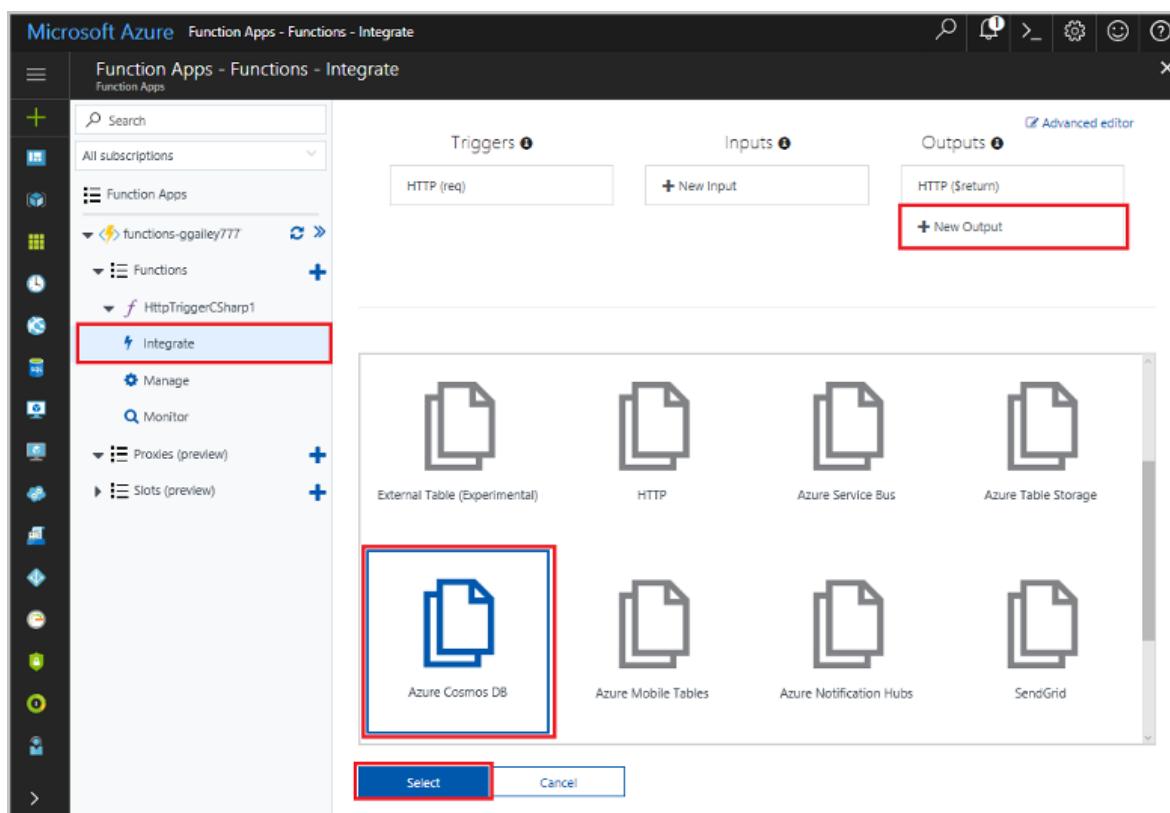
Prerequisites

To complete this tutorial:

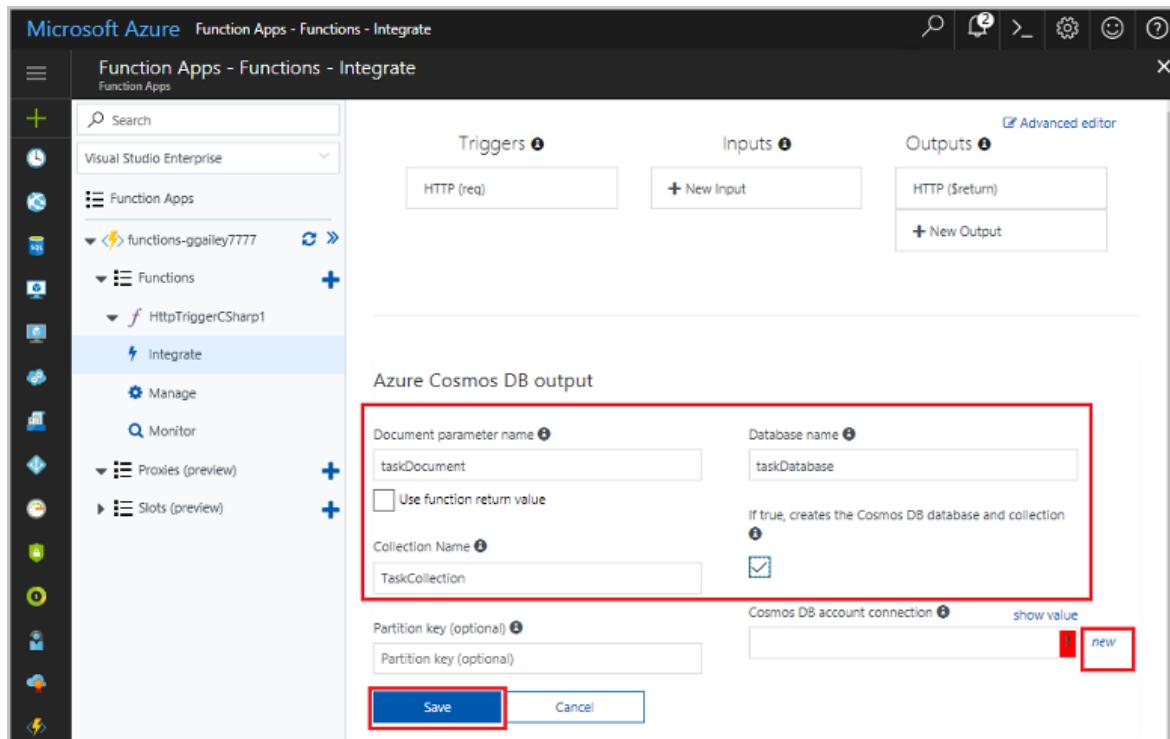
This topic uses as its starting point the resources created in [Create your first function from the Azure portal](#). If you haven't already done so, please complete these steps now to create your function app.

Add an output binding

1. 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**.



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

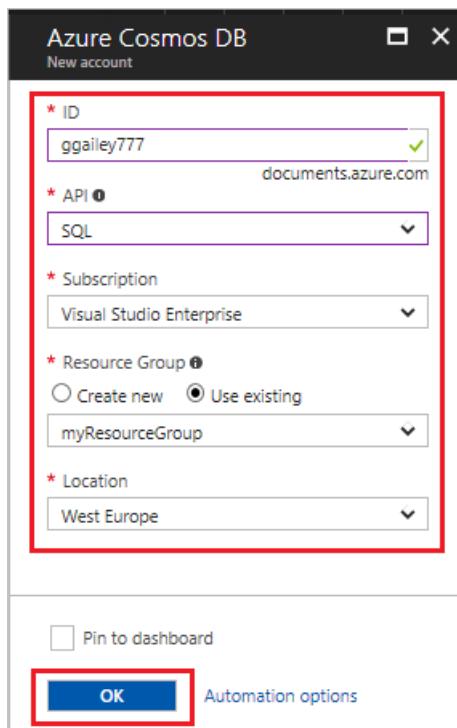


SETTING	SUGGESTED VALUE	DESCRIPTION
Document parameter name	taskDocument	Name that refers to the Cosmos DB object in code.

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

4. Select **New** next to the **Azure Cosmos DB document connection** label, and select **+ Create new**.

5. Use the **New account** settings as specified in the table:



SETTING	SUGGESTED VALUE	DESCRIPTION
ID	Name of database	Unique ID for the Azure Cosmos DB database
API	SQL	Select the SQL API. At this time, the Azure Cosmos DB trigger, input bindings, and output bindings work with SQL API and Graph API accounts only.
Subscription	Azure Subscription	Azure Subscription
Resource Group	myResourceGroup	Use the existing resource group that contains your function app.
Location	WestEurope	Select a location near to either your function app or to other apps that use the stored documents.

6. Click **OK** to create the database. It may take a few minutes to create the database. After the database is created, the database connection string is stored as a function app setting. The name of this app setting is

inserted in **Azure Cosmos DB account connection**.

- After the connection string is set, select **Save** to create the binding.

Update the function code

Replace the existing C# function code with the following code:

```
using System.Net;

public static HttpResponseMessage Run(HttpRequestMessage req, out object taskDocument, TraceWriter log)
{
    string name = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
        .Value;

    string task = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "task", true) == 0)
        .Value;

    string duedate = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "duedate", true) == 0)
        .Value;

    taskDocument = new {
        name = name,
        duedate = duedate.ToString(),
        task = task
    };

    if (name != "" && task != "") {
        return req.CreateResponse(HttpStatusCode.OK);
    }
    else {
        return req.CreateResponse(HttpStatusCode.BadRequest);
    }
}
```

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

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

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

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

The screenshot shows the Microsoft Azure Function Apps - Functions interface. On the left, there's a sidebar with icons for search, visual studio enterprise, function apps, functions, proxies (preview), and slots (preview). The main area has a code editor for `run.csx` containing C# code for an HTTP trigger. To the right of the code editor is a "Test" tab with fields for "HTTP method" (set to GET), "Query" parameters (name: Maria Anders, task: Shopping, duedate: 07/27/2017), and a "Request body" field with JSON input. Below the test tab is an "Output" section showing a green status message "Status: 200 OK". At the bottom right of the interface is a "Run" button. A red box highlights the "Test" tab, the "Run" button, and the "Logs" section below.

- 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 Microsoft Azure Function Apps - Functions - Integrate interface. On the left, there's a sidebar with icons for plus, minus, and a search bar containing "cosmos". The main area displays a search results list with two items: "Azure Cosmos DB" (Keywords: Cosmos) and "NoSQL (DocumentDB) accounts" (Keywords: Cosmos). A red box highlights the search bar and the first search result item.

- 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 Microsoft Azure Data Explorer (Preview) interface. On the left, there's a sidebar with icons for overview, activity log, access control (IAM), tags, diagnose and solve problems, quick start, and data explorer (highlighted with a red box). The main area has a "COLLECTIONS" section showing a "taskDatabase" node with a "TaskCollection" child node. Under "TaskCollection", there's a "Documents" section with a list of documents. One document is selected, showing its properties in the "SELECT * FROM c" pane, which includes fields like id, name, duedate, task, and attachments. A red box highlights the "Data Explorer (Preview)" section in the sidebar and the selected document in the main pane.

You have successfully added a binding to your HTTP trigger that stores unstructured data in a 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

Learn how to create functions with other kinds of triggers or how to integrate functions with other Azure services.

- [Create a function that runs on a schedule](#)
- [Create a function triggered by Storage queue messages](#)
- [Create a function triggered by a generic webhook](#)
- [Create a function triggered by a GitHub webhook](#)
- [Add messages to an Azure Storage queue using Functions](#)
- [Store unstructured data in Azure Cosmos DB using Functions](#)

For more information about binding to a Cosmos DB database, see [Azure Functions Cosmos DB bindings](#).

Create a function that integrates with Azure Logic Apps

1/19/2018 • 9 min 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 Microsoft Cognitive Services on Azure to analyze sentiment 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 for a logic app named "TweetSentiment". The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Development Tools (Logic App Designer, Logic App Code View, Versions, API Connections, Quick Start Guides, Release notes), Settings (Integration account, Access control configuration, Access keys), and Help. The main area has tabs for Run Trigger, Refresh, Edit, Delete, Disable, Update Schema, and Clone. The "Essentials" tab is selected, displaying details about the resource group (myResourceGroup), location (East US), subscription (Visual Studio Enterprise), and plan (Consumption). It also shows the trigger history for the "When_a_new_tweet_is_posted" trigger, which fired 20 times successfully. The "Runs history" section shows the last five runs, all successful, with start times ranging from May 13, 2018, at 10:04 to 10:17.

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.

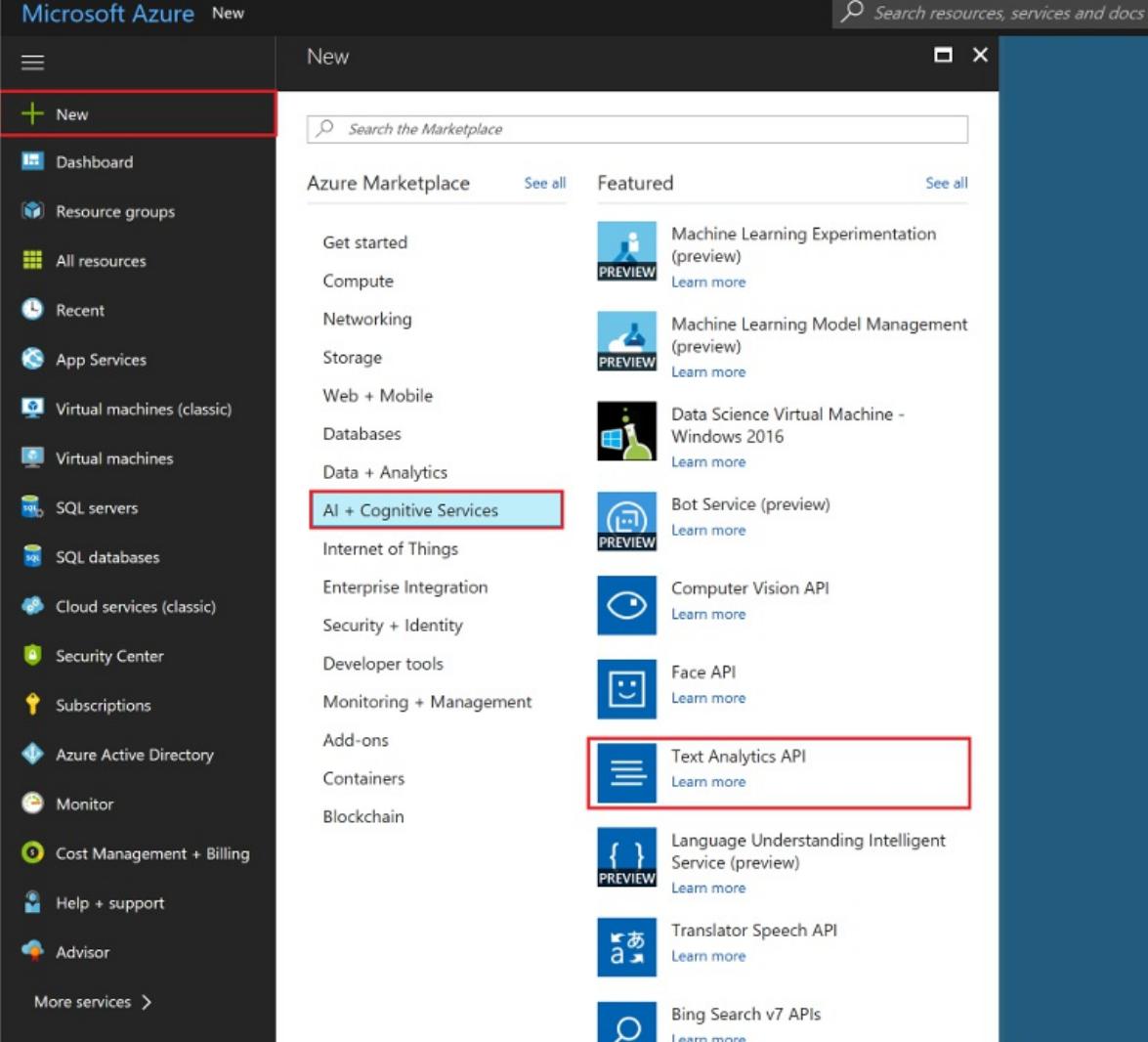
Prerequisites

- An active [Twitter](#) account.
- An [Outlook.com](#) account (for sending notifications).
- This 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, 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 the **New** button found on the upper left-hand corner of the Azure portal.
3. Click **AI + Analytics > Text Analytics API**. Then, use the settings as specified in the table, accept the terms, and check **Pin to dashboard**.



The screenshot shows the Microsoft Azure portal's 'New' blade. On the left, a sidebar lists various services like Dashboard, Resource groups, and App Services. The 'New' button is highlighted with a red box. In the main area, a search bar says 'Search the Marketplace'. Below it, there are tabs for 'Azure Marketplace' and 'Featured'. A list of services is shown, with 'AI + Cognitive Services' being the current category, indicated by a blue box. Underneath, the 'Text Analytics API' service is highlighted with a red box. Other visible services include Machine Learning Experimentation (preview), Machine Learning Model Management (preview), Data Science Virtual Machine - Windows 2016, Bot Service (preview), Computer Vision API, Face API, Language Understanding Intelligent Service (preview), Translator Speech API, and Bing Search v7 APIs.

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. After it is created, select your new Cognitive Services resource pinned to the dashboard.

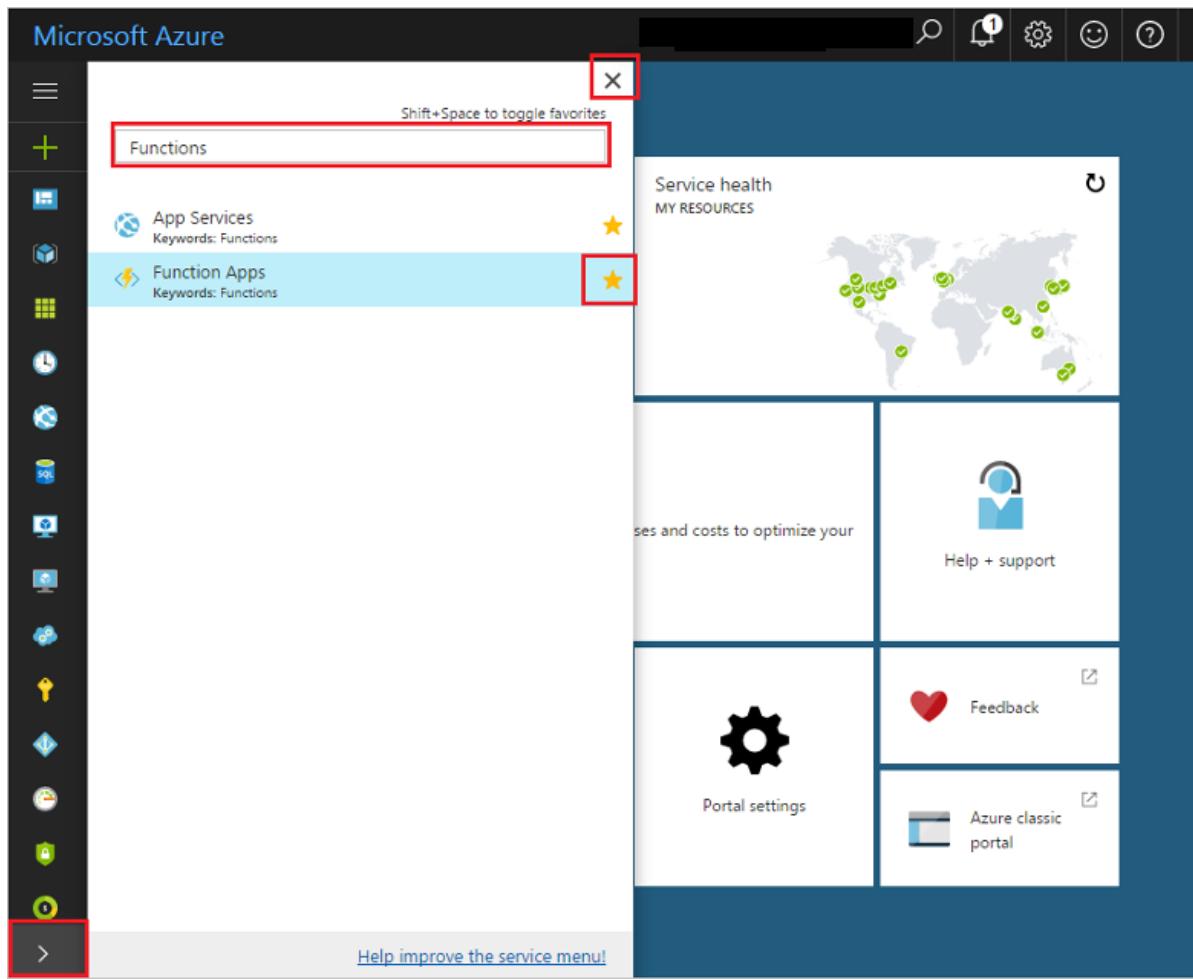
5. In the left navigation column, click **Keys**, and then copy the value of **Key 1** and save it. You use this key to connect the logic app to your Cognitive Services API.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a navigation sidebar with various service icons like Dashboard, Resource groups, and App Services. A red box highlights the 'Keys' option under 'RESOURCE MANAGEMENT'. The main content area is titled 'MyTextAnalyticsAPI - Keys' and shows a 'NAME' field with 'MyTextAnalyticsAPI'. Below it, the 'KEY 1' field is highlighted with a red box and contains the placeholder '<your-api-key>'. There are also 'Regenerate Key1' and 'Regenerate Key2' buttons at the top. A note at the top says 'Notice: It may take up to 10 minutes for the newly (re)generated keys to take effect.'

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

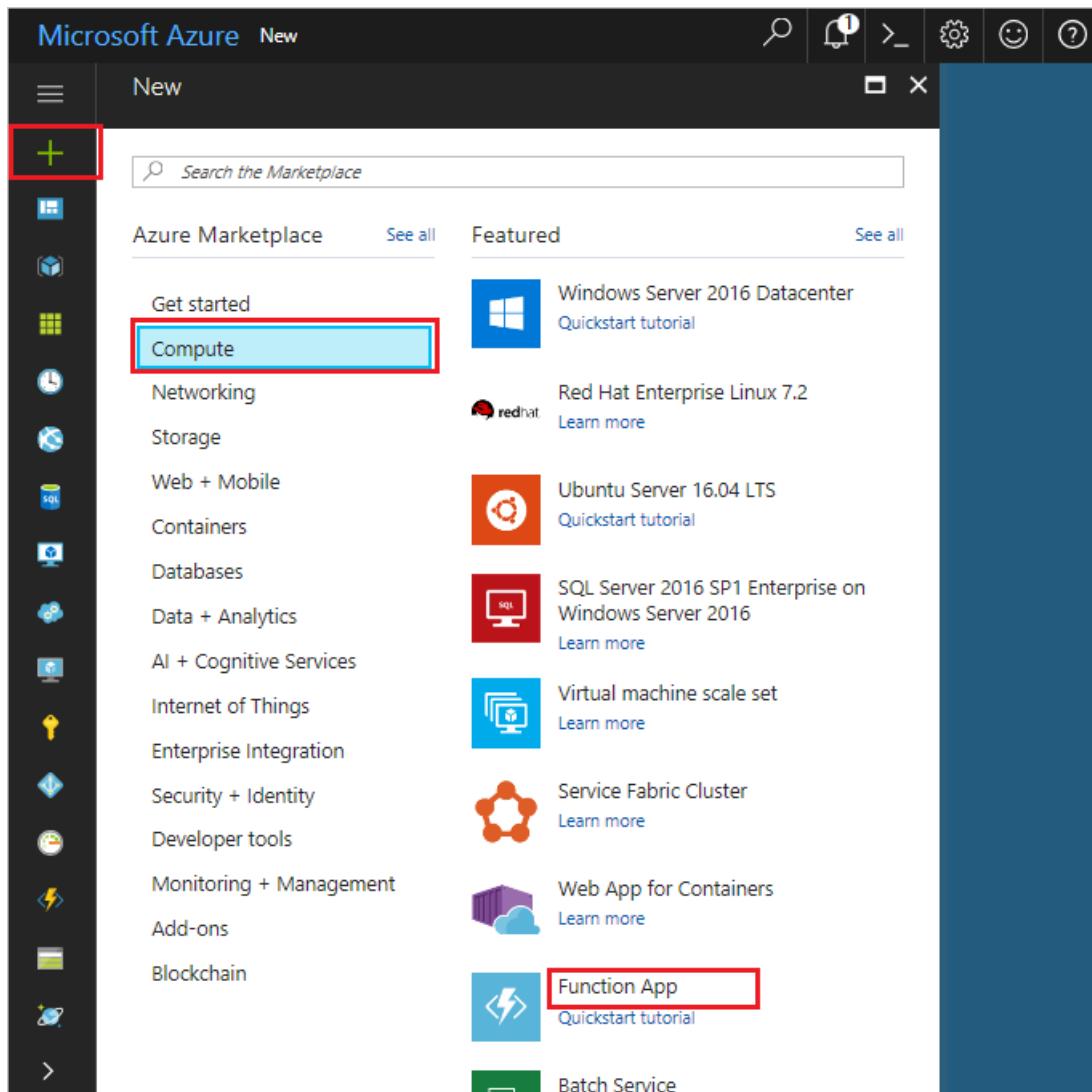
The screenshot shows the 'Function Apps' blade in the Microsoft Azure portal. The left sidebar has a red box around the 'Functions' icon. The main content area shows a table of function apps. The first row of the table is as follows:

NAME	SUBSCRIPTION ID	RESOURCE GROUP	LOCATION
functions-ggailey777	Visual Studio Enterprise	functions-ggailey777	southcentralus

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.

1. Click the **New** 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
functions-ggailey777 .azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group ⓘ
 Create new Use existing
functions-ggailey777

* OS
Windows Linux

* Hosting Plan ⓘ
Consumption Plan

* Location
West Europe

* Storage ⓘ
 Create new Use existing
functionsggaile87e8

Application Insights ⓘ On

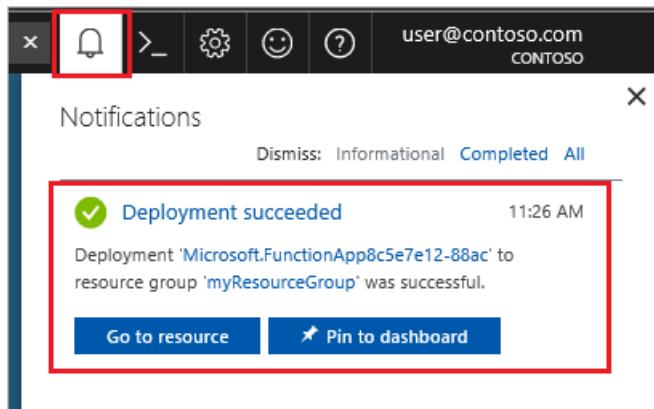
Pin to dashboard

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.
Resource Group	myResourceGroup	Name for the new resource group in which to create your function app.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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 <i>serverless</i> hosting, you only pay for the time your functions run.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

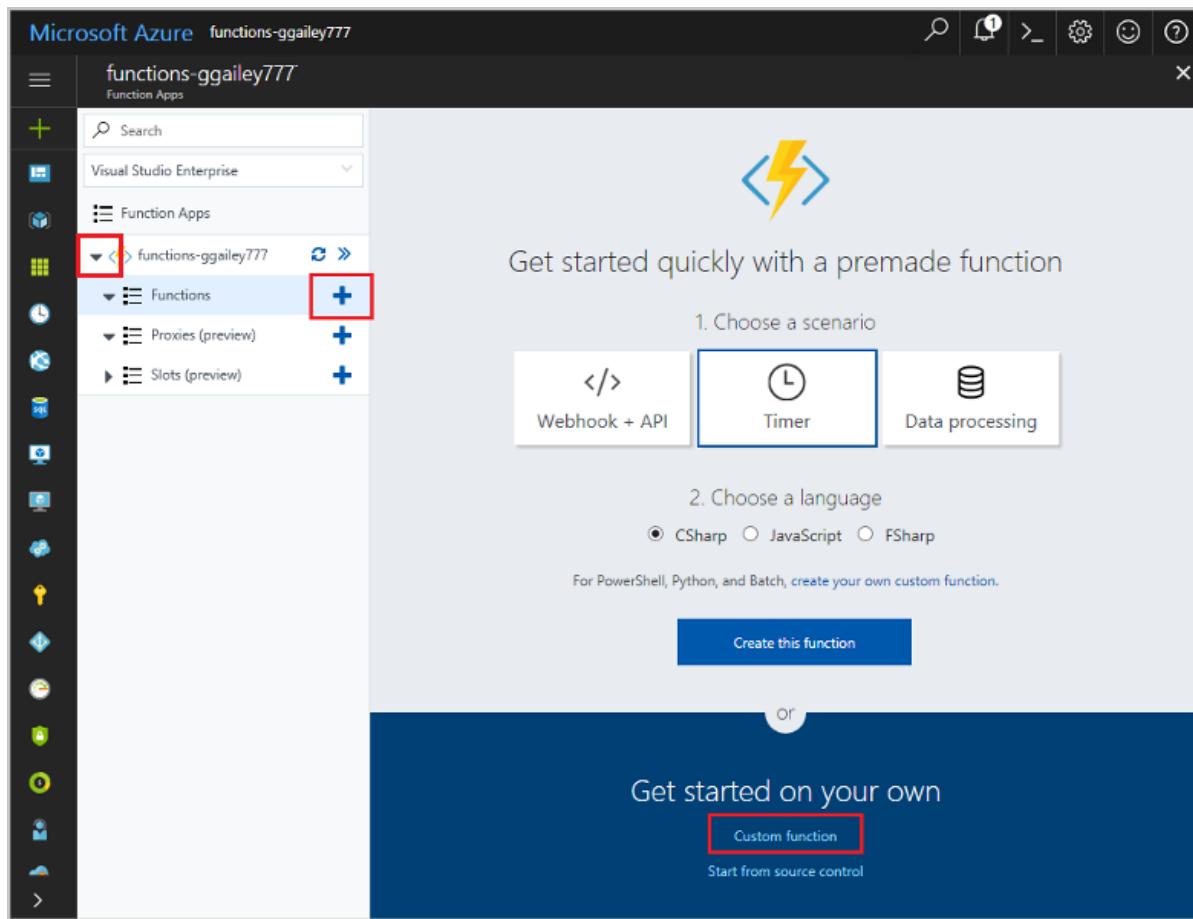
3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.



Clicking **Go to resource** takes you to your new function app.

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 **Custom function**. This displays the complete set of function templates.



2. In the search field, type `http` and then choose **C#** for the HTTP trigger template.

Choose a template below or go to the quickstart

Language: All Scenario: All

HTTP trigger

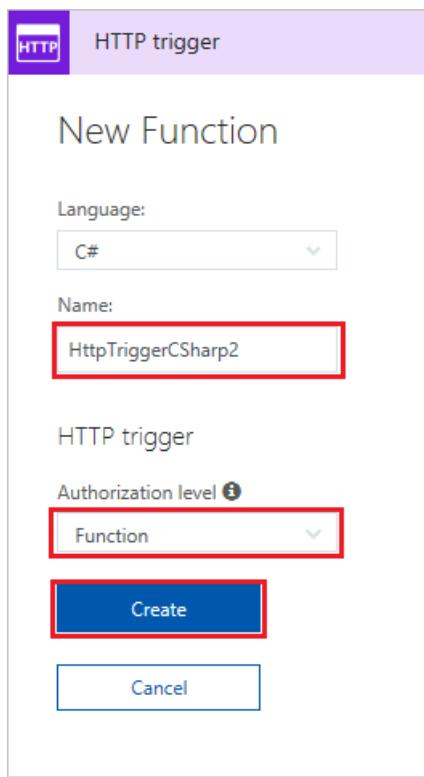
A function that will be run whenever it receives an HTTP request, responding based on data in the body or query string

Batch C# F# JavaScript PowerShell
Python TypeScript

HTTP GET

A function that fetches entities from a Storage table when it

3. Type a **Name** for your function, choose `Function` for **Authentication level**, and then select **Create**.



This creates a C# script function using the HTTP Trigger template. Your code appears in a new window as `run.csx`.

4. Replace the contents of the `run.csx` file with the following code, then click **Save**:

```
using System.Net;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
{
    // The sentiment category defaults to 'GREEN'.
    string category = "GREEN";

    // Get the sentiment score from the request body.
    double score = await req.Content.ReadAsAsync<double>();
    log.Info(string.Format("The sentiment score received is '{0}'.",
        score.ToString()));

    // Set the category based on the sentiment score.
    if (score < .3)
    {
        category = "RED";
    }
    else if (score < .6)
    {
        category = "YELLOW";
    }
    return req.CreateResponse(HttpStatusCode.OK, category);
}
```

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

5. 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.

```

1 using System.Net;
2
3 public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
4 {
5     // The sentiment category defaults to 'GREEN'.
6     string category = "GREEN";
7
8     // Get the sentiment score from the request body.
9     double score = await req.Content.ReadAsAsync<double>();
10    log.Info(string.Format("The sentiment score received is '{0}'.", score.ToString()));
11
12    // Set the category based on the sentiment score.
13    if (score < .3)
14    {
15        category = "RED";
16    }
17    else if (score < .6)
18    {
19        category = "YELLOW";
20    }
21
22    return req.CreateResponse(HttpStatusCode.OK, category);
23 }

```

Logs

2017-05-13T05:05:12 No new trace in the past 1 min(s).
2017-05-13T05:05:46.870 Script for function 'CategorizeSentiment' changed. Reloading.
2017-05-13T05:05:47.076 Compilation succeeded.
2017-05-13T05:06:03.170 Function started (Id=057192ff-17fa-442b-9912-65eaf37c544a)
2017-05-13T05:06:03.310 The sentiment score received is '0.2'.
2017-05-13T05:06:03.310 Function completed (Success, Id=057192ff-17fa-442b-9912-65eaf37c544a, Duration: 00:00:00.141)

Output

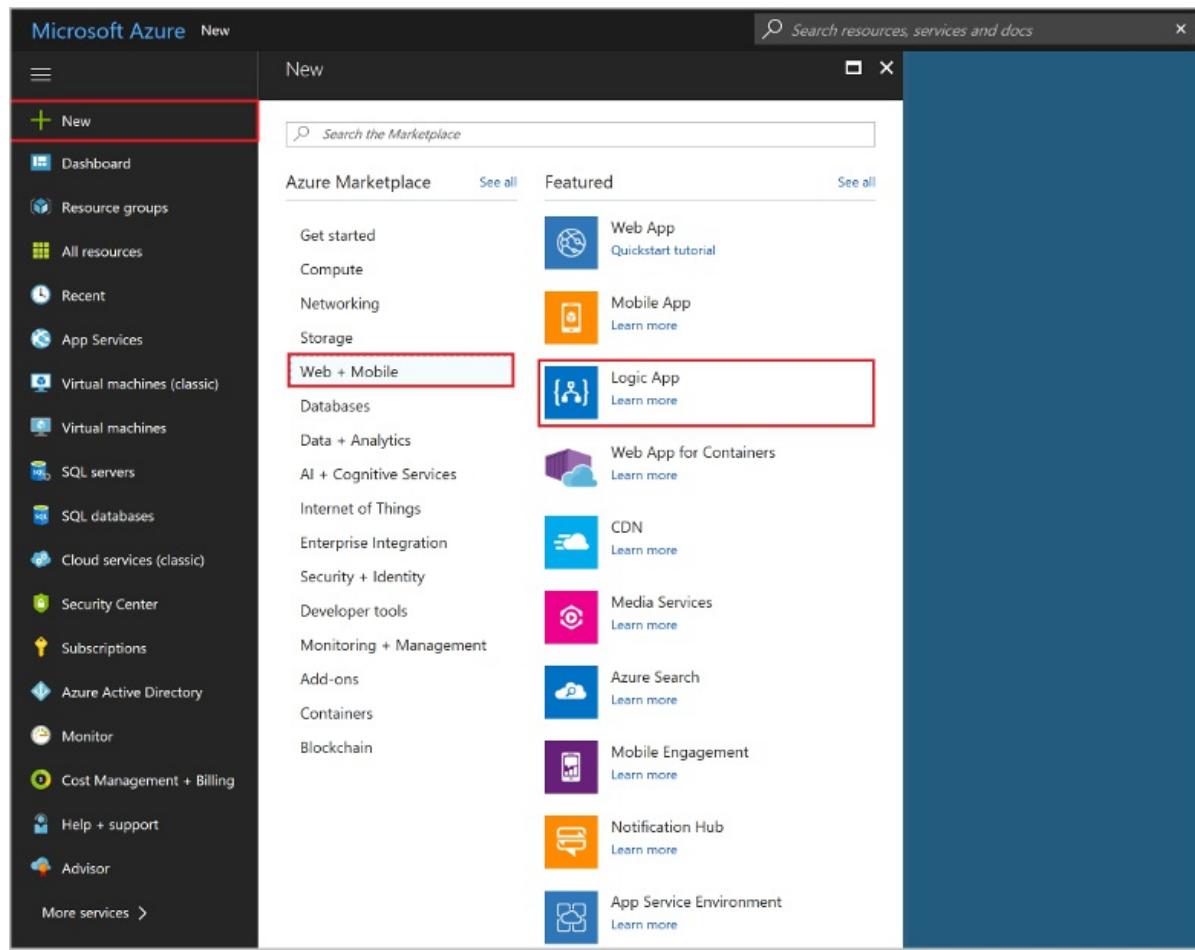
Status: 200 OK
"RED"

Run

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.

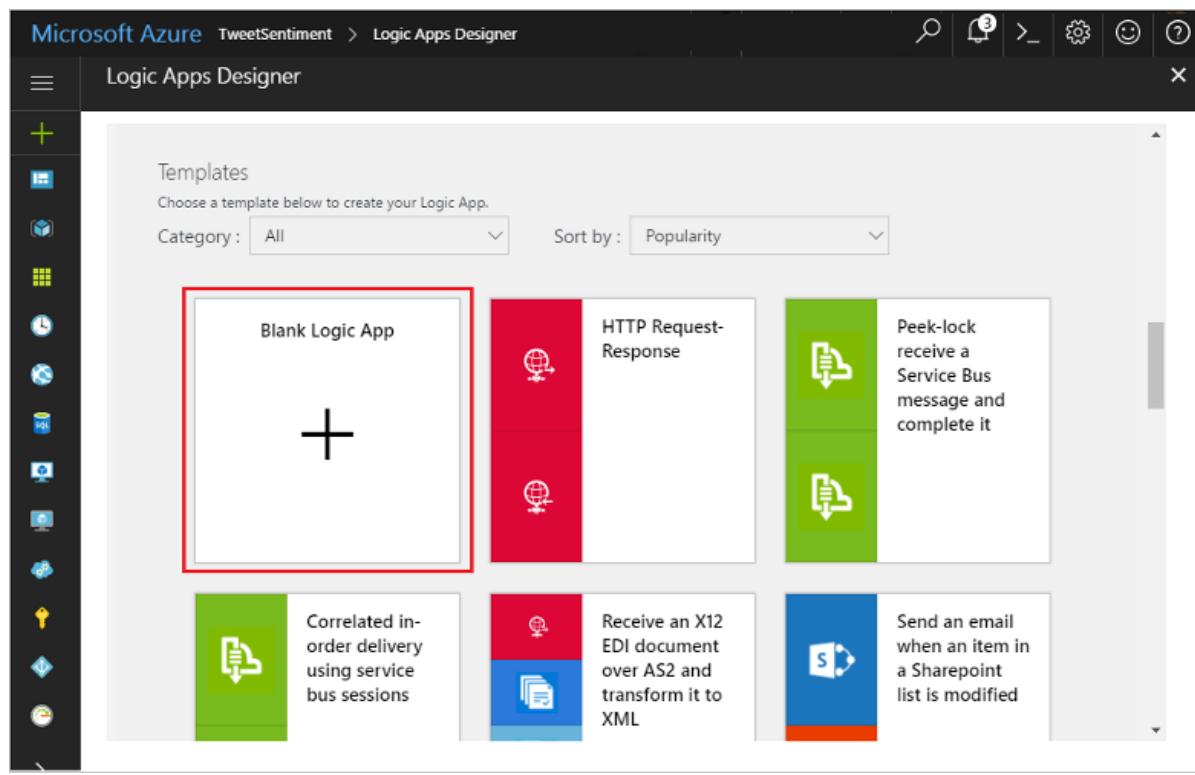
Create a logic app

1. In the Azure portal, click the **New** button found on the upper left-hand corner of the Azure portal.
2. Click **Enterprise Integration > Logic App**. Then, use the settings as specified in the table, check **Pin to dashboard**, and click **Create**.
3. Then, type a **Name** like `TweetSentiment`, use the settings as specified in the table, accept the terms, and check **Pin to dashboard**.



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. Choose **Pin to dashboard**, and then 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.

1. 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.
2. Use the Twitter trigger settings as specified in the table.

SETTING	SUGGESTED VALUE	DESCRIPTION
---------	-----------------	-------------

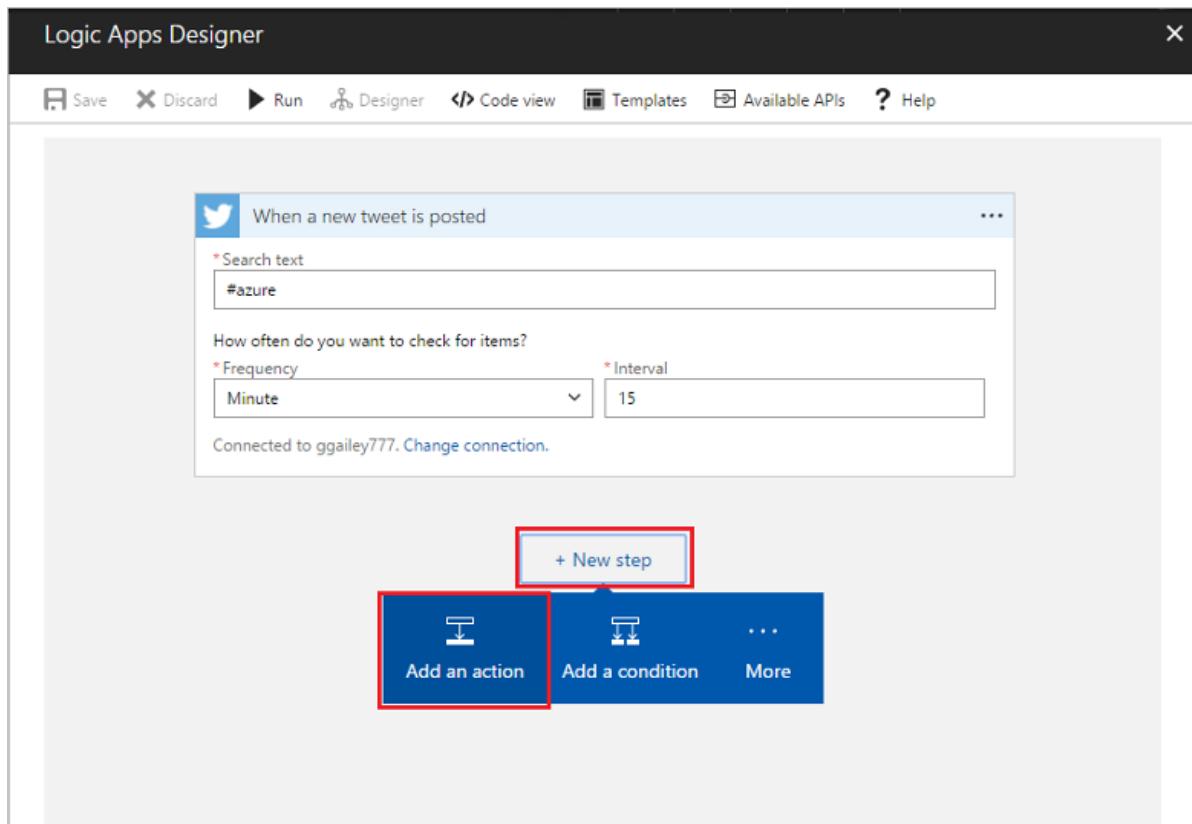
SETTING	SUGGESTED VALUE	DESCRIPTION
Search text	#Azure	Use a hashtag that is popular enough to generate new tweets in the chosen interval. When using the Free tier and your hashtag is too popular, you can quickly use up the transaction quota in your Cognitive Services API.
Frequency	Minute	The frequency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

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

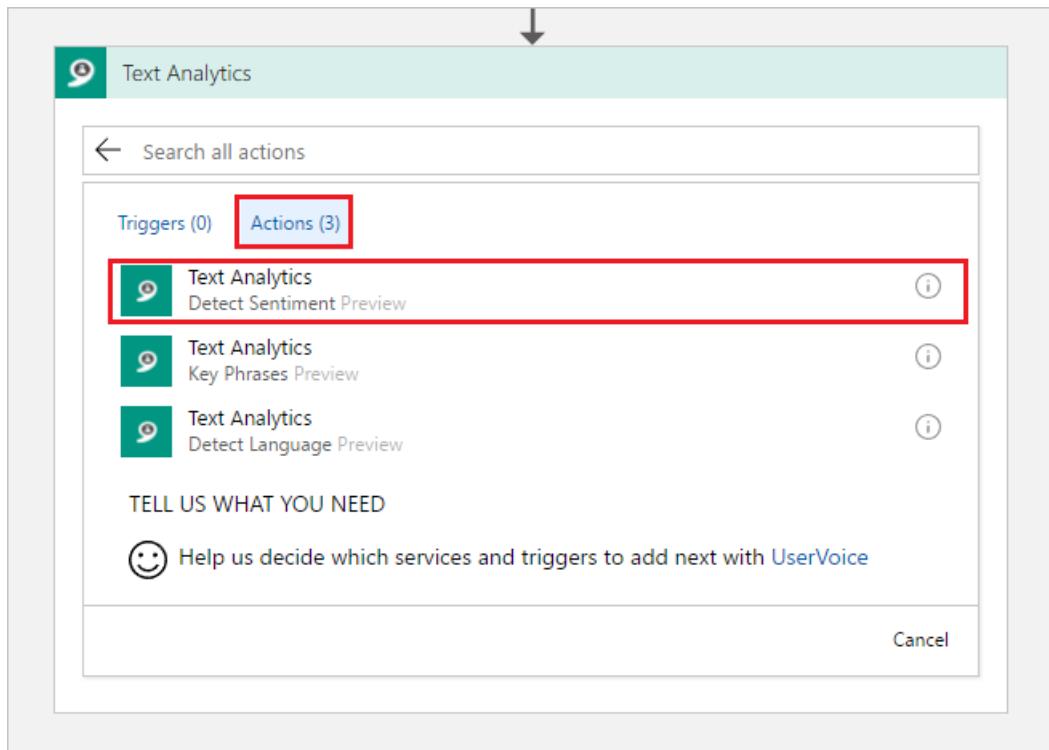
Now your app is connected to Twitter. Next, you connect to text analytics to detect the sentiment of collected tweets.

Add sentiment detection

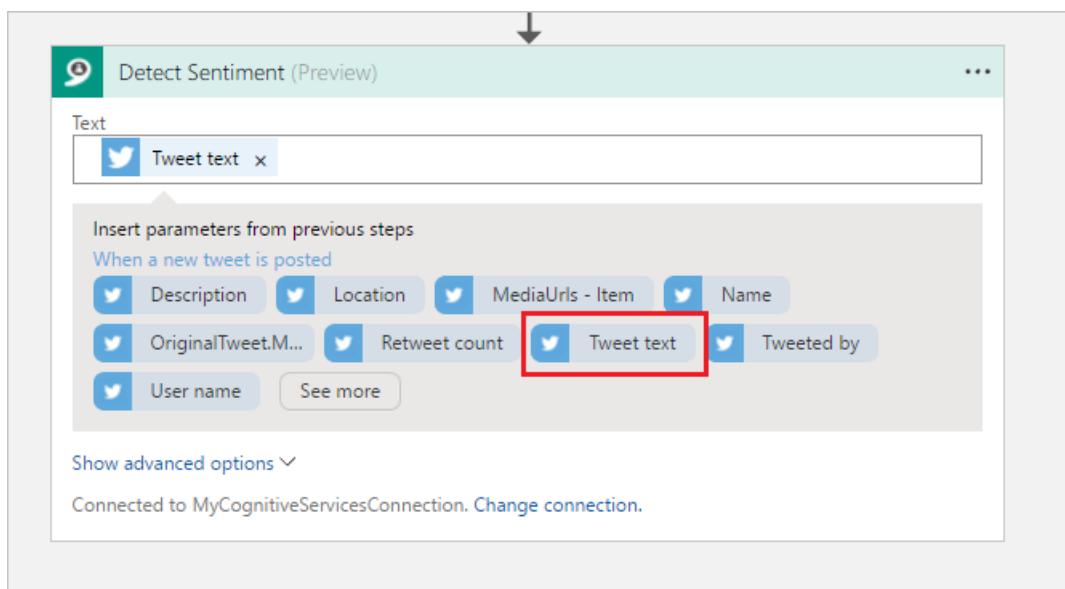
1. Click **New Step**, and then **Add an action**.



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



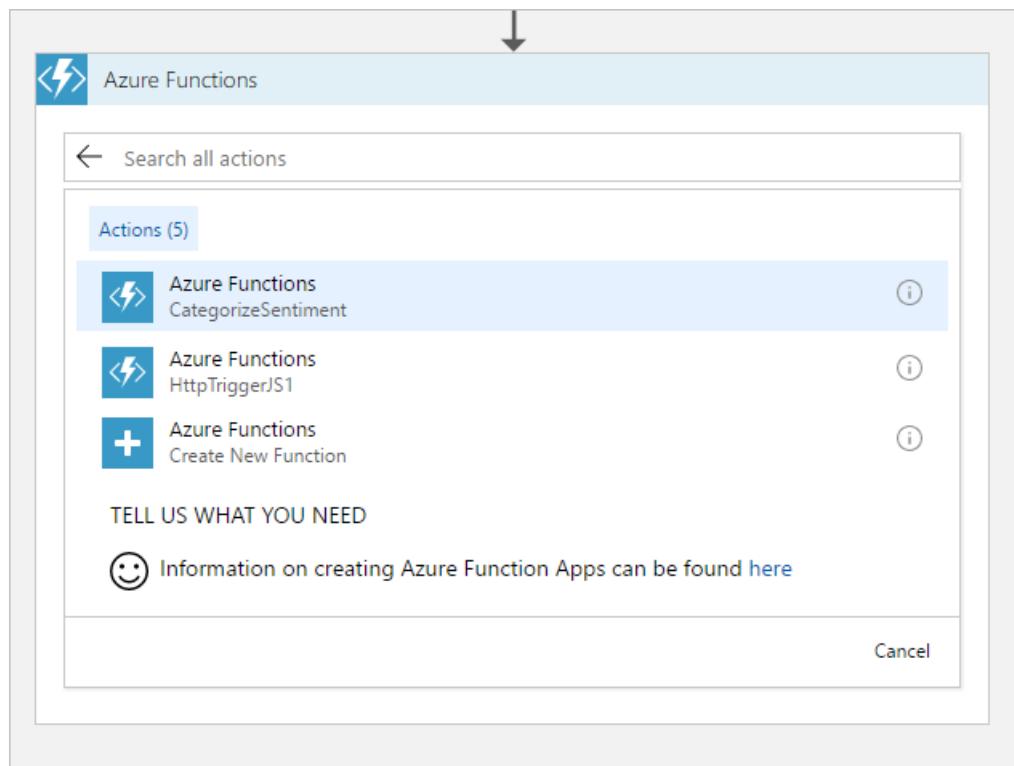
3. Type a connection name such as `MyCognitiveServicesConnection`, paste the key for your Cognitive Services API that you saved, and click **Create**.
4. Click **Text to analyze** > **Tweet text**, and then click **Save**.



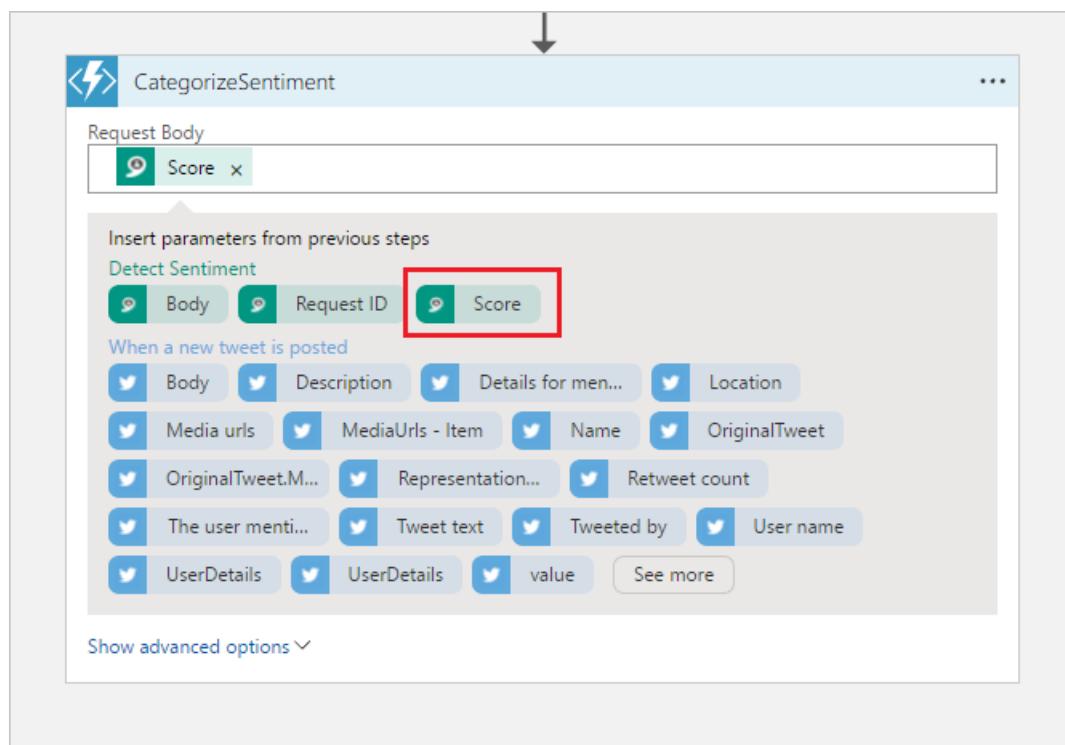
Now that sentiment detection is configured, you can add a connection to your function that consumes the sentiment score output.

Connect sentiment output to your function

1. In the Logic Apps Designer, click **New step** > **Add an action**, and then click **Azure Functions**.
2. Click **Choose an Azure function**, select the **CategorizeSentiment** function you created earlier.



3. 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.

The screenshot shows the configuration of a 'Condition' step in Microsoft Flow. The condition is set to check if the 'Body' field is equal to 'RED'. Below the condition, there is a list of parameters from previous steps, including 'CategorizeSentiment' (Body, Headers, Status code), 'Detect Sentiment' (Body, Request ID, Score), and 'When a new tweet is posted' (Body, Description, Details for men..., Location, Media urls, MediaUrls - Item, Name, OriginalTweet, OriginalTweet.M..., Representation..., Retweet count, The user menti..., Tweet text, Tweeted by, User name, UserDetails, UserDetails, value). At the bottom, there are links to 'Edit in advanced mode' and 'Collapse condition'. A red box highlights the 'Add an action' button, which is located in the 'IF YES, DO NOTHING' section below the condition.

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

If true

Choose an action

outlook.com

Connectors

Outlook.co...

See more

Triggers (7) Actions (28)

See more

Action	Description	Info
Outlook.com	Create contact	(i)
Outlook.com	Create event (V1)	(i)
Outlook.com	Create event (V2) Preview	(i)
Outlook.com	Send an email	(i)
Outlook.com	Send approval email	(i)
Outlook.com	Delete contact	(i)
Outlook.com	Delete email	(i)

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.

If true

Send an email

*Body

Location x Tweet text x

Insert parameters from previous steps

HttpTriggerCSharp1

Body Status code

Detect Sentiment

Request ID Score

When a new tweet is posted

Description Location MediaUrls - Item Name

OriginalTweet... Retweet count Tweet text Tweeted by

User name See more

* Subject

Negative Sentiment Detected

* To

<your-email-address>

Show advanced options ▾

Add an action ••• More

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.

5. Click **Save**.

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

Test the workflow

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

STATUS	START TIME	IDENTIFIER	DURATION
Succeeded	10/12/2017 6:19 PM	08586937499331052...	740 Milliseco...
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

3. (Optional) Click one of the runs to see details of the execution.
4. Go to your function, view the logs, and verify that sentiment values were received and processed.

```

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

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

```
return req.CreateResponse(HttpStatusCode.OK, category);
```

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. This stops the logic app 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 icons. The main area has a title bar with the logic app name and a search bar. Below the title bar, there are several tabs: 'Overview' (which is selected and highlighted with a blue box), 'Activity log', 'Access control (IAM)', and 'Tags'. To the right of these tabs, there are buttons for 'Run Trigger', 'Edit', 'Delete', and 'Disable' (which is also highlighted with a red box). Further down, there are sections for 'Definition', 'Runs history', and 'Trigger history'. The 'Runs history' section contains a table with columns: STATUS, START TIME, IDENTIFIER, and DURATION. It lists 12 successful runs from October 12, 2017, with durations ranging from 1.29 to 7.49 seconds. The 'Trigger history' section shows a table with columns: STATUS, START TIME, and FIRED. It lists 12 successful runs for the trigger 'When a new tweet is posted', all fired on October 12, 2017, by user 'Fred'.

STATUS	START TIME	IDENTIFIER	DURATION
Succeeded	10/12/2017 6:19 PM	0858693749331852...	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

STATUS	START TIME	FIRED
Skipped	10/12/2...	
Succeeded	10/12/2...	Fred
Succeeded	10/12/2...	
Succeeded	10/12/2...	Fred
Succeeded	10/12/2...	
Succeeded	10/12/2...	Fred
Succeeded	10/12/2...	
Succeeded	10/12/2...	Fred
Succeeded	10/12/2...	
Succeeded	10/12/2...	Fred
Succeeded	10/12/2...	

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

11/15/2017 • 6 min 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 NodeJS, 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 'Integrate' blade for a function named 'HttpTriggerCSharp1'. On the left, the navigation bar shows 'ProxiesBackEnd' selected. Under 'Functions', 'HttpTriggerCSharp1' is selected, and 'Integrate' is highlighted. The main area displays the 'HTTP trigger' settings:

- 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), DELETE, PATCH, OPTIONS, POST, HEAD, PUT, TRACE

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 and webhook 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".

The screenshot shows the 'Proxies' creation form in the Azure portal. The 'Name' field contains 'HelloProxy'. The 'Route template' field contains '/api/hello'. The 'Allowed HTTP methods' dropdown is set to 'All methods'. The 'Backend URL' field contains 'https://%HELLO_HOST%/api/hello'. At the bottom is a blue 'Create' button.

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/hello	Determines what route is used to invoke this proxy
Backend URL	https://%HELLO_HOST%/api/hello	Specifies the endpoint to which the request should be proxied

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:
a. `https://YOURPROXYAPP.azurewebsites.net/api/hello?name="Proxies"`
 - b. For a function with authorization use:
a. `https://YOURPROXYAPP.azurewebsites.net/api/hello?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/hello"  
      },  
      "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/hello"
            },
            "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 and webhook bindings](#)
- [Working with Azure Functions Proxies](#)
- [Documenting an Azure Functions API \(preview\)](#)

Create an OpenAPI definition for a function

12/18/2017 • 8 min to read • [Edit Online](#)

REST APIs are often described 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.

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 so that the function can be called from other apps and services.

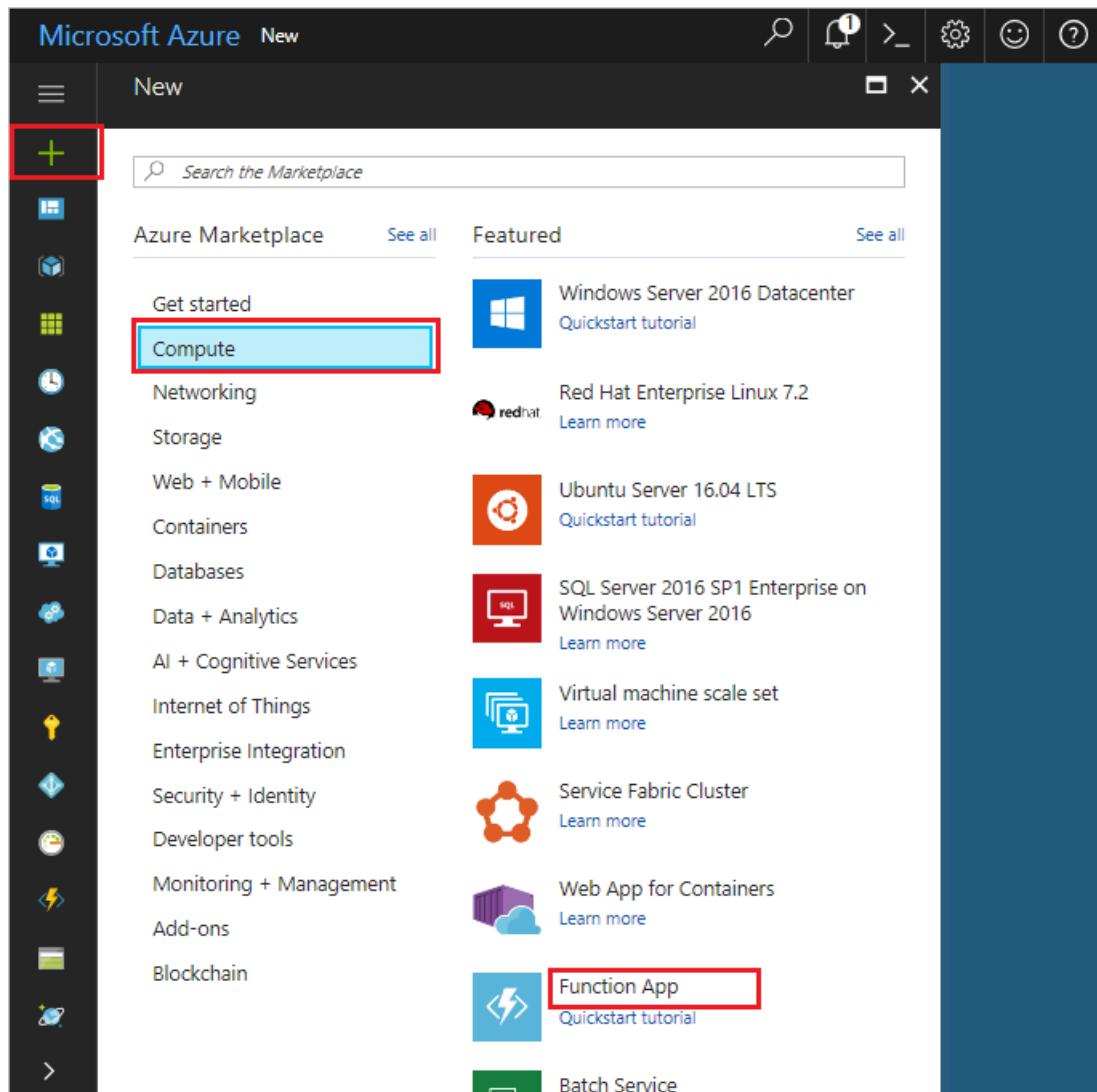
In this tutorial, you learn how to:

- Create a function in Azure
- Generate an OpenAPI definition using OpenAPI tools
- Modify the definition to provide additional metadata
- Test the definition by calling the function

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, scaling, and sharing of resources.

1. Click the **New** 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 X

Create

* App name
functions-ggailey777 ✓
.azurewebsites.net

* Subscription
Visual Studio Enterprise

* Resource Group i
 Create new Use existing
functions-ggailey777 ✓

* OS Windows Linux

* Hosting Plan i
Consumption Plan

* Location
West Europe

* Storage i
 Create new Use existing
functions-ggaile87e8 ✓

Application Insights i On Off

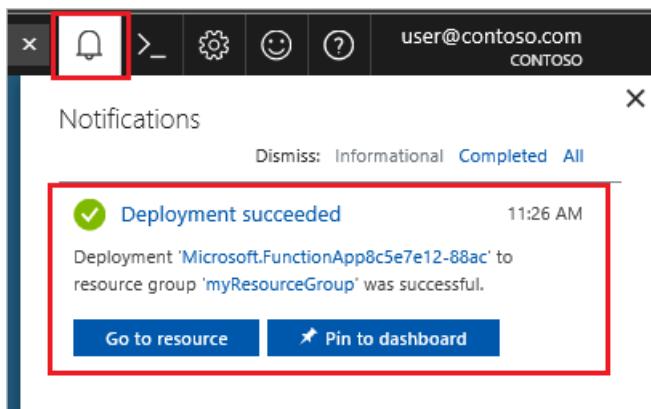
Pin to dashboard

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.
OS	Windows	Serverless hosting is currently only available when running on Windows. For Linux hosting, see Create your first function running on Linux using the Azure CLI .

SETTING	SUGGESTED VALUE	DESCRIPTION
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.
Location	West Europe	Choose a region near you or near other services your functions access.
Storage account	Globally unique name	Name of the new 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.

3. Click **Create** to provision and deploy the new function app. You can monitor the status of the deployment by clicking the Notification icon in the upper-right corner of the portal.

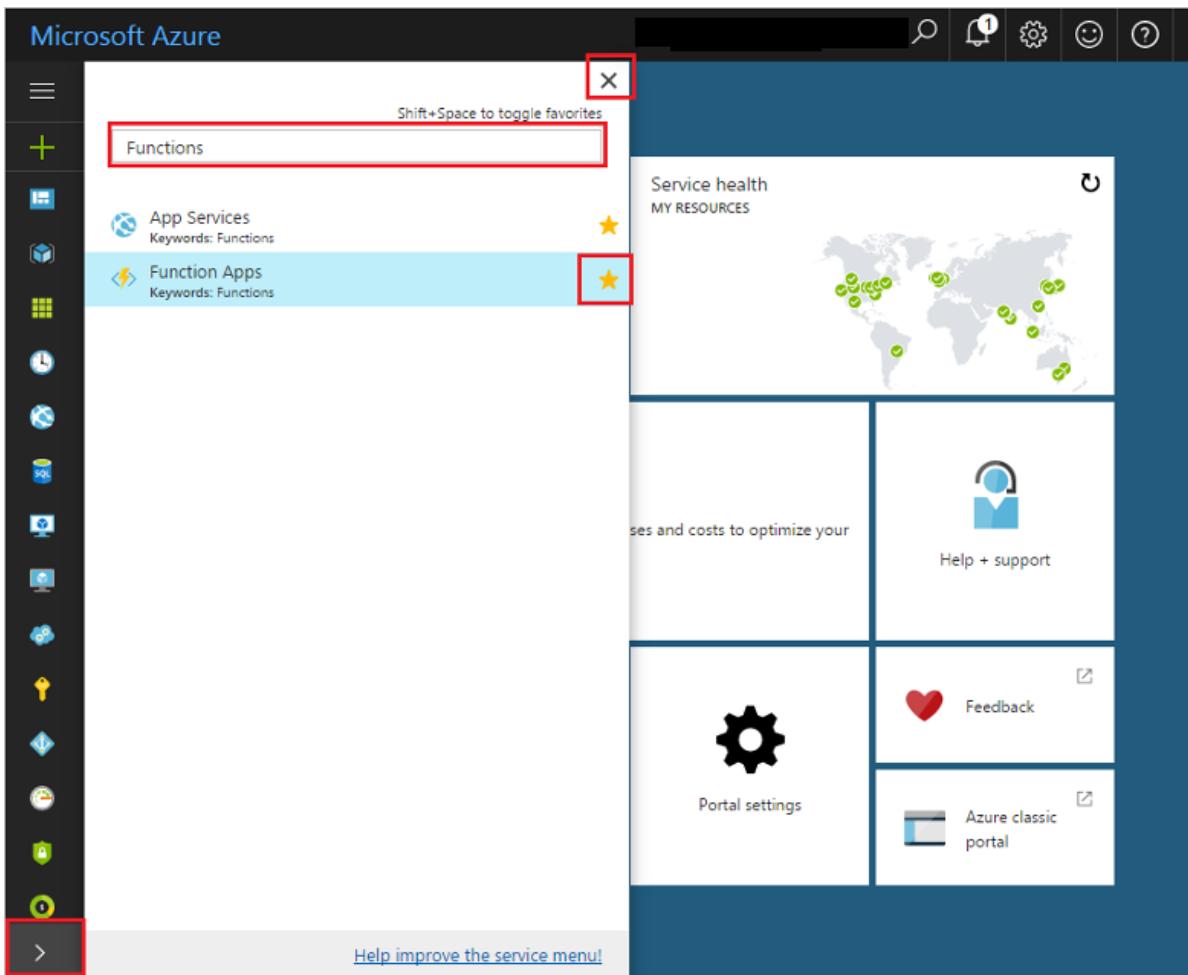


Clicking **Go to resource** takes you to your new function app.

Favorite Functions in the portal

If you haven't already done so, add Function Apps to your favorites in the Azure portal. This makes it easier to find your function apps. If you have already done this, skip to the next section.

1. Log in to the [Azure portal](#).
2. Click the arrow at the bottom left to expand all services, type **Functions** in the **Filter** field, and then click the star next to **Function Apps**.



This adds the Functions icon to the menu on the left of the portal.

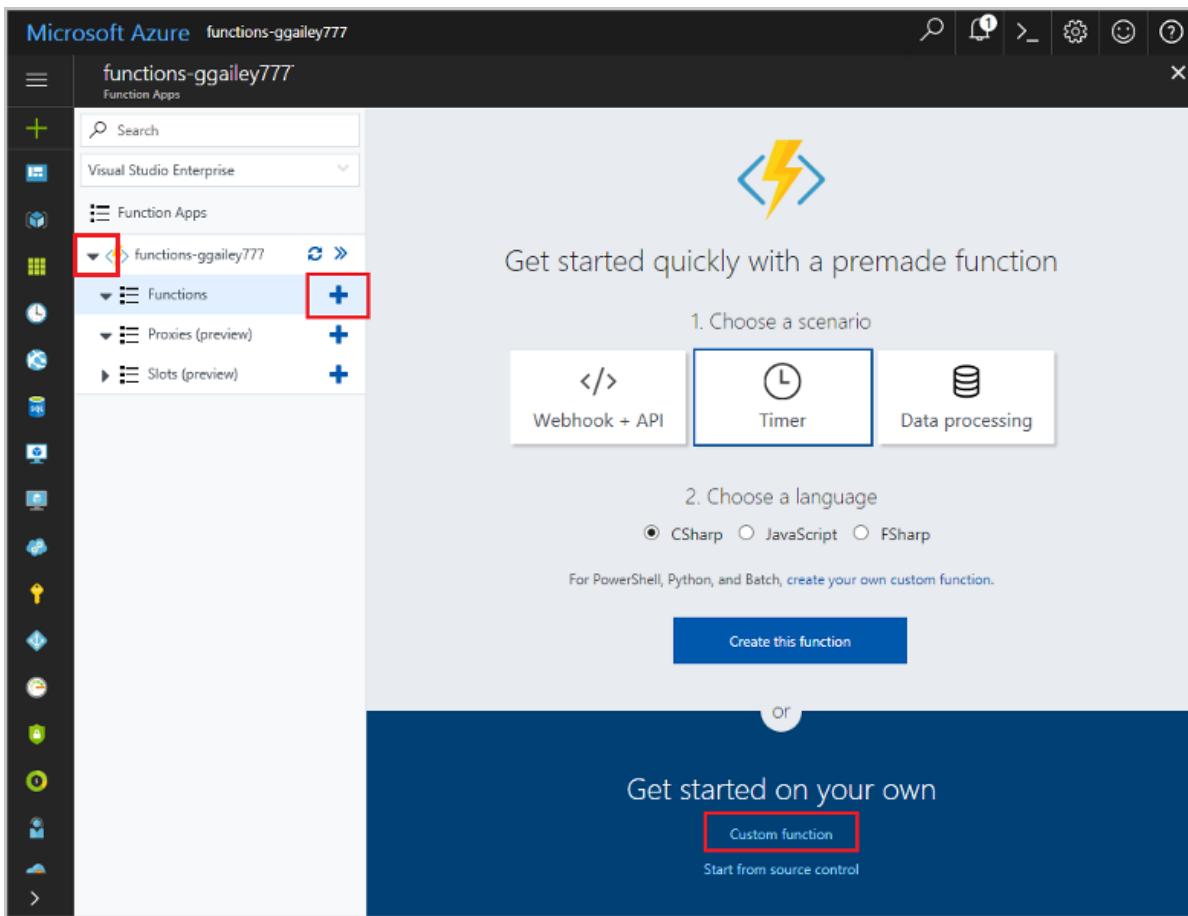
3. Close the menu, then scroll down to the bottom to see the Functions icon. Click this icon to see a list of all your function apps. Click your function app to work with functions in this app.

The screenshot shows the Microsoft Azure portal with the 'Function Apps' menu expanded. The left sidebar has a red box around the 'Function Apps' icon. The main content area shows a list of function apps under the heading 'Function Apps'. One item, 'functions-ggailey777', is listed with a yellow lightning bolt icon. The rest of the sidebar and main content area are visible.

Create the function

This tutorial uses an HTTP triggered function that takes two parameters: the estimated time to make a turbine repair (in hours); and 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.

1. Expand your function app and select the + button next to **Functions**. If this is the first function in your function app, select **Custom function**. This displays the complete set of function templates.



2. In the search field, type `http` and then choose **C#** for the HTTP trigger template.

Choose a template below or go to the quickstart

X

Language: AllScenario: All

HTTP trigger

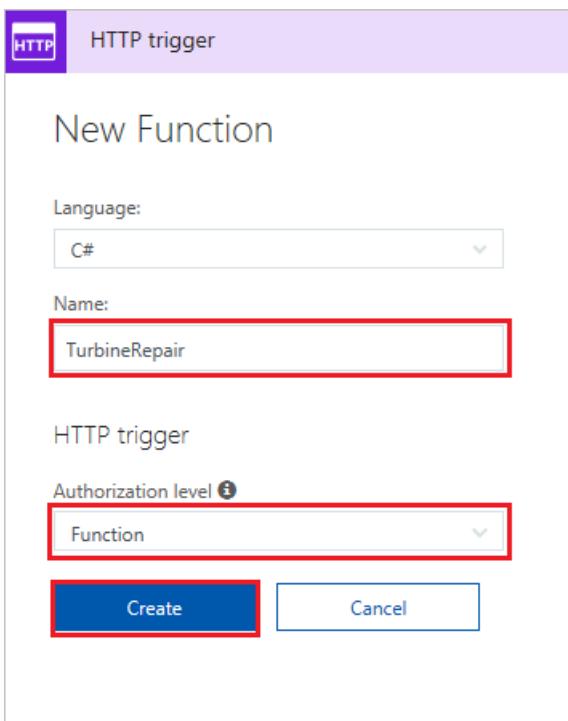
A function that will be run whenever it receives an HTTP request, responding based on data in the body or query string

Batch C# F# JavaScript PowerShell
Python TypeScript

HTTP GET

A function that fetches entities from a Storage table when it

3. Type `TurbineRepair` for the function **Name**, choose `Function` for **Authentication level**, and then select **Create**.



4. Replace the contents of the run.csx file with the following code, then click **Save**:

```
using System.Net;

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

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
{

    //Get request body
    dynamic data = await req.Content.ReadAsAsync<object>();
    int hours = data.hours;
    int capacity = data.capacity;

    //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 req.CreateResponse(HttpStatusCode.OK, 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.

5. 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 developer portal. On the left, there's a code editor with a C# file named 'run.csx'. The 'Run' button is highlighted with a red box. On the right, the 'Test' tab is selected, also highlighted with a red box. The 'Request body' section contains the following JSON:

```

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

The 'Output' section shows the response:

```

{"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 and modify an OpenAPI definition for the function app.

Generate the OpenAPI definition

Now you're ready to generate the OpenAPI definition. This definition can be used by other Microsoft technologies, like API Apps, [PowerApps](#) and [Microsoft Flow](#), as well as third party developer tools like [Postman](#) and [many more packages](#).

1. Select only the *verbs* that your API supports (in this case POST). This makes the generated API definition cleaner.
 - a. On the **Integrate** tab of your new HTTP Trigger function, change **Allowed HTTP methods** to **Selected methods**
 - b. In **Selected HTTP methods**, clear every option except **POST**, then click **Save**.

The screenshot shows the Azure portal interface for managing a function app named "function-demo-energy". On the left, the navigation pane shows "Function Apps" and the selected app "function-demo-energy". Under "Functions", there is an "HttpTriggerCSharp1" function and a "TurbineRepair" function. The "Integrate" section is highlighted with a red box. In the main content area, under "HTTP trigger", the "Allowed HTTP methods" dropdown is set to "All methods". The "Selected methods" checkbox for "POST" is checked, while others like GET, HEAD, OPTIONS, PATCH, TRACE, DELETE, and PUT are unchecked. Other settings like "Mode", "Route template", and "Authorization level" are also visible.

2. Click your function app name (like **function-demo-energy**) > **Platform features** > **API definition**.

The screenshot shows the Azure portal interface with the "Platform features" tab selected for the "function-demo-energy" function app. The left sidebar shows the function app structure. The "API" section in the main content area is highlighted with a red box. It contains options for "API definition" and "CORS". Other sections like "GENERAL SETTINGS", "NETWORKING", "APP SERVICE PLAN", and "RESOURCE MANAGEMENT" are also visible.

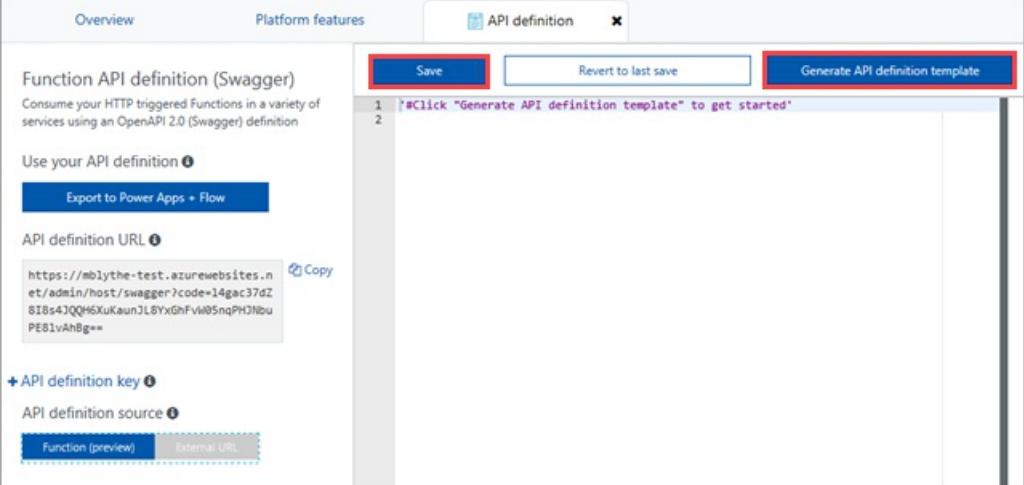
3. On the **API definition** tab, click **Function**.

The screenshot shows the Azure portal interface with the "API definition" tab selected for the "function-demo-energy" function app. The left sidebar shows the function app structure. The "API definition source" dropdown is highlighted with a red box, showing "Function (preview)" selected. Other options like "External URL" are available. Below the dropdown, there are buttons for "Set external definition URL" and "Documentation".

This step enables a suite of OpenAPI options for your function app, including an endpoint to host an OpenAPI file from your function app's domain, an inline copy of the [OpenAPI Editor](#), and an API definition

template generator.

4. Click **Generate API definition template** > **Save**.



The screenshot shows the 'API definition' blade in the Azure portal. At the top, there are tabs for 'Overview', 'Platform features', and 'API definition'. Below the tabs, there's a section for 'Function API definition (Swagger)' with a note: 'Consume your HTTP triggered Functions in a variety of services using an OpenAPI 2.0 (Swagger) definition'. There are buttons for 'Save', 'Revert to last save', and 'Generate API definition template' (the latter is highlighted with a red box). Below these buttons is a code editor containing the generated OpenAPI definition. The code starts with '#Click "Generate API definition template" to get started'. The 'API definition URL' is also displayed, along with an 'API definition key' and 'API definition source' section. The 'External URL' tab is selected in the 'API definition source' dropdown.

Azure scans your function app for HTTP Trigger functions and uses the info in `functions.json` to generate an OpenAPI definition. Here's the definition that is generated:

```
swagger: '2.0'
info:
  title: function-demo-energy.azurewebsites.net
  version: 1.0.0
  host: function-demo-energy.azurewebsites.net
  basePath: /
  schemes:
    - https
    - http
  paths:
    /api/TurbineRepair:
      post:
        operationId: /api/TurbineRepair/post
        produces: []
        consumes: []
        parameters: []
        description: >-
          Replace with Operation Object
          #http://swagger.io/specification/#operationObject
        responses:
          '200':
            description: Success operation
        security:
          - apikeyQuery: []
  definitions: {}
  securityDefinitions:
    apikeyQuery:
      type: apiKey
      name: code
      in: query
```

This definition is described as a *template* because it requires more metadata to be a full OpenAPI definition. You'll modify the definition in the next step.

Modify the OpenAPI definition

Now that you have a template definition, you modify it to provide additional metadata about the API's operations and data structures. In **API definition**, delete the generated definition from `post` to the bottom of the definition, paste in the content below, and click **Save**.

```

post:
  operationId: CalculateCosts
  description: Determines if a technician should be sent for repair
  summary: Calculates costs
  x-ms-summary: Calculates costs
  x-ms-visibility: important
  produces:
    - application/json
  consumes:
    - application/json
  parameters:
    - name: body
      in: body
      description: Hours and capacity used to calculate costs
      x-ms-summary: Hours and capacity
      x-ms-visibility: important
      required: true
      schema:
        type: object
        properties:
          hours:
            description: The amount of effort in hours required to conduct repair
            type: number
            x-ms-summary: Hours
            x-ms-visibility: important
          capacity:
            description: The max output of a turbine in kilowatts
            type: number
            x-ms-summary: Capacity
            x-ms-visibility: important
  responses:
    200:
      description: Message with cost and revenue numbers
      x-ms-summary: Message
      schema:
        type: object
        properties:
          message:
            type: string
            description: Returns Yes or No depending on calculations
            x-ms-summary: Message
          revenueOpportunity:
            type: string
            description: The revenue opportunity cost
            x-ms-summary: RevenueOpportunity
          costToFix:
            type: string
            description: The cost in $ to fix the turbine
            x-ms-summary: CostToFix
          security:
            - apikeyQuery: []
  definitions: {}
  securityDefinitions:
    apikeyQuery:
      type: apiKey
      name: code
      in: query

```

In this case you could just paste in updated metadata, but it's important to understand the types of modifications we made to the default template:

- Specified that the API produces and consumes data in a JSON format.
- Specified the required parameters, with their names and data types.
- Specified the return values for a successful response, with their names and data types.

- Provided friendly summaries and descriptions for the API, and its operations and parameters. This is important for people who will use this function.
- Added x-ms-summary and x-ms-visibility, which are used in the UI for Microsoft Flow and Logic Apps. For more information, see [OpenAPI extensions for custom APIs in Microsoft Flow](#).

NOTE

We left the security definition with the default authentication method of API key. You would change this section if you used a different type of authentication.

For more information about defining API operations, see the [Open API specification](#).

Test the OpenAPI definition

Before you use the API definition, it's a good idea to test it in the Azure Functions UI.

1. On the **Manage** tab of your function, under **Host Keys**, copy the **default** key.

NAME	VALUE	ACTIONS
_master	Click to show	Copy Renew Revoke
default	Click to show	Copy Renew Revoke

NOTE

You use this key for testing, and you also use it when you call the API from an app or service.

2. Go back to the API definition: **function-demo-energy > Platform features > API definition**.
3. In the right pane, click **Authenticate**, enter the API key that you copied, and click **Authenticate**.

4. Scroll down and click **Try this operation**.

The screenshot shows the 'Responses' and 'Security' sections of an API definition. The 'Responses' section includes a table with a single row for status code 200, which describes a message with cost and revenue numbers. The 'Security' section includes a table with a single row for 'apikeyQuery', which is associated with the 'Scopes' column. A red box highlights the 'Try this operation' button at the bottom.

5. Enter values for **hours** and **capacity**.

The screenshot shows the 'Parameters' section of the API definition. It includes a 'body' section with two fields: 'hours' (containing '6') and 'capacity' (containing '2500'). Both fields have red boxes around them, indicating they are selected or highlighted. Descriptions for each field are provided below them.

Notice how the UI uses the descriptions from the API definition.

6. Click **Send Request**, then click the **Pretty** tab to see the output.

The screenshot shows the 'Response' section of the API definition after sending a request. It displays a green bar labeled 'SUCCESS'. Below it are three tabs: 'Rendered' (highlighted with a red box), 'Pretty' (also highlighted with a red box), and 'Raw'. The 'Pretty' tab is currently selected, showing a JSON response with a red box around it. The JSON content is:

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

Next steps

In this tutorial, you learned how to:

- Create a function in Azure
- Generate an OpenAPI definition using OpenAPI tools
- Modify the definition to provide additional metadata
- Test the definition by calling the function

Advance to the next topic to learn how to create a PowerApps app that uses the OpenAPI definition you created.

[Call a function from PowerApps](#)

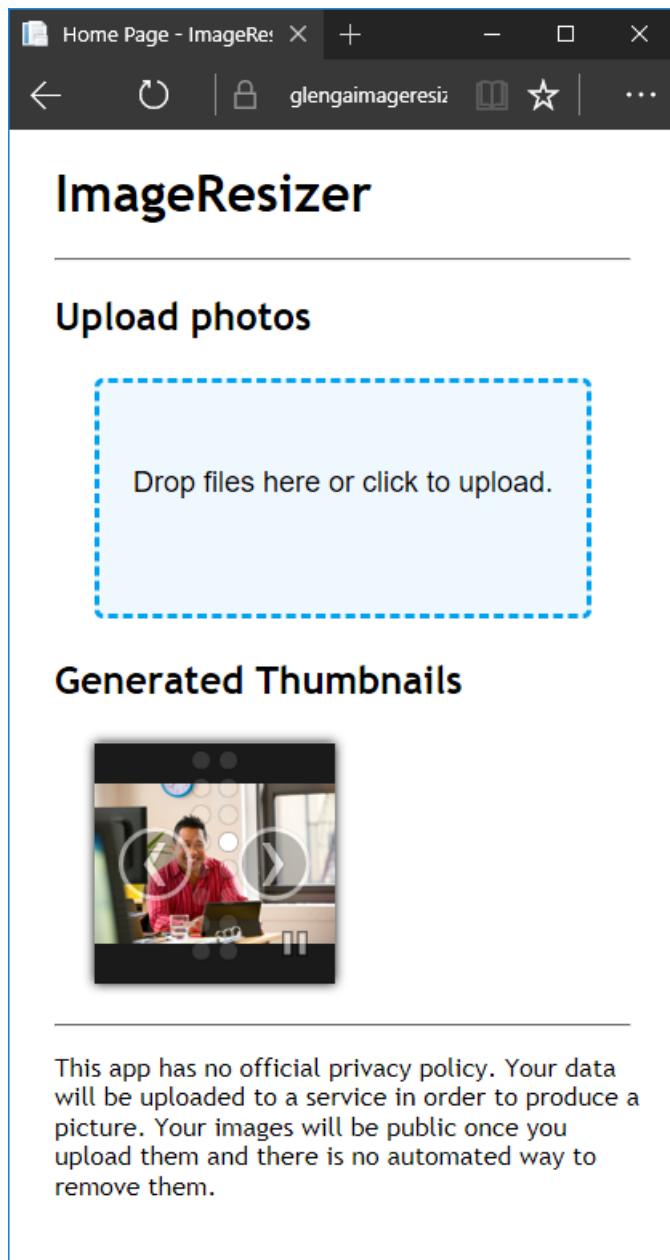
Automate resizing uploaded images using Event Grid

12/4/2017 • 6 min 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.



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

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.

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account. Just click the **Copy** to copy the code, paste it into the Cloud Shell, and then press enter to run it. There are a few ways to launch the Cloud Shell:

Click Try It in the upper right corner of a code block.	
Open Cloud Shell in your browser.	
Click the Cloud Shell button on the menu in the upper right of the Azure portal .	

If you choose to install and use the CLI locally, this topic requires that you are running 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 2.0](#).

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. 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.

In the following command, substitute your own globally unique name for the general storage account where you see the `<general_storage_account>` placeholder.

```
az storage account create --name <general_storage_account> \
--location westcentralus --resource-group myResourceGroup \
--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, substitute your own unique function app name where you see the `<function_app>`

placeholder. The `<function_app>` is used as the default DNS domain for the function app, and so the name needs to be unique across all apps in Azure. In this case, `<general_storage_account>` is the name of the general storage account you created.

```
az functionapp create --name <function_app> --storage-account <general_storage_account> \
--resource-group myResourceGroup --consumption-plan-location westcentralus
```

Now you must configure the function app to connect to blob storage.

Configure the function app

The function needs the connection string to connect to the blob storage account. In this case, `<blob_storage_account>` is the name of the Blob storage account you created in the previous tutorial. Get the connection string with the [az storage account show-connection-string](#) command. The thumbnail image container name must also be set to `thumbs`. 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 <blob_storage_account> \
--query connectionString --output tsv)

az functionapp config appsettings set --name <function_app> \
--resource-group myResourceGroup \
--settings myblobstorage_STORAGE=$storageConnectionString \
myContainerName=thumbs
```

You can now deploy a function code project to this function app.

Deploy the function code

The C# function that performs image resizing is available in this [sample GitHub repository](#). Deploy this Functions code project to the function app by using the [az functionapp deployment source config](#) command.

In the following command, `<function_app>` is the same function app you created in the previous script.

```
az functionapp deployment source config --name <function_app> \
--resource-group myResourceGroup --branch master --manual-integration \
--repo-url https://github.com/Azure-Samples/function-image-upload-resize
```

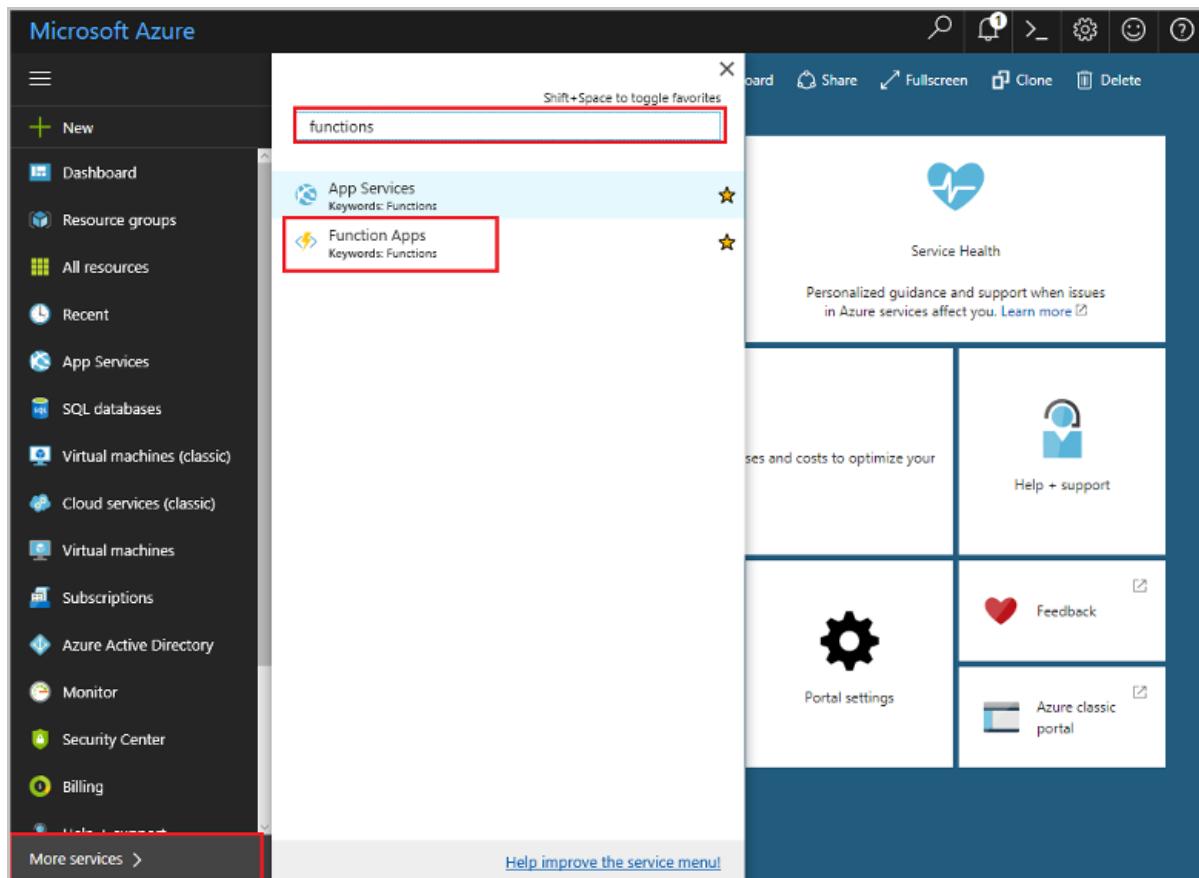
The image resize function is triggered by an event subscription to a Blob created event. The data passed to the trigger includes the URL of the blob, which is in turn passed to the input binding to obtain the uploaded image from Blob storage. The function generates a thumbnail image and writes the resulting stream to a separate container in Blob storage. To learn more about this function, see the [readme file in the sample repository](#).

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](#).

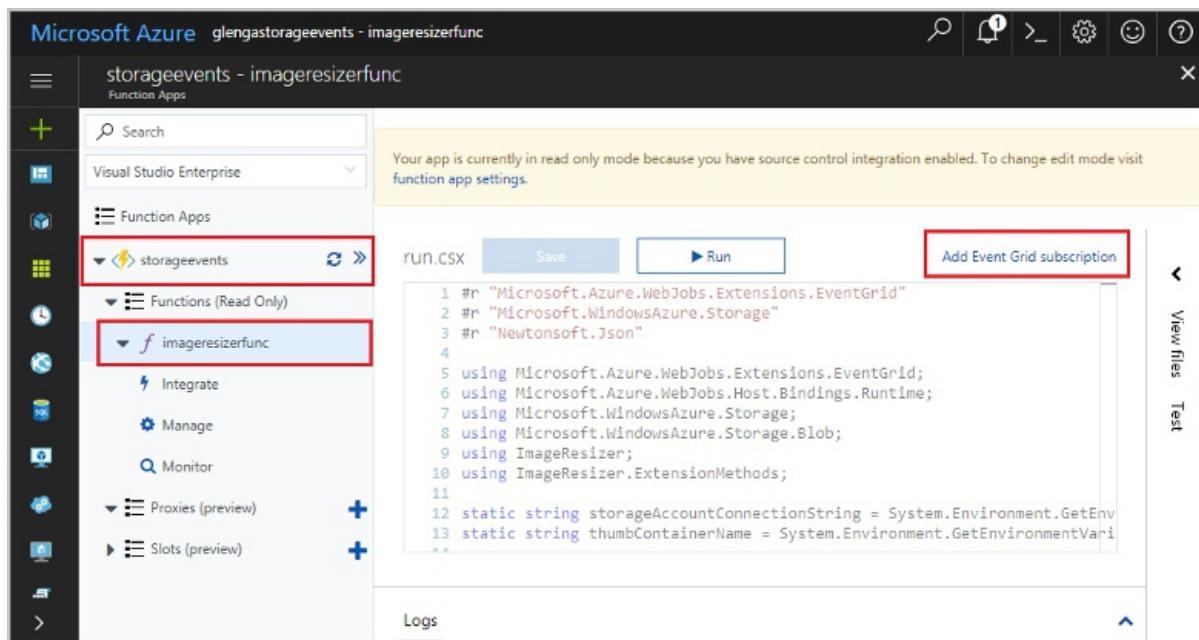
Create your event subscription

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 from your function in the Azure portal:

1. In the [Azure portal](#), click the arrow at the bottom left to expand all services, type `functions` in the **Filter** field, and then choose **Function Apps**.



2. Expand your function app, choose the **imageresizerfunc** function, and then select **Add Event Grid subscription**.



3. Use the event subscription settings as specified in the table.

Create Event Subscription

Event Grid - PREVIEW

* Name
imageresizersub

Topic Type
Storage Accounts

Subscription
Visual Studio Enterprise

Resource group
 Use existing
myResourceGroup

Instance ⓘ
blobstorage

* Event Types
Blob Created

* Subscriber Endpoint
https://storageevents.azurewebsites.net/admin/extensions/EventGridExtensionConfig?func

Prefix Filter
/blobServices/default/containers/images/blobs/

Suffix Filter
.jpg

Filter Case Sensitive

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 subscription	By default, your current subscription should be selected.
Resource group	myResourceGroup	Select Use existing and choose the resource group you have been using in this topic.
Instance	<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 Microsoft.Storage.BlobCreated are passed to the function.

SETTING	SUGGESTED VALUE	DESCRIPTION
Subscriber endpoint	autogenerated	Use the endpoint URL that is generated for you.
Prefix filter	/blobServices/default/containers/images/blobs/	Filters storage events to only those on the images container.

- Click **Create** to add the event subscription. This creates an event subscription that triggers **imageresizerfunc** when a blob is added to the **images** container. Resized images are added to the **thumbs** 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.

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 thumbs container, and downloaded by the web client.

The screenshot shows a web browser window titled "Home Page - ImageResizer". The page has a header with the title "ImageResizer". Below the header, there is a section titled "Upload photos" with a dashed blue border. Inside this section, the text "Drop files here or click to upload." is displayed. Below this, there is a section titled "Generated Thumbnails" containing a thumbnail image of a person sitting at a desk with a laptop. At the bottom of the page, 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.

Next steps

In this tutorial, you learned how to:

- Create a general Azure Storage account
- Deploy serverless code using Azure Functions
- Create a Blob storage event subscription in Event Grid

Advance to part three of the Storage tutorial series to learn how to secure access to the storage account.

[Secure access to an applications data in the cloud](#)

- To learn more about Event Grid, see [An introduction to Azure Event Grid](#).
- To try another tutorial that features Azure Functions, see [Create a function that integrates with Azure Logic Apps](#).

Create a function on Linux using a custom image (preview)

12/4/2017 • 9 min to read • [Edit Online](#)

Azure Functions lets you host your functions on Linux in your own custom container. This functionality is currently in preview. You can also [host on a default Azure App Service container](#).

In this tutorial, you learn how to deploy a function app as a custom Docker image. This pattern is useful when you need to customize the built-in App Service 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.

This tutorial walks you through how to use Azure Functions to create and push a custom image to Docker Hub. You then use this image as the deployment source for a function app that runs on Linux. You use Docker to build and push the image. You use the Azure CLI to create a function app and deploy the image from Docker Hub.

In this tutorial, you learn how to:

- Build a custom image using Docker.
- Publish a custom image to a container registry.
- Create an Azure Storage account.
- Create a Linux App Service plan.
- Deploy a function app from Docker Hub.
- Add application settings to the function app.

The following steps are supported on a Mac, Windows, or Linux computer.

Prerequisites

To complete this tutorial, you need:

- [Git](#)
- An active [Azure subscription](#)
- [Docker](#)
- A [Docker Hub account](#)

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

Download the sample

In a terminal window, run the following command to clone the sample app repository to your local machine, then change to the directory that contains the sample code.

```
git clone https://github.com/Azure-Samples/functions-linux-custom-image.git --config core.autocrlf=input  
cd functions-linux-custom-image
```

Build the image from the Docker file

In this Git repository, take a look at the *Dockerfile*. This file describes the environment that is required to run the

function app on Linux.

```
# Base the image on the built-in Azure Functions Linux image.  
FROM microsoft/azure-functions-runtime:2.0.0-jessie  
ENV AzureWebJobsScriptRoot=/home/site/wwwroot  
  
# Add files from this repo to the root site folder.  
COPY . /home/site/wwwroot
```

NOTE

When hosting an image in a private container registry, you should add the connection settings to the function app by using **ENV** variables in the Dockerfile. Because this tutorial cannot guarantee that you use a private registry, the connection settings are [added after the deployment by using the Azure CLI](#) as a security best practice.

Run the Build command

To build the Docker image, run the `docker build` command, and provide a name, `mydockerimage`, and tag, `v1.0.0`. Replace `<docker-id>` with your Docker Hub account ID.

```
docker build --tag <docker-id>/mydockerimage:v1.0.0 .
```

The command produces output similar to the following:

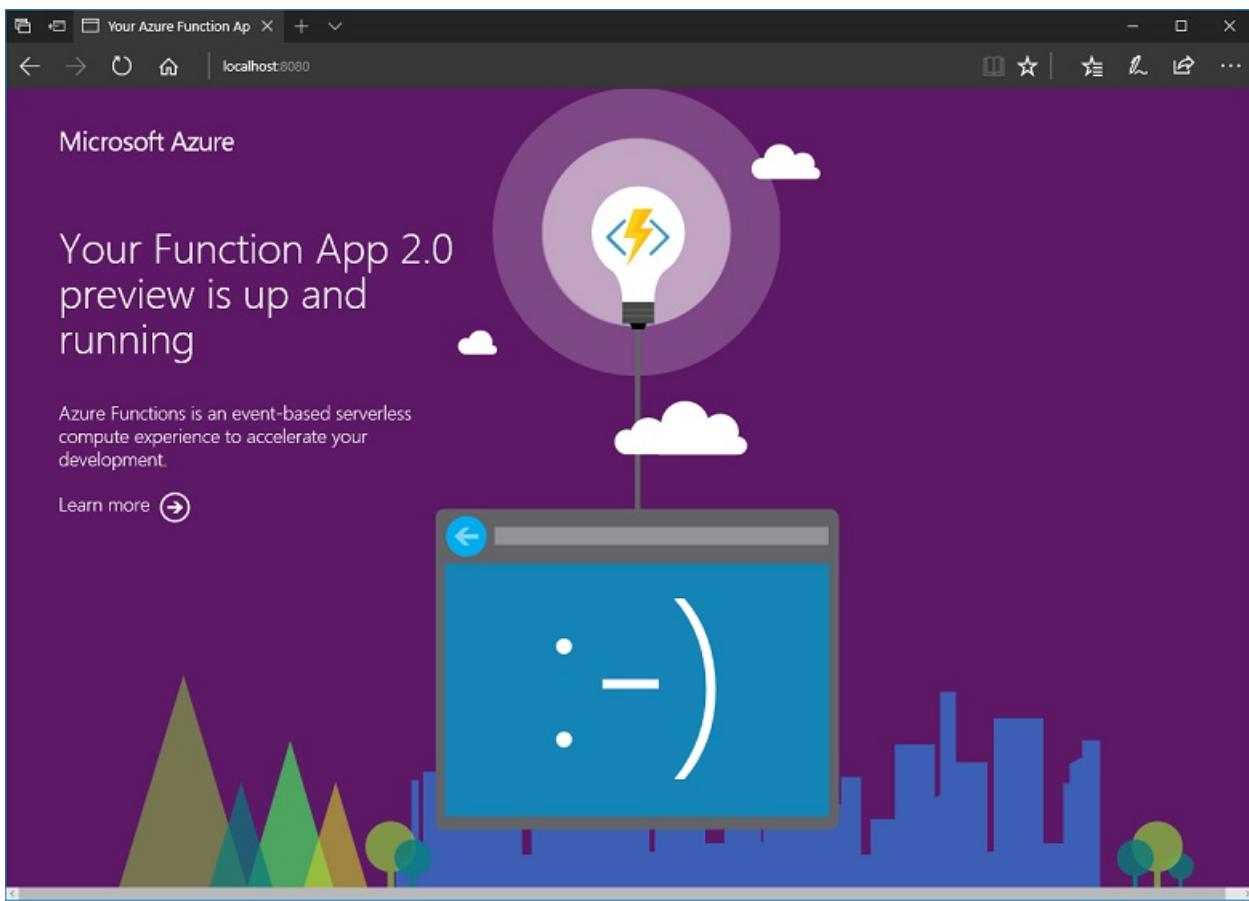
```
Sending build context to Docker daemon 169.5kB  
Step 1/3 : FROM microsoft/azure-functions-runtime:v2.0.0-jessie  
v2.0.0-jessie: Pulling from microsoft/azure-functions-runtime  
b178b12f7913: Pull complete  
2d9ce077a781: Pull complete  
4775d4ba55c8: Pull complete  
Digest: sha256:073f45fc167b3b5c6642ef4b3c99064430d6b17507095...  
Status: Downloaded newer image for microsoft/azure-functions-runtime:v2.0.0-jessie  
---> 217799efa500  
Step 2/3 : ENV AzureWebJobsScriptRoot /home/site/wwwroot  
---> Running in 528fa2077d17  
---> 7cc6323b8ae0  
Removing intermediate container 528fa2077d17  
Step 3/3 : COPY . /home/site/wwwroot  
---> 5bdac9878423  
Successfully built 5bdac9878423  
Successfully tagged ggailey777/mydockerimage:v1.0.0
```

Test the image locally

Verify that the built image 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>.



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. In order 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 .
```

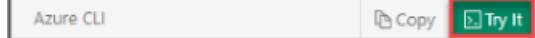
Verify that the push succeeded by examining the command's output.

```
The push refers to a repository [docker.io/<docker-id>/mydockerimage:v1.0.0]
24d81eb139bf: Pushed
fd9e998161c9: Mounted from microsoft/azure-functions-runtime
e7796c35add2: Mounted from microsoft/azure-functions-runtime
ae9a05b85848: Mounted from microsoft/azure-functions-runtime
45c86e20670d: Mounted from microsoft/azure-functions-runtime
v1.0.0: digest: sha256:be080d80770df71234eb893fbe4d... size: 2422
```

Now, you can use this image as the deployment source for a new function app in Azure.

Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account. Just click the **Copy** to copy the code, paste it into the Cloud Shell, and then press enter to run it. There are a few ways to launch the Cloud Shell:

Click Try It in the upper right corner of a code block.	
Open Cloud Shell in your browser.	
Click the Cloud Shell button on the menu in the upper right of the Azure portal .	

If you choose to install and use the CLI locally, this topic requires the Azure CLI version 2.0.21 or later. Run `az --version` to find the version you have. If you need to install or upgrade, see [Install Azure CLI 2.0](#).

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. To see all supported locations for App Service plans, run the [az appservice list-locations](#) command.

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
```

After the storage account has been created, the Azure CLI shows information similar to the following example:

```
{
  "creationTime": "2017-04-15T17:14:39.320307+00:00",
  "id": "/subscriptions/bbbef702-e769-477b-9f16-bc4d3aa97387/resourceGroups/myresourcegroup/...",
  "kind": "Storage",
  "location": "westeurope",
  "name": "myfunctionappstorage",
  "primaryEndpoints": {
    "blob": "https://myfunctionappstorage.blob.core.windows.net/",
    "file": "https://myfunctionappstorage.file.core.windows.net/",
    "queue": "https://myfunctionappstorage.queue.core.windows.net/",
    "table": "https://myfunctionappstorage.table.core.windows.net/"
  },
  ...
  // Remaining output has been truncated for readability.
}
```

Create a Linux App Service plan

Linux hosting for Functions is currently not supported on consumption plans. You must run on a Linux App Service plan. To learn more about hosting, see [Azure Functions hosting plans comparison](#).

In the Cloud Shell, create an App Service plan in the resource group with the `az appservice plan create` command.

The following example creates an App Service plan named `myAppServicePlan` in the **Standard** pricing tier (`--sku S1`) and in a Linux container (`--is-linux`).

```
az appservice plan create --name myAppServicePlan --resource-group myResourceGroup --sku S1 --is-linux
```

When the App Service plan has been created, the Azure CLI shows information similar to the following example:

```
{
  "adminSiteName": null,
  "appServicePlanName": "myAppServicePlan",
  "geoRegion": "West Europe",
  "hostingEnvironmentProfile": null,
  "id": "/subscriptions/0000-
0000/resourceGroups/myResourceGroup/providers/Microsoft.Web/serverfarms/myAppServicePlan",
  "kind": "linux",
  "location": "West Europe",
  "maximumNumberOfWorkers": 1,
  "name": "myAppServicePlan",
  < JSON data removed for brevity. >
  "targetWorkerSizeId": 0,
  "type": "Microsoft.Web/serverfarms",
  "workerTierName": null
}
```

Create and deploy the custom image

The function app hosts the execution of your functions. 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 myAppServicePlan --deployment-container-image-name <docker-id>/mydockerimage:v1.0.0
```

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.
}
```

The *deployment-container-image-name* parameter indicates the image hosted on Docker Hub to use to create the function app.

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 equivalent.

In this case, `<storage_account>` 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_account> \
--query connectionString --output tsv)

az functionapp config appsettings set --name <function_app> \
--resource-group myResourceGroup \
--settings AzureWebJobsDashboard=$storageConnectionString \
AzureWebJobsStorage=$storageConnectionString
```

You can now test your functions running on Linux in Azure.

Test the function

Use cURL to test the deployed function on a Mac or Linux computer or using Bash on Windows. Execute the following cURL command, replacing the `<app_name>` placeholder with the name of your function app. Append the query string `&name=<yourname>` to the URL.

```
curl http://<app_name>.azurewebsites.net/api/HttpTriggerJS1?name=<yourname>
```

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

If you don't have cURL available in your command line, enter the same URL in the address of your web browser.

Again, replace the `<app_name>` placeholder with the name of your function app, and append the query string

`&name=<yourusername>` to the URL and execute the request.

```
http://<app_name>.azurewebsites.net/api/HttpTriggerJS1?name=<yourusername>
```



Clean up resources

Other quickstarts in this collection build upon this quickstart. If you plan to continue on to work with subsequent quickstarts or with the tutorials, do not clean up the resources created in this quickstart. If you do not plan to continue, use the following command to delete all resources created by this quickstart:

```
az group delete --name myResourceGroup
```

Type `y` when prompted.

Next steps

In this tutorial, you learned how to:

- Build a custom image using Docker.
- Publish a custom image to a container registry.
- Create an Azure Storage account.
- Create a Linux App Service plan.
- Deploy a function app from Docker Hub.
- Add application settings to the function app.

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

[Code and test Azure Functions locally](#)

Deploy Azure Function as an IoT Edge module - preview

12/8/2017 • 6 min to read • [Edit Online](#)

You can use Azure Functions to deploy code that implements your business logic directly to your IoT Edge devices. This tutorial walks you through creating and deploying an Azure Function that filters sensor data on the simulated IoT Edge device that you created in the Deploy Azure IoT Edge on a simulated device on [Windows](#) or [Linux](#) tutorials. 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 your registry
- Deploy the module to your IoT Edge device
- View generated data

The Azure Function that you create in this tutorial filters the temperature data generated by your device and only sends messages upstream to Azure IoT Hub when the temperature is above a specified threshold.

Prerequisites

- The Azure IoT Edge device that you created in the quickstart or previous tutorial.
- [Visual Studio Code](#).
- [C# for Visual Studio Code \(powered by OmniSharp\) extension](#).
- [Azure IoT Edge extension for Visual Studio Code](#).
- [Docker](#). The Community Edition (CE) for your platform is sufficient for this tutorial.
- [.NET Core 2.0 SDK](#).

Create a container registry

In this tutorial, you use the Azure IoT Edge extension for VS Code to build a module and create a **container image** from the files. Then you push this image to a **registry** that stores and manages your images. Finally, you deploy your image from your registry to run on your IoT Edge device.

You can use any Docker-compatible registry for this tutorial. Two popular Docker registry services available in the cloud are [Azure Container Registry](#) and [Docker Hub](#). This tutorial uses Azure Container Registry.

1. In the [Azure portal](#), select **Create a resource > Containers > Azure Container Registry**.
2. Give your registry a name, choose a subscription, choose a resource group, and set the SKU to **Basic**.
3. Select **Create**.
4. Once your container registry is created, navigate to it and select **Access keys**.
5. Toggle **Admin user** to **Enable**.
6. Copy the values for **Login server**, **Username**, and **Password**. You'll use these values later in the tutorial.

Create a function project

The following steps show you how to create an IoT Edge function using Visual Studio Code and the Azure IoT Edge extension.

1. Open Visual Studio Code.
2. To open the VS Code integrated terminal, select **View > Integrated Terminal**.

3. To install (or update) the **AzureIoTEdgeFunction** template in dotnet, run the following command in the integrated terminal:

```
dotnet new -i Microsoft.Azure.IoT.Edge.Function
```

4. Create a project for the new module. The following command creates the project folder, **FilterFunction**, in the current working folder:

```
dotnet new aziotedgefunction -n FilterFunction
```

5. Select **File > Open Folder**, then browse to the **FilterFunction** folder and open the project in VS Code.

6. In VS Code explorer, expand the **EdgeHubTrigger-Csharp** folder, then open the **run.csx** file.

7. Replace the contents of the file with the following code:

```

#r "Microsoft.Azure.Devices.Client"
#r "Newtonsoft.Json"

using System.IO;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;

// Filter messages based on the temperature value in the body of the message and the temperature
threshold value.
public static async Task Run(Message messageReceived, IAsyncCollector<Message> output, TraceWriter log)
{
    const int temperatureThreshold = 25;
    byte[] messageBytes = messageReceived.GetBytes();
    var messageString = System.Text.Encoding.UTF8.GetString(messageBytes);

    if (!string.IsNullOrEmpty(messageString))
    {
        // 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);
            }
            // 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);
            log.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;}
}

```

8. Save the file.

Publish a Docker image

1. Build the Docker image.
 - a. In VS Code explorer, expand the **Docker** folder. Then expand the folder for your container platform, either **linux-x64** or **windows-nano**.
 - b. Right-click the **Dockerfile** file and click **Build IoT Edge module Docker image**.

- c. Navigate to the **FilterFunction** project folder and click **Select Folder as EXE_DIR**.
- d. In the pop-up text box at the top of the VS Code window, enter the image name. For example:
`<your container registry address>/filterfunction:latest`. The container registry address is the same as the login server that you copied from your registry. It should be in the form of
`<your container registry name>.azurecr.io`.

2. Sign in to Docker. In the integrated terminal, enter the following command:

```
docker login -u <username> -p <password> <Login server>
```

To find the user name, password and login server to use in this command, go to the [Azure portal](#). From **All resources**, click the tile for your Azure container registry to open its properties, then click **Access keys**. Copy the values in the **Username**, **password**, and **Login server** fields.

3. Push the image to your Docker repository. Select **View > Command Palette...** then search for **Edge: Push IoT Edge module Docker image**.
4. In the pop-up text box, enter the same image name that you used in step 1.d.

Add registry credentials to your Edge device

Add the credentials for your registry to the Edge runtime on the computer where you are running your Edge device. This gives the runtime access to pull the container.

- For Windows, run the following command:

```
iotedgedctl login --address <your container registry address> --username <username> --password <password>
```

- For Linux, run the following command:

```
sudo iotedgedctl login --address <your container registry address> --username <username> --password <password>
```

Run the solution

1. In the **Azure portal**, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. If you've already deployed the **tempSensor** module to this device, it may be automatically populated. If not, follow these steps to add it:
 - a. Select **Add IoT Edge Module**.
 - b. In the **Name** field, enter `tempSensor`.
 - c. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
 - d. Leave the other settings unchanged and click **Save**.
5. Add the **filterFunction** module.
 - a. Select **Add IoT Edge Module** again.
 - b. In the **Name** field, enter `filterFunction`.
 - c. In the **Image** field, enter your image address; for example
`<docker registry address>/filterfunction:latest`.
 - d. Click **Save**.

6. Click **Next**.
7. In the **Specify Routes** step, copy the JSON below into the text box. The first route transports messages from the temperature sensor to the filter module via the "input1" endpoint. The second route transports messages from the filter module to IoT Hub. In this route, `$upstream` is a special destination that tells Edge Hub to send messages to IoT Hub.

```
{  
  "routes":{  
    "sensorToFilter":"FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO  
BrokeredEndpoint(\"/modules/filterFunction/inputs/input1\"),  
    "filterToIoTHub":"FROM /messages/modules/filterFunction/outputs/* INTO $upstream"  
  }  
}
```

8. Click **Next**.
9. In the **Review Template** step, click **Submit**.
10. Return to the IoT Edge device details page and click **Refresh**. You should see the new **filterfunction** module running along with the **tempSensor** module and the **IoT Edge runtime**.

View generated data

To monitor device to cloud messages sent from your IoT Edge device to your IoT hub:

1. Configure the Azure IoT Toolkit extension with connection string for your IoT hub:
 - a. In the Azure portal, navigate to your IoT hub and select **Shared access policies**.
 - b. Select **iothubowner** then copy the value of **Connection string-primary key**.
 - c. In the VS Code explorer, click **IOT HUB DEVICES** and then click
 - d. Select **Set IoT Hub Connection String** and enter the IoT Hub connection string in the pop-up window.
2. To monitor data arriving at the IoT hub, select the **View > Command Palette...** and search for **IoT: Start monitoring D2C message**.
3. To stop monitoring data, use the **IoT: Stop monitoring D2C message** command in the Command Palette.

Next steps

In this tutorial, you created an Azure Function that contains code to filter raw data generated by your IoT Edge device. To keep exploring Azure IoT Edge, learn how to use an IoT Edge device as a gateway.

[Create an IoT Edge gateway device](#)

Azure CLI Samples

1/9/2018 • 1 min to read • [Edit Online](#)

The following table includes links to bash scripts for Azure Functions that use the Azure CLI.

Create app	
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	
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	
Deploy from GitHub	Create a function app that deploys from a GitHub repository.
Deploy from VSTS	Create a function app that deploys from a Visual Studio Team Services (VSTS) repository.
Configure app	
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 scale and hosting

1/9/2018 • 7 min to read • [Edit Online](#)

You can run Azure Functions in two different modes: Consumption plan and Azure App Service 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 don't have to reserve capacity in advance. This article focuses on the Consumption plan, a [serverless](#) app model. For details about how the App Service plan works, see the [Azure App Service plans in-depth overview](#).

NOTE

Linux hosting is currently only available on an App Service plan.

If you aren't familiar with Azure Functions, see the [Azure Functions overview](#).

When you create a function app, you must configure a hosting plan for functions that the app contains. In either mode, an instance of the *Azure Functions host* executes the functions. The type of plan controls:

- How host instances are scaled out.
- The resources that are available to each host.

Currently, you must choose the type of hosting plan during the creation of the function app. You can't change it afterward.

On an App Service plan you can scale between tiers to allocate different amount of resources. On the Consumption plan, Azure Functions automatically handles all resource allocation.

Consumption plan

When you're using a Consumption plan, instances of the Azure Functions host are dynamically added and removed based on the number of incoming events. This plan scales automatically, and you are charged for compute resources only when your functions are running. On a Consumption plan, a function can run for a maximum of 10 minutes.

NOTE

The default timeout for functions on a Consumption plan is 5 minutes. The value can be increased to 10 minutes for the Function App by changing the property `functionTimeout` in the [host.json](#) project file.

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.

App Service plan

In the App Service plan, your function apps run on dedicated VMs on Basic, Standard, Premium, and Isolated SKUs, similar to Web Apps, API Apps, and Mobile Apps. Dedicated VMs are allocated to your App Service apps,

which means the functions host is always running. App Service plans support Linux.

Consider an App Service plan in the following cases:

- You have existing, underutilized VMs that are already running other App Service instances.
- You expect your function apps to run continuously, or nearly continuously. In this case, an App Service Plan can be more cost-effective.
- You need more CPU or memory options than what is provided on the Consumption plan.
- You need to run longer than the maximum execution time allowed on the Consumption plan (of 10 minutes).
- You require features that are only available on an App Service plan, such as support for App Service Environment, VNET/VPN connectivity, and larger VM sizes.
- You want to run your function app on Linux, or you want to provide a custom image on which to run your functions.

A VM decouples cost from number of executions, execution time, and memory used. As a result, you won't pay more than the cost of the VM instance that you allocate. 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, or you can 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](#).

If you are planning to run JavaScript functions on an App Service plan, you should choose a plan that has fewer vCPUs. For more information, see the [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 will go idle after a few minutes of inactivity, so only HTTP triggers will "wake up" your functions. This is similar to how WebJobs must have Always On enabled.

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

Storage account requirements

On either a Consumption plan or an App Service plan, a function app requires a general Azure Storage account that supports Azure Blob, Queue, Files, and Table storage. Internally, Azure Functions uses Azure Storage for operations such as managing triggers and logging function executions. Some storage accounts do not support queues and tables, such as blob-only storage accounts (including premium storage) and general-purpose storage accounts with zone-redundant storage replication. These accounts are filtered from the **Storage Account** blade when you're creating a function app.

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

How the Consumption plan works

In the Consumption plan, the scale controller automatically 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 is limited to 1.5 GB of memory. An instance of the host is the Function App, meaning all functions within a function app share resources within an instance and scale at the same time.

When you use the Consumption hosting plan, 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

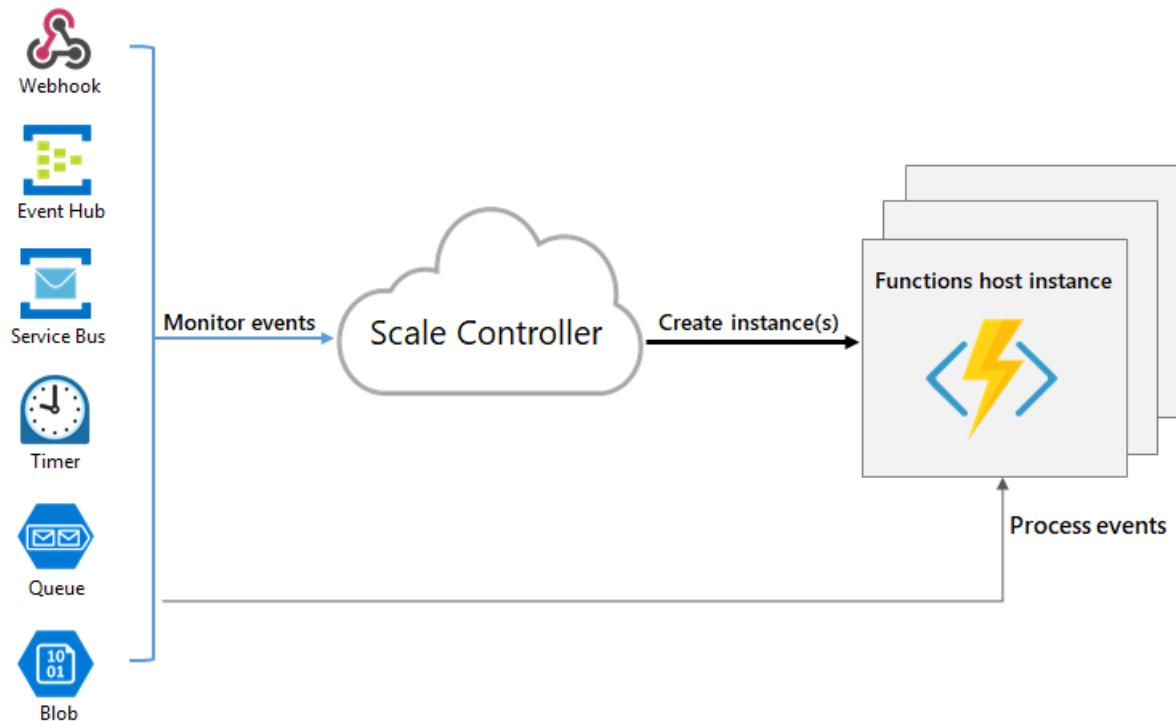
When you're using a blob trigger on a Consumption plan, there can be up to a 10-minute delay in processing new blobs if a function app has gone idle. After the function app is running, blobs are processed immediately. To avoid this initial delay, consider one of the following options:

- Host the function app on an App Service plan, with Always On enabled.
- Use another mechanism to trigger the blob processing, such as an Event Grid subscription or a queue message that contains the blob name. For an example, see the [examples for the blob input binding](#).

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 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. However there are a few aspects of scaling that exist in the system today:

- A single function app will only scale 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.
- New instances will only be allocated at most once every 10 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. View the [scalability section of the performance considerations article](#) for more information.

Billing model

Billing for the Consumption plan 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.

Azure Functions triggers and bindings concepts

1/10/2018 • 11 min to read • [Edit Online](#)

This article is a conceptual overview of triggers and bindings in Azure Functions. Features that are common to all bindings and all supported languages are described here.

Overview

A *trigger* defines how a function is invoked. A function must have exactly one trigger. Triggers have associated data, which is usually the payload that triggered the function.

Input and output *bindings* provide a declarative way to connect to data from within your code. Bindings are optional and a function can have multiple input and output bindings.

Triggers and bindings let you avoid hardcoding the details of the services that you're working with. You 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, an `out` parameter, or a [collector object](#).

When you develop functions by using the Azure portal, triggers and bindings are configured in a `function.json` file. The portal provides a UI for this configuration but you can edit the file directly by changing to the **Advanced editor**.

When you develop functions by using Visual Studio to create a class library, you configure triggers and bindings by decorating methods and parameters with attributes.

Supported bindings

The following table shows the bindings that are supported in the two major versions of the Azure Functions runtime.

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
Blob Storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓ ¹	✓	✓	✓
Event Hubs	✓	✓	✓		✓
External File ²	✓			✓	✓
External Table ²	✓			✓	✓
HTTP	✓	✓	✓		✓
Microsoft Graph Excel tables		✓ ¹		✓	✓
Microsoft Graph OneDrive files		✓ ¹		✓	✓

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
Microsoft Graph Outlook email		✓ ¹			✓
Microsoft Graph Events		✓ ¹	✓	✓	✓
Microsoft Graph Auth tokens		✓ ¹		✓	
Mobile Apps	✓	✓ ¹		✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓ ¹			✓
Service Bus	✓	✓ ¹	✓		✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓ ¹			✓
Webhooks	✓		✓		✓

¹ Must be registered as a binding extension in 2.x. See [Known issues in 2.x](#).

² Experimental — not supported and might be abandoned in the future.

For information about which bindings are in preview or are approved for production use, see [Supported languages](#).

Example: queue trigger and table output binding

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": [
    {
      "name": "order",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "myqueue-items",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "name": "$return",
      "type": "table",
      "direction": "out",
      "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 will receive 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 will provide 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.

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 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, TraceWriter 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; }
}

```

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);
}

```

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:

```

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,
        TraceWriter 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; }
}

```

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 in the **Integrate** tab.

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.

Using the function return type to return a single output

The preceding example shows how to use the function return value to provide output to a binding, which is specified in `function.json` by using the special value `$return` for the `name` property. (This is only supported in languages that have a return value, such as C# script, JavaScript, and F#.) If a function has multiple output bindings, use `$return` for only one of the output bindings.

```

// excerpt of function.json
{
    "name": "$return",
    "type": "blob",
    "direction": "out",
    "path": "output-container/{id}"
}

```

The examples below show how return types are used with output bindings in C# script, JavaScript, and F#.

```
// C# example: use method return value for output binding
public static string Run(WorkItem input, TraceWriter log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.Info($"C# script processed queue message. Item={json}");
    return json;
}
```

```
// C# example: async method, using return value for output binding
public static Task<string> Run(WorkItem input, TraceWriter log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.Info($"C# script processed queue message. Item={json}");
    return Task.FromResult(json);
}
```

```
// JavaScript: return a value in the second parameter to 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);
}
```

```
// F# example: use return value for output binding
let Run(input: WorkItem, log: TraceWriter) =
    let json = String.Format("{{ \"id\": \"{0}\" }}", input.Id)
    log.Info(sprintf "F# script processed queue message '%s'" json)
    json
```

Binding dataType property

In .NET, use the parameter type to define 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 deserialize to a POCO 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`:

```
{
    "type": "httpTrigger",
    "name": "req",
    "direction": "in",
    "dataType": "binary"
}
```

Other options for `dataType` are `stream` and `string`.

Resolving 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 `function.json` in a public source control repository.

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 settings are resolved whenever a value is enclosed in percent signs, such as `%MyAppSetting%`. Note that the `connection` property of triggers and bindings is a special case and automatically resolves values as app settings.

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,  
    TraceWriter log)  
{  
    log.Info($"C# Queue trigger function processed: {myQueueItem}");  
}
```

Trigger metadata properties

In addition to the data payload provided by a trigger (such as 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.

Binding expressions and patterns

One of the most powerful features of triggers and bindings is *binding expressions*. In the configuration for a binding, you can define pattern expressions which can then be used in other bindings or your code. Trigger metadata can also be used in binding expressions, as shown in the preceding section.

For example, suppose you want to resize images in a particular blob storage container, similar to the **Image Resizer** template in the **New Function** page of the Azure portal (see the **Samples** scenario).

Here is the *function.json* definition:

```

{
  "bindings": [
    {
      "name": "image",
      "type": "blobTrigger",
      "path": "sample-images/{filename}",
      "direction": "in",
      "connection": "MyStorageConnection"
    },
    {
      "name": "imageSmall",
      "type": "blob",
      "path": "sample-images-sm/{filename}",
      "direction": "out",
      "connection": "MyStorageConnection"
    }
  ],
}

```

Notice that the `filename` parameter is used in both the blob trigger definition and the blob output binding. This parameter can also be used in function code.

```

// C# example of binding to {filename}
public static void Run(Stream image, string filename, Stream imageSmall, TraceWriter log)
{
  log.Info($"Blob trigger processing: {filename}");
  // ...
}

```

The same ability to use binding expressions and patterns applies to attributes in class libraries. For example, here is a image resizing function in a class library:

```
[FunctionName("ResizeImage")]
[StorageAccount("AzureWebJobsStorage")]
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)
{
    var imageBuilder = ImageResizer.ImageBuilder.Current;
    var size = imageDimensionsTable[ImageSize.Small];

    imageBuilder.Build(image, imageSmall,
        new ResizeSettings(size.Item1, size.Item2, FitMode.Max, null), false);

    image.Position = 0;
    size = imageDimensionsTable[ImageSize.Medium];

    imageBuilder.Build(image, imageMedium,
        new ResizeSettings(size.Item1, size.Item2, FitMode.Max, null), false);
}

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) }
};
```

Create GUIDs

The `{rand-guid}` binding expression creates a GUID. The following example uses a GUID to create a unique blob name:

```
{
    "type": "blob",
    "name": "blobOutput",
    "direction": "out",
    "path": "my-output-container/{rand-guid}"
}
```

Current time

The binding expression `DateTime` resolves to `DateTime.UtcNow`.

```
{
    "type": "blob",
    "name": "blobOutput",
    "direction": "out",
    "path": "my-output-container/{DateTime}"
}
```

Bind to custom input properties

Binding expressions can also reference properties that are defined in the trigger payload itself. For example, you may want to dynamically bind to a blob storage file from a filename provided in a webhook.

For example, the following `function.json` uses a property called `BlobName` from the trigger payload:

```
{
  "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"
    }
  ]
}
```

To accomplish this in C# and F#, you must define a POCO that defines the fields that will be deserialized in the trigger payload.

```
using System.Net;

public class BlobInfo
{
    public string BlobName { get; set; }
}

public static HttpResponseMessage Run(HttpRequestMessage req, BlobInfo info, string blobContents)
{
    if (blobContents == null) {
        return req.CreateResponse(HttpStatusCode.NotFound);
    }

    return req.CreateResponse(HttpStatusCode.OK, new {
        data = $"{blobContents}"
    });
}
```

In JavaScript, JSON deserialization is automatically performed and you can use the properties directly.

```
module.exports = function (context, info) {
    if ('BlobName' in info) {
        context.res = {
            body: { 'data': context.bindings.blobContents }
        }
    } else {
        context.res = {
            status: 404
        };
    }
    context.done();
}
```

Configuring binding data 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 [Binding at runtime via imperative bindings](#) in the C# developer reference.

function.json file schema

The *function.json* file schema is available at <http://json.schemastore.org/function>.

Next steps

For more information on a specific binding, see the following articles:

- [HTTP and webhooks](#)
- [Timer](#)
- [Queue storage](#)
- [Blob storage](#)
- [Table storage](#)
- [Event Hub](#)
- [Service Bus](#)
- [Azure Cosmos DB](#)
- [Microsoft Graph](#)
- [SendGrid](#)
- [Twilio](#)
- [Notification Hubs](#)
- [Mobile Apps](#)
- [External file](#)

Supported languages in Azure Functions

11/16/2017 • 1 min 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 1.x runtime is GA. It's the only runtime that is approved for production applications. The 2.x runtime is currently in preview, so the languages it supports are in preview. The following table shows which languages are supported in each runtime version.

LANGUAGE	1.X	2.X
C#	GA	Preview
JavaScript	GA	Preview
F#	GA	
Java		Preview
Python	Experimental	
PHP	Experimental	
TypeScript	Experimental	
Batch (.cmd, .bat)	Experimental	
Bash	Experimental	
PowerShell	Experimental	

For information about planned changes to language support, see [Azure roadmap](#).

Experimental languages

The experimental languages in 1.x don't scale well and don't support all bindings. For example, Python is slow because the Functions runtime runs `python.exe` with each function invocation. And while Python supports HTTP bindings, it can't access the request object.

Experimental support for PowerShell is limited to version 4.0 because that is what's installed on the VMs that

Function apps run on. If you want to run PowerShell scripts, consider [Azure Automation](#).

The 2.x runtime doesn't support experimental languages. In 2.x, we will add support for a language only when it scales well and supports advanced triggers.

If you want to use one of the languages that are only available in 1.x, stay on the 1.x runtime. But don't use experimental languages for anything that you rely on, as there is no official support for them. You can request help by [creating GitHub issues](#), but support cases should not be opened for problems with experimental languages.

Language extensibility

The 2.x runtime is designed to offer [language extensibility](#). Among the first languages to be based on this extensibility model is Java, which is in preview in 2.x.

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](#)

Azure Functions C# developer reference

1/10/2018 • 6 min 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 2017 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.

FunctionName and trigger attributes

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,
        TraceWriter log)
    {
        log.Info($"C# function processed: {myQueueItem}");
    }
}
```

The `FunctionName` attribute marks the method as a function entry point. The name must be unique within a project.

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.

Additional binding attributes

Additional input and output binding attributes may be used. The following example modifies the preceding one by adding an output queue binding. The function writes the input queue message 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,
        TraceWriter log)
    {
        log.Info($"CopyQueueMessage function processed: {myQueueItem}");
        myQueueItemCopy = myQueueItem;
    }
}

```

Conversion to `function.json`

The build process creates a `function.json` file in a function folder in the build folder. 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"
}
```

The `function.json` file generation is performed by the NuGet package [Microsoft.NET.Sdk.Functions](#). The source code is available in the GitHub repo [azure-functions-vs-build-sdk](#).

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, review the article [Improper Instantiation antipattern](#).

Binding to method return value

You can use a method return value for an output binding, as shown in the following example:

```

public static class ReturnValueOutputBinding
{
    [FunctionName("CopyQueueMessageUsingReturnValue")]
    [return: Queue("myqueue-items-destination")]
    public static string Run(
        [QueueTrigger("myqueue-items-source-2")] string myQueueItem,
        TraceWriter log)
    {
        log.Info($"C# function processed: {myQueueItem}");
        return myQueueItem;
    }
}

```

Writing multiple output values

To write multiple values to an 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> myQueueItemCopy,
        TraceWriter log)
    {
        log.Info($"C# function processed: {myQueueItem}");
        myQueueItemCopy.Add($"Copy 1: {myQueueItem}");
        myQueueItemCopy.Add($"Copy 2: {myQueueItem}");
    }
}

```

Logging

To log output to your streaming logs in C#, include an argument of type `TraceWriter`. We recommend that you name it `log`. Avoid using `Console.WriteLine` in Azure Functions.

`TraceWriter` is defined in the [Azure WebJobs SDK](#). The log level for `TraceWriter` can be configured in `host.json`.

```

public static class SimpleExample
{
    [FunctionName("QueueTrigger")]
    public static void Run(
        [QueueTrigger("myqueue-items")] string myQueueItem,
        TraceWriter log)
    {
        log.Info($"C# function processed: {myQueueItem}");
    }
}

```

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 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,
        TraceWriter log)
    {
        log.Info($"BlobCopy function processed.");
        await blobInput.CopyToAsync(blobOutput, 4096, token);
    }
}
```

Cancellation tokens

Some operations require graceful shutdown. While it's always best to write code that can handle crashing, in cases where you want to handle shutdown requests, define a `CancellationToken` typed argument. A `CancellationToken` is provided to signal that a host shutdown is triggered.

```
public static class CancellationTokenExample
{
    [FunctionName("BlobCopy")]
    public static async Task RunAsync(
        [BlobTrigger("sample-images/{blobName}")] Stream blobInput,
        [Blob("sample-images-copies/{blobName}", FileAccess.Write)] Stream blobOutput,
        CancellationToken token)
    {
        await blobInput.CopyToAsync(blobOutput, 4096, token);
    }
}
```

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, TraceWriter log)
    {
        log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
        log.Info(GetEnvironmentVariable("AzureWebJobsStorage"));
        log.Info(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 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` or `IAsyncCollector` with imperative binding.

Single attribute example

The following example code creates a [Storage blob output binding](#) with blob path that's defined at runtime, 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,
        TraceWriter log)
    {
        log.Info($"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,
        TraceWriter log)
    {
        log.Info($"CreateBlobInDifferentStorageAccount function processed: {myQueueItem}");
        var attributes = new Attribute[]
        {
            new BlobAttribute($"samples-output/{myQueueItem}", FileAccess.Write),
            new StorageAccountAttribute("MyStorageAccount")
        };
        using (var writer = await binder.BindAsync<TextWriter>(attributes))
        {
            await writer.WriteAsync("Hello World!!");
        }
    }
}

```

Triggers and bindings

The following table lists the trigger and binding attributes that are available in an Azure Functions class library project. All attributes are in the namespace `Microsoft.Azure.WebJobs`.

TRIGGER	INPUT	OUTPUT
BlobTrigger	Blob	Blob
CosmosDBTrigger	DocumentDB	DocumentDB
EventHubTrigger		EventHub
HTTPTrigger		
QueueTrigger		Queue
ServiceBusTrigger		ServiceBus
TimerTrigger		
	ApiHubFile	ApiHubFile
	MobileTable	MobileTable
	Table	Table
		NotificationHub
		SendGrid
		Twilio

Next steps

[Learn more about triggers and bindings](#)

[Learn more about best practices for Azure Functions](#)

Azure Functions C# script (.csx) developer reference

1/9/2018 • 11 min 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.

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.WindowsAzure.Storage.Queue;  
using System;  
  
public static void Run(CloudQueueMessage myQueueItem, TraceWriter log)  
{  
    log.Info($"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, review the article [Improper Instantiation antipattern](#).

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"

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Verbose($"Log by run.csx: {DateTime.Now}");
    MyLogger(log, $"Log by MyLogger: {DateTime.Now}");
}
```

Example `mylogger.csx`:

```
public static void MyLogger(TraceWriter log, string logtext)
{
    log.Verbose(logtext);
}
```

Using a shared `.csx` file is a common pattern when you want to strongly type the data passed between functions by using a POCO object. In the following simplified example, an HTTP trigger and queue trigger share a POCO object named `Order` to strongly type the order data:

Example `run.csx` for HTTP trigger:

```

#load "..\shared\order.csx"

using System.Net;

public static async Task<HttpResponseMessage> Run(Order req, IAsyncCollector<Order> outputQueueItem, TraceWriter log)
{
    log.Info("C# HTTP trigger function received an order.");
    log.Info(req.ToString());
    log.Info("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;

public static void Run(Order myQueueItem, out Order outputQueueItem, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed order...");
    log.Info(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\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*:

```
{  
    "type": "queue",  
    "direction": "out",  
    "name": "$return",  
    "queueName": "outqueue",  
    "connection": "MyStorageConnectionString",  
}
```

```
public static string Run(string input, TraceWriter log)  
{  
    return input;  
}
```

Writing multiple output values

To write multiple values to an 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> myQueueItem, TraceWriter log)  
{  
    myQueueItem.Add("Hello");  
    myQueueItem.Add("World!");  
}
```

Logging

To log output to your streaming logs in C#, include an argument of type `TraceWriter`. We recommend that you name it `log`. Avoid using `Console.WriteLine` in Azure Functions.

`TraceWriter` is defined in the [Azure WebJobs SDK](#). The log level for `TraceWriter` can be configured in `host.json`.

```
public static void Run(string myBlob, TraceWriter log)  
{  
    log.Info($"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, token);  
}
```

Cancellation tokens

Some operations require graceful shutdown. While it's always best to write code that can handle crashing, in cases where you want to handle shutdown requests, define a `CancellationToken` typed argument. A `CancellationToken` is provided to

signal that a host shutdown is triggered.

```
public async static Task ProcessQueueMessageAsyncCancellationToken(
    string blobName,
    Stream blobInput,
    Stream blobOutput,
    CancellationToken token)
{
    await blobInput.CopyToAsync(blobOutput, 4096, token);
}
```

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;

public static Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
```

The following namespaces are automatically imported and are therefore optional:

- `System`
- `System.Collections.Generic`
- `System.IO`
- `System.Linq`
- `System.Net.Http`
- `System.Threading.Tasks`
- `Microsoft.Azure.WebJobs`
- `Microsoft.Azure.WebJobs.Host`

Referencing external assemblies

For framework assemblies, add references by using the `#r "AssemblyName"` directive.

```
#r "System.Web.Http"

using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public static Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter 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 your function app, such as in a `bin` folder in the function app root.
- 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 C# function, upload 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 reference to `Microsoft.ProjectOxford.Face` version 1.1.0:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Microsoft.ProjectOxford.Face": "1.1.0"
      }
    }
  }
}
```

In Azure Functions 1.x, 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. To use the types defined in the NuGet packages; just add the required `using` statements to your `run.csx` file.

In the Functions runtime, NuGet restore works by comparing `project.json` and `project.lock.json`. If the date and time stamps of the files **do not** match, a NuGet restore runs and NuGet downloads updated packages. However, if the date and time stamps of the files **do** match, NuGet does not perform a restore. Therefore, `project.lock.json` should not be deployed, as it causes NuGet to skip package restore. To avoid deploying the lock file, add the `project.lock.json` to the `.gitignore` file.

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](#).

Using a `project.json` file

1. Open the function in the Azure portal. The logs tab displays the package installation output.
2. To upload a `project.json` 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 `project.json` file is uploaded, you see output like 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\facaavalfunc\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`, as shown in the following code example:

```
public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
    log.Info(GetEnvironmentVariable("AzureWebJobsStorage"));
    log.Info(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` or `IAsyncCollector` 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.Write("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.Notifica"
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>

BINDING	ATTRIBUTE	ADD REFERENCE
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

10/5/2017 • 6 min 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.

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.

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: TraceWriter) =
    { 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 `TraceWriter`. For consistency, we recommend this argument is named `log`. For example:

```
let Run(blob: string, output: byref<string>, log: TraceWriter) =
    log.Verbose(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

let Run(req: HttpRequestMessage, log: TraceWriter) =
    ...
```

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

let Run(req: HttpRequestMessage, log: TraceWriter) =
    ...
```

The following assemblies are automatically added by the Azure Functions hosting environment:

- `mscorlib`,
- `System`
- `System.Core`
- `System.Xml`
- `System.Net.Http`
- `Microsoft.Azure.WebJobs`
- `Microsoft.Azure.WebJobs.Host`
- `Microsoft.Azure.WebJobs.Extensions`
- `System.Web.Http`
- `System.Net.Http.Formatting`.

In addition, the following assemblies are special cased and may be referenced by `simplename` (e.g.

`#r "AssemblyName"`):

- `Newtonsoft.Json`
- `Microsoft.WindowsAzure.Storage`
- `Microsoft.ServiceBus`
- `Microsoft.AspNet.WebHooks.Receivers`
- `Microsoft.AspNet.WebHooks.Common`.

If you need to reference a private assembly, you can upload the assembly file into a `bin` folder relative to your function and reference it by using the file name (e.g. `#r "MyAssembly.dll"`). For information on how to upload files to your function folder, see the following section on package management.

Editor Prelude

An editor that supports F# Compiler Services will not be aware of the namespaces and assemblies that Azure Functions automatically includes. As such, it can be useful to include a prelude that helps the editor find the assemblies you are using, and to explicitly open namespaces. For example:

```

#if !COMPILED
#I "../../bin/Binaries/WebJobs.Script.Host"
#r "Microsoft.Azure.WebJobs.Host.dll"
#endif

open System
open Microsoft.Azure.WebJobs.Host

let Run(blob: string, output: byref<string>, log: TraceWriter) =
    ...

```

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 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\facavalfunctest\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

let Run(timer: TimerInfo, log: TraceWriter) =
    log.Info("Storage = " + GetEnvironmentVariable("AzureWebJobsStorage"))
    log.Info("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: TraceWriter) =
    mylog log (sprintf "Timer: %s" DateTime.Now.ToString())
```

`logger.fsx`

```
let mylog(log: TraceWriter, text: string) =
    log.Verbose(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

11/29/2017 • 9 min to read • [Edit Online](#)

The JavaScript experience for Azure Functions makes it easy to export a function, which is passed as a `context` object for communicating with the runtime and for receiving and sending data via bindings.

This article assumes that you've already read the [Azure Functions developer reference](#).

Exporting a function

All JavaScript functions must export a single `function` via `module.exports` for the runtime to find the function and run it. This function must always include a `context` object.

```
// You must include a context, but other arguments are optional
module.exports = function(context) {
    // Additional inputs can be accessed by the arguments property
    if(arguments.length === 4) {
        context.log('This function has 4 inputs');
    }
};

// or you can include additional inputs in your arguments
module.exports = function(context, myTrigger, myInput, myOtherInput) {
    // function logic goes here :
};
```

Bindings of `direction === "in"` are passed along as function arguments, which means that you can use `arguments` to dynamically handle new inputs (for example, by using `arguments.length` to iterate over all your inputs). This functionality is convenient when you have only a trigger and no additional inputs, because you can predictably access your trigger data without referencing your `context` object.

The arguments are always passed along to the function in the order in which they occur in `function.json`, even if you don't specify them in your exports statement. For example, if you have `function(context, a, b)` and change it to `function(context, a)`, you can still get the value of `b` in function code by referring to `arguments[2]`.

All bindings, regardless of direction, are also passed along on the `context` object (see the following script).

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 is always the first parameter to a function and must be included because it has methods such as `context.done` and `context.log`, which are required to use the runtime correctly. 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(context) {
    // function logic goes here :
};
```

context.bindings property

```
context.bindings
```

Returns a named object that contains all your input and output data. For example, the following binding definition in your *function.json* lets you access the contents of the queue from the `context.bindings.myInput` object.

```
{
    "type": "queue",
    "direction": "in",
    "name": "myInput"
    ...
}
```

```
// 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
};
```

context.done method

```
context.done([err],[propertyBag])
```

Informs the runtime that your code has finished. You must call `context.done`, or else the runtime never knows that your function is complete, and the execution will time out.

The `context.done` method allows you to pass back both a user-defined error to the runtime and a property bag of properties that overwrite the properties 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 will overwrite the myOutput binding to be:
// -> text: hello there, world, noNumber: true
```

context.log method

```
context.log(message)
```

Allows you to write to the streaming console logs at the default trace level. On `context.log`, additional logging methods are available that let you write to the console log at other trace levels:

METHOD	DESCRIPTION
error(<i>message</i>)	Writes to error level logging, or lower.
warn(<i>message</i>)	Writes to warning level logging, or lower.
info(<i>message</i>)	Writes to info level logging, or lower.

METHOD	DESCRIPTION
verbose(message)	Writes to verbose level logging.

The following example writes to the console at the warning trace level:

```
context.log.warn("Something has happened.");
```

You can set the trace-level threshold for logging in the host.json file, or turn it off. For more information about how to write to the logs, see the next section.

Binding 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"
}
```

Other options for `dataType` are `stream` and `string`.

Writing trace output to the console

In Functions, you use the `context.log` methods to write trace output to the console. At this point, 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'});
```

The preceding code is equivalent to the following code:

```
context.log.info({hello: 'world'});
```

The following 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 to the console 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 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 functions. 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 `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 `response` object has the following properties:

PROPERTY	DESCRIPTION
<code>body</code>	An object that contains the body of the response.

PROPERTY	DESCRIPTION
<i>headers</i>	An object that contains the response headers.
<i>isRaw</i>	Indicates that formatting is skipped for the response.
<i>status</i>	The HTTP status code of the response.

Accessing the request and response

When you work with HTTP triggers, you can access the HTTP request and response objects in any of three ways:

- 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:

```
context.bindings.response = { status: 201, body: "Insert succeeded." };
```

- 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!' };
```

- By calling `context.done()`. A special kind 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"
}
```

This output binding expects you to supply the response when you call `done()`, as follows:

```
// Define a valid response object.
res = { status: 201, body: "Insert succeeded." };
context.done(null, res);
```

Node version and package management

The node version is currently locked at `6.5.0`. We're investigating adding support for more versions and making it configurable.

The following steps let you include packages in your function app:

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.

After the packages you need are installed, you import them to your function by calling `require('packagename')`, as in the following example:

```
// 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'});
```

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.

Environment variables

To get an environment variable or an app setting value, use `process.env`, as shown in the following code example:

```
module.exports = function (context, myTimer) {
    var timeStamp = new Date().toISOString();

    context.log('Node.js timer trigger function ran!', timeStamp);
    context.log(GetEnvironmentVariable("AzureWebJobsStorage"));
    context.log(GetEnvironmentVariable("WEBSITE_SITE_NAME"));

    context.done();
};

function GetEnvironmentVariable(name)
{
    return name + ": " + process.env[name];
}
```

Considerations for JavaScript functions

When you work with JavaScript functions, be aware of the considerations in the following two 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 auto-scale. For more information, see [Scale instance count manually or automatically](#).

TypeScript and CoffeeScript support

Because direct support does not yet exist for auto-compiling TypeScript or CoffeeScript via the runtime, such support needs to be handled outside the runtime, at deployment time.

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

1/8/2018 • 6 min to read • [Edit Online](#)

Programming model

Your Azure function should be a stateless class method that processes input and produces output. Although you are allowed to write instance methods, your function must not depend on any instance fields of the class. All function methods must have a `public` access modifier.

Triggers and annotations

Typically an Azure function is invoked because of an external trigger. Your function needs to process that trigger and its associated inputs and produce one or more outputs.

Java annotations are included in the `azure-functions-java-core` package to bind input and outputs to your methods. The supported input triggers and output binding annotations are included in the following table:

BINDING	ANNOTATION
CosmosDB	N/A
HTTP	<ul style="list-style-type: none">• <code>HttpTrigger</code>• <code>HttpOutput</code>
Mobile Apps	N/A
Notification Hubs	N/A
Storage Blob	<ul style="list-style-type: none">• <code>BlobTrigger</code>• <code>BlobInput</code>• <code>BlobOutput</code>
Storage Queue	<ul style="list-style-type: none">• <code>QueueTrigger</code>• <code>QueueOutput</code>
Storage Table	<ul style="list-style-type: none">• <code>TableInput</code>• <code>TableOutput</code>
Timer	<ul style="list-style-type: none">• <code>TimerTrigger</code>
Twilio	N/A

Trigger inputs and outputs can also be defined in the `function.json` for your application.

IMPORTANT

You must configure an Azure Storage account in your `local.settings.json` to run Azure Storage Blob, Queue, or Table triggers locally.

Example using annotations:

```
import com.microsoft.azure.serverless.functions.annotation.HttpTrigger;
import com.microsoft.azure.serverless.functions.ExecutionContext;

public class Function {
    public String echo(@HttpTrigger(name = "req", methods = {"post"}, authLevel =
AuthorizationLevel.ANONYMOUS)
        String req, ExecutionContext context) {
        return String.format(req);
    }
}
```

The same function written without annotations:

```
package com.example;

public class MyClass {
    public static String echo(String in) {
        return in;
    }
}
```

with the corresponding `function.json`:

```
{
  "scriptFile": "azure-functions-example.jar",
  "entryPoint": "com.example.MyClass.echo",
  "bindings": [
    {
      "type": "httpTrigger",
      "name": "req",
      "direction": "in",
      "authLevel": "anonymous",
      "methods": [ "post" ]
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ]
}
```

Data Types

You are free to use all the data types in Java for the input and output data, including native types; customized Java types and specialized Azure types defined in `azure-functions-java-core` package. The Azure Functions runtime attempts convert the input received into the type requested by your code.

Strings

Values passed into function methods will be cast to Strings if the corresponding input parameter type for the function is of type `String`.

Plain old Java objects (POJOs)

Strings formatted with JSON will be cast to Java types if the input of the function method expects that Java type. This conversion allows you to pass JSON inputs into your functions and work with Java types in your code without having to implement the conversion in your own code.

POJO types used as inputs to functions must have the same `public` access modifier as the function methods they are being used in. You don't have to declare POJO class fields `public`. For example, a JSON string `{ "x": 3 }` is able to be converted to the following POJO type:

```
public class MyData {  
    private int x;  
}
```

Binary data

Binary data is represented as a `byte[]` in your Azure Functions code. Bind binary inputs or outputs to your functions by setting the `dataType` field in your `function.json` to `binary`:

```
{  
    "scriptFile": "azure-functions-example.jar",  
    "entryPoint": "com.example.MyClass.echo",  
    "bindings": [  
        {  
            "type": "blob",  
            "name": "content",  
            "direction": "in",  
            "dataType": "binary",  
            "path": "container/myfile.bin",  
            "connection": "ExampleStorageAccount"  
        },  
    ]  
}
```

Then use it in your function code:

```
// Class definition and imports are omitted here  
public static String echoLength(byte[] content) {  
}
```

Use `OutputBinding<byte[]>` type to make a binary output binding.

Function method overloading

You are allowed to overload function methods with the same name but with different types. For example, you can have both `String echo(String s)` and `String echo(MyType s)` in one class, and Azure Functions runtime decides which one to invoke by examining the actual input type (for HTTP input, MIME type `text/plain` leads to `String` while `application/json` represents `MyType`).

Inputs

Inputs are divided into two categories in Azure Functions: one is the trigger input and the other is the additional input. Although they are different in `function.json`, the usage is identical in Java code. Let's take the following code snippet as an example:

```

package com.example;

import com.microsoft.azure.serverless.functions.annotation.BindingName;
import java.util.Optional;

public class MyClass {
    public static String echo(Optional<String> in, @BindingName("item") MyObject obj) {
        return "Hello, " + in.orElse("Azure") + " and " + obj.getKey() + ".";
    }

    private static class MyObject {
        public String getKey() { return this.RowKey; }
        private String RowKey;
    }
}

```

The `@BindingName` annotation accepts a `String` property that represents the name of the binding/trigger defined in `function.json`:

```
{
  "scriptFile": "azure-functions-example.jar",
  "entryPoint": "com.example.MyClass.echo",
  "bindings": [
    {
      "type": "httpTrigger",
      "name": "req",
      "direction": "in",
      "authLevel": "anonymous",
      "methods": [ "put" ],
      "route": "items/{id}"
    },
    {
      "type": "table",
      "name": "item",
      "direction": "in",
      "tableName": "items",
      "partitionKey": "Example",
      "rowKey": "{id}",
      "connection": "ExampleStorageAccount"
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ]
}
```

So when this function is invoked, the HTTP request payload passes an optional `String` for argument `in` and an Azure Table Storage `MyObject` type passed to argument `obj`. Use the `Optional<T>` type to handle inputs into your functions that can be null.

Outputs

Outputs can be expressed both in return value or output parameters. If there is only one output, you are recommended to use the return value. For multiple outputs, you have to use output parameters.

Return value is the simplest form of output, you just return the value of any type, and Azure Functions runtime will try to marshal it back to the actual type (such as an HTTP response). In `functions.json`, you use `$return` as the name of the output binding.

To produce multiple output values, use `OutputBinding<T>` type defined in the `azure-functions-java-core` package. If you need to make an HTTP response and push a message to a queue as well, you can write something like:

```
package com.example;

import com.microsoft.azure.serverless.functions.OutputBinding;
import com.microsoft.azure.serverless.functions.annotation.BindingName;

public class MyClass {
    public static String echo(String body,
        @QueueOutput(queueName = "messages", connection = "AzureWebJobsStorage", name = "queue")
    OutputBinding<String> queue) {
        String result = "Hello, " + body + ".";
        queue.setValue(result);
        return result;
    }
}
```

which should define the output binding in `function.json`:

```
{
  "scriptFile": "azure-functions-example.jar",
  "entryPoint": "com.example.MyClass.echo",
  "bindings": [
    {
      "type": "httpTrigger",
      "name": "req",
      "direction": "in",
      "authLevel": "anonymous",
      "methods": [ "post" ]
    },
    {
      "type": "queue",
      "name": "queue",
      "direction": "out",
      "queueName": "messages",
      "connection": "AzureWebJobsStorage"
    },
    {
      "type": "http",
      "name": "$return",
      "direction": "out"
    }
  ]
}
```

Specialized Types

Sometimes a function must have detailed control over inputs and outputs. Specialized types in the `azure-functions-java-core` package are provided for you to manipulate request information and tailor the return status of a HTTP trigger:

SPECIALIZED TYPE	TARGET	TYPICAL USAGE
<code>HttpRequestMessage<T></code>	HTTP Trigger	Get method, headers, or queries
<code>HttpResponseMessage<T></code>	HTTP Output Binding	Return status other than 200

NOTE

You can also use `@BindingName` annotation to get HTTP headers and queries. For example, `@BindingName("name") String query` iterates the HTTP request headers and queries and pass that value to the method. For example, `query` will be `"test"` if the request URL is `http://example.org/api/echo?name=test`.

Metadata

Metadata comes from different sources, like HTTP headers, HTTP queries, and [trigger metadata](#). Use the `@BindingName` annotation together with the metadata name to get the value.

For example, the `queryValue` in the following code snippet will be `"test"` if the requested URL is `http://{example.host}/api/metadata?name=test`.

```
package com.example;

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

public class MyClass {
    @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);
    }
}
```

Functions execution context

You interact with Azure Functions execution environment via the `ExecutionContext` object defined in the `azure-functions-java-core` package. Use the `ExecutionContext` object to use invocation information and functions runtime information in your code.

Logging

Access to the Functions runtime logger is available through the `ExecutionContext` object. This logger is tied to the Azure monitor and allows you to flag warnings and errors encountered during function execution.

The following example code logs a warning message when the request body received is empty.

```
import com.microsoft.azure.serverless.functions.annotation.HttpTrigger;
import com.microsoft.azure.serverless.functions.ExecutionContext;

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);
    }
}
```

Next steps

For more information, see the following resources:

- [Best practices for Azure Functions](#)
- [Azure Functions developer reference](#)
- [Azure Functions triggers and bindings](#)

Optimize the performance and reliability of Azure Functions

1/19/2018 • 7 min 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.

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, review the article [Improper Instantiation antipattern](#).

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 `functions.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": 20,  
    "maxConcurrentRequests": 10,  
    "dynamicThrottlesEnabled": false  
  }  
}
```

PROPERTY	DEFAULT	DESCRIPTION
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.
maxOutstandingRequests	-1	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows callers to employ time-based retry strategies, and also helps you to control maximum request latencies. This only controls queuing that occurs within the script host execution path. Other queues such as the ASP.NET request queue will still be in effect and unaffected by this setting. The default is unbounded.

PROPERTY	DEFAULT	DESCRIPTION
maxConcurrentRequests	-1	The maximum number of http functions that will be executed in parallel. This allows you to control concurrency, which can help manage resource utilization. For example, you might have an http function that uses a lot of system resources (memory/cpu/sockets) such that it causes issues when concurrency is too high. Or you might have a function that makes outbound requests to a third party service, and those calls need to be rate limited. In these cases, applying a throttle here can help. The default is unbounded.
dynamicThrottlesEnabled	false	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels.

Other host configuration options can be found in the [host configuration document](#).

Next steps

For more information, see the following resources:

Because Azure Functions uses Azure App Service, you should also be aware of App Service guidelines.

- [Patterns and Practices HTTP Performance Optimizations](#)

Work with Azure Functions Proxies

12/22/2017 • 8 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

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](#).

Currently, there is no portal experience for modifying back-end requests. To learn how to apply this capability from `proxies.json`, see [Define a requestOverrides object](#).

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](#).

Currently, there is no portal experience for modifying responses. To learn how to apply this capability from `proxies.json`, see [Define a responseOverrides object](#).

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 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.

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. The Azure Functions Proxies feature must be [enabled](#) for the file to be processed.

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.

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 `responseOverrides` 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.

Enable Azure Functions Proxies

Proxies are now enabled by default! If you were using an older version of the proxies preview and disabled proxies, you will need to manually enable proxies once in order for proxies to execute.

1. Open the [Azure portal](#), and then go to your function app.
2. Select **Function app settings**.
3. Switch **Enable Azure Functions Proxies (preview)** to **On**.

You can also return here to update the proxy runtime as new features become available.

Azure Functions Runtime Overview

12/5/2017 • 1 min to read • [Edit Online](#)

The Azure Functions Runtime provides a new way for you to take advantage of the simplicity and flexibility of the Azure Functions programming model on-premises. Built on the same open source roots as Azure Functions, Azure Functions Runtime is deployed on-premises to provide a nearly identical development experience as the cloud service.

The screenshot shows the Azure Functions Runtime interface. On the left, there's a sidebar with 'Subscriptions' (All subscriptions, Function Apps, TestFunction, Functions, TimerTriggerCSharp1), 'Integrate', and 'Manage' buttons. The main area has tabs for 'run.csx' (selected) and 'Save'. The code editor contains the following C# code:

```
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 }
```

On the right, there's a 'Logs' section with a scrollable list of log entries:

```
11/29/2017 10:25:05 AM [ANNESTG1] C# Timer trigger function executed at: 11/29/2017 10:25:04 AM
11/29/2017 10:25:05 AM [ANNESTG1] Function completed (Success, Id=f88202c8-c195-4f85-823d-a7dedfcbb02, Duration=2ms)
11/29/2017 10:25:10 AM [ANNESTG1] Function started (Id=ab749622-5663-4fef-93ee-243e644283fe)
11/29/2017 10:25:10 AM [ANNESTG1] C# Timer trigger function executed at: 11/29/2017 10:25:10 AM
11/29/2017 10:25:10 AM [ANNESTG1] Function completed (Success, Id=ab749622-5663-4fef-93ee-243e644283fe, Duration=12ms)
11/29/2017 10:25:15 AM [ANNESTG1] Function started (Id=97840bfc-0024-4837-bf98-28d897a7800c)
11/29/2017 10:25:15 AM [ANNESTG1] C# Timer trigger function executed at: 11/29/2017 10:25:15 AM
11/29/2017 10:25:15 AM [ANNESTG1] Function completed (Success, Id=97840bfc-0024-4837-bf98-28d897a7800c, Duration=2ms)
```

The Azure Functions Runtime provides a way for you to experience Azure Functions before committing to the cloud. In this way, the code assets you build can then be taken with you to the cloud when you migrate. The runtime also opens up new options for you, such as using the spare compute power of your on-premises computers to run batch processes overnight. You can also use devices within your organization to conditionally send data to other systems, both on-premises and in the cloud.

The Azure Functions Runtime consists of two pieces:

- Azure Functions Runtime Management Role
- Azure Functions Runtime Worker Role

Azure Functions Management Role

The Azure Functions Management Role provides a host for the management of your Functions on-premises. This role performs the following tasks:

- Hosting of the Azure Functions Management Portal, which is the same one you see in the [Azure portal](#). The portal provides a consistent experience that lets you develop your functions in the same way as you would in the Azure portal.
- Distributing functions across multiple Functions workers.
- Providing a publishing endpoint so that you can publish your functions direct from Microsoft Visual Studio by downloading and importing the publishing profile.

Azure Functions Worker Role

The Azure Functions Worker Roles are deployed in Windows Containers and are where your function code

executes. You can deploy multiple Worker Roles throughout your organization and this option is a key way in which customers can make use of spare compute power. One example of where spare compute exists in many organizations is machines powered on constantly but not being used for large periods of time.

Minimum Requirements

To get started with the Azure Functions Runtime, you must have a machine with Windows Server 2016 or Windows 10 Creators Update with access to a SQL Server instance.

Next Steps

Install the [Azure Functions Runtime preview](#)

Durable Functions overview (preview)

1/3/2018 • 12 min to read • [Edit Online](#)

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

The extension lets you define stateful workflows in a new type of function called an *orchestrator function*. Here are some of the advantages of orchestrator functions:

- They define workflows in code. No JSON schemas or designers are needed.
- They can call other functions synchronously and asynchronously. Output from called functions can be saved to local variables.
- They automatically checkpoint their progress whenever the function awaits. Local state is never lost if the process recycles or the VM reboots.

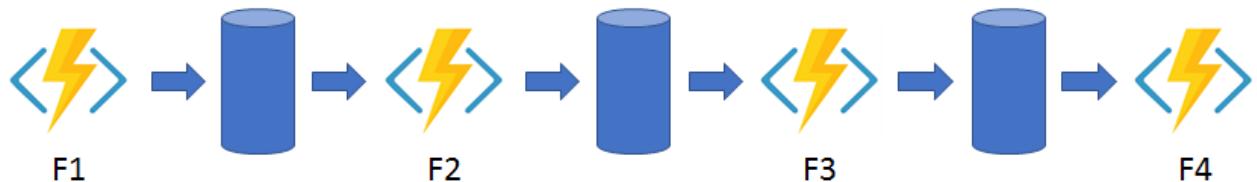
NOTE

Durable Functions is in preview and is an advanced extension for Azure Functions that is not appropriate for all applications. The rest of this article assumes that you have a strong familiarity with [Azure Functions](#) concepts and the challenges involved in serverless application development.

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

Pattern #1: Function chaining

Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function.



Durable Functions allows you to implement this pattern concisely in code.

```

public static async Task<object> Run(DurableOrchestrationContext ctx)
{
    try
    {
        var x = await ctx.CallActivityAsync<object>("F1");
        var y = await ctx.CallActivityAsync<object>("F2", x);
        var z = await ctx.CallActivityAsync<object>("F3", y);
        return await ctx.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // error handling/compensation goes here
    }
}

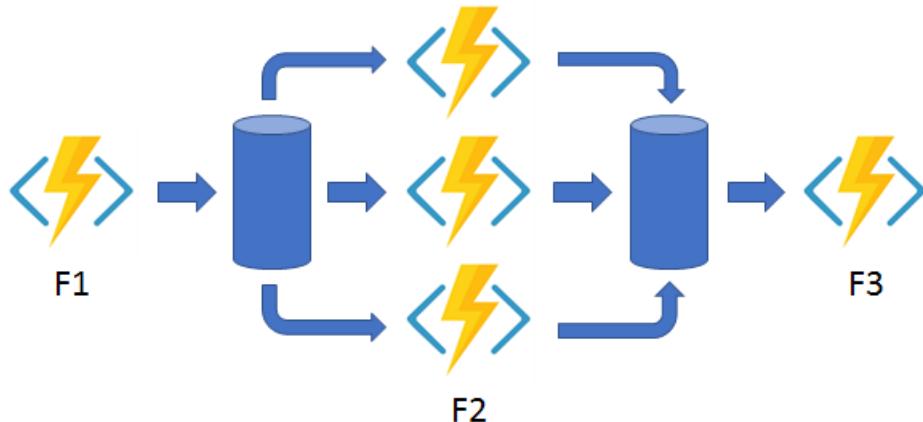
```

The values "F1", "F2", "F3", and "F4" are the names of other functions in the function app. Control flow is implemented using normal imperative coding constructs. That is, code executes top-down and can involve existing language control flow semantics, like conditionals, and loops. Error handling logic can be included in try/catch/finally blocks.

The `ctx` parameter ([DurableOrchestrationContext](#)) provides methods for invoking other functions by name, passing parameters, and returning function output. Each time the code calls `await`, the Durable Functions framework *checkpoints* the progress of the current function instance. If the process or VM recycles midway through the execution, the function instance resumes from the previous `await` call. More on this restart behavior later.

Pattern #2: Fan-out/fan-in

Fan-out/fan-in refers to the pattern of executing multiple functions in parallel, and then waiting for all to finish. Often some aggregation work is done on results returned from the functions.



With normal functions, fanning out can be done by having the function send multiple messages to a queue. However, fanning back in is much more challenging. You'd have to write code to track when the queue-triggered functions end and store function outputs. The Durable Functions extension handles this pattern with relatively simple code.

```

public static async Task Run(DurableOrchestrationContext ctx)
{
    var parallelTasks = new List<Task<int>>();

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

    await Task.WhenAll(parallelTasks);

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

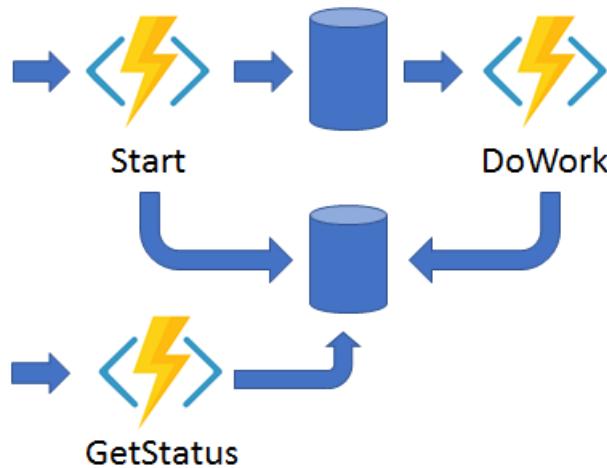
```

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

The automatic checkpointing that happens at the `await` call on `Task.WhenAll` ensures that any crash or reboot midway through does not require a restart of any already completed tasks.

Pattern #3: Async HTTP APIs

The third pattern is all about the problem of coordinating the state of long-running operations with external clients. A common way to implement this pattern is by having the long-running action triggered by an HTTP call, and then redirecting the client to a status endpoint that they can poll to learn when the operation completes.



Durable Functions provides built-in APIs that simplify the code you write for interacting with long-running function executions. The [samples](#) show a simple REST command that can be used to start new orchestrator function instances. Once an instance is started, the extension exposes webhook HTTP APIs that query the orchestrator function status. The following example shows the REST commands to start an orchestrator and to query its status. For clarity, some details are omitted from the example.

```

> curl -X POST https://myfunc.azurewebsites.net/orchestrators/DoWork -H "Content-Length: 0" -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec

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

> curl
https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec

{"runtimeStatus":"Running","lastUpdatedTime":"2017-03-16T21:20:47Z", ...}

> curl
https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 200 OK
Content-Length: 175
Content-Type: application/json

{"runtimeStatus":"Completed","lastUpdatedTime":"2017-03-16T21:20:57Z", ...}

```

Because the state is managed by the Durable Functions runtime, you don't have to implement your own status tracking mechanism.

Even though the Durable Functions extension has built-in webhooks for managing long-running orchestrations, you can implement this pattern yourself using your own function triggers (such as HTTP, queue, or Event Hub) and the `orchestrationClient` binding. For example, you could use a queue message to trigger termination. Or you could use an HTTP trigger protected by an Azure Active Directory authentication policy instead of the built-in webhooks that use a generated key for authentication.

```

// HTTP-triggered function to start a new orchestrator function instance.
public static async Task<HttpResponseMessage> Run(
    HttpRequestMessage req,
    DurableOrchestrationClient starter,
    string functionName,
    TraceWriter log)
{
    // Function name comes from the request URL.
    // Function input comes from the request content.
    dynamic eventData = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName, eventData);

    log.Info($"Started orchestration with ID = '{instanceId}'.");

    return starter.CreateCheckStatusResponse(req, instanceId);
}

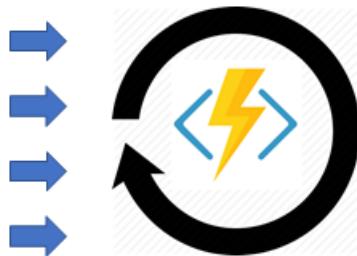
```

The `DurableOrchestrationClient` `starter` parameter is a value from the `orchestrationClient` output binding, which is part of the Durable Functions extension. It provides methods for starting, sending events to, terminating, and querying for new or existing orchestrator function instances. In the above example, an HTTP triggered-function takes in a `functionName` value from the incoming URL and passes that value to `StartNewAsync`. This binding API then returns a response that contains a `Location` header and additional information about the instance that can later be used to look up the status of the started instance or terminate it.

Pattern #4: Stateful singletons

Most functions have an explicit start and end and don't directly interact with external event sources. However, orchestrations support a [stateful singleton](#) pattern that allows them to behave like reliable [actors](#) in distributed computing.

The following diagram illustrates a function that runs in an infinite loop while processing events received from external sources.



While Durable Functions is not an implementation of the actor model, orchestrator functions do have many of the same runtime characteristics. For example, they are long-running (possibly endless), stateful, reliable, single-threaded, location-transparent, and globally addressable. This makes orchestrator functions useful for "actor"-like scenarios.

Ordinary functions are stateless and therefore not suited to implement a stateful singleton pattern. However, the Durable Functions extension makes the stateful singleton pattern relatively trivial to implement. The following code is a simple orchestrator function that implements a counter.

```
public static async Task Run(DurableOrchestrationContext ctx)
{
    int counterState = ctx.GetInput<int>();

    string operation = await ctx.WaitForExternalEvent<string>("operation");
    if (operation == "incr")
    {
        counterState++;
    }
    else if (operation == "decr")
    {
        counterState--;
    }

    ctx.ContinueAsNew(counterState);
}
```

This code is what you might describe as an "eternal orchestration" — that is, one that starts and never ends. It executes the following steps:

- Starts with an input value in `counterState`.
- Waits indefinitely for a message called `operation`.
- Performs some logic to update its local state.
- "Restarts" itself by calling `ctx.ContinueAsNew`.
- Awaits again indefinitely for the next operation.

Pattern #5: Human interaction

Many processes involve some kind of human interaction. The tricky thing about involving humans in an automated process is that people are not always as highly available and responsive as cloud services. Automated processes must allow for this, and they often do so by using timeouts and compensation logic.

One example of a business process that involves human interaction is an approval process. For example,

approval from a manager might be required for an expense report that exceeds a certain amount. If the manager does not approve within 72 hours (maybe they went on vacation), an escalation process kicks in to get the approval from someone else (perhaps the manager's manager).



This pattern can be implemented using an orchestrator function. The orchestrator would use a [durable timer](#) to request approval and escalate in case of timeout. It would wait for an [external event](#), which would be the notification generated by some human interaction.

```
public static async Task Run(DurableOrchestrationContext ctx)
{
    await ctx.CallActivityAsync("RequestApproval");
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = ctx.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = ctx.CreateTimer(dueTime, timeoutCts.Token);

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

The durable timer is created by calling `ctx.CreateTimer`. The notification is received by `ctx.WaitForExternalEvent`. And `Task.WhenAny` is called to decide whether to escalate (timeout happens first) or process approval (approval is received before timeout).

The technology

Behind the scenes, the Durable Functions extension is built on top of the [Durable Task Framework](#), an open source library on GitHub for building durable task orchestrations. Much like how Azure Functions is the serverless evolution of Azure WebJobs, Durable Functions is the serverless evolution of the Durable Task Framework. The Durable Task Framework is used heavily within Microsoft and outside as well to automate mission-critical processes. It's a natural fit for the serverless Azure Functions environment.

Event sourcing, checkpointing, and replay

Orchestrator functions reliably maintain their execution state using a cloud design pattern known as [Event Sourcing](#). Instead of directly storing the *current* state of an orchestration, the durable extension uses an append-only store to record the *full series of actions* taken by the function orchestration. This has many benefits, including improving performance, scalability, and responsiveness compared to "dumping" the full runtime state. Other benefits include providing eventual consistency for transactional data and maintaining full audit trails and

history. The audit trails themselves enable reliable compensating actions.

The use of Event Sourcing by this extension is transparent. Under the covers, the `await` operator in an orchestrator function yields control of the orchestrator thread back to the Durable Task Framework dispatcher. The dispatcher then commits any new actions that the orchestrator function scheduled (such as calling one or more child functions or scheduling a durable timer) to storage. This transparent commit action appends to the *execution history* of the orchestration instance. The history is stored in a storage table. The commit action then adds messages to a queue to schedule the actual work. At this point, the orchestrator function can be unloaded from memory. Billing for it stops if you're using the Azure Functions Consumption Plan. When there is more work to do, the function is restarted and its state is reconstructed.

Once an orchestration function is given more work to do (for example, a response message is received or a durable timer expires), the orchestrator wakes up again and re-executes the entire function from the start in order to rebuild the local state. If during this replay the code tries to call a function (or do any other async work), the Durable Task Framework consults with the *execution history* of the current orchestration. If it finds that the activity function has already executed and yielded some result, it replays that function's result, and the orchestrator code continues running. This continues happening until the function code gets to a point where either it is finished or it has scheduled new async work.

Orchestrator code constraints

The replay behavior creates constraints on the type of code that can be written in an orchestrator function. For example, orchestrator code must be deterministic, as it will be replayed multiple times and must produce the same result each time. The complete list of constraints can be found in the [Orchestrator code constraints](#) section of the [Checkpointing and restart](#) article.

Language support

Currently C# is the only supported language for Durable Functions. This includes orchestrator functions and activity functions. In the future, we will add support for all languages that Azure Functions supports. See the Azure Functions [GitHub repository issues list](#) to see the latest status of our additional language support work.

Monitoring and diagnostics

The Durable Functions extension automatically emits structured tracking data to [Application Insights](#) when the function app is configured with an Application Insights instrumentation key. This tracking data can be used to monitor the behavior and progress of your orchestrations.

Here is an example of what the Durable Functions tracking events look like in the Application Insights portal using [Application Insights Analytics](#):

New Query 1*									
traces									
extend hubName = customDimensions["prop_hubName"]									
extend functionName = customDimensions ["prop_functionName"]									
extend state = customDimensions ["prop_state"]									
extend instanceId = customDimensions ["prop_instanceId"]									
extend category = customDimensions ["prop_category"]									
extend extensionVersion = customDimensions ["prop_extensionVersion"]									
where timestamp >= ago(10m)									
where category == "Host.Triggers.DurableTask"									
where hubName == "UnhandledActivityException"									
order by timestamp asc									
project timestamp, extensionVersion, instanceId, category, hubName, state, isReplay, message									
Completed									
TABLE									
Columns v									
Drag a column header and drop it here to group by that column									
timestamp [UTC]									
extensionVersion									
instanceId									
category									
hubName									
state									
isReplay									
message									
2017-09-07T01:57:30.008	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Scheduled	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Orchestrator)', version '' scheduled...	
2017-09-07T01:57:30.312	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Started	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Orchestrator)', version '' started...	
2017-09-07T01:57:30.314	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Scheduled	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Activity)', version '' scheduled...	
2017-09-07T01:57:30.315	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Awaited	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Orchestrator)', version '' awaited...	
2017-09-07T01:57:30.663	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Started	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Activity)', version '' started. I...	
2017-09-07T01:57:30.709	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Failed	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Activity)', version '' failed wit...	
2017-09-07T01:57:31.266	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Scheduled	True	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Activity)', version '' scheduled...	
2017-09-07T01:57:31.266	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Started	True	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Orchestrator)', version '' started...	
2017-09-07T01:57:31.268	0.2.2.0	3917b16e77b3439891dc2299c9c41ed3	Host.Triggers.DurableTask	UnhandledActivityException	Failed	False	3917b16e77b3439891dc2299c9c41ed3:	Function 'Throw (Orchestrator)', version '' failed...	

There is a lot of useful structured data packed into the `customDimensions` field in each log entry. Here is an example of one such entry fully expanded.

customDimensions	{"prop__{OriginalFormat}":"{instanceId}: Function '{functionName}' triggered by '{triggerType}' at {startTime}."}
prop__{OriginalFormat}	{instanceId}: Function '{functionName}' ({functionType}) triggered by '{triggerType}' at {startTime}.
Category	Microsoft.Azure.WebJobs.Extensions.DurableTask
prop__functionName	Throw
prop__functionType	Orchestrator
prop__instanceId	c444d880b6d7481182bc4e98a50da265
prop__slotName	Production
prop__isReplay	False
prop__hubName	UnhandledActivityException
prop__appName	LingsDurableFunctionApp
prop__state	Scheduled
prop__reason	NewInstance

Because of the replay behavior of the Durable Task Framework dispatcher, you can expect to see redundant log entries for replayed actions. This can be useful to understand the replay behavior of the core engine. The [Diagnostics](#) article shows sample queries that filter out replay logs so you can see just the "real-time" logs.

Storage and scalability

The Durable Functions extension uses Azure Storage queues, tables, and blobs to persist execution history state and trigger function execution. The default storage account for the function app can be used, or you can configure a separate storage account. You might want a separate account due to storage throughput limits. The orchestrator code you write does not need to (and should not) interact with the entities in these storage accounts. The entities are managed directly by the Durable Task Framework as an implementation detail.

Orchestrator functions schedule activity functions and receive their responses via internal queue messages. When a function app runs in the Azure Functions Consumption plan, these queues are monitored by the [Azure Functions Scale Controller](#) and new compute instances are added as needed. When scaled out to multiple VMs, an orchestrator function may run on one VM while activity functions it calls run on several different VMs. You can find more details on the scale behavior of Durable Functions in [Performance and scale](#).

Table storage is used to store the execution history for orchestrator accounts. Whenever an instance rehydrates on a particular VM, it fetches its execution history from table storage so that it can rebuild its local state. One of the convenient things about having the history available in Table storage is that you can take a look and see the history of your orchestrations using tools such as [Microsoft Azure Storage Explorer](#).

The screenshot shows the Microsoft Azure Storage Explorer interface with the 'DurableFunctionsHubHistory' table selected in the top navigation bar. The table contains the following data:

	RowKey	Timestamp	EventId	EventType	IsPlayed	Name	Version	OrchestrationStatus	Input
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000000	2017-04-29T00:45:49.244Z	-1	OrchestratorStarted	false				
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000001	2017-04-29T00:45:49.244Z	-1	ExecutionStarted	true	ProcessWorkBatch			0
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000002	2017-04-29T00:45:49.245Z	0	TaskScheduled	false	HelloWorld			
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000003	2017-04-29T00:45:49.245Z	-1	OrchestratorCompleted	false				
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000004	2017-04-29T00:45:50.962Z	-1	OrchestratorStarted	false				
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000005	2017-04-29T00:45:50.962Z	-1	TaskCompleted	true				
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000006	2017-04-29T00:45:50.962Z	1	ExecutionCompleted	false			Completed	
jcd38f22c	1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000007	2017-04-29T00:45:50.963Z	-1	OrchestratorCompleted	false				

Showing 1 to 8 of 8 cached items

WARNING

While it's easy and convenient to see execution history in table storage, avoid taking any dependency on this table. It may change as the Durable Functions extension evolves.

Known issues and FAQ

In general, all known issues should be tracked in the [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. Even if you simply want to ask a question, feel free to open a GitHub issue and tag it as a question.

Next steps

[Continue reading Durable Functions documentation](#)

[Install the Durable Functions extension and samples](#)

Bindings for Durable Functions (Azure Functions)

1/8/2018 • 8 min to read • [Edit Online](#)

The [Durable Functions](#) extension introduces two new trigger bindings that control the execution of orchestrator and activity functions. It also introduces an output binding that acts as a client for the Durable Functions runtime.

Orchestration triggers

The orchestration trigger enables you to author durable orchestrator functions. This trigger supports starting new orchestrator function instances and resuming existing orchestrator function instances that are "awaiting" a task.

When you use the Visual Studio tools for Azure Functions, the orchestration trigger is configured using the [OrchestrationTriggerAttribute](#) .NET attribute.

When you write orchestrator functions in scripting languages (for example, in the Azure portal), the orchestration trigger is defined by the following JSON object in the `bindings` array of the *function.json* file:

```
{  
  "name": "<Name of input parameter in function signature>",  
  "orchestration": "<Optional - name of the orchestration>",  
  "version": "<Optional - version label of this orchestrator function>",  
  "type": "orchestrationTrigger",  
  "direction": "in"  
}
```

- `orchestration` is the name of the orchestration. This is the value that clients must use when they want to start new instances of this orchestrator function. This property is optional. If not specified, the name of the function is used.
- `version` is a version label of the orchestration. Clients that start a new instance of an orchestration must include the matching version label. This property is optional. If not specified, the empty string is used. For more information on versioning, see [Versioning](#).

NOTE

Setting values for `orchestration` or `version` properties is not recommended at this time.

Internally this trigger binding polls a series of queues in the default storage account for the function app. These queues are internal implementation details of the extension, which is why they are not explicitly configured in the binding properties.

Trigger behavior

Here are some notes about the orchestration trigger:

- **Single-threading** - A single dispatcher thread is used for all orchestrator function execution on a single host instance. For this reason, it is important to ensure that orchestrator function code is efficient and doesn't perform any I/O. It is also important to ensure that this thread does not do any async work except when awaiting on Durable Functions-specific task types.
- **Poison-message handling** - There is no poison message support in orchestration triggers.
- **Message visibility** - Orchestration trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and

healthy.

- **Return values** - Return values are serialized to JSON and persisted to the orchestration history table in Azure Table storage. These return values can be queried by the orchestration client binding, described later.

WARNING

Orchestrator functions should never use any input or output bindings other than the orchestration trigger binding. Doing so has the potential to cause problems with the Durable Task extension because those bindings may not obey the single-threading and I/O rules.

Trigger usage

The orchestration trigger binding supports both inputs and outputs. Here are some things to know about input and output handling:

- **inputs** - Orchestration functions support only [DurableOrchestrationContext](#) as a parameter type. Deserialization of inputs directly in the function signature is not supported. Code must use the [GetInput<T>](#) method to fetch orchestrator function inputs. These inputs must be JSON-serializable types.
- **outputs** - Orchestration triggers support output values as well as inputs. The return value of the function is used to assign the output value and must be JSON-serializable. If a function returns `Task` or `void`, a `null` value will be saved as the output.

NOTE

Orchestration triggers are only supported in C# at this time.

Trigger sample

The following is an example of what the simplest "Hello World" C# orchestrator function might look like:

```
[FunctionName("HelloWorld")]
public static string Run([OrchestrationTrigger] DurableOrchestrationContext context)
{
    string name = context.GetInput<string>();
    return $"Hello {name}!";
}
```

Most orchestrator functions call activity functions, so here is a "Hello World" example that demonstrates how to call an activity function:

```
[FunctionName("HelloWorld")]
public static async Task<string> Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    string name = await context.GetInput<string>();
    string result = await context.CallActivityAsync<string>("SayHello", name);
    return result;
}
```

Activity triggers

The activity trigger enables you to author functions that are called by orchestrator functions.

If you're using Visual Studio, the activity trigger is configured using the [ActivityTriggerAttribute](#) .NET attribute.

If you're using the Azure portal for development, the activity trigger is defined by the following JSON object in the

`bindings` array of `function.json`:

```
{  
    "name": "<Name of input parameter in function signature>",  
    "activity": "<Optional - name of the activity>",  
    "version": "<Optional - version label of this activity function>",  
    "type": "activityTrigger",  
    "direction": "in"  
}
```

- `activity` is the name of the activity. This is the value that orchestrator functions use to invoke this activity function. This property is optional. If not specified, the name of the function is used.
- `version` is a version label of the activity. Orchestrator functions that invoke an activity must include the matching version label. This property is optional. If not specified, the empty string is used. For more information, see [Versioning](#).

NOTE

Setting values for `activity` or `version` properties is not recommended at this time.

Internally this trigger binding polls a queue in the default storage account for the function app. This queue is an internal implementation detail of the extension, which is why it is not explicitly configured in the binding properties.

Trigger behavior

Here are some notes about the activity trigger:

- **Threading** - Unlike the orchestration trigger, activity triggers don't have any restrictions around threading or I/O. They can be treated like regular functions.
- **Poison-message handling** - There is no poison message support in activity triggers.
- **Message visibility** - Activity trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Return values are serialized to JSON and persisted to the orchestration history table in Azure Table storage.

WARNING

The storage backend for activity functions is an implementation detail and user code should not interact with these storage entities directly.

Trigger usage

The activity trigger binding supports both inputs and outputs, just like the orchestration trigger. Here are some things to know about input and output handling:

- **inputs** - Activity functions natively use `DurableActivityContext` as a parameter type. Alternatively, an activity function can be declared with any parameter type that is JSON-serializable. When you use `DurableActivityContext`, you can call `GetInput<T>` to fetch and deserialize the activity function input.
- **outputs** - Activity functions support output values as well as inputs. The return value of the function is used to assign the output value and must be JSON-serializable. If a function returns `Task` or `void`, a `null` value will be saved as the output.
- **metadata** - Activity functions can bind to a `string instanceId` parameter to get the instance ID of the parent orchestration.

NOTE

Activity triggers are not currently supported in Node.js functions.

Trigger sample

The following is an example of what a simple "Hello World" C# activity function might look like:

```
[FunctionName("SayHello")]
public static string SayHello([ActivityTrigger] DurableActivityContext helloContext)
{
    string name = helloContext.GetInput<string>();
    return $"Hello {name}!";
}
```

The default parameter type for the `ActivityTriggerAttribute` binding is `DurableActivityContext`. However, activity triggers also support binding directly to JSON-serializable types (including primitive types), so the same function could be simplified as follows:

```
[FunctionName("SayHello")]
public static string SayHello([ActivityTrigger] string name)
{
    return $"Hello {name}!";
}
```

Orchestration client

The orchestration client binding enables you to write functions which interact with orchestrator functions. For example, you can act on orchestration instances in the following ways:

- Start them.
- Query their status.
- Terminate them.
- Send events to them while they're running.

If you're using Visual Studio, you can bind to the orchestration client by using the `OrchestrationClientAttribute` .NET attribute.

If you're using scripting languages (e.g. .csx files) for development, the orchestration trigger is defined by the following JSON object in the `bindings` array of `function.json`:

```
{
    "name": "<Name of input parameter in function signature>",
    "taskHub": "<Optional - name of the task hub>",
    "connectionName": "<Optional - name of the connection string app setting>",
    "type": "orchestrationClient",
    "direction": "out"
}
```

- `taskHub` - Used in scenarios where multiple function apps share the same storage account but need to be isolated from each other. If not specified, the default value from `host.json` is used. This value must match the value used by the target orchestrator functions.
- `connectionName` - The name of an app setting that contains a storage account connection string. The storage account represented by this connection string must be the same one used by the target orchestrator functions. If not specified, the default storage account connection string for the function app is used.

NOTE

In most cases, we recommend that you omit these properties and rely on the default behavior.

Client usage

In C# functions, you typically bind to `DurableOrchestrationClient`, which gives you full access to all client APIs supported by Durable Functions. APIs on the client object include:

- [StartNewAsync](#)
- [GetStatusAsync](#)
- [TerminateAsync](#)
- [RaiseEventAsync](#)

Alternatively, you can bind to `IAsyncCollector<T>` where `T` is [StartOrchestrationArgs](#) or [JObject](#).

See the [DurableOrchestrationClient](#) API documentation for additional details on these operations.

Client sample (Visual Studio development)

Here is an example queue-triggered function that starts a "HelloWorld" orchestration.

```
[FunctionName("QueueStart")]
public static Task Run(
    [QueueTrigger("durable-function-trigger")] string input,
    [OrchestrationClient] DurableOrchestrationClient starter)
{
    // Orchestration input comes from the queue message content.
    return starter.StartNewAsync("HelloWorld", input);
}
```

Client sample (not Visual Studio)

If you're not using Visual Studio for development, you can create the following `function.json` file. This example shows how to configure a queue-triggered function that uses the durable orchestration client binding:

```
{
  "bindings": [
    {
      "name": "input",
      "type": "queueTrigger",
      "queueName": "durable-function-trigger",
      "direction": "in"
    },
    {
      "name": "starter",
      "type": "orchestrationClient",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Following are language-specific samples that start new orchestrator function instances.

C# Sample

The following sample shows how to use the durable orchestration client binding to start a new function instance from a C# script function:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static Task<string> Run(string input, DurableOrchestrationClient starter)
{
    return starter.StartNewAsync("HelloWorld", input);
}
```

Node.js Sample

The following sample shows how to use the durable orchestration client binding to start a new function instance from a Node.js function:

```
module.exports = function (context, input) {
    var id = generateSomeUniqueId();
    context.bindings.starter = [
        {
            FunctionName: "HelloWorld",
            Input: input,
            InstanceId: id
        }
    ];

    context.done(null, id);
};
```

More details on starting instances can be found in [Instance management](#).

Next steps

[Learn about checkpointing and replay behaviors](#)

Checkpoints and replay in Durable Functions (Azure Functions)

12/15/2017 • 6 min to read • [Edit Online](#)

One of the key attributes of Durable Functions is **reliable execution**. Orchestrator functions and activity functions may be running on different VMs within a data center, and those VMs or the underlying networking infrastructure is not 100% reliable.

In spite of this, Durable Functions ensures reliable execution of orchestrations. It does so by using storage queues to drive function invocation and by periodically checkpointing execution history into storage tables (using a cloud design pattern known as [Event Sourcing](#)). That history can then be replayed to automatically rebuild the in-memory state of an orchestrator function.

Orchestration history

Suppose you have the following orchestrator function.

```
[FunctionName("E1_HelloSequence")]
public static async Task<List<string>> Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    var outputs = new List<string>();

    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "London"));

    // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
    return outputs;
}
```

At each `await` statement, the Durable Task Framework checkpoints the execution state of the function into table storage. This state is what is referred to as the *orchestration history*.

History table

Generally speaking, the Durable Task Framework does the following at each checkpoint:

1. Saves execution history into Azure Storage tables.
2. Enqueues messages for functions the orchestrator wants to invoke.
3. Enqueues messages for the orchestrator itself — for example, durable timer messages.

Once the checkpoint is complete, the orchestrator function is free to be removed from memory until there is more work for it to do.

NOTE

Azure Storage does not provide any transactional guarantees between saving data into table storage and queues. To handle failures, the Durable Functions storage provider uses *eventual consistency* patterns. These patterns ensure that no data is lost if there is a crash or loss of connectivity in the middle of a checkpoint.

Upon completion, the history of the function shown earlier looks something like the following in Azure Table Storage (abbreviated for illustration purposes):

PARTITIONKEY (INSTANCEID)	EVENTTYPE	TIMESTAMP	INPUT	NAME	RESULT	STATUS
eaee885b	OrchestratorStarted	2017-05-05T18:45:32.362Z				
eaee885b	ExecutionStarted	2017-05-05T18:45:28.852Z	null	E1_HelloSequence		
eaee885b	TaskScheduled	2017-05-05T18:45:32.670Z		E1_SayHello		
eaee885b	OrchestratorCompleted	2017-05-05T18:45:32.670Z				
eaee885b	OrchestratorStarted	2017-05-05T18:45:34.232Z				
eaee885b	TaskCompleted	2017-05-05T18:45:34.201Z			""Hello Tokyo!""	
eaee885b	TaskScheduled	2017-05-05T18:45:34.435Z		E1_SayHello		
eaee885b	OrchestratorCompleted	2017-05-05T18:45:34.435Z				
eaee885b	OrchestratorStarted	2017-05-05T18:45:34.857Z				
eaee885b	TaskCompleted	2017-05-05T18:45:34.763Z			""Hello Seattle!""	
eaee885b	TaskScheduled	2017-05-05T18:45:34.857Z		E1_SayHello		
eaee885b	OrchestratorCompleted	2017-05-05T18:45:34.857Z				
eaee885b	OrchestratorStarted	2017-05-05T18:45:35.032Z				

PARTITIONKEY (INSTANCEID)	EVENTTYPE	TIMESTAMP	INPUT	NAME	RESULT	STATUS
eaee885b	TaskCompleted	2017-05-05T18:45:34.919Z			""Hello London!""	
eaee885b	ExecutionCompleted	2017-05-05T18:45:35.044Z			"""Hello Tokyo!""; ""Hello Seattle!""; ""Hello London!"""	Completed
eaee885b	OrchestratorCompleted	2017-05-05T18:45:35.044Z				

A few notes on the column values:

- **PartitionKey:** Contains the instance ID of the orchestration.
- **EventType:** Represents the type of the event. May be one of the following types:
 - **OrchestrationStarted:** The orchestrator function resumed from an await or is running for the first time. The `Timestamp` column is used to populate the deterministic value for the [CurrentUtcDateTime](#) API.
 - **ExecutionStarted:** The orchestrator function started executing for the first time. This event also contains the function input in the `Input` column.
 - **TaskScheduled:** An activity function was scheduled. The name of the activity function is captured in the `Name` column.
 - **TaskCompleted:** An activity function completed. The result of the function is in the `Result` column.
 - **TimerCreated:** A durable timer was created. The `FireAt` column contains the scheduled UTC time at which the timer expires.
 - **TimerFired:** A durable timer fired.
 - **EventRaised:** An external event was sent to the orchestration instance. The `Name` column captures the name of the event and the `Input` column captures the payload of the event.
 - **OrchestratorCompleted:** The orchestrator function awaited.
 - **ContinueAsNew:** The orchestrator function completed and restarted itself with new state. The `Result` column contains the value, which is used as the input in the restarted instance.
 - **ExecutionCompleted:** The orchestrator function ran to completion (or failed). The outputs of the function or the error details are stored in the `Result` column.
- **Timestamp:** The UTC timestamp of the history event.
- **Name:** The name of the function that was invoked.
- **Input:** The JSON-formatted input of the function.
- **Result:** The output of the function; that is, its return value.

WARNING

While it's useful as a debugging tool, don't take any dependency on this table. It may change as the Durable Functions extension evolves.

Every time the function resumes from an `await`, the Durable Task Framework reruns the orchestrator function from scratch. On each rerun it consults the execution history to determine whether the current async operation has taken place. If the operation took place, the framework replays the output of that operation immediately and moves on to the next `await`. This process continues until the entire history has been replayed, at which point all

the local variables in the orchestrator function are restored to their previous values.

Orchestrator code constraints

The replay behavior creates constraints on the type of code that can be written in an orchestrator function:

- Orchestrator code must be **deterministic**. It will be replayed multiple times and must produce the same result each time. For example, no direct calls to get the current date/time, get random numbers, generate random GUIDs, or call into remote endpoints.

If orchestrator code needs to get the current date/time, it should use the [CurrentUtcDateTime](#) API, which is safe for replay.

Non-deterministic operations must be done in activity functions. This includes any interaction with other input or output bindings. This ensures that any non-deterministic values will be generated once on the first execution and saved into the execution history. Subsequent executions will then use the saved value automatically.

- Orchestrator code should be **non-blocking**. For example, that means no I/O and no calls to `Thread.Sleep` or equivalent APIs.

If an orchestrator needs to delay, it can use the [CreateTimer](#) API.

- Orchestrator code must **never initiate any async operation** except by using the [DurableOrchestrationContext](#) API. For example, no `Task.Run`, `Task.Delay` or `HttpClient.SendAsync`. The Durable Task Framework executes orchestrator code on a single thread and cannot interact with any other threads that could be scheduled by other async APIs.
- **Infinite loops should be avoided** in orchestrator code. Because the Durable Task Framework saves execution history as the orchestration function progresses, an infinite loop could cause an orchestrator instance to run out of memory. For infinite loop scenarios, use APIs such as [ContinueAsNew](#) to restart the function execution and discard previous execution history.

While these constraints may seem daunting at first, in practice they aren't hard to follow. The Durable Task Framework attempts to detect violations of the above rules and throws a `NonDeterministicOrchestrationException`. However, this detection behavior is best-effort, and you shouldn't depend on it.

NOTE

All of these rules apply only to functions triggered by the `orchestrationTrigger` binding. Activity functions triggered by the `activityTrigger` binding, and functions that use the `orchestrationClient` binding, have no such limitations.

Durable tasks

NOTE

This section describes internal implementation details of the Durable Task Framework. You can use Durable Functions without knowing this information. It is intended only to help you understand the replay behavior.

Tasks that can be safely awaited in orchestrator functions are occasionally referred to as *durable tasks*. These are tasks that are created and managed by the Durable Task Framework. Examples are the tasks returned by `CallActivityAsync`, `WaitForExternalEvent`, and `CreateTimer`.

These *durable tasks* are internally managed by using a list of `TaskCompletionSource` objects. During replay, these tasks get created as part of orchestrator code execution and are completed as the dispatcher enumerates the

corresponding history events. This is all done synchronously using a single thread until all the history has been replayed. Any durable tasks, which are not completed by the end of history replay has appropriate actions carried out. For example, a message may be enqueued to call an activity function.

The execution behavior described here should help you understand why orchestrator function code must never `await` a non-durable task: the dispatcher thread cannot wait for it to complete and any callback by that task could potentially corrupt the tracking state of the orchestrator function. Some runtime checks are in place to try to prevent this.

If you'd like more information about how the Durable Task Framework executes orchestrator functions, the best thing to do is to consult the [Durable Task source code on GitHub](#). In particular, see [TaskOrchestrationExecutor.cs](#) and [TaskOrchestrationContext.cs](#)

Next steps

[Learn about instance management](#)

Manage instances in Durable Functions (Azure Functions)

12/15/2017 • 3 min to read • [Edit Online](#)

Durable Functions orchestration instances can be started, terminated, queried, and sent notification events. All instance management is done using the [orchestration client binding](#). This article goes into the details of each instance management operation.

Starting instances

The [StartNewAsync](#) method on the [DurableOrchestrationClient](#) starts a new instance of an orchestrator function. Instances of this class can be acquired using the `orchestrationClient` binding. Internally, this method enqueues a message into the control queue, which then triggers the start of a function with the specified name that uses the `orchestrationTrigger` trigger binding.

The parameters to [StartNewAsync](#) are as follows:

- **Name:** The name of the orchestrator function to schedule.
- **Input:** Any JSON-serializable data which should be passed as the input to the orchestrator function.
- **InstanceId:** (Optional) The unique ID of the instance. If not specified, a random instance ID will be generated.

Here is a simple C# example:

```
[FunctionName("HelloWorldManualStart")]
public static Task Run(
    [ManualTrigger] string input,
    [OrchestrationClient] DurableOrchestrationClient starter,
    TraceWriter log)
{
    string instanceId = starter.StartNewAsync("HelloWorld", input);
    log.Info($"Started orchestration with ID = '{instanceId}'.");
}
```

For non-.NET languages, the function output binding can be used to start new instances as well. In this case, any JSON-serializable object which has the above three parameters as fields can be used. For example, consider the following Node.js function:

```
module.exports = function (context, input) {
    var id = generateSomeUniqueId();
    context.bindings.starter = [
        {
            FunctionName: "HelloWorld",
            Input: input,
            InstanceId: id
        }
    ];

    context.done(null);
};
```

NOTE

We recommend that you use a random identifier for the instance ID. This will help ensure an equal load distribution when scaling orchestrator functions across multiple VMs. The proper time to use non-random instance IDs is when the ID must come from an external source or when implementing the [singleton orchestrator](#) pattern.

Querying instances

The [GetStatusAsync](#) method on the [DurableOrchestrationClient](#) class queries the status of an orchestration instance. It takes an `instanceId` as a parameter and returns an object with the following properties:

- **Name**: The name of the orchestrator function.
- **InstanceId**: The instance ID of the orchestration (should be the same as the `instanceId` input).
- **CreatedTime**: The time at which the orchestrator function started running.
- **LastUpdatedTime**: The time at which the orchestration last checkpointed.
- **Input**: The input of the function as a JSON value.
- **Output**: The output of the function as a JSON value (if the function has completed). If the orchestrator function failed, this property will include the failure details. If the orchestrator function was terminated, this property will include the provided reason for the termination (if any).
- **RuntimeStatus**: One of the following values:
 - **Running**: The instance has started running.
 - **Completed**: The instance has completed normally.
 - **ContinuedAsNew**: The instance has restarted itself with a new history. This is a transient state.
 - **Failed**: The instance failed with an error.
 - **Terminated**: The instance was abruptly terminated.

This method returns `null` if the instance either doesn't exist or has not yet started running.

```
[FunctionName("GetStatus")]
public static async Task Run(
    [OrchestrationClient] DurableOrchestrationClient client,
    [ManualTrigger] string instanceId)
{
    var status = await client.GetStatusAsync(instanceId);
    // do something based on the current status.
}
```

NOTE

Instance query is currently only supported for C# orchestrator functions.

Terminating instances

A running instance can be terminated using the [TerminateAsync](#) method of the [DurableOrchestrationClient](#) class.

The two parameters are an `instanceId` and a `reason` string, which will be written to logs and to the instance status. A terminated instance will stop running as soon as it reaches the next `await` point, or it will terminate immediately if it is already on an `await`.

```
[FunctionName("TerminateInstance")]
public static Task Run(
    [OrchestrationClient] DurableOrchestrationClient client,
    [ManualTrigger] string instanceId)
{
    string reason = "It was time to be done.";
    return client.TerminateAsync(instanceId, reason);
}
```

NOTE

Instance termination is currently only supported for C# orchestrator functions.

Sending events to instances

Event notifications can be sent to running instances using the [RaiseEventAsync](#) method of the [DurableOrchestrationClient](#) class. Instances that can handle these events are those that are awaiting a call to [WaitForExternalEvent](#).

The parameters to [RaiseEventAsync](#) are as follows:

- **InstanceId**: The unique ID of the instance.
- **EventName**: The name of the event to send.
- **EventData**: A JSON-serializable payload to send to the instance.

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

[FunctionName("RaiseEvent")]
public static Task Run(
    [OrchestrationClient] DurableOrchestrationClient client,
    [ManualTrigger] string instanceId)
{
    int[] eventData = new int[] { 1, 2, 3 };
    return client.RaiseEventAsync(instanceId, "MyEvent", eventData);
}
```

NOTE

Raising events is currently supported only for C# orchestrator functions.

WARNING

If there is no orchestration instance with the specified *instance ID* or if the instance is not waiting on the specified *event name*, the event message is discarded. For more information about this behavior, see the [GitHub issue](#).

Next steps

Learn how to use the HTTP APIs for instance management

HTTP APIs in Durable Functions (Azure Functions)

10/10/2017 • 6 min to read • [Edit Online](#)

The Durable Task extension exposes a set of HTTP APIs that can be used to perform the following tasks:

- Fetch the status of an orchestration instance.
- Send an event to a waiting orchestration instance.
- Terminate a running orchestration instance.

Each of these HTTP APIs are webhook operations that are handled directly by the Durable Task extension. They are not specific to any function in the function app.

NOTE

These operations can also be invoked directly using the instance management APIs on the [DurableOrchestrationClient](#) class. For more information, see [Instance Management](#).

HTTP API URL discovery

The [DurableOrchestrationClient](#) class exposes a [CreateCheckStatusResponse](#) API that can be used to generate an HTTP response payload containing links to all the supported operations. Here is an example HTTP-trigger function that demonstrates how to use this API:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Newtonsoft.Json"

using System.Net;
using System.Net.Http.Headers;

public static async Task<HttpResponseMessage> Run(
    HttpRequestMessage req,
    DurableOrchestrationClient starter,
    string functionName,
    TraceWriter log)
{
    // Function input comes from the request content.
    dynamic eventData = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName, eventData);

    log.Info($"Started orchestration with ID = '{instanceId}'.");

    var res = starter.CreateCheckStatusResponse(req, instanceId);
    res.Headers.RetryAfter = new RetryConditionHeaderValue(TimeSpan.FromSeconds(10));
    return res;
}
```

This example function produces the following JSON response data. The data type of all fields is `string`.

FIELD	DESCRIPTION
<code>id</code>	The ID of the orchestration instance.
<code>statusQueryGetUri</code>	The status URL of the orchestration instance.

FIELD	DESCRIPTION
sendEventPostUri	The "raise event" URL of the orchestration instance.
terminatePostUri	The "terminate" URL of the orchestration instance.

Here is an example response:

```

HTTP/1.1 202 Accepted
Content-Length: 923
Content-Type: application/json; charset=utf-8
Location:
https://{{host}}/webhookextensions/handler/DurableTaskExtension/instances/34ce9a28a6834d8492ce6a295f1a80e2?
taskHub=DurableFunctionsHub&connection=Storage&code=XXX

{
  "id": "34ce9a28a6834d8492ce6a295f1a80e2",

  "statusQueryGetUri": "https://{{host}}/webhookextensions/handler/DurableTaskExtension/instances/34ce9a28a6834d8492ce6a295f1a80e2?taskHub=DurableFunctionsHub&connection=Storage&code=XXX",

  "sendEventPostUri": "https://{{host}}/webhookextensions/handler/DurableTaskExtension/instances/34ce9a28a6834d8492ce6a295f1a80e2/raiseEvent/{eventName}?taskHub=DurableFunctionsHub&connection=Storage&code=XXX",

  "terminatePostUri": "https://{{host}}/webhookextensions/handler/DurableTaskExtension/instances/34ce9a28a6834d8492ce6a295f1a80e2/terminate?reason={text}&taskHub=DurableFunctionsHub&connection=Storage&code=XXX"
}

```

NOTE

The format of the webhook URLs may differ depending on which version of the Azure Functions host you are running. The above example is for the Azure Functions 2.0 host.

Async operation tracking

The HTTP response mentioned previously is designed to help implementing long-running HTTP async APIs with Durable Functions. This is sometimes referred to as the *Polling Consumer Pattern*. The client/server flow works as follows:

1. The client issues an HTTP request to start a long running process, such as an orchestrator function.
2. The target HTTP trigger returns an HTTP 202 response with a `Location` header with the `statusQueryGetUri` value.
3. The client polls the URL in the `Location` header. It continues to see HTTP 202 responses with a `Location` header.
4. When the instance completes (or fails), the endpoint in the `Location` header returns HTTP 200.

This protocol allows coordinating long-running processes with external clients or services that support polling an HTTP endpoint and following the `Location` header. The fundamental pieces are already built into the Durable Functions HTTP APIs.

NOTE

By default, all HTTP-based actions provided by [Azure Logic Apps](#) support the standard asynchronous operation pattern. This makes it possible to embed a long-running durable function as part of a Logic Apps workflow. More details on Logic Apps support for asynchronous HTTP patterns can be found in the [Azure Logic Apps workflow actions and triggers documentation](#).

HTTP API reference

All HTTP APIs implemented by the extension take the following parameters. The data type of all parameters is `string`.

PARAMETER	PARAMETER TYPE	DESCRIPTION
instanceId	URL	The ID of the orchestration instance.
taskHub	Query string	The name of the task hub . If not specified, the current function app's task hub name is assumed.
connection	Query string	The name of the connection string for the storage account. If not specified, the default connection string for the function app is assumed.
systemKey	Query string	The authorization key required to invoke the API.

`systemKey` is an authorization key auto-generated by the Azure Functions host. It specifically grants access to the Durable Task extension APIs and can be managed the same way as [other authorization keys](#). The simplest way to discover the `systemKey` value is by using the `CreateCheckStatusResponse` API mentioned previously.

The next few sections cover the specific HTTP APIs supported by the extension and provide examples of how they can be used.

Get instance status

Gets the status of a specified orchestration instance.

Request

For Functions 1.0, the request format is as follows:

```
GET /admin/extensions/DurableTaskExtension/instances/{instanceId}?taskHub={taskHub}&connection={connection}&code={systemKey}
```

The Functions 2.0 format has all the same parameters but has a slightly different URL prefix:

```
GET /webhookextensions/handler/DurableTaskExtension/instances/{instanceId}?taskHub={taskHub}&connection={connection}&code={systemKey}
```

Response

Several possible status code values can be returned.

- **HTTP 200 (OK)**: The specified instance is in a completed state.
- **HTTP 202 (Accepted)**: The specified instance is in progress.

- **HTTP 400 (Bad Request)**: The specified instance failed or was terminated.
- **HTTP 404 (Not Found)**: The specified instance doesn't exist or has not started running.

The response payload for the **HTTP 200** and **HTTP 202** cases is a JSON object with the following fields.

FIELD	DATA TYPE	DESCRIPTION
runtimeStatus	string	The runtime status of the instance. Values include <i>Running</i> , <i>Pending</i> , <i>Failed</i> , <i>Canceled</i> , <i>Terminated</i> , <i>Completed</i> .
input	JSON	The JSON data used to initialize the instance.
output	JSON	The JSON output of the instance. This field is <code>null</code> if the instance is not in a completed state.
createdTime	string	The time at which the instance was created. Uses ISO 8601 extended notation.
lastUpdatedTime	string	The time at which the instance last persisted. Uses ISO 8601 extended notation.

Here is an example response payload (formatted for readability):

```
{
  "runtimeStatus": "Completed",
  "input": null,
  "output": [
    "Hello Tokyo!",
    "Hello Seattle!",
    "Hello London!"
  ],
  "createdTime": "2017-10-06T18:30:24Z",
  "lastUpdatedTime": "2017-10-06T18:30:30Z"
}
```

The **HTTP 202** response also includes a **Location** response header that references the same URL as the `statusQueryGetUri` field mentioned previously.

Raise event

Sends an event notification message to a running orchestration instance.

Request

For Functions 1.0, the request format is as follows:

```
POST /admin/extensions/DurableTaskExtension/instances/{instanceId}/raiseEvent/{eventName}?
taskHub=DurableFunctionsHub&connection={connection}&code={systemKey}
```

The Functions 2.0 format has all the same parameters but has a slightly different URL prefix:

```
POST /webhookextensions/handler/DurableTaskExtension/instances/{instanceId}/raiseEvent/{eventName}?taskHub=DurableFunctionsHub&connection={connection}&code={systemKey}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters.

FIELD	PARAMETER TYPE	DATA TYPE	DESCRIPTION
eventName	URL	string	The name of the event that the target orchestration instance is waiting on.
{content}	Request content	JSON	The JSON-formatted event payload.

Response

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The raised event was accepted for processing.
- **HTTP 400 (Bad request)**: The request content was not of type `application/json` or was not valid JSON.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed or failed and cannot process any raised events.

Here is an example request that sends the JSON string `"incr"` to an instance waiting for an event named **operation** (taken from the [Counter](#) sample):

```
POST /admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a/raiseEvent/operation?taskHub=DurableFunctionsHub&connection=Storage&code=XXX
Content-Type: application/json
Content-Length: 6

"incr"
```

The responses for this API do not contain any content.

Terminate instance

Terminates a running orchestration instance.

Request

For Functions 1.0, the request format is as follows:

```
DELETE /admin/extensions/DurableTaskExtension/instances/{instanceId}/terminate?reason={reason}&taskHub={taskHub}&connection={connection}&code={systemKey}
```

The Functions 2.0 format has all the same parameters but has a slightly different URL prefix:

```
DELETE /webhookextensions/handler/DurableTaskExtension/instances/{instanceId}/terminate?reason={reason}&taskHub={taskHub}&connection={connection}&code={systemKey}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameter.

FIELD	PARAMETER TYPE	DATA TYPE	DESCRIPTION
reason	Query string	string	Optional. The reason for terminating the orchestration instance.

Response

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The terminate request was accepted for processing.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed or failed.

Here is an example request that terminates a running instance and specifies a reason of **buggy**:

```
DELETE /admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a/terminate?
reason=buggy&taskHub=DurableFunctionsHub&connection=Storage&code=XXX
```

The responses for this API do not contain any content.

Next steps

[Learn how to handle errors](#)

Handling errors in Durable Functions (Azure Functions)

10/10/2017 • 2 min to read • [Edit Online](#)

Durable Function orchestrations are implemented in code and can use the error-handling capabilities of the programming language. With this in mind, there really aren't any new concepts you need to learn about when incorporating error handling and compensation into your orchestrations. However, there are a few behaviors that you should be aware of.

Errors in activity functions

Any exception that is thrown in an activity function is marshalled back to the orchestrator function and thrown as a `TaskFailedException`. You can write error handling and compensation code that suits your needs in the orchestrator function.

For example, consider the following orchestrator function which transfers funds from one account to another:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static async Task Run(DurableOrchestrationContext context)
{
    var transferDetails = ctx.GetInput<TransferOperation>();

    await context.CallActivityAsync("DebitAccount",
        new
        {
            Account = transferDetails.SourceAccount,
            Amount = transferDetails.Amount
        });

    try
    {
        await context.CallActivityAsync("CreditAccount",
            new
            {
                Account = transferDetails.DestinationAccount,
                Amount = transferDetails.Amount
            });
    }
    catch (Exception)
    {
        // Refund the source account.
        // Another try/catch could be used here based on the needs of the application.
        await context.CallActivityAsync("CreditAccount",
            new
            {
                Account = transferDetails.SourceAccount,
                Amount = transferDetails.Amount
            });
    }
}
```

If the call to the **CreditAccount** function fails for the destination account, the orchestrator function compensates for this by crediting the funds back to the source account.

Automatic retry on failure

When you call activity functions or sub-orchestration functions you can specify an automatic retry policy. The following example attempts to call a function up to 3 times and waits 5 seconds between each retry:

```
public static async Task Run(DurableOrchestrationContext context)
{
    var retryOptions = new RetryOptions(
        firstRetryInterval: TimeSpan.FromSeconds(5),
        maxNumberOfAttempts: 3);

    await ctx.CallActivityWithRetryAsync("FlakyFunction", retryOptions);

    // ...
}
```

The `CallActivityWithRetryAsync` API takes a `RetryOptions` parameter. Sub-orchestration calls using the `CallSubOrchestratorWithRetryAsync` API can use these same retry policies.

There are several options for customizing the automatic retry policy. They include the following:

- **Max number of attempts:** The maximum number of retry attempts.
- **First retry interval:** The amount of time to wait before the first retry attempt.
- **Backoff coefficient:** The coefficient used to determine rate of increase of backoff. Defaults to 1.
- **Max retry interval:** The maximum amount of time to wait in between retry attempts.
- **Retry timeout:** The maximum amount of time to spend doing retries. The default behavior is to retry indefinitely.
- **Custom:** A user-defined callback can be specified which determines whether or not a function call should be retried.

Function timeouts

You might want to abandon a function call within an orchestrator function if it is taking too long to complete. The proper way to do this today is by creating a [durable timer](#) using `context.CreateTimer` in conjunction with `Task.WhenAny`, as in the following example:

```
public static async Task<bool> Run(DurableOrchestrationContext context)
{
    TimeSpan timeout = TimeSpan.FromSeconds(30);
    DateTime deadline = context.CurrentUtcDateTime.Add(timeout);

    using (var cts = new CancellationTokenSource())
    {
        Task activityTask = context.CallActivityAsync("FlakyFunction");
        Task timeoutTask = context.CreateTimer(deadline, cts.Token);

        Task winner = await Task.WhenAny(activityTask, timeoutTask);
        if (winner == activityTask)
        {
            // success case
            cts.Cancel();
            return true;
        }
        else
        {
            // timeout case
            return false;
        }
    }
}
```

Unhandled exceptions

If an orchestrator function fails with an unhandled exception, the details of the exception are logged and the instance completes with a `Failed` status.

Next steps

[Learn how to diagnose problems](#)

Diagnostics in Durable Functions (Azure Functions)

10/10/2017 • 6 min to read • [Edit Online](#)

There are several options for diagnosing issues with [Durable Functions](#). Some of these options are the same for regular functions and some of them are unique to Durable Functions.

Application Insights

[Application Insights](#) is the recommended way to do diagnostics and monitoring in Azure Functions. The same applies to Durable Functions. For an overview of how to leverage Application Insights in your function app, see [Monitor Azure Functions](#).

The Azure Functions Durable Extension also emits *tracking events* which allow you to trace the end-to-end execution of an orchestration. These can be found and queried using the [Application Insights Analytics](#) tool in the Azure portal.

Tracking data

Each lifecycle event of an orchestration instance causes a tracking event to be written to the **traces** collection in Application Insights. This event contains a **customDimensions** payload with several fields. Field names are all prepended with `prop_`.

- **hubName**: The name of the task hub in which your orchestrations are running.
- **appName**: The name of the function app. This is useful when you have multiple function apps sharing the same Application Insights instance.
- **slotName**: The [deployment slot](#) in which the current function app is running. This is useful when you leverage deployment slots to version your orchestrations.
- **functionName**: The name of the orchestrator or activity function.
- **functionType**: The type of the function, such as **Orchestrator** or **Activity**.
- **instanceId**: The unique ID of the orchestration instance.
- **state**: The lifecycle execution state of the instance. Valid values include:
 - **Scheduled**: The function was scheduled for execution but hasn't started running yet.
 - **Started**: The function has started running but has not yet awaited or completed.
 - **Awaited**: The orchestrator has scheduled some work and is waiting for it to complete.
 - **Listening**: The orchestrator is listening for an external event notification.
 - **Completed**: The function has completed successfully.
 - **Failed**: The function failed with an error.
- **reason**: Additional data associated with the tracking event. For example, if an instance is waiting for an external event notification, this field indicates the name of the event it is waiting for. If a function has failed, this will contain the error details.
- **isReplay**: Boolean value indicating whether the tracking event is for replayed execution.
- **extensionVersion**: The version of the Durable Task extension. This is especially important data when reporting possible bugs in the extension. Long-running instances may report multiple versions if an update occurs while it is running.

The verbosity of tracking data emitted to Application Insights can be configured in the `logger` section of the `host.json` file.

```
{  
    "logger": {  
        "categoryFilter": {  
            "categoryLevels": {  
                "Host.Triggers.DurableTask": "Information"  
            }  
        }  
    }  
}
```

By default, all tracking events are emitted. The volume of data can be reduced by setting `Host.Triggers.DurableTask` to `"Warning"` or `"Error"` in which case tracking events will only be emitted for exceptional situations.

WARNING

By default, Application Insights telemetry is sampled by the Azure Functions runtime to avoid emitting data too frequently. This can cause tracking information to be lost when many lifecycle events occur in a short period of time. The [Azure Functions Monitoring](#) article explains how to configure this behavior.

Single instance query

The following query shows historical tracking data for a single instance of the [Hello Sequence](#) function orchestration. It's written using the [Application Insights Query Language \(AIQL\)](#). It filters out replay execution so that only the *logical* execution path is shown.

```
let targetInstanceId = "bf71335b26564016a93860491aa50c7f";  
let start = datetime(2017-09-29T00:00:00);  
traces  
| where timestamp > start and timestamp < start + 30m  
| where customDimensions.Category == "Host.Triggers.DurableTask"  
| extend functionName = customDimensions["prop__functionName"]  
| extend instanceId = customDimensions["prop__instanceId"]  
| extend state = customDimensions["prop__state"]  
| extend isReplay = tobool(tolower(customDimensions["prop__isReplay"]))  
| where isReplay == false  
| where instanceId == targetInstanceId  
| project timestamp, functionName, state, instanceId, appName = cloud_RoleName
```

The result is a list of tracking events that show the execution path of the orchestration, including any activity functions.

timestamp [UTC]	functionName	state	instanceId	appName
2017-09-29T00:20:16.270	E1_HelloSequence	Scheduled	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.536	E1_HelloSequence	Started	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.567	E1_SayHello	Scheduled	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.567	E1_HelloSequence	Awaited	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.754	E1_SayHello	Started	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.801	E1_SayHello	Completed	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.911	E1_SayHello	Scheduled	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.911	E1_HelloSequence	Awaited	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.942	E1_SayHello	Completed	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:16.942	E1_SayHello	Started	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:17.005	E1_HelloSequence	Awaited	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:17.005	E1_SayHello	Scheduled	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:17.036	E1_SayHello	Started	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:17.051	E1_SayHello	Completed	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled
2017-09-29T00:20:17.114	E1_HelloSequence	Completed	bf71335b26564016a93860491aa50c7f	durablefunctionssample-compiled

NOTE

Some of these tracking events may be out of order due to the lack of precision in the `timestamp` column. This is being tracked in GitHub as [issue #71](#).

Instance summary query

The following query displays the status of all orchestration instances that were run in a specified time range.

```
let start = datetime(2017-09-30T04:30:00);
traces
| where timestamp > start and timestamp < start + 1h
| where customDimensions.Category == "Host.Triggers.DurableTask"
| extend functionName = tostring(customDimensions["prop__functionName"])
| extend instanceId = tostring(customDimensions["prop__instanceId"])
| extend state = tostring(customDimensions["prop__state"])
| extend isReplay = tobool(tolower(customDimensions["prop__isReplay"]))
| extend output = tostring(customDimensions["prop__output"])
| where isReplay == false
| summarize arg_max(timestamp, *) by instanceId
| project timestamp, instanceId, functionName, state, output, appName = cloud_RoleName
| order by timestamp asc
```

The result is a list of instance IDs and their current runtime status.

timestamp [UTC]	instanceId	functionName	state	output	appName
2017-09-30T04:59:30.146	902deae5c5384289ada7cb17ae25e280	E1_HelloSequence	Completed	(49 bytes)	durablefunctionssample-compiled
2017-09-30T05:00:00.737	ffbb064e035d6444b882160870df98b57	E2_BackupSiteContent	Failed		durablefunctionssample-compiled
2017-09-30T05:00:30.250	0d24c02c0e16443ead86a911ff113c27	E1_HelloSequence	Completed	(49 bytes)	durablefunctionssample-compiled
2017-09-30T05:00:30.310	254a8ca78b7049d5acbf62fa760a71b03	E1_HelloSequence	Completed	(49 bytes)	durablefunctionssample-compiled
2017-09-30T05:01:14.878	4652f7a63d024ddc9b95ded383557bfd	E3_Counter	Listening		durablefunctionssample-compiled
2017-09-30T05:02:09.318	dab5b982e89d47c1aaeec54880df10b4	E4_SmsPhoneVerification	Failed		durablefunctionssample-compiled
2017-09-30T05:02:38.120	86b51576ef53459b9b3aca0e5b507b0a	E1_HelloSequence	Completed	(49 bytes)	durablefunctionssample-compiled
2017-09-30T05:04:48.144	6395013d2ecf4ff18433c9745f7731a7	E3_Counter	Started		durablefunctionssample-compiled

Logging

It's important to keep the orchestrator replay behavior in mind when writing logs directly from an orchestrator function. For example, consider the following orchestrator function:

```

public static async Task Run(
    DurableOrchestrationContext ctx,
    TraceWriter log)
{
    log.Info("Calling F1.");
    await ctx.CallActivityAsync("F1");
    log.Info("Calling F2.");
    await ctx.CallActivityAsync("F2");
    log.Info("Calling F3.");
    await ctx.CallActivityAsync("F3");
    log.Info("Done!");
}

```

The resulting log data is going to look something like the following:

```

Calling F1.
Calling F1.
Calling F2.
Calling F1.
Calling F2.
Calling F3.
Calling F1.
Calling F2.
Calling F3.
Done!

```

NOTE

Remember that while the logs claim to be calling F1, F2, and F3, those functions are only *actually* called the first time they are encountered. Subsequent calls that happen during replay are skipped and the outputs are replayed to the orchestrator logic.

If you want to only log on non-replay execution, you can write a conditional expression to log only if `IsReplaying` is `false`. Consider the example above, but this time with replay checks.

```

public static async Task Run(
    DurableOrchestrationContext ctx,
    TraceWriter log)
{
    if (!ctx.IsReplaying) log.Info("Calling F1.");
    await ctx.CallActivityAsync("F1");
    if (!ctx.IsReplaying) log.Info("Calling F2.");
    await ctx.CallActivityAsync("F2");
    if (!ctx.IsReplaying) log.Info("Calling F3.");
    await ctx.CallActivityAsync("F3");
    log.Info("Done!");
}

```

With this change, the log output is as follows:

```

Calling F1.
Calling F2.
Calling F3.
Done!

```

Debugging

Azure Functions supports debugging function code directly, and that same support carries forward to Durable Functions, whether running in Azure or locally. However, there are a few behaviors to be aware of when debugging:

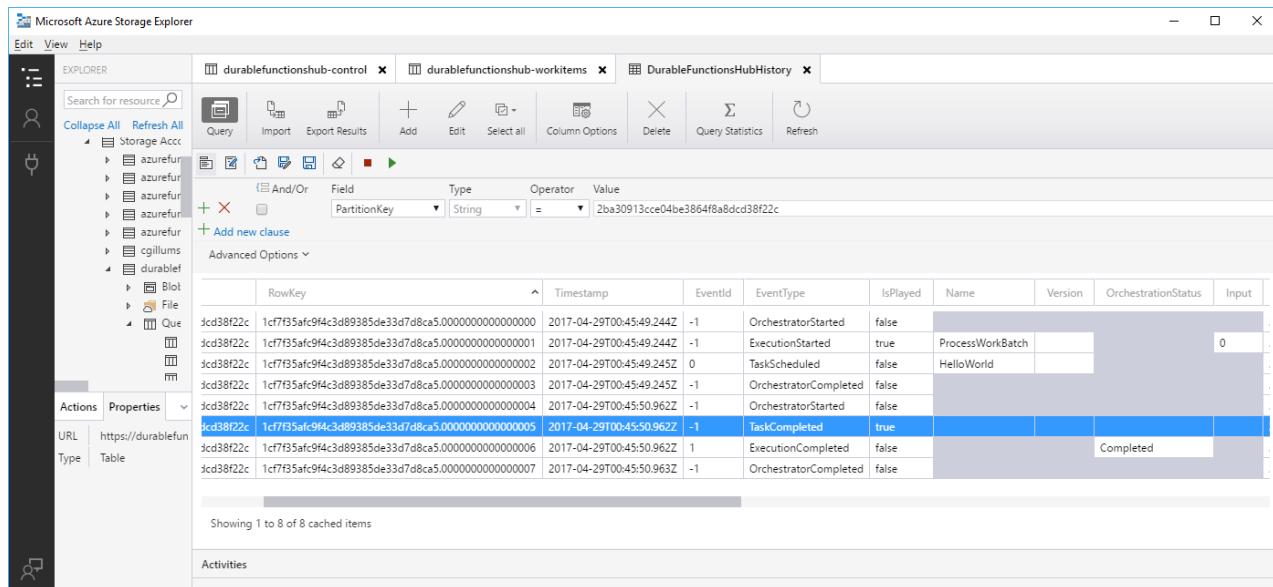
- **Replay:** Orchestrator functions regularly replay when new inputs are received. This means a single *logical* execution of an orchestrator function can result in hitting the same breakpoint multiple times, especially if it is set early in the function code.
- **Await:** Whenever an `await` is encountered, it yields control back to the Durable Task Framework dispatcher. If this is the first time a particular `await` has been encountered, the associated task is *never* resumed. Because the task never resumes, stepping over the await (F10 in Visual Studio) is not actually possible. Stepping over only works when a task is being replayed.
- **Messaging timeouts:** Durable Functions internally uses queue messages to drive execution of both orchestrator functions and activity functions. In a multi-VM environment, breaking into the debugging for extended periods of time could cause another VM to pick up the message, resulting in duplicate execution. This behavior exists for regular queue-trigger functions as well, but is important to point out in this context since the queues are an implementation detail.

TIP

When setting breakpoints, if you want to only break on non-replay execution, you can set a conditional breakpoint that breaks only if `IsReplaying` is `false`.

Storage

By default, Durable Functions stores state in Azure Storage. This means you can inspect the state of your orchestrations using tools such as [Microsoft Azure Storage Explorer](#).



The screenshot shows the Microsoft Azure Storage Explorer interface. On the left, the 'EXPLORER' sidebar lists storage accounts and tables, with 'durablefunctionshub-control', 'durablefunctionshub-workitems', and 'DurableFunctionsHubHistory' selected. The main pane displays the 'DurableFunctionsHubHistory' table with the following data:

	RowKey	Timestamp	EventId	EventType	IsPlayed	Name	Version	OrchestrationStatus	Input
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000000	2017-04-29T00:45:49.244Z	-1	OrchestratorStarted	false				
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000001	2017-04-29T00:45:49.244Z	-1	ExecutionStarted	true	ProcessWorkBatch			0
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000002	2017-04-29T00:45:49.245Z	0	TaskScheduled	false	HelloWorld			
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000003	2017-04-29T00:45:49.245Z	-1	OrchestratorCompleted	false				
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000004	2017-04-29T00:45:50.962Z	-1	OrchestratorStarted	false				
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000005	2017-04-29T00:45:50.962Z	-1	TaskCompleted	true				
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000006	2017-04-29T00:45:50.962Z	1	ExecutionCompleted	false				Completed
jcd38f22c	1cf7f35acf9f4c3d89385de33d7d8ca5.0000000000000007	2017-04-29T00:45:50.963Z	-1	OrchestratorCompleted	false				

Showing 1 to 8 of 8 cached items

This is useful for debugging because you see exactly what state an orchestration may be in. Messages in the queues can also be examined to learn what work is pending (or stuck in some cases).

WARNING

While it's convenient to see execution history in table storage, avoid taking any dependency on this table. It may change as the Durable Functions extension evolves.

Next steps

[Learn how to use durable timers](#)

Timers in Durable Functions (Azure Functions)

1/8/2018 • 2 min to read • [Edit Online](#)

[Durable Functions](#) provides *durable timers* for use in orchestrator functions to implement delays or to set up timeouts on async actions. Durable timers should be used in orchestrator functions instead of `Thread.Sleep` or `Task.Delay`.

You create a durable timer by calling the `CreateTimer` method in [DurableOrchestrationContext](#). The method returns a task that resumes on a specified date and time.

Timer limitations

When you create a timer that expires at 4:30 pm, the underlying Durable Task Framework enqueues a message which becomes visible only at 4:30 pm. When running in the Azure Functions consumption plan, the newly visible timer message will ensure that the function app gets activated on an appropriate VM.

WARNING

- Durable timers cannot last longer than 7 days due to limitations in Azure Storage. We are working on a [feature request to extend timers beyond 7 days](#).
- Always use `CurrentUtcDateTime` instead of `DateTime.UtcNow` as shown in the examples below when computing a relative deadline of a durable timer.

Usage for delay

The following example illustrates how to use durable timers for delaying execution. The example is issuing a billing notification every day for ten days.

```
[FunctionName("BillingIssuer")]
public static async Task Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    for (int i = 0; i < 10; i++)
    {
        DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.FromDays(1));
        await context.CreateTimer(deadline, CancellationToken.None);
        await context.CallFunctionAsync("SendBillingEvent");
    }
}
```

WARNING

Avoid infinite loops in orchestrator functions. For information about how to safely and efficiently implement infinite loop scenarios, see [Eternal Orchestrations](#).

Usage for timeout

This example illustrates how to use durable timers to implement timeouts.

```

[FunctionName("TryGetQuote")]
public static async Task<bool> Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    TimeSpan timeout = TimeSpan.FromSeconds(30);
    DateTime deadline = context.CurrentUtcDateTime.Add(timeout);

    using (var cts = new CancellationTokenSource())
    {
        Task activityTask = context.CallFunctionAsync("GetQuote");
        Task timeoutTask = context.CreateTimer(deadline, cts.Token);

        Task winner = await Task.WhenAny(activityTask, timeoutTask);
        if (winner == activityTask)
        {
            // success case
            cts.Cancel();
            return true;
        }
        else
        {
            // timeout case
            return false;
        }
    }
}

```

WARNING

Use a `CancellationTokenSource` to cancel a durable timer if your code will not wait for it to complete. The Durable Task Framework will not change an orchestration's status to "completed" until all outstanding tasks are completed or cancelled.

This mechanism does not actually terminate in-progress activity function execution. Rather, it simply allows the orchestrator function to ignore the result and move on. If your function app uses the Consumption plan, you will still be billed for any time and memory consumed by the abandoned activity function. By default, functions running in the Consumption plan have a timeout of five minutes. If this limit is exceeded, the Azure Functions host is recycled to stop all execution and prevent a runaway billing situation. The [function timeout is configurable](#).

For a more in-depth example of how to implement timeouts in orchestrator functions, see the [Human Interaction & Timeouts - Phone Verification](#) walkthrough.

Next steps

[Learn how to raise and handle external events](#)

Handling external events in Durable Functions (Azure Functions)

1/8/2018 • 2 min to read • [Edit Online](#)

Orchestrator functions have the ability to wait and listen for external events. This feature of [Durable Functions](#) is often useful for handling human interaction or other external triggers.

Wait for events

The [WaitForExternalEvent](#) method allows an orchestrator function to asynchronously wait and listen for an external event. The listening orchestrator function declares the *name* of the event and the *shape of the data* it expects to receive.

```
[FunctionName("BudgetApproval")]
public static async Task Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    bool approved = await context.WaitForExternalEvent<bool>("Approval");
    if (approved)
    {
        // approval granted - do the approved action
    }
    else
    {
        // approval denied - send a notification
    }
}
```

The preceding example listens for a specific single event and takes action when it's received.

You can listen for multiple events concurrently, like in the following example, which waits for one of three possible event notifications.

```
[FunctionName("Select")]
public static async Task Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    var event1 = context.WaitForExternalEvent<float>("Event1");
    var event2 = context.WaitForExternalEvent<bool>("Event2");
    var event3 = context.WaitForExternalEvent<int>("Event3");

    var winner = await Task.WhenAny(event1, event2, event3);
    if (winner == event1)
    {
        // ...
    }
    else if (winner == event2)
    {
        // ...
    }
    else if (winner == event3)
    {
        // ...
    }
}
```

The previous example listens for *any* of multiple events. It's also possible to wait for *all* events.

```
[FunctionName("NewBuildingPermit")]
public static async Task Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    string applicationId = context.GetInput<string>();

    var gate1 = context.WaitForExternalEvent("CityPlanningApproval");
    var gate2 = context.WaitForExternalEvent("FireDeptApproval");
    var gate3 = context.WaitForExternalEvent("BuildingDeptApproval");

    // all three departments must grant approval before a permit can be issued
    await Task.WhenAll(gate1, gate2, gate3);

    await context.CallActivityAsync("IssueBuildingPermit", applicationId);
}
```

`WaitForExternalEvent` waits indefinitely for some input. The function app can be safely unloaded while waiting. If and when an event arrives for this orchestration instance, it is awakened automatically and immediately processes the event.

NOTE

If your function app uses the Consumption Plan, no billing charges are incurred while an orchestrator function is awaiting a task from `WaitForExternalEvent`, no matter how long it waits.

If the event payload cannot be converted into the expected type `T`, an exception is thrown.

Send events

The `RaiseEventAsync` method of the `DurableOrchestrationClient` class sends the events that `WaitForExternalEvent` waits for. The `RaiseEventAsync` method takes `eventName` and `eventData` as parameters. The event data must be JSON-serializable.

Below is an example queue-triggered function that sends an "Approval" event to an orchestrator function instance. The orchestration instance ID comes from the body of the queue message.

```
[FunctionName("ApprovalQueueProcessor")]
public static async Task Run(
    [QueueTrigger("approval-queue")] string instanceId,
    [OrchestrationClient] DurableOrchestrationClient client)
{
    await client.RaiseEventAsync(instanceId, "Approval", true);
}
```

Internally, `RaiseEventAsync` enqueues a message that gets picked up by the waiting orchestrator function.

WARNING

If there is no orchestration instance with the specified *instance ID* or if the instance is not waiting on the specified *event name*, the event message is discarded. For more information about this behavior, see the [GitHub issue](#).

Next steps

[Learn how to set up eternal orchestrations](#)

[Run a sample that waits for external events](#)

[Run a sample that waits for human interaction](#)

Eternal orchestrations in Durable Functions (Azure Functions)

1/8/2018 • 2 min to read • [Edit Online](#)

Eternal orchestrations are orchestrator functions that never end. They are useful when you want to use [Durable Functions](#) for aggregators and any scenario that requires an infinite loop.

Orchestration history

As explained in [Checkpointing and Replay](#), the Durable Task Framework keeps track of the history of each function orchestration. This history grows continuously as long as the orchestrator function continues to schedule new work. If the orchestrator function goes into an infinite loop and continuously schedules work, this history could grow critically large and cause significant performance problems. The *eternal orchestration* concept was designed to mitigate these kinds of problems for applications that need infinite loops.

Resetting and restarting

Instead of using infinite loops, orchestrator functions reset their state by calling the [ContinueAsNew](#) method. This method takes a single JSON-serializable parameter, which becomes the new input for the next orchestrator function generation.

When `ContinueAsNew` is called, the instance enqueues a message to itself before it exits. The message restarts the instance with the new input value. The same instance ID is kept, but the orchestrator function's history is effectively truncated.

NOTE

The Durable Task Framework maintains the same instance ID but internally creates a new *execution ID* for the orchestrator function that gets reset by `ContinueAsNew`. This execution ID is generally not exposed externally, but it may be useful to know about when debugging orchestration execution.

Periodic work example

One use case for eternal orchestrations is code that needs to do periodic work indefinitely.

```
[FunctionName("Periodic_Cleanup_Loop")]
public static async Task Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    await context.CallActivityAsync("DoCleanup");

    // sleep for one hour between cleanups
    DateTime nextCleanup = context.CurrentUtcDateTime.AddHours(1);
    await context.CreateTimer<string>(nextCleanup);

    context.ContinueAsNew(null);
}
```

The difference between this example and a timer-triggered function is that cleanup trigger times here are not based on a schedule. For example, a CRON schedule that executes a function every hour will execute it at 1:00,

2:00, 3:00 etc. and could potentially run into overlap issues. In this example, however, if the cleanup takes 30 minutes, then it will be scheduled at 1:00, 2:30, 4:00, etc. and there is no chance of overlap.

Counter example

Here is a simplified example of a *counter* function that listens for *increment* and *decrement* events eternally.

```
[FunctionName("SimpleCounter")]
public static async Task Run(
    [OrchestrationTrigger] DurableOrchestrationContext context)
{
    int counterState = context.GetInput<int>();

    string operation = await context.WaitForExternalEvent<string>("operation");

    if (operation == "incr")
    {
        counterState++;
    }
    else if (operation == "decr")
    {
        counterState--;
    }

    context.ContinueAsNew(counterState);
}
```

Exit from an eternal orchestration

If an orchestrator function needs to eventually complete, then all you need to do is *not* call `ContinueAsNew` and let the function exit.

If an orchestrator function is in an infinite loop and needs to be stopped, use the `TerminateAsync` method to stop it. For more information, see [Instance Management](#).

Next steps

[Learn how to implement singleton orchestrations](#)

[Run a sample eternal orchestration](#)

Singleton orchestrators in Durable Functions (Azure Functions)

10/13/2017 • 1 min to read • [Edit Online](#)

For background jobs or actor-style orchestrations, you often need to ensure that only one instance of a particular orchestrator runs at a time. This can be done in [Durable Functions](#) by assigning a specific instance ID to an orchestrator when creating it.

Singleton example

The following C# example shows an HTTP-trigger function that creates a singleton background job orchestration. The code ensures that only one instance exists for a specified instance ID.

```
[FunctionName("HttpStartSingle")]
public static async Task<HttpResponseMessage> RunSingle(
    [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}/{instanceId}")] HttpRequestMessage req,
    [OrchestrationClient] DurableOrchestrationClient starter,
    string functionName,
    string instanceId,
    TraceWriter log)
{
    // Check if an instance with the specified ID already exists.
    var existingInstance = await starter.GetStatusAsync(instanceId);
    if (existingInstance == null)
    {
        // An instance with the specified ID doesn't exist, create one.
        dynamic eventData = await req.Content.ReadAsAsync<object>();
        await starter.StartNewAsync(functionName, instanceId, eventData);
        log.Info($"Started orchestration with ID = '{instanceId}'.");
        return starter.CreateCheckStatusResponse(req, instanceId);
    }
    else
    {
        // An instance with the specified ID exists, don't create one.
        return req.CreateErrorResponse(
            HttpStatusCode.Conflict,
            $"An instance with ID '{instanceId}' already exists.");
    }
}
```

By default, instance IDs are randomly generated GUIDs. But in this case, the instance ID is passed in route data from the URL. The code calls [GetStatusAsync](#) to check if an instance having the specified ID is already running. If not, an instance is created with that ID.

The implementation details of the orchestrator function do not actually matter. It could be a regular orchestrator function that starts and completes, or it could be one that runs forever (that is, an [Eternal Orchestration](#)). The important point is that there is only ever one instance running at a time.

Next steps

[Learn how to call sub-orchestrations](#)

[Run a sample singleton](#)

Sub-orchestrations in Durable Functions (Azure Functions)

10/26/2017 • 1 min to read • [Edit Online](#)

In addition to calling activity functions, orchestrator functions can call other orchestrator functions. For example, you can build a larger orchestration out of a library of orchestrator functions. Or you can run multiple instances of an orchestrator function in parallel.

An orchestrator function can call another orchestrator function by calling the [CallSubOrchestratorAsync](#) or the [CallSubOrchestratorWithRetryAsync](#) method. The [Error Handling & Compensation](#) article has more information on automatic retry.

Sub-orchestrator functions behave just like activity functions from the caller's perspective. They can return a value, throw an exception, and can be awaited by the parent orchestrator function.

Example

The following example illustrates an IoT ("Internet of Things") scenario where there are multiple devices that need to be provisioned. There is a particular orchestration that needs to happen for each of the devices, which might look something like the following:

```
public static async Task DeviceProvisioningOrchestration(
    [OrchestrationTrigger] DurableOrchestrationContext ctx)
{
    string deviceId = ctx.GetInput<string>();

    // Step 1: Create an installation package in blob storage and return a SAS URL.
    Uri sasUrl = await ctx.CallActivityAsync<Uri>("CreateInstallationPackage", deviceId);

    // Step 2: Notify the device that the installation package is ready.
    await ctx.CallActivityAsync("SendPackageUrlToDevice", Tuple.Create(deviceId, sasUrl));

    // Step 3: Wait for the device to acknowledge that it has downloaded the new package.
    await ctx.WaitForExternalEvent<bool>("DownloadCompletedAck");

    // Step 4: ...
}
```

This orchestrator function can be used as-is for one-off device provisioning or it can be part of a larger orchestration. In the latter case, the parent orchestrator function can schedule instances of

`DeviceProvisioningOrchestration` using the `CallSubOrchestratorAsync` API.

Here is an example that shows how to run multiple orchestrator functions in parallel.

```
[FunctionName("ProvisionNewDevices")]
public static async Task ProvisionNewDevices(
    [OrchestrationTrigger] DurableOrchestrationContext ctx)
{
    string[] deviceIds = await ctx.CallActivityAsync<string[]>("GetNewDeviceIds");

    // Run multiple device provisioning flows in parallel
    var provisioningTasks = new List<Task>();
    foreach (string deviceId in deviceIds)
    {
        Task provisionTask = ctx.CallSubOrchestratorAsync("DeviceProvisioningOrchestration", deviceId);
        provisioningTasks.Add(provisionTask);
    }

    await Task.WhenAll(provisioningTasks);

    // ...
}
```

Next steps

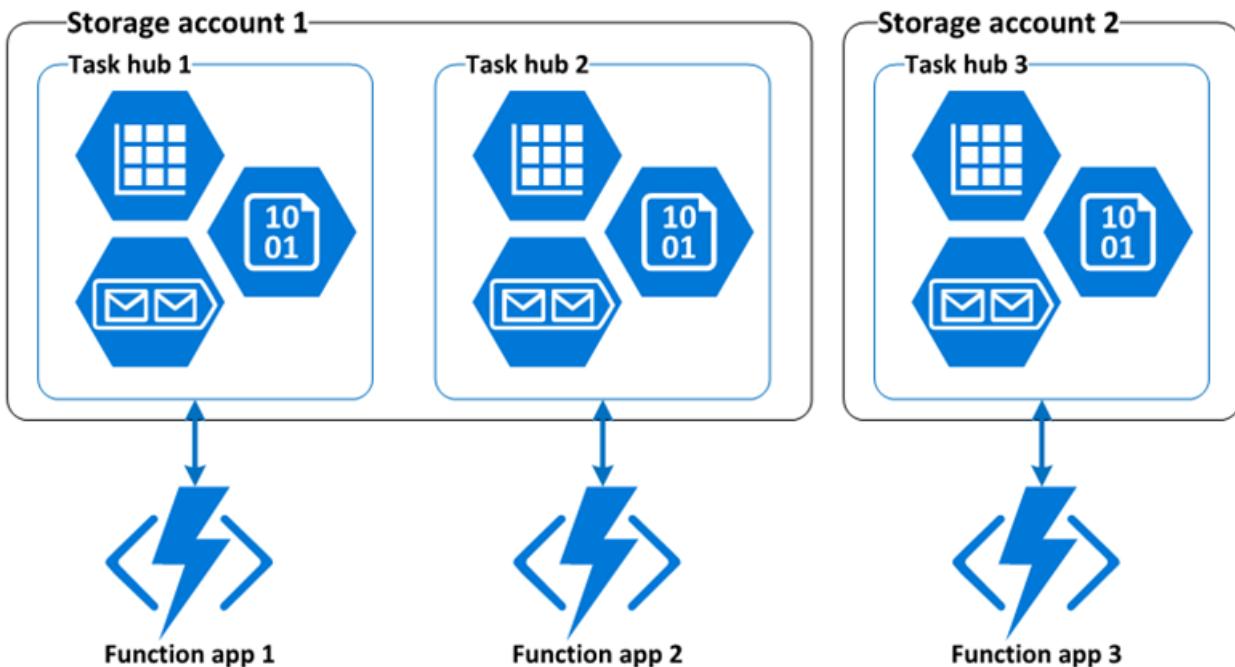
[Learn what task hubs are and how to configure them](#)

Task hubs in Durable Functions (Azure Functions)

10/18/2017 • 1 min to read • [Edit Online](#)

A [task hub](#) in [Durable Functions](#) is a logical container for Azure Storage resources that are used for orchestrations. Orchestrator and activity functions can only interact with each other when they belong to the same task hub.

Each function app has a separate task hub. If multiple function apps share a storage account, the storage account contains multiple task hubs. The following diagram illustrates one task hub per function app in shared and dedicated storage accounts.



Azure Storage resources

A task hub consists of the following storage resources:

- One or more control queues.
- One work-item queue.
- One history table.
- One storage container containing one or more lease blobs.

All of these resources are created automatically in the default Azure Storage account when orchestrator or activity functions run or are scheduled to run. The [Performance and Scale](#) article explains how these resources are used.

Task hub names

Task hubs are identified by a name that is declared in the `host.json` file, as shown in the following example:

```
{  
    "durableTask": {  
        "HubName": "MyTaskHub"  
    }  
}
```

Task hub names must start with a letter and consist of only letters and numbers. If not specified, the default name is **DurableFunctionsHub**.

NOTE

The name is what differentiates one task hub from another when there are multiple task hubs in a shared storage account. If you have multiple function apps sharing a shared storage account, you have to configure different names for each task hub in the `host.json` files.

Next steps

[Learn how to handle versioning](#)

Versioning in Durable Functions (Azure Functions)

10/10/2017 • 4 min to read • [Edit Online](#)

It is inevitable that functions will be added, removed, and changed over the lifetime of an application. [Durable Functions](#) allows chaining functions together in ways that weren't previously possible, and this chaining affects how you can handle versioning.

How to handle breaking changes

There are several examples of breaking changes to be aware of. This article discusses the most common ones. The main theme behind all of them is that both new and existing function orchestrations are impacted by changes to function code.

Changing activity function signatures

A signature change refers to a change in the name, input, or output of a function. If this kind of change is made to an activity function, it could break the orchestrator function that depends on it. If you update the orchestrator function to accommodate this change, you could break existing in-flight instances.

As an example, suppose we have the following function.

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] DurableOrchestrationContext context)
{
    bool result = await context.CallActivityAsync<bool>("Foo");
    await context.CallActivityAsync("Bar", result);
}
```

This simplistic function takes the results of **Foo** and passes it to **Bar**. Let's assume we need to change the return value of **Foo** from `bool` to `int` to support a wider variety of result values. The result looks like this:

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] DurableOrchestrationContext context)
{
    int result = await context.CallActivityAsync<int>("Foo");
    await context.CallActivityAsync("Bar", result);
}
```

This change works fine for all new instances of the orchestrator function but breaks any in-flight instances. For example, consider the case where an orchestration instance calls **Foo**, gets back a boolean value and then checkpoints. If the signature change is deployed at this point, the checkpointed instance will fail immediately when it resumes and replays the call to `context.CallActivityAsync<int>("Foo")`. This is because the result in the history table is `bool` but the new code tries to deserialize it into `int`.

This is just one of many different ways that a signature change can break existing instances. In general, if an orchestrator needs to change the way it calls a function, then the change is likely to be problematic.

Changing orchestrator logic

The other class of versioning problems come from changing the orchestrator function code in a way that confuses the replay logic for in-flight instances.

Consider the following orchestrator function:

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] DurableOrchestrationContext context)
{
    bool result = await context.CallActivityAsync<bool>("Foo");
    await context.CallActivityAsync("Bar", result);
}
```

Now let's assume you want to make a seemingly innocent change to add another function call.

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] DurableOrchestrationContext context)
{
    bool result = await context.CallActivityAsync<bool>("Foo");
    if (result)
    {
        await context.CallActivityAsync("SendNotification");
    }

    await context.CallActivityAsync("Bar", result);
}
```

This change adds a new function call to **SendNotification** between **Foo** and **Bar**. There are no signature changes. The problem arises when an existing instance resumes from the call to **Bar**. During replay, if the original call to **Foo** returned `true`, then the orchestrator replay will call into **SendNotification** which is not in its execution history. As a result, the Durable Task Framework fails with a `NonDeterministicOrchestrationException` because it encountered a call to **SendNotification** when it expected to see a call to **Bar**.

Mitigation strategies

Here are some of the strategies for dealing with versioning challenges:

- Do nothing
- Stop all in-flight instances
- Side-by-side deployments

Do nothing

The easiest way to handle a breaking change is to let in-flight orchestration instances fail. New instances successfully run the changed code.

Whether this is a problem depends on the importance of your in-flight instances. If you are in active development and don't care about in-flight instances, this might be good enough. However, you will need to deal with exceptions and errors in your diagnostics pipeline. If you want to avoid those things, consider the other versioning options.

Stop all in-flight instances

Another option is to stop all in-flight instances. This can be done by clearing the contents of the internal **control-queue** and **workitem-queue** queues. The instances will be forever stuck where they are, but they will not clutter your telemetry with failure messages. This is ideal in rapid prototype development.

WARNING

The details of these queues may change over time, so don't rely on this technique for production workloads.

Side-by-side deployments

The most fail-proof way to ensure that breaking changes are deployed safely is by deploying them side-by-side

with your older versions. This can be done using any of the following techniques:

- Deploy all the updates as entirely new functions (new names).
- Deploy all the updates as a new function app with a different storage account.
- Deploy a new copy of the function app but with an updated `TaskHub` name. This is the recommended technique.

How to change task hub name

The task hub can be configured in the `host.json` file as follows:

```
{  
  "durableTask": {  
    "HubName": "MyTaskHubV2"  
  }  
}
```

The default value is `DurableFunctionsHub`.

All Azure Storage entities are named based on the `HubName` configuration value. By giving the task hub a new name, you ensure that separate queues and history table are created for the new version of your application.

We recommend that you deploy the new version of the function app to a new [Deployment Slot](#). Deployment slots allow you to run multiple copies of your function app side-by-side with only one of them as the active *production* slot. When you are ready to expose the new orchestration logic to your existing infrastructure, it can be as simple as swapping the new version into the production slot.

NOTE

This strategy works best when you use HTTP and webhook triggers for orchestrator functions. For non-HTTP triggers, such as queues or Event Hubs, the trigger definition should derive from an app setting that gets updated as part of the swap operation.

Next steps

[Learn how to handle performance and scale issues](#)

Performance and scale in Durable Functions (Azure Functions)

1/3/2018 • 4 min to read • [Edit Online](#)

To optimize performance and scalability, it's important to understand the unique scaling characteristics of [Durable Functions](#).

To understand the scale behavior, you have to understand some of the details of the underlying Azure Storage provider used by Durable Functions.

History table

The history table is an Azure Storage table that contains the history events for all orchestration instances. As instances run, new rows are added to this table. The partition key of this table is derived from the instance ID of the orchestration. These values are random in most cases, which ensures optimal distribution of internal partitions in Azure Storage.

Internal queue triggers

Orchestrator functions and activity functions are both triggered by internal queues in the function app's default storage account. There are two types of queues in Durable Functions: the **control queue** and the **work-item queue**.

The work-item queue

There is one work-item queue per task hub in Durable Functions. This is a basic queue and behaves similarly to any other `queueTrigger` queue in Azure Functions. This queue is used to trigger stateless *activity functions*. When a Durable Functions application scales out to multiple VMs, these VMs all compete to acquire work from the work-item queue.

Control queue(s)

The *control queue* is more sophisticated than the simpler work-item queue. It's used to trigger the stateful orchestrator functions. Because the orchestrator function instances are stateful singletons, it's not possible to use a competing consumer model to distribute load across VMs. Instead, orchestrator messages are load-balanced across multiple control queues. More details on this in subsequent sections.

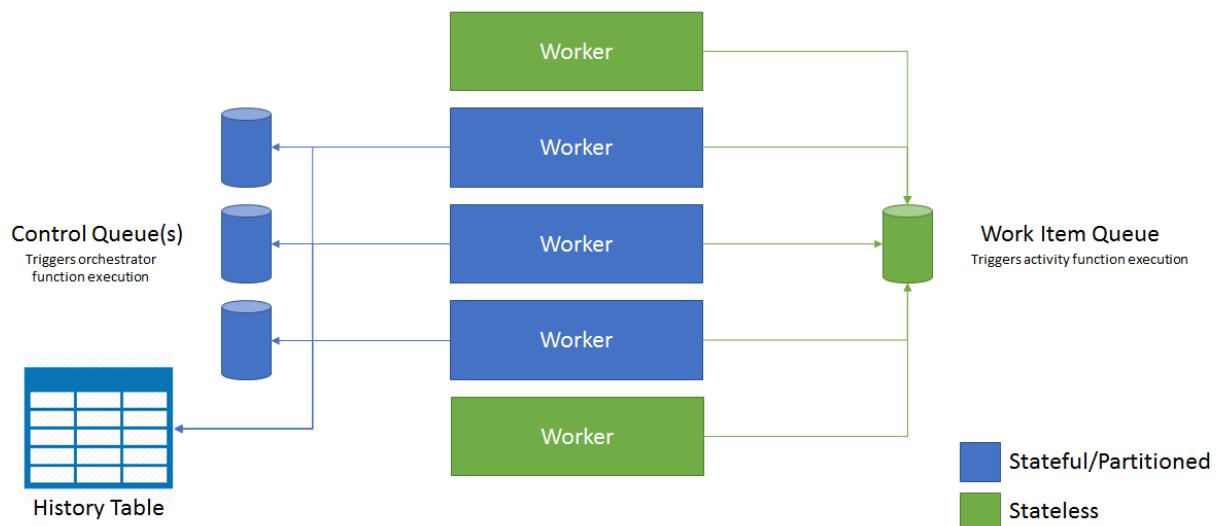
Control queues contain a variety of orchestration lifecycle message types. Examples include [orchestrator control messages](#), activity function *response* messages, and timer messages.

Orchestrator scale-out

Activity functions are stateless and scale out automatically by adding VMs. Orchestrator functions, on the other hand, are *partitioned* across one or more control queues. The number of control queues is fixed and cannot be changed after you start creating load.

When scaling out to multiple function host instances (typically on different VMs), each instance acquires a lock on one of the control queues. This lock ensures that an orchestration instance only runs on a single VM at a time. This means that if a task hub is configured with three control queues, orchestration instances can be load-balanced across as many as three VMs. Additional VMs can be added to increase capacity for activity function execution. But the additional resources will not be used to run orchestrator functions.

The following diagram illustrates how the Azure Functions host interacts with the storage entities in a scaled out environment.



As you can see, all VMs can compete for messages on the work-item queue. However, only three VMs can acquire messages from control queues, and each VM locks a single control queue.

Orchestration instances are distributed across control queue instances by running an internal hash function against the orchestration's instance ID. Instance IDs are auto-generated and random by default that ensures that instances are balanced across all available control queues. The current default number of supported control queue partitions is **4**.

NOTE

It's not currently possible to configure the number of control queue partitions in Azure Functions. [Work to support this configuration option is being tracked](#).

In general, orchestrator functions are intended to be lightweight and should not need a lot of computing power. For this reason, it is not necessary to create a large number of control queue partitions to get great throughput. Rather, most of the heavy work is done in stateless activity functions, which can be scaled out infinitely.

Auto-scale

As with all Azure Functions running in the Consumption plan, Durable Functions support auto-scale via the [Azure Functions scale controller](#). The Scale Controller monitors the length of the work-item queue and each of the control queues, adding or removing VM instances accordingly. If the control queue lengths are increasing over time, the scale controller will continue adding VM instances until it reaches the control queue partition count. If work-item queue lengths are increasing over time, the scale controller will continue adding VM instances until it can match the load, regardless of the control queue partition count.

Thread usage

Orchestrator functions are executed on a single thread. This is required to ensure that orchestrator function execution is deterministic. With this in mind, it's important to never keep the orchestrator function thread unnecessarily busy with tasks such as I/O (which is forbidden for a variety of reasons), blocking, or spinning operations. Any work which may require I/O, blocking, or multiple threads should be moved into activity functions.

Activity functions have all the same behaviors as regular queue-triggered functions. This means they can safely do I/O, execute CPU intensive operations, and use multiple threads. Because activity triggers are stateless, they can freely scale out to an unbounded number of VMs.

Next steps

[Install the Durable Functions extension and samples](#)

Azure Functions developers guide

11/29/2017 • 6 min 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](#) and are familiar with [WebJobs SDK concepts such as triggers, bindings, and the JobHost runtime](#). Azure Functions is based on the WebJobs SDK.

Function code

A *function* is the primary concept in Azure Functions. You write code for a function in a language of your choice and save the code and configuration files in the same folder. The configuration is named `function.json`, which contains JSON configuration data. Various languages are supported, and each one has a slightly different experience optimized to work best for that language.

The `function.json` file defines the function bindings and other configuration settings. The runtime uses this file to determine the events to monitor and how to pass data into and return data from 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
    }
  ]
}
```

Set the `disabled` property to `true` to prevent the function from being executed.

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.

PROPERTY	VALUES/TYPES	COMMENTS
<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 is comprised of one or more individual functions that are managed together by Azure App Service. All of the functions in a function app share the same pricing plan, continuous deployment and runtime version. Functions written in multiple languages can all share the same function app. Think of a function app as a way to organize and collectively manage your functions.

Runtime (script host and web host)

The runtime, or script host, is the underlying WebJobs SDK host that listens for events, gathers and sends data, and ultimately runs your code.

To facilitate HTTP triggers, there is also a web host that is designed to sit in front of the script host in production scenarios. Having two hosts helps to isolate the script host from the front end traffic managed by the web host.

Folder Structure

The code for all the functions in a specific function app is located in a root folder that contains a host configuration file and one or more subfolders. Each subfolder contains the code for a separate function, as in the following example:

```
wwwroot
| - host.json
| - mynodefunction
| | - function.json
| | - index.js
| | - node_modules
| | | - ... packages ...
| | - package.json
| - mycsharpfunction
| | - function.json
| | - run.csx
```

The `host.json` file contains some runtime-specific configurations, and sits in the root folder of the function app. For information about settings that are available, see the [host.json reference](#).

Each function has a folder that contains one or more code files, the `function.json` configuration, and other dependencies.

When setting-up a project for deploying functions to a function app in Azure App Service, you can treat this folder structure as your site code. You can use existing tools like continuous integration and deployment, or custom deployment scripts for doing deploy time package installation or code transpilation.

NOTE

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.

How to update function app files

The function editor built into the Azure portal lets you update the `function.json` file and the code file for a function. To upload or update other files such as `package.json` or `project.json` or dependencies, you have to use other deployment methods.

Function apps are built on App Service, so all the [deployment options available to standard web apps](#) are also available for function apps. Here are some methods you can use to upload or update function app files.

To use App Service Editor

1. In the Azure Functions portal, click **Platform features**.
2. In the **DEVELOPMENT TOOLS** section, click **App Service Editor**.

After App Service Editor loads, you'll see the `host.json` file and function folders under `wwwroot`.

3. Open files to edit them, or drag and drop from your development machine to upload files.

To use the function app's SCM (Kudu) endpoint

1. Navigate to: `https://<function_app_name>.scm.azurewebsites.net`.
2. Click **Debug Console > CMD**.
3. Navigate to `D:\home\site\wwwroot\` to update `host.json` or `D:\home\site\wwwroot\<function_name>` to update a function's files.
4. Drag-and-drop a file you want to upload into the appropriate folder in the file grid. There are two areas in the file grid where you can drop a file. For `.zip` files, a box appears with the label "Drag here to upload and unzip." For other file types, drop in the file grid but outside the "unzip" box.

To use continuous deployment

Follow the instructions in the topic [Continuous deployment for Azure Functions](#).

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 "`~1`" indicates that your Function App will use 1 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 runtime](#)
- [Azure Functions portal](#)
- [Azure Functions templates](#)
- [Azure WebJobs SDK](#)
- [Azure WebJobs SDK Extensions](#)

Bindings

Here is a table of all supported bindings.

The following table shows the bindings that are supported in the two major versions of the Azure Functions runtime.

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
Blob Storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓ ¹	✓	✓	✓
Event Hubs	✓	✓	✓		✓
External File ²	✓			✓	✓
External Table ²	✓			✓	✓
HTTP	✓	✓	✓		✓
Microsoft Graph Excel tables		✓ ¹		✓	✓
Microsoft Graph OneDrive files		✓ ¹		✓	✓
Microsoft Graph Outlook email		✓ ¹			✓
Microsoft Graph Events		✓ ¹	✓	✓	✓
Microsoft Graph Auth tokens		✓ ¹		✓	
Mobile Apps	✓	✓ ¹		✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓ ¹			✓
Service Bus	✓	✓ ¹	✓		✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓ ¹			✓
Webhooks	✓		✓		✓

¹ Must be registered as a binding extension in 2.x. See [Known issues in 2.x](#).

² Experimental — not supported and might be abandoned in the future.

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:

- [Best Practices for Azure Functions](#)
- [Azure Functions C# developer reference](#)
- [Azure Functions F# developer reference](#)
- [Azure Functions NodeJS developer reference](#)
- [Azure Functions triggers and bindings](#)
- [Azure Functions: The Journey](#) on the Azure App Service team blog. A history of how Azure Functions was developed.

Strategies for testing your code in Azure Functions

8/28/2017 • 12 min to read • [Edit Online](#)

This topic demonstrates the various ways to test functions, including using the following general approaches:

- HTTP-based tools, such as cURL, Postman, and even a web browser for web-based triggers
- Azure Storage Explorer, to test Azure Storage-based triggers
- Test tab in the Azure Functions portal
- Timer-triggered function
- Testing application or framework

All these testing methods use an HTTP trigger function that accepts input through either a query string parameter or the request body. You create this function in the first section.

Create a function for testing

For most of this tutorial, we use a slightly modified version of the `HttpTrigger` JavaScript function template that is available when you create a function. If you need help creating a function, review this [tutorial](#). Choose the **HttpTrigger- JavaScript** template when creating the test function in the [Azure portal](#).

The default function template is basically a "hello world" function that echoes back the name from the request body or query string parameter, `name=<your name>`. We'll update the code to also allow you to provide the name and an address as JSON content in the request body. Then the function echoes these back to the client when available.

Update the function with the following code, which we will use for testing:

```

module.exports = function (context, req) {
    context.log("HTTP trigger function processed a request. RequestUri=%s", req.originalUrl);
    context.log("Request Headers = " + JSON.stringify(req.headers));
    var res;

    if (req.query.name || (req.body && req.body.name)) {
        if (typeof req.query.name != "undefined") {
            context.log("Name was provided as a query string param...");
            res = ProcessNewUserInformation(context, req.query.name);
        }
        else {
            context.log("Processing user info from request body...");
            res = ProcessNewUserInformation(context, req.body.name, req.body.address);
        }
    }
    else {
        res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
    context.done(null, res);
};

function ProcessNewUserInformation(context, name, address) {
    context.log("Processing user information...");
    context.log("name = " + name);
    var echoString = "Hello " + name;
    var res;

    if (typeof address != "undefined") {
        echoString += "\n" + "The address you provided is " + address;
        context.log("address = " + address);
    }
    res = {
        // status: 200, /* Defaults to 200 */
        body: echoString
    };
    return res;
}

```

Test a function with tools

Outside the Azure portal, there are various tools that you can use to trigger your functions for testing. These include HTTP testing tools (both UI-based and command line), Azure Storage access tools, and even a simple web browser.

Test with a browser

The web browser is a simple way to trigger functions via HTTP. You can use a browser for GET requests that do not require a body payload, and that use only query string parameters.

To test the function we defined earlier, copy the **Function Url** from the portal. It has the following form:

```
https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code>
```

Append the `name` parameter to the query string. Use an actual name for the `<Enter a name here>` placeholder.

```
https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code>&name=<Enter a name here>
```

Paste the URL into your browser, and you should get a response similar to the following.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Hello Glenn from a browser</string>
```

This example is the Chrome browser, which wraps the returned string in XML. Other browsers display just the string value.

In the portal **Logs** window, output similar to the following is logged in executing the function:

```
2016-03-23T07:34:59 Welcome, you are now connected to log-streaming service.
2016-03-23T07:35:09.195 Function started (Id=61a8c5a9-5e44-4da0-909d-91d293f20445)
2016-03-23T07:35:10.338 Node.js HTTP trigger function processed a request.
RequestUri=https://functionsExample.azurewebsites.net/api/WesmcHttpTriggerNodeJS1?code=XXXXXXXXXX==&name=Glenn
from a browser
2016-03-23T07:35:10.338 Request Headers = {"cache-control":"max-age=0","connection":"Keep-
Alive","accept":"text/html","accept-encoding":"gzip","accept-language":"en-US"}
2016-03-23T07:35:10.338 Name was provided as a query string param.
2016-03-23T07:35:10.338 Processing User Information...
2016-03-23T07:35:10.369 Function completed (Success, Id=61a8c5a9-5e44-4da0-909d-91d293f20445)
```

Test with Postman

The recommended tool to test most of your functions is Postman, which integrates with the Chrome browser. To install Postman, see [Get Postman](#). Postman provides control over many more attributes of an HTTP request.

TIP

Use the HTTP testing tool that you are most comfortable with. Here are some alternatives to Postman:

- [Fiddler](#)
- [Paw](#)

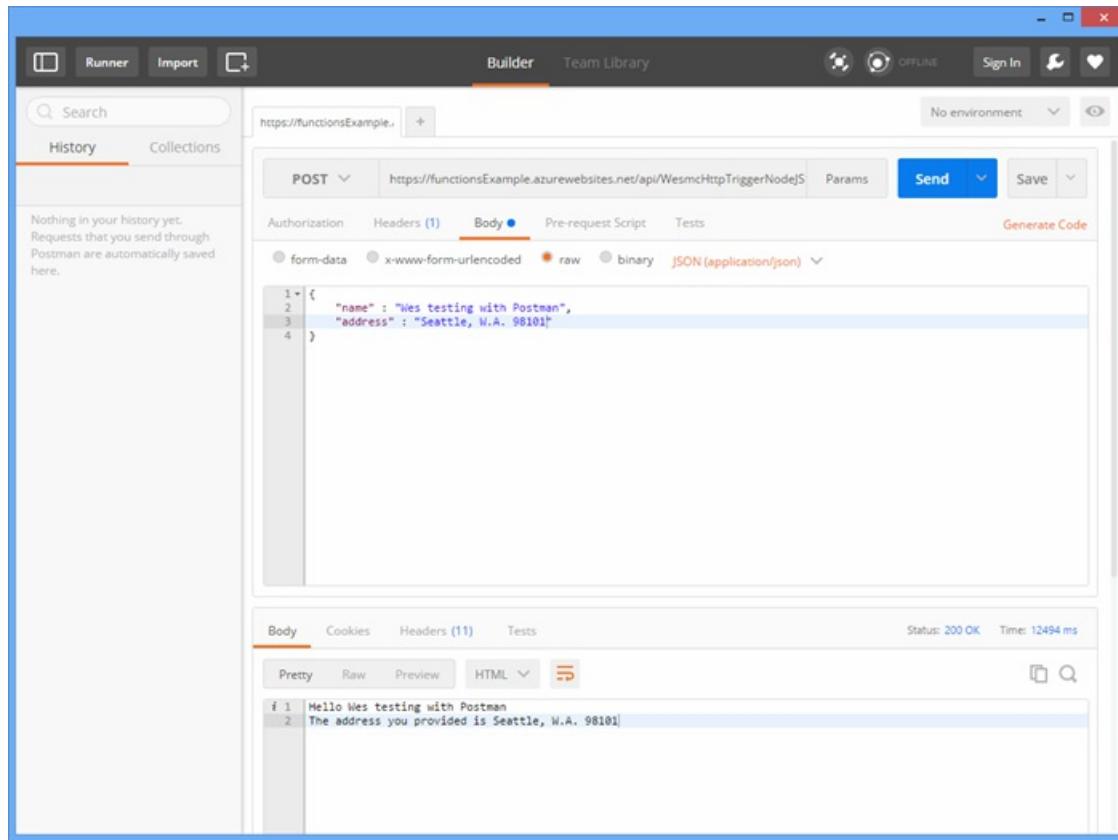
To test the function with a request body in Postman:

1. Start Postman from the **Apps** button in the upper-left corner of a Chrome browser window.
2. Copy your **Function Url**, and paste it into Postman. It includes the access code query string parameter.
3. Change the HTTP method to **POST**.
4. Click **Body > raw**, and add a JSON request body similar to the following:

```
{
  "name" : "Wes testing with Postman",
  "address" : "Seattle, WA 98101"
}
```

5. Click **Send**.

The following image shows testing the simple echo function example in this tutorial.



In the portal **Logs** window, output similar to the following is logged in executing the function:

```
2016-03-23T08:04:51 Welcome, you are now connected to log-streaming service.  
2016-03-23T08:04:57.107 Function started (Id=dc5db8b1-6f1c-4117-b5c4-f6b602d538f7)  
2016-03-23T08:04:57.763 HTTP trigger function processed a request.  
RequestUri=https://functions841def78.azurewebsites.net/api/WesmcHttpTriggerNodeJS1?code=XXXXXXXXXX=  
2016-03-23T08:04:57.763 Request Headers = {"cache-control":"no-cache", "connection":"Keep-Alive", "accept":"/*", "accept-encoding":"gzip", "accept-language":"en-US"}  
2016-03-23T08:04:57.763 Processing user info from request body...  
2016-03-23T08:04:57.763 Processing User Information...  
2016-03-23T08:04:57.763 name = Wes testing with Postman  
2016-03-23T08:04:57.763 address = Seattle, W.A. 98101  
2016-03-23T08:04:57.795 Function completed (Success, Id=dc5db8b1-6f1c-4117-b5c4-f6b602d538f7)
```

Test with cURL from the command line

Often when you're testing software, it's not necessary to look any further than the command line to help debug your application. This is no different with testing functions. Note that the cURL is available by default on Linux-based systems. On Windows, you must first download and install the [cURL tool](#).

To test the function that we defined earlier, copy the **Function URL** from the portal. It has the following form:

```
https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code>
```

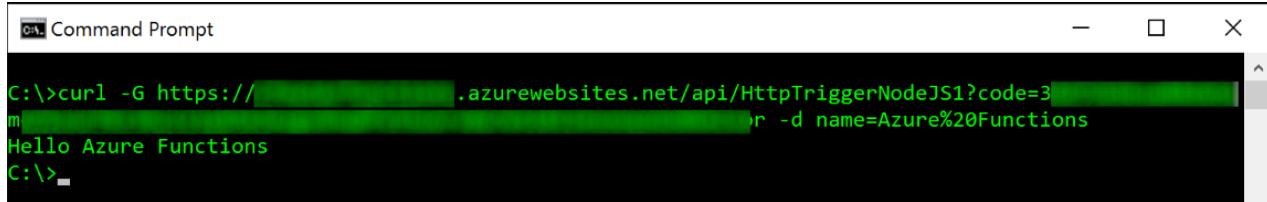
This is the URL for triggering your function. Test this by using the cURL command on the command line to make a GET (`-G` or `--get`) request against the function:

```
curl -G https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code>
```

This particular example requires a query string parameter, which can be passed as Data (`-d`) in the cURL command:

```
curl -G https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code> -d name=<Enter a name here>
```

Run the command, and you see the following output of the function on the command line:



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the command: "curl -G https://<Your Function App>.azurewebsites.net/api/HttpTriggerNodeJS1?code=3...r -d name=Azure%20Functions". The output of the command is displayed below the command line, showing the response from the function: "Hello Azure Functions".

In the portal **Logs** window, output similar to the following is logged in executing the function:

```
2016-04-05T21:55:09  Welcome, you are now connected to log-streaming service.
2016-04-05T21:55:30.738 Function started (Id=ae6955da-29db-401a-b706-482fc1b8f7a)
2016-04-05T21:55:30.738 Node.js HTTP trigger function processed a request.
RequestUri=https://functionsExample.azurewebsites.net/api/HttpTriggerNodeJS1?code=XXXXXXX&name=Azure Functions
2016-04-05T21:55:30.738 Function completed (Success, Id=ae6955da-29db-401a-b706-482fc1b8f7a)
```

Test a blob trigger by using Storage Explorer

You can test a blob trigger function by using [Azure Storage Explorer](#).

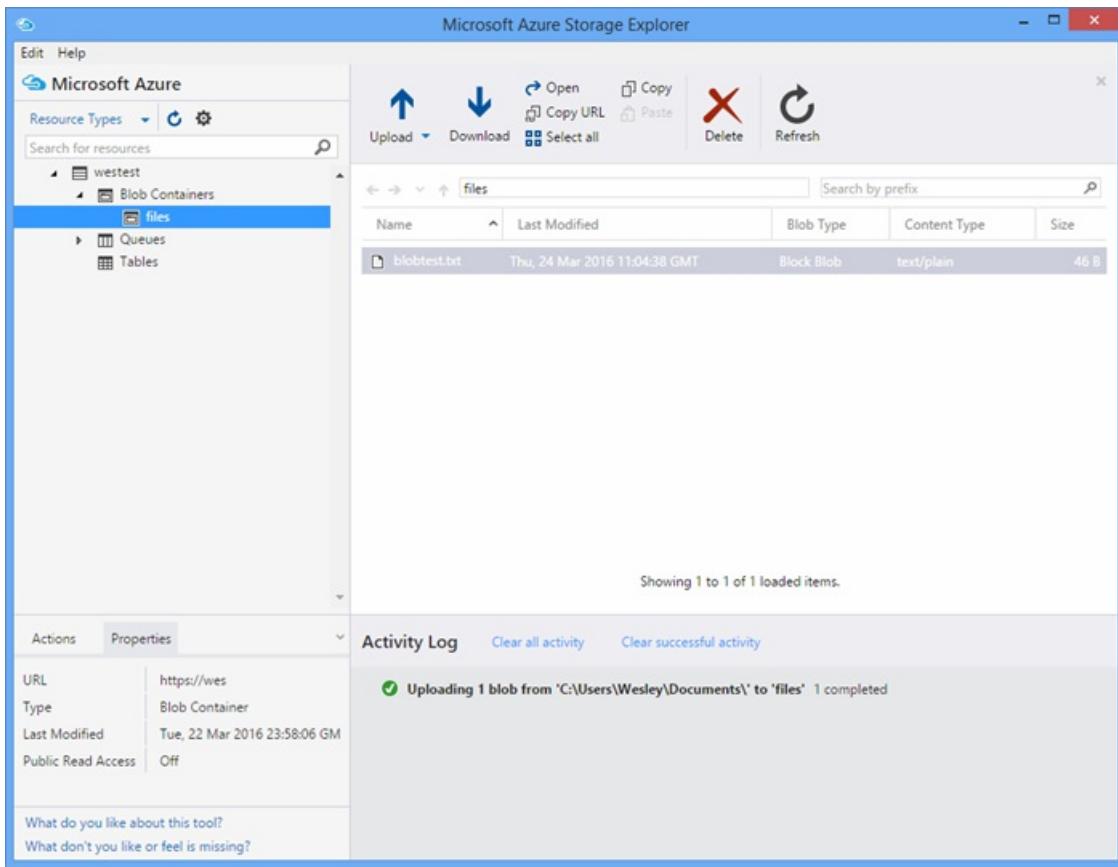
1. In the [Azure portal](#) for your function app, create a C#, F# or JavaScript blob trigger function. Set the path to monitor to the name of your blob container. For example:

```
files
```

2. Click the + button to select or create the storage account you want to use. Then click **Create**.
3. Create a text file with the following text, and save it:

```
A text file for blob trigger function testing.
```

4. Run [Azure Storage Explorer](#), and connect to the blob container in the storage account being monitored.
5. Click **Upload** to upload the text file.



The default blob trigger function code reports the processing of the blob in the logs:

```
2016-03-24T11:30:10  Welcome, you are now connected to log-streaming service.
2016-03-24T11:30:34.472 Function started (Id=739ebc07-ff9e-4ec4-a444-e479cec2e460)
2016-03-24T11:30:34.472 C# Blob trigger function processed: A text file for blob trigger function testing.
2016-03-24T11:30:34.472 Function completed (Success, Id=739ebc07-ff9e-4ec4-a444-e479cec2e460)
```

Test a function within functions

The Azure Functions portal is designed to let you test HTTP and timer triggered functions. You can also create functions to trigger other functions that you are testing.

Test with the Functions portal Run button

The portal provides a **Run** button that you can use to do some limited testing. You can provide a request body by using the button, but you can't provide query string parameters or update request headers.

Test the HTTP trigger function we created earlier by adding a JSON string similar to the following in the **Request body** field. Then click the **Run** button.

```
{
  "name" : "Wes testing Run button",
  "address" : "USA"
}
```

In the portal **Logs** window, output similar to the following is logged in executing the function:

```
2016-03-23T08:03:12 Welcome, you are now connected to log-streaming service.
2016-03-23T08:03:17.357 Function started (Id=753a01b0-45a8-4125-a030-3ad543a89409)
2016-03-23T08:03:18.697 HTTP trigger function processed a request.
RequestUri=https://functions841def78.azurewebsites.net/api/wesmchttptriggernodejs1
2016-03-23T08:03:18.697 Request Headers = {"connection":"Keep-Alive","accept":"*/*","accept-encoding":"gzip","accept-language":"en-US"}
2016-03-23T08:03:18.697 Processing user info from request body...
2016-03-23T08:03:18.697 Processing User Information...
2016-03-23T08:03:18.697 name = Wes testing Run button
2016-03-23T08:03:18.697 address = USA
2016-03-23T08:03:18.744 Function completed (Success, Id=753a01b0-45a8-4125-a030-3ad543a89409)
```

Test with a timer trigger

Some functions can't be adequately tested with the tools mentioned previously. For example, consider a queue trigger function that runs when a message is dropped into [Azure Queue storage](#). You can always write code to drop a message into your queue, and an example of this in a console project is provided later in this article. However, there is another approach you can use that tests functions directly.

You can use a timer trigger configured with a queue output binding. That timer trigger code can then write the test messages to the queue. This section walks through an example.

For more in-depth information on using bindings with Azure Functions, see the [Azure Functions developer reference](#).

Create a queue trigger for testing

To demonstrate this approach, we first create a queue trigger function that we want to test for a queue named `queue-newusers`. This function processes name and address information dropped into Queue storage for a new user.

NOTE

If you use a different queue name, make sure the name you use conforms to the [Naming Queues and MetaData](#) rules. Otherwise, you get an error.

1. In the [Azure portal](#) for your function app, click **New Function** > **QueueTrigger - C#**.

2. Enter the queue name to be monitored by the queue function:

```
queue-newusers
```

3. Click the **+** button to select or create the storage account you want to use. Then click **Create**.

4. Leave this portal browser window open, so you can monitor the log entries for the default queue function template code.

Create a timer trigger to drop a message in the queue

1. Open the [Azure portal](#) in a new browser window, and navigate to your function app.
2. Click **New Function** > **TimerTrigger - C#**. Enter a cron expression to set how often the timer code tests your queue function. Then click **Create**. If you want the test to run every 30 seconds, you can use the following [CRON expression](#):

```
*/30 * * * *
```

3. Click the **Integrate** tab for your new timer trigger.

4. Under **Output**, click **+ New Output**. Then click **queue** and **Select**.

5. Note the name you use for the **queue message object**. You use this in the timer function code.

```
myQueue
```

6. Enter the queue name where the message is sent:

```
queue-newusers
```

7. Click the **+** button to select the storage account you used previously with the queue trigger. Then click **Save**.

8. Click the **Develop** tab for your timer trigger.

9. You can use the following code for the C# timer function, as long as you used the same queue message object name shown earlier. Then click **Save**.

```
using System;

public static void Run(TimerInfo myTimer, out String myQueue, TraceWriter log)
{
    String newUser =
    "{\"name\":\"User testing from C# timer function\", \"address\":\"XYZ\"}";

    log.Verbose($"C# Timer trigger function executed at: {DateTime.Now}");
    log.Verbose($"{newUser}");

    myQueue = newUser;
}
```

At this point, the C# timer function executes every 30 seconds if you used the example cron expression. The logs for the timer function report each execution:

```
2016-03-24T10:27:02 Welcome, you are now connected to log-streaming service.
2016-03-24T10:27:30.004 Function started (Id=04061790-974f-4043-b851-48bd4ac424d1)
2016-03-24T10:27:30.004 C# Timer trigger function executed at: 3/24/2016 10:27:30 AM
2016-03-24T10:27:30.004 {"name":"User testing from C# timer function", "address": "XYZ"}
2016-03-24T10:27:30.004 Function completed (Success, Id=04061790-974f-4043-b851-48bd4ac424d1)
```

In the browser window for the queue function, you can see each message being processed:

```
2016-03-24T10:27:06 Welcome, you are now connected to log-streaming service.
2016-03-24T10:27:30.607 Function started (Id=e304450c-ff48-44dc-ba2e-1df7209a9d22)
2016-03-24T10:27:30.607 C# Queue trigger function processed: {"name":"User testing from C# timer function", "address": "XYZ"}
2016-03-24T10:27:30.607 Function completed (Success, Id=e304450c-ff48-44dc-ba2e-1df7209a9d22)
```

Test a function with code

You may need to create an external application or framework to test your functions.

Test an HTTP trigger function with code: Node.js

You can use a Node.js app to execute an HTTP request to test your function. Make sure to set:

- The `host` in the request options to your function app host.
- Your function name in the `path`.
- Your access code (`<your code>`) in the `path`.

Code example:

```

var http = require("http");

var nameQueryString = "name=Wes%20Query%20String%20Test%20From%20Node.js";

var nameBodyJSON = {
    name : "Wes testing with Node.JS code",
    address : "Dallas, T.X. 75201"
};

var bodyString = JSON.stringify(nameBodyJSON);

var options = {
    host: "functions841def78.azurewebsites.net",
    //path: "/api/HttpTriggerNodeJS2?
code=sc1wt62opn7k9buhrm8jpds4ikxvvj42m5ojdt0p91lz5jnhfr2c74ipoujyq26wab3wk5gkf9&" + nameQueryString,
    path: "/api/HttpTriggerNodeJS2?
code=sc1wt62opn7k9buhrm8jpds4ikxvvj42m5ojdt0p91lz5jnhfr2c74ipoujyq26wab3wk5gkf9",
    method: "POST",
    headers : {
        "Content-Type": "application/json",
        "Content-Length": Buffer.byteLength(bodyString)
    }
};

callback = function(response) {
    var str = ""
    response.on("data", function (chunk) {
        str += chunk;
    });
    response.on("end", function () {
        console.log(str);
    });
}

var req = http.request(options, callback);
console.log("*** Sending name and address in body ***");
console.log(bodyString);
req.end(bodyString);

```

Output:

```

C:\Users\Wesley\testing\Node.js>node testHttpTriggerExample.js
*** Sending name and address in body ***
{"name" : "Wes testing with Node.JS code","address" : "Dallas, T.X. 75201"}
Hello Wes testing with Node.JS code
The address you provided is Dallas, T.X. 75201

```

In the portal **Logs** window, output similar to the following is logged in executing the function:

```

2016-03-23T08:08:55  Welcome, you are now connected to log-streaming service.
2016-03-23T08:08:59.736 Function started (Id=607b891c-08a1-427f-910c-af64ae4f7f9c)
2016-03-23T08:09:01.153 HTTP trigger function processed a request.
RequestUri=http://functionsExample.azurewebsites.net/api/WesmcHttpTriggerNodeJS1/?code=XXXXXXXXXX=
2016-03-23T08:09:01.153 Request Headers = {"connection": "Keep-
Alive", "host": "functionsExample.azurewebsites.net"}
2016-03-23T08:09:01.153 Name not provided as query string param. Checking body...
2016-03-23T08:09:01.153 Request Body Type = object
2016-03-23T08:09:01.153 Request Body = [object Object]
2016-03-23T08:09:01.153 Processing User Information...
2016-03-23T08:09:01.215 Function completed (Success, Id=607b891c-08a1-427f-910c-af64ae4f7f9c)

```

Test a queue trigger function with code: C#

We mentioned earlier that you can test a queue trigger by using code to drop a message in your queue. The following example code is based on the C# code presented in the [Getting started with Azure Queue storage](#) tutorial. Code for other languages is also available from that link.

To test this code in a console app, you must:

- [Configure your storage connection string in the app.config file.](#)
- Pass a `name` and `address` as parameters to the app. For example,

`C:\myQueueConsoleApp\test.exe "Wes testing queues" "in a console app"`. (This code accepts the name and address for a new user as command-line arguments during runtime.)

Example C# code:

```
static void Main(string[] args)
{
    string name = null;
    string address = null;
    string queueName = "queue-newusers";
    string JSON = null;

    if (args.Length > 0)
    {
        name = args[0];
    }
    if (args.Length > 1)
    {
        address = args[1];
    }

    // Retrieve storage account from connection string
    CloudStorageAccount storageAccount =
    CloudStorageAccount.Parse(ConfigurationManager.AppSettings["StorageConnectionString"]);

    // Create the queue client
    CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

    // Retrieve a reference to a queue
    CloudQueue queue = queueClient.GetQueueReference(queueName);

    // Create the queue if it doesn't already exist
    queue.CreateIfNotExists();

    // Create a message and add it to the queue.
    if (name != null)
    {
        if (address != null)
            JSON = String.Format("{{\"name\":\"{0}\",\"address\":\"{1}\",\"}}", name, address);
        else
            JSON = String.Format("{{\"name\":\"{0}\",\"}}", name);
    }

    Console.WriteLine("Adding message to " + queueName + "...");
    Console.WriteLine(JSON);

    CloudQueueMessage message = new CloudQueueMessage(JSON);
    queue.AddMessage(message);
}
```

In the browser window for the queue function, you can see each message being processed:

```
2016-03-24T10:27:06 Welcome, you are now connected to log-streaming service.  
2016-03-24T10:27:30.607 Function started (Id=e304450c-ff48-44dc-ba2e-1df7209a9d22)  
2016-03-24T10:27:30.607 C# Queue trigger function processed: {"name":"Wes testing queues","address":"in a  
console app"}  
2016-03-24T10:27:30.607 Function completed (Success, Id=e304450c-ff48-44dc-ba2e-1df7209a9d22)
```

Code and test Azure Functions locally

1/3/2018 • 10 min to read • [Edit Online](#)

While the [Azure portal](#) provides a full set of tools for developing and testing Azure Functions, many developers prefer a local development experience. Azure Functions makes it easy to use your favorite code editor and local development tools to develop and test your functions on your local computer. Your functions can trigger on events in Azure, and you can debug your C# and JavaScript functions on your local computer.

If you are a Visual Studio C# developer, Azure Functions also [integrates with Visual Studio 2017](#).

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.

Install the Azure Functions Core Tools

[Azure Functions Core Tools](#) is a local version of the Azure Functions runtime that you can run on your local development computer. It's not an emulator or simulator. It's the same runtime that powers Functions in Azure. There are two versions of Azure Functions Core Tools, one for version 1.x of the runtime and one for version 2.x. Both versions are provided as an [npm package](#).

NOTE

Before you install either version, you must [install NodeJS](#), which includes npm. For version 2.x of the tools, only Node.js 8.5 and later versions are supported.

Version 1.x runtime

The original version of the tools uses the Functions 1.x runtime. This version uses the .NET Framework and is only supported on Windows computers. Use the following command to install the version 1.x tools:

```
npm install -g azure-functions-core-tools
```

Version 2.x runtime

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. Use this version for cross-platform development and when the Functions runtime 2.x is required.

IMPORTANT

Before installing Azure Functions Core Tools, [install .NET Core 2.0](#).

Azure Functions runtime 2.0 is in preview, and currently not all features of Azure Functions are supported. For more information, see [Azure Functions runtime 2.0 known issues](#)

Use the following command to install the version 2.0 tools:

```
npm install -g azure-functions-core-tools@core
```

When installing on Ubuntu use `sudo`, as follows:

```
sudo npm install -g azure-functions-core-tools@core
```

When installing on macOS and Linux, you may need to include the `unsafe-perm` flag, as follows:

```
sudo npm install -g azure-functions-core-tools@core --unsafe-perm true
```

Run Azure Functions Core Tools

Azure Functions Core Tools adds the following command aliases:

- **func**
- **azfun**
- **azurefunctions**

Any of these aliases can be used where `func` is shown in the examples.

```
func init MyFunctionProj
```

Create a local Functions project

When running locally, a Functions project is a directory that has the files `host.json` and `local.settings.json`. This directory is the equivalent of a function app in Azure. To learn more about the Azure Functions folder structure, see the [Azure Functions developers guide](#).

In the terminal window or from a command prompt, run the following command to create the project and local Git repository:

```
func init MyFunctionProj
```

The output looks like the following example:

```
Writing .gitignore
Writing host.json
Writing local.settings.json
Created launch.json
Initialized empty Git repository in D:/Code/Playground/MyFunctionProj/.git/
```

To create the project without a local Git repository, use the `--no-source-control [-n]` option.

Local settings file

The file `local.settings.json` stores app settings, connection strings, and settings for Azure Functions Core Tools. It has the following structure:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "<connection string>",
    "AzureWebJobsDashboard": "<connection string>"
  },
  "Host": {
    "LocalHttpPort": 7071,
    "CORS": "*"
  },
  "ConnectionStrings": {
    "SQLConnectionString": "Value"
  }
}
```

SETTING	DESCRIPTION
IsEncrypted	When set to true , all values are encrypted using a local machine key. Used with <code>func settings</code> commands. Default value is false .
Values	Collection of application settings used when running locally. AzureWebJobsStorage and AzureWebJobsDashboard are examples; for a complete list, see app settings reference .
Host	Settings in this section customize the Functions host process when running locally.
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 value.
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.
ConnectionStrings	Contains the database connection strings for your functions. Connection strings in this object are added to the environment with the provider type of System.Data.SqlClient .

Most triggers and bindings have a **Connection** property that maps to the name of an environment variable or app setting. For each connection property, there must be app setting defined in local.settings.json file.

These settings can also be read in your code as environment variables. In C#, use `System.Environment.GetEnvironmentVariable` or `ConfigurationManager.AppSettings`. In JavaScript, use `process.env`. Settings specified as a system environment variable take precedence over values in the local.settings.json file.

Settings in the local.settings.json file are only used by Functions tools when running locally. 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.

When no valid storage connection string is set for **AzureWebJobsStorage**, 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' or specify a connection string in local.settings.json.

NOTE

Your function app can use the Azure Storage Emulator for the **AzureWebJobsStorage** and **AzureWebJobsDashboard** connection settings that are required by the project. To use the emulator, set the values of these keys to `UseDevelopmentStorage=true`.

Configure app settings

To set a value for connection strings, you can do one of the following options:

- Enter the connection string from [Azure Storage Explorer](#).
- Use one of the following commands:

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

```
func azure storage fetch-connection-string <StorageAccountName>
```

Both commands require you to first sign-in to Azure.

Create a function

To create a function, run the following command:

```
func new
```

`func new` supports the following optional arguments:

ARGUMENT	DESCRIPTION
<code>--language -l</code>	The template programming language, such as C#, F#, or JavaScript.
<code>--template -t</code>	The template name.
<code>--name -n</code>	The function name.

For example, to create a JavaScript HTTP trigger, run:

```
func new --language JavaScript --template HttpTrigger --name MyHttpTrigger
```

To create a queue-triggered function, run:

```
func new --language JavaScript --template QueueTrigger --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
```

`func host start` supports the following options:

OPTION	DESCRIPTION
<code>--port -p</code>	The local port to listen on. Default value: 7071.
<code>--debug <type></code>	The options are <code>vsCode</code> and <code>vs</code> .
<code>--cors</code>	A comma-separated list of CORS origins, with no spaces.
<code>--nodeDebugPort -n</code>	The port for the node debugger to use. Default: A value from launch.json or 5858.
<code>--debugLevel -d</code>	The console trace level (off, verbose, info, warning, or error). Default: Info.
<code>--timeout -t</code>	The timeout for the Functions host to start, in seconds. Default: 20 seconds.
<code>--useHttps</code>	Bind to https://localhost:{port} rather than to http://localhost:{port}. By default, this option creates a trusted certificate on your computer.
<code>--pause-on-error</code>	Pause for additional input before exiting the process. Useful when launching Azure Functions Core Tools from an integrated development environment (IDE).

When the Functions host starts, it outputs the URL of HTTP-triggered functions:

```
Found the following functions:  
Host.Functions.MyHttpTrigger  
  
ob host started  
Http Function MyHttpTrigger: http://localhost:7071/api/MyHttpTrigger
```

Debug in VS Code or Visual Studio

To attach a debugger, pass the `--debug` argument. To debug JavaScript functions, use Visual Studio Code. For C# functions, use Visual Studio.

To debug C# functions, use `--debug vs`. You can also use [Azure Functions Visual Studio 2017 Tools](#).

To launch the host and set up JavaScript debugging, run:

```
func host start --debug vscode
```

Then, in Visual Studio Code, in the **Debug** view, select **Attach to Azure Functions**. You can attach breakpoints, inspect variables, and step through code.

The screenshot shows the Azure Functions local developer tools interface. On the left, the 'VARIABLES' pane is open, showing 'Local' variables: context (Object), req (Object), and this (#<Object>). In the center, the code editor displays the following JavaScript function:

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

    if (req.query.name || (req.body && req.body.name)) {
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
}
```

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"}'
```

Note that 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.
<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

To publish a Functions project to a function app in Azure, use the `publish` command:

```
func azure functionapp publish <FunctionAppName>
```

You can use the following options:

OPTION	DESCRIPTION
--publish-local-settings -i	Publish settings in local.settings.json to Azure, prompting to overwrite if the setting already exists.
--overwrite-settings -y	Must be used with <code>-i</code> . Overwrites AppSettings in Azure with local value if different. Default is prompt.

This command publishes to an existing function app in Azure. An error occurs when the `<FunctionAppName>` 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](#).

The `publish` command uploads the contents of the Functions project directory. If you delete files locally, the `publish` command does not delete them from Azure. You can delete files in Azure by using the [Kudu tool](#) in the [Azure portal](#).

IMPORTANT

When you create a function app in Azure, it uses version 1.x of the Function runtime by default. To make the function app use version 2.x of the runtime, add the application setting `FUNCTIONS_EXTENSION_VERSION=beta`.

Use the following Azure CLI code to add this setting to your function app:

```
az functionapp config appsettings set --name <function_app> \
--resource-group myResourceGroup \
--settings FUNCTIONS_EXTENSION_VERSION=beta
```

Next steps

Azure Functions Core Tools is [open source and hosted on GitHub](#).

To file a bug or feature request, [open a GitHub issue](#).

Azure Functions Tools for Visual Studio

1/19/2018 • 7 min to read • [Edit Online](#)

Azure Functions Tools for Visual Studio 2017 is an extension for Visual Studio that lets you develop, test, and deploy C# functions to Azure. If this experience is your first with Azure Functions, you can learn more at [An introduction to Azure Functions](#).

The Azure Functions Tools provides the following benefits:

- Edit, build, and run functions on your local development computer.
- Publish your Azure Functions project directly to Azure.
- Use WebJobs attributes to declare function bindings directly in the C# code instead of maintaining a separate function.json for binding definitions.
- 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.

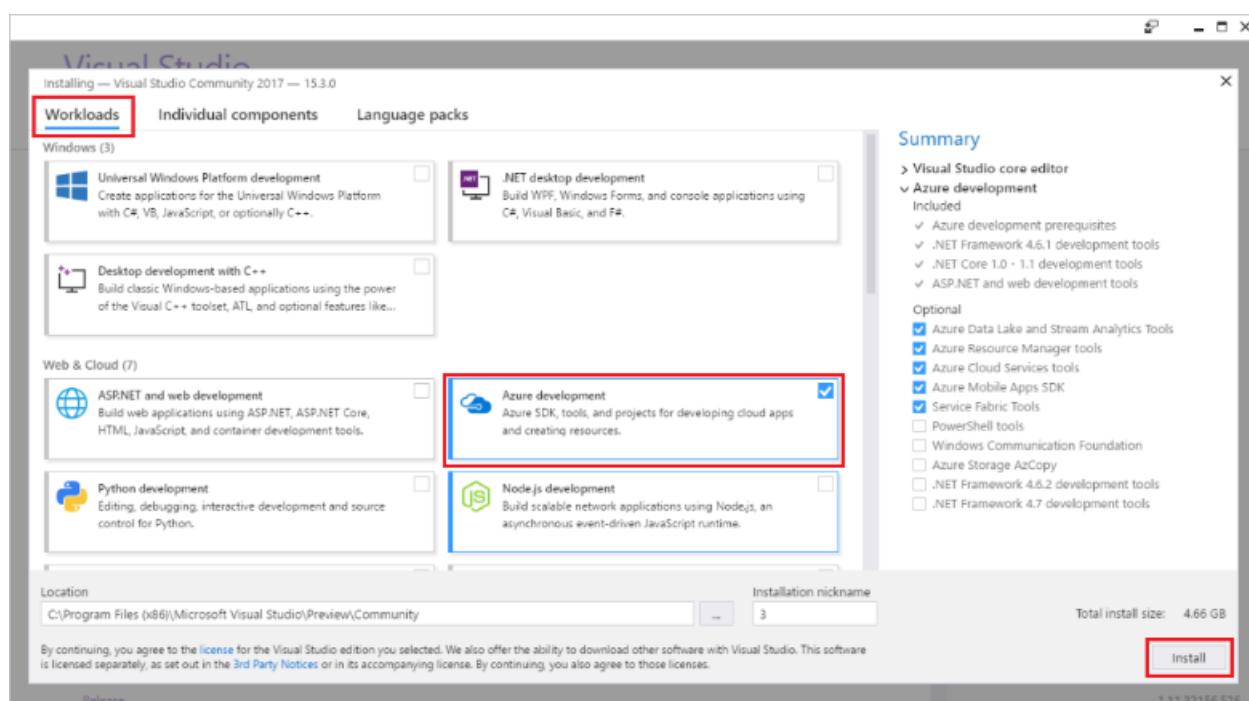
This topic shows you how to use the Azure Functions Tools for Visual Studio 2017 to develop your functions in C#. You also learn how to publish your project to Azure as a .NET assembly.

IMPORTANT

Don't mix local development with portal development in the same 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

Azure Functions Tools is included in the Azure development workload of [Visual Studio 2017 version 15.4](#), or a later version. Make sure you include the **Azure development** workload in your Visual Studio 2017 installation:



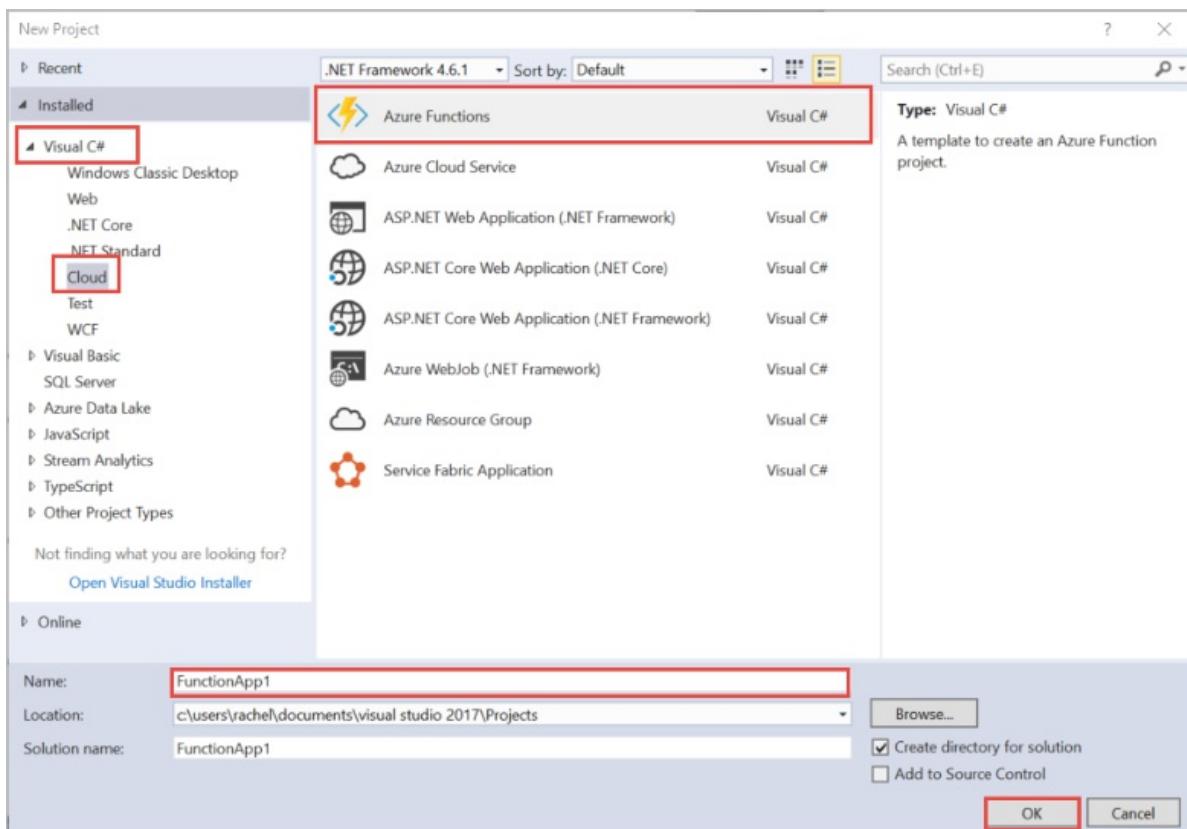
To create and deploy functions, you also need:

- An active Azure subscription. If you don't have an Azure subscription, [free accounts](#) are available.
- An Azure Storage account. To create a storage account, see [Create a storage account](#).

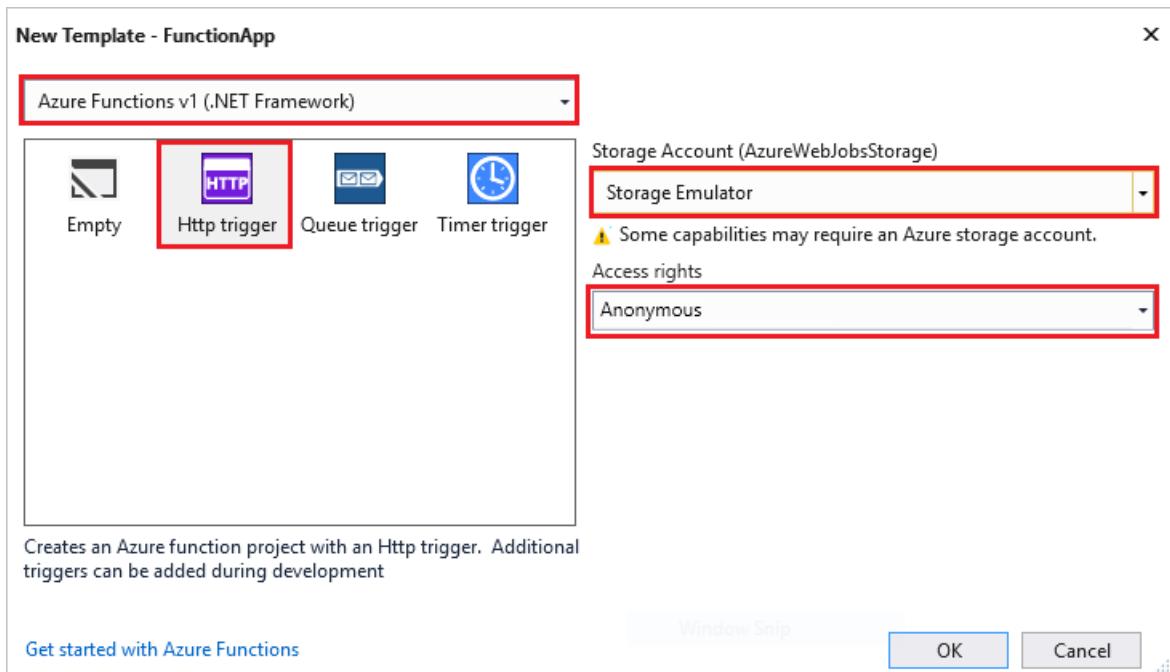
Create an Azure Functions project

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

1. In Visual Studio, select **New > Project** from the **File** menu.
2. In the **New Project** dialog, select **Installed**, expand **Visual C# > Cloud**, select **Azure Functions**, type a **Name** for your project, and click **OK**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other nonalphanumeric characters.



3. Use the settings specified in the table that follows the image.



SETTING	SUGGESTED VALUE	DESCRIPTION
Version	Azure Functions v1 (.NET Framework)	This creates a function project that uses the version 1 runtime of Azure Functions. The version 2 runtime, which supports .NET Core, is currently in preview. For more information, see How to target Azure Functions runtime version .
Template	HTTP trigger	This creates a function triggered by an HTTP request.
Storage account	Storage Emulator	An HTTP trigger doesn't use the Storage account connection. All other trigger types require a valid Storage account connection string.
Access rights	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 .

4. Click **OK** to create the function project and HTTP triggered function.

Configure the project for local development

When you create a new project using the Azure Functions template, you get an empty C# project that contains 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 by Azure, they are used by the [Azure Functions Core Tools](#). Use this file to specify settings, such as connection

strings to other Azure services. Add a new key to the **Values** array for each connection required by functions in your project. For more information, see [Local settings file](#) in the Azure Functions Core Tools topic.

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.

NOTE

Your function app can use the Azure Storage Emulator for the **AzureWebJobsStorage** and **AzureWebJobsDashboard** connection settings that are required by the project. To use the emulator, set the values of these keys to `UseDevelopmentStorage=true`.

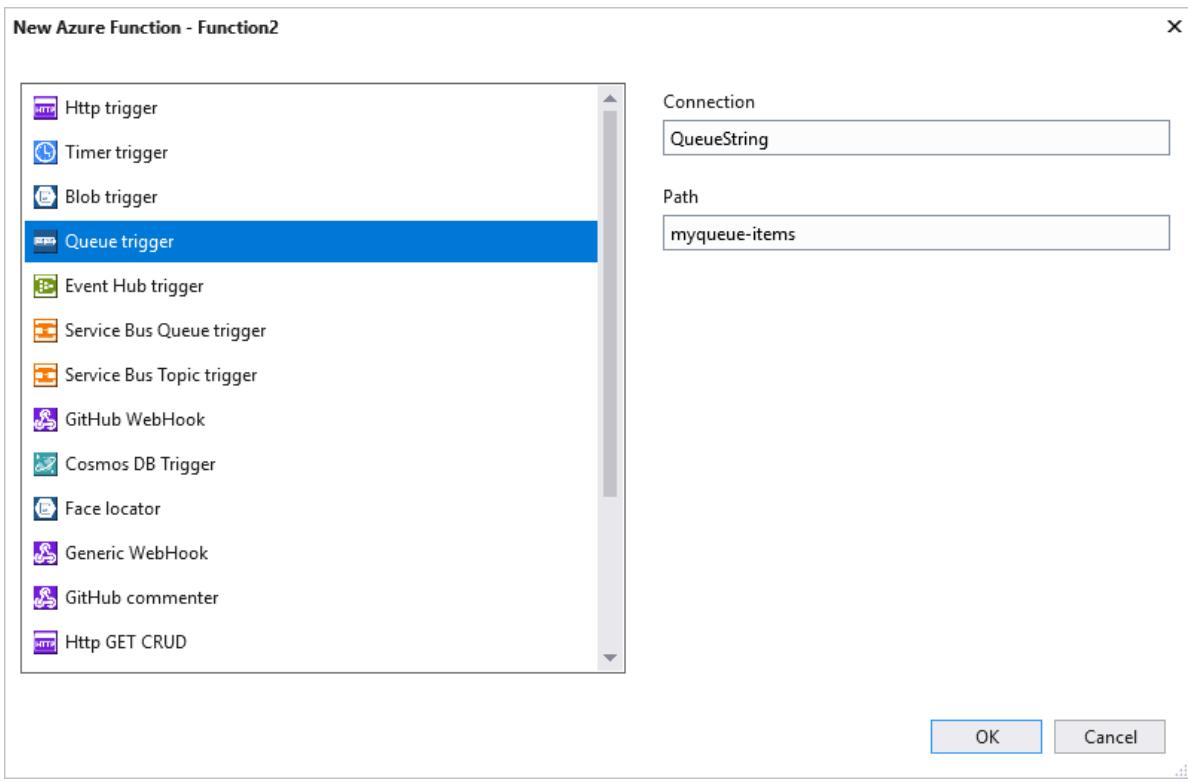
To set the storage account connection string:

1. In Visual Studio, open **Cloud Explorer**, expand **Storage Account > Your Storage Account**, 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.

Create a function

In pre-compiled functions, the bindings used by the function are defined by applying attributes in the code. When you use the Azure Functions Tools to create your functions from the provided templates, these 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;

namespace FunctionApp1
{
    public static class Function1
    {
        [FunctionName("QueueTriggerCSharp")]
        public static void Run([QueueTrigger("myqueue-items", Connection = "QueueStorage")]string
myQueueItem, TraceWriter log)
        {
            log.Info($"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.

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.

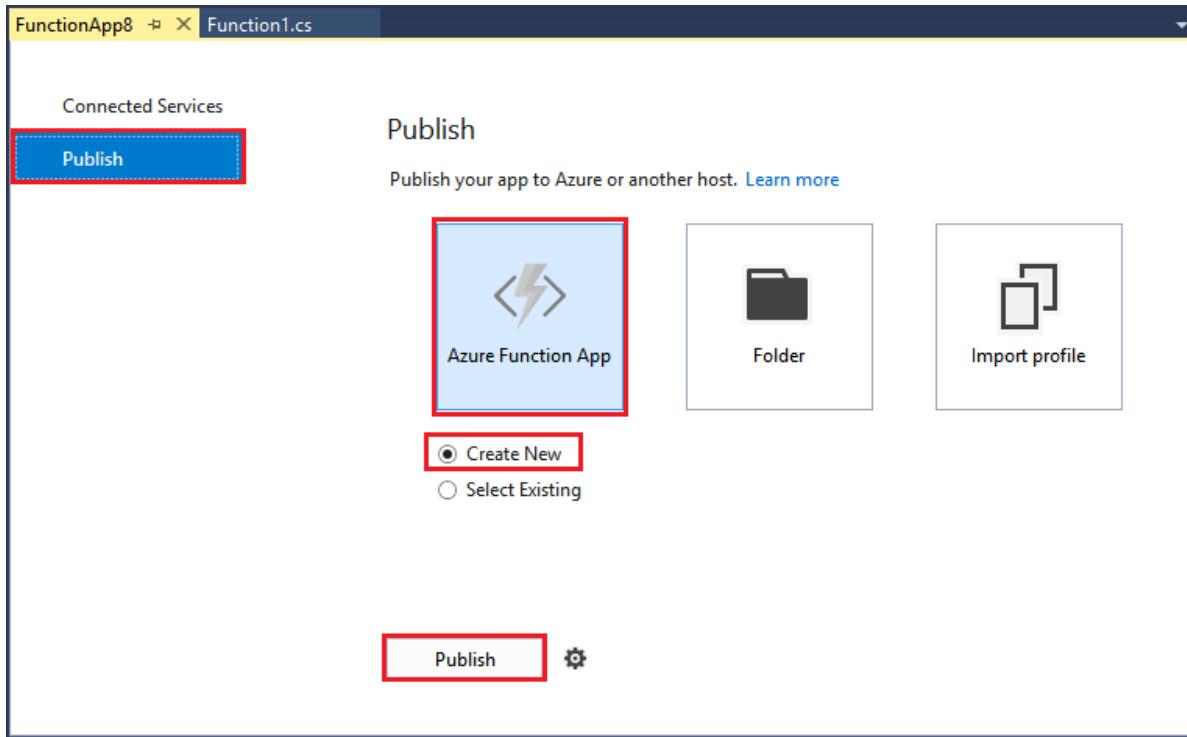
With the project running, you can test your code as you would test deployed function. For more information, see [Strategies for testing your code in Azure Functions](#). When running in debug mode, breakpoints are hit in Visual Studio as expected.

For an example of how to test a queue triggered function, see the [queue triggered function quickstart tutorial](#).

To learn more about using the Azure Functions Core Tools, see [Code and test Azure functions locally](#).

Publish to Azure

1. In **Solution Explorer**, right-click the project and select **Publish**. Choose **Create New** and then **Publish**.



2. If you haven't already connected Visual Studio to your Azure account, select **Add an account...**.
3. In the **Create App Service** dialog, use the **Hosting** settings as specified in the following table:

Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Hosting **Services**

App Name: FunctionApp20180114125726

Subscription: Visual Studio Enterprise

Resource Group: myResourceGroup (southcentralus) **New...**

App Service Plan: FunctionApp20180114125726Plan* **New...**

Clicking the Create button will create the following Azure resources

[Explore additional Azure services](#)

App Service - FunctionApp20180114125726
App Service Plan - FunctionApp20180114125726Plan

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

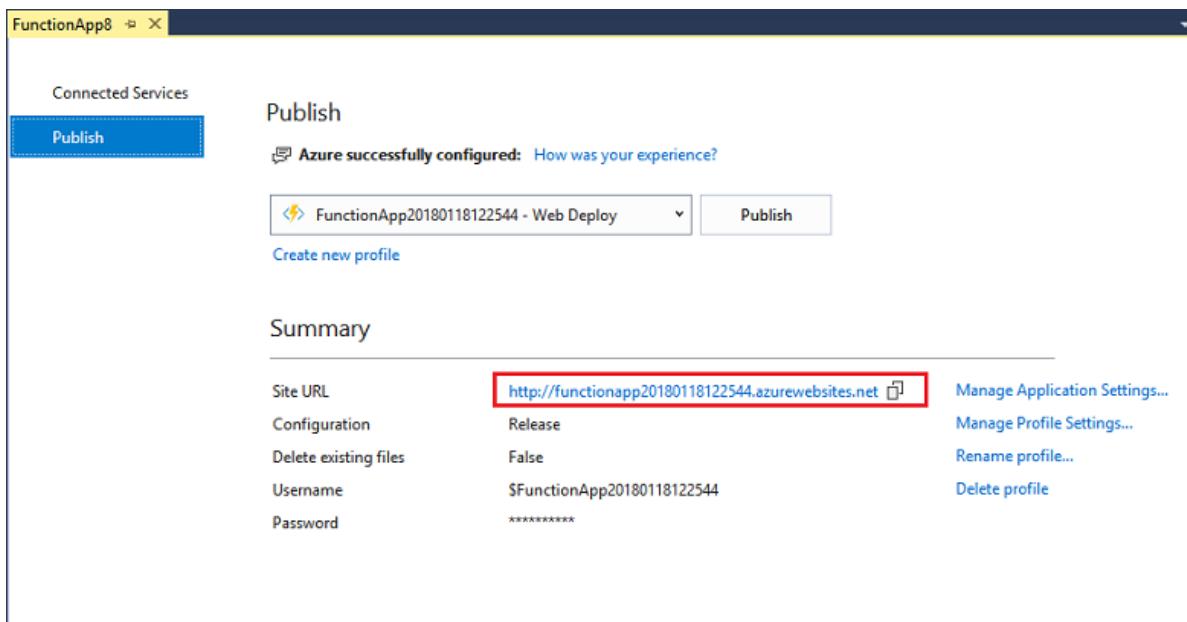
Export... **Create** **Cancel**

SETTING	SUGGESTED VALUE	DESCRIPTION
App Name	Globally unique name	Name that uniquely identifies your new function app.
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.
App Service Plan	Consumption plan	Make sure to choose the Consumption under Size after you click New to create a new plan. Also, choose a Location in a region near you or near other services your functions access.

NOTE

An Azure storage account is required by the Functions runtime. Because of this, a new Azure Storage account is created for you when you create a function app.

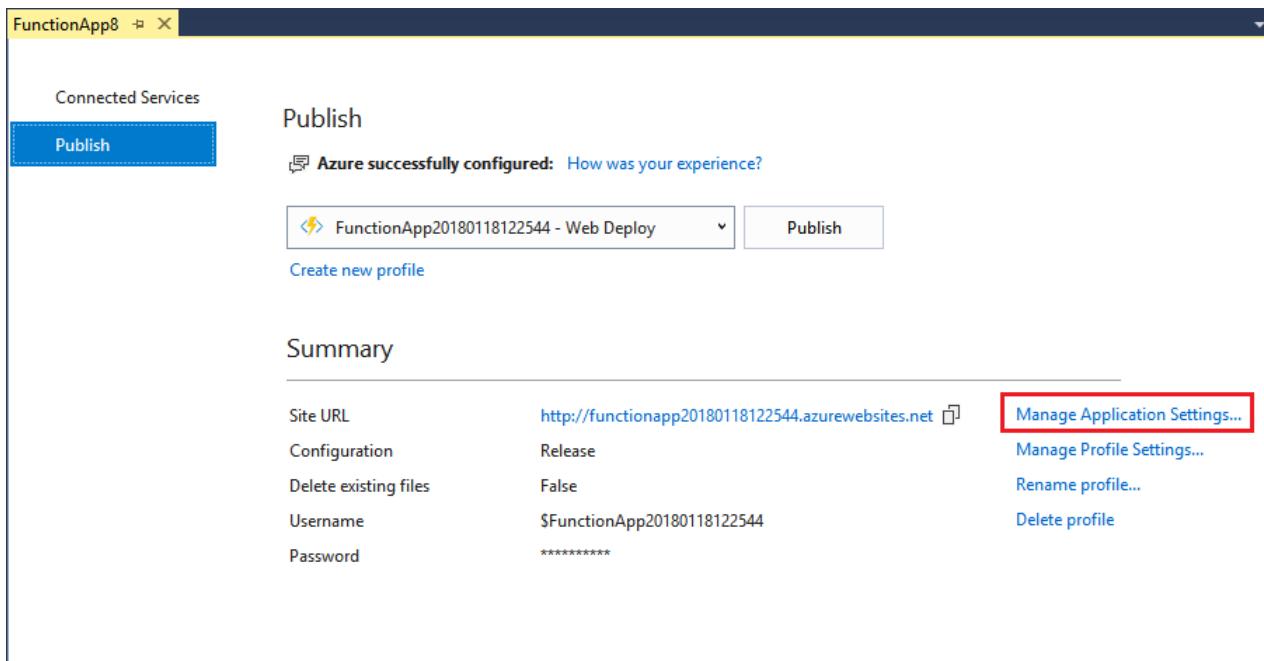
4. Click **Create** to create a function app and related resources in Azure with these settings and deploy your function project code.
5. After the deployment is complete, make a note of the **Site URL** value, which is the address of your function app in Azure.



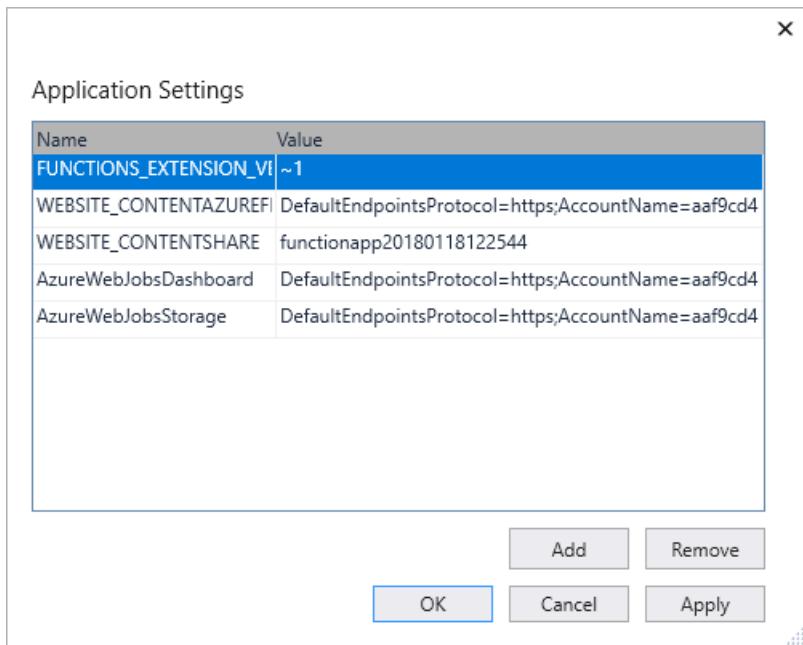
Function app settings

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.



This displays the **Application Settings** dialog for the function app, where you can add new application settings or modify existing ones.



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.](#)

Next steps

For more information about Azure Functions Tools, see the Common Questions section of the [Visual Studio 2017 Tools for Azure Functions](#) blog post.

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 topic also links to examples of how to use attributes to declare the various types of bindings supported by Azure Functions.

Continuous deployment for Azure Functions

9/28/2017 • 5 min to read • [Edit Online](#)

Azure Functions makes it easy to deploy your function app using App Service continuous integration. Functions integrates with BitBucket, Dropbox, GitHub, and Visual Studio Team Services (VSTS). This enables a workflow where function code updates made by using one of these integrated services trigger deployment to Azure. If you are new to Azure Functions, start with [Azure Functions Overview](#).

Continuous deployment is a great option for projects where multiple and frequent contributions are being integrated. It also lets you maintain source control on your functions code. The following deployment sources are currently supported:

- [Bitbucket](#)
- [Dropbox](#)
- External repository (Git or Mercurial)
- [Git local repository](#)
- [GitHub](#)
- [OneDrive](#)
- [Visual Studio Team Services](#)

Deployments are configured on a per-function app basis. After continuous deployment is enabled, access to function code in the portal is set to *read-only*.

Continuous deployment requirements

You must have your deployment source configured and your functions code in the deployment source before you set up continuous deployment. In a given function app deployment, each function lives in a named subdirectory, where the directory name is the name of the function.

The code for all the functions in a specific function app is located in a root folder that contains a host configuration file and one or more subfolders. Each subfolder contains the code for a separate function, as in the following example:

```
wwwroot
| - host.json
| - mynodefunction
| | - function.json
| | | - index.js
| | | - node_modules
| | | | - ... packages ...
| | | - package.json
| - mycsharpfunction
| | - function.json
| | - run.csx
```

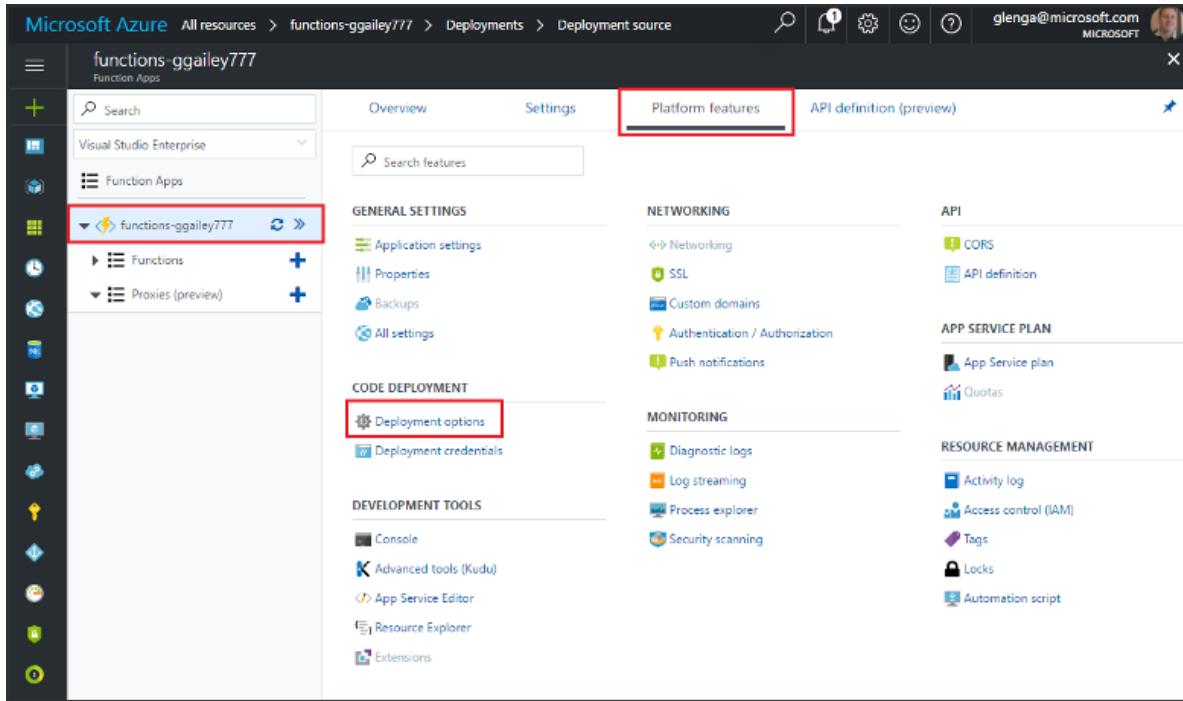
The host.json file contains some runtime-specific configurations, and sits in the root folder of the function app. For information about settings that are available, see the [host.json reference](#).

Each function has a folder that contains one or more code files, the function.json configuration, and other dependencies.

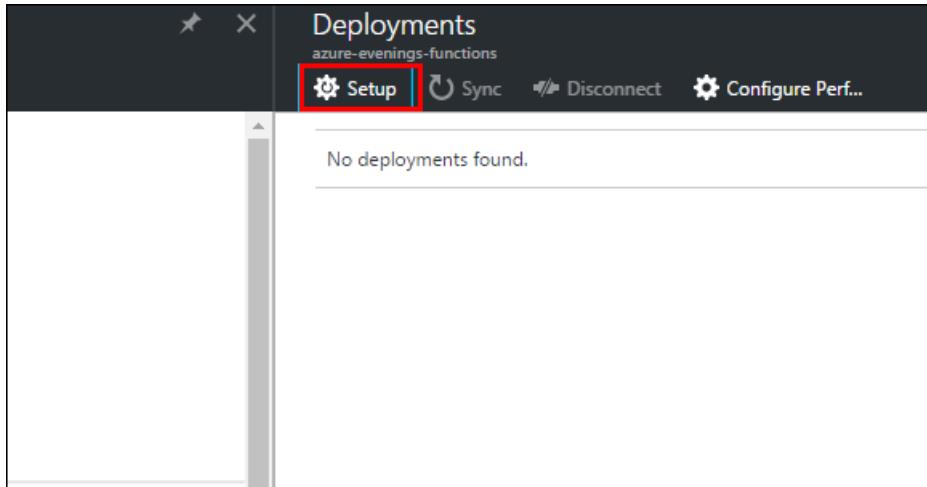
Set up continuous deployment

Use this procedure to configure continuous deployment for an existing function app. These steps demonstrate integration with a GitHub repository, but similar steps apply for Visual Studio Team Services or other deployment services.

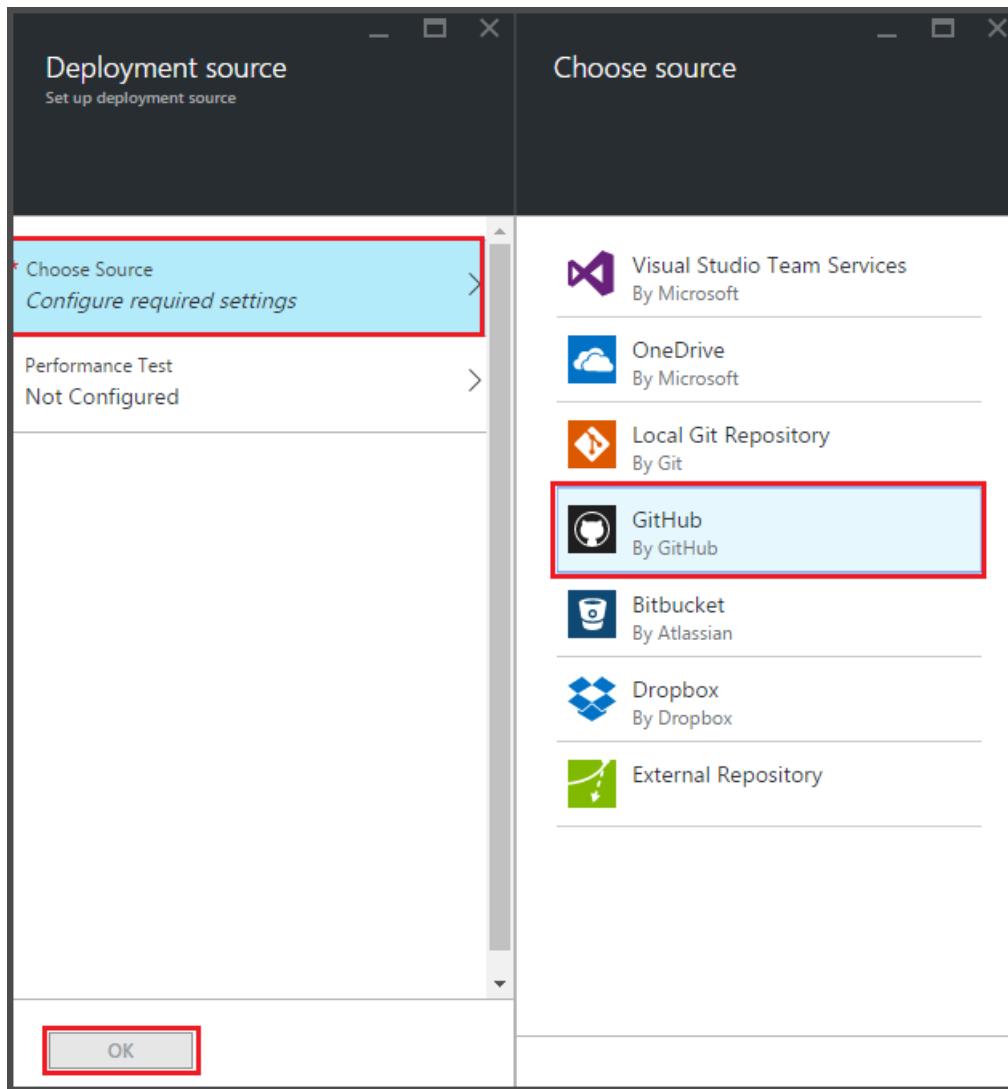
1. In your function app in the [Azure portal](#), click **Platform features** and **Deployment options**.



2. Then in the **Deployments** blade click **Setup**.



3. In the **Deployment source** blade, click **Choose source**, then fill in the information for your chosen deployment source and click **OK**.



After continuous deployment is configured, all file changes in your deployment source are copied to the function app and a full site deployment is triggered. The site is redeployed when files in the source are updated.

Deployment options

The following are some typical deployment scenarios:

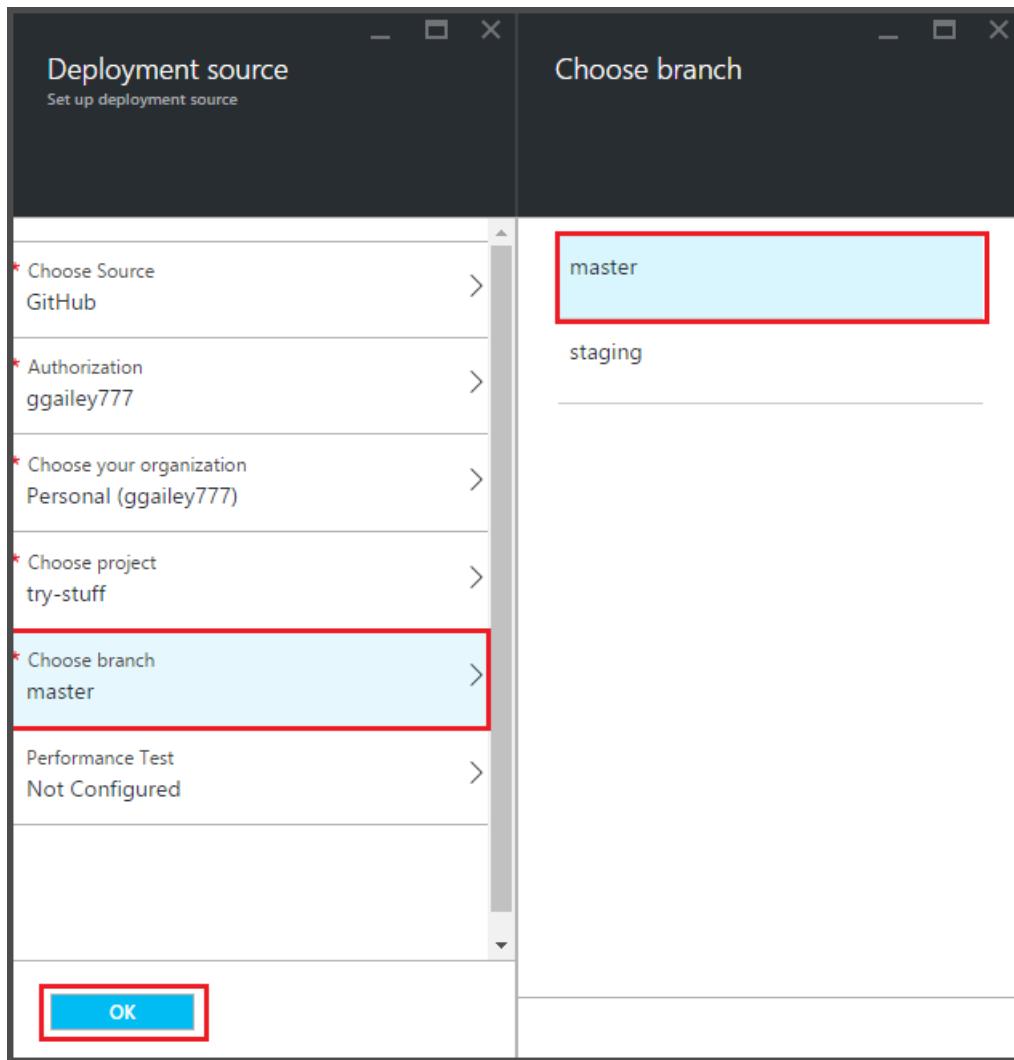
- [Create a staging deployment](#)
- [Move existing functions to continuous deployment](#)

Create a staging deployment

Function Apps doesn't yet support deployment slots. However, you can still manage separate staging and production deployments by using continuous integration.

The process to configure and work with a staging deployment looks generally like this:

1. Create two function apps in your subscription, one for the production code and one for staging.
2. Create a deployment source, if you don't already have one. This example uses [GitHub](#).
3. For your production function app, complete the preceding steps in **Set up continuous deployment** and set the deployment branch to the master branch of your GitHub repository.



4. Repeat this step for the staging function app, but choose the staging branch instead in your GitHub repo. If your deployment source doesn't support branching, use a different folder.
5. Make updates to your code in the staging branch or folder, then verify that those changes are reflected in the staging deployment.
6. After testing, merge changes from the staging branch into the master branch. This merge triggers deployment to the production function app. If your deployment source doesn't support branches, overwrite the files in the production folder with the files from the staging folder.

Move existing functions to continuous deployment

When you have existing functions that you have created and maintained in the portal, you need to download your existing function code files using FTP or the local Git repository before you can set up continuous deployment as described above. You can do this in the App Service settings for your function app. After your files are downloaded, you can upload them to your chosen continuous deployment source.

NOTE

After you configure continuous integration, you will no longer be able to edit your source files in the Functions portal.

- [How to: Configure deployment credentials](#)
- [How to: Download files using FTP](#)
- [How to: Download files using the local Git repository](#)

How to: Configure deployment credentials

Before you can download files from your function app with FTP or local Git repository, you must configure your

credentials to access the site. Credentials are set at the Function app level. Use the following steps to set deployment credentials in the Azure portal:

1. In your function app in the [Azure portal](#), click **Platform features** and **Deployment credentials**.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various icons and sections like Overview, Settings, Platform features, API definition, etc. The 'Platform features' tab is currently selected. Under the 'CODE DEPLOYMENT' section, the 'Deployment credentials' option is highlighted with a red box. On the right, a modal window titled 'Set deployment credentials' is open. It contains fields for 'New name and password' (Git and FTP can't authenticate using the account and password to use with those technologies). There are three input fields: 'FTP/deployment username' (containing 'myusername'), 'Password', and 'Confirm password', all of which are also highlighted with red boxes. At the bottom right of the modal are 'Save' and 'Discard' buttons, with 'Save' being the one highlighted.

2. Type in a username and password, then click **Save**. You can now use these credentials to access your function app from FTP or the built-in Git repo.

How to: Download files using FTP

1. In your function app in the [Azure portal](#), click **Platform features** and **Properties**, then copy the values for **FTP/Deployment User**, **FTP Host Name**, and **FTPS Host Name**.

FTP/Deployment User must be entered as displayed in the portal, including the app name, to provide proper context for the FTP server.

The screenshot shows the Microsoft Azure Properties blade for a function app named "functions-ggailey7777". The "Platform features" tab is selected, indicated by a red box. The blade includes sections for General Settings (Application settings, SSL, Backups, All settings), Code Deployment (Deployment options, Deployment credentials), Development Tools (Console, Advanced tools (Kudu), App Service Editor, Resource Explorer, Extensions), Networking (Networking, SSL, Custom domains, Authentication / Authorization, Push notifications), Monitoring (Diagnostic logs, Log streaming, Process explorer, Security scanning), and FTP-related settings (Deployment trigger URL, FTP/Deployment user, FTP host name, FTP diagnostic logs, FTPS host name, FTPS diagnostic logs, Resource ID). Several fields in the right-hand section are highlighted with red boxes.

Microsoft Azure < Properties

Overview Settings Platform features AB/de

Search features

GENERAL SETTINGS

- Application settings
- SSL **Properties**
- Backups
- All settings

CODE DEPLOYMENT

- Deployment options
- Deployment credentials

DEVELOPMENT TOOLS

- Console
- Advanced tools (Kudu)
- App Service Editor
- Resource Explorer
- Extensions

NETWORKING

- Networking
- SSL
- Custom domains
- Authentication / Authorization
- Push notifications

MONITORING

- Diagnostic logs
- Log streaming
- Process explorer
- Security scanning

Properties
functions-ggailey7777

DEPLOYMENT TRIGGER URL
`https://$functions-ggailey7777:DelCGcbN`

FTP/DEPLOYMENT USER
`functions-ggailey7777/ggailey777`

FTP HOST NAME
`ftp://waws-prod-sn1-041.ftp.azurewebsites.net`

FTP DIAGNOSTIC LOGS
`ftp://waws-prod-sn1-041.ftp.azurewebsites.net`

FTPS HOST NAME
`ftps://waws-prod-sn1-041.ftp.azurewebsites.net`

FTPS DIAGNOSTIC LOGS
`ftps://waws-prod-sn1-041.ftp.azurewebsites.net`

RESOURCE ID
`/subscriptions/3e9dd381-f085-4f40-8ec5`

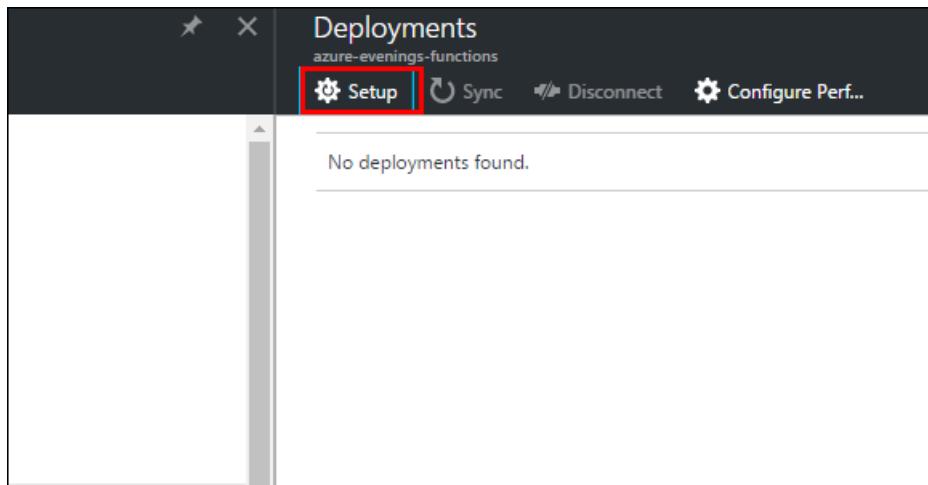
- From your FTP client, use the connection information you gathered to connect to your app and download the source files for your functions.

How to: Download files using a local Git repository

1. In your function app in the [Azure portal](#), click **Platform features** and **Deployment options**.

The screenshot shows the Azure portal interface for a Function App named "functions-ggailey777". The left sidebar lists various resources under "Visual Studio Enterprise". The main content area has tabs for "Overview", "Settings", "Platform features" (which is highlighted with a red box), and "API definition (preview)". The "Platform features" tab contains sections for "GENERAL SETTINGS" (Application settings, Properties, Backups, All settings), "CODE DEPLOYMENT" (Deployment options, Deployment credentials), "DEVELOPMENT TOOLS" (Console, Advanced tools (Kudu), App Service Editor, Resource Explorer, Extensions), "NETWORKING" (Networking, SSL, Custom domains, Authentication / Authorization, Push notifications), "MONITORING" (Diagnostic logs, Log streaming, Process explorer, Security scanning), "API" (CORS, API definition), "APP SERVICE PLAN" (App Service plan, Quotas), and "RESOURCE MANAGEMENT" (Activity log, Access control (IAM), Tags, Locks, Automation script).

2. Then in the **Deployments** blade click **Setup**.



3. In the **Deployment source** blade, click **Local Git repository** and then click **OK**.

4. In **Platform features**, click **Properties** and note the value of Git URL.

Properties
functions-ggailey777

GIT URL
https://ggailey777@functions-ggailey77

5. Clone the repository on your local machine using a Git-aware command prompt or your favorite Git tool.

The Git clone command looks like this:

```
git clone https://username@my-function-app.scm.azurewebsites.net:443/my-function-app.git
```

6. Fetch files from your function app to the clone on your local computer, as in the following example:

```
git pull origin master
```

If requested, supply your [configured deployment credentials](#).

Next steps

[Best Practices for Azure Functions](#)

Zip push deployment for Azure Functions

12/20/2017 • 5 min 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 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](#).

For faster iteration during development, it's often easier to deploy your function app project files directly from a compressed .zip file. 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 push deployment reference](#).

Deployment .zip file requirements

The .zip file that you use for push deployment must contain all of the project files in your function app, including your function code.

IMPORTANT

When you use .zip push deployment, any files from an existing deployment that aren't found in the .zip file are deleted from your function app.

Function app folder structure

The code for all the functions in a specific function app is located in a root folder that contains a host configuration file and one or more subfolders. Each subfolder contains the code for a separate function, as in the following example:

```
wwwroot
| - host.json
| - mynodefunction
| | - function.json
| | | - index.js
| | | - node_modules
| | | | - ... packages ...
| | | - package.json
| | - mycsharpfunction
| | | - function.json
| | | - run.csx
```

The host.json file contains some runtime-specific configurations, and sits in the root folder of the function app. For information about settings that are available, see the [host.json reference](#).

Each function has a folder that contains one or more code files, the function.json configuration, and other dependencies.

Download your function app project

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. To download your function app project from the portal:

1. Sign in to the [Azure portal](#), and then go to your function app.
2. On the **Overview** tab, select **Download app content**. Select your download options, and then select **Download**.

The screenshot shows the Azure portal's Overview tab for a function app named 'functions-ggailey777'. The 'Download app content' button is highlighted with a red box. The tab also includes other buttons for Stop, Swap, Restart, Download publish profile, Reset publish credentials, and Delete. Below the buttons, there are sections for Status (Running), Subscription (Visual Studio Enterprise), Resource group (functions-ggailey777), and URL (https://functions-ggailey777.azurewebsites.net). There are also sections for Subscription ID and Location (South Central US) and App Service plan / pricing tier (SouthCentralUSPlan (Consumption)). A 'Configured features' section is shown below, with links to Function app settings and Application settings.

The downloaded .zip file is in the correct format to be republished to your function app by using .zip push deployment.

You can also download a .zip file from a GitHub repository. Keep in mind that 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 by using REST APIs

You can use the [deployment service REST APIs](#) to deploy the .zip file to your app in Azure. Just 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](#).

The following example uses the cURL tool to send a request that contains the .zip file. You can run cURL from the terminal on a Mac or Linux computer or by using Bash on Windows. Replace the `<zip_file_path>` placeholder with the path to the location of your project .zip file. Also replace `<app_name>` with the unique name of your app.

Replace the `<deployment_user>` placeholder with the username of your deployment credentials. When prompted by cURL, type in the password. To learn how to set deployment credentials for your app, see [Set and reset user-level credentials](#).

```
curl 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
```

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](#)

Automate resource deployment for your function app in Azure Functions

1/19/2018 • 5 min to read • [Edit Online](#)

You can use an Azure Resource Manager template to deploy a function app. This article outlines the required resources and parameters for doing so. You might need to deploy additional resources, depending on the [triggers and bindings](#) in your function app.

For more information about creating templates, see [Authoring Azure Resource Manager templates](#).

For sample templates, see:

- [Function app on Consumption plan](#)
- [Function app on Azure App Service plan](#)

Required resources

A function app requires these resources:

- An [Azure Storage](#) account
- A hosting plan (Consumption plan or App Service plan)
- A function 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": "2015-06-15",  
    "location": "[resourceGroup().location]",  
    "properties": {  
        "accountType": "[parameters('storageAccountType')]"  
    }  
}
```

In addition, the properties `AzureWebJobsStorage` and `AzureWebJobsDashboard` must be specified as app settings in the site configuration. The Azure Functions runtime uses the `AzureWebJobsStorage` connection string to create internal queues. The connection string `AzureWebJobsDashboard` is used 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)]"
  }
]

```

Hosting plan

The definition of the hosting plan varies, depending on whether you use a Consumption or App Service plan. See [Deploy a function app on the Consumption plan](#) and [Deploy a function app on the App Service plan](#).

Function app

The function app resource is defined by using a resource of type **Microsoft.Web/Site** and kind **functionapp**:

```

{
  "apiVersion": "2015-08-01",
  "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'))]"
  ]
}

```

Deploy a function app on the Consumption plan

You can run a function app in two different modes: the Consumption plan and the App Service 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. So, you don't have to pay for idle VMs, and you don't have to reserve capacity in advance. To learn more about hosting plans, see [Azure Functions Consumption and App Service plans](#).

For a sample Azure Resource Manager template, see [Function app on Consumption plan](#).

Create a Consumption plan

A Consumption plan is a special type of "serverfarm" resource. You 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"
  }
}

```

Create a function app

In addition, 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": "2015-08-01",  
    "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": "AzureWebJobsDashboard",  
                    "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',  
variables('storageAccountName'), ';AccountKey=', listKeys(variables('storageAccountid'), '2015-05-01-  
preview').key1)]"  
                },  
                {  
                    "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_EXTENSION_VERSION",  
                    "value": "~1"  
                }  
            ]  
        }  
    }  
}
```

Deploy a function app on the 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

```
{  
  "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  
  }  
}
```

Create a function app

After you've selected a scaling option, create a function app. The app is the container that holds all your functions.

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": "~1" },
        { "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)]"
      }
    },
    {
      "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 `azuredetect.json` file in GitHub.

Here is an example that uses markdown:

```
[![Deploy to Azure](http://azuredetect.net/deploybutton.png)]  
(https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-to-azuredetect-json>)
```

Here is an example that uses HTML:

```
<a href="https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-to-azuredetect-json>"  
target="_blank"></a>
```

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

12/5/2017 • 4 min to read • [Edit Online](#)

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

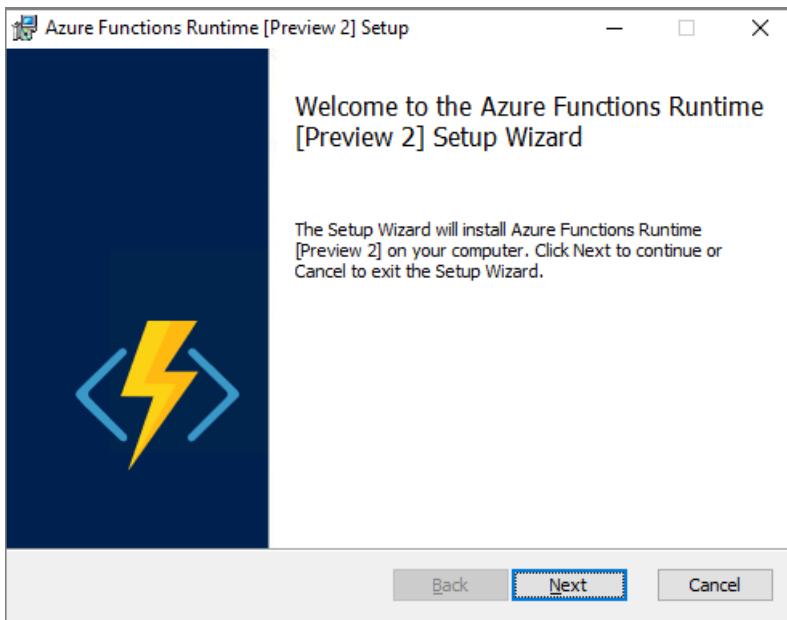
If you have previously installed the Azure Functions Runtime preview, you must uninstall before installing the latest release. Uninstall the Azure Functions Runtime preview by removing the program in Add/Remove Programs in Windows.

Install the Azure Functions Runtime Preview

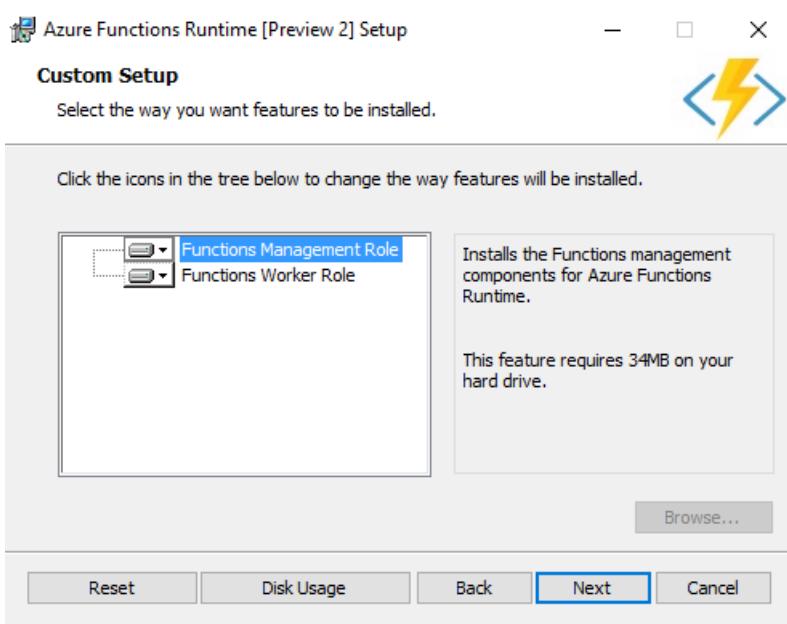
The Azure Functions Runtime Preview Installer guides you through the installation of the Azure Functions Runtime preview Management and Worker Roles. It is possible to install the Management and Worker role on the same machine. However, as you add more function apps, you must deploy more worker roles 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.



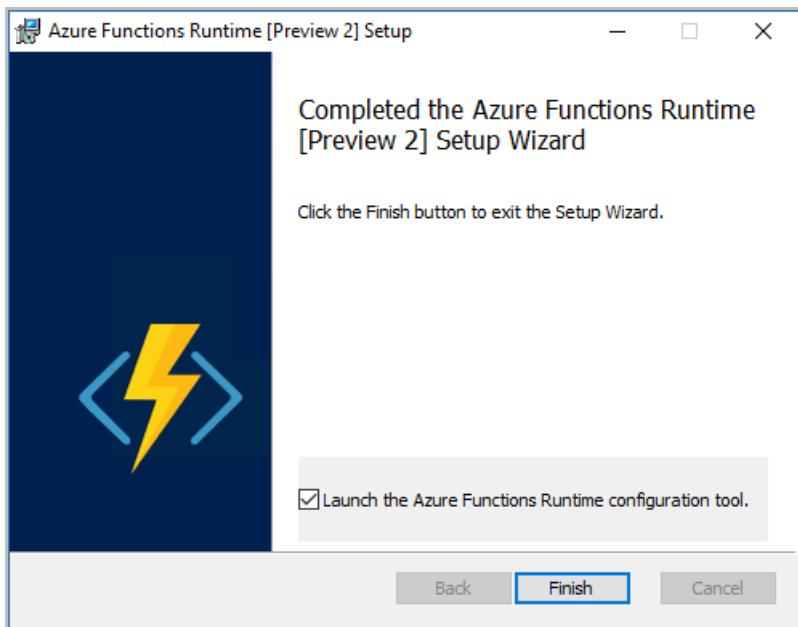
2. Click **Next**.
3. Once you have read the terms of the **EULA**, **check the box** to accept the terms and click **Next** to advance.
4. Select the roles you want to install on this machine **Functions Management Role** and/or **Functions Worker Role** and click **Next**.



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.



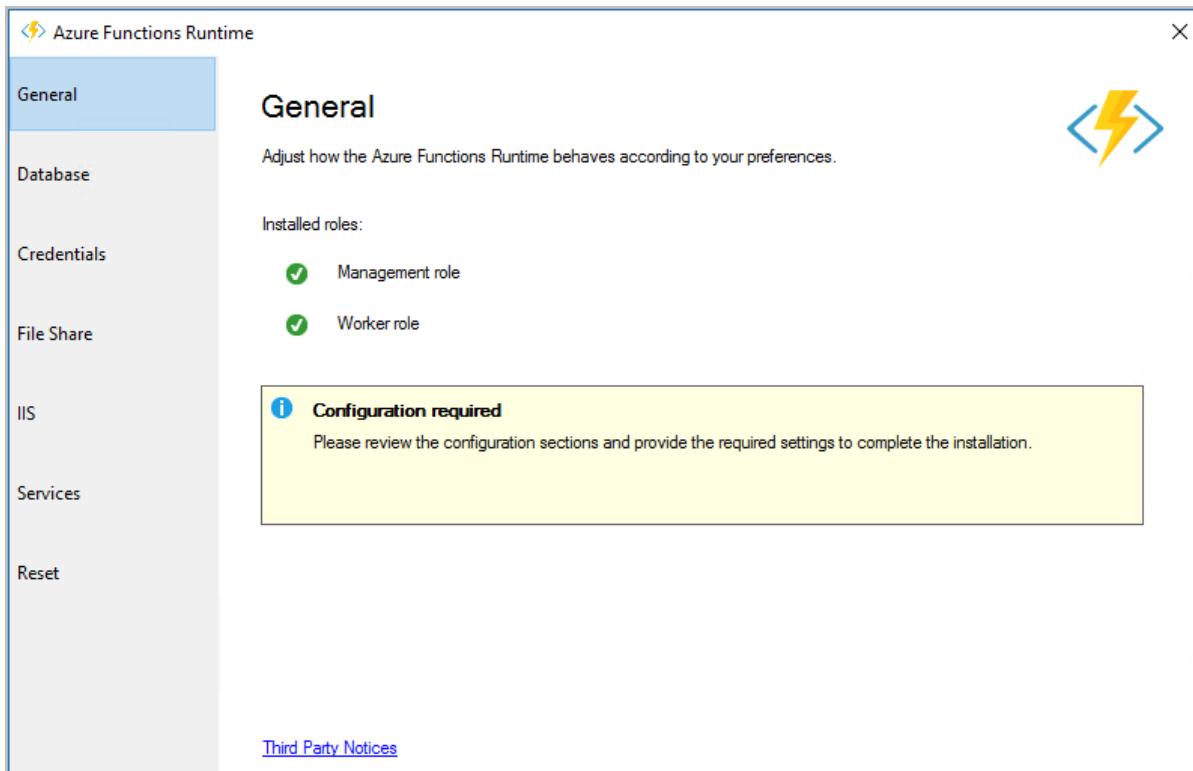
NOTE

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.

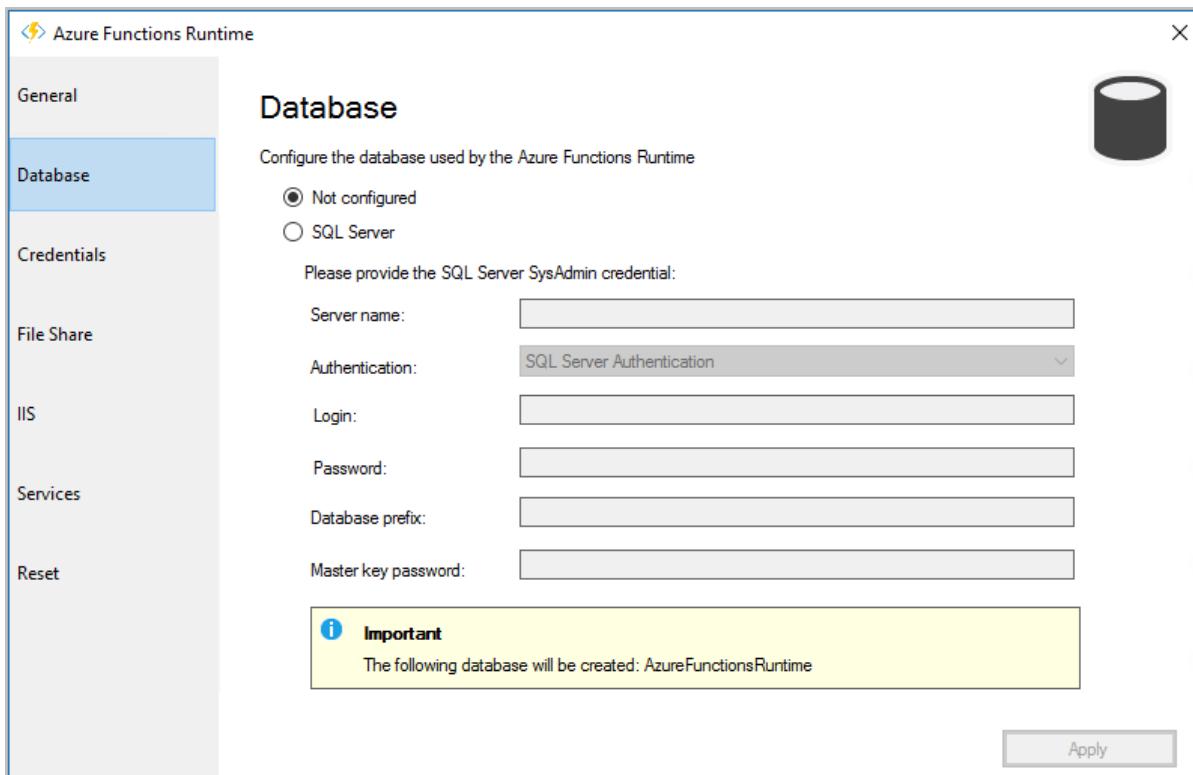
Configure the Azure Functions Runtime

To complete the Azure Functions Runtime installation, you must complete the configuration.

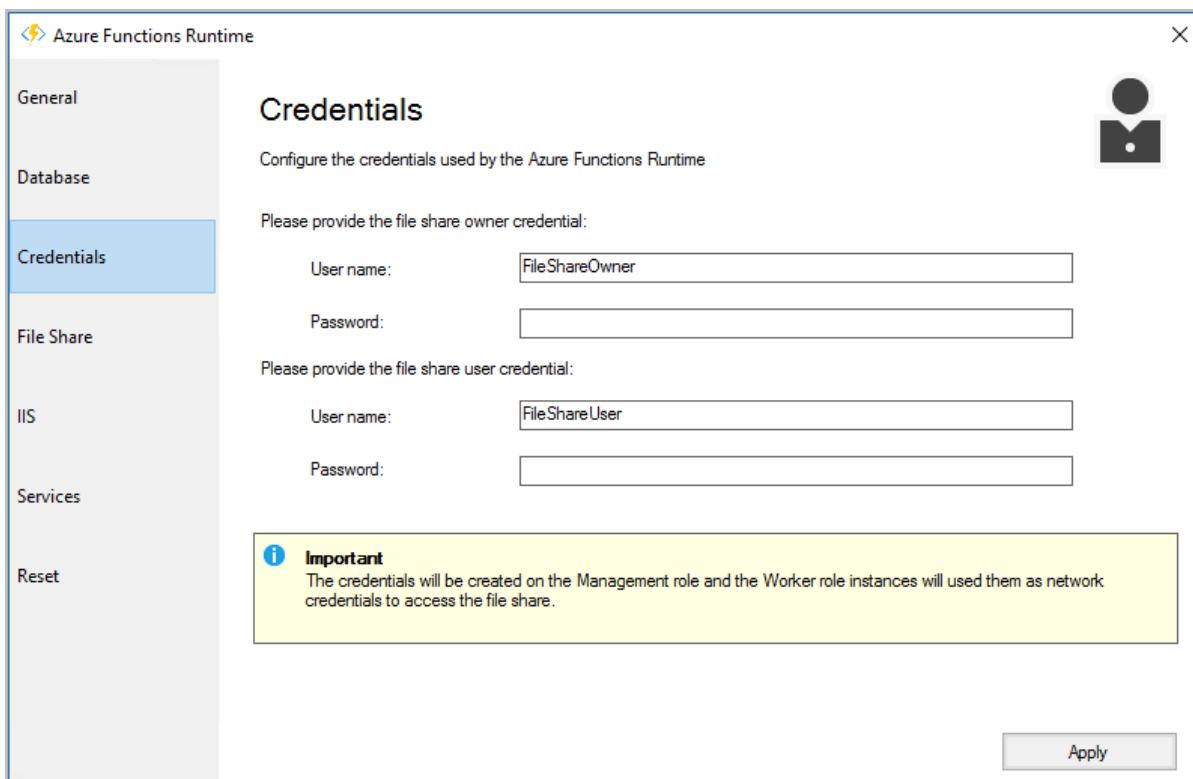
1. The **Azure Functions Runtime** configuration tool shows which roles are installed on your machine.



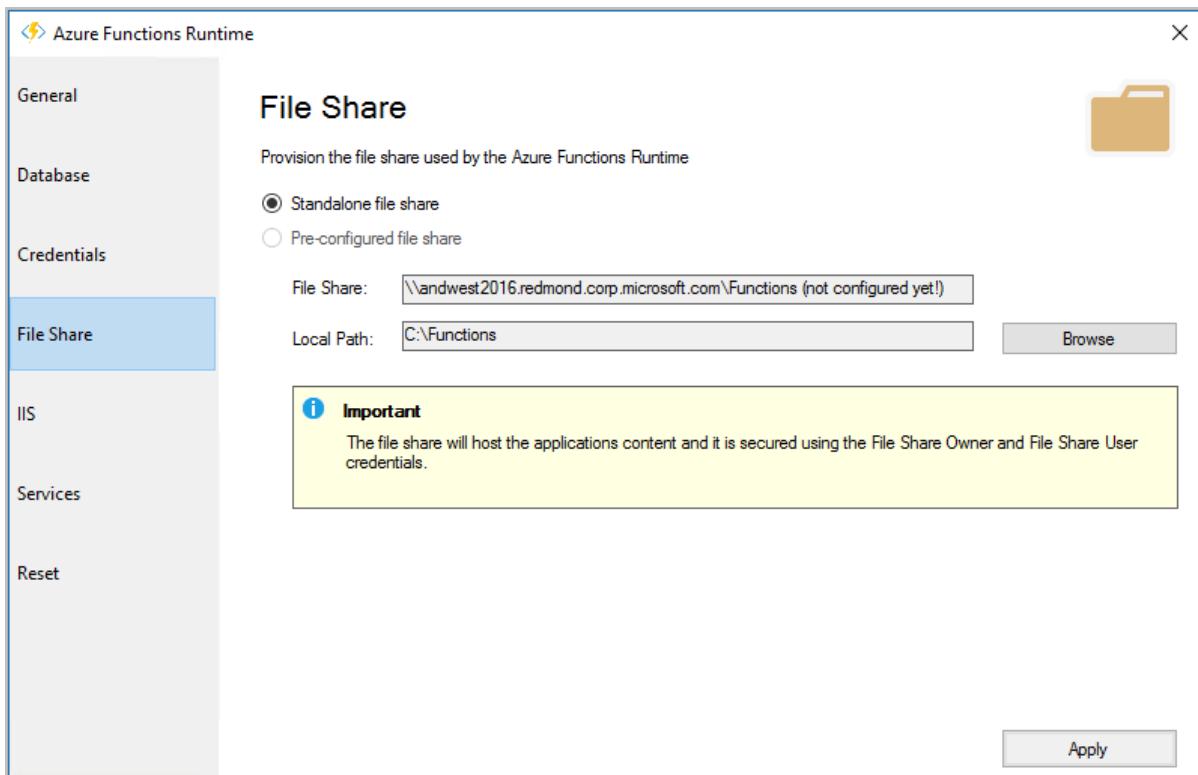
2. Click the **Database** tab, enter the connection details for your SQL Server instance, including specifying a **Database master key**, and click **Apply**. Connectivity to a SQL Server instance is required in order for the Azure Functions Runtime to create a database to support the Runtime.



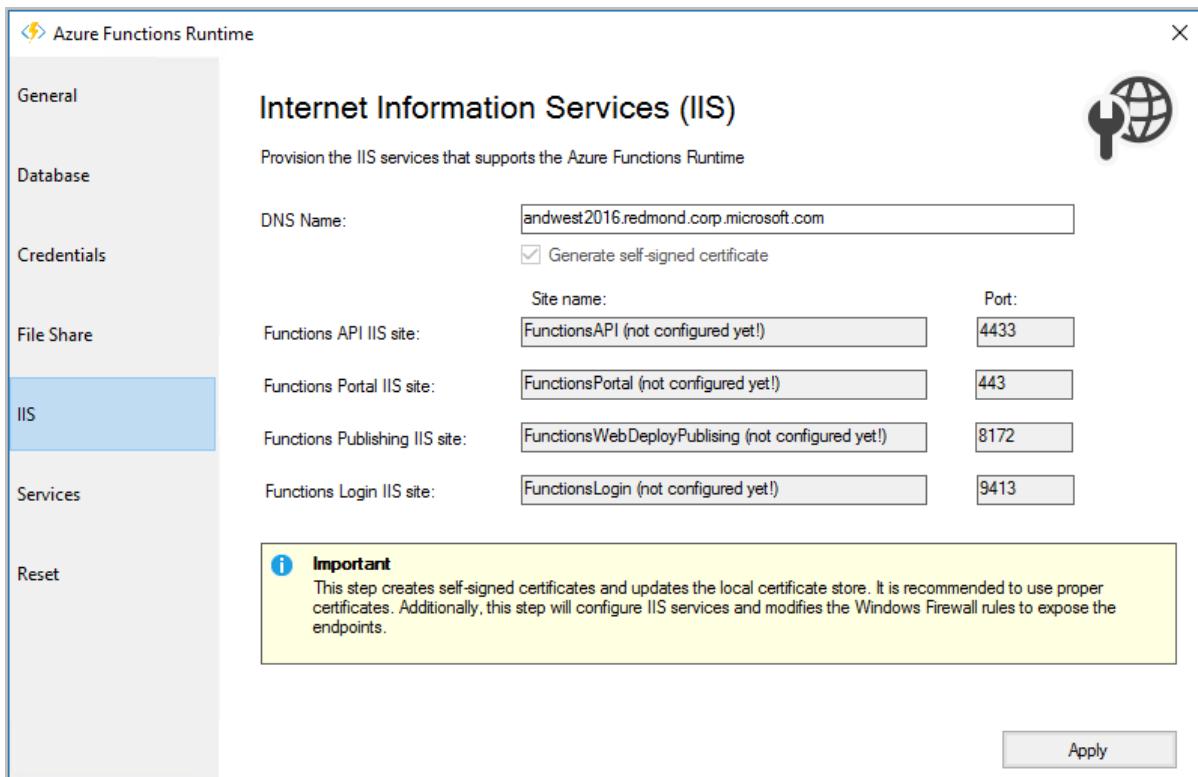
- Click the **Credentials** tab. Here, you must create two new credentials for use with a file share for hosting all your function apps. Specify **User name** and **Password** combinations for the **file share owner** and for the **file share user**, then click **Apply**.



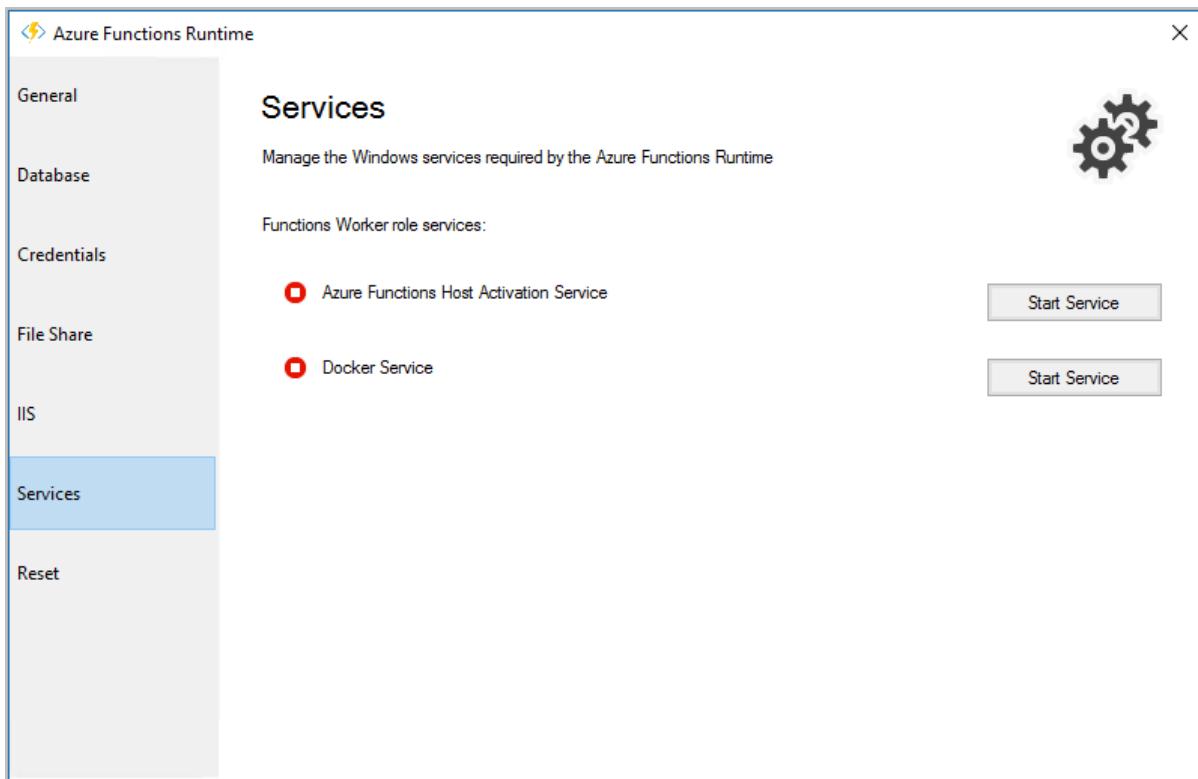
- Click the **File Share** tab. Here you must specify the details of the file share location. The file share can be created for you or you can use an existing File Share and click **Apply**. If you select a new File Share location, you must specify a directory for use by the Azure Functions Runtime.



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 has a search bar, filter options for 'Location' (All locations), 'Resource Group' (All resource groups), and 'No grouping', and sorting columns for 'NAME', 'SUBSCRIPTION ID', 'RESOURCE GROUP', and 'LOCATION'. Below this, there's a large lightning bolt icon and the text 'No function apps to display'. A descriptive message follows: 'Azure Functions are an event-based serverless compute experience to accelerate your development. Scale based on demand and pay only for the resources you consume.' At the bottom, there's a link 'Learn more about azure functions'.

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://. 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

Azure Function Runtime Portal

Login

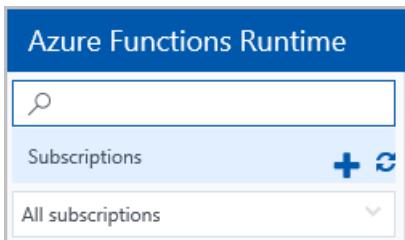
DOMAIN\Username

Password

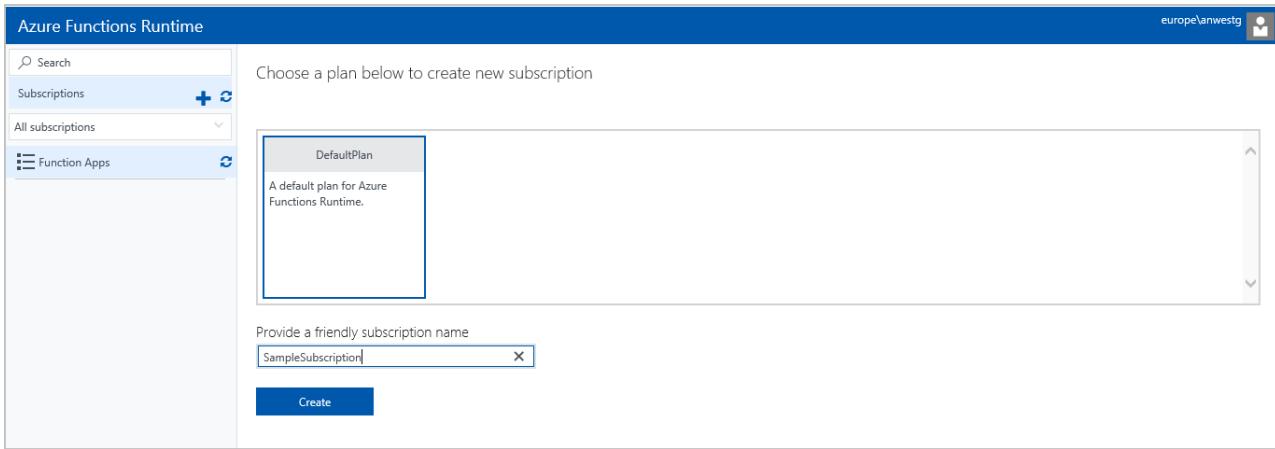
Sign in

Copyright © 2017. All Rights Reserved

1. 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



1. Choose **DefaultPlan**, enter a name for your Subscription, and click **Create**.



1. 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.
2. 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

- Your new function app is listed in the left-hand pane of the portal. Select Functions and then click **New Function** at the top of the center pane in the portal.

Azure Functions Runtime

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

Subscriptions

All subscriptions

Function Apps

TestFunction

Functions

TimeTriggerCSharp1

Integrate

Manage

TestFunctionV1

Functions

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

- 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**

- 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.

The screenshot shows the Azure Functions Runtime portal interface. On the left, there's a sidebar with 'Subscriptions' (All subscriptions), 'Function Apps', 'TestFunction', and 'Functions' (TimerTriggerCSharp1 selected). Below that are 'Integrate' and 'Manage' buttons. The main area has tabs for 'run.csx' (selected) and 'Save'. The code editor contains the following C# code:

```
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 }
```

Below the code editor is a 'Logs' section with a scrollable list of log entries:

```
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-a7deddfcb02, 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)
```

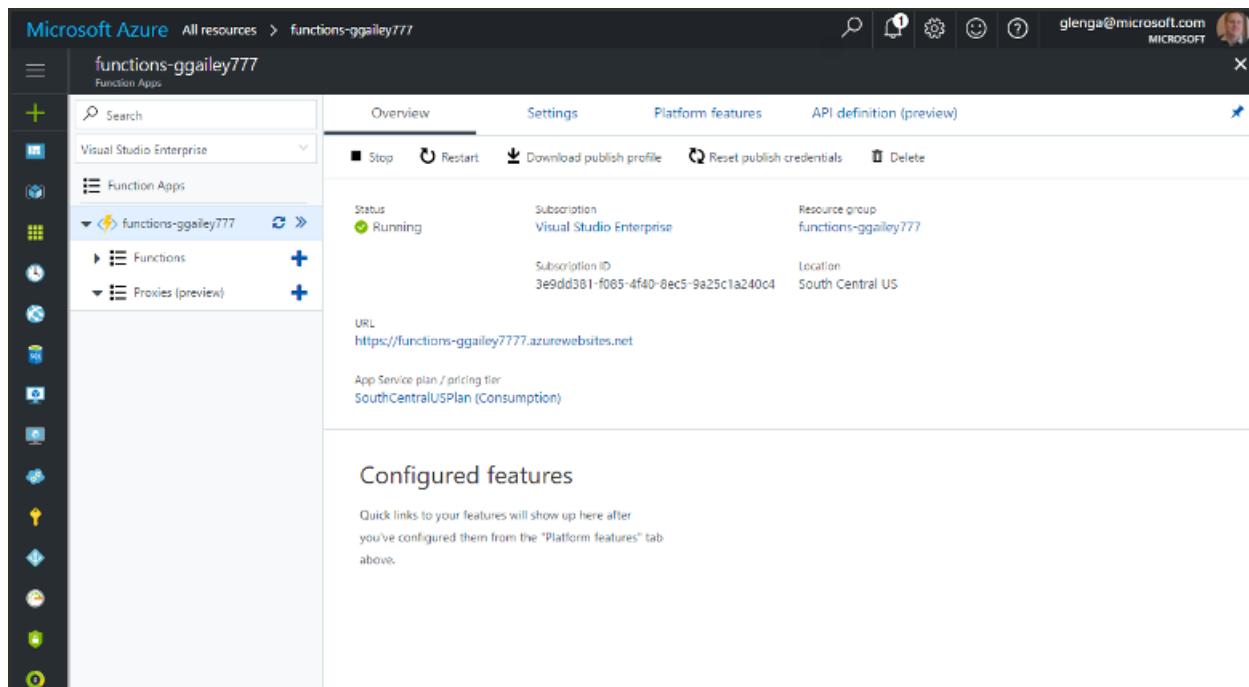
On the right side of the logs panel, there are buttons for 'Pause', 'Clear', 'Copy logs', 'Expand', and a dropdown menu. A vertical 'View files' pane is visible on the far right.

How to manage a function app in the Azure portal

1/12/2018 • 4 min to read • [Edit Online](#)

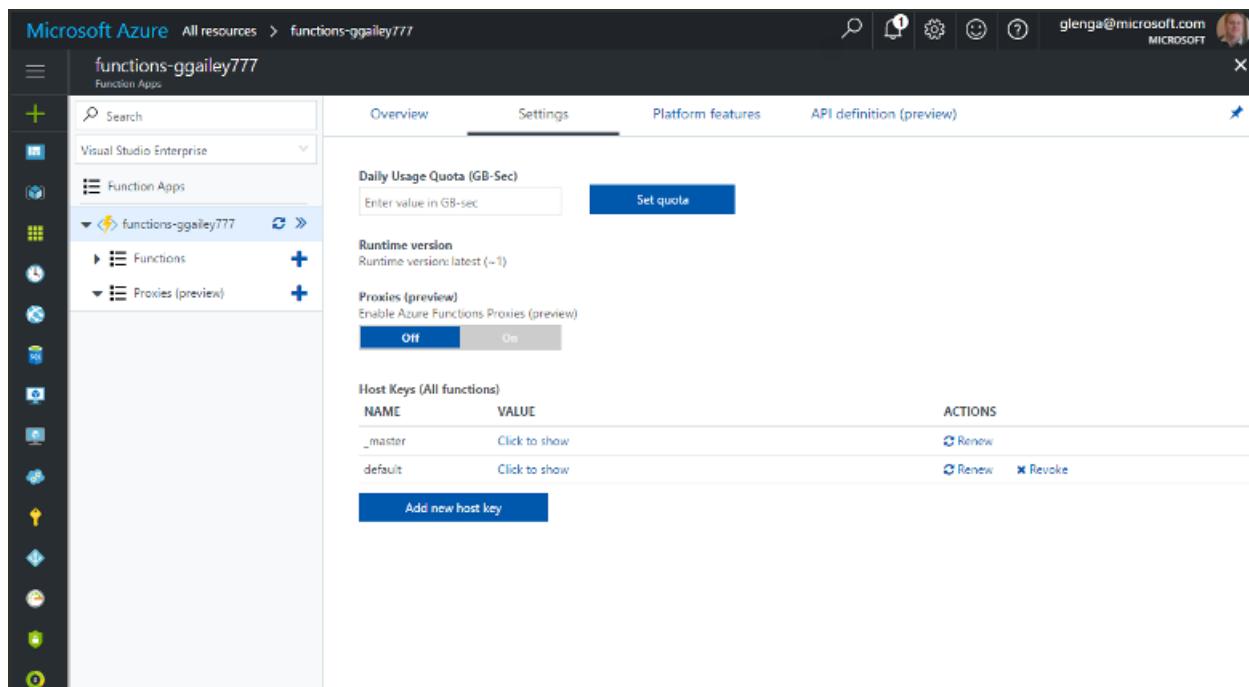
In Azure Functions, a function app provides the execution context for your individual functions. Function app behaviors apply to all functions hosted by a given function app. This topic describes how to configure and manage your function apps in the Azure portal.

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 Microsoft Azure portal interface. The top navigation bar includes 'Microsoft Azure', 'All resources > functions-ggalley777', a search bar, and user profile information ('glenga@microsoft.com MICROSOFT'). The left sidebar lists 'functions-ggalley777' under 'Function Apps' and includes icons for 'Search', 'Visual Studio Enterprise', 'Functions', and 'Proxies (preview)'. The main content 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-ggalley777), 'Subscription ID' (3e9dd381-f085-4f40-8ec5-9a25c1a240c4), 'Location' (South Central US), and 'URL' (https://functions-ggalley777.azurewebsites.net). Below this, it says 'App Service plan / pricing tier' (SouthCentralUSPlan (Consumption)). A 'Configured features' section is present but currently empty.

Function app settings tab

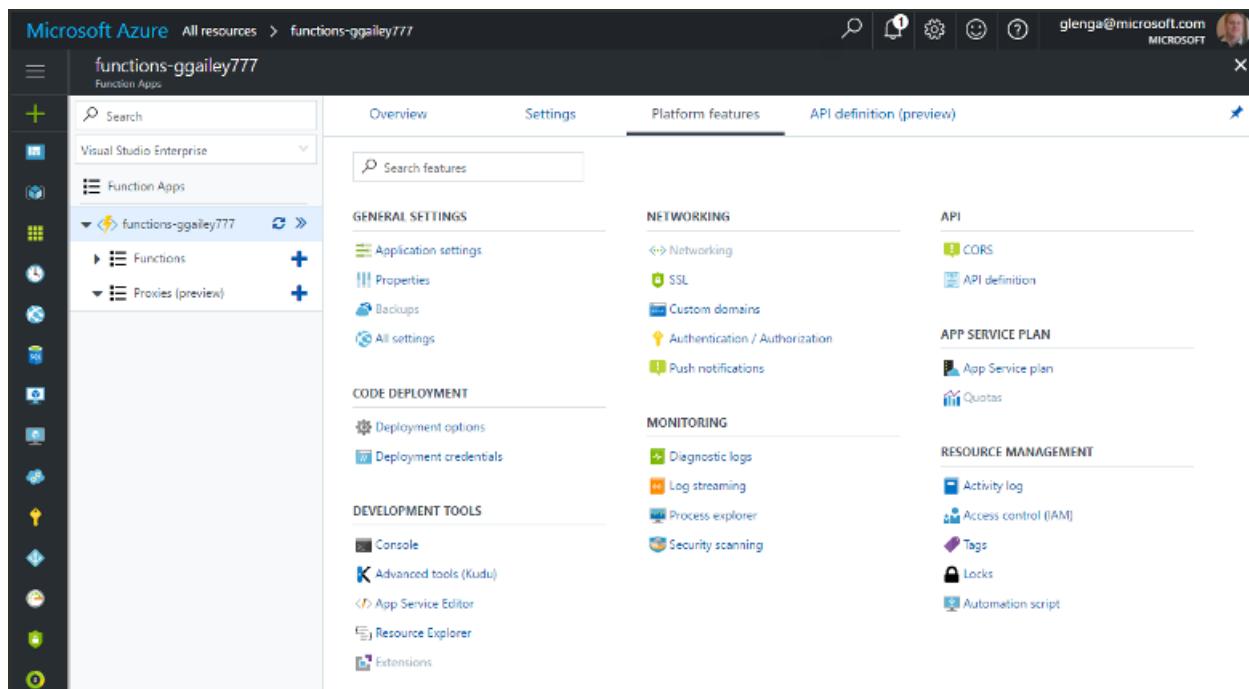


The screenshot shows the 'Settings' tab of the Azure portal for the 'functions-ggalley777' function app. The left sidebar is identical to the previous screenshot. The main content area has tabs for 'Overview', 'Settings' (which is selected), 'Platform features', and 'API definition (preview)'. Under 'Settings', there is a 'Daily Usage Quota (GB-Sec)' input field with a placeholder 'Enter value in GB-sec.' and a 'Set quote' button. Below that, 'Runtime version' is listed as 'Runtime version: latest (~1)'. The 'Proxies (preview)' section shows a toggle switch that is currently off. The 'Host Keys (All functions)' table lists two entries: '_master' and 'default', both with a 'Click to show' link. There are also 'Renew' and 'Revoke' buttons for each entry. At the bottom of the table is a 'Add new host key' button.

The **Settings** tab is where you can update the Functions runtime version used by your function app. It is also where you manage the host keys used to restrict HTTP access to all functions hosted by the function app.

Functions supports both Consumption hosting and App Service hosting plans. For more information, see [Choose the correct service plan for Azure Functions](#). For better predictability in the Consumption plan, Functions lets you limit platform usage by setting a daily usage quota, in gigabytes-seconds. Once the daily usage quota is reached, the function app is stopped. A function app stopped as a result of reaching the spending quota can be re-enabled from the same context as establishing the daily spending quota. See the [Azure Functions pricing page](#) for details on billing.

Platform features tab

A screenshot of the Microsoft Azure portal interface. The top navigation bar shows 'Microsoft Azure All resources > functions-ggailey777 Function Apps'. The user's email 'glenga@microsoft.com' and profile picture are in the top right. Below the navigation is a search bar and several icons. The main content area has tabs: 'Overview', 'Settings', 'Platform features' (which is selected), and 'API definition (preview)'. On the left is a sidebar with a tree view: 'Visual Studio Enterprise' > 'Function Apps' > 'functions-ggailey777' > 'Functions' > 'Proxies (preview)'. The main pane is divided into sections: 'GENERAL SETTINGS' (Application settings, Properties, Backups, All settings), 'CODE DEPLOYMENT' (Deployment options, Deployment credentials), 'DEVELOPMENT TOOLS' (Console, Advanced tools (Kudu), App Service Editor, Resource Explorer, Extensions), 'NETWORKING' (Networking, SSL, Custom domains, Authentication / Authorization, Push notifications), 'MONITORING' (Diagnostic logs, Log streaming, Process explorer, Security scanning), 'API' (CORS, API definition), 'APP SERVICE PLAN' (App Service plan, Quotas), and 'RESOURCE MANAGEMENT' (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](#)
- [Application settings](#)
- [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 Azure 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):

```
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 };
```

Application settings



The App Service **Application settings** blade is where you configure and manage framework versions, remote debugging, app settings, and connection strings. When you integrate your function app with other Azure and third-party services, you can modify those settings here. To delete a setting, scroll to the right and select the X icon at the right end of the line (not shown in the following image).

 Application settings

functions-ggailey777

Save Discard

Auto swap destinations cannot be configured from production slot

Auto Swap

Auto Swap Slot

Debugging

Remote debugging

Remote Visual Studio version

App settings

AzureWebJobsDashboard	DefaultEndpointsProtocol=h...	<input type="checkbox"/> Slot setting	...
AzureWebJobsStorage	DefaultEndpointsProtocol=h...	<input type="checkbox"/> Slot setting	...
FUNCTIONS_EXTENSION_VE...	~1	<input type="checkbox"/> Slot setting	...
WEBSITE_CONTENTAZUREF...	DefaultEndpointsProtocol=h...	<input type="checkbox"/> Slot setting	...
WEBSITE_CONTENTSHARE	functions-ggailey7777b1c81...	<input type="checkbox"/> Slot setting	...
WEBSITE_NODE_DEFAULT_V...	6.5.0	<input type="checkbox"/> Slot setting	...
<input type="text" value="Key"/>	<input type="text" value="Value"/>	<input type="checkbox"/> Slot setting	...

Connection strings

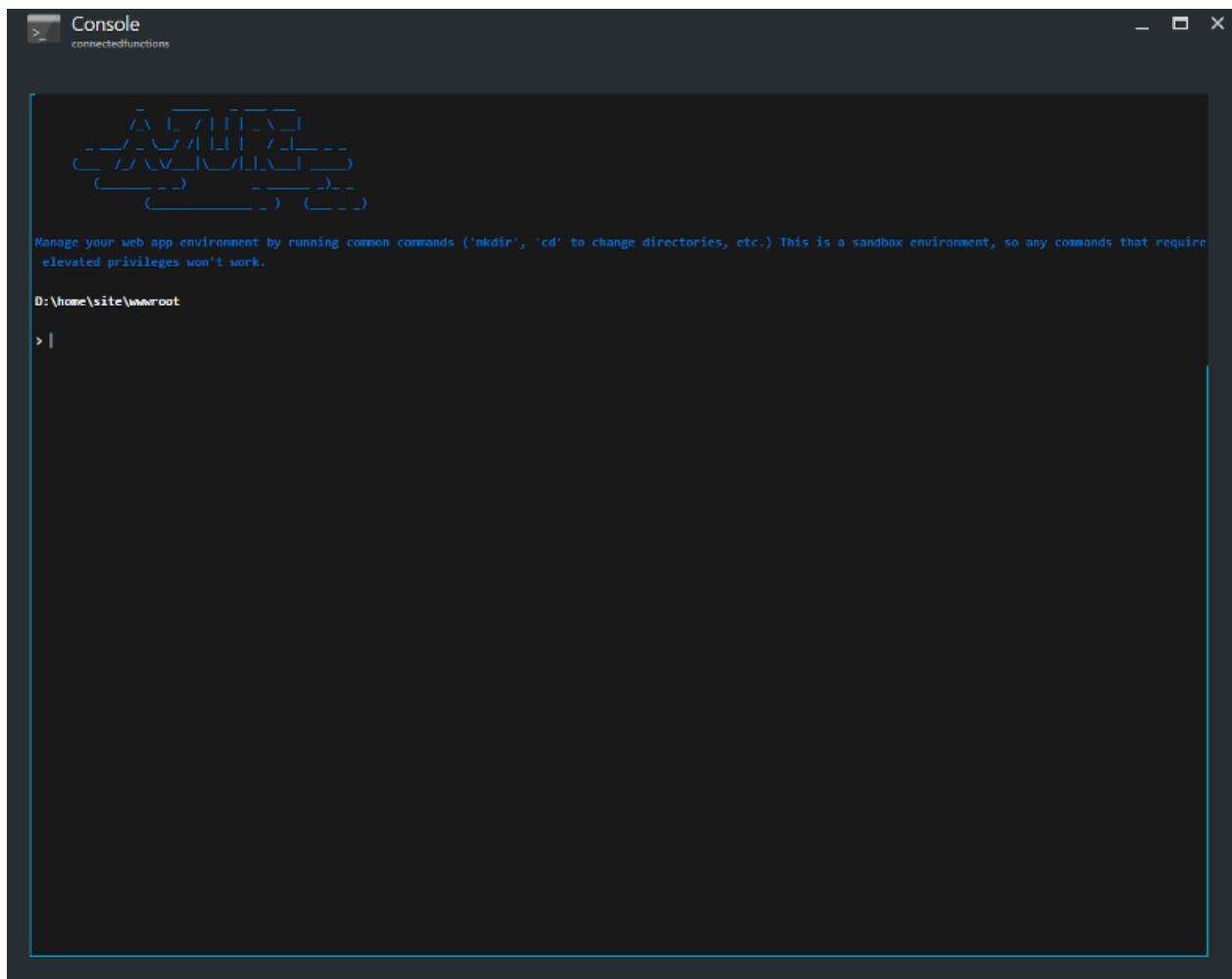
No results

<input type="text" value="Name"/>	<input type="text" value="Value"/>	<input type="button" value="SQL Database"/> <input type="button" value="▼"/>	<input type="checkbox"/> Slot setting	...
-----------------------------------	------------------------------------	--	---------------------------------------	-----

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 sub-path "connectedfunctions". The window contains a stylized tree icon at the top. Below it, a message reads: "Manage your web app environment by running common commands ('mkdir', 'cd' to change directories, etc.) This is a sandbox environment, so any commands that require elevated privileges won't work." The command line shows the path "D:\home\site\wwwroot" followed by a prompt "> |".

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, VSTS, 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 shared...
wl.emails	Read access to a user's personal, preferred, and business email address...
wl.events_create	Read access to Microsoft OneDrive files that other users have shared...
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 numbers...

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 [Get Started with API Apps and Swagger in Azure](#). You can also use Functions Proxies to define a single API surface for multiple functions. For more information, see [Working with Azure Functions Proxies](#).

The screenshot shows the Azure portal's 'API Definition' blade. At the top, there's a title bar with the 'API Definition' icon and the function name 'ConnectedFunctions'. Below the title bar are 'Save' and 'Discard' buttons. A large text area contains an informational message: 'API definition lets you configure the location of the Swagger 2.0 metadata describing your API. This makes it easy for others to discover and consume your API. Note: the URL can be a relative or absolute path, but must be publicly accessible.' Below this message is a text input field with the placeholder 'URL of API definition (e.g., http://www.yoursite.com/apidefinition.json)'. The overall interface is clean and modern, typical of the Azure developer portal.

Next steps

- [Configure Azure App Service Settings](#)
- [Continuous deployment for Azure Functions](#)

How to target Azure Functions runtime versions

11/29/2017 • 4 min 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. This article explains how to choose which major version to use and 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](#).

Differences between runtime 1.x and 2.x

IMPORTANT

Runtime 1.x is the only version approved for production use.

RUNTIME	STATUS
1.x	Generally Available (GA)
2.x	Preview

The following sections explain differences in languages, bindings, and cross-platform development support.

Languages

The following table indicates which programming languages are supported in each runtime version.

LANGUAGE	1.X	2.X
C#	GA	Preview
JavaScript	GA	Preview
F#	GA	
Java		Preview
Python	Experimental	
PHP	Experimental	
TypeScript	Experimental	
Batch (.cmd, .bat)	Experimental	
Bash	Experimental	
PowerShell	Experimental	

For information about planned changes to language support, see [Azure roadmap](#).

For more information, see [Supported languages](#).

Bindings

Runtime 2.x lets you create custom [binding extensions](#). Built-in bindings that use this extensibility model are only available in 2.x; among the first of these is the [Microsoft Graph bindings](#).

The following table shows the bindings that are supported in the two major versions of the Azure Functions runtime.

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
Blob Storage	✓	✓	✓	✓	✓
Cosmos DB	✓	✓ ¹	✓	✓	✓
Event Hubs	✓	✓	✓		✓
External File ²	✓			✓	✓
External Table ²	✓			✓	✓
HTTP	✓	✓	✓		✓
Microsoft Graph Excel tables		✓ ¹		✓	✓
Microsoft Graph OneDrive files		✓ ¹		✓	✓
Microsoft Graph Outlook email		✓ ¹			✓
Microsoft Graph Events		✓ ¹	✓	✓	✓
Microsoft Graph Auth tokens		✓ ¹		✓	
Mobile Apps	✓	✓ ¹		✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
SendGrid	✓	✓ ¹			✓
Service Bus	✓	✓ ¹	✓		✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓ ¹			✓

TYPE	1.X	2.X	TRIGGER	INPUT	OUTPUT
Webhooks	✓		✓		✓

¹ Must be registered as a binding extension in 2.x. See [Known issues in 2.x](#).

² Experimental — not supported and might be abandoned in the future.

For more information about bindings support and other functional gaps in 2.x, see [Runtime 2.0 known issues](#).

Cross-platform development

Runtime 1.x supports function development only in the portal or on Windows; with 2.x you can develop and run Azure Functions on Linux or macOS.

Automatic and manual version updates

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 ("~1" for 1.x or "beta" for 2.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, "1.0.11360"), the function app is kept on that version until you explicitly change it.

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 "beta" for 2.x.

View the current runtime version

Use the following procedure to view the runtime version currently used by a function app.

1. In the [Azure portal](#), navigate to the function app, and under **Configured Features**, choose **Function app settings**.

Microsoft Azure functions-ggailey777

functions-ggailey777 Function Apps

Overview Platform features

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)

Configured features

Function app settings Application settings CORS Rules (6 defined)

2. 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 FUNCTIONS_EXTENSION_VERSION app setting is set to `~1`.

Microsoft Azure functions-ggailey777

functions-ggailey777 Function Apps

Overview Platform features Function app settings

Daily Usage Quota (GB-Sec) Enter value in GB-sec Set quota

Application settings Manage application settings

Runtime version Runtime version: 1.0.11296.0 (~1)

~1 beta

Proxies (preview) Enable Azure Functions Proxies (preview)

Off On

Function app edit mode Change the edit mode of your function app

Read/Write Read Only

Target the version 2.0 runtime

IMPORTANT

Azure Functions runtime 2.0 is in preview and currently not all features of Azure Functions are supported. For more information, see [Differences between runtime 1.x and 2.x](#) earlier in this article.

You can move a function app to the runtime version 2.0 preview in the Azure portal. In the **Function app settings** tab, choose **beta** under **Runtime version**.

The screenshot shows the Azure portal interface for a function app named 'functions-ggailey777'. The 'Function app settings' tab is selected. In the 'Runtime version' section, the current value '1.0.11296.0 (~1)' is shown next to a 'beta' button, which is highlighted with a red box. Other settings visible include 'Daily Usage Quota (GB-Sec)', 'Proxies (preview)', and 'Function app edit mode'.

This setting is equivalent to setting the `FUNCTIONS_EXTENSION_VERSION` application setting to `beta`. Choose the `~1` button to move back to the current publicly supported major version. You can also use the Azure CLI to update this application setting.

Target a version using the portal

When you need to target a version other than the current major version or version 2.0, you must set the `FUNCTIONS_EXTENSION_VERSION` application setting.

1. In the [Azure portal](#), navigate to your function app, and under **Configured Features**, choose **Application settings**.

The screenshot shows the Azure portal interface for the same function app. The 'Overview' tab is selected. In the 'Configured features' section, the 'Application settings' tab is highlighted with a red box. Other tabs shown include 'Function app settings' and 'CORS Rules (6 defined)'.

2. In the **Application settings** tab, find the `FUNCTIONS_EXTENSION_VERSION` setting and change the value to a valid version of the 1.x runtime or `beta` for version 2.0.

The screenshot shows the Azure portal interface for managing a function app named 'functions-ggailey777'. On the left, there's a sidebar with various icons and a search bar. The main area has tabs for 'Overview' and 'Platform features', with 'Application settings' selected. A red box highlights the 'Save' button at the top right of the application settings table. Another red box highlights the 'FUNCTIONS_EXTENSION_VERSION' setting, which is currently set to 'beta'. The table also lists other settings like 'AzureWebJobsDashboard', 'AzureWebJobsStorage', and 'WEBSITE_NODE_DEFAULT_VERSION'.

3. Click **Save** to save the application setting update.

Target a version using Azure CLI

You can also set the `FUNCTIONS_EXTENSION_VERSION` from the Azure CLI. Using the Azure CLI, update the application 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>
```

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. Replace `<version>` with a valid version of the 1.x runtime or `beta` for version 2.0.

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](#)

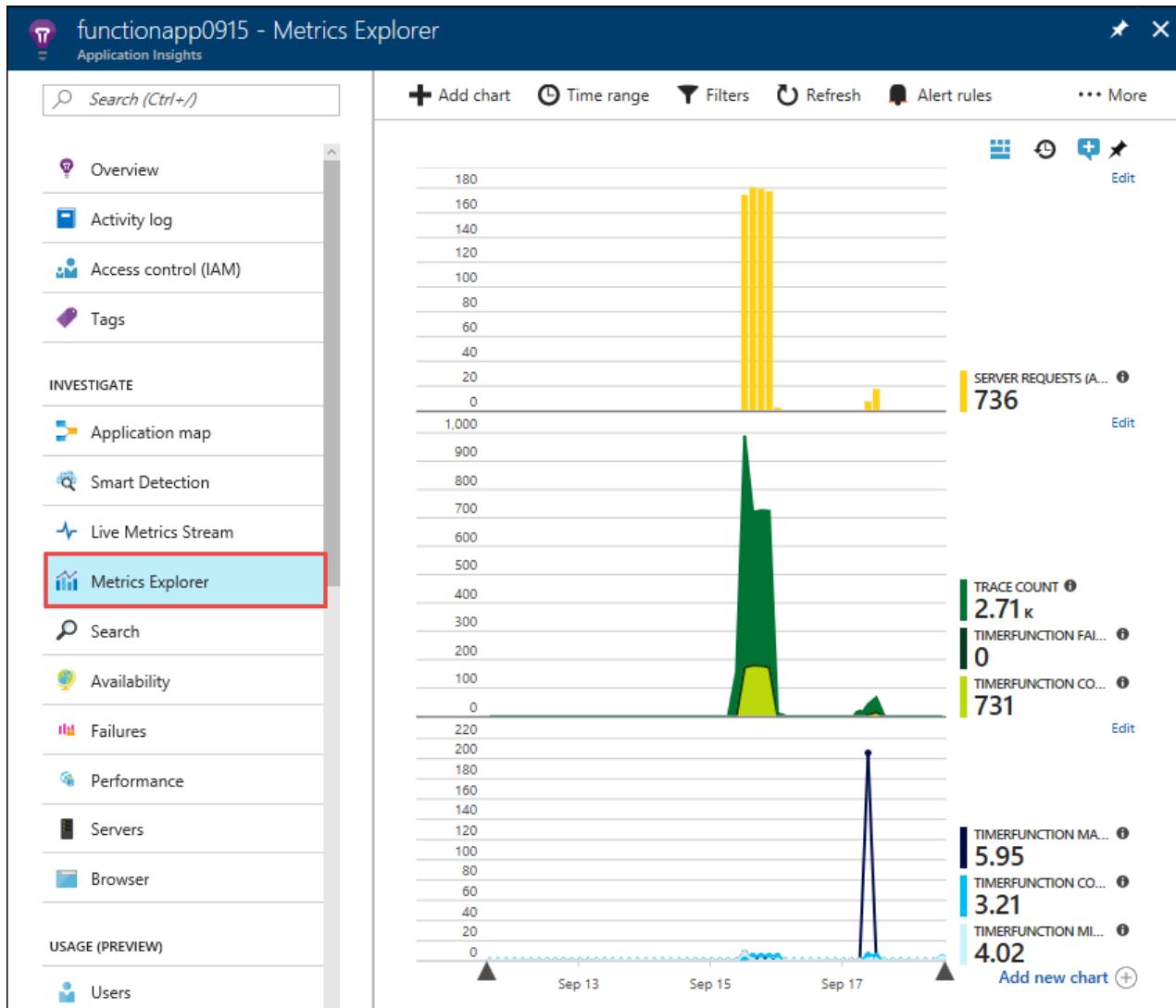
[See Release notes for runtime versions](#)

Monitor Azure Functions

1/12/2018 • 13 min to read • [Edit Online](#)

Overview

Azure Functions offers built-in integration with [Azure Application Insights](#) for monitoring functions. This article shows how to configure Functions to send telemetry data to Application Insights.



Functions also has built-in monitoring that doesn't use Application Insights. We recommend Application Insights because it offers more data and better ways to analyze the data. For information about the built-in monitoring, see the [last section of this article](#).

Enable Application Insights integration

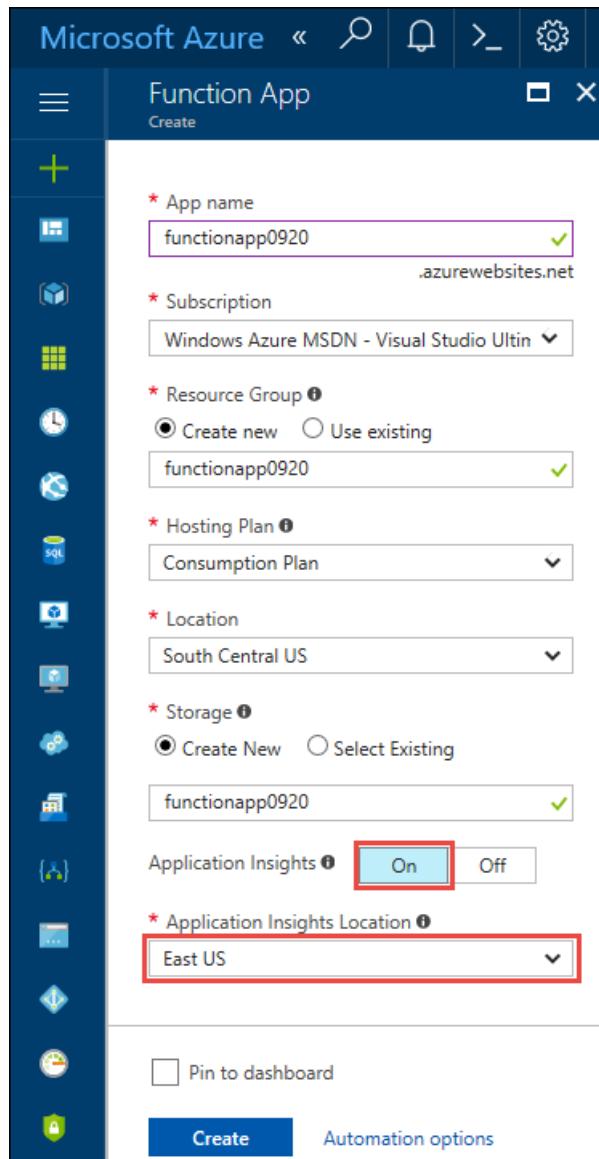
For a function app to send data to Application Insights, it needs to know the instrumentation key of an Application Insights instance. There are two ways to make that connection in the [Azure portal](#):

- [Create a connected Application Insights instance when you create the function app](#).
- [Connect an Application Insights instance to an existing function app](#).

New function app

Enable Application Insights on the Function App **Create** page:

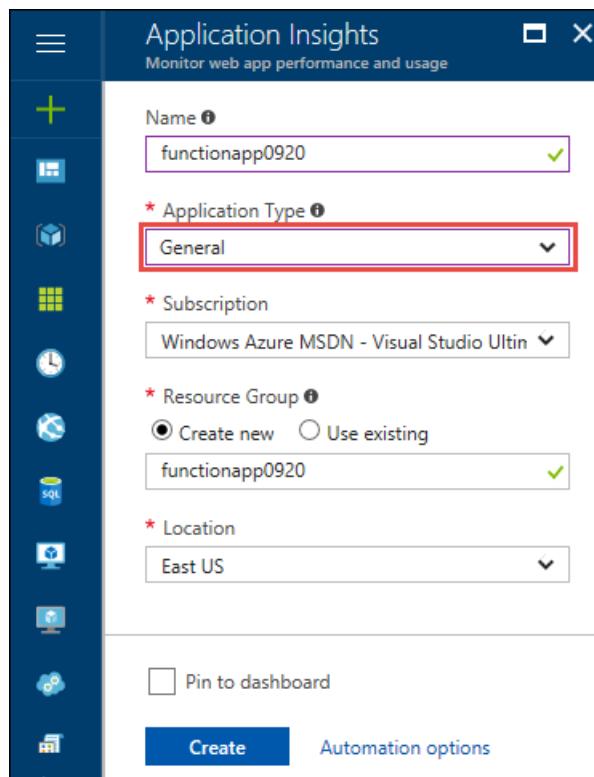
1. Set the **Application Insights** switch **On**.
2. Select an **Application Insights Location**.



Existing function app

Get the instrumentation key and save it in a function app:

1. Create the Application Insights instance. Set application type to **General**.



2. Copy the instrumentation key from the **Essentials** page of the Application Insights instance. Hover over the end of the displayed key value to get a **Click to copy** button.

3. In the function app's **Application settings** page, [add an app setting](#) by clicking **Add new setting**. Name the new setting **APPINSIGHTS_INSTRUMENTATIONKEY** and paste the copied instrumentation key.

4. Click **Save**.

Disable built-in logging

If you enable Application Insights, we recommend that you disable the [built-in logging that uses Azure storage](#). The built-in logging is useful for testing with light workloads but is not intended for high-load production use. For production monitoring, Application Insights is recommended. If built-in logging is used in production, the logging record may be incomplete due to 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](#).

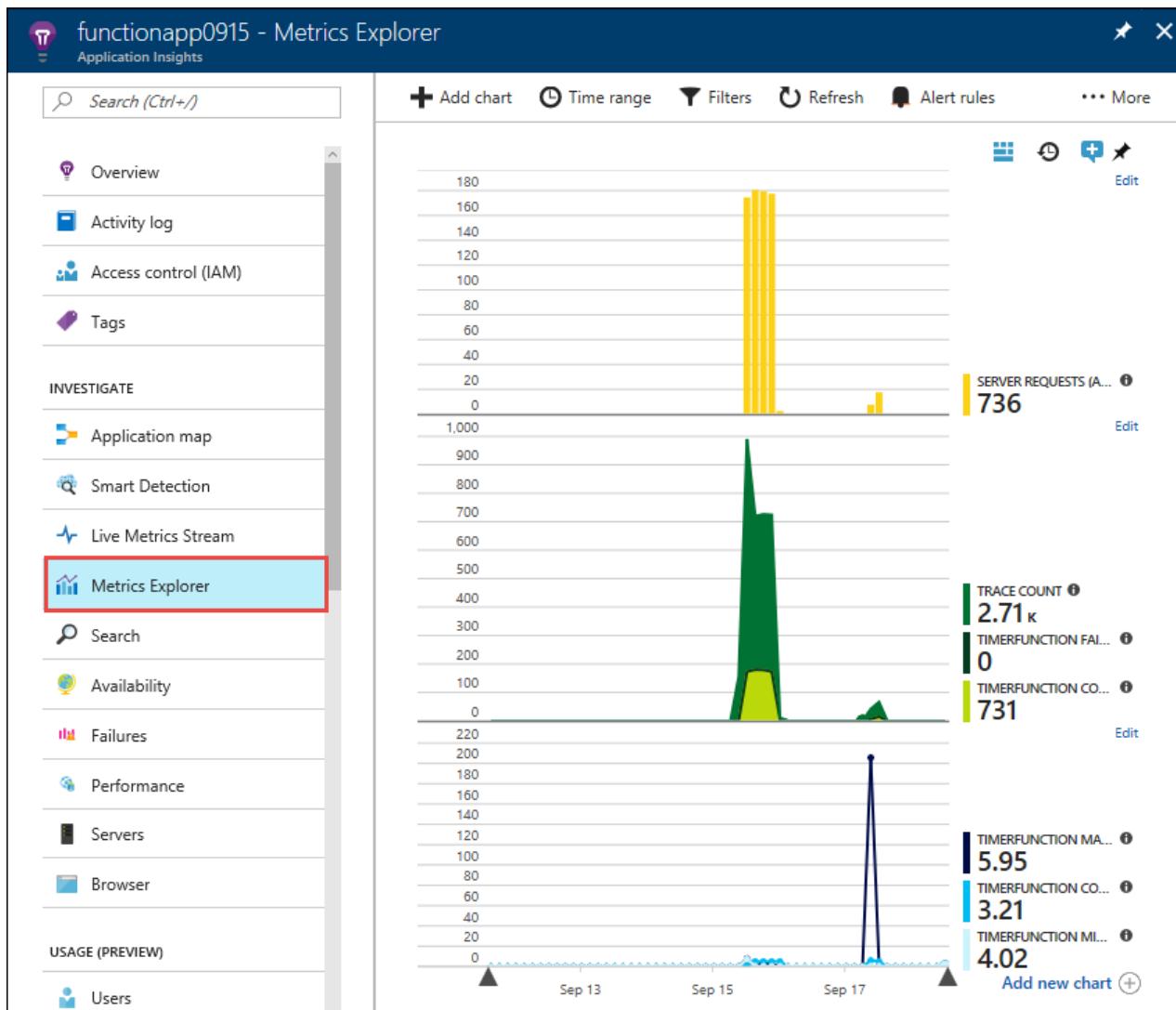
When you enable Application Insights and disable built-in logging, the **Monitor** tab for a function in the Azure portal takes you to Application Insights.

View telemetry data

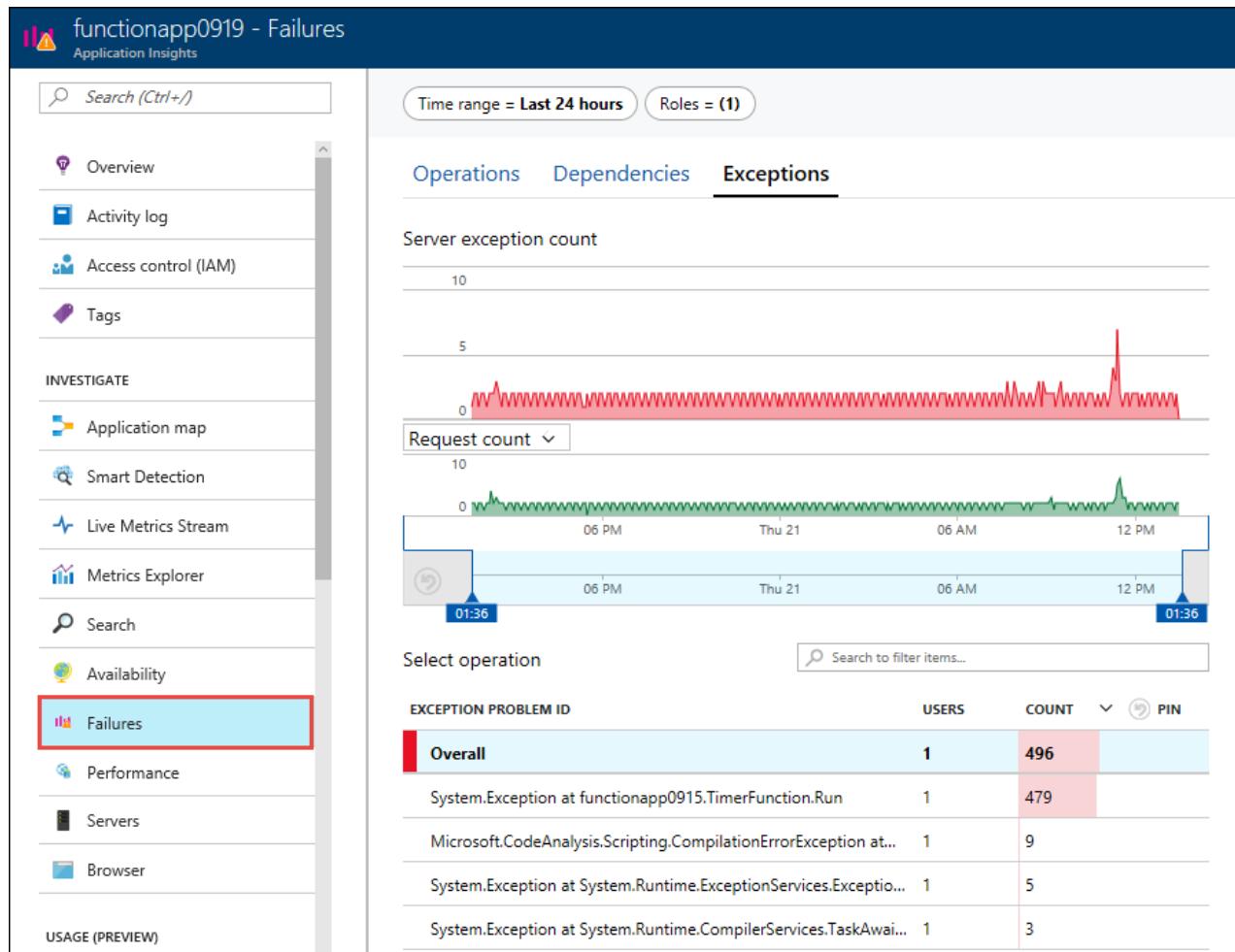
To navigate to the connected Application Insights instance from a function app in the portal, select the **Application Insights** link on the function app's **Overview** page.

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 are already familiar with Application Insights, you can go directly to [the sections about configuring and customizing the telemetry data](#).

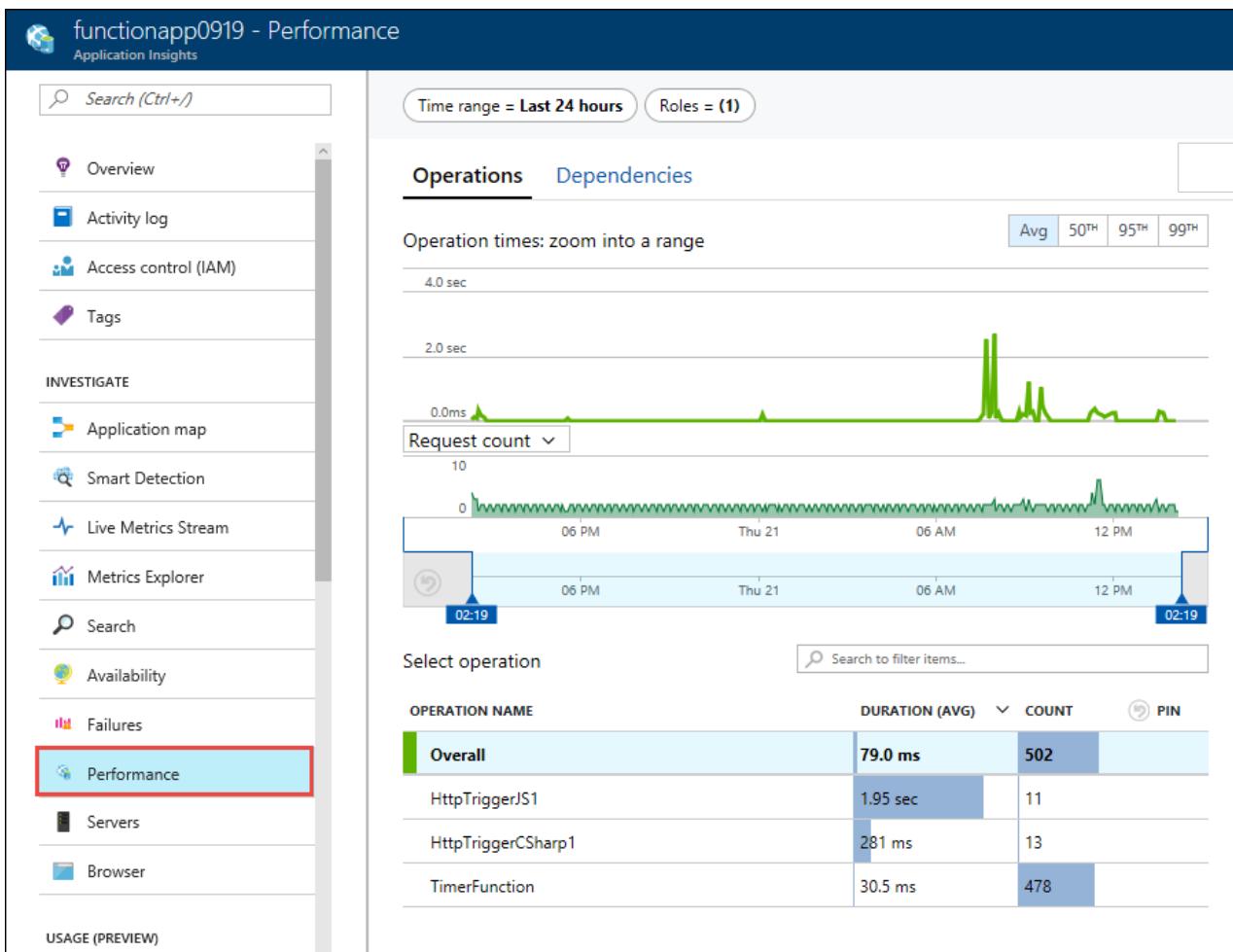
In [Metrics Explorer](#), you can create charts and alerts based on metrics such as number of function invocations, execution time, and success rate.



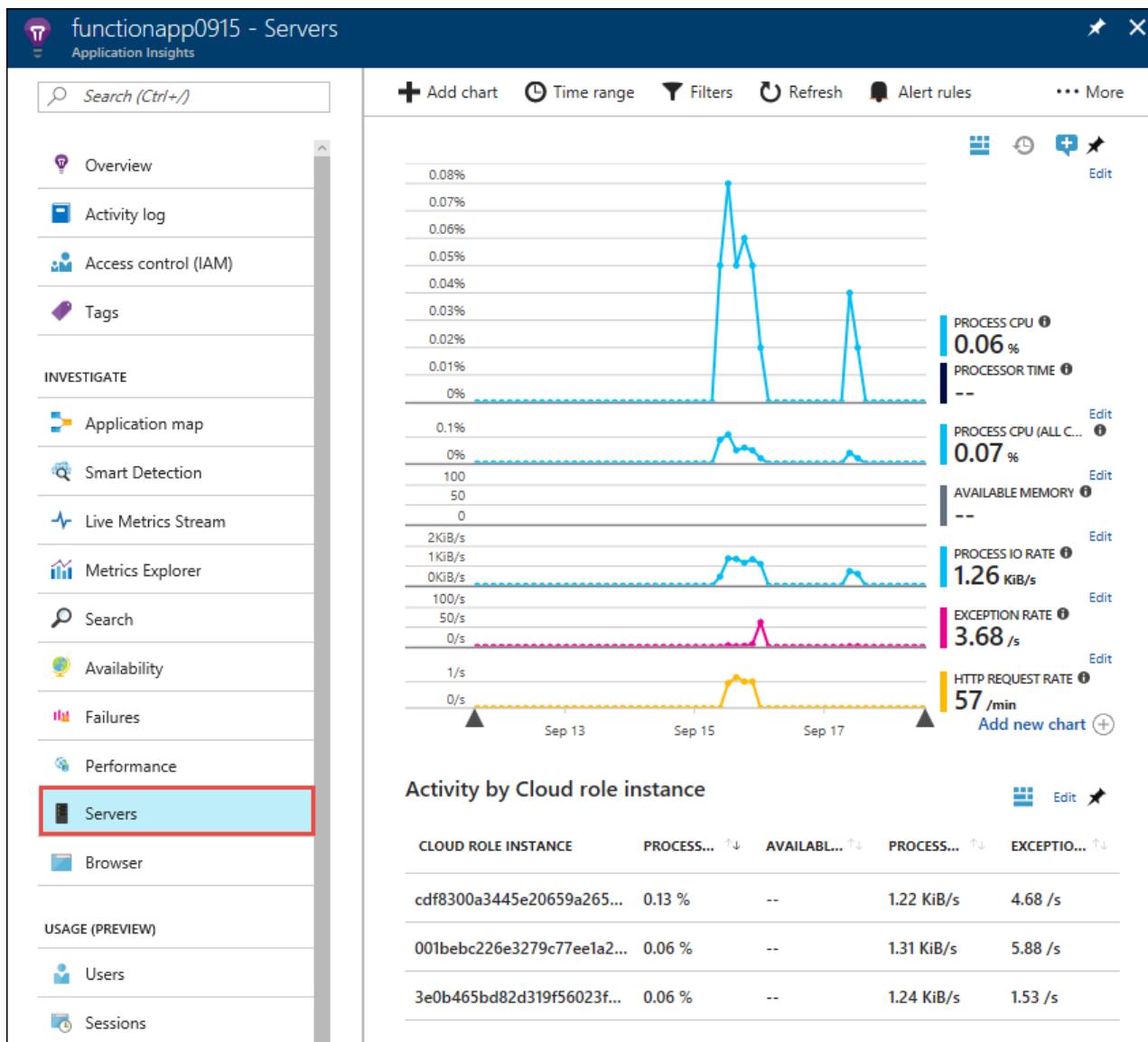
On the [Failures](#) tab, you can create charts and alerts based on function failures and server exceptions. The **Operation Name** is the function name. Failures in dependencies are not shown unless you implement [custom telemetry](#) for dependencies.



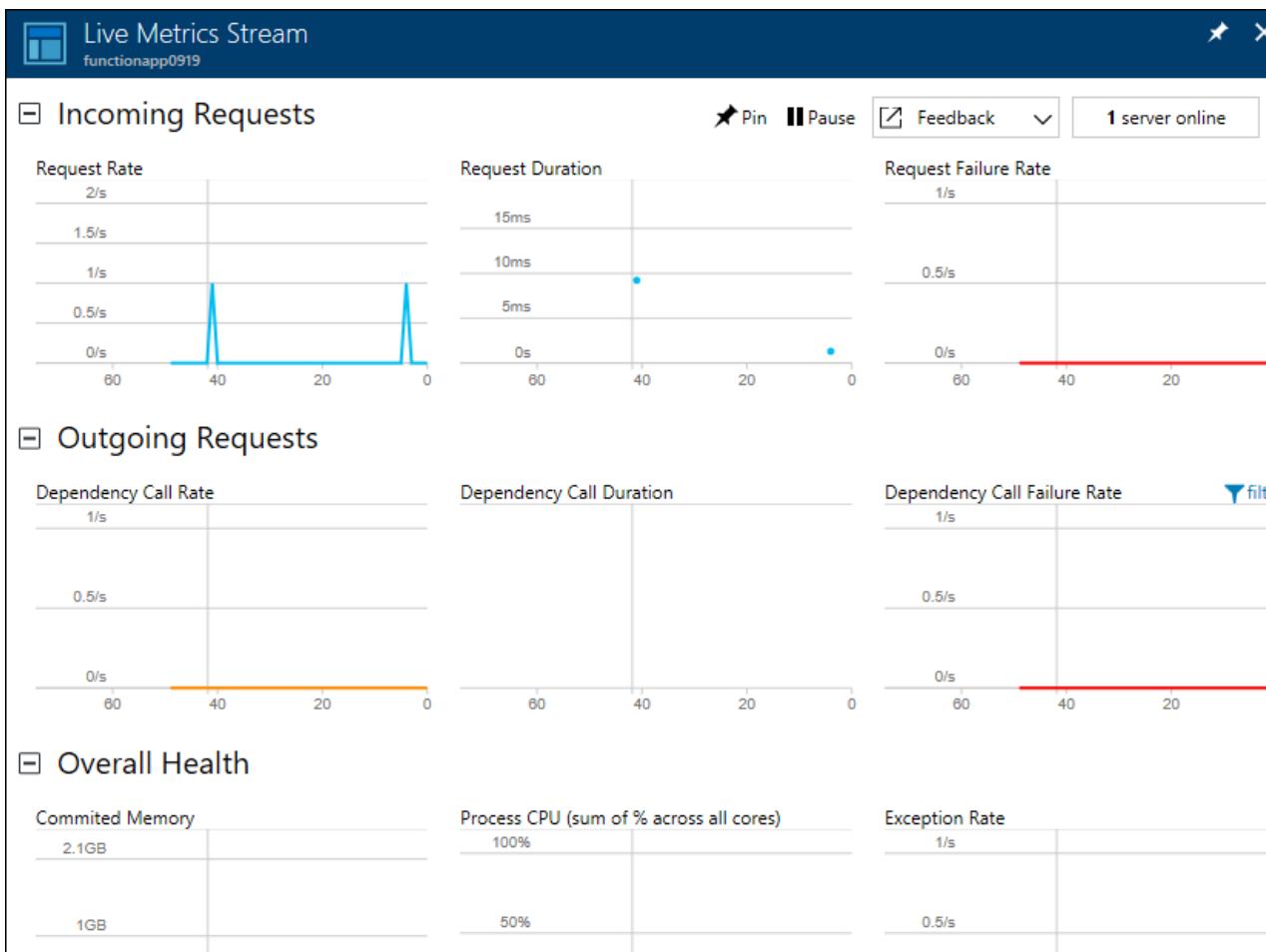
On the [Performance](#) tab, you can analyze performance issues.



The **Servers** tab shows 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**.



The [Live Metrics Stream](#) tab shows metrics data as it is created in real time.



Query telemetry data

[Application Insights Analytics](#) gives you access to all of the telemetry data in the form of tables in a database. Analytics provides a query language for extracting, manipulating, and visualizing the data.

functionapp0915

Application Insights - Last 24 hours (30 minute granularity) - ASP.NET web application

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

INVESTIGATE

Application map

Smart Detection

Live Metrics Stream

Metrics Explorer

Analytics

Time range

Refresh

More

NEW - Analyze conversion rates within your app with the Funnels tool. →

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
aeb4ae60-b7cb-4f3d-966d-fa43b6607f30

0 Alerts

1 Live Stream

0 Servers

0 Users (1 day)

0 Smart Detection

0 Detections (7d)

-- Availability

App map

The screenshot shows the Azure Application Insights Analytics interface. On the left, there's a sidebar titled "ACTIVE" with a tree view of data sources. Under "APPLICATION INSIGHTS", "traces" is selected. Other items like "customEvents", "pageViews", and "requests" are also listed. Under "OTHER DATA SOURCES", there's an option to "Add new data source". The main pane displays a table of log entries from the last 24 hours. The table has columns: timestamp [UTC], message, and severityLevel. The first few rows show logs related to host configuration and function startup.

timestamp [UTC]	message	severityLevel
2017-09-18T20:44:25.738	Reading host configuration file 'D:\home\site\wwwroot\host.json'	1
2017-09-18T20:44:25.738	Host configuration file read: { "logger": { "category": "Function", "level": "Information" }}	1
2017-09-18T20:44:26.114	Function 'TimerFunction' is disabled	1
2017-09-18T20:44:26.194	Loaded custom extension: EventGridExtensionConfig from assembly 'Microsoft.Azure.WebJobs.Extensions.EventGrid, Version=1.0.1.0, Culture=neutral, PublicKeyToken=null'	1
2017-09-18T20:44:26.194	Loaded custom extension: SendGridConfiguration from assembly 'Microsoft.Azure.WebJobs.Extensions.SendGrid, Version=1.0.1.0, Culture=neutral, PublicKeyToken=null'	1
2017-09-18T20:44:26.194	Loaded custom extension: BotFrameworkConfiguration from assembly 'Microsoft.Bot.Builder.Azure, Version=3.0.0.0, Culture=neutral, PublicKeyToken=null'	1
2017-09-18T20:44:27.097	Generating 3 job function(s)	1
2017-09-18T20:44:27.146	Starting Host (HostId=functionapp0915, Version=1.0.1.0)	1
2017-09-18T20:44:27.334	Found the following functions: functionapp0915.HttpTrigger	1
2017-09-18T20:44:27.394	Host lock lease acquired by instance ID '78e1fb3ed5a...'	1

Here's a query example. This one 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 of the left pane. You can find data generated by function invocations in the following tables:

- **traces** - Logs created by the runtime and by function code.
- **requests** - One for each function invocation.
- **exceptions** - Any exceptions thrown by the runtime.
- **customMetrics** - Count of successful and failing invocations, success rate, duration.
- **customEvents** - Events tracked by the runtime, for example: HTTP requests that trigger a function.
- **performanceCounters** - Info about the performance of the servers that the functions are running on.

The other tables are for availability tests and client/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 `customDimensions.LogLevel` and `customDimensions.Category`. You can provide additional fields in logs 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, but 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. The remainder of this article shows how to configure and customize the data that your functions send to Application Insights.

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 that have a category beginning with "Host". For example, the "function started," "function executed," and "function completed" logs have category "Host.Executor".

If you write logs in your function code, their category is "Function".

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
Debug	1
Information	2
Warning	3
Error	4
Critical	5
None	6

Log level `None` is explained in the next section.

Configure logging 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. Here's an example:

```
{  
  "logger": {  
    "categoryFilter": {  
      "defaultLevel": "Information",  
      "categoryLevels": {  
        "Host.Results": "Error",  
        "Function": "Error",  
        "Host.Aggregator": "Information"  
      }  
    }  
  }  
}
```

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 only **Information** level and above to Application Insights. Logs for **Debug** level and below are ignored.
- 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. For example, "Host" in *host.json* controls logging for "Host.General", "Host.Executor", "Host.Results", and so forth.

If *host.json* includes multiple categories that start with the same string, the longer ones are matched first. For example, suppose you want everything from the runtime except "Host.Aggregator" to log at **Error** level, while "Host.Aggregator" logs at **Information** level:

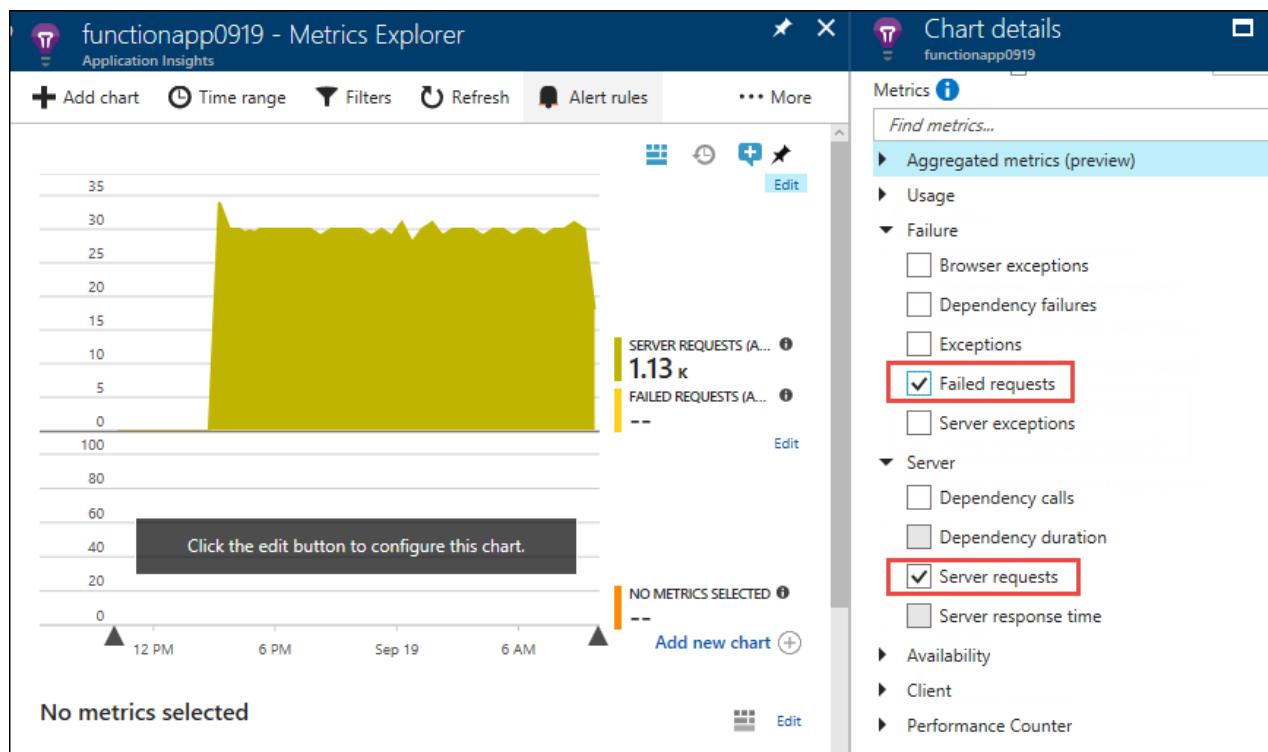
```
{
  "logger": {
    "categoryFilter": {
      "defaultLevel": "Information",
      "categoryLevels": {
        "Host": "Error",
        "Function": "Error",
        "Host.Aggregator": "Information"
      }
    }
  }
}
```

To suppress all logs for a category, you can use log level **None**. No logs are written with that category and there is 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, so 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 number of runs, success rate, and duration.

The screenshot shows the Application Insights Analytics interface for the 'functionapp0919' resource. On the left, the 'ACTIVE' schema tree is expanded to show 'functionapp0919' and its sub-categories: APPLICATION INSIGHTS (traces, customEvents, pageViews, requests, dependencies, exceptions, availabilityResults), and Host.Aggregator (customMetrics, performanceCounters, browserTimings). The 'customMetrics' node is highlighted with a red box. In the center, a query editor window titled 'New Query 1' is open, with the search bar containing 'customMetrics'. Below the search bar, the results are displayed in a 'TABLE' view. The table has columns: timestamp..., name, value, valueCount, valueSum, valueMin, and valueMax. The data shows five rows of metrics for a TimerFunction:

timestamp...	name	value	valueCount	valueSum	valueMin	valueMax
2017-09-2...	TimerFunction Success Rate	0	1	0	0	0
2017-09-2...	TimerFunction Duration	1.8301	1	1.8301	1.8301	1.8301
2017-09-2...	TimerFunction Count	1	1	1	1	1
2017-09-2...	TimerFunction Successes	0	1	0	0	0
2017-09-2...	TimerFunction Failures	1	1	1	1	1

All of these logs are written at [Information](#) level, so 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.

The screenshot shows the Azure Application Insights Analytics interface. On the left, there's a sidebar with 'ACTIVE' logs for 'functionapp0915'. Under 'APPLICATION INSIGHTS', 'traces' is selected. The main area shows a table of log entries with columns: timestamp [UTC], message, and severityLevel. The table contains 10 log entries from September 18, 2017, at 20:44:25.738 UTC. The logs are mostly informational, with one warning about a disabled function.

timestamp [UTC]	message	severityLevel
2017-09-18T20:44:25.738	Reading host configuration file 'D:\home\site\wwwroot\host.json'	1
2017-09-18T20:44:25.738	Host configuration file read: { "logger": { "category": "Function", "level": "Information" }}	1
2017-09-18T20:44:26.114	Function 'TimerFunction' is disabled	1
2017-09-18T20:44:26.194	Loaded custom extension: EventGridExtensionConfig from assembly 'Microsoft.Azure.WebJobs.Extensions.EventGrid, Version=1.0.1.0'	1
2017-09-18T20:44:26.194	Loaded custom extension: SendGridConfiguration from assembly 'Microsoft.Azure.WebJobs.Extensions.SendGrid, Version=1.0.1.0'	1
2017-09-18T20:44:26.194	Loaded custom extension: BotFrameworkConfiguration from assembly 'Microsoft.Bot.Builder.Azure, Version=3.0.0.0'	1
2017-09-18T20:44:27.097	Generating 3 job function(s)	1
2017-09-18T20:44:27.146	Starting Host (HostId=functionapp0915, Version=1.0.1.0)	1
2017-09-18T20:44:27.334	Found the following functions: functionapp0915.HttpTrigger, functionapp0915.TimerFunction	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 may 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 at times of peak load. When the number of telemetry items exceeds a specified rate, Application Insights starts to randomly ignore some of the incoming items. You can configure sampling in `host.json`. Here's an example:

```
{  
  "applicationInsights": {  
    "sampling": {  
      "isEnabled": true,  
      "maxTelemetryItemsPerSecond" : 5  
    }  
  }  
}
```

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` do 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. For example, 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. For example, 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 in addition to the message string. The result is that the message arguments become fields that you can query on.

For example, if your logger method call looks like the previous example, you could query the field `customDimensions.prop__rowKey`. The `prop__` prefix is added to ensure that 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"  
    }  
}
```

Logging custom metrics

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` using [the Application Insights API for .NET](#).

Write logs in JavaScript functions

In Node.js functions, use `context.log` to write logs. Structured logging is not enabled.

```
context.log('JavaScript HTTP trigger function processed a request.' + context.invocationId);
```

Logging custom metrics

In Node.js functions, you can use the `context.log.metric` method to create custom metrics in Application Insights. Here's a sample method call:

```
context.log.metric("TestMetric", 1234);
```

This code is an alternative to calling `trackMetric` using [the Node.js SDK for Application Insights](#).

Custom telemetry in C# functions

You can use the [Microsoft.ApplicationInsights](#) NuGet package to send custom telemetry data to Application Insights.

Here's an example of C# code that uses the [custom telemetry API](#). The example is for a .NET class library, but the Application Insights code is the same for C# script.

```

using System;
using System.Net;
using Microsoft.ApplicationInsights;
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 telemetry =
            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;

            telemetry.Context.Operation.Id = context.InvocationId.ToString();
            telemetry.Context.Operation.Name = "cs-http";
            if (!String.IsNullOrEmpty(name))
            {
                telemetry.Context.User.Id = name;
            }
            telemetry.TrackEvent("Function called");
            telemetry.TrackMetric("Test Metric", DateTime.Now.Millisecond);
            telemetry.TrackDependency("Test Dependency",
                "swapi.co/api/planets/1/",
                start, DateTime.UtcNow - start, true);

            return name == null
                ? req.CreateResponse(HttpStatusCode.BadRequest,
                    "Please pass a name on the query string or in the request body")
                : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
        }
    }
}

```

Don't call `TrackRequest` or `StartOperation<RequestTelemetry>`, because you'll see duplicate requests for a function invocation. The Functions runtime automatically tracks requests.

Set `telemetry.Context.Operation.Id` to the invocation ID each time your function is started. This makes it possible to correlate all telemetry items for a given function invocation.

```
telemetry.Context.Operation.Id = context.InvocationId.ToString();
```

Custom telemetry in JavaScript functions

The [Application Insights Node.js SDK](#) is currently in beta. Here's some sample code that sends custom telemetry to Application Insights:

```
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}});

    if (req.query.name || (req.body && 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"
        };
    }
    context.done();
};

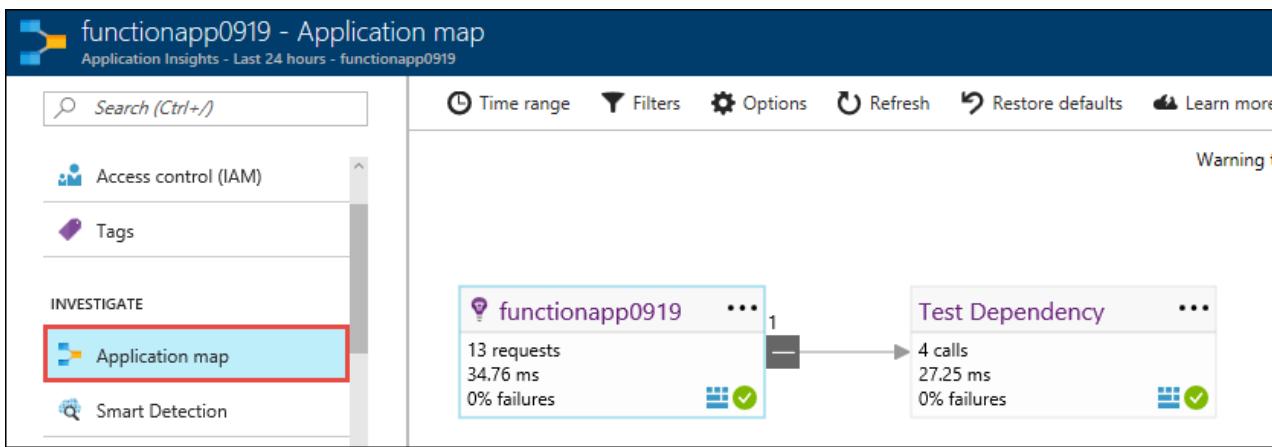
};
```

The `tagOverrides` parameter sets `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.

Known issues

Dependencies

Dependencies that the function has to other services don't show up automatically, but you can write custom code to show the dependencies. The sample code in the [C# custom telemetry section](#) shows how. The sample code results in an *application map* in Application Insights that looks like this:



Report issues

To report an issue with Application Insights integration in Functions, or to make a suggestion or request, [create an issue in GitHub](#).

Monitoring without Application Insights

We recommend Application Insights for monitoring functions because it offers more data and better ways to analyze the data. But you can also find logs and telemetry data in the Azure portal pages for a Function app.

Logging to storage

Built-in logging uses the storage account specified by the connection string in the `AzureWebJobsDashboard` app setting. If that app setting is configured, you can see the logging data in the Azure portal. In a function app page, select a function and then select the **Monitor** tab, and you get a list of function executions. Select a function execution to review the duration, input data, errors, and associated log files.

If you use Application Insights and have [built-in logging disabled](#), the **Monitor** tab takes you to Application Insights.

Real-time monitoring

You can stream log files to a command-line session on a local workstation using the [Azure Command Line Interface \(CLI\) 2.0](#) or [Azure PowerShell](#).

For Azure CLI 2.0, use the following commands to sign in, choose your subscription, and stream log files:

```
az login
az account list
az account set <subscriptionNameOrId>
az appservice web log tail --resource-group <resource group name> --name <function app name>
```

For Azure PowerShell, use the following commands to add your Azure account, choose your subscription, and stream log files:

```
PS C:\> Add-AzureAccount
PS C:\> Get-AzureSubscription
PS C:\> Get-AzureSubscription -SubscriptionName "<subscription name>" | Select-AzureSubscription
PS C:\> Get-AzureWebSiteLog -Name <function app name> -Tail
```

For more information, see [How to stream logs](#).

Next steps

For more information, see the following resources:

- [Application Insights](#)
- [ASP.NET Core logging](#)

Use Azure Functions to connect to an Azure SQL Database

11/29/2017 • 3 min to read • [Edit Online](#)

This topic shows you how to use Azure Functions to create a scheduled job that cleans up rows in a table in an Azure SQL Database. The new C# function is created based on a pre-defined timer trigger template in the Azure portal. 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.

To have your function process individual create, read, update, and delete (CRUD) operations in a Mobile Apps table, you should instead use [Mobile Apps bindings](#).

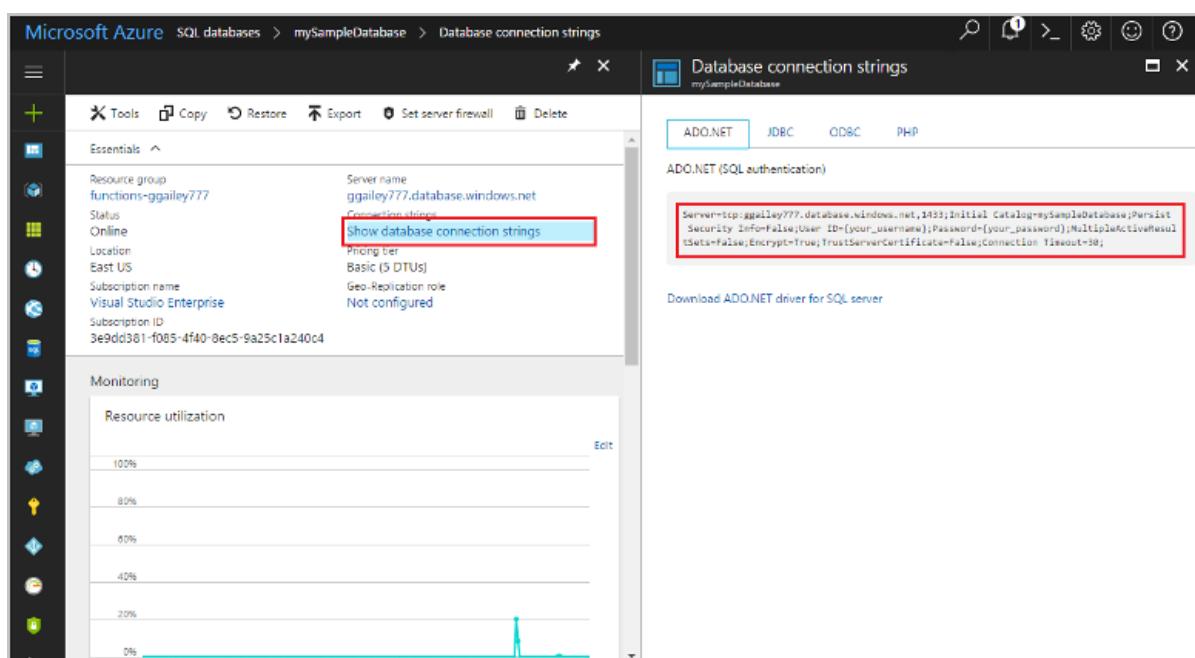
Prerequisites

- This topic uses a timer triggered function. Complete the steps in the topic [Create a function in Azure that is triggered by a timer](#) to create a C# version of this function.
- This topic 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 topic [Create an Azure SQL database in the Azure portal](#).

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. Log 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 **Show database connection strings** and copy the complete **ADO.NET** connection string.



Set the connection string

A function app hosts the execution of your functions in Azure. It is a best practice to store connection strings and other secrets in your function app settings. Using application settings prevents accidental disclosure of the connection string with your code.

1. Navigate to your function app you created [Create a function in Azure that is triggered by a timer](#).
2. Select **Platform features > Application settings**.

The screenshot shows the Azure portal interface for a function app. The left sidebar lists 'Visual Studio Enterprise' and 'Function Apps'. Under 'Function Apps', 'functions-ggailey777' is selected, also highlighted with a red box. The main content area has tabs for 'Overview', 'Settings', 'Platform features' (which is selected and highlighted with a red box), and 'API definition (preview)'. The 'Application settings' section under 'GENERAL SETTINGS' is also highlighted with a red box. Other sections visible include 'CODE DEPLOYMENT', 'DEVELOPMENT TOOLS', 'NETWORKING', 'MONITORING', and 'APP SERVICE PLAN'.

3. Scroll down to **Connection strings** and add a connection string using the settings as specified in the table.

The screenshot shows the 'Application settings' dialog for the 'functions-ggailey777' function app. At the top, there are 'Save' and 'Discard' buttons, with 'Save' highlighted with a red box. Below is a table for 'Connection strings'. One row is selected, showing 'sqldb_connection' in the 'Name' column, 'Server=tcp:ggaile...' in the 'Value' column, and 'SQL Database' in the 'Type' dropdown, which is also highlighted with a red box. There are also 'Slot setting' checkboxes and ellipsis buttons for each row.

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	sqldb_connection	Used to access the stored connection string in your function code.
Value	Copied string	Paste the connection string you copied in the previous section and replace {your_username} and {your_password} placeholders with real values.
Type	SQL Database	Use the default SQL Database connection.

4. Click **Save**.

Now, you can add the C# function code that connects to your SQL Database.

Update your function code

1. In your function app, select the timer-triggered function.
2. Add the following assembly references at the top of the existing function code:

```
#r "System.Configuration"
#r "System.Data"
```

3. Add the following `using` statements to the function:

```
using System.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
```

4. Replace the existing `Run` function with the following code:

```
public static async Task Run(TimerInfo myTimer, TraceWriter log)
{
    var str = ConfigurationManager.ConnectionStrings["sqldb_connection"].ConnectionString;
    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.Info($"{rows} rows were updated");
        }
    }
}
```

This sample command updates the `Status` column based on the ship date. It should update 32 rows of data.

5. Click **Save**, watch the **Logs** windows for the next function execution, then note the number of rows updated in the **SalesOrderHeader** table.



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 topics:

- [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.

4 min to read •

Exporting an Azure-hosted API to PowerApps and Microsoft Flow

12/18/2017 • 7 min 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 [Create a RESTful API in Azure Web Apps](#).

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's 'Platform features' section for an App Service application named 'function-demo-energy'. The 'API' section is highlighted with a red box. Under the 'API' heading, there is a sub-section titled 'API definition'.

If using Azure App Service, select **API definition** from the settings list.

The screenshot shows the Azure portal's 'app-demo-energy' App Service blade. The 'API' section is highlighted with a red box. Under the 'API' heading, there is a sub-section titled 'API definition'.

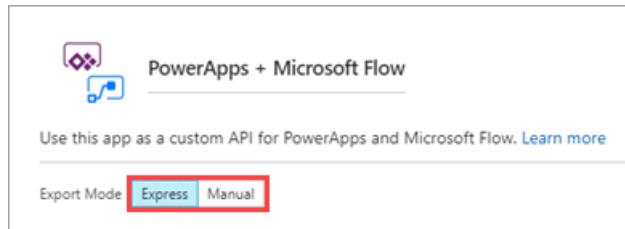
2. The **Export to PowerApps + Microsoft Flow** button should be available (if not, you must first create an OpenAPI definition). Click this button to begin the export process.

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 the 'PowerApps + Microsoft Flow' app interface. It has three main sections:

- 1 Configure Custom API**: You have permission to create custom APIs in some environments and can create a custom API immediately.
- 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
- 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.

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 are two icons: one for PowerApps and one for Microsoft Flow. The title 'PowerApps + Microsoft Flow' is displayed. Below the title, a sub-header says 'Use this app as a custom API for PowerApps and Microsoft Flow. [Learn more](#)'. There are three buttons at the top: 'Export Mode' (gray), 'Express' (white), and 'Manual' (blue). A large number '1' is on the left, followed by the heading 'Prepare your metadata'. A note below says: 'Your API definition is available as an OpenAPI (Swagger) document. Download a copy or make note of the link below. You will use this in step 3.' An 'API definition location' input field contains the URL 'https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=' followed by a copy icon. A red box highlights the 'Download' button.

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 the 'PowerApps + Microsoft Flow' app interface. A large number '2' is on the left, followed by the heading 'Prepare security configuration'. A note below says: 'Your metadata has security definitions. PowerApps and Microsoft Flow will ask for related configuration settings during import. [Learn more](#)'. A table is shown with two rows:

DEFINITION NAME	TYPE
apikeyQuery	apiKey

A red box highlights the entire table.

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**.



3. Click **Create custom connector**, then click **Import an OpenAPI definition**.

The screenshot shows a 'Create custom connector' dialog. At the top is a red box containing a plus sign and the text 'Create custom connector'. Below are four options:

- Create from blank
- Import an OpenAPI file** (this option is highlighted with a red box)
- Import an OpenAPI from URL
- Import a Postman collection

4. Enter a name for the custom connector, then navigate to the OpenAPI definition that you exported, and click **Continue**.

Create custom connector

Custom connector title

Turbine Repair

Upload an OpenAPI file

ApiDef.json

Continue Cancel

5. On the **General** tab, review the information that comes from the OpenAPI definition.
6. On the **Security** tab, if you are prompted to provide authentication details, enter the values appropriate for the authentication type. Click **Continue**.

Authentication type

Choose what authentication is implemented by your API *

API Key

API Key

Users will be required to provide the API Key when creating a connection

Parameter label	Parameter name	Parameter location
API Key (contact n)	code	Query

This example shows the required fields for API key authentication. The fields differ depending on the authentication type.

7. On the **Definitions** tab, all the operations defined in your OpenAPI file are auto-populated. If all your required operations are defined, you can go to the next step. If not, you can add and modify operations here.

Actions (1)

Actions determine the operations that users can perform. Actions can be used to read, create, update or delete resources in the underlying connector.

1 CalculateCosts ...

+ New action

References (0)

References are reusable parameters used by both actions and triggers.

General

* Summary Learn more
Calculates costs

* Description Learn more
Determines if a technician should be sent for repair

* Operation ID
This is the unique string used to identify the operation.

CalculateCosts

Visibility Learn more

none advanced internal important

This example has one operation, named `CalculateCosts`. The metadata, like **Description**, all comes from the OpenAPI file.

8. Click **Create connector** at the top of the page.

You can now connect to the custom connector in PowerApps and Microsoft Flow. For more information on creating connectors in the PowerApps and Microsoft Flow portals, see [Register your custom connector](#)

(PowerApps) and Register your custom connector (Microsoft Flow).

Specify authentication type

PowerApps and Microsoft Flow support a collection of identity providers that provide authentication for custom connectors. If your API requires authentication, ensure that it is captured as a *security definition* in your OpenAPI document, like the following example:

```
"securityDefinitions": {  
    "AAD": {  
        "type": "oauth2",  
        "flow": "accessCode",  
        "authorizationUrl": "https://login.windows.net/common/oauth2/authorize",  
        "scopes": {}  
    }  
}
```

During export, you provide configuration values that allow PowerApps and Microsoft Flow to authenticate users.

This section covers the authentication types that are supported in **Express** mode: API key, Azure Active Directory, and Generic OAuth 2.0. PowerApps and Microsoft Flow also support Basic Authentication, and OAuth 2.0 for specific services like Dropbox, Facebook, and SalesForce.

API key

When using an API key, the users of your connector are prompted to provide the key when they create a connection. You specify an API key name to help them understand which key is needed. In the earlier example, we use the name `API Key (contact meganb@contoso.com)` so people know where to get information about the API key. For Azure Functions, the key is typically one of the host keys, covering several functions within the function app.

Azure Active Directory (Azure AD)

When using Azure AD, you need two Azure AD application registrations: one for the API itself, and one for the custom connector:

- To configure registration for the API, use the [App Service Authentication/Authorization](#) feature.
- To configure registration for the connector, follow the steps in [Adding an Azure AD application](#). The registration must have delegated access to your API and a reply URL of
`https://msmanaged-na.consent.azure-apim.net/redirect`.

For more information, see the Azure AD registration examples for [PowerApps](#) and [Microsoft Flow](#). These examples use Azure Resource Manager as the API; substitute your API if you follow the steps.

The following configuration values are required:

- **Client ID** - the client ID of your connector Azure AD registration
- **Client secret** - the client secret of your connector Azure AD registration
- **Login URL** - the base URL for Azure AD. In Azure, 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

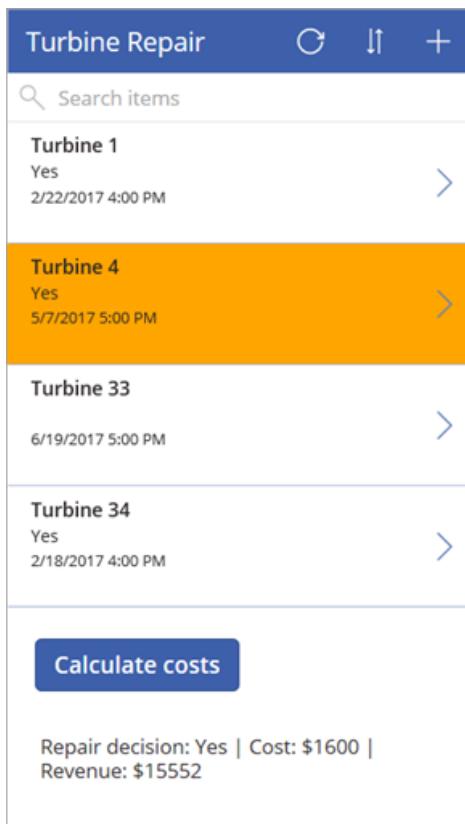
12/18/2017 • 7 min to read • [Edit Online](#)

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.

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](#).

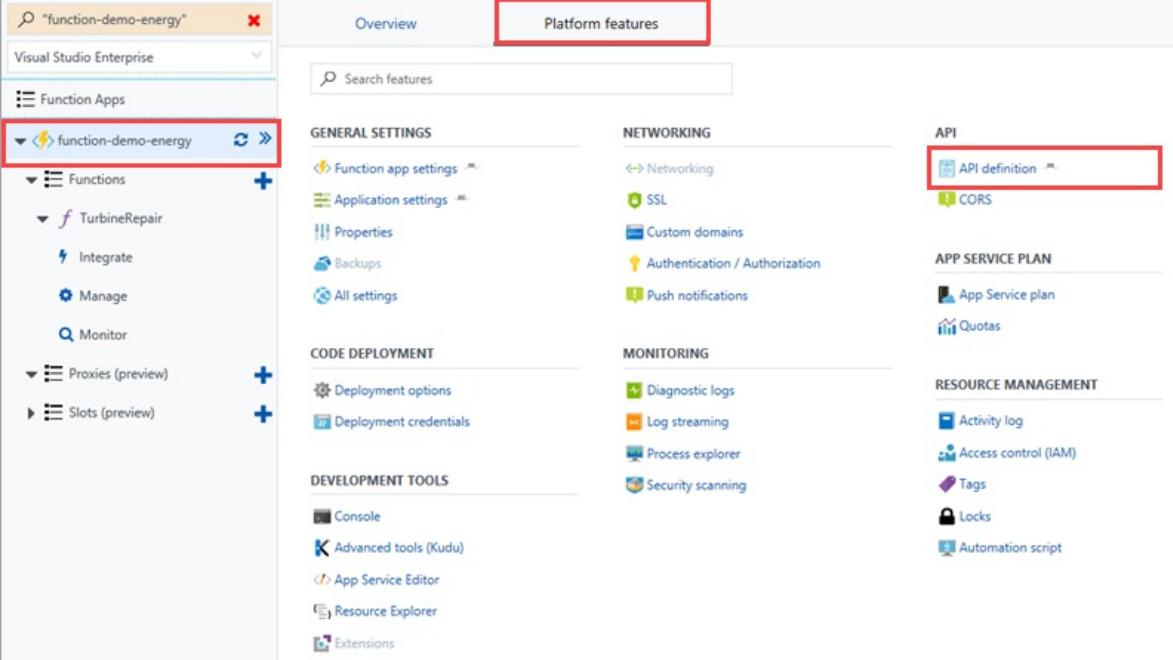
Export an API definition

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

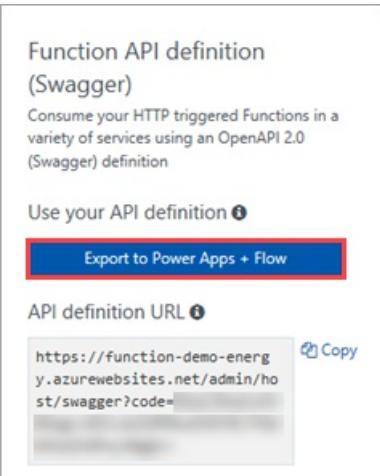
Remember that you must be signed into Azure with the same credentials that you use for your PowerApps and Microsoft Flow tenants. This enables Azure to create the custom API and make it available for both PowerApps and Microsoft Flow.

1. In the [Azure portal](#), click your function app name (like **function-demo-energy**) > **Platform features** > **API definition**.



The screenshot shows the Azure portal's 'Platform features' page 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 'Functions' section is expanded, showing 'TurbineRepair', 'Integrate', 'Manage', and 'Monitor' options. The main content area is divided into 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), and 'RESOURCE MANAGEMENT' (Activity log, Access control (IAM), Tags, Locks, Automation script). The 'API definition' link under 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 includes a brief description: 'Consume your HTTP triggered Functions in a variety of services using an OpenAPI 2.0 (Swagger) definition'. Below this, there's a link 'Use your API definition' with a help icon, and a large blue button labeled 'Export to Power Apps + Flow' which is also highlighted with a red box. At the bottom, there's a section for the 'API definition URL' with a copy icon, showing the URL: <https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=...>.

3. In the right pane, use the settings as specified in the table.

SETTING	DESCRIPTION
---------	-------------

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 Name

3 Export to PowerApps + Microsoft Flow

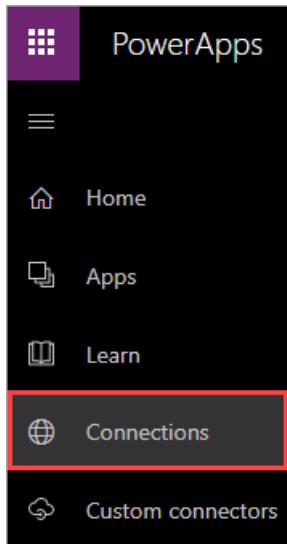
Click OK to create a new custom API using the above parameters. By default, the custom API will be shared only with you. Visit the PowerApps or Microsoft Flow portal to share with the other members of the organization.

4. Click **OK**. The custom API is now built and added to the environment you specified.

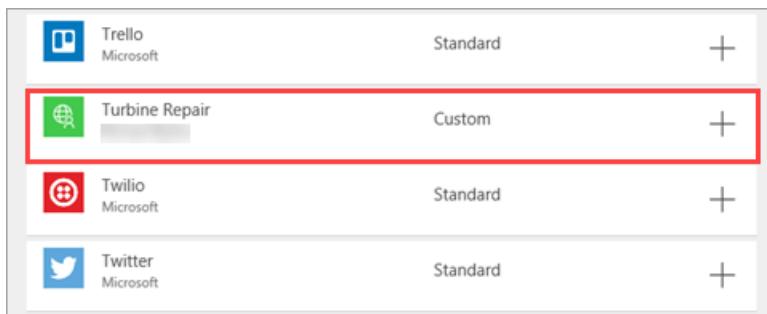
Add a connection to the API

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.

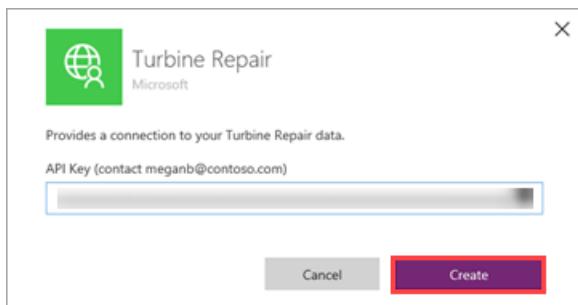
1. In web.powerapps.com, click **Connections**.



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.

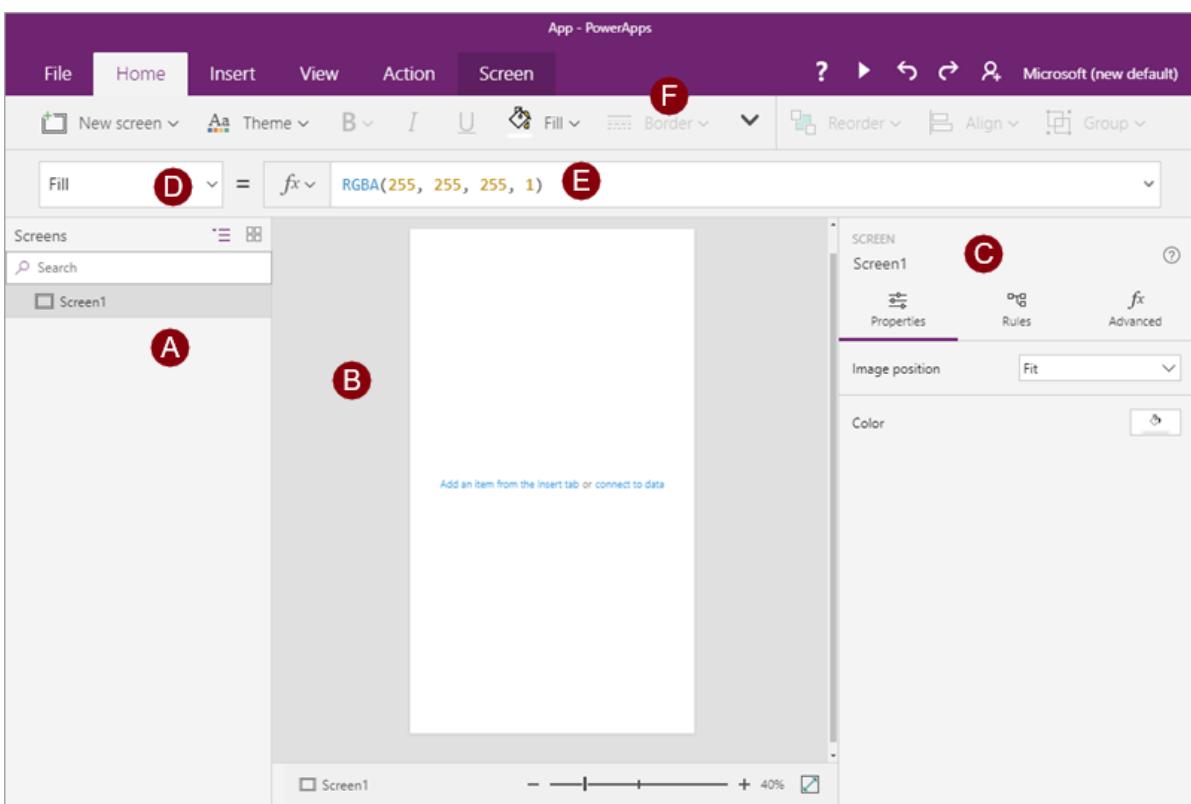
Create an app and add data sources

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

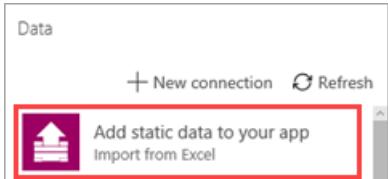
(F) Ribbon, where you add controls and customize design elements

2. Add the Excel file as a data source.

The data you will import looks like the following:

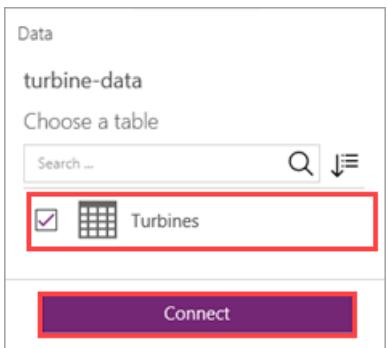
Title	Latitude	Longitude	LastServiceDate	MaxOutput	ServiceRequired	EstimatedEffort	InspectionNotes
Turbine 1	47.438401	-121.383767	2/23/2017	2850	Yes	6	This is the second issue this month.
Turbine 4	47.433385	-121.383767	5/8/2017	5400	Yes	6	
Turbine 33	47.428229	-121.404641	6/20/2017	2800			
Turbine 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.

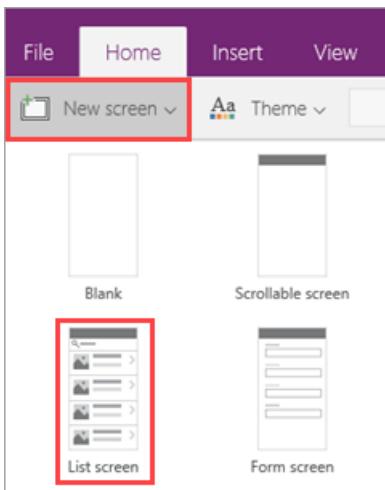
- On the **Data** panel, click **Add data source**.
- Click **Turbine Repair**.



Add controls to view data in the app

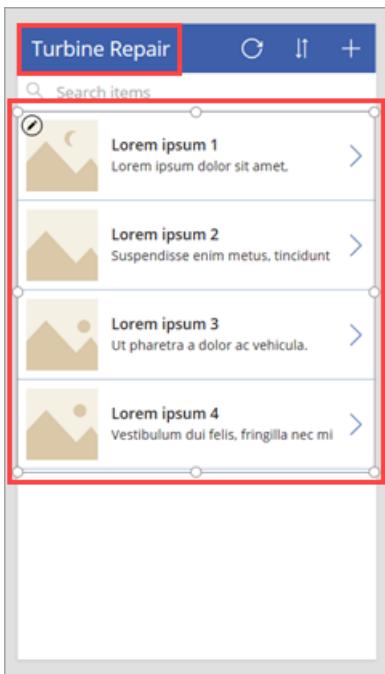
Now that the data sources are available in the app, you add a screen to your app so you can view the turbine data.

- On the **Home** tab, click **New screen > List screen**.

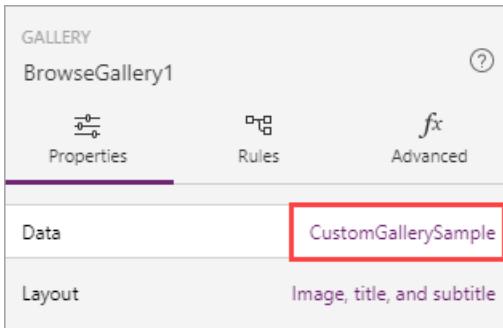


PowerApps adds a screen that contains a *gallery* to display items, and other controls that enable searching, sorting, and filtering.

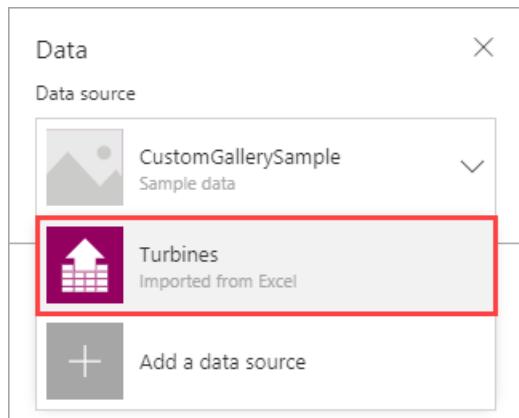
2. Change the title bar to `Turbine Repair`, and resize the gallery so there's room for more controls under it.



3. With the gallery selected, in the right pane, under **Properties**, click `CustomGallerySample`.

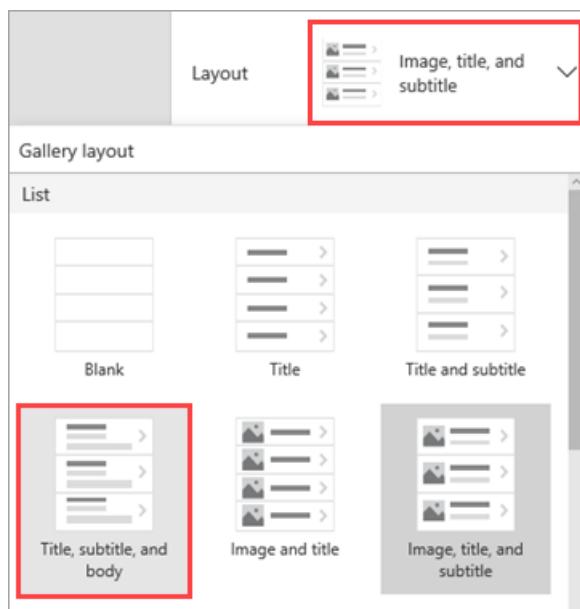


4. In the **Data** panel, select **Turbines** from the list.



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.



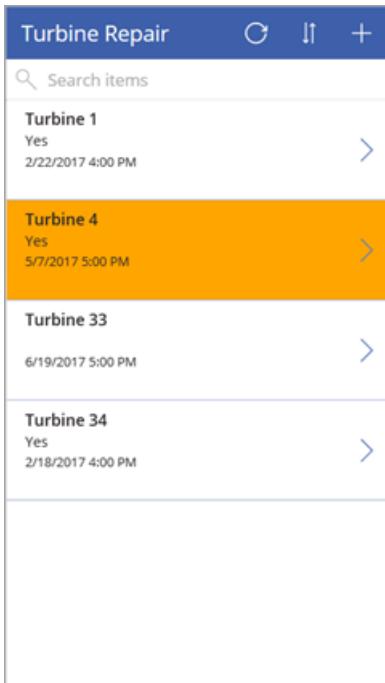
- **Body1** = LastServiceDate
- **Subtitle2** = ServiceRequired
- **Title2** = Title

7. With the gallery selected, set the **TemplateFill** property to the following formula:

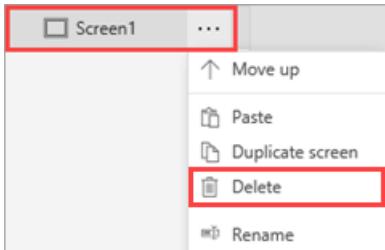
```
If(ThisItem.IsSelected, Orange, White).
```



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**.



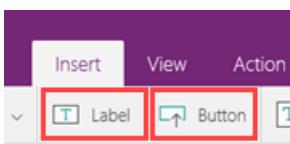
9. Click **File**, and name the app. Click **Save** on the left menu, then click **Save** in the bottom right corner.

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.

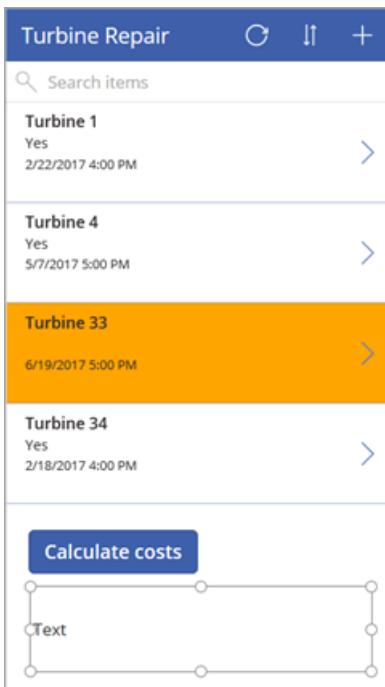
Add controls to call the function and display data

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.EstimatdEffort, 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()`.

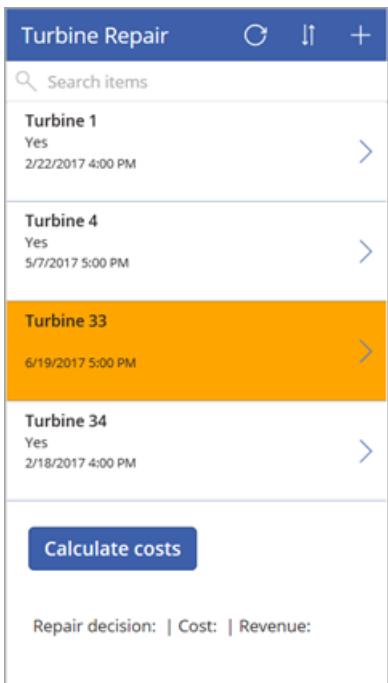
The values passed to the function come from the **EstimatedEffort** and **MaxOutput** fields for the item selected in the gallery. These fields aren't displayed in the gallery, but they're still available to use in formulas.

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.

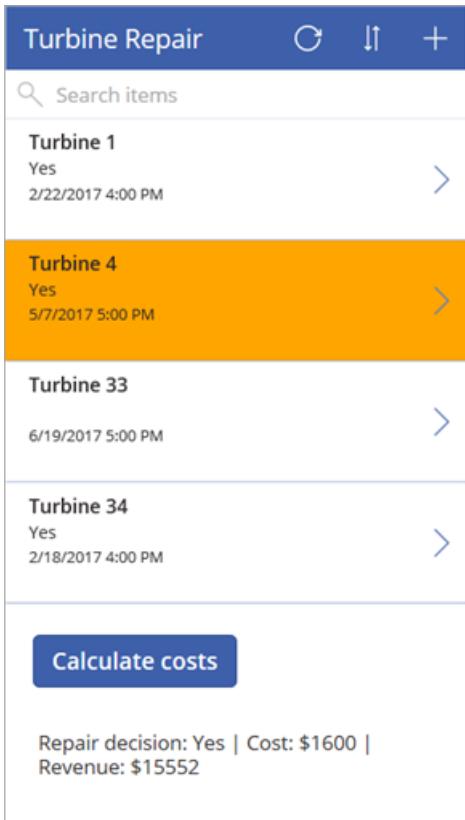
The completed app should look like the following image.



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 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

12/18/2017 • 8 min 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.

Prerequisites

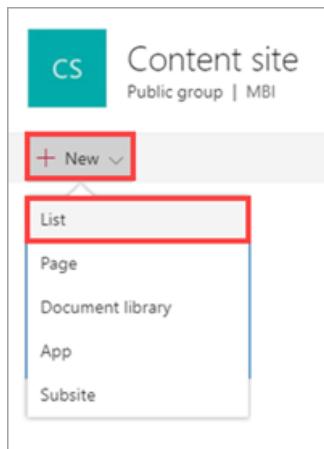
- An active [Microsoft Flow account](#) with the same sign in credentials as your Azure account.
- SharePoint, which you use as a data source for this flow. Sign up for [an Office 365 trial](#) if you don't already have SharePoint.
- Complete the tutorial [Create an OpenAPI definition for a function](#).

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

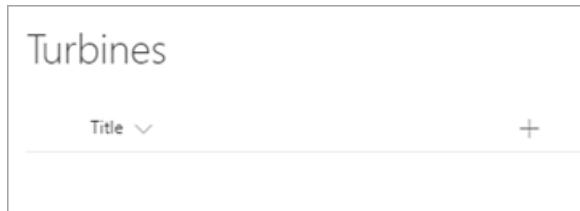
1. In your SharePoint site, click or tap **New**, then **List**.



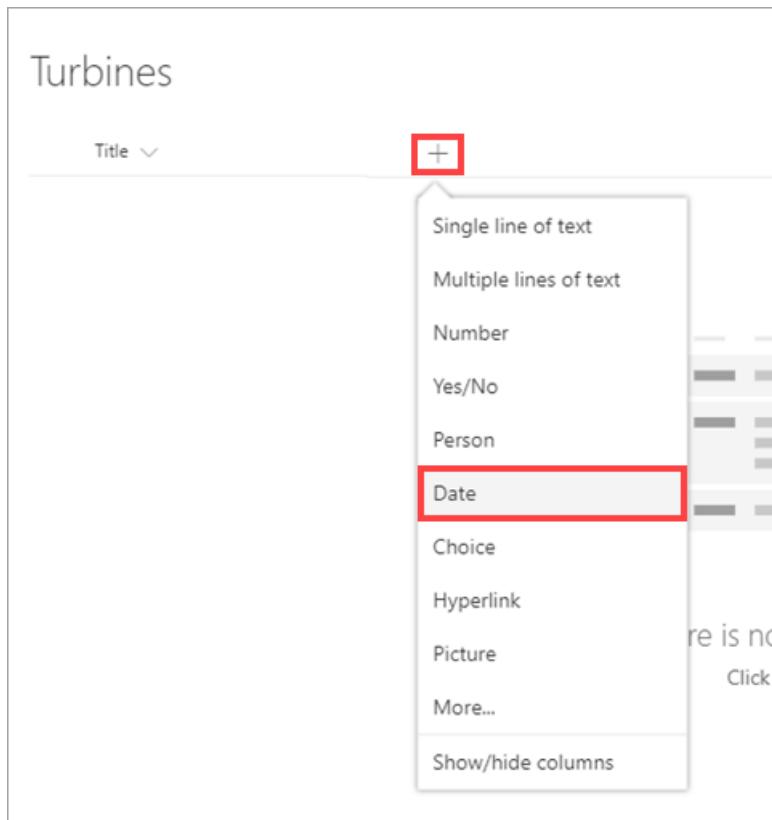
2. Enter the name **Turbines**, then click or tap **Create**.

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. At the bottom are 'Create' and 'Cancel' buttons, with 'Create' highlighted by a red box.

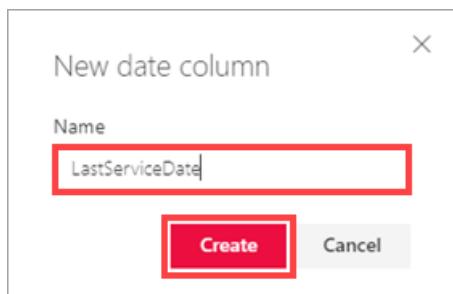
The **Turbines** list is created, with the default **Title** field.



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:

- Number** > "MaxOutput"
- Yes/No** > "ServiceRequired"
- 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 into 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-menu includes 'Functions', 'TurbineRepair', 'Integrate', 'Manage', 'Monitor', 'Proxies (preview)', and 'Slots (preview)'. 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 is highlighted with a red box.

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 is a 'Copy' link next to a URL: <https://function-demo-energy.azurewebsites.net/admin/host/swagger?code=...>. The URL ends with a redacted portion.

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.

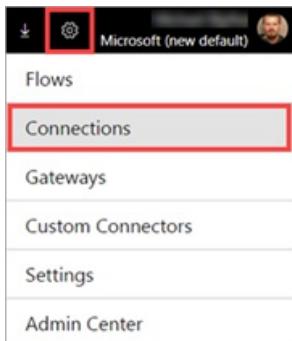
OK

4. Click **OK**. The custom API is now built and added to the environment you specified.

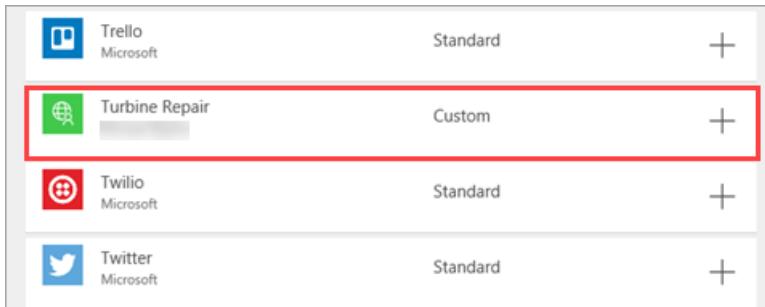
Add a connection to the API

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.

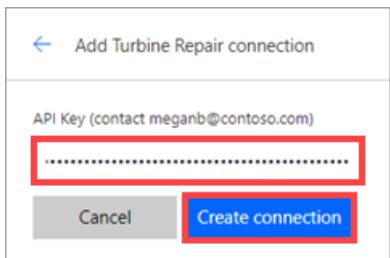
1. In flow.microsoft.com, click the gear icon (in the upper right), then click **Connections**.



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.

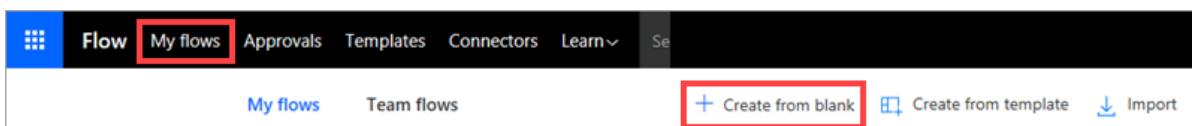
Create a flow

Now you're ready to create a flow that uses the custom connector and the SharePoint list you created.

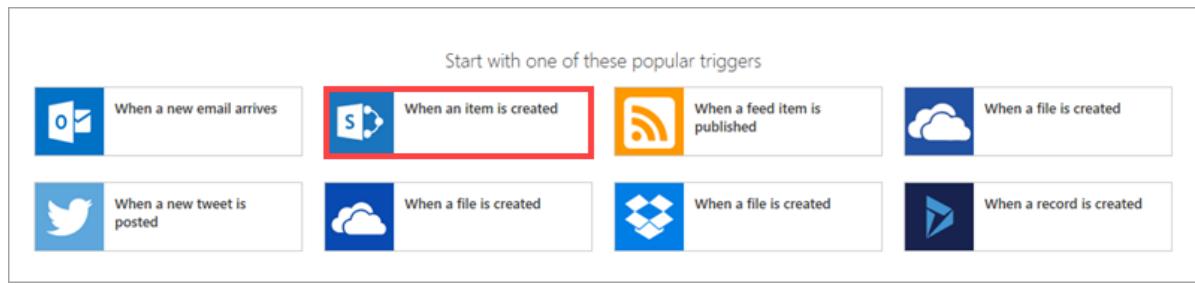
Add a trigger and specify a condition

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**.

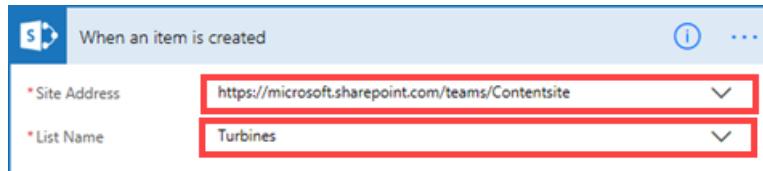


2. Click the SharePoint trigger **When an item is created**.

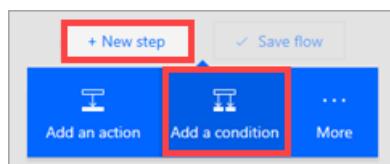


If you're not already signed into SharePoint, you will be prompted to do so.

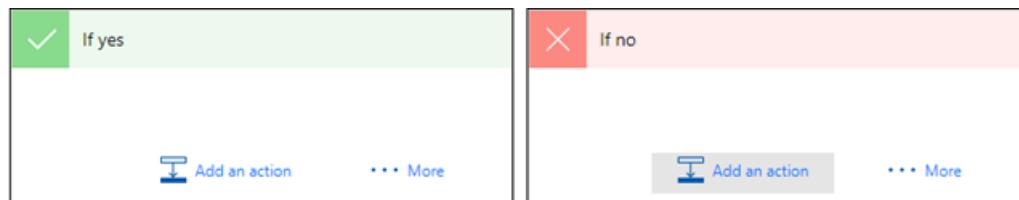
3. For **Site Address**, enter your SharePoint site name, and for **List Name**, enter the list that contains the turbine data.



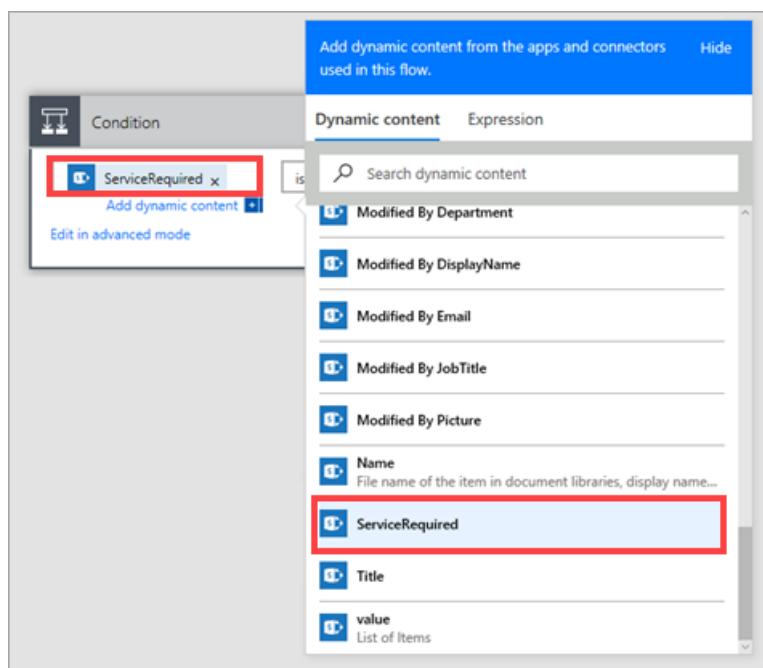
4. Click **New step**, then **Add a condition**.



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.



5. 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**.

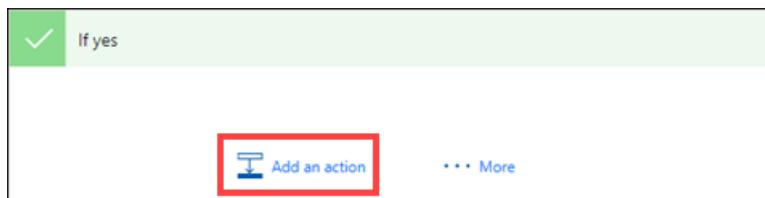
7. Click **Create flow** at the top of the page. Be sure to click **Update Flow** periodically.

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.

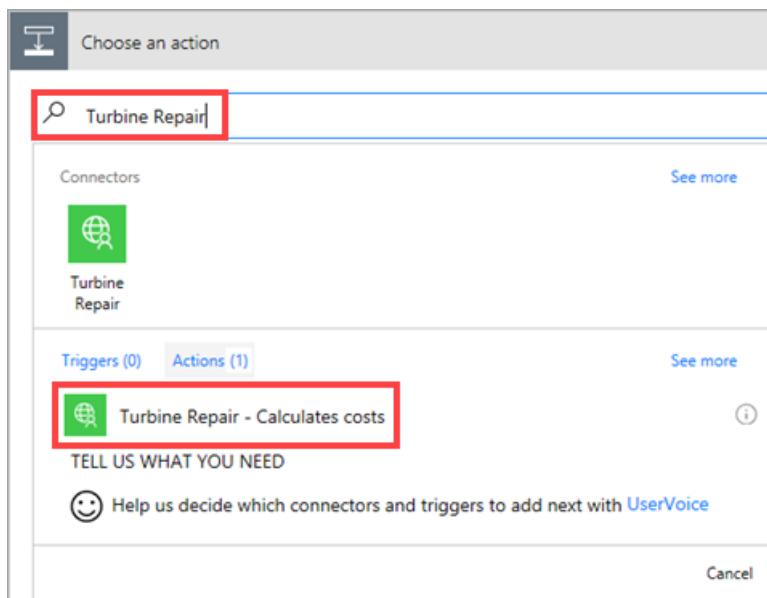
Add the custom connector

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.

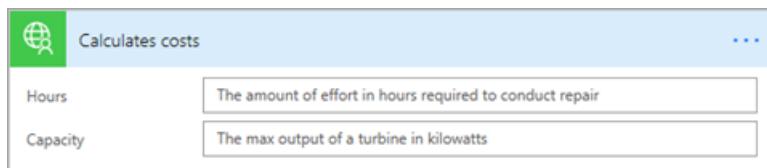
1. In the **If yes** branch, click **Add an action**.



2. In the **Choose an action** dialog box, search for **Turbine Repair**, then select the action **Turbine Repair - Calculates costs**.



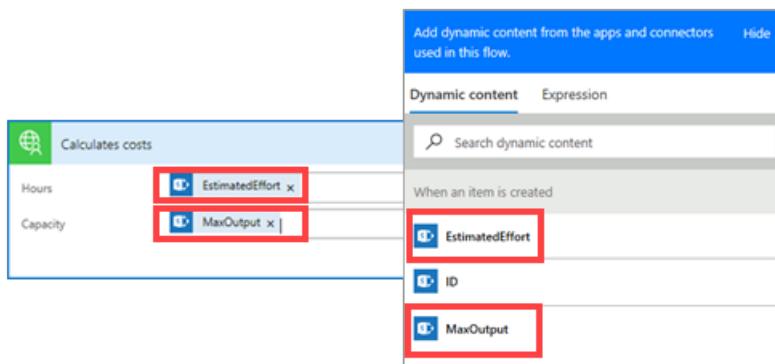
The following image shows the card that is added to the flow. The fields and descriptions come from the OpenAPI definition for the connector.



3. On the **Calculates costs** card, use the **Dynamic content** dialog box to select inputs for the function.

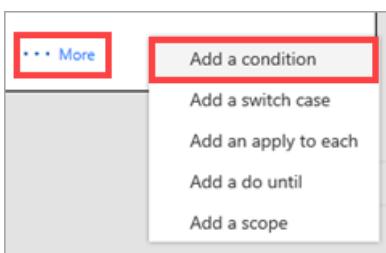
Microsoft Flow shows numeric fields but not the date field, because the OpenAPI definition specifies that **Hours** and **Capacity** are numeric.

For **Hours**, select **EstimatedEffort**, and for **Capacity**, select **MaxOutput**.

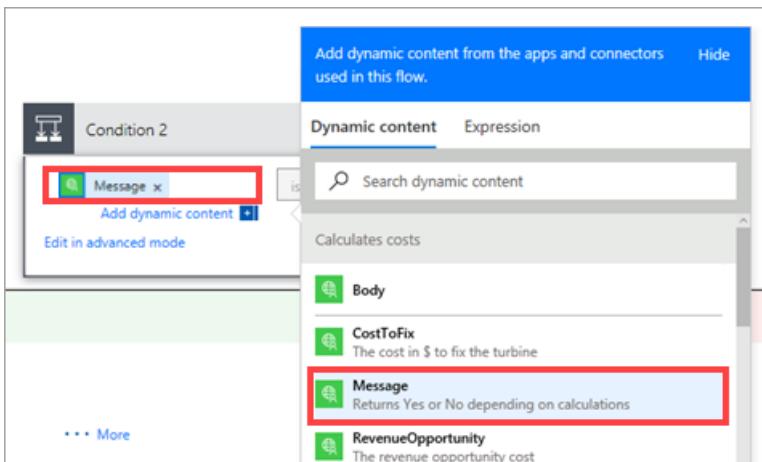


Now you add another condition based on the output of the function.

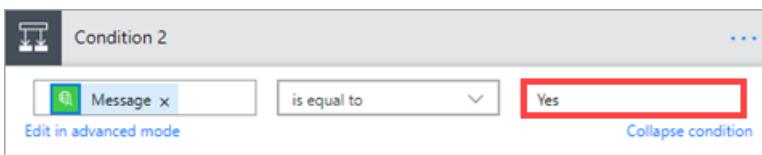
4. At the bottom of the **If yes** branch, click **More**, then **Add a condition**.



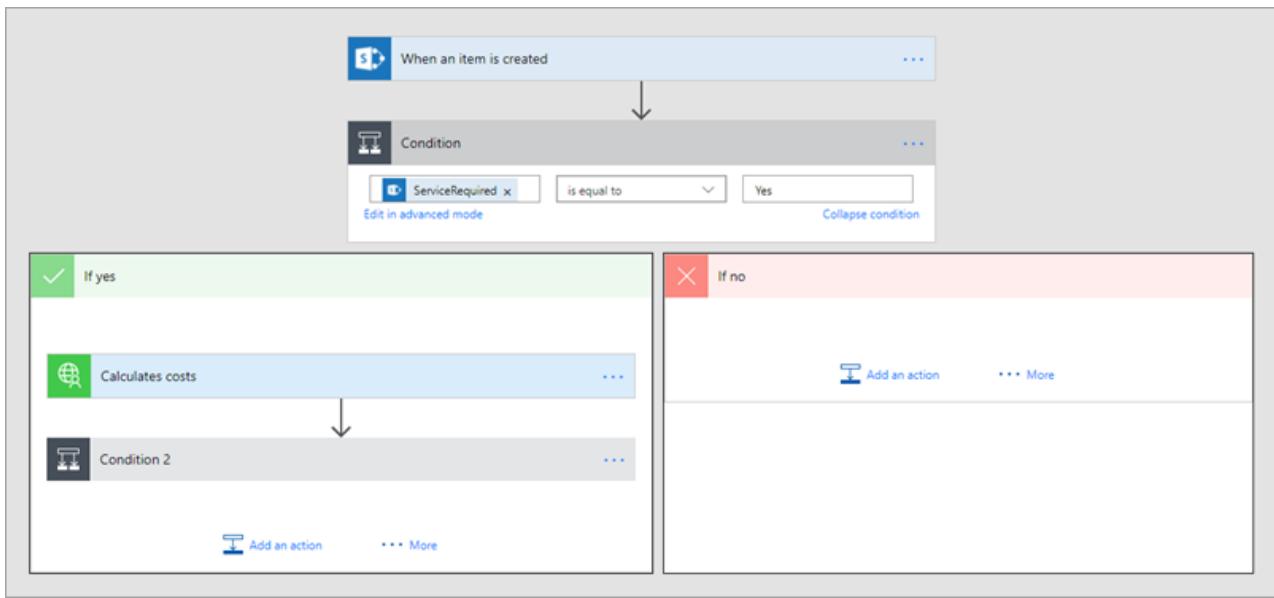
5. On the **Condition 2** card, click the first box, then select **Message** from the **Dynamic content** dialog box.



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).



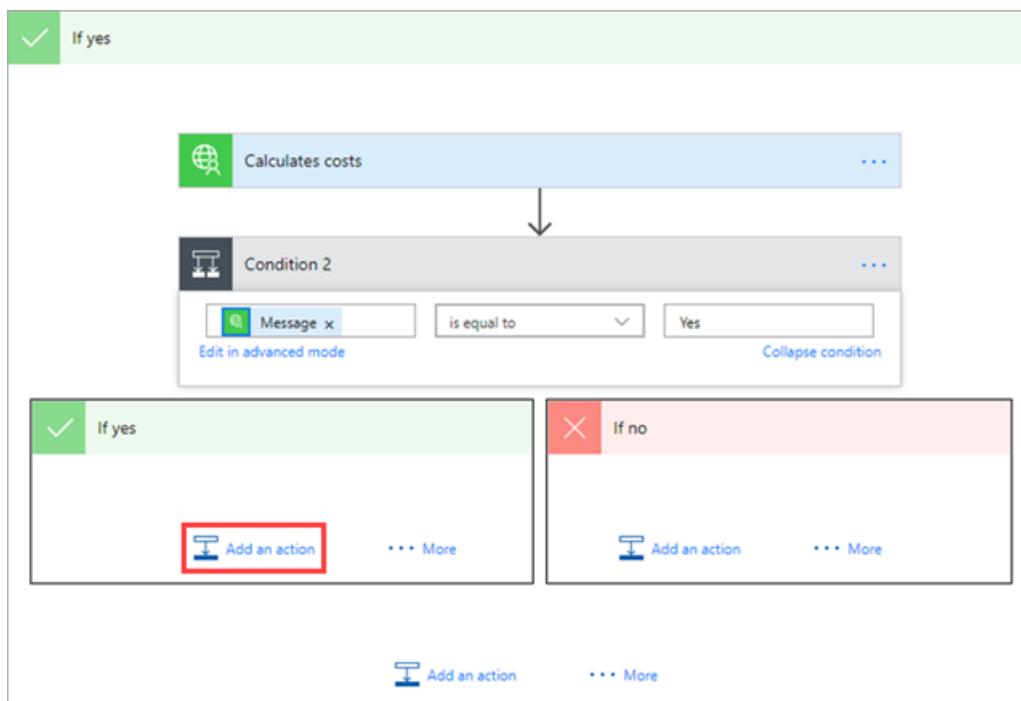
The flow should now look like the following image.



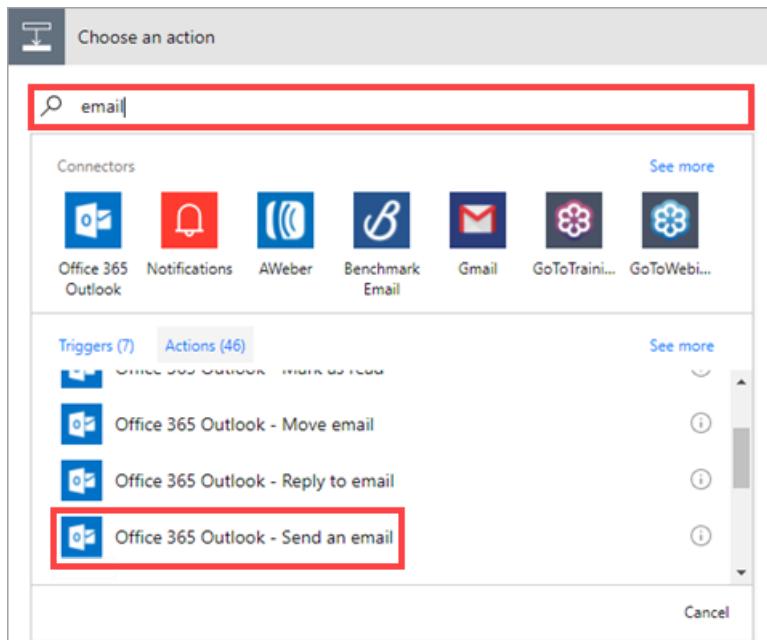
Send email based on function results

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](#).

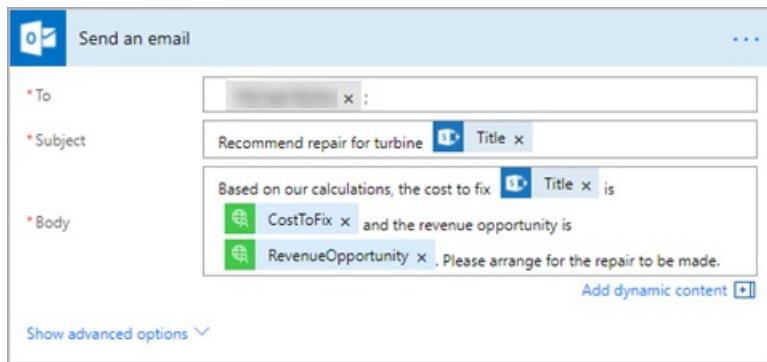
1. In the **If yes** branch of the second condition, click **Add an action**.



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).

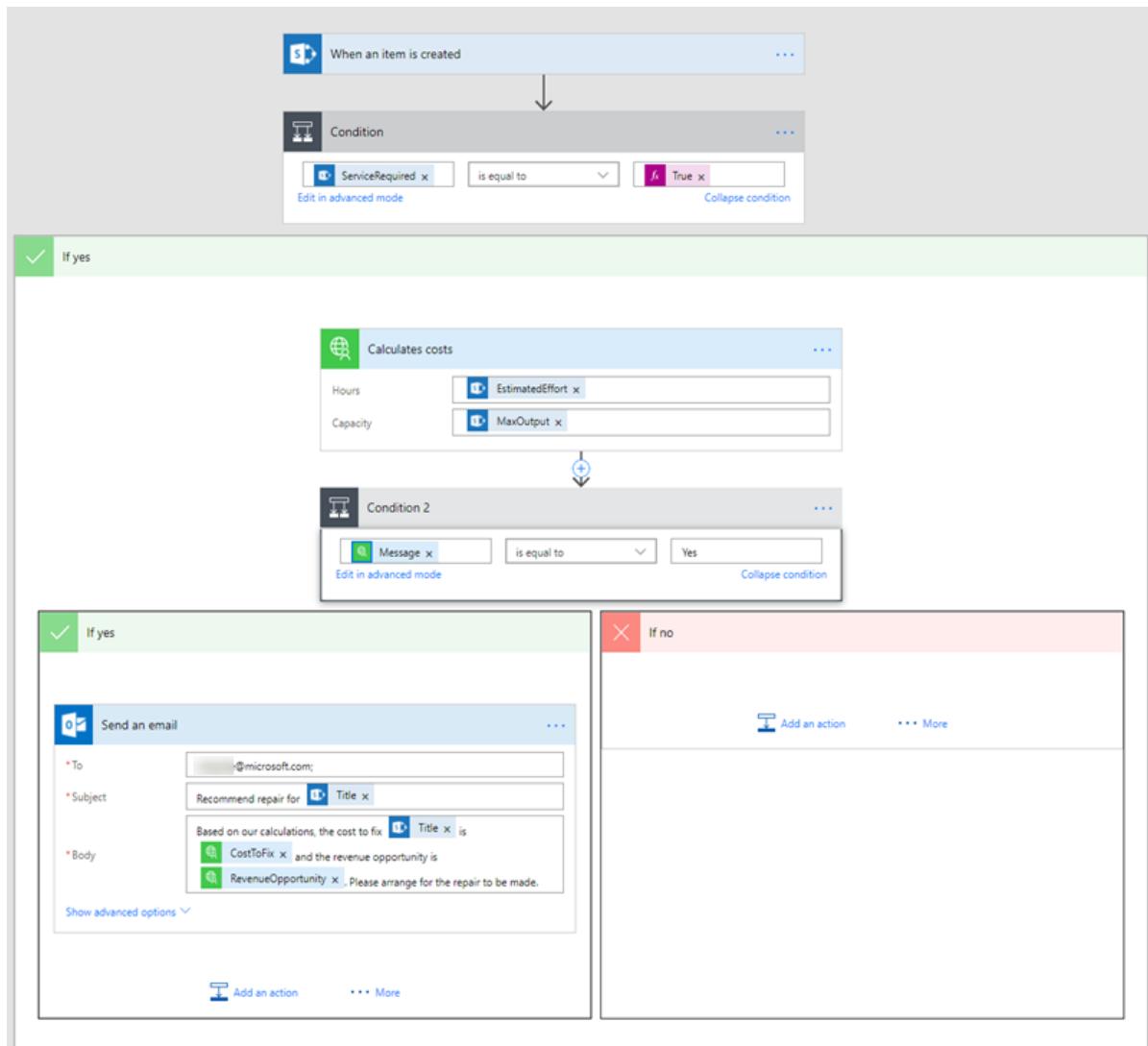


3. On the **Send an email** card, compose an email. Enter a valid name in your organization for the **To** field. In the image below you can see the other fields are a combination of text and tokens from the **Dynamic content** dialog box.



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).

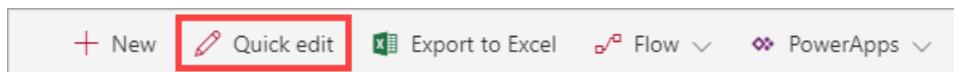


4. Click **Update Flow** at the top of the page, then click **Done**.

Run the flow

Now that the flow is completed, you add a row to the SharePoint list and see how the flow responds.

1. Go back to the SharePoint list, and click **Quick Edit**.



2. Enter the following values in the edit grid.

LIST COLUMN	VALUE
Title	Turbine 60
LastServiceDate	08/04/2017
MaxOutput	2500
ServiceRequired	Yes
EstimatedEffort	10

3. 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 Azure Managed Service Identity (public preview) in App Service and Azure Functions

12/6/2017 • 6 min to read • [Edit Online](#)

NOTE

Managed Service Identity for App Service and Azure Functions is currently in preview.

This topic shows you how to create a managed app identity for App Service and Azure Functions applications and how to use it to access other resources. A managed service 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 Service Identity, see the [Managed Service Identity overview](#).

Creating an app with an identity

Creating an app with an identity requires an additional property to be set on the application.

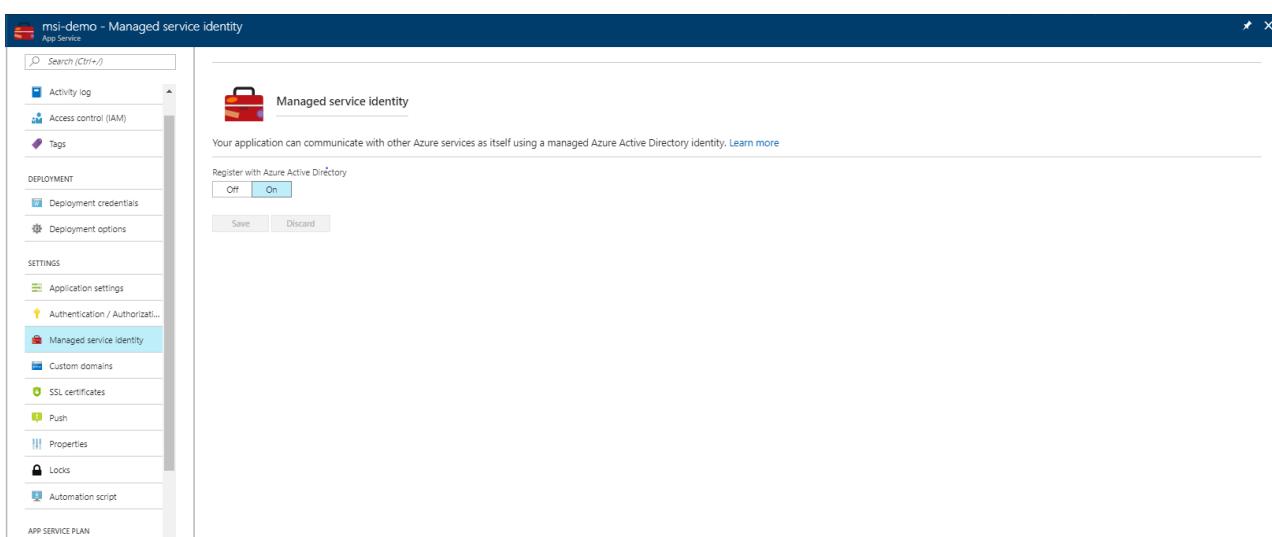
NOTE

Only the primary slot for a site will receive the identity. Managed service identities for deployment slots are not yet supported.

Using the Azure portal

To set up a managed service 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 service identity**.
4. Switch **Register with Azure Active Directory** to **On**. Click **Save**.



Using the Azure CLI

To set up a managed service identity using the Azure CLI, you will need to use the `az webapp assign-identity` 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 CLI 2.0](#) (2.0.21 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 `assign-identity` command to create the identity for this application:

```
az webapp assign-identity --name myApp --resource-group 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"  
}
```

This 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": {
    "tenantId": "<TENANTID>",
    "principalId": "<PRINCIPALID>"
}
```

Where `<TENANTID>` and `<PRINCIPALID>` are replaced with GUIDs. The `tenantId` property identifies what AAD tenant the application belongs to. The `principalId` is a unique identifier for the application's new identity. Within AAD, the application has the same name that you gave to your App Service or Azure Functions instance.

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 Managed Service Identity 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 service 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 2.0](#), 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 [Microsoft.Azure.KeyVault](#) NuGet packages to your application.

2. Add the following code to your application:

```
using Microsoft.Azure.Services.AppAuthentication;
using Microsoft.Azure.KeyVault;
// ...
var azureServiceTokenProvider = new AzureServiceTokenProvider();
string accessToken = await azureServiceTokenProvider.GetAccessTokenAsync("https://management.azure.com/");
// 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 REST protocol

An app with a managed service identity has two environment variables defined:

- MSI_ENDPOINT
- MSI_SECRET

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.
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.

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 Managed Service 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) {
    var 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
```

Install the Durable Functions extension and samples (Azure Functions)

12/7/2017 • 4 min to read • [Edit Online](#)

The [Durable Functions](#) extension for Azure Functions is provided in the NuGet package [Microsoft.Azure.WebJobs.Extensions.DurableTask](#). This article shows how to install the package and a set of samples for the following development environments:

- Visual Studio 2017 (Recommended)
- Azure portal

Visual Studio 2017

Visual Studio currently provides the best experience for developing apps that use Durable Functions. Your functions can be run locally and can also be published to Azure. You can start with an empty project or with a set of sample functions.

Prerequisites

- Install the [latest version of Visual Studio](#) (version 15.3 or greater). Include the **Azure development** workload in your setup options.

Start with sample functions

1. Download the [Sample App .zip file for Visual Studio](#). You don't need to add the NuGet reference because the sample project already has it.
2. Install and run [Azure Storage Emulator](#) version 5.2 or later. Alternatively, you can update the *local.appsettings.json* file with real Azure Storage connection strings.
3. Open the project in Visual Studio 2017.
4. For instructions on how to run the sample, start with [Function chaining - Hello sequence sample](#). The sample can be run locally or published to Azure.

Start with an empty project

Follow the same directions as for starting with the sample, but do the following steps instead of downloading the *.zip* file:

1. Create a Function App project.
2. Add the following NuGet package reference to your *.csproj* file:

```
<PackageReference Include="Microsoft.Azure.WebJobs.Extensions.DurableTask" Version="1.0.0-beta" />
```

Visual Studio Code

Visual Studio Code provides a local development experience covering all major platforms - Windows, macOS, and Linux. Your functions can be run locally and can also be published to Azure. You can start with an empty project or with a set of sample functions.

Prerequisites

- Install the [latest version of Visual Studio Code](#)

- Follow the instructions under "Install the Azure Functions Core Tools" at [Code and test Azure Functions locally](#)

IMPORTANT

If you already have the Azure Functions Cross Platform Tools, update them to the latest available version.

- Install and run [Azure Storage Emulator](#) version 5.2 or later. Alternatively, you can update the `local.appsettings.json` file with real Azure Storage connection.

Start with sample functions

- Clone the [Durable Functions repository](#).
- Navigate on your machine to the [C# script samples folder](#).
- Install Azure Functions Durable Extension by running the following in a command prompt / terminal window:

```
func extensions install -p Microsoft.Azure.WebJobs.Extensions.DurableTask -v 1.1.0-beta2
```
- Run Azure Storage Emulator or update the `local.appsettings.json` file with real Azure Storage connection string.
- Open the project in Visual Studio Code.
- For instructions on how to run the sample, start with [Function chaining - Hello sequence sample](#). The sample can be run locally or published to Azure.
- Start the project by running in command prompt / terminal the following command: `bash func host start`

Start with an empty project

- In command prompt / terminal navigate to the folder that will host your function app.
- Install the Azure Functions Durable Extension by running the following in a command prompt / terminal window:

```
func extensions install -p Microsoft.Azure.WebJobs.Extensions.DurableTask -v 1.1.0-beta2
```

- Create a Function App project by running the following command:

```
func init
```

- Run Azure Storage Emulator or update the `local.appsettings.json` file with real Azure Storage connection string.
- Next, create a new function by running the following command and follow the wizard steps:

```
func new
```

IMPORTANT

Currently the Durable Function template is not available but you can start with one of the supported options and then modify the code. Use for reference the samples for [Orchestration Client](#), [Orchestration Trigger](#), and [Activity Trigger](#).

- Open the project folder in Visual Studio Code and continue by modifying the template code.
- Start the project by running in command prompt / terminal the following command: `bash func host start`

Azure portal

If you prefer, you can use the Azure portal for Durable Functions development.

Create an orchestrator function

1. Create a new function app at functions.azure.com.
2. Configure the function app to [use the 2.0 runtime version](#).
3. Create a new function by selecting "**create your own custom function.**".
4. Change the **Language** to **C#**, **Scenario** to **Durable Functions** and select the **Durable Functions Http Starter - C#** template.
5. Under **Extensions not installed**, click **Install** to download the extension from NuGet.org.
6. After the installation is complete, proceed with the creation of an orchestration client function – "**HttpStart**" that is created by selecting **Durable Functions Http Starter - C#** template.
7. Now, create an orchestration function "**HelloSequence**" from **Durable Functions Orchestrator - C#** template.
8. And the last function will be called "**Hello**" from **Durable Functions Activity - C#** template.
9. Go to "**HttpStart**" function and copy its URL.
10. Use Postman or cURL to call the durable function. Before testing, replace in the URL **{functionName}** with the orchestrator function name - **HelloSequence**. No data is required, just use POST verb.

```
curl -X POST https://{{your function app name}}.azurewebsites.net/api/orchestrators/HelloSequence
```

11. Then, call the "**statusQueryGetUri**" endpoint and you see the current status of the Durable Function

```
{  
    "runtimeStatus": "Running",  
    "input": null,  
    "output": null,  
    "createdTime": "2017-12-01T05:37:33Z",  
    "lastUpdatedTime": "2017-12-01T05:37:36Z"  
}
```

12. Continue calling the "**statusQueryGetUri**" endpoint until the status changes to "**Completed**"

```
{  
    "runtimeStatus": "Completed",  
    "input": null,  
    "output": [  
        "Hello Tokyo!",  
        "Hello Seattle!",  
        "Hello London!"  
    ],  
    "createdTime": "2017-12-01T05:38:22Z",  
    "lastUpdatedTime": "2017-12-01T05:38:28Z"  
}
```

Congratulations! Your first durable function is up and running in Azure Portal!

Next steps

Run the function chaining sample

Function chaining in Durable Functions - Hello sequence sample

12/5/2017 • 4 min to read • [Edit Online](#)

Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function. This article explains a sample that uses [Durable Functions](#) to implement function chaining.

Prerequisites

- Follow the instructions in [Install Durable Functions](#) to set up the sample.

The functions

This article explains the following functions in the sample app:

- `E1_HelloSequence` : An orchestrator function that calls `E1_SayHello` multiple times in a sequence. It stores the outputs from the `E1_SayHello` calls and records the results.
- `E1_SayHello` : An activity function that prepends a string with "Hello".

The following sections explain the configuration and code that are used for Azure portal development. The code for Visual Studio development is shown at the end of the article.

function.json file

If you use Visual Studio Code or the Azure portal for development, here's the content of the `function.json` file for the orchestrator function. Most orchestrator `function.json` files look almost exactly like this.

```
{  
  "bindings": [  
    {  
      "name": "context",  
      "type": "orchestrationTrigger",  
      "direction": "in"  
    }  
,  
    {"disabled": false  
  }  
}
```

The important thing is the `orchestrationTrigger` binding type. All orchestrator functions must use this trigger type.

WARNING

To abide by the "no I/O" rule of orchestrator functions, don't use any input or output bindings when using the `orchestrationTrigger` trigger binding. If other input or output bindings are needed, they should instead be used in the context of `activityTrigger` functions, which are called by the orchestrator.

C# script (Visual Studio Code and Azure portal sample code)

Here is the source code:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static async Task<List<string>> Run(DurableOrchestrationContext context)
{
    var outputs = new List<string>();

    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "London"));

    // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
    return outputs;
}
```

All C# orchestration functions must have a parameter of type `DurableOrchestrationContext`, which exists in the `Microsoft.Azure.WebJobs.Extensions.DurableTask` assembly. If you're using C# script, the assembly can be referenced using the `#r` notation. This context object lets you call other *activity* functions and pass input parameters using its `CallActivityAsync` method.

The code calls `E1_SayHello` three times in sequence with different parameter values. The return value of each call is added to the `outputs` list, which is returned at the end of the function.

The `function.json` file for the activity function `E1_SayHello` is similar to that of `E1_HelloSequence` except that it uses an `activityTrigger` binding type instead of an `orchestrationTrigger` binding type.

```
{
  "bindings": [
    {
      "name": "name",
      "type": "activityTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

NOTE

Any function called by an orchestration function must use the `activityTrigger` binding.

The implementation of `E1_SayHello` is a relatively trivial string formatting operation.

```
public static string Run(string name)
{
    return $"Hello {name}!";
}
```

This function has a parameter of type `DurableActivityContext`, which it uses to get the input that was passed to it by the orchestrator function's call to `CallActivityAsync<T>`.

Run the sample

To execute the `E1_HelloSequence` orchestration, send the following HTTP POST request.

```
POST http://{host}/orchestrators/E1_HelloSequence
```

For example, if you're running the sample in a function app named "myfunctionapp", replace "{host}" with "myfunctionapp.azurewebsites.net".

The result is an HTTP 202 response, like this (trimmed for brevity):

```
HTTP/1.1 202 Accepted
Content-Length: 719
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/96924899c16d43b08a536de376ac786b?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

(...trimmed...)
```

At this point, the orchestration is queued up and begins to run immediately. The URL in the `Location` header can be used to check the status of the execution.

```
GET http://{host}/admin/extensions/DurableTaskExtension/instances/96924899c16d43b08a536de376ac786b?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
```

The result is the status of the orchestration. It runs and completes quickly, so you see it in the *Completed* state with a response that looks like this (trimmed for brevity):

```
HTTP/1.1 200 OK
Content-Length: 179
Content-Type: application/json; charset=utf-8

{"runtimeStatus":"Completed","input":null,"output":["Hello Tokyo!","Hello Seattle!","Hello
London!"],"createdTime":"2017-06-29T05:24:57Z","lastUpdatedTime":"2017-06-29T05:24:59Z"}
```

As you can see, the `runtimeStatus` of the instance is *Completed* and the `output` contains the JSON-serialized result of the orchestrator function execution.

NOTE

The HTTP POST endpoint that started the orchestrator function is implemented in the sample app as an HTTP trigger function named "HttpStart". You can implement similar starter logic for other trigger types, like `queueTrigger`, `eventHubTrigger`, or `timerTrigger`.

Look at the function execution logs. The `E1_HelloSequence` function started and completed multiple times due to the replay behavior described in the [overview](#). On the other hand, there were only three executions of `E1_SayHello` since those function executions do not get replayed.

Visual Studio sample code

Here is the orchestration as a single C# file in a Visual Studio project:

```

// Copyright (c) .NET Foundation. All rights reserved.
// Licensed under the MIT License. See License.txt in the project root for license information.

using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;

namespace VSSample
{
    public static class HelloSequence
    {
        [FunctionName("E1_HelloSequence")]
        public static async Task<List<string>> Run(
            [OrchestrationTrigger] DurableOrchestrationContext context)
        {
            var outputs = new List<string>();

            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "London"));

            // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
            return outputs;
        }

        [FunctionName("E1_SayHello")]
        public static string SayHello([ActivityTrigger] string name)
        {
            return $"Hello {name}!";
        }
    }
}

```

Next steps

This sample has demonstrated a simple function-chaining orchestration. The next sample shows how to implement the fan-out/fan-in pattern.

[Run the Fan-out/fan-in sample](#)

Fan-out/fan-in scenario in Durable Functions - Cloud backup example

12/5/2017 • 7 min to read • [Edit Online](#)

Fan-out/fan-in refers to the pattern of executing multiple functions concurrently and then performing some aggregation on the results. This article explains a sample that uses [Durable Functions](#) to implement a fan-in/fan-out scenario. The sample is a durable function that backs up all or some of an app's site content into Azure Storage.

Prerequisites

- Follow the instructions in [Install Durable Functions](#) to set up the sample.
- This article assumes you have already gone through the [Hello Sequence](#) sample walkthrough.

Scenario overview

In this sample, the functions upload all files under a specified directory recursively into blob storage. They also count the total number of bytes that were uploaded.

It's possible to write a single function that takes care of everything. The main problem you would run into is **scalability**. A single function execution can only run on a single VM, so the throughput will be limited by the throughput of that single VM. Another problem is **reliability**. If there's a failure midway through, or if the entire process takes more than 5 minutes, the backup could fail in a partially-completed state. It would then need to be restarted.

A more robust approach would be to write two regular functions: one would enumerate the files and add the file names to a queue, and another would read from the queue and upload the files to blob storage. This is better in terms of throughput and reliability, but it requires you to provision and manage a queue. More importantly, significant complexity is introduced in terms of **state management** and **coordination** if you want to do anything more, like report the total number of bytes uploaded.

A Durable Functions approach gives you all of the mentioned benefits with very low overhead.

The functions

This article explains the following functions in the sample app:

- `E2_BackupSiteContent`
- `E2_GetFileList`
- `E2_CopyFileToBlob`

The following sections explain the configuration and code that are used for Azure portal development. The code for Visual Studio development is shown at the end of the article.

The cloud backup orchestration (Visual Studio Code and Azure portal sample code)

The `E2_BackupSiteContent` function uses the standard *function.json* for orchestrator functions.

```
{
  "bindings": [
    {
      "name": "backupContext",
      "type": "orchestrationTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here is the code that implements the orchestrator function:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static async Task<long> Run(DurableOrchestrationContext backupContext)
{
    string rootDirectory = Environment.ExpandEnvironmentVariables(backupContext.GetInput<string>() ?? "");
    if (string.IsNullOrEmpty(rootDirectory))
    {
        throw new ArgumentException("A directory path is required as an input.");
    }

    if (!Directory.Exists(rootDirectory))
    {
        throw new DirectoryNotFoundException($"Could not find a directory named '{rootDirectory}'.");
    }

    string[] files = await backupContext.CallActivityAsync<string[]>(
        "E2_GetFileList",
        rootDirectory);

    var tasks = new Task<long>[files.Length];
    for (int i = 0; i < files.Length; i++)
    {
        tasks[i] = backupContext.CallActivityAsync<long>(
            "E2_CopyFileToBlob",
            files[i]);
    }

    await Task.WhenAll(tasks);

    long totalBytes = tasks.Sum(t => t.Result);
    return totalBytes;
}
```

This orchestrator function essentially does the following:

1. Takes a `rootDirectory` value as an input parameter.
2. Calls a function to get a recursive list of files under `rootDirectory`.
3. Makes multiple parallel function calls to upload each file into Azure Blob Storage.
4. Waits for all uploads to complete.
5. Returns the sum total bytes that were uploaded to Azure Blob Storage.

Notice the `await Task.WhenAll(tasks);` line. All the calls to the `E2_CopyFileToBlob` function were *not* awaited. This is intentional to allow them to run in parallel. When we pass this array of tasks to `Task.WhenAll`, we get back a task that won't complete *until all the copy operations have completed*. If you're familiar with the Task Parallel Library (TPL) in .NET, then this is not new to you. The difference is that these tasks could be running on multiple VMs concurrently, and the Durable Functions extension ensures that the end-to-end execution is resilient to process recycling.

After awaiting from `Task.WhenAll`, we know that all function calls have completed and have returned values back to us. Each call to `E2_CopyFileToBlob` returns the number of bytes uploaded, so calculating the sum total byte count is a matter of adding all those return values together.

Helper activity functions

The helper activity functions, as with other samples, are just regular functions that use the `activityTrigger` trigger binding. For example, the `function.json` file for `E2_GetFileList` looks like the following:

```
{  
  "bindings": [  
    {  
      "name": "rootDirectory",  
      "type": "activityTrigger",  
      "direction": "in"  
    }  
  ],  
  "disabled": false  
}
```

And here is the implementation:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"  
  
public static string[] Run(string rootDirectory, TraceWriter log)  
{  
    string[] files = Directory.GetFiles(rootDirectory, "*", SearchOption.AllDirectories);  
    log.Info($"Found {files.Length} file(s) under {rootDirectory}.");  
  
    return files;  
}
```

NOTE

You might be wondering why you couldn't just put this code directly into the orchestrator function. You could, but this would break one of the fundamental rules of orchestrator functions, which is that they should never do I/O, including local file system access.

The `function.json` file for `E2_CopyFileToBlob` is similarly simple:

```
{  
  "bindings": [  
    {  
      "name": "filePath",  
      "type": "activityTrigger",  
      "direction": "in"  
    }  
  ],  
  "disabled": false  
}
```

The implementation is also pretty straightforward. It happens to use some advanced features of Azure Functions bindings (that is, the use of the `Binder` parameter), but you don't need to worry about those details for the purpose of this walkthrough.

```

#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Microsoft.WindowsAzure.Storage"

using Microsoft.WindowsAzure.Storage.Blob;

public static async Task<long> Run(
    string filePath,
    Binder binder,
    TraceWriter log)
{
    long byteCount = new FileInfo(filePath).Length;

    // strip the drive letter prefix and convert to forward slashes
    string blobPath = filePath
        .Substring(Path.GetPathRoot(filePath).Length)
        .Replace('\\\\', '/');
    string outputLocation = $"backups/{blobPath}";

    log.Info($"Copying '{filePath}' to '{outputLocation}'. Total bytes = {byteCount}.");

    // copy the file contents into a blob
    using (Stream source = File.Open(filePath, FileMode.Open, FileAccess.Read, FileShare.ReadWrite))
    using (Stream destination = await binder.BindAsync<CloudBlobStream>(
        new BlobAttribute(outputLocation)))
    {
        await source.CopyToAsync(destination);
    }

    return byteCount;
}

```

The implementation loads the file from disk and asynchronously streams the contents into a blob of the same name in the "backups" container. The return value is the number of bytes copied to storage, that is then used by the orchestrator function to compute the aggregate sum.

NOTE

This is a perfect example of moving I/O operations into an `activityTrigger` function. Not only can the work be distributed across many different VMs, but you also get the benefits of checkpointing the progress. If the host process gets terminated for any reason, you know which uploads have already completed.

Run the sample

You can start the orchestration by sending the following HTTP POST request.

```

POST http://{host}/orchestrators/E2_BackupSiteContent
Content-Type: application/json
Content-Length: 20

"D:\\home\\\\LogFiles"

```

NOTE

The `HttpStart` function that you are invoking only works with JSON-formatted content. For this reason, the `Content-Type: application/json` header is required and the directory path is encoded as a JSON string.

This HTTP request triggers the `E2_BackupSiteContent` orchestrator and passes the string `D:\\home\\\\LogFiles` as a parameter. The response provides a link to get the status of the backup operation:

```
HTTP/1.1 202 Accepted
Content-Length: 719
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/b4e9bdcc435d460f8dc008115ff0a8a9?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

(...trimmed...)
```

Depending on how many log files you have in your function app, this operation could take several minutes to complete. You can get the latest status by querying the URL in the `Location` header of the previous HTTP 202 response.

```
GET http://{host}/admin/extensions/DurableTaskExtension/instances/b4e9bdcc435d460f8dc008115ff0a8a9?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
```

```
HTTP/1.1 202 Accepted
Content-Length: 148
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/b4e9bdcc435d460f8dc008115ff0a8a9?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

{"runtimeStatus": "Running", "input": "D:\\home\\\\LogFiles", "output": null, "createdTime": "2017-06-
29T18:50:55Z", "lastUpdatedTime": "2017-06-29T18:51:16Z"}
```

In this case, the function is still running. You are able to see the input that was saved into the orchestrator state and the last updated time. You can continue to use the `Location` header values to poll for completion. When the status is "Completed", you see an HTTP response value similar to the following:

```
HTTP/1.1 200 OK
Content-Length: 152
Content-Type: application/json; charset=utf-8

{"runtimeStatus": "Completed", "input": "D:\\home\\\\LogFiles", "output": 452071, "createdTime": "2017-06-
29T18:50:55Z", "lastUpdatedTime": "2017-06-29T18:51:26Z"}
```

Now you can see that the orchestration is complete and approximately how much time it took to complete. You also see a value for the `output` field, which indicates that around 450 KB of logs were uploaded.

Visual Studio sample code

Here is the orchestration as a single C# file in a Visual Studio project:

```
// Copyright (c) .NET Foundation. All rights reserved.
// Licensed under the MIT License. See License.txt in the project root for license information.

using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.WindowsAzure.Storage.Blob;

namespace VSSample
{
    public static class BackupSiteContent
    {
        [FunctionName("E2_BackupSiteContent")]
        public static async Task<long> Run(
```

```

    [OrchestrationTrigger] DurableOrchestrationContext backupContext)
{
    string rootDirectory = backupContext.GetInput<string>()?.Trim();
    if (string.IsNullOrEmpty(rootDirectory))
    {
        rootDirectory = Directory.GetParent(typeof(BackupSiteContent).Assembly.Location).FullName;
    }

    string[] files = await backupContext.CallActivityAsync<string[]>(
        "E2_GetFileList",
        rootDirectory);

    var tasks = new Task<long>[files.Length];
    for (int i = 0; i < files.Length; i++)
    {
        tasks[i] = backupContext.CallActivityAsync<long>(
            "E2_CopyFileToBlob",
            files[i]);
    }

    await Task.WhenAll(tasks);

    long totalBytes = tasks.Sum(t => t.Result);
    return totalBytes;
}

[FunctionName("E2_GetFileList")]
public static string[] GetFileList(
    [ActivityTrigger] string rootDirectory,
    TraceWriter log)
{
    log.Info($"Searching for files under '{rootDirectory}'...");
    string[] files = Directory.GetFiles(rootDirectory, "*", SearchOption.AllDirectories);
    log.Info($"Found {files.Length} file(s) under {rootDirectory}.");

    return files;
}

[FunctionName("E2_CopyFileToBlob")]
public static async Task<long> CopyFileToBlob(
    [ActivityTrigger] string filePath,
    Binder binder,
    TraceWriter log)
{
    long byteCount = new FileInfo(filePath).Length;

    // strip the drive letter prefix and convert to forward slashes
    string blobPath = filePath
        .Substring(Path.GetPathRoot(filePath).Length)
        .Replace('\\', '/');
    string outputLocation = $"backups/{blobPath}";

    log.Info($"Copying '{filePath}' to '{outputLocation}'. Total bytes = {byteCount}.");

    // copy the file contents into a blob
    using (Stream source = File.Open(filePath, FileMode.Open, FileAccess.Read, FileShare.Read))
    using (Stream destination = await binder.BindAsync<CloudBlobStream>(
        new BlobAttribute(outputLocation, FileAccess.Write)))
    {
        await source.CopyToAsync(destination);
    }

    return byteCount;
}
}
}

```

Next steps

This sample has shown how to implement the fan-out/fan-in pattern. The next sample shows how to implement the [stateful singleton](#) pattern in an [eternal orchestration](#).

[Run the stateful singleton sample](#)

Stateful singletons in Durable Functions - Counter sample

12/5/2017 • 5 min to read • [Edit Online](#)

Stateful singletons are long-running (potentially eternal) orchestrator functions that can store state and be invoked and queried by other functions. Stateful singletons are similar to the [Actor model](#) in distributed computing.

While not a proper "actor" implementation, orchestrator functions have many of the same runtime characteristics. For example, they are stateful, reliable, single-threaded, location-transparent, and globally addressable. These are characteristics that make real actor implementations especially useful, but without the need for a separate framework.

This article shows how to run the *counter* sample. The sample demonstrates a singleton object that supports *increment* and *decrement* operations and updates its internal state accordingly.

Prerequisites

- Follow the instructions in [Install Durable Functions](#) to set up the sample.
- This article assumes you have already gone through the [Hello Sequence](#) sample walkthrough.

Scenario overview

The counter scenario is surprisingly difficult to implement using regular stateless functions. One of the main challenges you have is managing **concurrency**. Operations like *increment* and *decrement* need to be atomic or else there could be race conditions that cause operations to overwrite each other.

Using a single VM to host the counter data is one option, but this is expensive, and managing **reliability** can be a challenge since a single VM could be periodically rebooted. You could alternatively use a distributed platform with synchronization tools like blob leases to help manage concurrency, but this introduces a great deal of **complexity**.

Durable Functions makes this kind of scenario trivial to implement because orchestration instances are affinitized to a single VM and orchestrator function execution is always single-threaded. Not only that, but they are long-running, stateful, and can react to external events. The sample code below demonstrates how to implement such a counter as a long-running orchestrator function.

The sample function

This article walks through the **E3_Counter** function in the sample app.

The counter orchestration

The following sections explain the code that is used for Visual Studio Code and Azure Portal development.

C# Script

The function.json file:

```
{
  "bindings": [
    {
      "name": "counterContext",
      "type": "orchestrationTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The run.csx file:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static async Task<int> Run(
    DurableOrchestrationContext counterContext,
    TraceWriter log)
{
    int counterState = counterContext.GetInput<int>();
    log.Info($"Current counter state is {counterState}. Waiting for next operation.");

    string operation = await counterContext.WaitForExternalEvent<string>("operation");
    log.Info($"Received '{operation}' operation.");

    operation = operation?.ToLowerInvariant();
    if (operation == "incr")
    {
        counterState++;
    }
    else if (operation == "decr")
    {
        counterState--;
    }

    if (operation != "end")
    {
        counterContext.ContinueAsNew(counterState);
    }

    return counterState;
}
```

Precompiled C#

The following sections explain the code that is used for Visual Studio development.

Here is the code that implements the orchestrator function:

```

// Copyright (c) .NET Foundation. All rights reserved.
// Licensed under the MIT License. See License.txt in the project root for license information.

using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;

namespace VSSample
{
    public static class Counter
    {
        [FunctionName("E3_Counter")]
        public static async Task<int> Run(
            [OrchestrationTrigger] DurableOrchestrationContext counterContext,
            TraceWriter log)
        {
            int counterState = counterContext.GetInput<int>();
            log.Info($"Current counter state is {counterState}. Waiting for next operation.");

            string operation = await counterContext.WaitForExternalEvent<string>("operation");
            log.Info($"Received '{operation}' operation.");

            operation = operation?.ToLowerInvariant();
            if (operation == "incr")
            {
                counterState++;
            }
            else if (operation == "decr")
            {
                counterState--;
            }

            if (operation != "end")
            {
                counterContext.ContinueAsNew(counterState);
            }

            return counterState;
        }
    }
}

```

Explanation of the code

This orchestrator function essentially does the following:

1. Listens for an external event named *operation* using [WaitForExternalEvent](#).
2. Increments or decrements the `counterState` local variable depending on the operation requested.
3. Restarts the orchestrator using the [ContinueAsNew](#) method, setting the latest value of `counterState` as the new input.
4. Continues running forever or until an *end* message is received.

This is an example of an *eternal orchestration* — that is, one that potentially never ends. It responds to messages sent to it by the [RaiseEventAsync](#) method, which can be called by any non-orchestrator function.

One unique characteristic of this orchestrator function is that it effectively has no history: the `ContinueAsNew` method resets the history after each processed event. This is the preferred way to implement an orchestrator which has an arbitrary lifetime. Using a `while` loop could cause the orchestrator function's history to grow unbounded, resulting in unnecessarily high memory usage.

NOTE

The `ContinueAsNew` method has other use-cases besides eternal orchestrations. For more information, see [Eternal Orchestrations](#).

Run the sample

You can start the orchestration by sending the following HTTP POST request. To allow `counterState` to start at zero (the default value for `int`), there is no content in this request.

```
POST http://{host}/orchestrators/E3_Counter
Content-Length: 0
```

```
HTTP/1.1 202 Accepted
Content-Length: 719
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

{
  "id": "bcf6fb5067b046fbb021b52ba7deae5a",

  "statusQueryGetUri": "http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7
deae5a?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}",

  "sendEventPostUri": "http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7d
eae5a/raiseEvent/{eventName}?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}",

  "terminatePostUri": "http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7d
eae5a/terminate?reason={text}&taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}"}
```

The **E3_Counter** instance starts and then immediately waits for an event to be sent to it using `RaiseEventAsync` or using the **sendEventUrl** HTTP POST webhook referenced in the 202 response. Valid `eventName` values include *incr*, *decr*, and *end*.

```
POST
http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a/raiseEvent/o
peration?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
Content-Type: application/json
Content-Length: 6

"incr"
```

You can see the results of the "incr" operation by looking at the function logs in the Azure Functions portal.

```
2017-06-29T18:54:53.998 Function started (Id=34e34a61-38b3-4eac-b6e2-98b85e32eec8)
2017-06-29T18:54:53.998 Current counter state is 0. Waiting for next operation.
2017-06-29T18:58:01.458 Function started (Id=b45d6c2f-39f3-42a2-b904-7761b2614232)
2017-06-29T18:58:01.458 Current counter state is 0. Waiting for next operation.
2017-06-29T18:58:01.458 Received 'incr' operation.
2017-06-29T18:58:01.458 Function completed (Success, Id=b45d6c2f-39f3-42a2-b904-7761b2614232, Duration=8ms)
2017-06-29T18:58:11.518 Function started (Id=e1f38cb2-546a-404d-ab22-1ac8f81a93d9)
2017-06-29T18:58:11.518 Current counter state is 1. Waiting for next operation.
```

Similarly, if you check the orchestrator status, you see the `input` field has been set to the updated value (1).

```
GET http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a?  
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey} HTTP/1.1
```

```
HTTP/1.1 202 Accepted  
Content-Length: 129  
Content-Type: application/json; charset=utf-8  
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a?  
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}  
  
{"runtimeStatus": "Running", "input": 1, "output": null, "createdTime": "2017-06-  
29T18:58:01Z", "lastUpdatedTime": "2017-06-29T18:58:11Z"}
```

You can continue sending new operations to this instance and observe that its state gets updated accordingly. If you wish to kill the instance, you can do so by sending an *end* operation.

WARNING

At the time of writing, there are known race conditions when calling `ContinueAsNew` while concurrently processing messages, such as external events or termination requests. For the latest information on these race conditions, see this [GitHub issue](#).

Next steps

This sample has demonstrated how to handle [external events](#) and implement [eternal orchestrations](#) in [stateful singletons](#). The next sample shows how to use external events and [durable timers](#) to handle human interaction.

[Run the human interaction sample](#)

Human interaction in Durable Functions - Phone verification sample

12/5/2017 • 8 min to read • [Edit Online](#)

This sample demonstrates how to build a [Durable Functions](#) orchestration that involves human interaction. Whenever a real person is involved in an automated process, the process must be able to send notifications to the person and receive responses asynchronously. It must also allow for the possibility that the person is unavailable. (This last part is where timeouts become important.)

This sample implements an SMS-based phone verification system. These types of flows are often used when verifying a customer's phone number or for multi-factor authentication (MFA). This is a powerful example because the entire implementation is done using a couple small functions. No external data store, such as a database, is required.

Prerequisites

- Follow the instructions in [Install Durable Functions](#) to set up the sample.
- This article assumes you have already gone through the [Hello Sequence](#) sample walkthrough.

Scenario overview

Phone verification is used to verify that end users of your application are not spammers and that they are who they say they are. Multi-factor authentication is a common use case for protecting user accounts from hackers. The challenge with implementing your own phone verification is that it requires a **stateful interaction** with a human being. An end user is typically provided some code (e.g. a 4-digit number) and must respond **in a reasonable amount of time**.

Ordinary Azure Functions are stateless (as are many other cloud endpoints on other platforms), so these types of interactions will involve explicitly managing state externally in a database or some other persistent store. In addition, the interaction must be broken up into multiple functions that can be coordinated together. For example, you need at least one function for deciding on a code, persisting it somewhere, and sending it to the user's phone. Additionally, you need at least one other function to receive a response from the user and somehow map it back to the original function call in order to do the code validation. A timeout is also an important aspect to ensure security. This can get fairly complex pretty quickly.

The complexity of this scenario is greatly reduced when you use Durable Functions. As you will see in this sample, an orchestrator function can manage the stateful interaction very easily and without involving any external data stores. Because orchestrator functions are *durable*, these interactive flows are also highly reliable.

Configuring Twilio integration

This sample involves using the [Twilio](#) service to send SMS messages to a mobile phone. Azure Functions already has support for Twilio via the [Twilio binding](#), and the sample uses that feature.

The first thing you need is a Twilio account. You can create one free at <https://www.twilio.com/try-twilio>. Once you have an account, add the following three **app settings** to your function app.

APP SETTING NAME	VALUE DESCRIPTION
------------------	-------------------

APP SETTING NAME	VALUE DESCRIPTION
TwilioAccountSid	The SID for your Twilio account
TwilioAuthToken	The Auth token for your Twilio account
TwilioPhoneNumber	The phone number associated with your Twilio account. This is used to send SMS messages.

The functions

This article walks through the following functions in the sample app:

- **E4_SmsPhoneVerification**
- **E4_SendSmsChallenge**

The following sections explain the configuration and code that are used for Azure portal development. The code for Visual Studio development is shown at the end of the article.

The SMS verification orchestration (Visual Studio Code and Azure portal sample code)

The **E4_SmsPhoneVerification** function uses the standard *function.json* for orchestrator functions.

```
{
  "bindings": [
    {
      "name": "context",
      "type": "orchestrationTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here is the code that implements the function:

```

#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

using System.Threading;

public static async Task<bool> Run(DurableOrchestrationContext context)
{
    string phoneNumber = context.GetInput<string>();
    if (string.IsNullOrEmpty(phoneNumber))
    {
        throw new ArgumentNullException(
            nameof(phoneNumber),
            "A phone number input is required.");
    }

    int challengeCode = await context.CallActivityAsync<int>(
        "E4_SendSmsChallenge",
        phoneNumber);

    using (var timeoutCts = new CancellationTokenSource())
    {
        // The user has 90 seconds to respond with the code they received in the SMS message.
        DateTime expiration = context.CurrentUtcDateTime.AddSeconds(90);
        Task timeoutTask = context.CreateTimer(expiration, timeoutCts.Token);

        bool authorized = false;
        for (int retryCount = 0; retryCount <= 3; retryCount++)
        {
            Task<int> challengeResponseTask =
                context.WaitForExternalEvent<int>("SmsChallengeResponse");

            Task winner = await Task.WhenAny(challengeResponseTask, timeoutTask);
            if (winner == challengeResponseTask)
            {
                // We got back a response! Compare it to the challenge code.
                if (challengeResponseTask.Result == challengeCode)
                {
                    authorized = true;
                    break;
                }
            }
            else
            {
                // Timeout expired
                break;
            }
        }

        if (!timeoutTask.IsCompleted)
        {
            // All pending timers must be complete or canceled before the function exits.
            timeoutCts.Cancel();
        }
    }

    return authorized;
}
}

```

Once started, this orchestrator function does the following:

1. Gets a phone number to which it will *send* the SMS notification.
2. Calls **E4_SendSmsChallenge** to send an SMS message to the user and returns back the expected 4-digit challenge code.
3. Creates a durable timer that triggers 90 seconds from the current time.
4. In parallel with the timer, waits for a **SmsChallengeResponse** event from the user.

The user receives an SMS message with a four digit code. They have 90 seconds to send that same 4-digit code back to the orchestrator function instance to complete the verification process. If they submit the wrong code, they get an additional three tries to get it right (within the same 90 second window).

NOTE

It may not be obvious at first, but this orchestrator function is completely deterministic. This is because the `CurrentUtcDateTime` property is used to calculate the timer expiration time, and this property returns the same value on every replay at this point in the orchestrator code. This is important to ensure that the same `winner` results from every repeated call to `Task.WhenAny`.

WARNING

It's important to cancel timers using a `CancellationTokenSource` if you no longer need them to expire, as in the example above when a challenge response is accepted.

Send the SMS message

The **E4_SendSmsChallenge** function uses the Twilio binding to send the SMS message with the 4-digit code to the end user. The `function.json` is defined as follows:

```
{
  "bindings": [
    {
      "name": "phoneNumber",
      "type": "activityTrigger",
      "direction": "in"
    },
    {
      "type": "twilioSms",
      "name": "message",
      "from": "%TwilioPhoneNumber%",
      "accountSidSetting": "TwilioAccountSid",
      "authTokenSetting": "TwilioAuthToken",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

And here is the code that generates the 4-digit challenge code and sends the SMS message:

```

#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Microsoft.Azure.WebJobs.Extensions.Twilio"
#r "Newtonsoft.Json"
#r "Twilio"

using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

public static int Run(
    string phoneNumber,
    TraceWriter log,
    out CreateMessageOptions message)
{
    // Get a random number generator with a random seed (not time-based)
    var rand = new Random(Guid.NewGuid().GetHashCode());
    int challengeCode = rand.Next(10000);

    log.Info($"Sending verification code {challengeCode} to {phoneNumber}.");

    message = new CreateMessageOptions(new PhoneNumber(phoneNumber));
    message.Body = $"Your verification code is {challengeCode:0000}";

    return challengeCode;
}

```

This **E4_SendSmsChallenge** function only gets called once, even if the process crashes or gets replayed. This is good because you don't want the end user getting multiple SMS messages. The `challengeCode` return value is automatically persisted, so the orchestrator function always knows what the correct code is.

Run the sample

Using the HTTP-triggered functions included in the sample, you can start the orchestration by sending the following HTTP POST request.

```

POST http://{host}/orchestrators/E4_SmsPhoneVerification
Content-Length: 14
Content-Type: application/json

"+1425XXXXXXX"

```

```

HTTP/1.1 202 Accepted
Content-Length: 695
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/741c65651d4c40cea29acdd5bb47baf1?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

{"id":"741c65651d4c40cea29acdd5bb47baf1","statusQueryGetUri":"http://{host}/admin/extensions/DurableTaskExtens
ion/instances/741c65651d4c40cea29acdd5bb47baf1?taskHub=DurableFunctionsHub&connection=Storage&code=
{systemKey}","sendEventPostUri":"http://{host}/admin/extensions/DurableTaskExtension/instances/741c65651d4c40c
ea29acdd5bb47baf1/raiseEvent/{eventName}?taskHub=DurableFunctionsHub&connection=Storage&code=
{systemKey}","terminatePostUri":"http://{host}/admin/extensions/DurableTaskExtension/instances/741c65651d4c40c
ea29acdd5bb47baf1/terminate?reason={text}&taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}"}

```

The orchestrator function receives the supplied phone number and immediately sends it an SMS message with a randomly generated 4-digit verification code — for example, 2168. The function then waits 90 seconds for a response.

To reply with the code, you can use `RaiseEventAsync` inside another function or invoke the **sendEventUrl** HTTP POST webhook referenced in the 202 response above, replacing `{eventName}` with the name of the event,

```
SmsChallengeResponse :
```

```
POST  
http://{host}/admin/extensions/DurableTaskExtension/instances/741c65651d4c40cea29acdd5bb47baf1/raiseEvent/SmsChallengeResponse?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}  
Content-Length: 4  
Content-Type: application/json  
  
2168
```

If you send this before the timer expires, the orchestration completes and the `output` field is set to `true`, indicating a successful verification.

```
GET http://{host}/admin/extensions/DurableTaskExtension/instances/741c65651d4c40cea29acdd5bb47baf1?  
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
```

```
HTTP/1.1 200 OK  
Content-Length: 144  
Content-Type: application/json; charset=utf-8  
  
{ "runtimeStatus": "Completed", "input": "+1425XXXXXX", "output": true, "createdTime": "2017-06-  
29T19:10:49Z", "lastUpdatedTime": "2017-06-29T19:12:23Z" }
```

If you let the timer expire, or if you enter the wrong code four times, you can query for the status and see a `false` orchestration function output, indicating that phone verification failed.

```
HTTP/1.1 200 OK  
Content-Type: application/json; charset=utf-8  
Content-Length: 145  
  
{ "runtimeStatus": "Completed", "input": "+1425XXXXXX", "output": false, "createdTime": "2017-06-  
29T19:20:49Z", "lastUpdatedTime": "2017-06-29T19:22:23Z" }
```

Visual Studio sample code

Here is the orchestration as a single C# file in a Visual Studio project:

```
// Copyright (c) .NET Foundation. All rights reserved.  
// Licensed under the MIT License. See License.txt in the project root for license information.  
  
using System;  
using System.Threading;  
using System.Threading.Tasks;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Host;  
#if NETSTANDARD2_0  
using Twilio.Rest.Api.V2010.Account;  
using Twilio.Types;  
#else  
using Twilio;  
#endif  
  
namespace VSSample  
{  
    public static class PhoneVerification  
    {  
        [FunctionName("E4_SmsPhoneVerification")]  
        public static async Task<bool> Run(  
            [OrchestrationTrigger] DurableOrchestrationContext context)
```

```

{
    string phoneNumber = context.GetInput<string>();
    if (string.IsNullOrEmpty(phoneNumber))
    {
        throw new ArgumentNullException(
            nameof(phoneNumber),
            "A phone number input is required.");
    }

    int challengeCode = await context.CallActivityAsync<int>(
        "E4_SendSmsChallenge",
        phoneNumber);

    using (var timeoutCts = new CancellationTokenSource())
    {
        // The user has 90 seconds to respond with the code they received in the SMS message.
        DateTime expiration = context.CurrentUtcDateTime.AddSeconds(90);
        Task timeoutTask = context.CreateTimer(expiration, timeoutCts.Token);

        bool authorized = false;
        for (int retryCount = 0; retryCount <= 3; retryCount++)
        {
            Task<int> challengeResponseTask =
                context.WaitForExternalEvent<int>("SmsChallengeResponse");

            Task winner = await Task.WhenAny(challengeResponseTask, timeoutTask);
            if (winner == challengeResponseTask)
            {
                // We got back a response! Compare it to the challenge code.
                if (challengeResponseTask.Result == challengeCode)
                {
                    authorized = true;
                    break;
                }
            }
            else
            {
                // Timeout expired
                break;
            }
        }

        if (!timeoutTask.IsCompleted)
        {
            // All pending timers must be complete or canceled before the function exits.
            timeoutCts.Cancel();
        }
    }

    return authorized;
}
}

[FunctionName("E4_SendSmsChallenge")]
public static int SendSmsChallenge(
    [ActivityTrigger] string phoneNumber,
    TraceWriter log,
    [TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting = "TwilioAuthToken", From =
"%TwilioPhoneNumber%")]
#if NETSTANDARD2_0
    out CreateMessageOptions message)
#else
    out SMSMessage message)
#endif
{
    // Get a random number generator with a random seed (not time-based)
    var rand = new Random(Guid.NewGuid().GetHashCode());
    int challengeCode = rand.Next(10000);

    log.Info($"Sending verification code {challengeCode} to {phoneNumber}.");
}

```

```
#if NETSTANDARD2_0
    message = new CreateMessageOptions(new PhoneNumber(phoneNumber));
#else
    message = new SMSMessage { To = phoneNumber };
#endif
    message.Body = $"Your verification code is {challengeCode:0000}";

    return challengeCode;
}
}
```

Next steps

This sample has demonstrated some of the advanced capabilities of Durable Functions, notably `WaitForExternalEvent` and `CreateTimer`. You've seen how these can be combined with `Task.WaitAny` to implement a reliable timeout system, which is often useful for interacting with real people. You can learn more about how to use Durable Functions by reading a series of articles that offer in-depth coverage of specific topics.

[Go to the first article in the series](#)

Azure Blob storage bindings for Azure Functions

1/9/2018 • 17 min 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.](#)
- [Azure Functions developer reference.](#)
- [C#, F#, Node, or Java developer reference.](#)
- [Azure Functions triggers and bindings concepts.](#)

NOTE

[Blob-only storage accounts](#) are not supported. Blob storage triggers and bindings require a general-purpose storage account.

Trigger

Use a Blob storage trigger to start a function when a new or updated blob is detected. The blob contents are provided as input to the function.

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 after a function app has gone idle. After the function app is running, blobs are processed immediately. To avoid this initial delay, consider one of the following options:

- Use an App Service plan with Always On enabled.
- Use another mechanism to trigger the blob processing, such as a queue message that contains the blob name. For an example, see the [blob input bindings example later in this article](#).

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

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, TraceWriter log)
{
    log.Info($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");
}
```

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 [C# script \(.csx\)](#) 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",
            "connection": "MyStorageAccountAppSetting"
        }
    ]
}
```

The [configuration](#) section explains these properties.

Here's C# script code that binds to a `Stream`:

```
public static void Run(Stream myBlob, TraceWriter log)
{
    log.Info($"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, TraceWriter log)
{
    log.Info($"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",
            "connection": "MyStorageAccountAppSetting"
        }
    ]
}
```

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 - attributes

In [C# class libraries](#), use the following attributes to configure a blob trigger:

- [BlobTriggerAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs](#)

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](#), defined in NuGet package [Microsoft.Azure.WebJobs](#)

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
type	n/a	Must be set to <code>blobTrigger</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. 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 container to monitor. May be a blob name pattern .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
connection	Connection	<p>The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set <code>connection</code> to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave <code>connection</code> empty, the Functions runtime uses the default Storage connection string in the app setting that is named <code>AzureWebJobsStorage</code>.</p> <p>The connection string must be for a general-purpose storage account, not a blob-only storage account.</p>

When you're developing locally, app settings go into the [local.settings.json file](#).

Trigger - usage

In C# and C# script, access the blob data by using a method parameter such as `T 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:

- `Stream`
- `TextReader`
- `Byte[]`
- `string`
- `ICloudBlob` (requires "inout" binding direction in `function.json`)
- `CloudBlockBlob` (requires "inout" binding direction in `function.json`)
- `CloudPageBlob` (requires "inout" binding direction in `function.json`)
- `CloudAppendBlob` (requires "inout" binding direction in `function.json`)

As noted, some of these types require an `inout` binding direction in `function.json`. This direction is not supported by the standard editor in the Azure portal, so you must use the advanced editor.

If text blobs are expected, you can bind to the `string` type. This 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.

In JavaScript, access the input blob data using `context.bindings.<name>`.

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](#).

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*.

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 value of the `blobname` and `blobextension` variables in function code are *original-Blob 1* and *.txt*.

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 [Cloud Blob](#) 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.

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.

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 - polling for large containers

If the blob container being monitored contains more than 10,000 blobs, 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. 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 \(.csx\)](#)
- [JavaScript](#)

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,
    TraceWriter log)
{
    log.Info($"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, TraceWriter log)
{
    log.Info($"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 - attributes

In [C# class libraries](#), use the [BlobAttribute](#), which is defined in NuGet package [Microsoft.Azure.WebJobs](#).

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,  
  TraceWriter log)  
{  
  log.Info($"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,
    TraceWriter log)
{
    log.Info($"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# class libraries and C# script, access the blob by using a method parameter such as `Stream 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:

- `TextReader`
- `string`
- `Byte[]`
- `Stream`
- `CloudBlobContainer`
- `CloudBlobDirectory`
- `ICloudBlob` (requires "inout" binding direction in `function.json`)
- `CloudBlockBlob` (requires "inout" binding direction in `function.json`)
- `CloudPageBlob` (requires "inout" binding direction in `function.json`)
- `CloudAppendBlob` (requires "inout" binding direction in `function.json`)

As noted, some of these types require an `inout` binding direction in `function.json`. This direction is not supported by the standard editor in the Azure portal, so you must use the advanced editor.

If you are reading text blobs, you can bind to a `string` type. This type 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.

In JavaScript, access the blob data using `context.bindings.<name>`.

Output

Use Blob storage output bindings to write blobs.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

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.

```

[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)
{
    var imageBuilder = ImageResizer.ImageBuilder.Current;
    var size = imageDimensionsTable[ImageSize.Small];

    imageBuilder.Build(image, imageSmall,
        new ResizeSettings(size.Item1, size.Item2, FitMode.Max, null), false);

    image.Position = 0;
    size = imageDimensionsTable[ImageSize.Medium];

    imageBuilder.Build(image, imageMedium,
        new ResizeSettings(size.Item1, size.Item2, FitMode.Max, null), false);
}

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, TraceWriter log)
{
    log.Info($"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 - attributes

In [C# class libraries](#), use the [BlobAttribute](#), which is defined in NuGet package [Microsoft.Azure.WebJobs](#).

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.

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# class libraries and C# script, access the blob by using a method parameter such as `Stream 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:

- `TextWriter`
- `out string`
- `out Byte[]`
- `CloudBlobStream`
- `Stream`
- `CloudBlobContainer`
- `CloudBlobDirectory`
- `ICloudBlob` (requires "inout" binding direction in *function.json*)
- `CloudBlockBlob` (requires "inout" binding direction in *function.json*)
- `CloudPageBlob` (requires "inout" binding direction in *function.json*)
- `CloudAppendBlob` (requires "inout" binding direction in *function.json*)

As noted, some of these types require an `inout` binding direction in *function.json*. This direction is not supported by the standard editor in the Azure portal, so you must use the advanced editor.

If you are reading text blobs, you can bind to a `string` type. This type 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.

In JavaScript, access the blob data using `context.bindings.<name>`.

Next steps

[Go to a quickstart that uses a Blob storage trigger](#)

[Learn more about Azure functions triggers and bindings](#)

Azure Cosmos DB bindings for Azure Functions

1/18/2018 • 15 min 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.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Create your first function](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

Trigger

The Azure Cosmos DB Trigger uses the [Azure Cosmos DB Change Feed](#) to listen for changes across partitions. The change feed publishes inserts and updates, not deletions.

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

Trigger - C# example

The following example shows a [C# function](#) that triggers from a specific database and collection.

```
using System.Collections.Generic;
using Microsoft.Azure.Documents;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;

[FunctionName("DocumentUpdates")]
public static void Run(
    [CosmosDBTrigger("database", "collection", ConnectionStringSetting = "myCosmosDB")]
    IReadOnlyList<Document> documents,
    TraceWriter log)
{
    log.Info("Documents modified " + documents.Count);
    log.Info("First document Id " + documents[0].Id);
}
```

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.Verbose("Documents modified " + documents.Count);
    log.Verbose("First document Id " + documents[0].Id);
}
```

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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.DocumentDB](#).

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.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
leaseDatabaseName	LeaseDatabaseName	(Optional) The name of the database that holds the collection used to store leases. When not set, the value of the <code>databaseName</code> setting is used. This parameter is automatically set when the binding is created in the portal.
leaseCollectionName	LeaseCollectionName	(Optional) The name of the collection used to store leases. When not set, the value <code>leases</code> is used.
createLeaseCollectionIfNotExists	CreateLeaseCollectionIfNotExists	(Optional) When set to <code>true</code> , the leases collection is automatically created when it doesn't already exist. The default value is <code>false</code> .
leasesCollectionThroughput	LeasesCollectionThroughput	(Optional) Defines the 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.
	LeaseOptions	Configure lease options by setting properties in an instance of the ChangeFeedHostOptions class.

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. Otherwise, only one of the functions will be triggered.

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 retrieves 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.

NOTE

Don't use Azure Cosmos DB input or output bindings if you're using MongoDB API on a Cosmos DB account. Data corruption is possible.

Input - example 1

See the language-specific example that reads a single document:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

Input - C# example

The following example shows a [C# function](#) that retrieves a single document from a specific database and collection.

First, `Id` and `Maker` values for a `CarReview` instance are passed to a queue. The Cosmos DB binding uses `Id` and `Maker` from the queue message to retrieve the document from the database.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;

namespace CosmosDB
{
    public static class SingleEntry
    {
        [FunctionName("SingleEntry")]
        public static void Run(
            [QueueTrigger("car-reviews", Connection = "StorageConnectionString")] CarReview carReview,
            [DocumentDB("cars", "car-reviews", PartitionKey = "{maker}", Id= "{id}",
ConnectionStringSetting = "CarReviewsConnectionString")] CarReview document,
            TraceWriter log)
        {
            log.Info( $"Selected Review - {document?.Review}");
        }
    }
}
```

Here's the `CarReview` POCO:

```
public class CarReview
{
    public string Id { get; set; }
    public string Maker { get; set; }
    public string Description { get; set; }
    public string Model { get; set; }
    public string Image { get; set; }
    public string Review { get; set; }
}
```

Input - C# script example

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.";  
}
```

Input - F# example

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 - JavaScript example

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": "inputDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger_payload_property}",
  "partitionKey": "{queueTrigger_payload_property}",
  "connection": "MyAccount_COSMOSDB",
  "direction": "in"
}
```

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();
};
```

Input - example 2

See the language-specific example that reads multiple documents:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

Input - C# example 2

The following example shows a [C# function](#) that executes a SQL query. To use the `SqlQuery` parameter, you need to install the latest beta version of `Microsoft.Azure.WebJobs.Extensions.DocumentDB` NuGet package.

```

using System.Net;
using System.Net.Http;
using System.Collections.Generic;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;

[FunctionName("CosmosDBSample")]
public static HttpResponseMessage Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get")] HttpRequestMessage req,
    [DocumentDB("test", "test", ConnectionStringSetting = "CosmosDB", SqlQuery = "SELECT top 2 * FROM c
order by c._ts desc")] IEnumerable<object> documents)
{
    return req.CreateResponse(HttpStatusCode.OK, documents);
}

```

Input - C# script example 2

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; }
}

```

Input - JavaScript example 2

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();  
};
```

Input - attributes

In [C# class libraries](#), use the [DocumentDB](#) attribute, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.DocumentDB](#).

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.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
id	Id	The ID of the document to retrieve. This property supports bindings parameters. To learn more, see Bind to custom input properties in a binding expression . 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 #ccc; padding: 5px; margin-top: 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.
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.

NOTE

Don't use Azure Cosmos DB input or output bindings if you're using MongoDB API on a Cosmos DB account. Data corruption is possible.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)

- [JavaScript](#)

Output - C# example

The following example shows a [C# function](#) that adds a document to a database, using data provided in message from Queue storage.

```
using System;
using Microsoft.Azure.WebJobs;

[FunctionName("QueueToDocDB")]
public static void Run(
    [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] string myQueueItem,
    [DocumentDB("ToDoList", "Items", Id = "id", ConnectionStringSetting = "myCosmosDB")] out dynamic
document)
{
    document = new { Text = myQueueItem, id = Guid.NewGuid() };
}
```

Output - C# script example

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
    };
}

```

To create multiple documents, you can bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

Output - F# example

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 - JavaScript example

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();
};
```

Output - attributes

In [C# class libraries](#), use the [DocumentDB](#) attribute, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.DocumentDB](#).

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 documentdb .
direction		Must be set to out .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
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.

Next steps

[Go to a quickstart that uses a Cosmos DB trigger](#)

[Learn more about serverless database computing with Cosmos DB](#)

[Learn more about Azure functions triggers and bindings](#)

Azure Event Hubs bindings for Azure Functions

1/2/2018 • 9 min 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.](#)
- [Azure Functions developer reference.](#)
- [C#, F#, Node, or Java developer reference.](#)
- [Azure Functions triggers and bindings concepts.](#)

Trigger

Use the Event Hubs trigger to respond to an event sent to an event hub event stream. You must have read access to the event hub to set up the trigger.

When an Event Hubs trigger function is triggered, the message that triggers it is passed into the function as a string.

Trigger - scaling

Each instance of an Event Hub-Triggered Function is backed by only 1 EventProcessorHost (EPH) instance. Event Hubs ensures that only 1 EPH can get a lease on a given partition.

For example, suppose we begin with the following setup and assumptions for an Event Hub:

1. 10 partitions.
2. 1000 events distributed evenly across all partitions => 100 messages in each partition.

When your function is first enabled, there is only 1 instance of the function. Let's call this function instance Function_0. Function_0 will have 1 EPH that manages to get a lease on all 10 partitions. It will start reading events from partitions 0-9. From this point forward, one of the following will happen:

- **Only 1 function instance is needed** - Function_0 is able to process all 1000 before the Azure Functions' scaling logic kicks in. Hence, all 1000 messages are processed by Function_0.
- **Add 1 more function instance** - Azure Functions' scaling logic determines that Function_0 has more messages than it can process, so a new instance, Function_1, is created. Event Hubs detects that a new EPH instance is trying read messages. Event Hubs will start load balancing the partitions across the EPH instances, e.g., partitions 0-4 are assigned to Function_0 and partitions 5-9 are assigned to Function_1.
- **Add N more function instances** - Azure Functions' scaling logic determines that both Function_0 and Function_1 have more messages than they can process. It will scale again for Function_2...N, where N is greater than the Event Hub partitions. Event Hubs will load balance the partitions across Function_0...9 instances.

Unique to Azure Functions' current scaling logic is the fact that N is greater than the number of partitions. This is done to ensure that there are always instances of EPH readily available to quickly get a lock on the partition(s) as they become available from other instances. Users are only charged for the resources used when the function instance executes, and are not charged for this over-provisioning.

If all function executions succeed without errors, checkpoints are added to the associated storage account. When check-pointing succeeds, all 1000 messages should never be retrieved again.

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

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 = "EventHubConnection")] string
myEventHubMessage, TraceWriter log)
{
    log.Info($"C# Event Hub trigger function processed a message: {myEventHubMessage}");
}
```

To get access to the event metadata, bind to an [EventData](#) object (requires a `using` statement for `Microsoft.ServiceBus.Messaging`).

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnection")] EventData
myEventHubMessage, TraceWriter log)
{
    log.Info($"{Encoding.UTF8.GetString(myEventHubMessage.GetBytes())}");
}
```

To receive events in a batch, make `string` or `EventData` an array:

```
[FunctionName("EventHubTriggerCSharp")]
public static void Run([EventHubTrigger("samples-workitems", Connection = "EventHubConnection")] string[]
eventHubMessages, TraceWriter log)
{
    foreach (var message in eventHubMessages)
    {
        log.Info($"C# Event Hub trigger function processed a message: {message}");
    }
}
```

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.

Here's the binding data in the `function.json` file:

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionString"
}
```

Here's the C# script code:

```
using System;

public static void Run(string myEventHubMessage, TraceWriter log)
{
    log.Info($"C# Event Hub trigger function processed a message: {myEventHubMessage}");
}
```

To get access to the event metadata, bind to an [EventData](#) object (requires a using statement for `Microsoft.ServiceBus.Messaging`).

```
#r "Microsoft.ServiceBus"
using System.Text;
using Microsoft.ServiceBus.Messaging;

public static void Run(EventData myEventHubMessage, TraceWriter log)
{
    log.Info($"{Encoding.UTF8.GetString(myEventHubMessage.GetBytes())}");
}
```

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# Event Hub trigger function processed 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.

Here's the binding data in the `function.json` file:

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionString"
}
```

Here's the F# code:

```
let Run(myEventHubMessage: string, log: TraceWriter) =
    log.Info(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 logs the message body of the event hub trigger.

Here's the binding data in the `function.json` file:

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionString"
}
```

Here's the JavaScript code:

```
module.exports = function (context, myEventHubMessage) {
    context.log('Node.js eventhub trigger function processed work item', myEventHubMessage);
    context.done();
};
```

Trigger - attributes

In [C# class libraries](#), use the `EventHubTriggerAttribute` attribute, which is defined in NuGet package `Microsoft.Azure.WebJobs.ServiceBus`.

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 = "EventHubConnection")] string
myEventHubMessage, 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 `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.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
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	The name of the event hub.
consumerGroup	ConsumerGroup	An optional property that sets the consumer group used to subscribe to events in the hub. If omitted, the <code>\$Default</code> consumer group is used.
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 at least read permissions to activate the trigger.

When you're developing locally, app settings go into the [local.settings.json file](#).

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.

Output - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

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 = "EventHubConnection")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
    return $"{DateTime.Now}";
}
```

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.

Here's the binding data in the *function.json* file:

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSend",
    "direction": "out"
}
```

Here's C# script code that creates one message:

```
using System;

public static void Run(TimerInfo myTimer, out string outputEventHubMessage, TraceWriter log)
{
    String msg = $"TimerTriggerCSharp1 executed at: {DateTime.Now}";
    log.Verbose(msg);
    outputEventHubMessage = msg;
}
```

Here's C# script code that creates multiple messages:

```

public static void Run(TimerInfo myTimer, ICollector<string> outputEventHubMessage, TraceWriter log)
{
    string message = $"Event Hub message created at: {DateTime.Now}";
    log.Info(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.

Here's the binding data in the *function.json* file:

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSend",
    "direction": "out"
}
```

Here's the F# code:

```

let Run(myTimer: TimerInfo, outputEventHubMessage: byref<string>, log: TraceWriter) =
    let msg = sprintf "TimerTriggerFSharp1 executed at: %s" DateTime.Now.ToString()
    log.Verbose(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.

Here's the binding data in the *function.json* file:

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSend",
    "direction": "out"
}
```

Here's JavaScript code that sends a single message:

```

module.exports = function (context, myTimer) {
    var timeStamp = new Date().toISOString();
    context.log('Event Hub message created at: ', timeStamp);
    context.bindings.outputEventHubMessage = "Event Hub 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 = 'Event Hub message created at: ' + timeStamp;

    context.bindings.outputEventHubMessage = [];

    context.bindings.outputEventHubMessage.push("1 " + message);
    context.bindings.outputEventHubMessage.push("2 " + message);
    context.done();
};


```

Output - attributes

For [C# class libraries](#), use the [EventHubAttribute](#) attribute, which is defined in NuGet package [Microsoft.Azure.WebJobs.ServiceBus](#).

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 = "EventHubConnection")]
public static string Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, 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 [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	The name of the event hub.
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`.

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Functions External File bindings (experimental)

1/5/2018 • 9 min to read • [Edit Online](#)

This article shows how to manipulate files from different SaaS providers (such as Dropbox or Google Drive) in Azure Functions. Azure Functions supports trigger, input, and output bindings for external files. These bindings create API connections to SaaS providers, or use existing API connections from your Function App's resource group.

IMPORTANT

The External File bindings are experimental and might never reach Generally Available (GA) status. They are included only in Azure Functions 1.x, and there are no plans to add them to Azure Functions 2.x. For scenarios that require access to data in SaaS providers, consider using [logic apps that call into functions](#). See the [Logic Apps File System connector](#).

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Create your first function](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

Available file connections

CONNECTOR	TRIGGER	INPUT	OUTPUT
Box	x	x	x
Dropbox	x	x	x
FTP	x	x	x
OneDrive	x	x	x
OneDrive for Business	x	x	x
SFTP	x	x	x
Google Drive		x	x

NOTE

External File connections can also be used in [Azure Logic Apps](#).

Trigger

The external file trigger lets you monitor a remote folder and run your function code when changes are detected.

Trigger - example

See the language-specific example:

- [C# script](#)
- [JavaScript](#)

Trigger - C# script example

The following example shows an external file trigger binding in a `function.json` file and a [C# script function](#) that uses the binding. The function logs the contents of each file that is added to the monitored folder.

Here's the binding data in the `function.json` file:

```
{  
    "disabled": false,  
    "bindings": [  
        {  
            "name": "myFile",  
            "type": "apiHubFileTrigger",  
            "direction": "in",  
            "path": "samples-workitems",  
            "connection": "<name of external file connection>"  
        }  
    ]  
}
```

Here's the C# script code:

```
public static void Run(string myFile, TraceWriter log)  
{  
    log.Info($"C# File trigger function processed: {myFile}");  
}
```

Trigger - JavaScript example

The following example shows an external file trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function logs the contents of each file that is added to the monitored folder.

Here's the binding data in the `function.json` file:

```
{  
    "disabled": false,  
    "bindings": [  
        {  
            "name": "myFile",  
            "type": "apiHubFileTrigger",  
            "direction": "in",  
            "path": "samples-workitems",  
            "connection": "<name of external file connection>"  
        }  
    ]  
}
```

Here's the JavaScript code:

```
module.exports = function(context) {  
    context.log('Node.js File trigger function processed', context.bindings.myFile);  
    context.done();  
};
```

Trigger - configuration

The following table explains the binding configuration properties that you set in the `function.json` file.

FUNCTION.JSON PROPERTY	DESCRIPTION
type	Must be set to <code>apiHubFileTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the event item in function code.
connection	Identifies the app setting that stores the connection string. The app setting is created automatically when you add a connection in the integrate UI in the Azure portal.
path	The folder to monitor, and optionally a name pattern.

Name patterns

You can specify a file name pattern in the `path` property. The folder referenced must exist in the SaaS provider.

Examples:

```
"path": "input/original-{name}",
```

This path would find a file named *original-File1.txt* in the *input* folder, and the value of the `name` variable in function code would be `File1.txt`.

Another example:

```
"path": "input/{filename}.{fileextension}",
```

This path would also find a file named *original-File1.txt*, and the value of the `filename` and `fileextension` variables in function code would be *original-File1* and *txt*.

You can restrict the file type of files by using a fixed value for the file extension. For example:

```
"path": "samples/{name}.png",
```

In this case, only *.png* files in the *samples* folder trigger the function.

Curly braces are special characters in name patterns. To specify file names that have curly braces in the name, double the curly braces. For example:

```
"path": "images/{{20140101}}-{name}",
```

This path would find a file named *{20140101}-soundfile.mp3* in the *images* folder, and the `name` variable value in the function code would be *soundfile.mp3*.

Trigger - usage

In C# functions, you bind to the input file data by using a named parameter in your function signature, like `<T> <name>`. Where `T` is the data type that you want to deserialize the data into, and `paramName` is the name you specified in the [trigger JSON](#). In Node.js functions, you access the input file data using `context.bindings.<name>`.

The file can be deserialized into any of the following types:

- Any [Object](#) - useful for JSON-serialized file data. If you declare a custom input type (e.g. `FooType`), Azure Functions attempts to deserialize the JSON data into your specified type.
- String - useful for text file data.

In C# functions, you can also bind to any of the following types, and the Functions runtime attempts to deserialize the file data using that type:

- `string`
- `byte[]`
- `Stream`
- `StreamReader`
- `TextReader`

Trigger - poison files

When an external file trigger function fails, Azure Functions retries that function up to 5 times by default (including the first try) for a given file. If all 5 tries fail, Functions adds a message to a Storage queue named `webjobs-apihubtrigger-poison`. The queue message for poison files is a JSON object that contains the following properties:

- `FunctionId` (in the format `<function app name>.Functions.<function name>`)
- `FileType`
- `FolderName`
- `FileName`
- `ETag` (a file version identifier, for example: "0x8D1DC6E70A277EF")

Input

The Azure external file input binding enables you to use a file from an external folder in your function.

Input - example

See the language-specific example:

- [C# script](#)
- [JavaScript](#)

Input - C# script example

The following example shows external file input and output bindings in a `function.json` file and a [C# script function](#) that uses the binding. The function copies an input file to an output file.

Here's the binding data in the `function.json` file:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnection",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputFile",
      "type": "apiHubFile",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "<name of external file connection>",
      "direction": "in"
    },
    {
      "name": "myOutputFile",
      "type": "apiHubFile",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "<name of external file connection>",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
public static void Run(string myQueueItem, string myInputFile, out string myOutputFile, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    myOutputFile = myInputFile;
}
```

Input - JavaScript example

The following example shows external file input and output bindings in a *function.json* file and a [JavaScript function](#) that uses the binding. The function copies an input file to an output file.

Here's the binding data in the *function.json* file:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnection",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputFile",
      "type": "apiHubFile",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "<name of external file connection>",
      "direction": "in"
    },
    {
      "name": "myOutputFile",
      "type": "apiHubFile",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "<name of external file connection>",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Here's the JavaScript code:

```
module.exports = function(context) {
  context.log('Node.js Queue trigger function processed', context.bindings.myQueueItem);
  context.bindings.myOutputFile = context.bindings.myInputFile;
  context.done();
};
```

Input - configuration

The following table explains the binding configuration properties that you set in the *function.json* file.

FUNCTION.JSON PROPERTY	DESCRIPTION
type	Must be set to <code>apiHubFile</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the event item in function code.
connection	Identifies the app setting that stores the connection string. The app setting is created automatically when you add a connection in the integrate UI in the Azure portal.
path	Must contain the folder name and the file name. For example, if you have a queue trigger in your function, you can use <code>"path": "samples-workitems/{queueTrigger}"</code> to point to a file in the <code>samples-workitems</code> folder with a name that matches the file name specified in the trigger message.

Input - usage

In C# functions, you bind to the input file data by using a named parameter in your function signature, like `<T> <name>`. `T` is the data type that you want to deserialize the data into, and `name` is the name you specified in the input binding. In Node.js functions, you access the input file data using `context.bindings.<name>`.

The file can be deserialized into any of the following types:

- Any [Object](#) - useful for JSON-serialized file data. If you declare a custom input type (e.g. `InputType`), Azure Functions attempts to deserialize the JSON data into your specified type.
- String - useful for text file data.

In C# functions, you can also bind to any of the following types, and the Functions runtime attempts to deserialize the file data using that type:

- `string`
- `byte[]`
- `Stream`
- `StreamReader`
- `TextReader`

Output

The Azure external file output binding enables you to write files to an external folder in your function.

Output - example

See the [input binding example](#).

Output - configuration

The following table explains the binding configuration properties that you set in the `function.json` file.

FUNCTION.JSON PROPERTY	DESCRIPTION
type	Must be set to <code>apiHubFile</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>out</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the event item in function code.
connection	Identifies the app setting that stores the connection string. The app setting is created automatically when you add a connection in the integrate UI in the Azure portal.
path	Must contain the folder name and the file name. For example, if you have a queue trigger in your function, you can use <code>"path": "samples-workitems/{queueTrigger}"</code> to point to a file in the <code>samples-workitems</code> folder with a name that matches the file name specified in the trigger message.

Output - usage

In C# functions, you bind to the output file by using the named `out` parameter in your function signature, like `out <T> <name>`, where `T` is the data type that you want to serialize the data into, and `<name>` is the name you specified in the output binding. In Node.js functions, you access the output file using `context.bindings.<name>`.

You can write to the output file using any of the following types:

- Any `Object` - useful for JSON-serialization. If you declare a custom output type (e.g. `out OutputType paramName`), Azure Functions attempts to serialize object into JSON. If the output parameter is null when the function exits, the Functions runtime creates a file as a null object.
- `String` - (`out string paramName`) useful for text file data. the Functions runtime creates a file only if the string parameter is non-null when the function exits.

In C# functions you can also output to any of the following types:

- `TextWriter`
- `Stream`
- `CloudFileStream`
- `ICloudFile`
- `CloudBlockFile`
- `CloudPageFile`

Next steps

[Learn more about Azure functions triggers and bindings](#)

External Table binding for Azure Functions (experimental)

1/5/2018 • 4 min to read • [Edit Online](#)

This article explains how to work with tabular data on SaaS providers, such as Sharepoint and Dynamics, in Azure Functions. Azure Functions supports input and output bindings for external tables.

IMPORTANT

The External Table binding is experimental and might never reach Generally Available (GA) status. It is included only in Azure Functions 1.x, and there are no plans to add it to Azure Functions 2.x. For scenarios that require access to data in SaaS providers, consider using [logic apps that call into functions](#).

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Create your first function](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

API connections

Table bindings leverage external API connections to authenticate with third-party SaaS providers.

When assigning a binding you can either create a new API connection or use an existing API connection within the same resource group.

Available API connections (tables)

CONNECTOR	TRIGGER	INPUT	OUTPUT
DB2		x	x
Dynamics 365 for Operations		x	x
Dynamics 365		x	x
Dynamics NAV		x	x
Google Sheets		x	x
Informix		x	x
Dynamics 365 for Financials		x	x
MySQL		x	x

CONNECTOR	TRIGGER	INPUT	OUTPUT
Oracle Database		x	x
Common Data Service		x	x
Salesforce		x	x
SharePoint		x	x
SQL Server		x	x
Teradata		x	x
UserVoice		x	x
Zendesk		x	x

NOTE

External Table connections can also be used in [Azure Logic Apps](#).

Creating an API connection: step by step

1. In the Azure portal page for your function app, select the plus sign (+) to create a function.
2. In the **Scenario** box, select **Experimental**.
3. Select **External table**.
4. Select a language.
5. Under **External Table connection**, select an existing connection or select **new**.
6. For a new connection, configure the settings, and select **Authorize**.
7. Select **Create** to create the function.
8. Select **Integrate > External Table**.
9. Configure the connection to use your target table. These settings will vary between SaaS providers.
Examples are included in the following section.

Example

This example connects to a table named "Contact" with Id, LastName, and FirstName columns. The code lists the Contact entities in the table and logs the first and last names.

Here's the *function.json* file:

```
{
  "bindings": [
    {
      "type": "manualTrigger",
      "direction": "in",
      "name": "input"
    },
    {
      "type": "apiHubTable",
      "direction": "in",
      "name": "table",
      "connection": "ConnectionAppSettingsKey",
      "dataSetName": "default",
      "tableName": "Contact",
      "entityId": ""
    }
  ],
  "disabled": false
}
```

Here's the C# script code:

```
#r "Microsoft.Azure.ApiHub.Sdk"
#r "Newtonsoft.Json"

using System;
using Microsoft.Azure.ApiHub;

//Variable name must match column type
//Variable type is dynamically bound to the incoming data
public class Contact
{
    public string Id { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}

public static async Task Run(string input, ITable<Contact> table, TraceWriter log)
{
    //Iterate over every value in the source table
    ContinuationToken continuationToken = null;
    do
    {
        //retrieve table values
        var contactsSegment = await table.ListEntitiesAsync(
            continuationToken: continuationToken);

        foreach (var contact in contactsSegment.Items)
        {
            log.Info(string.Format("{0} {1}", contact.FirstName, contact.LastName));
        }

        continuationToken = contactsSegment.ContinuationToken;
    }
    while (continuationToken != null);
}
```

SQL Server data source

To create a table in SQL Server to use with this example, here's a script. `dataSetName` is "default."

```

CREATE TABLE Contact
(
    Id int NOT NULL,
    LastName varchar(20) NOT NULL,
    FirstName varchar(20) NOT NULL,
    CONSTRAINT PK_Contact_Id PRIMARY KEY (Id)
)
GO
INSERT INTO Contact(Id, LastName, FirstName)
    VALUES (1, 'Bitt', 'Prad')
GO
INSERT INTO Contact(Id, LastName, FirstName)
    VALUES (2, 'Glooney', 'George')
GO

```

Google Sheets data source

To create a table to use with this example in Google Docs, create a spreadsheet with a worksheet named `Contact`. The connector cannot use the spreadsheet display name. The internal name (in bold) needs to be used as `dataSetName`, for example: `docs.google.com/spreadsheets/d/ 1UIz545JF_cx6Chm_5HpSPV0enU4DZh4bDxbFgJOSMz0` Add the column names `Id`, `LastName`, `FirstName` to the first row, then populate data on subsequent rows.

Salesforce

To use this example with Salesforce, `dataSetName` is "default."

Configuration

The following table explains the binding configuration properties that you set in the `function.json` file.

FUNCTION.JSON PROPERTY	DESCRIPTION
type	Must be set to <code>apiHubTable</code> . This property is set automatically when you create the trigger in the Azure portal.
direction	Must be set to <code>in</code> . This property is set automatically when you create the trigger in the Azure portal.
name	The name of the variable that represents the event item in function code.
connection	Identifies the app setting that stores the API connection string. The app setting is created automatically when you add an API connection in the integrate UI.
dataSetName	The name of the dataset that contains the table to read.
tableName	The name of the table
entityId	Must be empty for table bindings.

A tabular connector provides data sets, and each data set contains tables. The name of the default data set is "default." The titles for a dataset and a table in various SaaS providers are listed below:

CONNECTOR	DATASET	TABLE
SharePoint	Site	SharePoint List

CONNECTOR	DATASET	TABLE
SQL	Database	Table
Google Sheet	Spreadsheet	Worksheet
Excel	Excel file	Sheet

Next steps

[Learn more about Azure functions triggers and bindings](#)

Azure Functions HTTP and webhook bindings

1/19/2018 • 15 min to read • [Edit Online](#)

This article explains how to work with HTTP bindings in Azure Functions. Azure Functions supports HTTP triggers and output bindings.

An HTTP trigger can be customized to respond to [webhooks](#). A webhook trigger accepts only a JSON payload and validates the JSON. There are special versions of the webhook trigger that make it easier to handle webhooks from certain providers, such as GitHub and Slack.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- [Create your first function](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

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, review the article [Improper Instantiation antipattern](#).

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 responds to the request with an HTTP 200 OK status code and an empty body. To modify the response, configure an [HTTP output binding](#).

Trigger - example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

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.

```

[FunctionName("HttpTriggerCSharp")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]HttpRequestMessage req,
    TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    // parse query parameter
    string name = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
        .Value;

    // Get request body
    dynamic data = await req.Content.ReadAsAsync<object>();

    // Set name to query string or body data
    name = name ?? data?.name;

    return name == null
        ? req.CreateResponse(HttpStatusCode.BadRequest, "Please pass a name on the query string or in the
request body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
}

```

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 binding data in the `function.json` file:

```
{
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
    "authLevel": "function"
},
```

The [configuration](#) section explains these properties.

Here's C# script code that binds to `HttpRequestMessage`:

```

using System.Net;
using System.Threading.Tasks;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
{
    log.Info($"C# HTTP trigger function processed a request. RequestUri={req.RequestUri}");

    // 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;

    return name == null
        ? req.CreateResponse(HttpStatusCode.BadRequest, "Please pass a name on the query string or in the
request body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
}

```

You can bind to a custom object instead of `HttpRequestMessage`. This object is created from the body of the request, 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;

public static string Run(CustomObject req, TraceWriter log)
{
    return "Hello " + req?.name;
}

public class CustomObject {
    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 binding data in the `function.json` file:

```
{
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
    "authLevel": "function"
},
```

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 binding data in the `function.json` file:

```
{
  "name": "req",
  "type": "httpTrigger",
  "direction": "in",
  "authLevel": "function"
},
```

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: 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"
        };
    }
    context.done();
};

```

Trigger - webhook example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

Webhook - C# example

The following example shows a [C# function](#) that sends an HTTP 200 in response to a generic JSON request.

```

[FunctionName("HttpTriggerCSharp")]
public static HttpResponseMessage Run([HttpTrigger(AuthorizationLevel.Anonymous, WebHookType =
"genericJson")] HttpRequestMessage req)
{
    return req.CreateResponse(HttpStatusCode.OK);
}

```

Webhook - C# script example

The following example shows a webhook trigger binding in a *function.json* file and a [C# script function](#) that uses the binding. The function logs GitHub issue comments.

Here's the binding data in the *function.json* file:

```
{
    "webHookType": "github",
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
},

```

The [configuration](#) section explains these properties.

Here's the C# script code:

```

#r "Newtonsoft.Json"

using System;
using System.Net;
using System.Threading.Tasks;
using Newtonsoft.Json;

public static async Task<object> Run(HttpRequestMessage req, TraceWriter log)
{
    string jsonContent = await req.Content.ReadAsStringAsync();
    dynamic data = JsonConvert.DeserializeObject(jsonContent);

    log.Info($"WebHook was triggered! Comment: {data.comment.body}");

    return req.CreateResponse(HttpStatusCode.OK, new {
        body = $"New GitHub comment: {data.comment.body}"
    });
}

```

Webhook - F# example

The following example shows a webhook trigger binding in a `function.json` file and an [F# function](#) that uses the binding. The function logs GitHub issue comments.

Here's the binding data in the `function.json` file:

```
{
    "webHookType": "github",
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
},
}
```

The [configuration](#) section explains these properties.

Here's the F# code:

```

open System.Net
open System.Net.Http
open FSharp.Interop.Dynamic
open Newtonsoft.Json

type Response = {
    body: string
}

let Run(req: HttpRequestMessage, log: TraceWriter) =
    async {
        let! content = req.Content.ReadAsStringAsync() |> Async.AwaitTask
        let data = content |> JsonConvert.DeserializeObject
        log.Info(sprintf "GitHub WebHook triggered! %s" data?comment?body)
        return req.CreateResponse(
            HttpStatusCode.OK,
            { body = sprintf "New GitHub comment: %s" data?comment?body })
    } |> Async.StartAsTask

```

Webhook - JavaScript example

The following example shows a webhook trigger binding in a `function.json` file and a [JavaScript function](#) that uses the binding. The function logs GitHub issue comments.

Here's the binding data in the `function.json` file:

```
{  
    "webHookType": "github",  
    "name": "req",  
    "type": "httpTrigger",  
    "direction": "in",  
},
```

The [configuration](#) section explains these properties.

Here's the JavaScript code:

```
...  
  
```javascript  
module.exports = function (context, data) {
 context.log('GitHub WebHook triggered!', data.comment.body);
 context.res = { body: 'New GitHub comment: ' + data.comment.body };
 context.done();
};
```

## Trigger - attributes

In [C# class libraries](#), use the [HttpTrigger](#) attribute, defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.Http](#).

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 HttpResponseMessage Run(
 [HttpTrigger(AuthorizationLevel.Anonymous, WebHookType = "genericJson")] HttpRequestMessage 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
<b>type</b>	n/a	Required - must be set to <a href="#">httpTrigger</a> .
<b>direction</b>	n/a	Required - must be set to <a href="#">in</a> .
<b>name</b>	n/a	Required - the variable name used in function code for the request or request body.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>authLevel</b>	<b>AuthLevel</b>	<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"> <li>• <code>anonymous</code> —No API key is required.</li> <li>• <code>function</code> —A function-specific API key is required. This is the default value if none is provided.</li> <li>• <code>admin</code> —The master key is required.</li> </ul> <p>For more information, see the section about <a href="#">authorization keys</a>.</p>
<b>methods</b>	<b>Methods</b>	An array of the HTTP methods to which the function responds. If not specified, the function responds to all HTTP methods. See <a href="#">customize the http endpoint</a> .
<b>route</b>	<b>Route</b>	Defines the route template, controlling to which request URLs your function responds. The default value if none is provided is <code>&lt;functionname&gt;</code> . For more information, see <a href="#">customize the http endpoint</a> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>webHookType</b>	<b>WebHookType</b>	<p>Configures the HTTP trigger to act as a <a href="#">webhook</a> 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"> <li>• <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.</li> <li>• <code>github</code>—The function responds to <a href="#">GitHub webhooks</a>. Do not use the <code>authLevel</code> property with GitHub webhooks. For more information, see the GitHub webhooks section later in this article.</li> <li>• <code>slack</code>—The function responds to <a href="#">Slack webhooks</a>. Do not use the <code>authLevel</code> property with Slack webhooks. For more information, see the Slack webhooks section later in this article.</li> </ul>

## Trigger - usage

For C# and F# functions, you can declare the type of your trigger input to be either `HttpRequestMessage` or a custom type. If you choose `HttpRequestMessage`, you get full access to the request object. For a custom type, Functions 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](#).

### 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

For an example, see [Create a function triggered by a GitHub webhook](#).

## 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](#).

## Customize the HTTP endpoint

By default when you create a function for an HTTP trigger, or WebHook, 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<HttpResponseMessage> Run(HttpRequestMessage req, string category, int? id,
 TraceWriter log)
{
 if (id == null)
 return req.CreateResponse(HttpStatusCode.OK, $"All {category} items were requested.");
 else
 return req.CreateResponse(HttpStatusCode.OK, $"{category} item with id = {id} has been
requested.");
}

```

Here is Node.js 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: 200, /* Defaults to 200 */
 body: "All " + category + " items were requested."
 };
 }
 else {
 context.res = {
 // status: 200, /* 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": ""
 }
}
```

## Authorization keys

HTTP triggers let you use keys for added security. A standard HTTP trigger can use these as an API key, requiring the key to be present on the request. Webhooks can use keys to authorize requests in a variety of ways, depending on what the provider supports.

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 functions in the portal and select "Manage."

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.

The **master key** is a default host key named "\_master" that is defined for each function app. This key cannot be revoked. It provides administrative access to the runtime APIs. Using `"authLevel": "admin"` in the binding JSON requires this key to be presented on the request; any other key results in authorization failure.

#### IMPORTANT

Due to the elevated permissions 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.

### API key authorization

By default, an HTTP trigger requires an API key in the HTTP request. So your HTTP request normally looks like the following:

```
https://<yourapp>.azurewebsites.net/api/<function>?code=<ApiKey>
```

The key can be included in a query string variable named `code`, as above, or it can 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](#).

### Keys and webhooks

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, however 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://<yourapp>.azurewebsites.net/api/<funcname>?clientid=<keyname>`.
- **Request header:** The provider passes the key name in the `x-functions-clientid` header.

### Trigger - limits

The HTTP request length is limited to 100K (102,400) bytes, and the URL length is limited to 4k (4,096) bytes. These limits are specified by the `httpRuntime` element of the runtime's [Web.config file](#).

### Trigger - host.json properties

The `host.json` file contains settings that control HTTP trigger behavior.

```
{
 "http": {
 "routePrefix": "api",
 "maxOutstandingRequests": 20,
 "maxConcurrentRequests": 10,
 "dynamicThrottlesEnabled": false
 }
}
```

PROPERTY	DEFAULT	DESCRIPTION
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.
maxOutstandingRequests	-1	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows callers to employ time-based retry strategies, and also helps you to control maximum request latencies. This only controls queuing that occurs within the script host execution path. Other queues such as the ASP.NET request queue will still be in effect and unaffected by this setting. The default is unbounded.
maxConcurrentRequests	-1	The maximum number of http functions that will be executed in parallel. This allows you to control concurrency, which can help manage resource utilization. For example, you might have an http function that uses a lot of system resources (memory/cpu/sockets) such that it causes issues when concurrency is too high. Or you might have a function that makes outbound requests to a third party service, and those calls need to be rate limited. In these cases, applying a throttle here can help. The default is unbounded.
dynamicThrottlesEnabled	false	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels.

## 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.

## Output - configuration

For C# class libraries, there are no output-specific binding configuration properties. To send an HTTP response, make the function return type `HttpResponseMessage` or `Task<HttpResponseMessage>`.

For other languages, an HTTP output binding is defined as a JSON object in the `bindings` array of `function.json`, as shown in the following example:

```
{
 "name": "res",
 "type": "http",
 "direction": "out"
}
```

The following table explains the binding configuration properties that you set in the `function.json` file.

PROPERTY	DESCRIPTION
<b>type</b>	Must be set to <code>http</code> .
<b>direction</b>	Must be set to <code>out</code> .
<b>name</b>	The variable name used in function code for the response.

## Output - usage

You can use the output parameter to respond to the HTTP or webhook caller. You can also use the language-standard response patterns. For example responses, see the [trigger example](#) and the [webhook example](#).

## Next steps

[Learn more about Azure functions triggers and bindings](#)

# Microsoft Graph bindings for Azure Functions

1/4/2018 • 29 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

## NOTE

Microsoft Graph bindings are currently in preview.

## 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 [Target the version 2.0 runtime](#).

### 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.

#### **NOTE**

Each extension only needs to be installed once per function app. 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 these NuGet packages:

- [Microsoft.Azure.WebJobs.Extensions.AuthTokens](#)
- [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#)

#### **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;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, string graphToken, TraceWriter 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;

 let options = {
 uri: 'https://graph.microsoft.com/v1.0/me/',
 headers: {
 'Authorization': token
 }
 };

 rp(options)
 .then(function(profile) {
 context.res = {
 body: profile
 };
 context.done();
 })
 .catch(function(err) {
 context.res = {
 status: 500,
 body: err
 };
 context.done();
 });
};
```

## Auth token - attributes

In [C# class libraries](#), use the [Token](#) attribute, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.AuthTokens](#).

## 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
<b>name</b>		Required - the variable name used in function code for the auth token. See <a href="#">Using an auth token input binding from code</a> .
<b>type</b>		Required - must be set to <code>token</code> .
<b>direction</b>		Required - must be set to <code>in</code> .
<b>identity</b>	<b>Identity</b>	Required - The identity that will be used to perform the action. Can be one of the following values: <ul style="list-style-type: none"><li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li><li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li><li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li><li>• <code>clientCredentials</code> - Uses the identity of the function app.</li></ul>
<b>userId</b>	<b>UserId</b>	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.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
<b>Resource</b>	<b>resource</b>	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.

## 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;

public static IActionResult Run(HttpContext req, string[][] excelTableData, TraceWriter 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, which is defined in NuGet package

`Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph`.

### **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
<b>name</b>		Required - the variable name used in function code for the Excel table. See <a href="#">Using an Excel table input binding from code</a> .
<b>type</b>		Required - must be set to <code>excel</code> .
<b>direction</b>		Required - must be set to <code>in</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>identity</b>	<b>Identity</b>	<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"> <li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li> <li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li> <li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li> <li>• <code>clientCredentials</code> - Uses the identity of the function app.</li> </ul>
<b>userId</b>	<b>UserId</b>	Needed if and only if <code>identity</code> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <code>identity</code> is set to <code>userFromToken</code> . A token valid for the function app.
<b>path</b>	<b>Path</b>	Required - the path in OneDrive to the Excel workbook.
<b>worksheetName</b>	<b>WorksheetName</b>	The worksheet in which the table is found.
<b>tableName</b>	<b>TableName</b>	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;

public static async Task Run(HttpRequest req, IAsyncCollector<object> newExcelRow, TraceWriter 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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#).

## 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
<code>name</code>		Required - the variable name used in function code for the auth token. See <a href="#">Using an Excel table output binding from code</a> .
<code>type</code>		Required - must be set to <code>excel</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>direction</b>		Required - must be set to <code>out</code> .
<b>identity</b>	<b>Identity</b>	<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"> <li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li> <li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li> <li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li> <li>• <code>clientCredentials</code> - Uses the identity of the function app.</li> </ul>
<b>userId</b>	<b>userId</b>	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.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
<b>path</b>	<b>Path</b>	Required - the path in OneDrive to the Excel workbook.
<b>worksheetName</b>	<b>WorksheetName</b>	The worksheet in which the table is found.
<b>tableName</b>	<b>TableName</b>	The name of the table. If not specified, the contents of the worksheet will be used.
<b>updateType</b>	<b>UpdateType</b>	<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"> <li>• <code>update</code> - Replaces the contents of the table in OneDrive.</li> <li>• <code>append</code> - Adds the payload to the end of the table in OneDrive by creating new rows.</li> </ul>

### 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;

public static void Run(HttpRequestMessage req, Stream myOneDriveFile, TraceWriter log)
{
 log.Info(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": "onederive",
 "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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#).

#### 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
<b>name</b>		Required - the variable name used in function code for the file. See <a href="#">Using a OneDrive file input binding from code</a> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>type</b>		Required - must be set to <code>onederive</code> .
<b>direction</b>		Required - must be set to <code>in</code> .
<b>identity</b>	<b>Identity</b>	<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"> <li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li> <li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li> <li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li> <li>• <code>clientCredentials</code> - Uses the identity of the function app.</li> </ul>
<b>userId</b>	<b>UserId</b>	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.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
<b>path</b>	<b>Path</b>	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;

public static async Task Run(HttpRequest req, TraceWriter 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);
 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": "onederive",
 "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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#).

## 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
<b>name</b>		Required - the variable name used in function code for file. See <a href="#">Using a OneDrive file output binding from code</a> .
<b>type</b>		Required - must be set to <code>onederive</code> .
<b>direction</b>		Required - must be set to <code>out</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>identity</b>	<b>Identity</b>	Required - The identity that will be used to perform the action. Can be one of the following values: <ul style="list-style-type: none"> <li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li> <li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li> <li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li> <li>• <code>clientCredentials</code> - Uses the identity of the function app.</li> </ul>
<b>userId</b>	<b>userId</b>	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.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
<b>path</b>	<b>Path</b>	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;

public static void Run(HttpRequest req, out Message message, TraceWriter 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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#).

### 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
<b>name</b>		Required - the variable name used in function code for the mail message. See <a href="#">Using an Outlook message output binding from code</a> .
<b>type</b>		Required - must be set to <code>outlook</code> .
<b>direction</b>		Required - must be set to <code>out</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>identity</b>	<b>Identity</b>	<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"> <li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li> <li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li> <li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li> <li>• <code>clientCredentials</code> - Uses the identity of the function app.</li> </ul>
<b>userId</b>	<b>UserId</b>	Needed if and only if <code>identity</code> is set to <code>userFromId</code> . A user principal ID associated with a previously logged-in user.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <code>identity</code> 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;

public static async Task Run(Message msg, TraceWriter log)
{
 log.Info("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.Info($"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, which is defined in NuGet package `Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph`.

## 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
<code>name</code>		Required - the variable name used in function code for the mail message. See <a href="#">Using an Outlook message output binding from code</a> .
<code>type</code>		Required - must be set to <code>graphWebhook</code> .
<code>direction</code>		Required - must be set to <code>trigger</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>resourceType</b>	<b>ResourceType</b>	<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"> <li>• <code>#Microsoft.Graph.Message</code> - changes made to Outlook messages.</li> <li>• <code>#Microsoft.Graph.DriveItem</code> - changes made to OneDrive root items.</li> <li>• <code>#Microsoft.Graph.Contact</code> - changes made to personal contacts in Outlook.</li> <li>• <code>#Microsoft.Graph.Event</code> - changes made to Outlook calendar items.</li> </ul>

#### 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;

public static async Task Run(HttpRequest req, string[] existingSubscriptions, IAsyncCollector<string>
subscriptionsToDelete, TraceWriter log)
{
 log.Info("C# HTTP trigger function processed a request.");
 foreach (var subscription in existingSubscriptions)
 {
 log.Info($"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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#).

## 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
<b>name</b>		Required - the variable name used in function code for the mail message. See <a href="#">Using an Outlook message output binding from code</a> .
<b>type</b>		Required - must be set to <a href="#">graphWebhookSubscription</a> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>direction</b>		Required - must be set to <code>in</code> .
<b>filter</b>	<b>Filter</b>	If set to <code>userFromRequest</code> , then the binding will only retrieve subscriptions owned by the calling user (valid only with <a href="#">HTTP trigger</a> ).

## 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": "graphwebhook",
 "name": "clientState",
 "direction": "out",
 "action": "create",
 "listen": "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;

public static HttpResponseMessage run(HttpRequestMessage req, out string clientState, TraceWriter log)
{
 log.Info("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": "graphwebhook",
 "name": "clientState",
 "direction": "out",
 "action": "create",
 "listen": "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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MicrosoftGraph](#).

### 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
<b>name</b>		Required - the variable name used in function code for the mail message. See <a href="#">Using an Outlook message output binding from code</a> .
<b>type</b>		Required - must be set to <a href="#">graphWebhookSubscription</a> .
<b>direction</b>		Required - must be set to <a href="#">out</a> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>identity</b>	<b>Identity</b>	<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"> <li>• <code>userFromRequest</code> - Only valid with <a href="#">HTTP trigger</a>. Uses the identity of the calling user.</li> <li>• <code>userFromId</code> - Uses the identity of a previously logged-in user with the specified ID. See the <code>userId</code> property.</li> <li>• <code>userFromToken</code> - Uses the identity represented by the specified token. See the <code>userToken</code> property.</li> <li>• <code>clientCredentials</code> - Uses the identity of the function app.</li> </ul>
<b>userId</b>	<b>UserId</b>	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.
<b>userToken</b>	<b>UserToken</b>	Needed if and only if <i>identity</i> is set to <code>userFromToken</code> . A token valid for the function app.
<b>action</b>	<b>Action</b>	<p>Required - specifies the action the binding should perform. Can be one of the following values:</p> <ul style="list-style-type: none"> <li>• <code>create</code> - Registers a new subscription.</li> <li>• <code>delete</code> - Deletes a specified subscription.</li> <li>• <code>refresh</code> - Refreshes a specified subscription to keep it from expiring.</li> </ul>
<b>subscriptionResource</b>	<b>SubscriptionResource</b>	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 <a href="#">Working with webhooks in Microsoft Graph</a> .
<b>changeType</b>	<b>ChangeType</b>	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;

public static void Run(TimerInfo myTimer, string[] existingSubscriptions, ICollector<string>
subscriptionsToRefresh, TraceWriter log)
{
 // This template uses application permissions and requires consent from an Azure Active Directory
 admin.
 // See https://go.microsoft.com/fwlink/?linkid=858780
 log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
 foreach (var subscription in existingSubscriptions)
 {
 log.Info($"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(`Deleting subscription ${existing[i]}`);
 todelete.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;

public static async Task Run(TimerInfo myTimer, UserSubscription[] existingSubscriptions, IBinder binder,
TraceWriter log)
{
 log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
 foreach (var subscription in existingSubscriptions)
 {
 // binding in code to allow dynamic identity
 using (var subscriptionsToRefresh = await binder.BindAsync<IAsyncCollector<string>>(
 new GraphWebhookSubscriptionAttribute() {
 Action = "refresh",
 Identity = "userFromId",
 UserId = subscription.UserId
 }
))
 {
 log.Info($"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

1/2/2018 • 7 min to read • [Edit Online](#)

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.](#)
- [Azure Functions developer reference.](#)
- [C#, F#, Node, or Java developer reference.](#)
- [Azure Functions triggers and bindings concepts.](#)

## 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"
 }
],
 "disabled": false
}
```

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"
 }
],
 "disabled": false
}
```

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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.MobileApps](#).

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
<b>type</b>		Must be set to "mobileTable"
<b>direction</b>		Must be set to "in"
<b>name</b>		Name of input parameter in function signature.
<b>tableName</b>	<b>TableName</b>	Name of the mobile app's data table

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>id</b>	<b>Id</b>	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.
<b>connection</b>	<b>Connection</b>	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://&lt;appname&gt;.azurewebsites.net</code>
<b>apiKey</b>	<b>ApiKey</b>	The name of an app setting that has your mobile app's API key. Provide the API key if you <a href="#">implement an API key in your Node.js mobile app</a> , or <a href="#">implement an API key in your .NET mobile app</a> . 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"
 }
],
 "disabled": false
}
```

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, which is defined in NuGet package `Microsoft.Azure.WebJobs.Extensions.MobileApps`.

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
<b>type</b>		Must be set to "mobileTable"
<b>direction</b>		Must be set to "out"
<b>name</b>		Name of output parameter in function signature.
<b>tableName</b>	<b>TableName</b>	Name of the mobile app's data table
<b>connection</b>	<b>MobileAppUriSetting</b>	<p>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</p> <div style="background-color: #f0f0f0; padding: 2px;"><code>http://&lt;appname&gt;.azurewebsites.net</code></div>
<b>apiKey</b>	<b>ApiKeySetting</b>	<p>The name of an app setting that has your mobile app's API key. Provide the API key if you <a href="#">implement an API key in your Node.js mobile app backend</a>, or <a href="#">implement an API key in your .NET mobile app backend</a>. 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.</p>

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

1/3/2018 • 7 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

## 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.Info($"Sending APNS notification of a new user");
 dynamic user = JsonConvert.DeserializeObject(myQueueItem);
 string apnsNotificationPayload = "{\"aps\": {\"alert\": \"A new user wants to be added (" +
 user.name + ")\" }}";
 log.Info($"{apnsNotificationPayload}");
 await notification.AddAsync(new AppleNotification(apnsNotificationPayload));
}
```

## Example - GCM native

This C# script example shows how to send a native GCM 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 GCM notification is ...
 // { "data": { "message": "notification message" }}

 log.Info($"Sending GCM notification of a new user");
 dynamic user = JsonConvert.DeserializeObject(myQueueItem);
 string gcmNotificationPayload = "{\"data\": {\"message\": \"A new user wants to be added (" +
 user.name + ")\" }}";
 log.Info($"{gcmNotificationPayload}");
 await notification.AddAsync(new GcmNotification(gcmNotificationPayload));
}
```

## 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, which is defined in NuGet package [Microsoft.Azure.WebJobs.Extensions.NotificationHubs](#).

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
<b>type</b>	n/a	Must be set to "notificationHub".
<b>direction</b>	n/a	Must be set to "out".
<b>name</b>	n/a	Variable name used in function code for the notification hub message.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>tagExpression</b>	<b>TagExpression</b>	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 <a href="#">Routing and tag expressions</a> .
<b>hubName</b>	<b>HubName</b>	Name of the notification hub resource in the Azure portal.
<b>connection</b>	<b>ConnectionStringSetting</b>	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 <a href="#">Connection string setup</a> 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 <a href="#">Templates</a>. When set, <b>platform</b> must be one of the following values:</p> <ul style="list-style-type: none"> <li>• <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 <a href="#">Sending push notifications to iOS with Azure Notification Hubs</a>.</li> <li>• <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 <a href="#">Getting Started with Notification Hubs for Kindle apps</a>.</li> <li>• <code>gcm</code>—Google Cloud Messaging. Firebase Cloud Messaging, which is the new version of GCM, is also supported. For more information, see <a href="#">Sending push notifications to Android with Azure Notification Hubs</a>.</li> <li>• <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 <a href="#">Getting started with Notification Hubs for Windows Universal Platform Apps</a>.</li> <li>• <code>mpns</code>—Microsoft Push Notification Service. This platform supports Windows Phone 8 and earlier Windows Phone platforms. For more information, see <a href="#">Sending push notifications with Azure Notification Hubs on Windows Phone</a>.</li> </ul>

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.

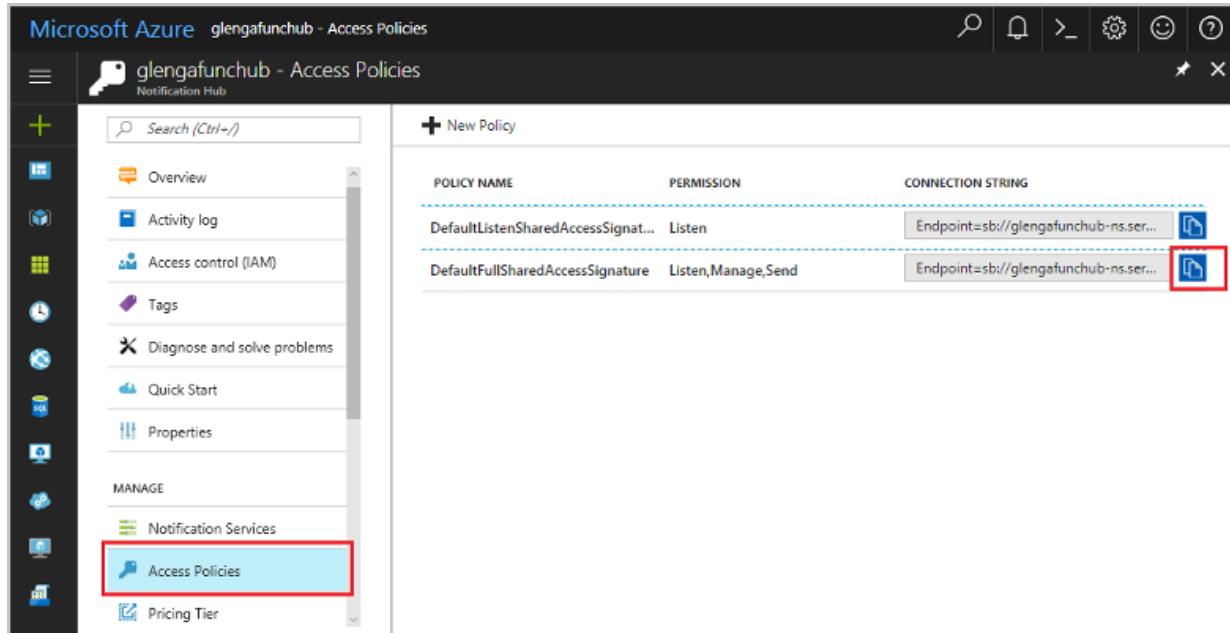
```
{
 "bindings": [
 {
 "type": "notificationHub",
 "direction": "out",
 "name": "notification",
 "tagExpression": "",
 "hubName": "my-notification-hub",
 "connection": "MyHubConnectionString",
 "platform": "gcm"
 }
],
 "disabled": false
}
```

## 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.



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.

## Next steps

[Learn more about Azure functions triggers and bindings](#)



# Azure Queue storage bindings for Azure Functions

1/3/2018 • 10 min 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.](#)
- [Azure Functions developer reference.](#)
- [C#, F#, Node, or Java developer reference.](#)
- [Azure Functions triggers and bindings concepts.](#)

## 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](#)

### 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,
 TraceWriter log)
 {
 log.Info($"C# function processed: {myQueueItem}");
 }
}
```

### Trigger - C# script example

The following example shows a blob 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.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,
 TraceWriter log)
{
 log.Info($"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 blob 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.

Here's the JavaScript code:

```
module.exports = function (context) {
 context.log('Node.js queue trigger function processed work item', context.bindings.myQueueItem);
 context.log('queueTrigger =', context.bindingData.queueTrigger);
 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 - attributes

In [C# class libraries](#), use the following attributes to configure a queue trigger:

- [QueueTriggerAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs](#)

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,
 TraceWriter 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,
 TraceWriter log)
{
 ...
}
```

For a complete example, see [Trigger - C# example](#).

- [StorageAccountAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs](#)

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
<b>type</b>	n/a	Must be set to <code>queueTrigger</code> . This property is set automatically when you create the trigger in the Azure portal.
<b>direction</b>	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.
<b>name</b>	n/a	The name of the variable that represents the queue in function code.
<b>queueName</b>	<b>QueueName</b>	The name of the queue to poll.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>connection</b>	<b>Connection</b>	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 blob data by using a method parameter such as `Stream 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:

- POCO object - The Functions runtime deserializes a JSON payload into a POCO object.
- `string`
- `byte[]`
- [CloudQueueMessage](#)

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. The values have the same semantics as [CloudQueueMessage](#).

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.
<code>Id</code>	<code>string</code>	Queue message ID.

PROPERTY	TYPE	DESCRIPTION
InsertionTime	DateTimeOffset?	The time that the message was added to the queue.
NextVisibleTime	DateTimeOffset?	The time that the message will next be visible.
PopReceipt	string	The message's pop receipt.

## 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 `<originalqueuename>-poison`. You can write a function to process messages from the poison queue by logging them or sending a notification that manual attention is needed.

To handle poison messages manually, check the [dequeueCount](#) of the queue message.

## Trigger - host.json properties

The [host.json](#) file contains settings that control queue trigger behavior.

```
{
 "queues": {
 "maxPollingInterval": 2000,
 "visibilityTimeout" : "00:00:30",
 "batchSize": 16,
 "maxDequeueCount": 5,
 "newBatchThreshold": 8
 }
}
```

PROPERTY	DEFAULT	DESCRIPTION
maxPollingInterval	60000	The maximum interval in milliseconds between queue polls.
visibilityTimeout	0	The time interval between retries when processing of a message fails.
batchSize	16	The number of queue messages to retrieve and process in parallel. The maximum is 32.
maxDequeueCount	5	The number of times to try processing a message before moving it to the poison queue.
newBatchThreshold	batchSize/2	The threshold at which a new batch of messages are fetched.

## 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](#)

## 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, TraceWriter log)
 {
 log.Info($"C# function processed: {input.Text}");
 return input.Text;
 }
}
```

## Output - C# script example

The following example shows a blob trigger binding in a *function.json* file and [C# script \(.csx\)](#) code that uses the binding. The function creates a queue item with a POCO 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, TraceWriter 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> myQueueItem,
 TraceWriter log)
{
 myQueueItem.Add(input);
 myQueueItem.Add(new CustomQueueMessage { PersonName = "You", Title = "None" });
}

```

## Output - JavaScript example

The following example shows a blob 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 - attributes

In [C# class libraries](#), use the `QueueAttribute`, which is defined in NuGet package [Microsoft.Azure.WebJobs](#).

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, TraceWriter 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, TraceWriter 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
<code>type</code>	n/a	Must be set to <code>queue</code> . This property is set automatically when you create the trigger in the Azure portal.
<code>direction</code>	n/a	Must be set to <code>out</code> . This property is set automatically when you create the trigger in the Azure portal.
<code>name</code>	n/a	The name of the variable that represents the queue in function code. Set to <code>\$return</code> to reference the function return value.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>queueName</b>	<b>QueueName</b>	The name of the queue.
<b>connection</b>	<b>Connection</b>	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:

- A POCO serializable as JSON
- `string`
- `byte[]`
- [CloudQueueMessage](#)

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.

## Next steps

[Go to a quickstart that uses a Queue storage trigger](#)

[Go to a tutorial that uses a Queue storage output binding](#)

[Learn more about Azure functions triggers and bindings](#)

# Azure Functions SendGrid bindings

1/2/2018 • 2 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

## Example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [JavaScript](#)

### C# example

The following example shows a [C# function](#) that uses a Service Bus queue trigger and a SendGrid output binding.

```
[FunctionName("SendEmail")]
public static void Run(
 [ServiceBusTrigger("myqueue", AccessRights.Manage, Connection = "ServiceBusConnection")] OutgoingEmail
 email,
 [SendGrid(ApiKey = "CustomSendGridKeyAppSettingName")] out SendGridMessage message)
{
 message = new SendGridMessage();
 message.AddTo(email.To);
 message.AddContent("text/html", email.Body);
 message.SetFrom(new EmailAddress(email.From));
 message.SetSubject(email.Subject);
}

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": [
 {
 "name": "message",
 "type": "sendGrid",
 "direction": "out",
 "apiKey" : "MySendGridKey",
 "to": "{ToEmail}",
 "from": "{FromEmail}",
 "subject": "SendGrid output bindings"
 }
]
}
```

The [configuration](#) section explains these properties.

Here's the C# script code:

```
#r "SendGrid"
using System;
using SendGrid.Helpers.Mail;

public static void Run(TraceWriter log, string input, out Mail message)
{
 message = new Mail
 {
 Subject = "Azure news"
 };

 var personalization = new Personalization();
 personalization.AddTo(new Email("recipient@contoso.com"));

 Content content = new Content
 {
 Type = "text/plain",
 Value = input
 };
 message.AddContent(content);
 message.AddPersonalization(personalization);
}
```

## 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, which is defined in NuGet package `Microsoft.Azure.WebJobs.Extensions.SendGrid`.

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", AccessRights.Manage, 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
<b>type</b>		Required - must be set to <code>sendGrid</code> .
<b>direction</b>		Required - must be set to <code>out</code> .
<b>name</b>		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.
<b>apiKey</b>	<b>ApiKey</b>	The name of an app setting that contains your API key. If not set, the default app setting name is "AzureWebJobsSendGridApiKey".
<b>to</b>	<b>To</b>	the recipient's email address.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>from</b>	<b>From</b>	the sender's email address.
<b>subject</b>	<b>Subject</b>	the subject of the email.
<b>text</b>	<b>Text</b>	the email content.

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 Service Bus bindings for Azure Functions

1/2/2018 • 12 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

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

#### Trigger - C# example

The following example shows a [C# function](#) that logs a Service Bus queue message.

```
[FunctionName("ServiceBusQueueTriggerCSharp")]
public static void Run(
 [ServiceBusTrigger("myqueue", AccessRights.Manage, Connection = "ServiceBusConnection")]
 string myQueueItem,
 TraceWriter log)
{
 log.Info($"C# ServiceBus queue trigger function processed message: {myQueueItem}");
}
```

#### 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 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:

```
public static void Run(string myQueueItem, TraceWriter log)
{
 log.Info($"C# ServiceBus queue trigger function processed message: {myQueueItem}");
}
```

### 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: TraceWriter) =
 log.Info(sprintf "F# ServiceBus queue trigger function processed message: %s" myQueueItem)
```

### 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 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.done();
};
```

## Trigger - attributes

In [C# class libraries](#), use the following attributes to configure a Service Bus trigger:

- [ServiceBusTriggerAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs.ServiceBus](#)

The attribute's constructor takes the name of the queue or the topic and subscription. You can also specify the connection's access rights. If you don't specify access rights, the default is [Manage](#). How to choose the access rights setting is explained in 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, TraceWriter 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, TraceWriter log)
{
 ...
}
```

For a complete example, see [Trigger - C# example](#).

- [ServiceBusAccountAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs.ServiceBus](#)

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, TraceWriter 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
<b>type</b>	n/a	Must be set to "serviceBusTrigger". This property is set automatically when you create the trigger in the Azure portal.
<b>direction</b>	n/a	Must be set to "in". This property is set automatically when you create the trigger in the Azure portal.
<b>name</b>	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.
<b>queueName</b>	<b>QueueName</b>	Name of the queue to monitor. Set only if monitoring a queue, not for a topic.
<b>topicName</b>	<b>TopicName</b>	Name of the topic to monitor. Set only if monitoring a topic, not for a queue.
<b>subscriptionName</b>	<b>SubscriptionName</b>	Name of the subscription to monitor. Set only if monitoring a topic, not for a queue.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>connection</b>	<b>Connection</b>	<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 <a href="#">Obtain the management credentials</a>. The connection string must be for a Service Bus namespace, not limited to a specific queue or topic.</p>
<b>accessRights</b>	<b>Access</b>	<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 <b>Manage</b> permission. If you use a connection string that does not have the <b>Manage</b> permission, set <code>accessRights</code> to "listen". Otherwise, the Functions runtime might fail trying to do operations that require manage rights.</p>

When you're developing locally, app settings go into the [local.settings.json file](#).

## Trigger - usage

In C# and C# script, access the queue or topic message 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 use any of the following types instead of `string`:

- `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\(\)](#) method.

In JavaScript, access the queue or topic message by using `context.bindings.<name>`. `<name>` is the value specified in the `name` property of `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.

## Trigger - host.json properties

The [host.json](#) file contains settings that control Service Bus trigger behavior.

```
{
 "serviceBus": {
 "maxConcurrentCalls": 16,
 "prefetchCount": 100,
 "autoRenewTimeout": "00:05:00"
 }
}
```

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.

## 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#](#)
- [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, TraceWriter log)
{
 log.Info($"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, TraceWriter log, out string outputSbQueue)
{
 string message = $"Service Bus queue message created at: {DateTime.Now}";
 log.Info(message);
 outputSbQueue = message;
}
```

Here's C# script code that creates multiple messages:

```
public static void Run(TimerInfo myTimer, TraceWriter log, ICollector<string> outputSbQueue)
{
 string message = $"Service Bus queue messages created at: {DateTime.Now}";
 log.Info(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: TraceWriter, outputSbQueue: byref<string>) =
 let message = sprintf "Service Bus queue message created at: %s" (DateTime.Now.ToString())
 log.Info(message)
 outputSbQueue = message
```

## 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.outputSbQueueMsg = 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.outputSbQueueMsg = [];
 context.bindings.outputSbQueueMsg.push("1 " + message);
 context.bindings.outputSbQueueMsg.push("2 " + message);
 context.done();
};


```

## Output - attributes

In [C# class libraries](#), use the [ServiceBusAttribute](#), which is defined in NuGet package [Microsoft.Azure.WebJobs.ServiceBus](#).

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, TraceWriter 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, TraceWriter 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
<b>type</b>	n/a	Must be set to "serviceBus". This property is set automatically when you create the trigger in the Azure portal.
<b>direction</b>	n/a	Must be set to "out". This property is set automatically when you create the trigger in the Azure portal.
<b>name</b>	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.
<b>queueName</b>	<b>QueueName</b>	Name of the queue. Set only if sending queue messages, not for a topic.
<b>topicName</b>	<b>TopicName</b>	Name of the topic to monitor. Set only if sending topic messages, not for a queue.
<b>subscriptionName</b>	<b>SubscriptionName</b>	Name of the subscription to monitor. Set only if sending topic messages, not for a queue.
<b>connection</b>	<b>Connection</b>	<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 <a href="#">Obtain the management credentials</a>. The connection string must be for a Service Bus namespace, not limited to a specific queue or topic.</p>
<b>accessRights</b>	<b>Access</b>	<p>Access rights for the connection string. Available values are "manage" and "listen". The default is "manage", which indicates that the connection has <b>Manage</b> permissions. If you use a connection string that does not have <b>Manage</b> permissions, set <code>accessRights</code> to "listen". Otherwise, the Functions runtime might fail trying to do operations that require manage rights.</p>

When you're developing locally, app settings go into the [local.settings.json file](#).

## Output - usage

In C# and C# script, access the queue or topic 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`. You can use any of the following parameter types:

- `out T paramName` - `T` can be any JSON-serializable type. If the parameter value is null when the function exits, Functions creates the message with a null object.
- `out string` - If the parameter value is null when the function exits, Functions does not create a message.
- `out byte[]` - If the parameter value is null when the function exits, Functions does not create a message.
- `out BrokeredMessage` - If the parameter value is null when the function exits, Functions does not create a message.

For creating multiple messages in a C# or C# script function, you can use `ICollector<T>` or `IAsyncCollector<T>`. A message is created when you call the `Add` method.

In JavaScript, access the queue or topic by using `context.bindings.<name>`. `<name>` is the value specified in the `<name>` property of `function.json`. You can assign a string, a byte array, or a Javascript object (deserialized into JSON) to `context.binding.<name>`.

## Next steps

[Learn more about Azure functions triggers and bindings](#)

# Azure Table storage bindings for Azure Functions

1/2/2018 • 13 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

## 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# read multiple entities](#)
- [C# script - read one entity](#)
- [C# script - read multiple entities](#)
- [F#](#)
- [JavaScript](#)

### Input - C# example 1

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,
 TraceWriter log)
 {
 log.Info($"PK={poco.PartitionKey}, RK={poco.RowKey}, Text={poco.Text}");
 }
}
```

## Input - C# example 2

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,
 TraceWriter log)
 {
 foreach (MyPoco poco in pocos)
 {
 log.Info($"PK={poco.PartitionKey}, RK={poco.RowKey}, Text={poco.Text}");
 }
 }
}
```

## Input - C# script example 1

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, TraceWriter log)
{
 log.Info($"C# Queue trigger function processed: {myQueueItem}");
 log.Info($"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 2

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;

public static void Run(string myQueueItem, IQueryable<Person> tableBinding, TraceWriter log)
{
 log.Info($"C# Queue trigger function processed: {myQueueItem}");
 foreach (Person person in tableBinding.Where(p => p.PartitionKey == myQueueItem).ToList())
 {
 log.Info($"Name: {person.Name}");
 }
}

public class Person : TableEntity
{
 public string Name { get; set; }
}

```

## 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

```

## 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 - attributes

In [C# class libraries](#), use the following attributes to configure a table input binding:

- [TableAttribute](#), which is defined in NuGet package [Microsoft.Azure.WebJobs](#).

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,
 TraceWriter 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,
 TraceWriter log)
{
 ...
}
```

For a complete example, see [Input - C# example](#).

- [StorageAccountAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs](#)

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 - 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
<b>type</b>	n/a	Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal.
<b>direction</b>	n/a	Must be set to <code>in</code> . This property is set automatically when you create the binding in the Azure portal.
<b>name</b>	n/a	The name of the variable that represents the table or entity in function code.
<b>tableName</b>	<b>TableName</b>	The name of the table.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>partitionKey</b>	<b>PartitionKey</b>	Optional. The partition key of the table entity to read. See the <a href="#">usage</a> section for guidance on how to use this property.
<b>rowKey</b>	<b>RowKey</b>	Optional. The row key of the table entity to read. See the <a href="#">usage</a> section for guidance on how to use this property.
<b>take</b>	<b>Take</b>	Optional. The maximum number of entities to read in JavaScript. See the <a href="#">usage</a> section for guidance on how to use this property.
<b>filter</b>	<b>Filter</b>	Optional. An OData filter expression for table input in JavaScript. See the <a href="#">usage</a> section for guidance on how to use this property.
<b>connection</b>	<b>Connection</b>	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` does not work in .NET Core, so it doesn't work 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.

- **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.

## 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, TraceWriter log)
 {
 log.Info($"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, TraceWriter log)
{
 for (int i = 1; i < 10; i++)
 {
 log.Info($"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: TraceWriter) =
 for i = 1 to 10 do
 log.Info(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](#), which is defined in NuGet package [Microsoft.Azure.WebJobs](#).

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,
 TraceWriter 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,
 TraceWriter 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
------------------------	--------------------	-------------

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>type</b>	n/a	Must be set to <code>table</code> . This property is set automatically when you create the binding in the Azure portal.
<b>direction</b>	n/a	Must be set to <code>out</code> . This property is set automatically when you create the binding in the Azure portal.
<b>name</b>	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.
<b>tableName</b>	<b>TableName</b>	The name of the table.
<b>partitionKey</b>	<b>PartitionKey</b>	The partition key of the table entity to write. See the <a href="#">usage section</a> for guidance on how to use this property.
<b>rowKey</b>	<b>RowKey</b>	The row key of the table entity to write. See the <a href="#">usage section</a> for guidance on how to use this property.
<b>connection</b>	<b>Connection</b>	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#**

In C# and C# script, access the output table entity by using a method parameter `ICollector<T> paramName`

or `ICollectorAsync<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 paramName` method parameter to write to the table by using the Azure Storage SDK.

- **Write one or more rows in JavaScript**

In JavaScript functions, access the table output using `context.bindings.<name>`.

## Next steps

[Learn more about Azure functions triggers and bindings](#)

# Timer trigger for Azure Functions

1/2/2018 • 4 min 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.](#)
- [Azure Functions developer reference.](#)
- [C#, F#, Node, or Java developer reference.](#)
- [Azure Functions triggers and bindings concepts.](#)

## Example

See the language-specific example:

- [C#](#)
- [C# script \(.csx\)](#)
- [F#](#)
- [JavaScript](#)

### C# example

The following example shows a [C# function](#) that runs every five minutes:

```
[FunctionName("TimerTriggerCSharp")]
public static void Run([TimerTrigger("0 */5 * * * *")]TimerInfo myTimer, TraceWriter log)
{
 log.Info($"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.

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, TraceWriter log)
{
 if(myTimer.IsPastDue)
 {
 log.Info("Timer is running late!");
 }
 log.Info($"C# Timer trigger function executed at: {DateTime.Now} ");
}
```

## F# example

The following example shows a timer trigger binding in a *function.json* file and a [F# script function](#) that uses the binding. The function writes a log indicating whether this function invocation is due to a missed schedule occurrence.

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: TraceWriter) =
 if (myTimer.IsPastDue) then
 log.Info("F# function is running late.")
 let now = DateTime.Now.ToString()
 log.Info(sprintf "F# function executed at %s!" now)
```

## 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.

Here's the binding data in the *function.json* file:

```
{
 "schedule": "0 */5 * * * *",
 "name": "myTimer",
 "type": "timerTrigger",
 "direction": "in"
}
```

Here's the JavaScript script code:

```

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.done();
};

```

## Attributes

In [C# class libraries](#), use the [TimerTriggerAttribute](#), defined in NuGet package [Microsoft.Azure.WebJobs.Extensions](#).

The attribute's constructor takes a CRON expression, as shown in the following example:

```

[FunctionName("TimerTriggerCSharp")]
public static void Run([TimerTrigger("0 */5 * * * *")]TimerInfo myTimer, TraceWriter log)
{
 ...
}

```

You can specify a `TimeSpan` instead of a CRON expression if your function app runs on an App Service plan (not a Consumption plan).

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 `TimerTrigger` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>type</b>	n/a	Must be set to "timerTrigger". This property is set automatically when you create the trigger in the Azure portal.
<b>direction</b>	n/a	Must be set to "in". This property is set automatically when you create the trigger in the Azure portal.
<b>name</b>	n/a	The name of the variable that represents the timer object in function code.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>schedule</b>	<b>ScheduleExpression</b>	<p>On the Consumption plan, you can define schedules with a CRON expression. If you're using an App Service Plan, you can also use a <code>TimeSpan</code> string. The following sections explain CRON expressions. You can put the schedule expression in an app setting and set this property to a value wrapped in % signs, as in this example:</p> <p><code>"%NameOfAppSettingWithCRONExpression%"</code>.</p>

When you're developing locally, app settings go into the [local.settings.json file](#).

### CRON format

A [CRON expression](#) for the Azure Functions timer trigger includes these six fields:

```
{second} {minute} {hour} {day} {month} {day-of-week}
```

#### NOTE

Many of the CRON expressions you find online omit the `{second}` field. If you copy from one of them, add the missing `{second}` field.

### CRON time zones

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 a new 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 * * *",
```

Alternatively, you could add a new app setting for your function app named `WEBSITE_TIME_ZONE` and set the value to **Eastern Standard Time**. Then the following CRON expression could be used for 10:00 AM EST:

```
"schedule": "0 0 10 * * *",
```

### CRON examples

Here are some examples of CRON expressions you can use for the timer trigger in Azure Functions.

To trigger once every five minutes:

```
"schedule": "0 */5 * * *"
```

To trigger once at the top of every hour:

```
"schedule": "0 0 * * * *",
```

To trigger once every two hours:

```
"schedule": "0 0 */2 * * *",
```

To trigger once every hour from 9 AM to 5 PM:

```
"schedule": "0 0 9-17 * * *",
```

To trigger At 9:30 AM every day:

```
"schedule": "0 30 9 * * *",
```

To trigger At 9:30 AM every weekday:

```
"schedule": "0 30 9 * * 1-5",
```

## Usage

When a timer trigger function is invoked, the [timer object](#) is passed into the function. The following JSON is an example representation of the timer object.

```
{
 "Schedule": {
 },
 "ScheduleStatus": {
 "Last": "2016-10-04T10:15:00.012699+00:00",
 "Next": "2016-10-04T10:20:00+00:00"
 },
 "IsPastDue": false
}
```

## Scale-out

The timer trigger supports multi-instance scale-out. A single instance of a particular timer function is run across all instances.

## Next steps

[Go to a quickstart that uses a timer trigger](#)

[Learn more about Azure functions triggers and bindings](#)

# Twilio binding for Azure Functions

1/2/2018 • 4 min 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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

## Example

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 =
"+1425XXXXXX")]
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": "+1704XXXXXX",
 "from": "+1425XXXXXX",
 "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 asynchronous code. Here's an asynchronous C# script code example:

```

#r "Newtonsoft.Json"
#r "Twilio.Api"

using System;
using Newtonsoft.Json;
using Twilio;

public static async Task Run(string myQueueItem, IAsyncCollector<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.
 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();
};

```

## Attributes

In [C# class libraries](#), use the `TwilioSms` attribute, which is defined in NuGet package `Microsoft.Azure.WebJobs.Extensions.Twilio`.

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 SMSMessage Run(
 [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] JObject order,
 TraceWriter 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.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>type</b>		must be set to <code>twilioSms</code> .
<b>direction</b>		must be set to <code>out</code> .
<b>name</b>		Variable name used in function code for the Twilio SMS text message.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>accountSid</b>	<b>AccountSid</b>	This value must be set to the name of an app setting that holds your Twilio Account Sid.
<b>authToken</b>	<b>AuthToken</b>	This value must be set to the name of an app setting that holds your Twilio authentication token.
<b>to</b>	<b>To</b>	This value is set to the phone number that the SMS text is sent to.
<b>from</b>	<b>From</b>	This value is set to the phone number that the SMS text is sent from.
<b>body</b>	<b>Body</b>	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

11/29/2017 • 6 min 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. The JSON schema is at <http://json.schemastore.org/host>.

There are other global configuration options in [app settings](#) and in the [local.settings.json](#) file.

## Sample host.json file

The following sample `host.json` file has 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",
 "http": {
 "routePrefix": "api",
 "maxOutstandingRequests": 20,
 "maxConcurrentRequests": 10,
 "dynamicThrottlesEnabled": false
 },
 "id": "9f4ea53c5136457d883d685e57164f08",
 "logger": {
 "categoryFilter": {
 "defaultLevel": "Information",
 "categoryLevels": {
 "Host": "Error",
 "Function": "Error",
 "Host.Aggregator": "Information"
 }
 }
 },
 "queues": {
 "maxPollingInterval": 2000,
 "visibilityTimeout" : "00:00:30",
 "batchSize": 16,
 "maxDequeueCount": 5,
 "newBatchThreshold": 8
 },
 "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	false	Enables or disables sampling.
maxTelemetryItemsPerSecond	5	The threshold at which sampling begins.

## 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.

PROPERTY	DEFAULT	DESCRIPTION
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 will run. An empty array means run all functions. Intended for use only when [running locally](#). In function apps, use the `function.json` `disabled` property rather than this property in `host.json`.

```
{
 "functions": ["QueueProcessor", "GitHubWebHook"]
}
```

## functionTimeout

Indicates the timeout duration for all functions. In Consumption plans, the valid range is from 1 second to 10 minutes, and the default value is 5 minutes. In App Service plans, there is no limit and the default value is null, which indicates no timeout.

```
{
 "functionTimeout": "00:05:00"
}
```

## http

Configuration settings for [http triggers and bindings](#).

```
{
 "http": {
 "routePrefix": "api",
 "maxOutstandingRequests": 20,
 "maxConcurrentRequests": 10,
 "dynamicThrottlesEnabled": false
 }
}
```

PROPERTY	DEFAULT	DESCRIPTION
routePrefix	api	The route prefix that applies to all routes. Use an empty string to remove the default prefix.

PROPERTY	DEFAULT	DESCRIPTION
maxOutstandingRequests	-1	The maximum number of outstanding requests that are held at any given time. This limit includes requests that are queued but have not started executing, as well as any in progress executions. Any incoming requests over this limit are rejected with a 429 "Too Busy" response. That allows callers to employ time-based retry strategies, and also helps you to control maximum request latencies. This only controls queuing that occurs within the script host execution path. Other queues such as the ASP.NET request queue will still be in effect and unaffected by this setting. The default is unbounded.
maxConcurrentRequests	-1	The maximum number of http functions that will be executed in parallel. This allows you to control concurrency, which can help manage resource utilization. For example, you might have an http function that uses a lot of system resources (memory/cpu/sockets) such that it causes issues when concurrency is too high. Or you might have a function that makes outbound requests to a third party service, and those calls need to be rate limited. In these cases, applying a throttle here can help. The default is unbounded.
dynamicThrottlesEnabled	false	When enabled, this setting causes the request processing pipeline to periodically check system performance counters like connections/threads/processes/memory/cpu/etc. and if any of those counters are over a built-in high threshold (80%), requests will be rejected with a 429 "Too Busy" response until the counter(s) return to normal levels.

## id

The unique ID for a job host. Can be a lower case GUID with dashes removed. Required when running locally. When running in Azure Functions, an ID is generated automatically if `id` is omitted.

```
{
 "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	The number of queue messages to retrieve and process in parallel. The maximum is 32.
maxDequeueCount	5	The number of times to try processing a message before moving it to the poison queue.
newBatchThreshold	batchSize/2	The threshold at which a new batch of messages are fetched.

## serviceBus

Configuration setting for [Service Bus triggers and bindings](#).

```
{
 "serviceBus": {
 "maxConcurrentCalls": 16,
 "prefetchCount": 100,
 "autoRenewTimeout": "00:05:00"
 }
}
```

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

Configuration settings for logs that you create by using a `TraceWriter` object. See [C# Logging](#) and [Nodejs 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"]
}
```

## durableTask

Task hub name for [Durable Functions](#).

```
{
 "durableTask": {
 "HubName": "MyTaskHub"
 }
}
```

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](#).

## Next steps

[Learn how to update the host.json file](#)

[See global settings in environment variables](#)

# App settings reference for Azure Functions

9/28/2017 • 2 min 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 in environment variables. This article lists the app settings that are available in function apps.

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

## 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](#).

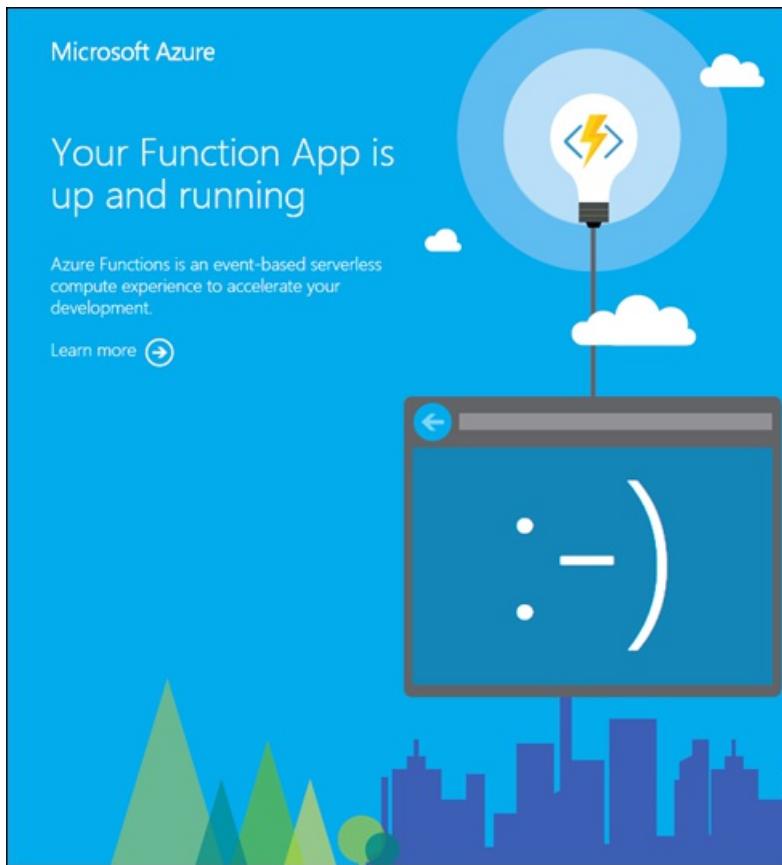
KEY	SAMPLE VALUE
AzureWebJobsDashboard	DefaultEndpointsProtocol=https;AccountName=[name];AccountKey=[key]

## 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
<code>AzureWebJobsDotNetReleaseCompilation</code>	<code>true</code>

## AzureWebJobsFeatureFlags

A comma-delimited list of beta features to enable. Beta features enabled by these flags are not production ready, but can be enabled for experimental use before they go live.

KEY	SAMPLE VALUE
<code>AzureWebJobsFeatureFlags</code>	<code>feature1,feature2</code>

## AzureWebJobsScriptRoot

The path to the root directory where the `host.json` file and function folders are located. In a function app, the default is `%HOME%\site\wwwroot`.

KEY	SAMPLE VALUE
<code>AzureWebJobsScriptRoot</code>	<code>%HOME%\site\wwwroot</code>

## AzureWebJobsSecretStorageType

Specifies the repository or provider to use for key storage. Currently, the supported repositories are blob ("Blob")

and file system ("disabled"). The default is file system ("disabled").

KEY	SAMPLE VALUE
AzureWebJobsSecretStorageType	disabled

## 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

Valid values are "readwrite" and "readonly".

KEY	SAMPLE VALUE
FUNCTION_APP_EDIT_MODE	readonly

## FUNCTIONS\_EXTENSION\_VERSION

The version of the Azure Functions runtime to use in this function app. A tilde with major version means use the latest version of that major version (for example, "~1"). 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, "1.0.12345"). Default is "~1".

KEY	SAMPLE VALUE
FUNCTIONS_EXTENSION_VERSION	~1

## WEBSITE\_CONTENTAZUREFILECONNECTIONSTRING

For consumption 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 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 for a preview feature.

KEY	SAMPLE VALUE
WEBSITE_MAX_DYNAMIC_APPLICATION_SCALE_OUT	10

## WEBSITE\_NODE\_DEFAULT\_VERSION

Default is "6.5.0".

KEY	SAMPLE VALUE
WEBSITE_NODE_DEFAULT_VERSION	6.5.0

## Next steps

[Learn how to update app settings](#)

[See global settings in the host.json file](#)

# OpenAPI 2.0 metadata support in Azure Functions (preview)

9/19/2017 • 2 min 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.

[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](#).
- [Azure Functions developer reference](#).
- [C#, F#, Node, or Java developer reference](#).
- [Azure Functions triggers and bindings concepts](#).

## 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**.

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	<b>Function app settings &gt; App Service settings &gt; Overview &gt; URL</b>	<i>Not present</i>
Paths	<b>Integrate &gt; Selected HTTP methods</b>	Bindings: Route
Path Item	<b>Integrate &gt; Route template</b>	Bindings: Methods
Security	<b>Keys</b>	<i>Not present</i>
operationID*	<b>Route + Allowed verbs</b>	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.