Table of Contents

Stack Overflow

Azure Functions GitHub repository

Service updates

Azure Functions Overview

1/18/2017 • 4 min to read • Edit on GitHub

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, Python or PHP. Pay only for the time your code runs and trust Azure to scale as needed.

This topic provides a high-level overview of Azure Functions. If you want to jump right in and get started with Azure 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 Azure Functions:

- **Choice of language** - Write functions using C#, F#, Node.js, Python, PHP, batch, bash, or any executable.
- **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?

Azure 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, long-running tasks that you want to run in a background thread, 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:

- **BlobTrigger** - Process Azure Storage blobs when they are added to containers. You might use this function for image resizing.
- **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.
- **Generic webhook** - Process webhook HTTP requests from any service that supports webhooks.
- **GitHub webhook** - Respond to events that occur in your GitHub repositories. For an example, see Create a webhook or API function.
- **HTTPTrigger** - Trigger the execution of your code by using an HTTP request.
- **QueueTrigger** - Respond to messages as they arrive in an Azure Storage queue. For an example, see Create an Azure Function that binds to an Azure service.
- **ServiceBusQueueTrigger** - Connect your code to other Azure services or on-premise services by listening to

message queues.

- **ServiceBusTopicTrigger** - Connect your code to other Azure services or on-premise services by subscribing to topics.
- **TimerTrigger** - Execute cleanup or other batch tasks on a predefined schedule. For an example, see Create an event processing function.

Azure Functions supports *triggers*, which are ways to start execution of your code, and *bindings*, which are ways to simplifying 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 DocumentDB
- Azure Event Hubs
- 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)

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.

Full pricing details are available on the Functions Pricing page. For more information about scaling your functions, see How to scale Azure Functions.

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 the Azure App Service platform for core functionality like deployments, environment variables, and diagnostics.

Create your first Azure Function

2/6/2017 • 3 min to read • Edit on GitHub

Overview

Azure Functions is an event-driven, compute-on-demand experience that extends the existing Azure application platform with capabilities to implement code triggered by events occurring in other Azure services, SaaS products, and on-premises systems. With Azure Functions, your applications scale based on demand and you pay only for the resources you consume. Azure Functions enables you to create scheduled or triggered units of code implemented in various programming languages. To learn more about Azure Functions, see the Azure Functions Overview.

This topic shows you how to use the Azure Functions quickstart in the portal to create a simple "hello world" JavaScript function that is invoked by an HTTP-trigger. You can also watch a short video to see how these steps are performed in the portal.

Watch the video

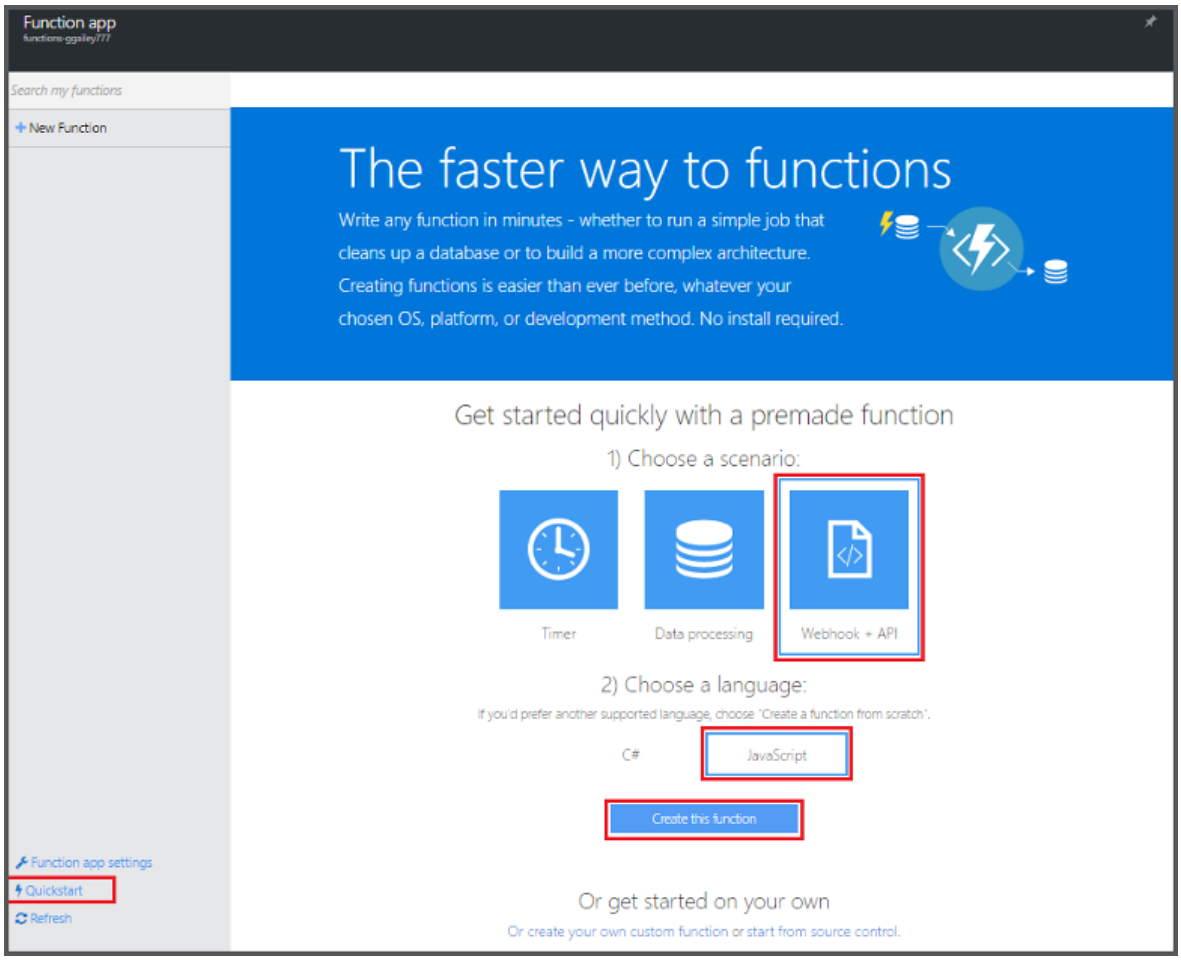
The following video shows how to perform the basic steps in this tutorial.

Create a function from the quickstart

A function app hosts the execution of your functions in Azure. Follow these steps to create a function app with the new function. The function app is created with a default configuration. For an example of how to explicitly create your function app, see the other Azure Functions quickstart tutorial.

Before you can create your first function, you need to have an active Azure account. If you don't already have an Azure account, free accounts are available.

1. Go to the Azure Functions portal and sign-in with your Azure account.
2. Type a unique **Name** for your new function app or accept the generated one, select your preferred **Region**, then click **Create + get started**. Note that you must enter a valid name, which can contain only letters, numbers, and hyphens. Underscore (_) is not an allowed character.
3. In the **Quickstart** tab, click **WebHook + API** and **JavaScript**, then click **Create a function**. A new predefined JavaScript function is created.



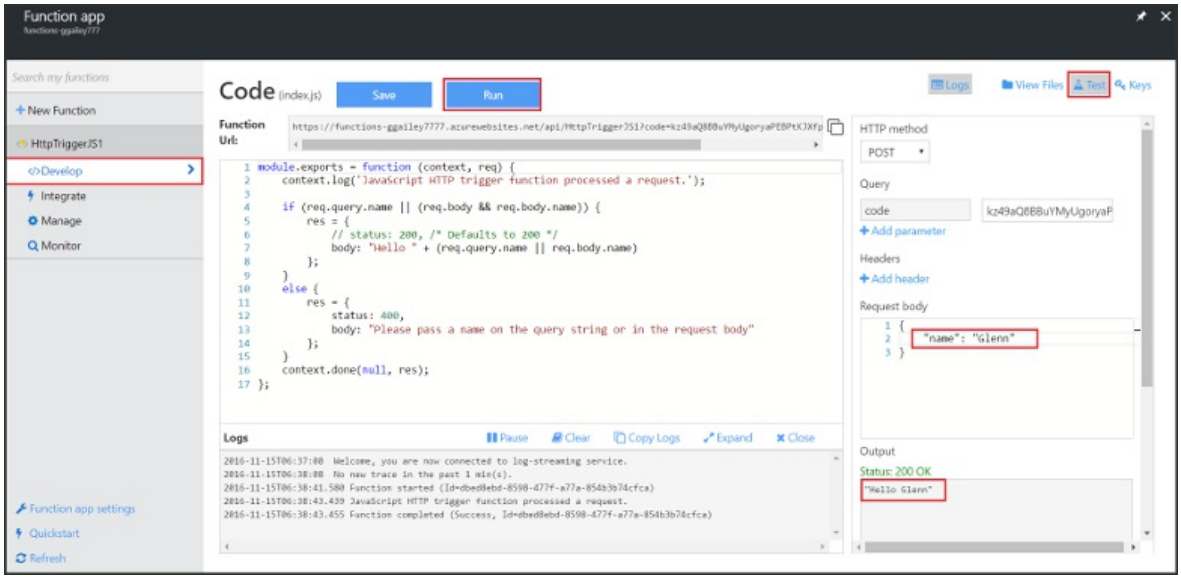
4. (Optional) At this point in the quickstart, you can choose to take a quick tour of Azure Functions features in the portal. After you have completed or skipped the tour, you can test your new function by using the HTTP trigger.

Test the function

Since the Azure Functions quickstarts contain functional code, you can immediately test your new function.

1. In the **Develop** tab, review the **Code** window and notice that this Node.js code expects an HTTP request with a *name* value passed either in the message body or in a query string. When the function runs, this value is returned in the response message.

2. Click **Test** to display the built-in HTTP test request pane for the function.



3. In the **Request body** text box, change the value of the *name* property to your name, and click **Run**. You see that execution is triggered by a test HTTP request, information is written to the logs, and the "hello" response is displayed in the **Output**.

4. To trigger execution of the same function from an HTTP testing tool or from another browser window, copy the **Function URL** value from the **Develop** tab and paste it into the tool or browser address bar. Append the query string value `&name=yourname` to the URL and execute the request. Note that the same information is written to the logs and the same string is contained in the body of the response message.

Next steps

This quickstart demonstrates a simple execution of a basic HTTP-triggered function. To learn more about using Azure Functions in your apps, see the following topics:

- Best Practices for Azure Functions
- 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.
- How to scale Azure Functions
  Discusses service plans available with Azure Functions, including the Consumption hosting plan, and how to choose the right plan.
- What is Azure App Service?
  Azure Functions uses the Azure App Service platform for core functionality like deployments, environment variables, and diagnostics.

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions`. - Visit Stack Overflow

Create a webhook or API Azure Function

2/6/2017 • 3 min to read • Edit on GitHub

Azure Functions is an event-driven, compute-on-demand experience that enables you to create scheduled or triggered units of code implemented various programming languages. To learn more about Azure Functions, see the Azure Functions Overview.

This topic shows you how to create a JavaScript function that is invoked by a GitHub webhook. The new function is created based on a pre-defined template in the Azure Functions portal. You can also watch a short video to see how these steps are performed in the portal.

The general steps in this tutorial can also be used to create a function in C# or F# instead of JavaScript.

Watch the video

The following video shows how to perform the basic steps in this tutorial

Prerequisites

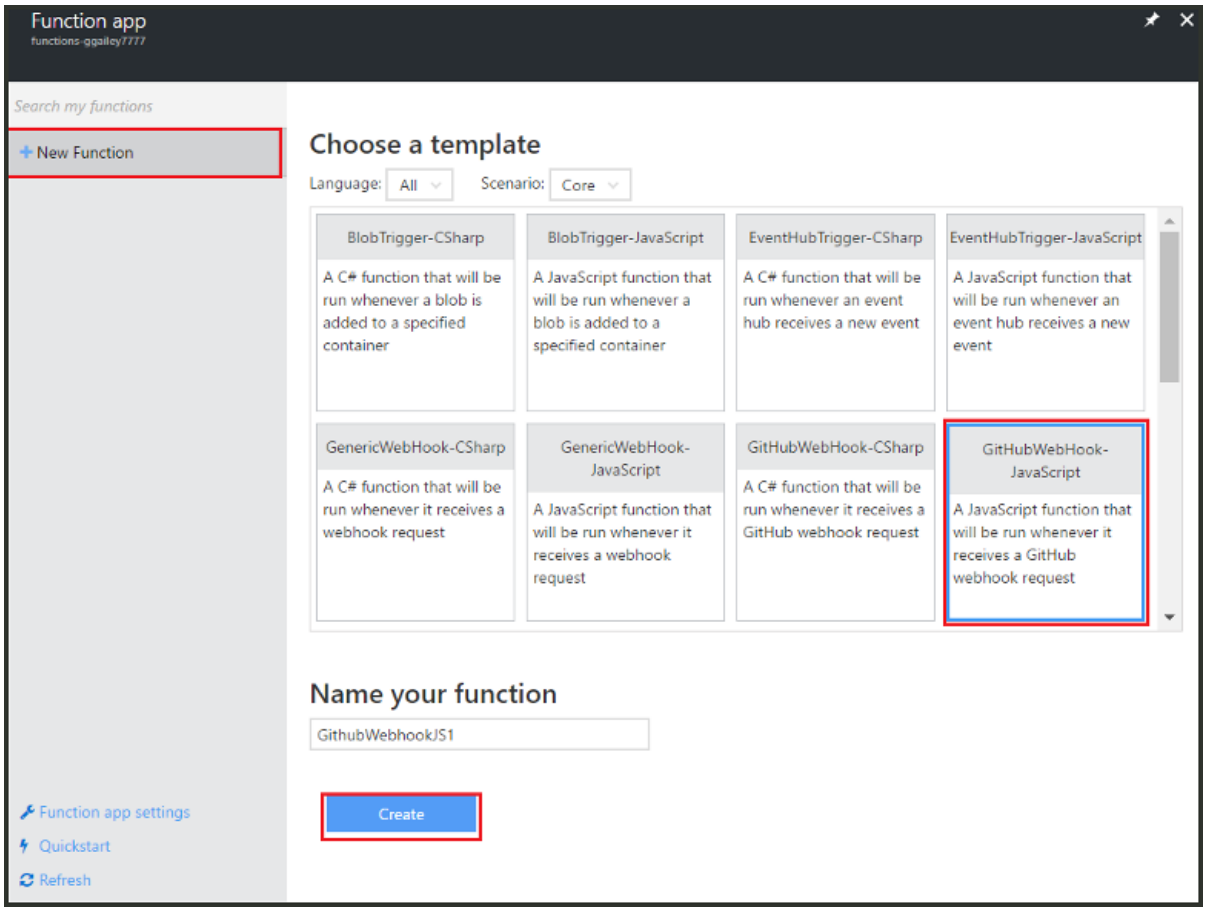
To complete this tutorial, you will need the following:

- An active Azure account. If you don't already have an account, you can sign up for a free Azure acccount. You can also use the Try Functions experience to complete this tutorial without an Azure account.
- A GitHub account. You can sign up for a free GitHub account, if you don't already have one.

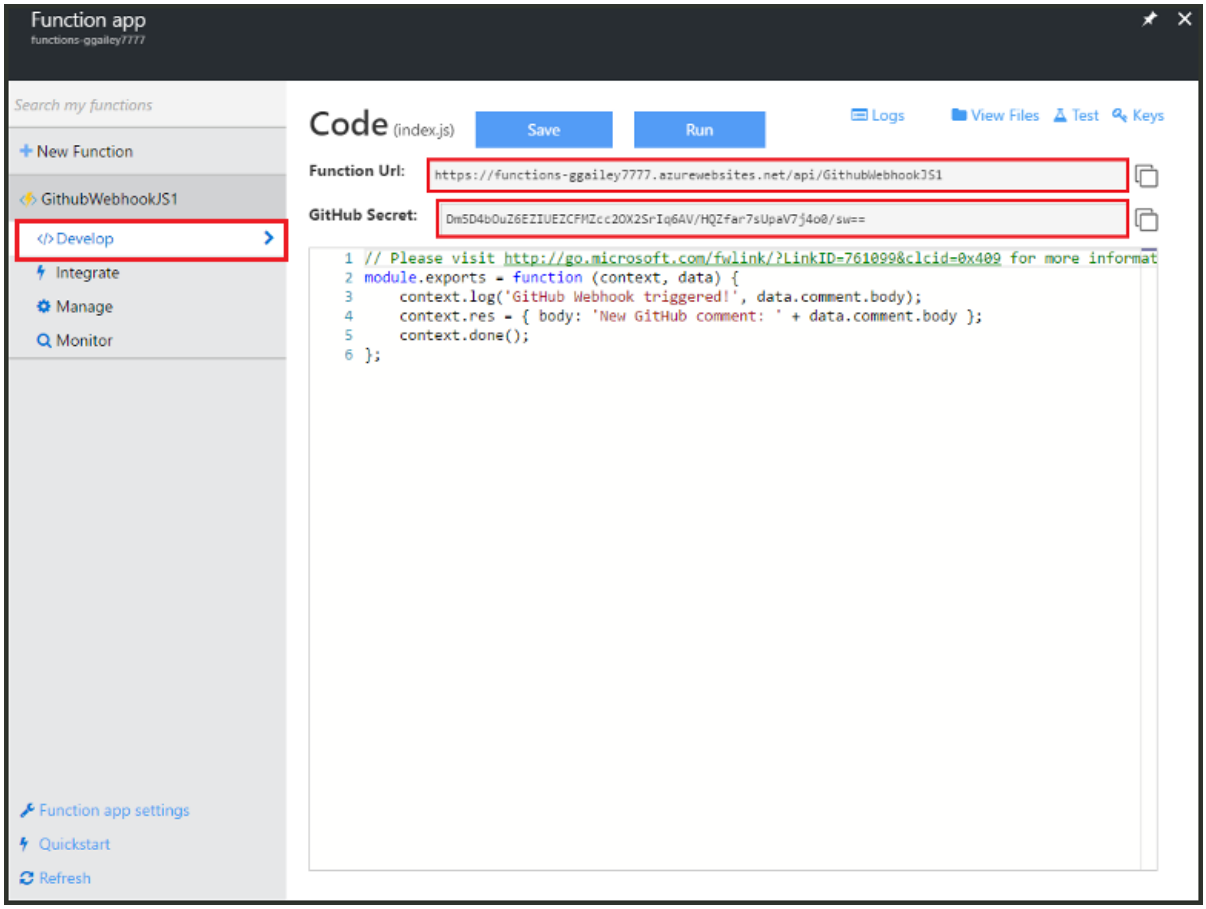
Create a webhook-triggered function from the template

A function app hosts the execution of your functions in Azure.

1. Go to the Azure Functions portal and sign-in with your Azure account.
2. If you have an existing function app to use, select it from **Your function apps** then click **Open**. To create a function app, type a unique **Name** for your new function app or accept the generated one, select your preferred **Region**, then click **Create + get started**.
3. In your function app, click **+ New Function** > **GitHub Webhook - JavaScript** > **Create**. This step creates a function with a default name that is based on the specified template. You can alternately create a C# or F# function.

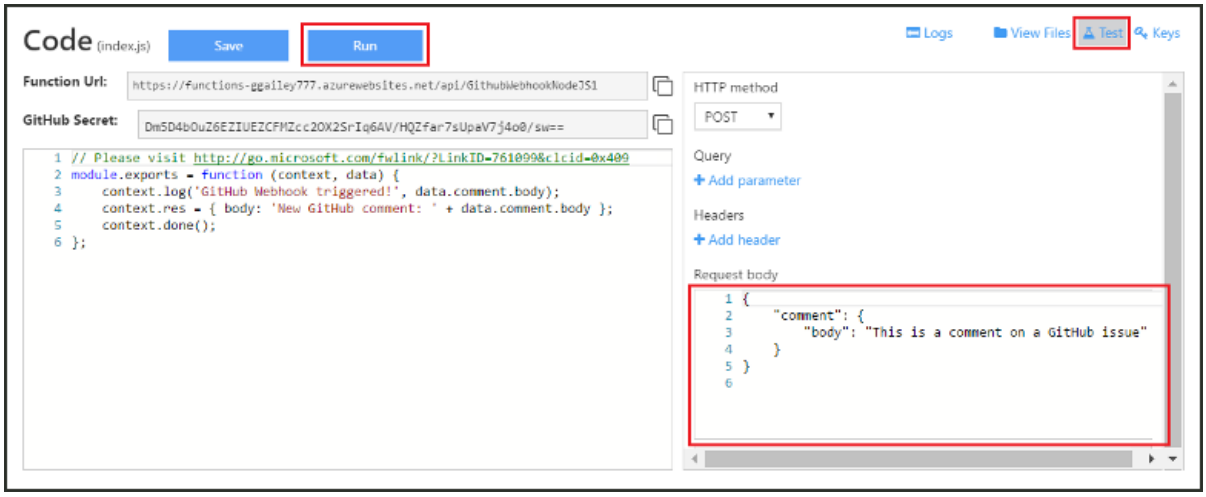


4. In **Develop**, note the sample express.js function in the **Code** window. This function receives a GitHub request from an issue comment webhook, logs the issue text and sends a response to the webhook as `New GitHub comment: <Your issue comment text>`.



5. Copy and save the **Function URL** and **GitHub Secret** values. You will use these values in the next section to configure the webhook in GitHub.

6. Click **Test**, note the predefined JSON body of an issue comment in the **Request body**, then click **Run**.

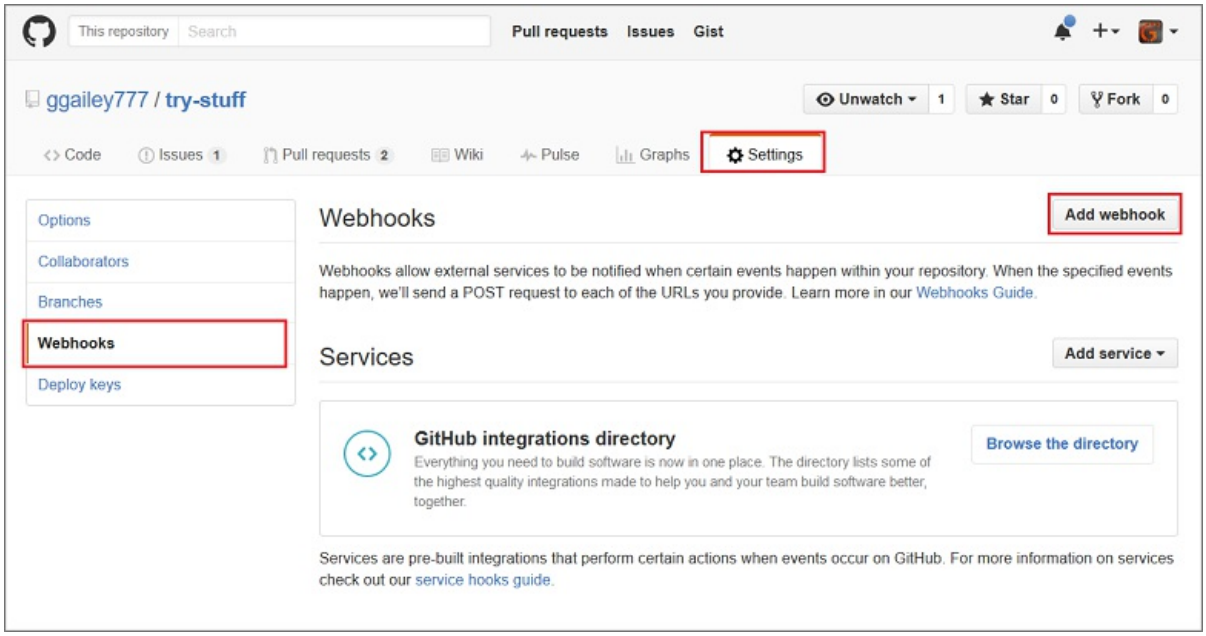


> **NOTE**
>
> You can always test a new template-based function right in the **Develop** tab by supplying any expected body JSON data and clicking the **Run** button. In this case, the template has a predefined body for an issue comment.

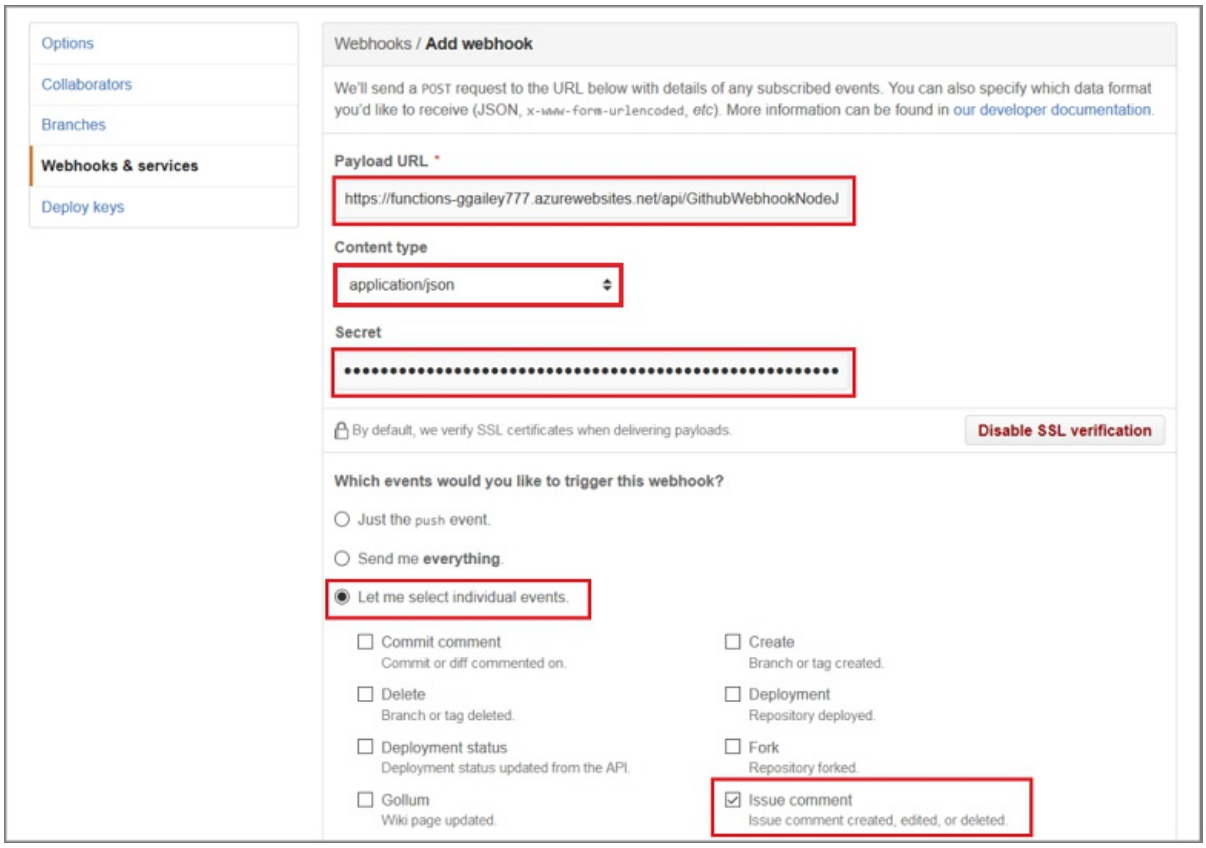
Next, you will create the actual webhook in your GitHub repository.

Configure the webhook

1. In GitHub, navigate to a repository that you own. You can also use any repositories that you have forked.
2. Click **Settings** > **Webhooks & services** > **Add webhook**.



3. Paste your function's URL and secret into **Payload URL** and **Secret** and select **application/json** for **Content type**.
4. Click **Let me select individual events**, select **Issue comment**, and click **Add webhook**.



At this point, the GitHub webhook is configured to trigger your function when a new issue comment is added. Now, it's time to test it out.

Test the function

1. In your GitHub repo, open the **Issues** tab in a new browser window.
2. In the new window, click **New Issue**, type a title then click **Submit new issue**. You can also open an existing issue.
3. In the issue, type a comment and click **Comment**.
4. In the other GitHub window, click **Edit** next to your new webhook, scroll down to **Recent Deliveries**, and verify that a webhook request was sent and that the body of response is `New GitHub comment: <Your issue comment text>`.
5. Back in the Functions portal, scroll down to the logs and see that the function has been triggered and the value `New GitHub comment: <Your issue comment text>` is written to the streaming logs.

Next steps

See these topics for more information about Azure Functions.

- Azure Functions developer reference
  Programmer reference for coding functions.
- 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.

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions`. - Visit Stack Overflow

Use Azure Functions to create a function that connects to other Azure services

2/6/2017 • 5 min to read • Edit on GitHub

This topic shows you how to create a function in Azure Functions that listens to messages on an Azure Storage queue and copies the messages to rows in an Azure Storage table. A timer triggered function is used to load messages into the queue. A second function reads from the queue and writes messages to the table. Both the queue and the table are created for you by Azure Functions based on the binding definitions.

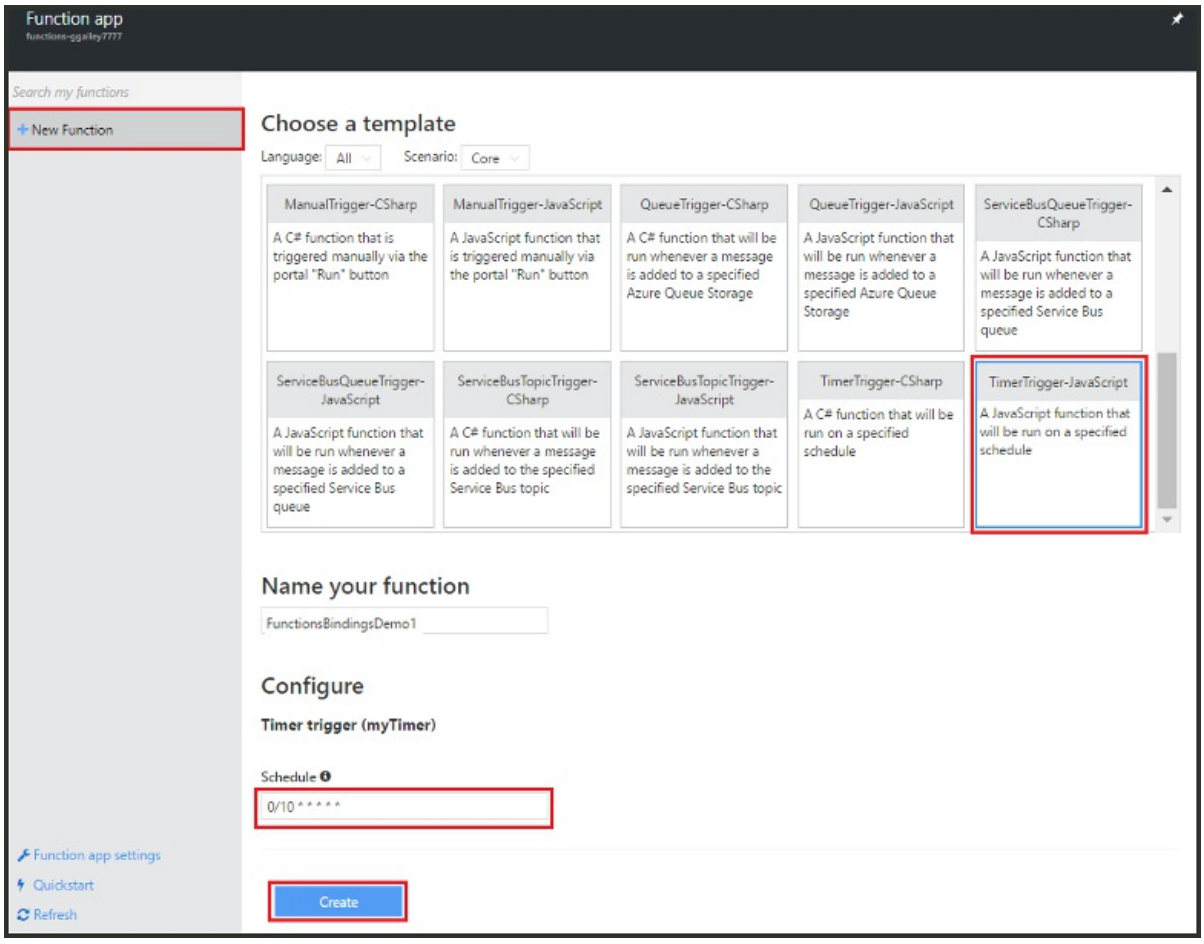
To make things more interesting, one function is written in JavaScript and the other is written in C# script. This demonstrates how a function app can have functions in various languages.

You can see this scenario demonstrated in a video on Channel 9.

Create a function that writes to the queue

Before you can connect to a storage queue, you need to create a function that loads the message queue. This JavaScript function uses a timer trigger that writes a message to the queue every 10 seconds. If you don't already have an Azure account, check out the Try Azure Functions experience, or create your free Azure acccount.

1. Go to the Azure portal and locate your function app.
2. Click **New Function** > **TimerTrigger-JavaScript**.
3. Name the function **FunctionsBindingsDemo1**, enter a cron expression value of `0/10 * * * * *` for **Schedule**, and then click **Create**.

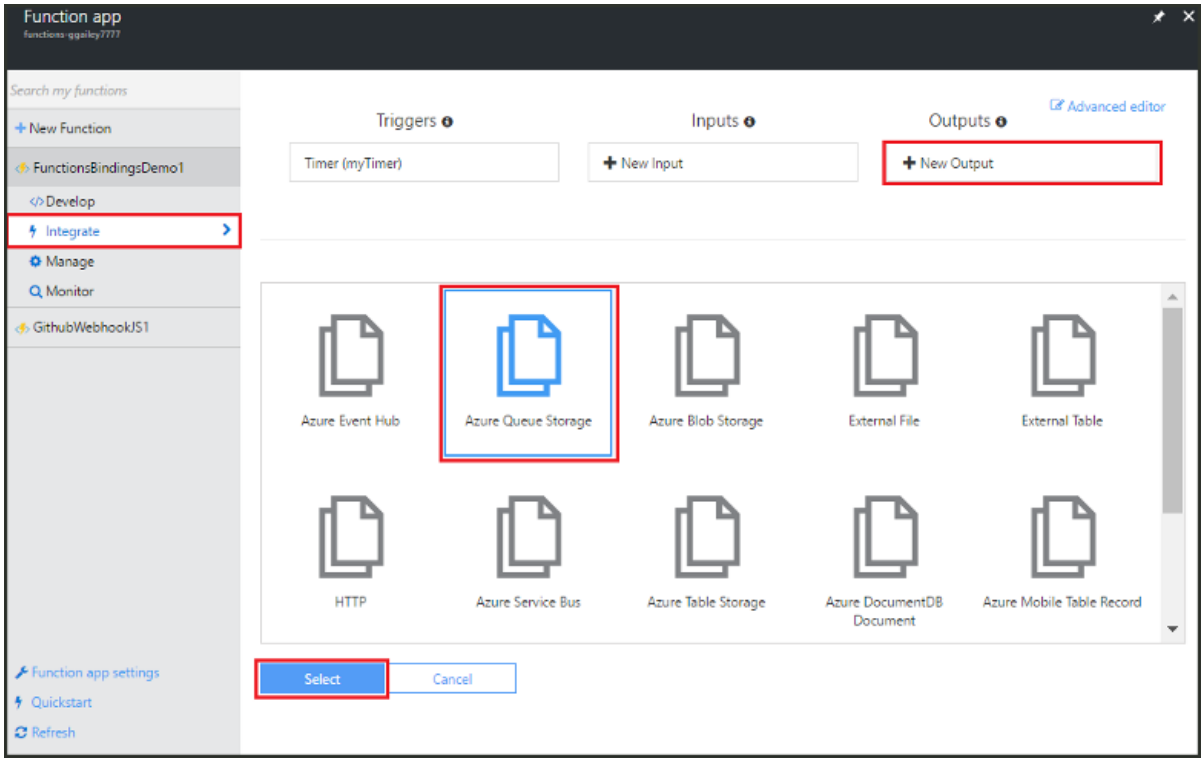


You have now created a timer triggered function that runs every 10 seconds.

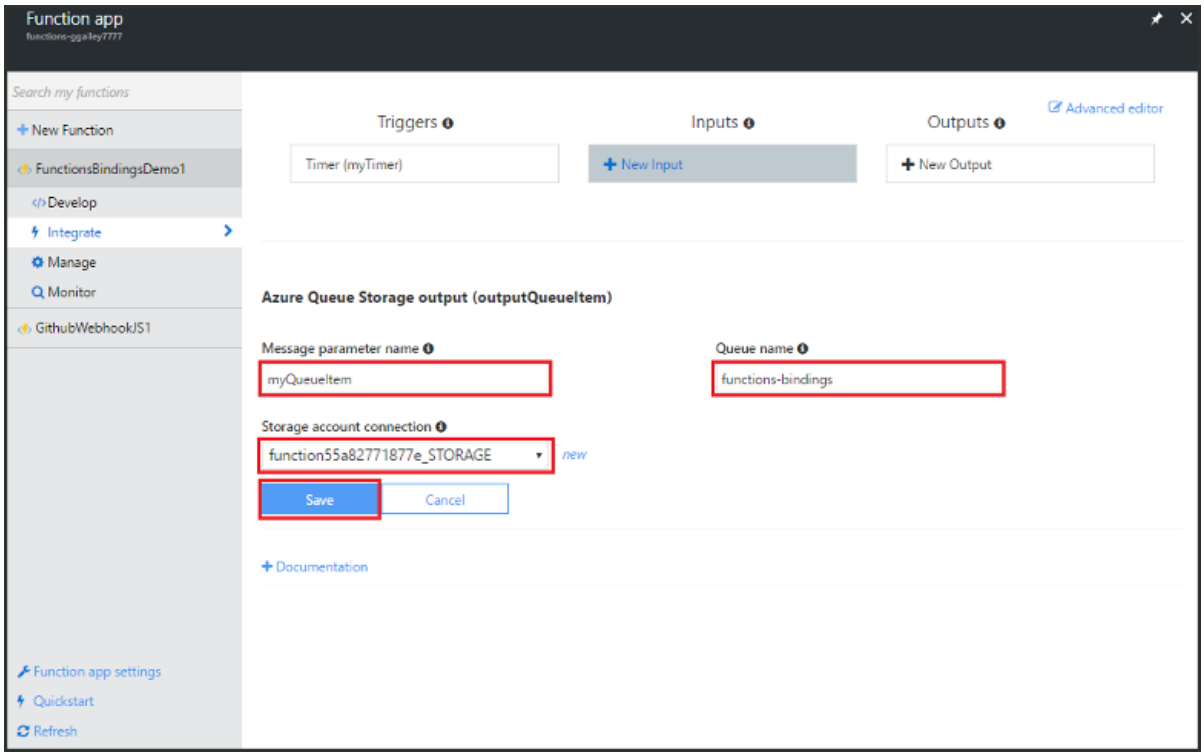
4. On the **Develop** tab, click **Logs** and view the activity in the log. You see a log entry written every 10 seconds.

Add a message queue output binding

1. On the **Integrate** tab, choose **New Output** > **Azure Queue Storage** > **Select**.



2. Enter `myQueueItem` for **Message parameter name** and `functions-bindings` for **Queue name**, select an existing **Storage account connection** or click **new** to create a storage account connection, and then click **Save**.



3. Back in the **Develop** tab, append the following code to the function:

```
function myQueueItem()
{
    return {
        msg: "some message goes here",
        time: "time goes here"
    }
}
```

4. Locate the *if* statement around line 9 of the function, and insert the following code after that statement.

```
var toBeQed = myQueueItem();
toBeQed.time = timeStamp;
context.bindings.myQueueItem = toBeQed;
```

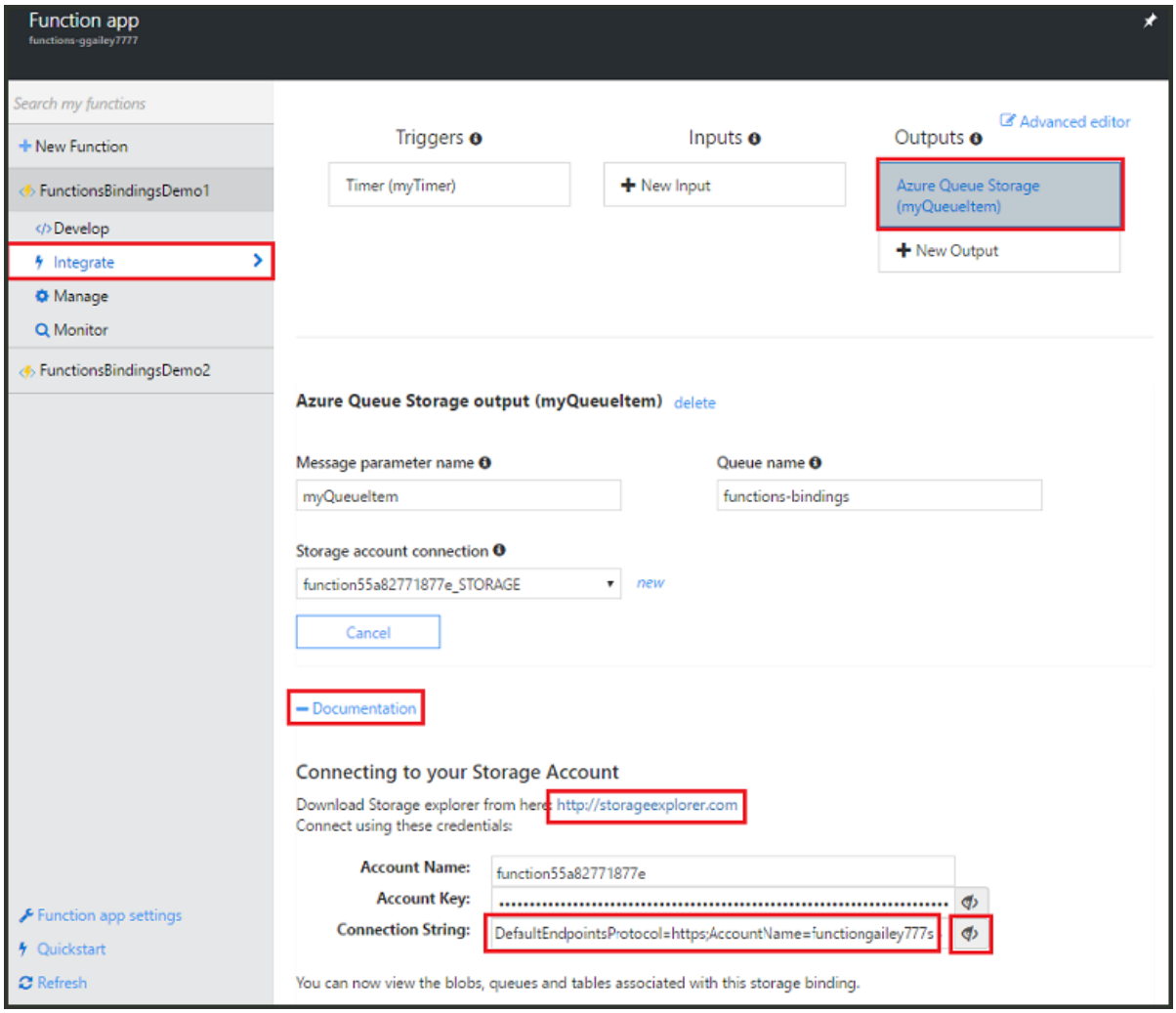
   This code creates a **myQueueItem** and sets its **time** property to the current timeStamp. It then adds the new queue item to the context's **myQueueItem** binding.

5. Click **Save and Run**.

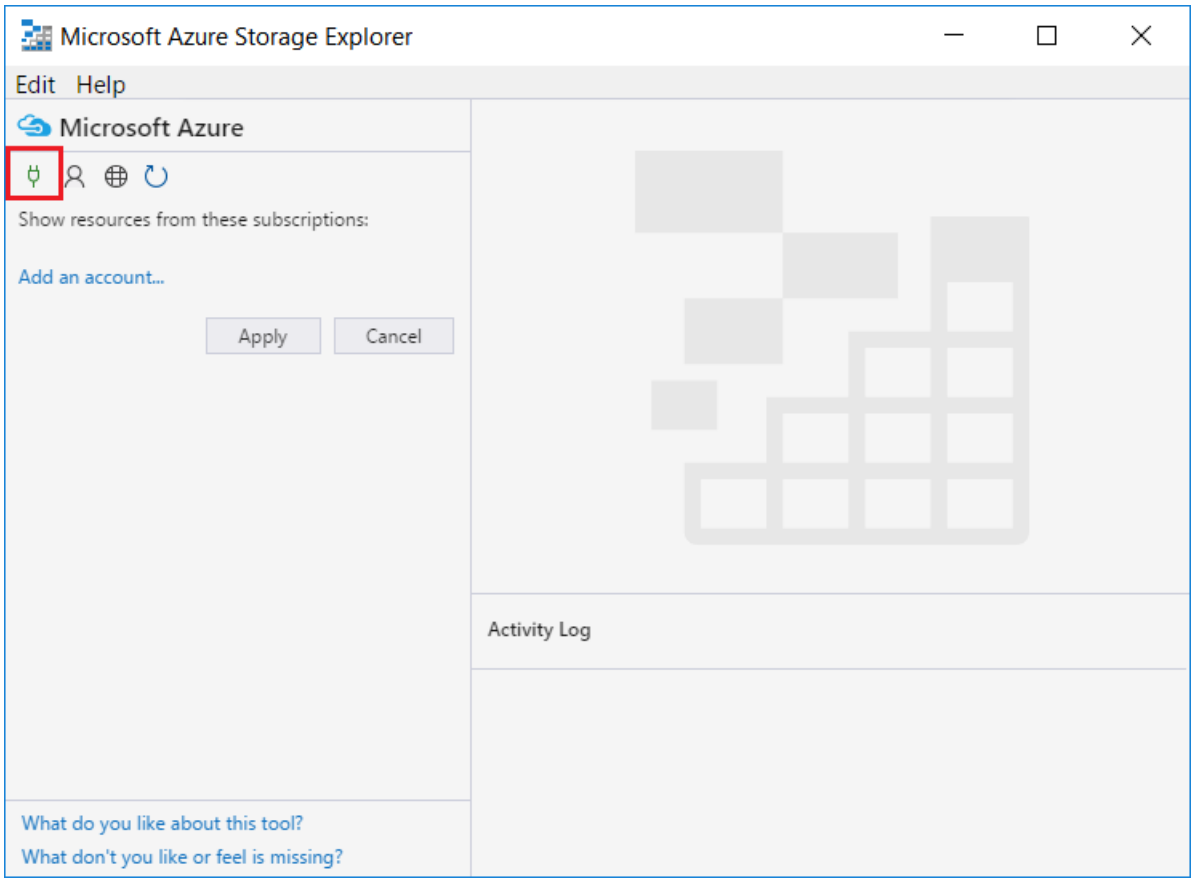
View storage updates by using Storage Explorer

You can verify that your function is working by viewing messages in the queue you created. You can connect to your storage queue by using Cloud Explorer in Visual Studio. However, the portal makes it easy to connect to your storage account by using Microsoft Azure Storage Explorer.

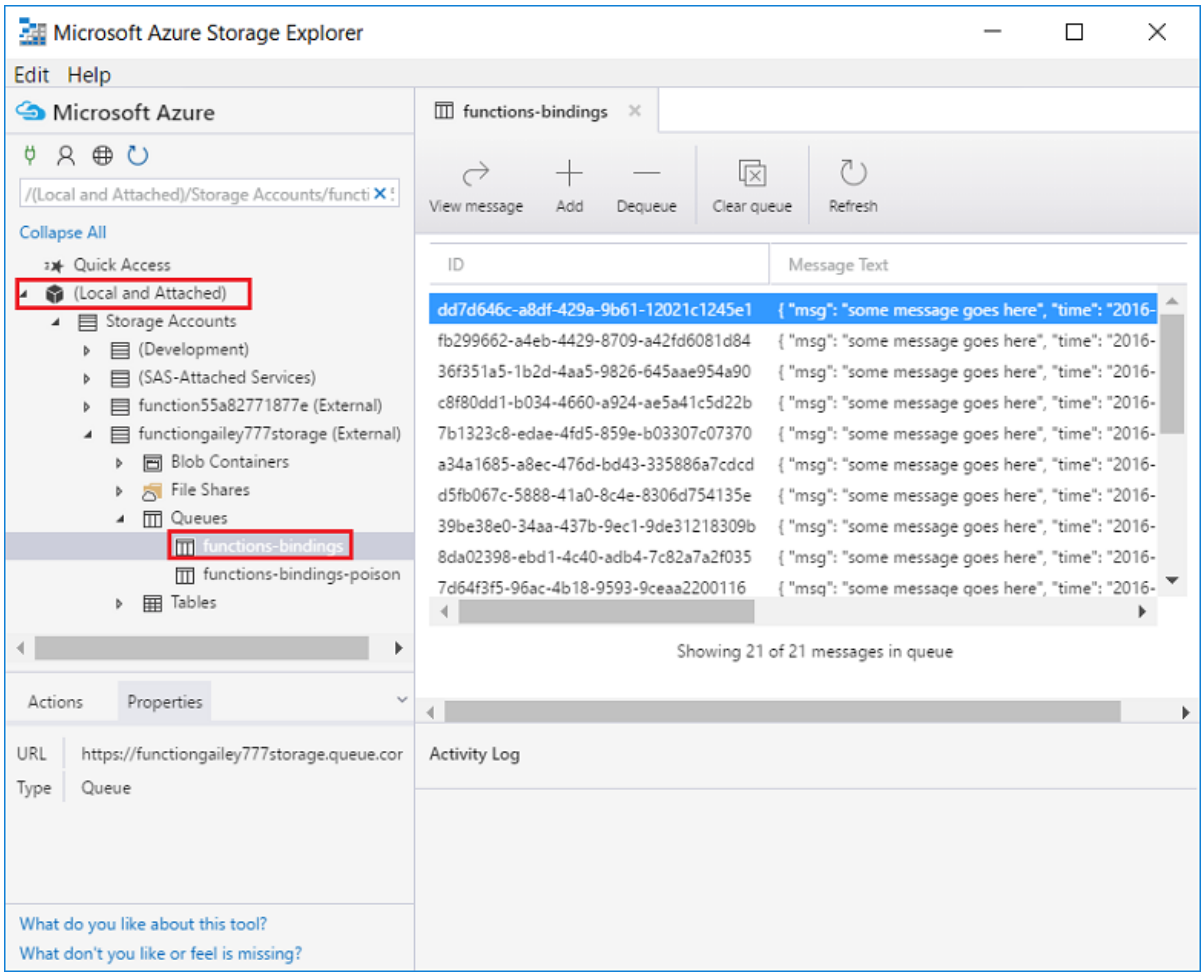
1. In the **Integrate** tab, click your queue output binding > **Documentation**, then unhide the Connection String for your storage account and copy the value. You use this value to connect to your storage account.



2. If you haven't already done so, download and install Microsoft Azure Storage Explorer.
3. In Storage Explorer, click the connect to Azure Storage icon, paste the connection string in the field, and complete the wizard.



4. Under **Local and attached**, expand **Storage Accounts** > your storage account > **Queues** > **functions-bindings** and verify that messages are written to the queue.

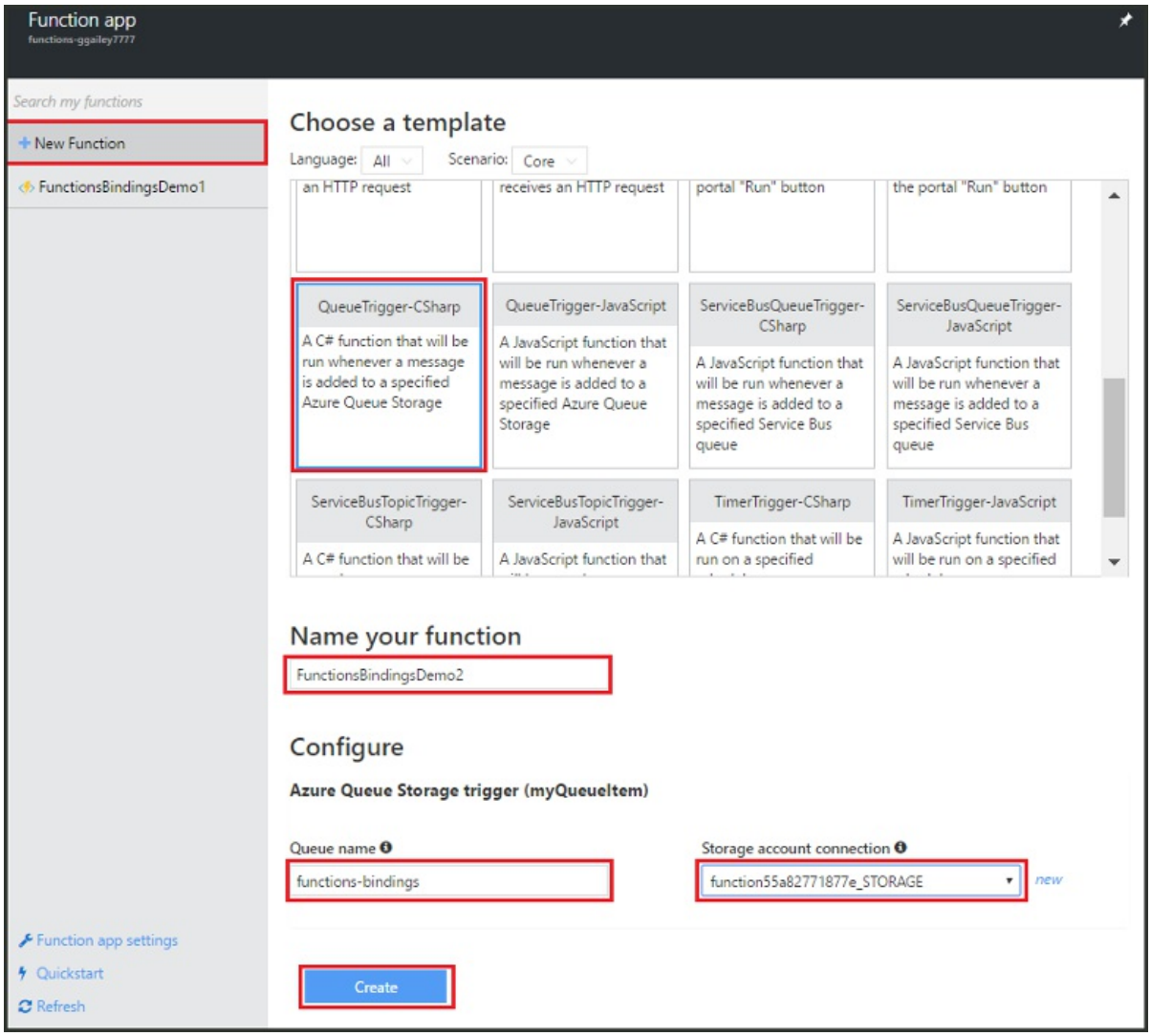


If the queue does not exist or is empty, there is most likely a problem with your function binding or code.

Create a function that reads from the queue

Now that you have messages being added to the queue, you can create another function that reads from the queue and writes the messages permanently to an Azure Storage table.

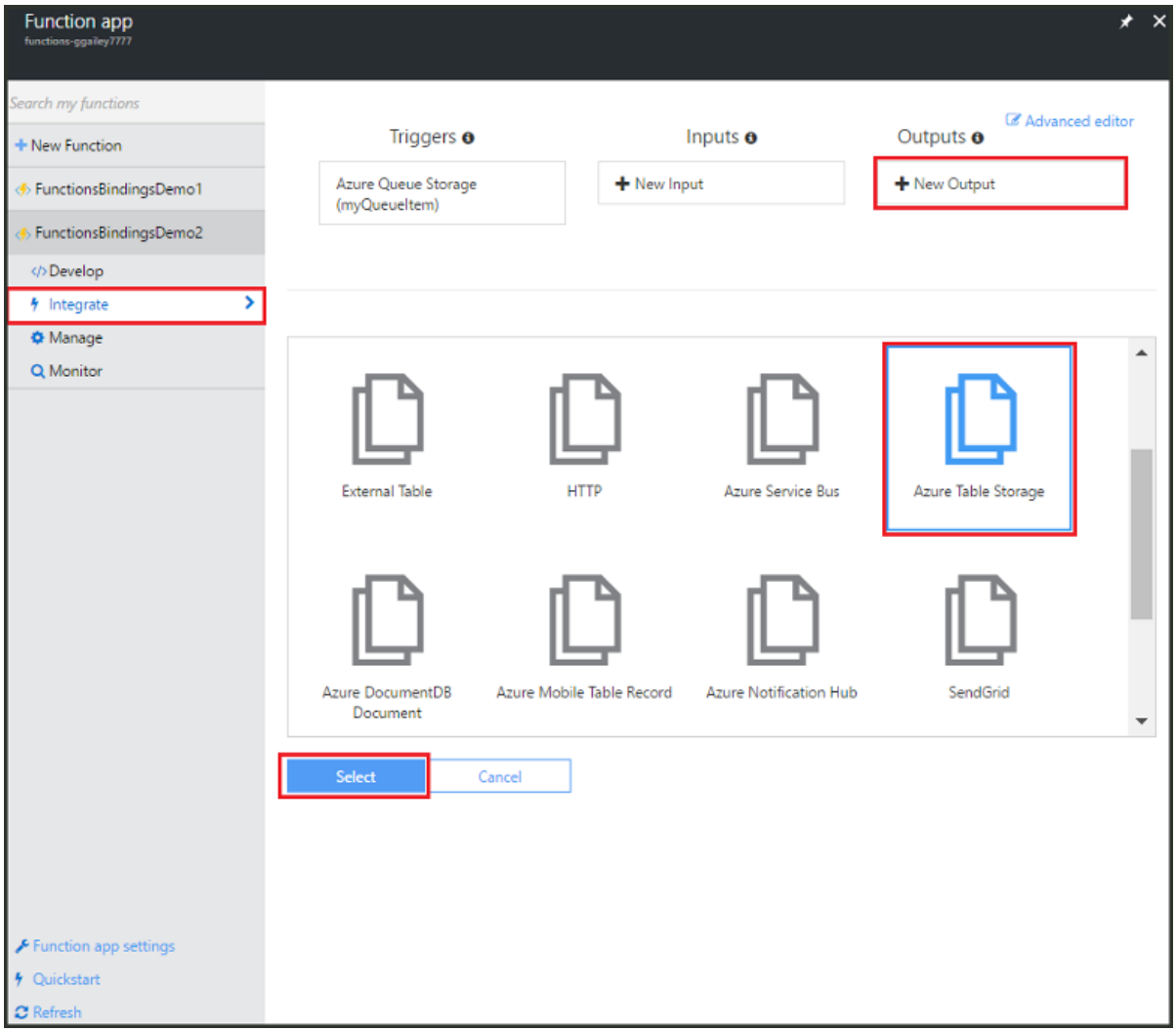
1. Click **New Function** > **QueueTrigger-CSharp**.
2. Name the function `FunctionsBindingsDemo2`, enter **functions-bindings** in the **Queue name** field, select an existing storage account or create one, and then click **Create**.



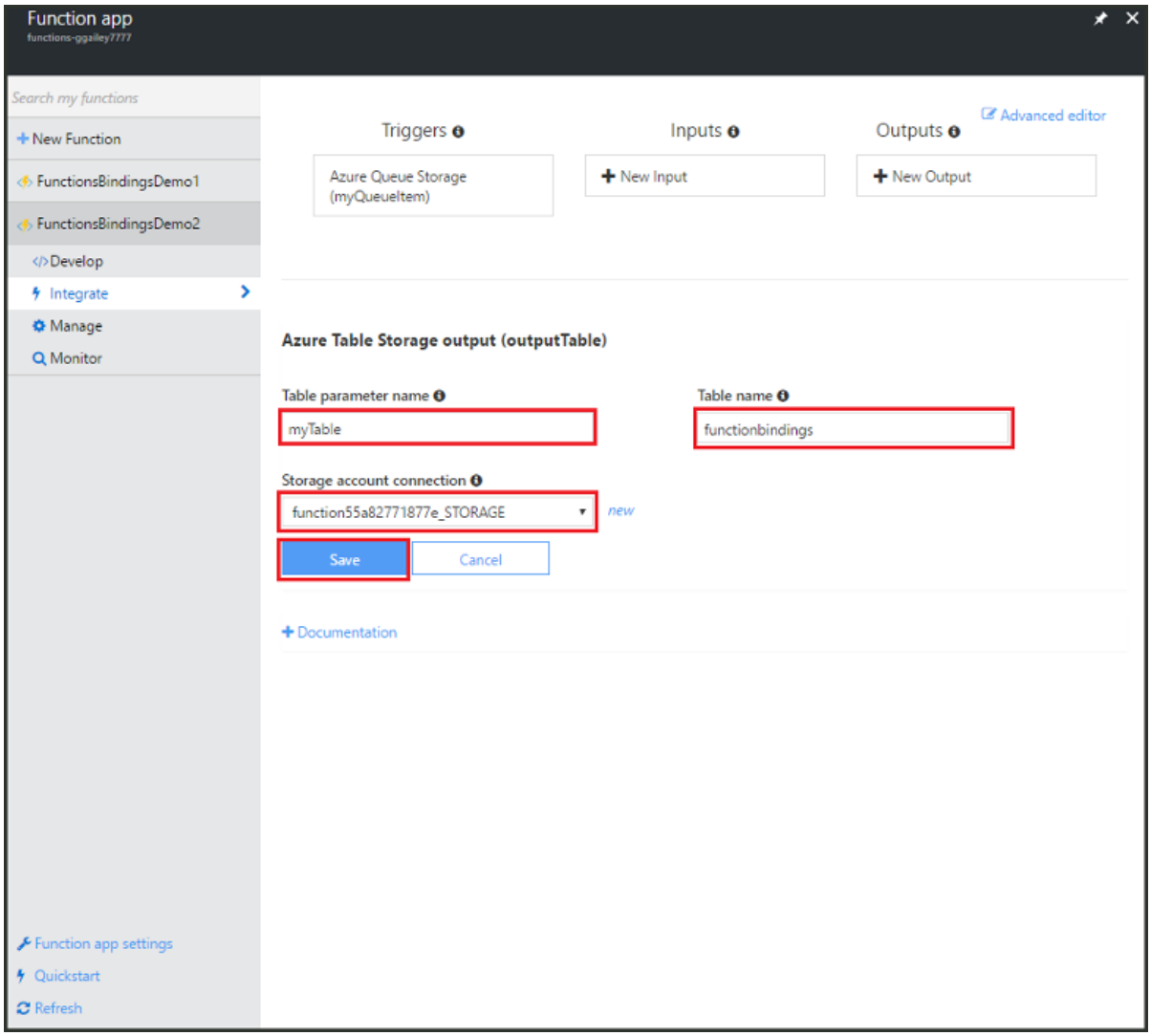
3. (Optional) You can verify that the new function works by viewing the new queue in Storage Explorer as before. You can also use Cloud Explorer in Visual Studio.
4. (Optional) Refresh the **functions-bindings** queue and notice that items have been removed from the queue. The removal occurs because the function is bound to the **functions-bindings** queue as an input trigger and the function reads the queue.

Add a table output binding

1. In FunctionsBindingsDemo2, click **Integrate** > **New Output** > **Azure Table Storage** > **Select**.



2. Enter `TableItem` for **Table name** and `functionbindings` for **Table parameter name**, choose a **Storage account connection** or create a new one, and then click **Save**.



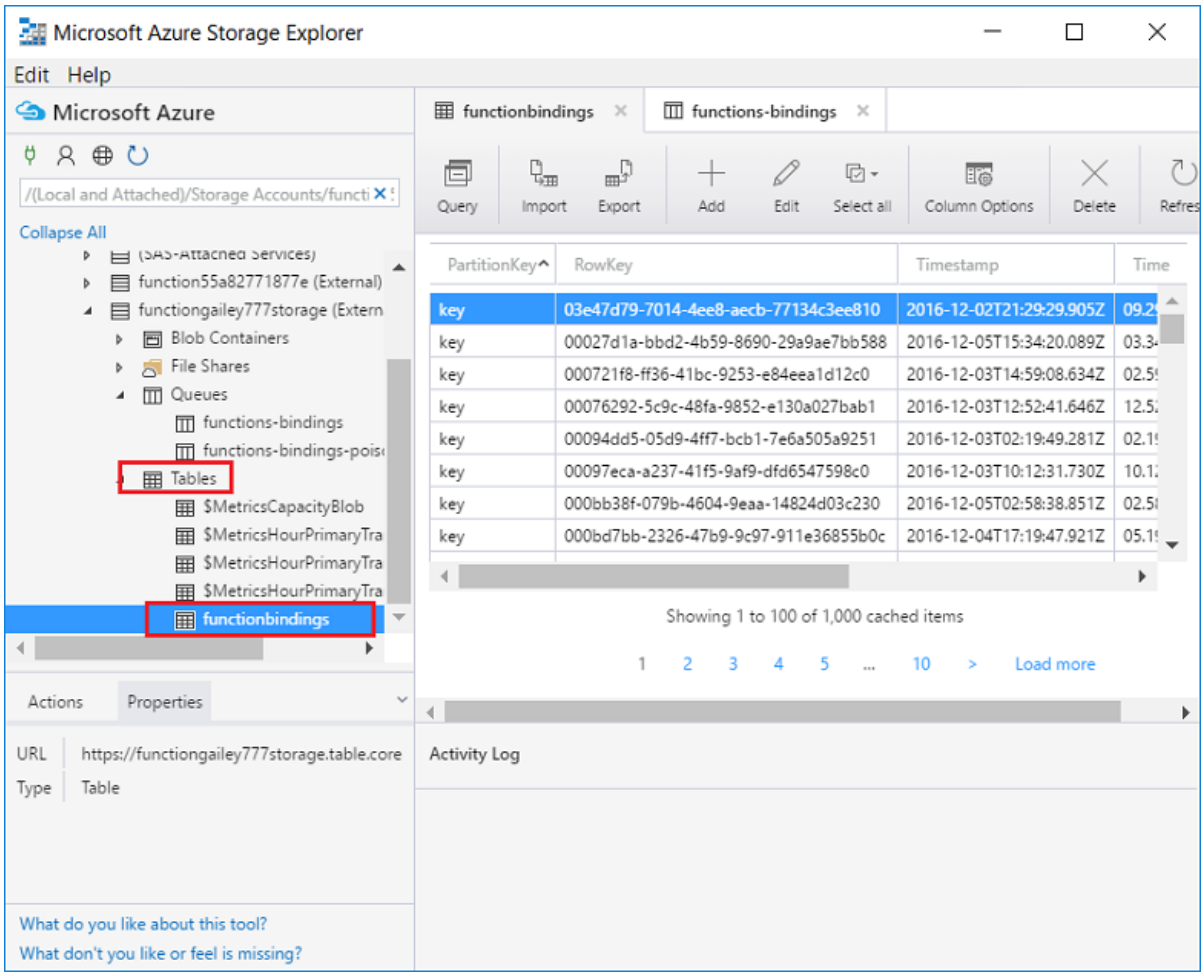
3. In the **Develop** tab, replace the existing function code with the following:

```
using System;

public static void Run(QItem myQueueItem, ICollector<TableItem> myTable, TraceWriter log)
{
    TableItem myItem = new TableItem
    {
        PartitionKey = "key",
        RowKey = Guid.NewGuid().ToString(),
        Time = DateTime.Now.ToString("hh.mm.ss.ffffff"),
        Msg = myQueueItem.Msg,
        OriginalTime = myQueueItem.Time
    };

    // Add the item to the table binding collection.
    myTable.Add(myItem);

    log.Verbose($"C# Queue trigger function processed: {myItem.RowKey} | {myItem.Msg} |
{myItem.Time}");
}

public class TableItem
{
    public string PartitionKey {get; set;}
    public string RowKey {get; set;}
    public string Time {get; set;}
    public string Msg {get; set;}
    public string OriginalTime {get; set;}
}

public class QItem
{
    public string Msg { get; set;}
    public string Time { get; set;}
}
```

The **TableItem** class represents a row in the storage table, and you add the item to the `myTable` collection of **TableItem** objects. You must set the **PartitionKey** and **RowKey** properties to be able to insert into the table.

4. Click **Save**. Finally, you can verify the function works by viewing the table in Storage explorer or Visual Studio Cloud Explorer.

5. (Optional) In your storage account in Storage Explorer, expand **Tables** > **functionsbindings** and verify that rows are added to the table. You can do the same in Cloud Explorer in Visual Studio.



If the table does not exist or is empty, there is most likely a problem with your function binding or code.

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Next steps

See these topics for more information about Azure Functions.

- 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.
- How to scale Azure Functions
  Discusses service plans available with Azure Functions, including the Consumption hosting plan, and how to choose the right plan.

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions`. - Visit Stack Overflow

Create an event processing Azure Function

1/17/2017 • 2 min to read • Edit on GitHub

Azure Functions is an event-driven, compute-on-demand experience that enables you to create scheduled or triggered units of code implemented in a variety of programming languages. To learn more about Azure Functions, see the Azure Functions Overview.

This topic shows you how to create a new function in C# that executes based on an event timer to add messages to a storage queue.

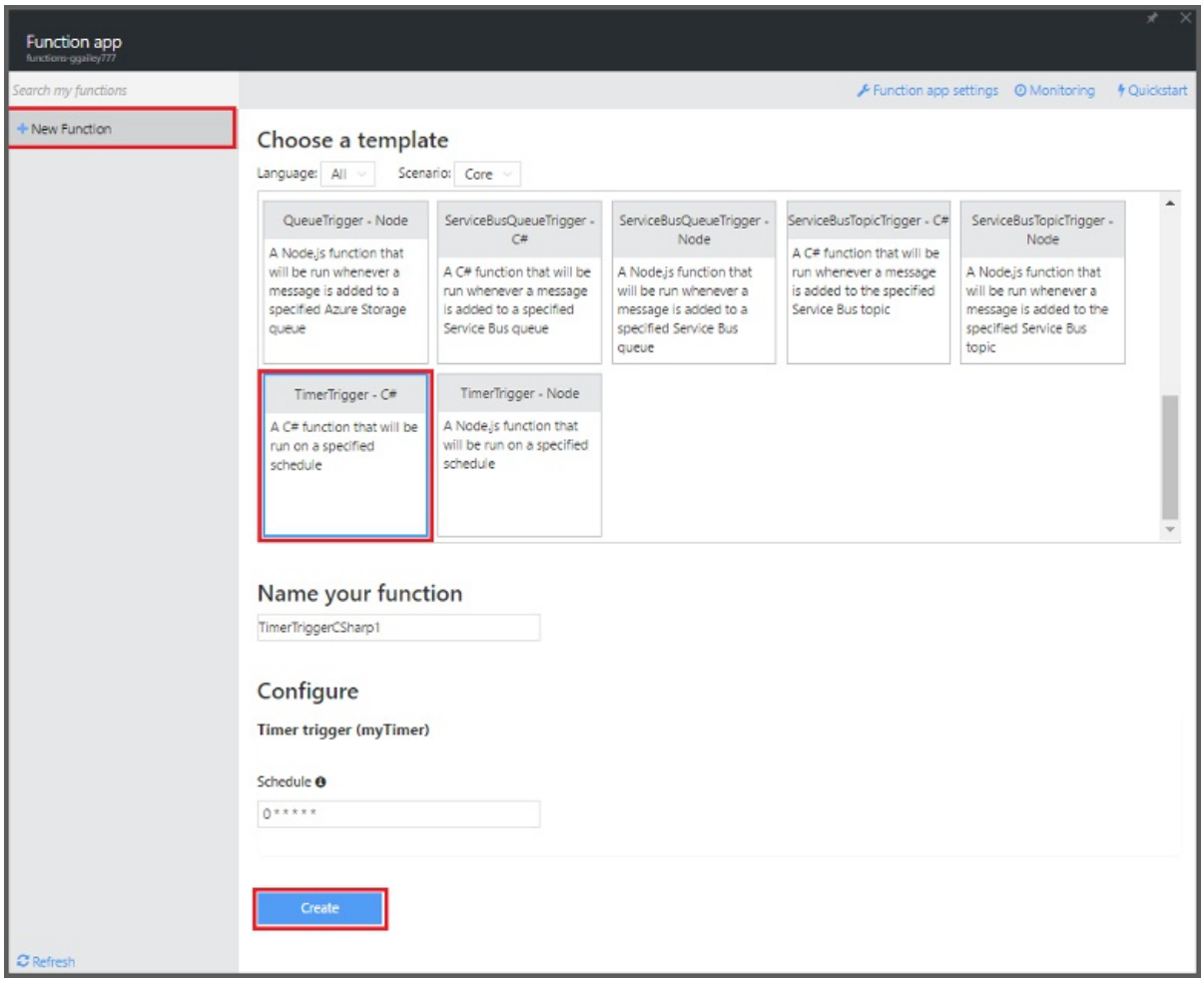
Prerequisites

A function app hosts the execution of your functions in Azure. If you don't already have an Azure account, check out the Try Functions experience or create a free Azure acccount.

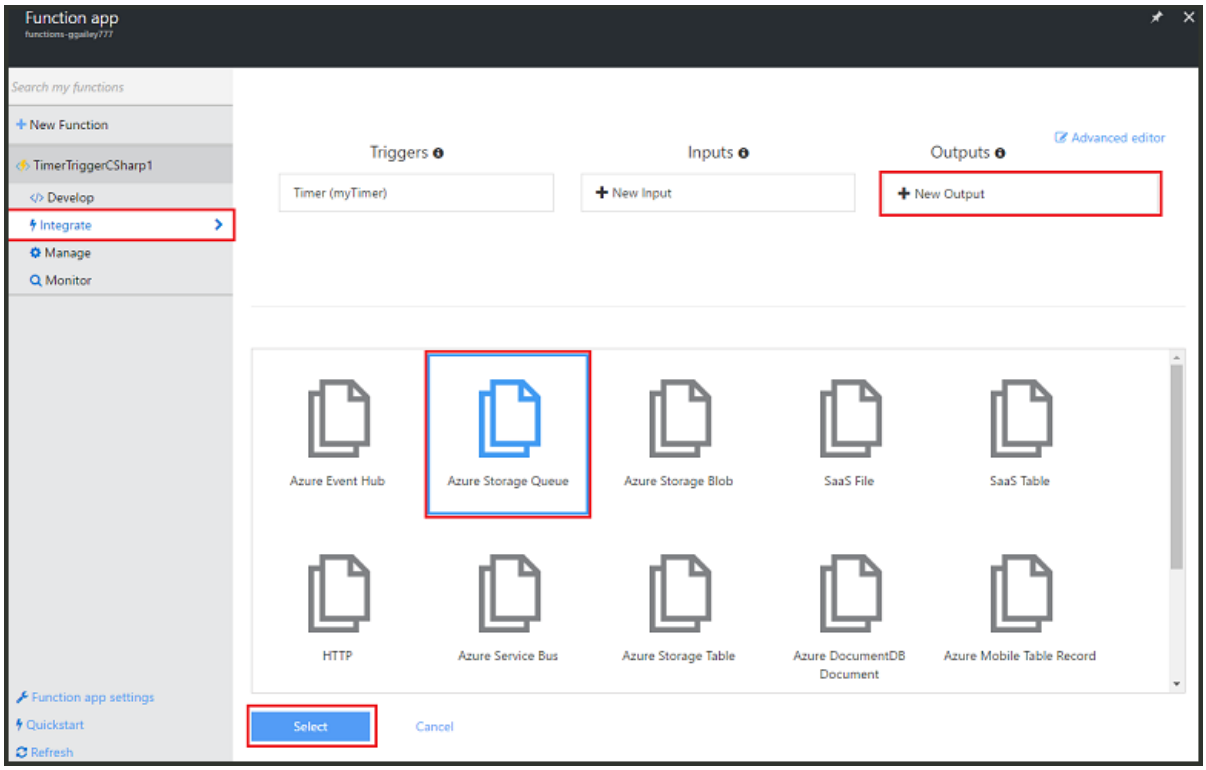
Create a timer-triggered function from the template

A function app hosts the execution of your functions in Azure. Before you can create a function, you need to have an active Azure account. If you don't already have an Azure account, free accounts are available.

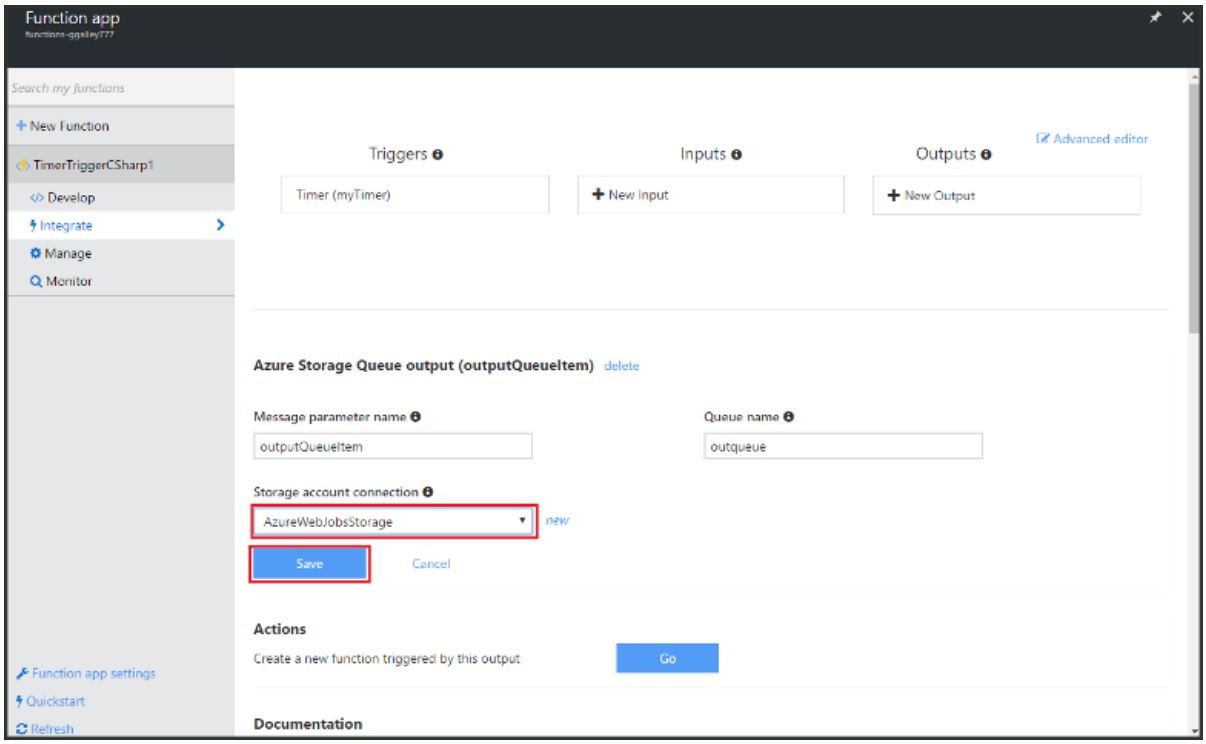
1. Go to the Azure Functions portal and sign-in with your Azure account.
2. If you have an existing function app to use, select it from **Your function apps** then click **Open**. To create a new function app, type a unique **Name** for your new function app or accept the generated one, select your preferred **Region**, then click **Create + get started**.
3. In your function app, click **+ New Function** > **TimerTrigger - C#** > **Create**. This creates a function with a default name that is run on the default schedule of once every minute.



4. In your new function, click the **Integrate** tab > **New Output** > **Azure Storage Queue** > **Select**.



5. In **Azure Storage Queue output**, select an existing **Storage account connection**, or create a new one, then click **Save**.



6. Back in the **Develop** tab, replace the existing C# script in the **Code** window with the following code:

```
using System;

public static void Run(TimerInfo myTimer, out string outputQueueItem, TraceWriter log)
{
    // Add a new scheduled message to the queue.
    outputQueueItem = $"Ping message added to the queue at: {DateTime.Now}.";

    // Also write the message to the logs.
    log.Info(outputQueueItem);
}
```

   This code adds a new message to the queue with the current date and time when the function is executed.

7. Click **Save** and watch the **Logs** windows for the next function execution.
8. (Optional) Navigate to the storage account and verify that messages are being added to the queue.
9. Go back to the **Integrate** tab and change the schedule field to `0 0 * * * *`. The function now runs once every hour.

This is a very simplified example of both a timer trigger and a storage queue output binding. For more information, see both the Azure Functions timer trigger and the Azure Functions triggers and bindings for Azure Storage topics.

Next steps

See these topics for more information about Azure Functions.

- 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.
- How to scale Azure Functions
  Discusses service plans available with Azure Functions, including the Consumption hosting plan, and how to choose the right plan.

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions`. - Visit Stack Overflow

Choose between Flow, Logic Apps, Functions, and WebJobs

1/20/2017 • 4 min to read • Edit on GitHub

This article compares and contrasts the following services in the Microsoft cloud, which can all solve integration problems and automation of 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 adds a unique set of value, 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, which makes 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

Flows empowers any office worker to perform simple integrations (e.g. get SMS for important emails) without going through developers or IT. On the other hand, Logic Apps can enable advanced or mission-critical integrations (e.g. B2B processes) where enterprise-level DevOps and security practices are required. It is typical for a business workflow to grow in complexity overtime. 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	IT pros, developers
Scenarios	Self-service	Mission-critical
Design Tool	In-browser, UI only	In-browser and Visual Studio, Code view available
DevOps	Ad-hoc, develop in production	source control, testing, support, and automation and manageability in Azure Resource Management
Admin Experience	https://flow.microsoft.com	https://portal.azure.com

	FLOW	LOGIC APPS
Security	Standard practices: data sovereignty, 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:

- 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 DocumentDB, Azure Event Hubs, HTTP/WebHook (GitHub, Slack), Azure App Service Mobile Apps, Azure Notification Hubs, Azure Service Bus, Azure Storage	Azure Storage, Azure Service Bus
In-browser development	x	
Window scripting	experimental	x
PowerShell	experimental	x
C#	x	x
F#	x	

	FUNCTIONS	WEBJOBS
Bash	experimental	x
PHP	experimental	x
Python	experimental	x
JavaScript	x	x

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, you should use WebJobs. If you want to run code snippets for other Azure services or even 3rd-party apps, or if you want to manage your integration code snippets separately from your App Service apps, or if you want to call your code snippets from a Logic app, you should take advantage of all the improvements in Functions.

Flow, Logic Apps, and Functions together

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

- For simple business optimization, then use Flow.
- If your integration scenario is too advanced for Flow, or you need DevOps capabilities and security compliances, then use Logic Apps.
- If a step in your integration scenario requires highly custom transformation or specialized code, then write a function app, and then trigger a 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 continue to improve overtime. 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
- Azure WebJobs documentation resources

Scaling Azure Functions

1/17/2017 • 3 min to read • Edit on GitHub

Introduction

The Azure Functions platform allocates compute power when your code is running, scales out as necessary to handle load, and then scales in when code is not running. This means you don't pay for idle VMs or have to reserve capacity before it is needed. The mechanism for this capability is the Consumption service plan. This article provides an overview of how the Consumption service plan works.

If you are not yet familiar with Azure Functions, see the Azure Functions overview article.

Choose a service plan

When you create a function app, you must configure a hosting plan for functions contained in the app. The available hosting plans are: the **Consumption Plan** and the App Service plan. Currently this choice must be made during the creation of the function app. You can not change between these two options after creation. You can scale between tiers on the App Service plan. No changes are currently supported for the Consumption plan as scaling is dynamic.

Consumption plan

In the **Consumption plan**, your Function Apps are assigned to a compute processing instance. If needed more instances are added or removed dynamically. Moreover, your functions run in parallel minimizing the total time needed to process requests. Execution time for each function is aggregated by the containing Function App. Cost is driven by memory size and total execution time across all functions in a Function App as measured in gigabyte-seconds. This is an excellent option if your compute needs are intermittent or your job times tend to be very short as it allows you to only pay for compute resources when they are actually in use.

App service plan

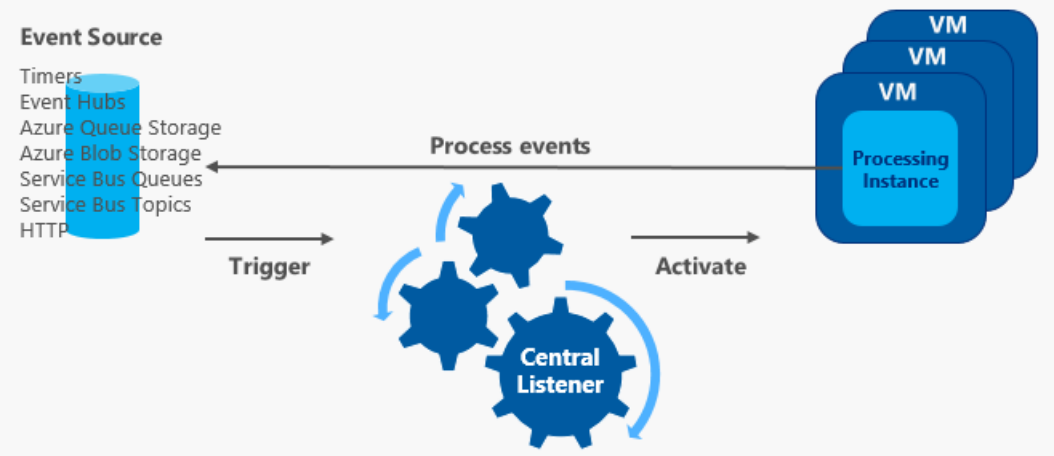
In the **App Service plan**, your Function Apps run on dedicated VMs, just like Web Apps work today for Basic, Standard, or Premium SKUs. Dedicated VMs are allocated to your App Service apps and Function Apps and are always available whether code is being actively executed or not. This is a good option if you have existing, under-utilized VMs that are already running other code or if you expect to run functions continuously or almost continuously. A VM decouples cost from both runtime and memory size. As a result, you can limit the cost of many long-running functions to the cost of the one or more VMs that they run on.

Consumption service plan

The Consumption service plan automatically scales CPU and memory resources by adding additional processing instances based on the runtime needs of the functions in the Function App. Every Function App processing instance is allocated memory resources up to 1.5GB.

Runtime scaling

The Azure Functions platform uses a central listener to evaluate compute needs based on the configured triggers and to decide when to scale out or scale in. The central listener constantly processes hints for memory requirements and trigger specific data points. For example, in the case of an Azure Queue Storage trigger the data points include queue length and queue time for oldest entry.



The unit of scaling is the Function App. Scaling out in this case means adding more instances of a Function App. Inversely as compute demand is reduced, Function App instances are removed - eventually scaling in to zero when none are running.

Billing model

Billing for the Consumption service plan is described in detail on the Azure Functions pricing page. Usage is reported per Function App, only for time when code is being executed. The following are units for billing:

- **Resource consumption in GB-s (gigabyte-seconds)** computed as a combination of memory size and execution time for all functions running in a Function App.
- **Executions** counted each time a function is executed in response to an event, triggered by a binding.

Azure Functions developers guide

1/30/2017 • 6 min to read • Edit on GitHub

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
`type`	string	Binding type. For example, `queueTrigger`.
`direction`	'in', 'out'	Indicates whether the binding is for receiving data into the function or sending data from the function.
`name`	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 of the functions in a given function app lives in a root folder that contains a host configuration file and one or more subfolders, each of which contain 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 configuration and sits in the root folder of the function app. For information on settings that are available, see host.json in the WebJobs.Script repository wiki.

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 **Function app settings**.
2. In the **Advanced Settings** section, click **Go to App Service Settings**.
3. Click **App Service Editor** in App Menu Nav under **DEVELOPMENT TOOLS**.
4. click **Go**.

   After App Service Editor loads, you'll see the *host.json* file and function folders under *wwwroot*.

5. 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 FTP

1. Follow the instructions here to get FTP configured.
2. When you're connected to the function app site, copy an updated *host.json* file to `/site/wwwroot` or copy function files to `/site/wwwroot/<function_name>`.

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.

Azure Functions Pulse

Pulse is a live event stream that shows how often your function runs, as well as successes and failures. You can also monitor your average execution time. We'll be adding more features and customization to it over time. You can access the **Pulse** page from the **Monitoring** tab.

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.

TYPE	SERVICE	TRIGGER	INPUT	OUTPUT
Schedule	Azure Functions	✔		
HTTP (REST or webhook)	Azure Functions	✔		✔*
Blob Storage	Azure Storage	✔	✔	✔
Events	Azure Event Hubs	✔		✔
Queues	Azure Storage	✔		✔
Queues and topics	Azure Service Bus	✔		✔
Tables	Azure Storage		✔	✔
Tables	Azure Mobile Apps		✔	✔
No-SQL DB	Azure DocumentDB		✔	✔
Push Notifications	Azure Notification Hubs			✔
Twilio SMS Text	Twilio			✔

(* - The HTTP output binding requires an HTTP trigger)

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.

Azure Functions C# developer reference

1/17/2017 • 6 min to read • Edit on GitHub

The C# 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 `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 .csx works

The `.csx` format allows you to write less "boilerplate" and focus on writing just a C# function. For Azure Functions, you just include any assembly references and namespaces you need up top, as usual, and instead of wrapping everything in a namespace and class, you can just define your `Run` method. If you need to include any classes, for instance to define Plain Old CLR Object (POCO) objects, you can include a class inside the same file.

Binding to arguments

The various bindings are bound to a C# function via the `name` property in the *function.json* configuration. Each binding has its own supported types which is documented per binding; for instance, a blob trigger can support a string, a POCO, or several other types. You can use the type which best suits your need. A POCO object must have a getter and setter defined for each property.

```
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; }
}
```

Logging

To log output to your streaming logs in C#, you can include a `TraceWriter` typed argument. We recommend that you name it `log`. We recommend you avoid `Console.Write` in Azure Functions.

```
public static void Run(string myBlob, TraceWriter log)
{
    log.Info($"C# Blob trigger function processed: {myBlob}");
}
```

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 Token

In certain cases, you may have operations which are sensitive to being shut down. While it's always best to write code which can handle crashing, in cases where you want to handle graceful shutdown requests, you define a `CancellationToken` typed argument. A `CancellationToken` will be provided if 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`.

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`
- `Microsoft.Azure.NotificationHubs`

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.

Package management

To use NuGet packages in a C# function, upload 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 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 `using` statements to your *run.csx* file to use the types defined in the NuGet packages.

How to upload a project.json file

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 the **How to update function app files** section of 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\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`, 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);
}
```

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* is a common pattern when you want to strongly type your arguments between functions using a POCO object. In the following simplified example, a 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{\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* (C# script) files, not with *.cs* files.

Next steps

For more information, see the following resources:

- Best Practices for Azure Functions
- Azure Functions developer reference
- Azure Functions F# developer reference
- Azure Functions NodeJS developer reference
- Azure Functions triggers and bindings

Azure Functions F# Developer Reference

1/17/2017 • 6 min to read • Edit on GitHub

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 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 Sytem
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 provides 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 C# developer reference
- Azure Functions NodeJS developer reference
- Azure Functions triggers and bindings
- Azure Functions testing
- Azure Functions scaling

Azure Functions JavaScript developer guide

2/6/2017 • 5 min to read • Edit on GitHub

The JavaScript experience for Azure Functions makes it easy to export a function which is passed 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, meaning you can use `arguments` to dynamically handle new inputs (for example, by using `arguments.length` to iterate over all your inputs). This functionality is very convenient if you only have a trigger with no additional inputs, as you can predictably access your trigger data without referencing your `context` object.

The arguments are always passed along to the function in the order 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[3]`.

All bindings, regardless of direction, are also passed along on the `context` object (see below).

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 should always be included because it has methods such as `context.done` and `context.log` which are required to correctly use the runtime. You can name the object whatever you like (i.e. `ctx` or `c`).

```
// You must include a context, but other arguments are optional
module.exports = function(context) {
    // function logic goes here :)
};
```

context.bindings

The `context.bindings` object collects all your input and output data. The data is added onto the `context.bindings` object via the `name` property of the binding. For instance, given the following binding definition in *function.json*, you can access the contents of the queue via `context.bindings.myInput`.

```
{
    "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([err],[propertyBag])`

The `context.done` function tells the runtime that you're done running. This is important to call when you're done with the function; if you don't, the runtime will still never know that your function completed.

The `context.done` function allows you to pass back a user-defined error to the runtime, as well as a property bag of properties which will 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(message)

The `context.log` method allows you to output log statements that are correlated together for logging purposes. If you use `console.log`, your messages will only show for process level logging, which isn't as useful.

```
/* You can use context.log to log output specific to this
function. You can access your bindings via context.bindings */
context.log({hello: 'world'}); // logs: { 'hello': 'world' }
```

The `context.log` method supports the same parameter format that the Node util.format method supports. So, for example, code like this:

```
context.log('Node.js HTTP trigger function processed a request. RequestUri=' + req.originalUrl);
context.log('Request Headers = ' + JSON.stringify(req.headers));
```

can be written like this:

```
context.log('Node.js HTTP trigger function processed a request. RequestUri=%s', req.originalUrl);
context.log('Request Headers = ', JSON.stringify(req.headers));
```

HTTP triggers: context.req and context.res

In the case of HTTP Triggers, because it is such a common pattern to use `req` and `res` for the HTTP request and response objects, we decided to make it easy to access those on the context object, instead of forcing you to use the full `context.bindings.name` pattern.

```
// You can access your http request off of 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!' };
```

Node Version & Package Management

The node version is currently locked at `6.5.0`. We're investigating adding support for more versions and making it configurable.

You can include packages in your function by uploading a *package.json* file to your function's folder in the function app's file system. For file upload instructions, see the **How to update function app files** section of the Azure Functions developer reference topic.

You can also use `npm install` in the function app's SCM (Kudu) command line interface:

1. Navigate to: `https://<function_app_name>.scm.azurewebsites.net`.
2. Click **Debug Console > CMD**.
3. Navigate to `D:\home\site\wwwroot\<function_name>`.
4. Run `npm install`.

Once the packages you need are installed, you import them to your function in the usual ways (i.e. via `require('packagename')`)

```
// 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'});
```

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];
}
```

TypeScript/CoffeeScript support

There isn't, yet, any direct support for auto-compiling TypeScript/CoffeeScript via the runtime, so that would all need 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 C# developer reference
- Azure Functions F# developer reference
- Azure Functions triggers and bindings

Learn how to work with triggers and bindings in Azure Functions

1/30/2017 • 9 min to read • Edit on GitHub

This topic shows you how to use triggers and bindings in Azure Functions to connect your code to a variety of triggers and Azure services and other cloud-based services. It features some of the advanced binding features and syntax supported by all binding types.

For detailed information about working with a specific type of trigger or binding, see one of the following reference topics:

HTTP/webhook	Timer	Mobile Apps	Service Bus
DocumentDB	Storage Blob	Storage Queue	Storage Table
Event Hubs	Notification Hubs	Twilio	

These articles assume that you've read the Azure Functions developer reference, and the C#, F#, or Node.js developer reference articles.

Overview

Triggers are event responses used to trigger your custom code. They allow you to respond to events across the Azure platform or on premise. Bindings represent the necessary meta data used to connect your code to the desired trigger or associated input or output data. The *function.json* file for each function contains all related bindings. There is no limit to the number of input and output bindings a function can have. However, only a single trigger binding is supported for each function.

To get a better idea of the different bindings you can integrate with your Azure Function app, refer to the following table.

TYPE	SERVICE	TRIGGER	INPUT	OUTPUT
Schedule	Azure Functions	✔		
HTTP (REST or webhook)	Azure Functions	✔		✔*
Blob Storage	Azure Storage	✔	✔	✔
Events	Azure Event Hubs	✔		✔
Queues	Azure Storage	✔		✔
Queues and topics	Azure Service Bus	✔		✔
Tables	Azure Storage		✔	✔
Tables	Azure Mobile Apps		✔	✔
No-SQL DB	Azure DocumentDB		✔	✔
Push Notifications	Azure Notification Hubs			✔

TYPE	SERVICE	TRIGGER	INPUT	OUTPUT
Twilio SMS Text	Twilio			✔

(* - The HTTP output binding requires an HTTP trigger)

To better understand triggers and bindings in general, suppose you want to execute some code to process a new item dropped into an Azure Storage queue. Azure Functions provides an Azure Queue trigger to support this. You would need, the following information to monitor the queue:

- The storage account where the queue exists.
- The queue name.
- A variable name that your code would use to refer to the new item that was dropped into the queue.

A queue trigger binding contains this information for an Azure function. Here is an example *function.json* containing a queue trigger binding.

```
{
  "bindings": [
    {
      "name": "myNewUserQueueItem",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "queue-newusers",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    }
  ],
  "disabled": false
}
```

Your code may send different types of output depending on how the new queue item is processed. For example, you might want to write a new record to an Azure Storage table. To do this, you create an output binding to an Azure Storage table. Here is an example *function.json* that includes a storage table output binding that could be used with a queue trigger.

```
{
  "bindings": [
    {
      "name": "myNewUserQueueItem",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "queue-newusers",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "type": "table",
      "name": "myNewUserTableBinding",
      "tableName": "newUserTable",
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

The following C# function responds to a new item being dropped into the queue and writes a new user entry into an Azure Storage table.

```
#r "Newtonsoft.Json"

using System;
using Newtonsoft.Json;

public static async Task Run(string myNewUserQueueItem, IAsyncCollector<Person> myNewUserTableBinding,
                             TraceWriter log)
{
    // In this example the queue item is a JSON string representing an order that contains the name,
    // address and mobile number of the new customer.
    dynamic order = JsonConvert.DeserializeObject(myNewUserQueueItem);

    await myNewUserTableBinding.AddAsync(
        new Person() {
            PartitionKey = "Test",
            RowKey = Guid.NewGuid().ToString(),
            Name = order.name,
            Address = order.address,
            MobileNumber = order.mobileNumber }
        );
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string MobileNumber { get; set; }
}
```

For more code examples and more specific information regarding Azure storage types that are supported, see Azure Functions triggers and bindings for Azure Storage.

To use the more advanced binding features in the Azure portal, click the **Advanced editor** option on the **Integrate** tab of your function. The advanced editor allows you to edit the *function.json* directly in the portal.

Random GUIDs

Azure Functions provides a syntax to generate random GUIDs with your bindings. The following binding syntax writes output to a new BLOB with a unique name in a Storage container:

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

Returning a single output

In cases where your function code returns a single output, you can use an output binding named `$return` to retain a more natural function signature in your code. This can only be used with languages that support a return value (C#, Node.js, F#). The binding would be similar to the following blob output binding that is used with a queue trigger.

```
{
  "bindings": [
    {
      "type": "queueTrigger",
      "name": "input",
      "direction": "in",
      "queueName": "test-input-node"
    },
    {
      "type": "blob",
      "name": "$return",
      "direction": "out",
      "path": "test-output-node/{id}"
    }
  ]
}
```

The following C# code returns the output more naturally without using an `out` parameter in the function signature.

```
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;
}
```

Async example:

```
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 json;
}
```

This same approach is demonstrated with Node.js, as follows:

```
module.exports = function (context, input) {
    var json = JSON.stringify(input);
    context.log('Node.js script processed queue message', json);
    context.done(null, json);
}
```

The following is an F# example:

```
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
```

This can also be used with multiple output parameters by designating a single output with `$return`.

Resolving app settings

It is a best practice to store sensitive information as part of the run-time environment using app settings. By keeping sensitive information out of your app's configuration files, you limit exposure when a public repository is used to store app files.

The Azure Functions run-time resolves app settings to values when the app setting name is enclosed in percent signs, `%your app setting%`. The following Twilio binding uses an app setting named `TWILIO_ACCT_PHONE` for the `from` field of the binding.

```
{
  "type": "twilioSms",
  "name": "$return",
  "accountSid": "TwilioAccountSid",
  "authToken": "TwilioAuthToken",
  "to": "{mobileNumber}",
  "from": "%TWILIO_ACCT_PHONE%",
  "body": "Thank you {name}, your order was received Node.js",
  "direction": "out"
},
```

Parameter binding

Instead of a static configuration setting for your output binding properties, you can configure the settings to be dynamically bound to data that is part of your trigger's input binding. Consider a scenario where new orders are processed using an Azure Storage queue. Each new queue item is a JSON string containing at least the following properties:

```
{
  "name" : "Customer Name",
  "address" : "Customer's Address",
  "mobileNumber" : "Customer's mobile number in the format - +1XXXYYYZZZZ."
}
```

You might want to send the customer an SMS text message using your Twilio account as an update that the order was received. You can configure the `body` and `to` field of your Twilio output binding to be dynamically bound to the `name` and `mobileNumber` that were part of the input as follows.

```
{
  "name": "myNewOrderItem",
  "type": "queueTrigger",
  "direction": "in",
  "queueName": "queue-newOrders",
  "connection": "orders_STORAGE"
},
{
  "type": "twilioSms",
  "name": "$return",
  "accountSid": "TwilioAccountSid",
  "authToken": "TwilioAuthToken",
  "to": "{mobileNumber}",
  "from": "%TWILIO_ACCT_PHONE%",
  "body": "Thank you {name}, your order was received",
  "direction": "out"
},
```

Now your function code only has to initialize the output parameter as follows. During execution, the output properties are bound to the desired input data.

```
#r "Newtonsoft.Json"
#r "Twilio.Api"

using System;
using System.Threading.Tasks;
using Newtonsoft.Json;
using Twilio;

public static async Task<SMSMessage> Run(string myNewOrderItem, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myNewOrderItem}");

    dynamic order = JsonConvert.DeserializeObject(myNewOrderItem);

    // 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();

    // The following isn't needed since we use parameter binding for this
    //string msg = "Hello " + order.name + ", thank you for your order.";
    //smsText.Body = msg;
    //smsText.To = order.mobileNumber;

    return smsText;
}
```

Node.js:

```
module.exports = function (context, myNewOrderItem) {
    context.log('Node.js queue trigger function processed work item', myNewOrderItem);

    // No need to set the properties of the text, we use parameters in the binding. We do need to
    // initialize the object.
    var smsText = {};

    context.done(null, smsText);
}
```

Advanced binding at runtime (imperative binding)

The standard input and output binding pattern using *function.json* is called *declarative* binding, where the binding is defined by the JSON declaration. However, you can use imperative binding. With this patttern, you can bind to any number of supported input and output binding on-the-fly in your function code. You might need imperative binding in cases where the computation of binding path or other inputs needs to happen at run time in your function instead of design time.

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(...)))
{
    ...
}
```

where `BindingTypeAttribute` is the .NET attribute that defines your binding and `T` is the input or output type that's supported by that binding type. `T` also 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`.

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. As is, the code gets the default app setting for the Storage account connection string (which is `AzureWebJobsStorage`). You can specify a custom app setting to use by adding the StorageAccountAttribute and passing the attribute array into `BindAsync<T>()`. 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 shows you the corresponding .NET attribute to use for each binding type and which package to reference.

BINDING	ATTRIBUTE	ADD REFERENCE
DocumentDB	`Microsoft.Azure.WebJobs.DocumentDBAttribu`	`#r "Microsoft.Azure.WebJobs.Extensions.DocumentD`
Event Hubs	`Microsoft.Azure.WebJobs.ServiceBus.EventH`, `Microsoft.Azure.WebJobs.ServiceBusAccountAttribute`	`#r "Microsoft.Azure.Jobs.ServiceBus"`
Mobile Apps	`Microsoft.Azure.WebJobs.MobileTableAttrib`	`#r "Microsoft.Azure.WebJobs.Extensions.MobileApp`
Notification Hubs	`Microsoft.Azure.WebJobs.NotificationHubAt`	`#r "Microsoft.Azure.WebJobs.Extensions.Notificat`
Service Bus	`Microsoft.Azure.WebJobs.ServiceBusAttribu`, `Microsoft.Azure.WebJobs.ServiceBusAccountAttribute`	`#r "Microsoft.Azure.WebJobs.ServiceBus"`
Storage queue	`Microsoft.Azure.WebJobs.QueueAttribute`, `Microsoft.Azure.WebJobs.StorageAccountAttribute`	
Storage blob	`Microsoft.Azure.WebJobs.BlobAttribute`, `Microsoft.Azure.WebJobs.StorageAccountAttribute`	

BINDING	ATTRIBUTE	ADD REFERENCE

Storage table	`Microsoft.Azure.WebJobs.TableAttribute`
	,
	`Microsoft.Azure.WebJobs.StorageAccountAttribute`	

Twilio	`Microsoft.Azure.WebJobs.TwilioSmsAttribut`	`#r "Microsoft.Azure.WebJobs.Extensions.Twilio"`

Next steps

For more information, see the following resources:

- Testing a function
- Scale a function

Azure Functions DocumentDB bindings

1/17/2017 • 6 min to read • Edit on GitHub

This article explains how to configure and code Azure DocumentDB bindings in Azure Functions. Azure Functions supports input and output bindings for DocumentDB.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

For more information on DocumentDB, see Introduction to DocumentDB and Build a DocumentDB console application.

DocumentDB input binding

The DocumentDB input binding retrieves a DocumentDB document and passes it to the named input parameter of the function. The document ID can be determined based on the trigger that invokes the function.

The DocumentDB input to a function uses the following JSON object in the `bindings` array of function.json:

```
{
  "name": "<Name of input parameter in function signature>",
  "type": "documentDB",
  "databaseName": "<Name of the DocumentDB database>",
  "collectionName": "<Name of the DocumentDB collection>",
  "id": "<Id of the DocumentDB document - see below>",
  "connection": "<Name of app setting with connection string - see below>",
  "direction": "in"
},
```

Note the following:

- `id` supports bindings similar to `{queueTrigger}`, which uses the string value of the queue message as the document Id.
- `connection` must be the name of an app setting that points to the endpoint for your DocumentDB account (with the value `AccountEndpoint=<Endpoint for your account>;AccountKey=<Your primary access key>`). If you create a DocumentDB account through the Functions portal UI, the account creation process creates an app setting for you. To use an existing DocumentDB account, you need to [configure this app setting manually]().
- If the specified document is not found, the named input parameter to the function is set to `null`.

Input usage

This section shows you how to use your DocumentDB input binding in your function code.

In C# and F# functions, any changes made to the input document (named input parameter) is automatically sent back to the collection when the function exits successfully. In Node.js functions, updates to the document in the input binding are not sent back to the collection. However, you can use `context.bindings.<documentName>In` and `context.bindings.<documentName>Out` to make updates to input documents. See how it is done in the Node.js sample.

Input sample

Suppose you have the following DocumentDB input binding in the `bindings` array of function.json:

```
{
  "name": "inputDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "id" : "{queueTrigger}",
  "connection": "MyAccount_DOCUMENTDB",
  "direction": "in"
}
```

See the language-specific sample that uses this input binding to update the document's text value.

- C#
- F#
- Node.js

Input sample in C#

```
public static void Run(string myQueueItem, dynamic inputDocument)
{
  inputDocument.text = "This has changed.";
}
```

Input sample in F#

```
open FSharp.Interop.Dynamic
let Run(myQueueItem: string, inputDocument: obj) =
  inputDocument?text <- "This has changed."
```

You need to add 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 sample in Node.js

```
module.exports = function (context) {
  context.bindings.inputDocumentOut = context.bindings.inputDocumentIn;
  context.bindings.inputDocumentOut.text = "This was updated!";
  context.done();
};
```

DocumentDB output binding

The DocumentDB output binding lets you write a new document to an Azure DocumentDB database.

The output binding uses the following JSON object in the `bindings` array of function.json:

```
{
  "name": "<Name of output parameter in function signature>",
  "type": "documentDB",
  "databaseName": "<Name of the DocumentDB database>",
  "collectionName": "<Name of the DocumentDB collection>",
  "createIfNotExists": <true or false - see below>,
  "connection": "<Value of AccountEndpoint in Application Setting - see below>",
  "direction": "out"
}
```

Note the following:

- Set `createIfNotExists` to `true` to create the database and collection if it doesn't exist. The default value is `false`. New collections are created with reserved throughput, which has pricing implications. For more information, see DocumentDB pricing.
- `connection` must be the name of an app setting that points to the endpoint for your DocumentDB account (with the value `AccountEndpoint=<Endpoint for your account>;AccountKey=<Your primary access key>`). If you create a DocumentDB account through the Functions portal UI, the account creation process creates a new app setting for you. To use an existing DocumentDB account, you need to [configure this app setting manually]().

Output usage

This section shows you how to use your DocumentDB output binding in your function code.

When you write to the output parameter in your function, by default a new document is generated in your database, with an automatically generated GUID as the document ID. You can specify the document ID of output document by specifying the `id` JSON property in the output parameter. If a document with that ID already exists, the output document overwrites it.

You can write to the output using any of the following types:

- Any Object - useful for JSON-serialization. If you declare a custom output type (e.g. `out FooType paramName`), Azure Functions attempts to serialize object into JSON. If the output parameter is null when the function exits, the Functions runtime creates a blob as a null object.
- String - (`out string paramName`) useful for text blob data. the Functions runtime creates a blob 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`
- `CloudBlobStream`
- `ICloudBlob`
- `CloudBlockBlob`
- `CloudPageBlob`

To output multiple documents, you can also bind to `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

Output sample

Suppose you have the following DocumentDB output binding in the `bindings` array of function.json:

```
{
  "name": "employeeDocument",
  "type": "documentDB",
  "databaseName": "MyDatabase",
  "collectionName": "MyCollection",
  "createIfNotExists": true,
  "connection": "MyAccount_DOCUMENTDB",
  "direction": "out"
}
```

And you have a queue input binding for a queue that receives JSON in the following format:

```
{
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

And you want to create DocumentDB documents in the following format for each record:

```
{
  "id": "John Henry-123456",
  "name": "John Henry",
  "employeeId": "123456",
  "address": "A town nearby"
}
```

See the language-specific sample that uses this output binding to add documents to your database.

- C#
- F#
- Node.js

Output sample in C#

```
#r "Newtonsoft.Json"

using System;
using Newtonsoft.Json;
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
  };
}
```

Output sample in F#

```
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 }
```

You need to add 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 sample in Node.js

```
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();
};
```

Azure Functions Event Hub bindings

1/20/2017 • 4 min to read • Edit on GitHub

This article explains how to configure and code Azure Event Hub 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 Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

If you are new to Azure Event Hubs, see the Azure Event Hub overview.

Event Hub trigger

Use the Event Hub 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.

The Event Hub trigger to a function uses the following JSON object in the `bindings` array of function.json:

```
{
    "type": "eventHubTrigger",
    "name": "<Name of trigger parameter in function signature>",
    "direction": "in",
    "path": "<Name of the Event Hub>",
    "consumerGroup": "Consumer group to use - see below",
    "connection": "<Name of app setting with connection string - see below>"
}
```

`consumerGroup` is an optional property used to set the consumer group used to subscribe to events in the hub. If omitted, the `$Default` consumer group is used.
`connection` must be the name of an app setting that contains the connection string to the event hub's namespace. Copy this connection string by clicking the **Connection Information** button for the *namespace*, not the event hub itself. This connection string must have at least read permissions to activate the trigger.

Additional settings can be provided in a host.json file to further fine tune Event Hub triggers.

Trigger usage

When an Event Hub trigger function is triggered, the message that triggers it is passed into the function as a string.

Trigger sample

Suppose you have the following Event Hub trigger in the `bindings` array of function.json:

```
{
  "type": "eventHubTrigger",
  "name": "myEventHubMessage",
  "direction": "in",
  "path": "MyEventHub",
  "connection": "myEventHubReadConnectionString"
}
```

See the language-specific sample that logs the message body of the event hub trigger.

- C#
- F#
- Node.js

Trigger sample in C#

```
using System;

public static void Run(string myEventHubMessage, TraceWriter log)
{
    log.Info($"C# Event Hub trigger function processed a message: {myEventHubMessage}");
}
```

Trigger sample in F#

```
let Run(myEventHubMessage: string, log: TraceWriter) =
    log.Info(sprintf "F# eventhub trigger function processed work item: %s" myEventHubMessage)
```

Trigger sample in Node.js

```
module.exports = function (context, myEventHubMessage) {
    context.log('Node.js eventhub trigger function processed work item', myEventHubMessage);
    context.done();
};
```

Event Hub output binding

Use the Event Hub output binding to write events to an event hub event stream. You must have send permission to an event hub to write events to it.

The output binding uses the following JSON object in the `bindings` array of function.json:

```
{
    "type": "eventHub",
    "name": "<Name of output parameter in function signature>",
    "path": "<Name of event hub>",
    "connection": "<Name of app setting with connection string - see below>"
    "direction": "out"
}
```

`connection` must be the name of an app setting that contains the connection string to the event hub's namespace. Copy this connection string by clicking the **Connection Information** button for the *namespace*, not the event hub itself. This connection string must have send permissions to send the message to the event stream.

Output usage

This section shows you how to use your Event Hub output binding in your function code.

You can output messages to the configured event hub with the following parameter types:

- `out string`
- `ICollector<string>` (to output multiple messages)
- `IAsyncCollector<string>` (async version of `ICollector<T>`)

Output sample

Suppose you have the following Event Hub output binding in the `bindings` array of function.json:

```
{
    "type": "eventHub",
    "name": "outputEventHubMessage",
    "path": "myeventhub",
    "connection": "MyEventHubSend",
    "direction": "out"
}
```

See the language-specific sample that writes an event to the even stream.

- C#
- F#
- Node.js

Output sample in C#

```
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;
}
```

Or, to create 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 sample in F#

```
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 sample for Node.js

```
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();
};
```

Or, to send 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();
};
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions HTTP and webhook bindings

1/17/2017 • 11 min to read • Edit on GitHub

This article explains how to configure and work with HTTP triggers and bindings in Azure Functions. With these, you can use Azure Functions to build serverless APIs and respond to webhooks.

Azure Functions provides the following bindings:

- An HTTP trigger lets you invoke a function with an HTTP request. This can be customized to respond to webhooks.
- An HTTP output binding allows you to respond to the request.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

HTTP trigger

The HTTP trigger will execute your function in response to an HTTP request. You can customize it to respond to a particular URL or set of HTTP methods. An HTTP trigger can also be configured to respond to webhooks.

If using the Functions portal, you can also get started right away using a pre-made template. Select **New function** and choose "API & Webhooks" from the **Scenario** dropdown. Select one of the templates and click **Create**.

By default, an HTTP trigger will respond to the request with an HTTP 200 OK status code and an empty body. To modify the response, configure an HTTP output binding

Configuring an HTTP trigger

An HTTP trigger is defined by including a JSON object similar to the following in the `bindings` array of function.json:

```
{
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
    "authLevel": "function",
    "methods": [ "GET" ],
    "route": "values/{id}"
},
```

The binding supports the following properties:

- **name** : Required - the variable name used in function code for the request or request body. See Working with an HTTP trigger from code.
- **type** : Required - must be set to "httpTrigger".
- **direction** : Required - must be set to "in".
- *authLevel* : This determines what keys, if any, need to be present on the request in order to invoke the function. See Working with keys below. The value can be one of the following:

    - *anonymous*: No API key is required.
    - *function*: A function-specific API key is required. This is the default value if none is provided.
    - *admin* : The master key is required.
- **methods** : This is an array of the HTTP methods to which the function will respond. If not specified, the function will respond to all HTTP methods. See Customizing the HTTP endpoint.
- **route** : This defines the route template, controlling to which request URLs your function will respond. The default value if none is provided is `<functionname>`. See Customizing the HTTP endpoint.
- **webHookType** : This configures the HTTP trigger to act as a webhook reciever for the specified provider. The *methods* property should not be set if this is chosen. See Responding to webhooks. The value can be one of the following:
    - *genericJson* : A general purpose webhook endpoint without logic for a specific provider.
    - *github* : The function will respond to GitHub webhooks. The *authLevel* property should not be set if this is chosen.
    - *slack* : The function will respond to Slack webhooks. The *authLevel* property should not be set if this is chosen.

Working with an HTTP trigger from code

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`, then you will get full access to the request object. For a custom type (such as a POCO), Functions will attempt to parse the request body as JSON to populate the object properties.

For Node.js functions, the Functions runtime provides the request body instead of the request object.

See HTTP trigger samples for example usages.

HTTP response output binding

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 will return HTTP 200 OK with an empty body.

Configuring an HTTP output binding

The HTTP output binding is defined by including a JSON object similar to the following in the `bindings` array of function.json:

```
{
    "name": "res",
    "type": "http",
    "direction": "out"
}
```

The binding contains the following properties:

- **name** : Required - the variable name used in function code for the response. See Working with an HTTP output binding from code.
- **type** : Required - must be set to "http".
- **direction** : Required - must be set to "out".

Working with an HTTP output binding from code

You can use the output parameter (e.g., "res") to respond to the http or webhook caller. Alternatively, you can use the standard `Request.CreateResponse()` (C#) or `context.res` (Node.JS) pattern to return your response. For examples on how to use the latter method, see HTTP trigger samples and Webhook trigger samples.

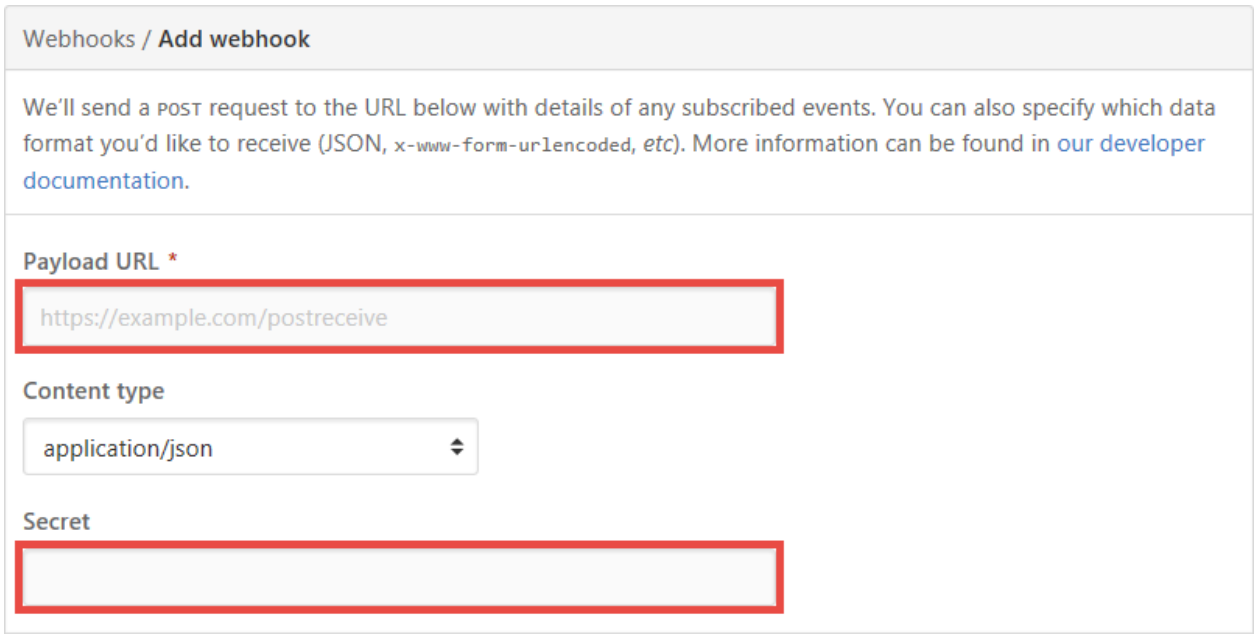
Responding to webhooks

An HTTP trigger with the *webHookType* property will be configured to respond to webhooks. The basic configuration uses the "genericJson" setting. This restricts requests to only those using HTTP POST and with the `application/json` content type.

The trigger can additionally be tailored to a specific webhook provider (e.g., GitHub and Slack). If a provider is specified, the Functions runtime can take care of the provider's validation logic for you.

Configuring Github as a webhook provider

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 your GitHub repository's **Add webhook** page. See GitHub's Creating Webhooks documentation for more.



Configuring Slack as a webhook provider

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 Working with keys.

Customizing 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 request, 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 to use 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": ""
  }
}
```

For detailed information on how to update the *host.json* file for your function, See, How to update function app files.

For information on other properties you can configure in your *host.json* file, see host.json reference.

Working with keys

HttpTriggers can leverage keys for added security. A standard HttpTrigger 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 within 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. The **master key** is a default host key named "_master" that is defined for each function app and cannot be revoked. It provides administrative access to the runtime APIs. Using `"authLevel": "admin"` in the binding JSON will require this key to be presented on the request; any other key will result in a authorization failure.

> **NOTE**
> 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. Exercise caution when choosing the admin authorization level.

API key authorization

By default, an HttpTrigger requires an API key in the HTTP request. So your HTTP request normally looks like this:

```
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 choose to allow requests without keys or specify that the master key must be used by changing the `authLevel` property in the binding JSON (see HTTP trigger).

Keys and webhooks

Webhook authorization is handled by the webhook reciever component, part of the HttpTrigger, and the mechanism varies based on the webhook type. Each mechanism does, however rely on a key. By default, the function key named "default" will be used. If you wish to use a different key, you will need to 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 (e.g., `https://<yourapp>.azurewebsites.net/api/<funcname>?clientid=<keyname>`).
- **Request header**: The provider passes the key name in the `x-functions-clientid` header.

> **NOTE**
> Function keys take precedence over host keys. If two keys are defined with the same name, the function key will be used.

HTTP trigger samples

Suppose you have the following HTTP trigger in the `bindings` array of function.json:

```
{
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
    "authLevel": "function"
},
```

See the language-specific sample that looks for a `name` parameter either in the query string or the body of the HTTP request.

- C#
- F#
- Node.js

HTTP trigger sample in C#

```
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);
}
```

HTTP trigger sample in F#

```
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, like this:

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "Dynamitey": "1.0.2",
        "FSharp.Interop.Dynamic": "3.0.0"
      }
    }
  }
}
```

This will use NuGet to fetch your dependencies and will reference them in your script.

HTTP trigger sample in Node.JS

```
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();
};
```

Webhook samples

Suppose you have the following webhook trigger in the `bindings` array of function.json:

```
{
    "webHookType": "github",
    "name": "req",
    "type": "httpTrigger",
    "direction": "in",
},
```

See the language-specific sample that logs GitHub issue comments.

- C#
- F#
- Node.js

Webhook sample in C#

```
#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 sample in F#

```
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 sample in Node.JS

```
module.exports = function (context, data) {
    context.log('GitHub WebHook triggered!', data.comment.body);
    context.res = { body: 'New GitHub comment: ' + data.comment.body };
    context.done();
};
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Mobile Apps bindings

1/17/2017 • 6 min to read • Edit on GitHub

This article explains how to configure and code Azure Mobile Apps bindings in Azure Functions. Azure Functions supports input and output bindings for Mobile Apps.

The Mobile Apps input and output bindings let you read from and write to data tables in your mobile app.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Mobile Apps input binding

The Mobile Apps input binding loads a record from a mobile table endpoint and passes it into your function. In a C# and F# functions, any changes made to the record are automatically sent back to the table when the function exits successfully.

The Mobile Apps input to a function uses the following JSON object in the `bindings` array of function.json:

```
{
    "name": "<Name of input parameter in function signature>",
    "type": "mobileTable",
    "tableName": "<Name of your mobile app's data table>",
    "id" : "<Id of the record to retrieve - see below>",
    "connection": "<Name of app setting that has your mobile app's URL - see below>",
    "apiKey": "<Name of app setting that has your mobile app's API key - see below>",
    "direction": "in"
}
```

Note the following:

- `id` can be static, or it can be based on the trigger that invokes the function. For example, if you use a [queue trigger]() for your function, then `"id": "{queueTrigger}"` uses the string value of the queue message as the record ID to retrieve.
- `connection` should contain the name of an app setting in your function app, which in turn contains the URL of your mobile app. The function uses this URL to construct the required REST operations against your mobile app. You [create an app setting in your function app]() that contains your mobile app's URL (which looks like `http://<appname>.azurewebsites.net`), then specify the name of the app setting in the `connection` property in your input binding.
- You need to specify `apiKey` if you implement an API key in your Node.js mobile app backend, or implement an API key in your .NET mobile app backend. To do this, you [create an app setting in your function app]() that contains the API key, then add the `apiKey` property in your input binding with the name of the app setting.

> **IMPORTANT**
> This API key must not be shared with your mobile app clients. It should only be distributed securely to service-side clients, like Azure Functions.

> **NOTE**
> 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

This section shows you how to use your Mobile Apps input binding in your function code.

When the record with the specified table and record ID is found, it is passed into the named JObject parameter (or, in Node.js, it is passed into the `context.bindings.<name>` object). When the record is not found, the parameter is `null`.

In C# and F# functions, any changes you make to the input record (input parameter) is automatically sent back to the Mobile Apps table when the function exits successfully. In Node.js functions, use `context.bindings.<name>` to access the input record. You cannot modify a record in Node.js.

Input sample

Suppose you have the following function.json, that retrieves a Mobile App table record with the id of the queue trigger message:

```
{
"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
}
```

See the language-specific sample that uses the input record from the binding. The C# and F# samples also modify the record's `text` property.

- C#
- Node.js

Input sample in C#

```
#r "Newtonsoft.Json"
using Newtonsoft.Json.Linq;

public static void Run(string myQueueItem, JObject record)
{
    if (record != null)
    {
        record["Text"] = "This has changed.";
    }
}
```

Input sample in Node.js

```
module.exports = function (context, myQueueItem) {
    context.log(context.bindings.record);
    context.done();
};
```

Mobile Apps output binding

Use the Mobile Apps output binding to write a new record to a Mobile Apps table endpoint.

The Mobile Apps output for a function uses the following JSON object in the `bindings` array of function.json:

```
{
    "name": "<Name of output parameter in function signature>",
    "type": "mobileTable",
    "tableName": "<Name of your mobile app's data table>",
    "connection": "<Name of app setting that has your mobile app's URL - see below>",
    "apiKey": "<Name of app setting that has your mobile app's API key - see below>",
    "direction": "out"
}
```

Note the following:

- `connection` should contain the name of an app setting in your function app, which in turn contains the URL of your mobile app. The function uses this URL to construct the required REST operations against your mobile app. You [create an app setting in your function app]() that contains your mobile app's URL (which looks like `http://<appname>.azurewebsites.net`), then specify the name of the app setting in the `connection` property in your input binding.
- You need to specify `apiKey` if you implement an API key in your Node.js mobile app backend, or implement an API key in your .NET mobile app backend. To do this, you [create an app setting in your function app]() that contains the API key, then add the `apiKey` property in your input binding with the name of the app setting.

  > **IMPORTANT**
  > This API key must not be shared with your mobile app clients. It should only be distributed securely to service-side clients, like Azure Functions.

  > **NOTE**
  > 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

This section shows you how to use your Mobile Apps output binding in your function code.

In C# functions, use a named output parameter of type `out object` to access the output record. In Node.js functions, use `context.bindings.<name>` to access the output record.

Output sample

Suppose you have the following function.json, that defines a queue trigger and a Mobile Apps output:

```
{
"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
}
```

See the language-specific sample that creates a record in the Mobile Apps table endpoint with the content of the queue message.

- C#
- Node.js

Output sample in C#

```
public static void Run(string myQueueItem, out object record)
{
    record = new {
        Text = $"I'm running in a C# function! {myQueueItem}"
    };
}
```

Output sample in Node.js

```
module.exports = function (context, myQueueItem) {

    context.bindings.record = {
        text : "I'm running in a Node function! Data: '" + myQueueItem + "'"
    }

    context.done();
};
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Notification Hub output binding

1/17/2017 • 8 min to read • Edit on GitHub

This article explains how to configure and code Azure Notification Hub bindings in Azure Functions.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Your functions can send push notifications using a configured Azure Notification Hub with a few lines of code. However, the Azure Notification Hub must be configured for the Platform Notifications Services (PNS) you want to use. For more information on configuring an Azure Notification Hub and developing a client applications that register to receive notifications, see Getting started with Notification Hubs and click your target client platform at the top.

The notifications you send can be native notifications or template notifications. Native notifications target a specific notification platform as configured in the `platform` property of the output binding. A template notification can be used to target multiple platforms.

Notification hub output binding properties

The function.json file provides the following properties:

- `name` : Variable name used in function code for the notification hub message.
- `type` : must be set to "*notificationHub*".
- `tagExpression` : Tag expressions allow you to specify that notifications be delivered to a set of devices who have registered to receive notifications that match the tag expression. For more information, see Routing and tag expressions.
- `hubName` : Name of the notification hub resource in the Azure portal.
- `connection` : This connection string must be an **Application Setting** connection string set to the *DefaultFullSharedAccessSignature* value for your notification hub.
- `direction` : must be set to "*out*".
- `platform` : The platform property indicates the notification platform your notification targets. Must be one of the following values:
  - `template` : This is the default platform if the platform property is omitted from the output binding. Template notifications can be used to target any platform configured on the Azure Notification Hub. For more information on using templates in general to send cross platform notifications with an Azure Notification Hub, see Templates.
  - `apns` : Apple Push Notification Service. For more information on configuring the notification hub for APNS and receiving the notification in a client app, see Sending push notifications to iOS with Azure Notification Hubs
  - `adm` : Amazon Device Messaging. For more information on configuring the notification hub for ADM and receiving the notification in a Kindle app, see Getting Started with Notification Hubs for Kindle apps
  - `gcm` : Google Cloud Messaging. Firebase Cloud Messaging, which is the new version of GCM, is also supported. For more information on configuring the notification hub for GCM/FCM and receiving the

notification in an Android client app, see Sending push notifications to Android with Azure Notification Hubs

- `wns` : Windows Push Notification Services targeting Windows platforms. Windows Phone 8.1 and later is also supported by WNS. For more information on configuring the notification hub for WNS and receiving the notification in a Universal Windows Platform (UWP) app, see Getting started with Notification Hubs for Windows Universal Platform Apps
- `mpns` : Microsoft Push Notification Service. This platform supports Windows Phone 8 and earlier Windows Phone platforms. For more information on configuring the notification hub for MPNS and receiving the notification in a Windows Phone app, see Sending push notifications with Azure Notification Hubs on Windows Phone

Example function.json:

```
{
  "bindings": [
    {
      "name": "notification",
      "type": "notificationHub",
      "tagExpression": "",
      "hubName": "my-notification-hub",
      "connection": "MyHubConnectionString",
      "platform": "gcm",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Notification hub connection string setup

To use a Notification hub output binding, you must configure the connection string for the hub. This can be done on the *Integrate* tab by selecting your notification hub or creating a new one.

You can also manually add a connection string for an existing hub by adding a connection string for the *DefaultFullSharedAccessSignature* to your notification hub. This connection string provides your function access permission to send notification messages. The *DefaultFullSharedAccessSignature* connection string value can be accessed from the **keys** button in the main blade of your notification hub resource in the Azure portal. To manually add a connection string for your hub, use the following steps:

1. On the **Function app** blade of the Azure portal, click **Function App Settings > Go to App Service settings**.
2. In the **Settings** blade, click **Application Settings**.
3. Scroll down to the **Connection strings** section, and add a named entry for *DefaultFullSharedAccessSignature* value for your notification hub. Change the type to **Custom**.
4. Reference your connection string name in the output bindings. Similar to **MyHubConnectionString** used in the example above.

APNS native notifications with C# queue triggers

This example shows how to use types defined in the Microsoft Azure Notification Hubs Library 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));
}
```

GCM native notifications with C# queue triggers

This example shows how to use types defined in the Microsoft Azure Notification Hubs Library 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));
}
```

WNS native notifications with C# queue triggers

This 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));
}
```

Template example for Node.js timer triggers

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();
};
```

Template example for F# timer triggers

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#!")]
```

Template example using an out parameter

This example sends a notification for a template registration that contains a `message` place holder 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;
}
```

Template example with asynchronous function

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;
}
```

Template example using JSON

This example sends a notification for a template registration that contains a `message` place holder 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!\"}";
}
```

Template example using Notification Hubs 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);
}
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Service Bus bindings

1/17/2017 • 6 min to read • Edit on GitHub

This article explains how to configure and code Azure Service Bus bindings in Azure Functions. Azure Functions supports trigger and output bindings for Notification Hubs 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 Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Service Bus trigger

Use the Service Bus trigger to respond to messages from a Service Bus queue or topic.

The Notification Hubs queue and topic triggers to a function use the following JSON objects in the `bindings` array of function.json:

- *queue* trigger:

```
{
    "name" : "<Name of input parameter in function signature>",
    "queueName" : "<Name of the queue>",
    "connection" : "<Name of app setting that has your queue's connection string - see below>",
    "accessRights" : "<Access rights for the connection string - see below>",
    "type" : "serviceBusTrigger",
    "direction" : "in"
}
```

- *topic* trigger:

```
{
    "name" : "<Name of input parameter in function signature>",
    "topicName" : "<Name of the topic>",
    "subscriptionName" : "<Name of the subscription>",
    "connection" : "<Name of app setting that has your topic's connection string - see below>",
    "accessRights" : "<Access rights for the connection string - see below>",
    "type" : "serviceBusTrigger",
    "direction" : "in"
}
```

Note the following:

- For `connection`, [create an app setting in your function app]() that contains the connection string to your Service Hub namespace, then specify the name of the app setting in the `connection` property in your trigger. You obtain the connection string by following the steps shown at Obtain the management credentials. The connection string must be for a Service Bus namespace, not limited to a specific queue or topic. If you leave `connection` empty, the trigger assumes that a default Service Bus connection string is specified in an app setting named `AzureWebJobsServiceBus`.
- For `accessRights`, available values are `manage` and `listen`. The default is `manage`, which indicates that the `connection` has the **Manage** permission. If you use a connection string that does not have the **Manage**

permission, set `accessRights` to `listen`. Otherwise, the Functions runtime might try and fail to do operations that require manage rights.

Trigger behavior

- **Single-threading** - 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 `serviceBus.maxConcurrrentCalls` to 1 in *host.json*. For information about *host.json*, see Folder Structure and host.json.
- **Poison message handling** - Service Bus does its own poison message handling, which can't be controlled or configured in Azure Functions configuration or code.
- **PeekLock behavior** - The Functions runtime receives a message in `PeekLock` mode and 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 usage

This section shows you how to use your Service Hub trigger in your function code.

In C# and F#, the Service Bus trigger message can be deserialized to any of the following input types:

- `string` - useful for string messages
- `byte[]` - useful for binary data
- Any Object - useful for JSON-serialized data. If you declare a custom input type (e.g. `FooType`), Azure Functions attempts to deserialize the JSON data into your specified type.
- `BrokeredMessage` - gives you the deserialized message with the BrokeredMessage.GetBody() method.

In Node.js, the Service Bus trigger message is passed into the function as either a string or, in the case of JSON message, a JavaScript object.

Trigger sample

Suppose you have the following function.json:

```
{
"bindings": [
    {
    "queueName": "testqueue",
    "connection": "MyServiceBusConnection",
    "name": "myQueueItem",
    "type": "serviceBusTrigger",
    "direction": "in"
    }
],
"disabled": false
}
```

See the language-specific sample that processes a Service Bus queue message.

- C#
- F#
- Node.js

Trigger sample in C#

```
public static void Run(string myQueueItem, TraceWriter log)
{
    log.Info($"C# ServiceBus queue trigger function processed message: {myQueueItem}");
}
```

Trigger sample in F#

```
let Run(myQueueItem: string, log: TraceWriter) =
    log.Info(sprintf "F# ServiceBus queue trigger function processed message: %s" myQueueItem)
```

Trigger sample in Node.js

```
module.exports = function(context, myQueueItem) {
    context.log('Node.js ServiceBus queue trigger function processed message', myQueueItem);
    context.done();
};
```

Service Bus output binding

The Notification Hubs queue and topic output for a function use the following JSON objects in the `bindings` array of function.json:

- *queue* output:

  ```
  {
      "name" : "<Name of output parameter in function signature>",
      "queueName" : "<Name of the queue>",
      "connection" : "<Name of app setting that has your queue's connection string - see below>",
      "accessRights" : "<Access rights for the connection string - see below>"
      "type" : "serviceBus",
      "direction" : "out"
  }
  ```

- *topic* output:

  ```
  {
      "name" : "<Name of output parameter in function signature>",
      "topicName" : "<Name of the topic>",
      "subscriptionName" : "<Name of the subscription>",
      "connection" : "<Name of app setting that has your topic's connection string - see below>",
      "accessRights" : "<Access rights for the connection string - see below>"
      "type" : "serviceBus",
      "direction" : "out"
  }
  ```

Note the following:

- For `connection`, [create an app setting in your function app]() that contains the connection string to your Service Hub namespace, then specify the name of the app setting in the `connection` property in your output binding. You obtain the connection string by following the steps shown at Obtain the management credentials. The connection string must be for a Service Bus namespace, not limited to a specific queue or topic. If you leave `connection` empty, the output binding assumes that a default Service Bus connection string is specified in an app setting named `AzureWebJobsServiceBus`.
- For `accessRights`, available values are `manage` and `listen`. The default is `manage`, which indicates that the `connection` has the **Manage** permission. If you use a connection string that does not have the **Manage**

permission, set `accessRights` to `listen`. Otherwise, the Functions runtime might try and fail to do operations that require manage rights.

Output usage

In C# and F#, Azure Functions can create a Service Bus queue message from any of the following types:

- Any Object - Parameter definition looks like `out T paramName` (C#). Functions deserializes the object into a JSON message. If the output value is null when the function exits, Functions creates the message with a null object.
- `string` - Parameter definition looks like `out string paraName` (C#). If the parameter value is non-null when the function exits, Functions creates a message.
- `byte[]` - Parameter definition looks like `out byte[] paraName` (C#). If the parameter value is non-null when the function exits, Functions creates a message.
- `BrokeredMessage` Parameter definition looks like `out BrokeredMessage paraName` (C#). If the parameter value is non-null when the function exits, Functions creates a message.

For creating multiple messages in a C# function, you can use `ICollector<T>` or `IAsyncCollector<T>`. A message is created when you call the `Add` method.

In Node.js, you can assign a string, a byte array, or a Javascript object (deserialized into JSON) to `context.binding.<paramName>`.

Output sample

Suppose you have the following function.json, that defines a Service Bus queue output:

```
{
    "bindings": [
        {
            "schedule": "0/15 * * * * *",
            "name": "myTimer",
            "runsOnStartup": true,
            "type": "timerTrigger",
            "direction": "in"
        },
        {
            "name": "outputSbQueue",
            "type": "serviceBus",
            "queueName": "testqueue",
            "connection": "MyServiceBusConnection",
            "direction": "out"
        }
    ],
    "disabled": false
}
```

See the language-specific sample that sends a message to the service bus queue.

- C#
- F#
- Node.js

Output sample in C#

```
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;
}
```

Or, to create multiple messages:

```
public static void Run(TimerInfo myTimer, TraceWriter log, ICollector<string> outputSbQueue)
{
    string message = $"Service Bus queue message created at: {DateTime.Now}";
    log.Info(message);
    outputSbQueue.Add("1 " + message);
    outputSbQueue.Add("2 " + message);
}
```

Output sample in F#

```
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 sample in Node.js

```
module.exports = function (context, myTimer) {
    var message = 'Service Bus queue message created at ' + timeStamp;
    context.log(message);
    context.bindings.outputSbQueueMsg = message;
    context.done();
};
```

Or, to create 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();
};
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Storage blob bindings

2/7/2017 • 9 min to read • Edit on GitHub

This article explains how to configure and code Azure Storage blob bindings in Azure Functions. Azure Functions supports trigger, input, and output bindings for Azure Storage blobs.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

> **NOTE**
> A blob only storage account is not supported. Azure Functions requires a general-purpose storage account to use with blobs.

Storage blob trigger

The Azure Storage blob trigger lets you monitor a storage container for new and updated blobs and run your function code when changes are detected.

The Storage blob trigger to a function uses the following JSON objects in the `bindings` array of function.json:

```
{
    "name": "<Name of input parameter in function signature>",
    "type": "blobTrigger",
    "direction": "in",
    "path": "<container to monitor, and optionally a blob name pattern - see below>",
    "connection":"<Name of app setting - see below>"
}
```

Note the following:

- For `path`, see Name patterns to find out how to format blob name patterns.
- `connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, the standard editor in the **Integrate** tab configures this app setting for you when you create a storage account or selects an existing one. To manually create this app setting, see [configure this app setting manually]().

Also, see one of the following subheadings for more information:

- Name patterns
- Blob receipts
- Handling poison blobs

Name patterns

You can specify a blob name pattern in the `path` property. For example:

```
"path": "input/original-{name}",
```

This path would find a blob named *original-Blob1.txt* in the *input* container, and the value of the `name` variable in function code would be `Blob1`.

Another example:

```
"path": "input/{blobname}.{blobextension}",
```

This path would also find a blob named *original-Blob1.txt*, and the value of the `blobname` and `blobextension` variables in function code would be *original-Blob1* and *txt*.

You can restrict the file type of blobs by using a fixed value for the file extension. For example:

```
"path": "samples/{name}.png",
```

In this case, only *.png* blobs in the *samples* container trigger the function.

Curly braces are special characters in name patterns. To specify blob names that have curly braces in the name, double the curly braces. For example:

```
"path": "images/{{20140101}}-{name}",
```

This path would find a blob named *{20140101}-soundfile.mp3* in the *images* container, and the `name` variable value in the function code would be *soundfile.mp3*.

Blob receipts

The Azure Functions runtime makes sure that no blob trigger function gets called more than once for the same new or updated blob. It does so by maintaining *blob receipts* to determine if a given blob version has been processed.

Blob receipts are stored in a container named *azure-webjobs-hosts* in the Azure storage account for your function app (specified by the `AzureWebJobsStorage` app setting). A blob receipt has the following information:

- The triggered function ("<*function app name*>.Functions.<*function name*>", for example: "functionsf74b96f7.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.

Handling poison blobs

When a blob trigger function fails, Azure Functions retries that function up to 5 times by default (including the first try) for a given blob. If all 5 tries fail, 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")

Blob polling for large containers

If the blob container watched by the binding contains more than 10,000 blobs, the Functions runtime scans log files to watch for new or changed blobs. This process is not real time. 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 efforts" basis. There is no guarantee that all events are captured. Under some conditions, logs may be missed. If the speed and reliability limitations of blob triggers for large containers are not acceptable for your application, the recommended method is to create a queue message when you create the blob, and use a queue trigger instead of a blob trigger to process the blob.

Trigger usage

In C# functions, you bind to the input blob 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 blob data using `context.bindings.<name>` .

The blob can be deserialized into any of the following types:

- Any Object - useful for JSON-serialized blob 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 blob data.

In C# functions, you can also bind to any of the following types, and the Functions runtime will attempt to deserialize the blob data using that type:

- `TextReader`
- `Stream`
- `ICloudBlob`
- `CloudBlockBlob`
- `CloudPageBlob`
- `CloudBlobContainer`
- `CloudBlobDirectory`
- `IEnumerable<CloudBlockBlob>`
- `IEnumerable<CloudPageBlob>`
- Other types deserialized by ICloudBlobStreamBinder

Trigger sample

Suppose you have the following function.json, that defines a Storage blob trigger:

```
{
    "disabled": false,
    "bindings": [
        {
            "name": "myBlob",
            "type": "blobTrigger",
            "direction": "in",
            "path": "samples-workitems",
            "connection":""
        }
    ]
}
```

See the language-specific sample that logs the contents of each blob that is added to the monitored container.

- C#
- Node.js

Trigger usage in C#

```
public static void Run(string myBlob, TraceWriter log)
{
    log.Info($"C# Blob trigger function processed: {myBlob}");
}
```

Trigger usage in Node.js

```
module.exports = function(context) {
    context.log('Node.js Blob trigger function processed', context.bindings.myBlob);
    context.done();
};
```

Storage Blob input binding

The Azure Storage blob input binding enables you to use a blob from a storage container in your function.

The Storage blob input to a function uses the following JSON objects in the `bindings` array of function.json:

```
{
  "name": "<Name of input parameter in function signature>",
  "type": "blob",
  "direction": "in"
  "path": "<Path of input blob - see below>",
  "connection":"<Name of app setting - see below>"
},
```

Note the following:

- `path` must contain the container name and the blob name. For example, if you have a queue trigger in your function, you can use `"path": "samples-workitems/{queueTrigger}"` to point to a blob in the `samples-workitems` container with a name that matches the blob name specified in the trigger message.
- `connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, the standard editor in the **Integrate** tab configures this app setting for you when you create a Storage account or selects an existing one. To manually create this app setting, see [configure this app setting manually]().

Input usage

In C# functions, you bind to the input blob 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 input binding. In Node.js functions, you access the input blob data using `context.bindings.<name>`.

The blob can be deserialized into any of the following types:

- Any Object - useful for JSON-serialized blob 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 blob data.

In C# functions, you can also bind to any of the following types, and the Functions runtime will attempt to deserialize the blob data using that type:

- `TextReader`
- `Stream`
- `ICloudBlob`
- `CloudBlockBlob`
- `CloudPageBlob`

Input sample

Suppose you have the following function.json, that defines a Storage queue trigger, a Storage blob input, and a Storage blob output:

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnection",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "myInputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}",
      "connection": "MyStorageConnection",
      "direction": "in"
    },
    {
      "name": "myOutputBlob",
      "type": "blob",
      "path": "samples-workitems/{queueTrigger}-Copy",
      "connection": "MyStorageConnection",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

See the language-specific sample that copies the input blob to the output blob.

- C#
- Node.js

Input usage in C#

```
public static void Run(string myQueueItem, string myInputBlob, out string myOutputBlob, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    myOutputBlob = myInputBlob;
}
```

Input usage in Node.js

```
module.exports = function(context) {
    context.log('Node.js Queue trigger function processed', context.bindings.myQueueItem);
    context.bindings.myOutputBlob = context.bindings.myInputBlob;
    context.done();
};
```

Storage Blob output binding

The Azure Storage blob output binding enables you to write blobs to a Storage container in your function.

The Storage blob output for a function uses the following JSON objects in the `bindings` array of function.json:

```
{
  "name": "<Name of output parameter in function signature>",
  "type": "blob",
  "direction": "out",
  "path": "<Path of input blob - see below>",
  "connection": "<Name of app setting - see below>"
}
```

Note the following:

- `path` must contain the container name and the blob name to write to. For example, if you have a queue trigger in your function, you can use `"path": "samples-workitems/{queueTrigger}"` to point to a blob in the `samples-workitems` container with a name that matches the blob name specified in the trigger message.
- `connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, the standard editor in the **Integrate** tab configures this app setting for you when you create a storage account or selects an existing one. To manually create this app setting, see [configure this app setting manually]().

Output usage

In C# functions, you bind to the output blob 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 `paramName` is the name you specified in the output binding. In Node.js functions, you access the output blob using `context.bindings.<name>`.

You can write to the output blob using any of the following types:

- Any Object - useful for JSON-serialization. If you declare a custom output type (e.g. `out FooType paramName`), Azure Functions attempts to serialize object into JSON. If the output parameter is null when the function exits, the Functions runtime creates a blob as a null object.
- String - (`out string paramName`) useful for text blob data. the Functions runtime creates a blob 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`
- `CloudBlobStream`
- `ICloudBlob`
- `CloudBlockBlob`
- `CloudPageBlob`
- `ICollector<T>` (to output multiple blobs)
- `IAsyncCollector<T>` (async version of `ICollector<T>`)

Output sample

See input sample.

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Storage queue bindings

1/19/2017 • 5 min to read • Edit on GitHub

This article explains how to configure and code Azure Storage queue bindings in Azure Functions. Azure Functions supports trigger and output bindings for Azure Storage queues.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Storage Queue trigger

The Azure Storage queue trigger enables you to monitor a storage queue for new messages and react to them.

The Storage queue trigger to a function use the following JSON objects in the `bindings` array of function.json:

```
{
    "name": "<Name of input parameter in function signature>",
    "queueName": "<Name of queue to poll>",
    "connection":"<Name of app setting - see below>",
    "type": "queueTrigger",
    "direction": "in"
}
```

`connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, you can configure this app setting in the **Integrate** tab when you create a storage account or select an existing one. To manually create this app setting, see Manage App Service settings.

Additional settings can be provided in a host.json file to further fine-tune storage queue triggers.

Handling poison queue messages

When a queue trigger function fails, Azure Functions retries that function up to five times for a given queue message, including the first try. If all five attempts fail, Functions adds a message to a Storage 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, you can get the number of times a message has been picked up for processing by checking `dequeueCount` (see Queue trigger metadata).

Trigger usage

In C# functions, you bind to the input message 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 binding. In Node.js functions, you access the input blob data using `context.bindings.<name>`.

The queue message can be deserialized to any of the following types:

- Object - used for JSON serialized messages. When you declare a custom input type, the runtime tries to deserialize the JSON object.

- String
- Byte array
- CloudQueueMessage (C# only)

Queue trigger metadata

You can get queue metadata in your function by using these variable names:

- expirationTime
- insertionTime
- nextVisibleTime
- id
- popReceipt
- dequeueCount
- queueTrigger (another way to retrieve the queue message text as a string)

See how to use the queue metadata in Trigger sample

Trigger sample

Suppose you have the following function.json, that defines a Storage queue trigger:

```
{
    "disabled": false,
    "bindings": [
        {
            "name": "myQueueItem",
            "queueName": "myqueue-items",
            "connection":"",
            "type": "queueTrigger",
            "direction": "in"
        }
    ]
}
```

See the language-specific sample that retrieves and logs queue metadata.

- C#
- Node.js

Trigger sample in C#

```
public static void Run(string 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}\n" +
        $"queueTrigger={queueTrigger}\n" +
        $"expirationTime={expirationTime}\n" +
        $"insertionTime={insertionTime}\n" +
        $"nextVisibleTime={nextVisibleTime}\n" +
        $"id={id}\n" +
        $"popReceipt={popReceipt}\n" +
        $"dequeueCount={dequeueCount}");
}
```

Trigger sample in Node.js

```
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();
};
```

Storage Queue output binding

The Azure Storage queue output binding enables you to write messages to a Storage queue in your function.

The Storage queue output for a function uses the following JSON objects in the `bindings` array of function.json:

```
{
  "name": "<Name of output parameter in function signature>",
    "queueName": "<Name of queue to write to>",
    "connection":"<Name of app setting - see below>",
  "type": "queue",
  "direction": "out"
}
```

`connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, the standard editor in the **Integrate** tab configures this app setting for you when you create a storage account or selects an existing one. To manually create this app setting, see Manage App Service settings.

Output usage

In C# functions, you write a queue message by using the named `out` parameter in your function signature, like `out <T> <name>`. In this case, `T` is the data type that you want to serialize the message into, and `paramName` is the name you specified in the output binding. In Node.js functions, you access the output using `context.bindings.<name>`.

You can output a queue message using any of the data types in your code:

- Any Object: `out MyCustomType paramName`
  Used for JSON serialization. When you declare a custom output type, the runtime tries to serialize the object into JSON. If the output parameter is null when the function exits, the runtime creates a queue message as a null object.
- String: `out string paramName`
  Used for test messages. The runtime creates message only when the string parameter is non-null when the function exits.
- Byte array: `out byte[]`

These additional output types are supported a C# function:

- CloudQueueMessage: `out CloudQueueMessage`
- `ICollector<T>` or `IAsyncCollector<T>` where `T` is one of the supported types.

Output sample

Suppose you have the following function.json, that defines a Storage queue trigger, a Storage blob input, and a Storage blob output:

Example *function.json* for a storage queue output binding that uses a manual trigger and writes the input to a queue message:

```
{
  "bindings": [
    {
      "type": "manualTrigger",
      "direction": "in",
      "name": "input"
    },
    {
      "type": "queue",
      "name": "myQueueItem",
      "queueName": "myqueue",
      "connection": "my_storage_connection",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

See the language-specific sample that writes an output queue message for each input queue message.

- C#
- Node.js

Output sample in C#

```
public static void Run(string input, out string myQueueItem, TraceWriter log)
{
    myQueueItem = "New message: " + input;
}
```

Or, to send multiple messages,

```
public static void Run(string input, ICollector<string> myQueueItem, TraceWriter log)
{
    myQueueItem.Add("Message 1: " + input);
    myQueueItem.Add("Message 2: " + "Some other message.");
}
```

Output sample in Node.js

```
module.exports = function(context) {
    // Define a new message for the myQueueItem output binding.
    context.bindings.myQueueItem = "new message";
    context.done();
};
```

Or, to send multiple messages,

```
module.exports = function(context) {
    // Define a message array for the myQueueItem output binding.
    context.bindings.myQueueItem = ["message 1","message 2"];
    context.done();
};
```

Next steps

For an example of a function that uses aStorage queue triggers and bindings, see Create an Azure Function connected to an Azure service.

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Storage table bindings

1/17/2017 • 6 min to read • Edit on GitHub

This article explains how to configure and code Azure Storage table triggers and bindings in Azure Functions. Azure Functions supports input and output bindings for Azure Storage tables.

The Storage table binding supports the following scenarios:

- **Read a single row in a C# or Node.js function** - Set `partitionKey` and `rowKey`. The `filter` and `take` properties are not used in this scenario.
- **Read multiple rows in a C# function** - The Functions runtime provides an `IQueryable<T>` object bound to the table. Type `T` must derive from `TableEntity` or implement `ITableEntity`. The `partitionKey`, `rowKey`, `filter`, and `take` properties are not used in this scenario; you can use the `IQueryable` object to do any filtering required.
- **Read multiple rows in a Node function** - Set the `filter` and `take` properties. Don't set `partitionKey` or `rowKey`.
- **Write one or more rows in a C# function** - The Functions runtime provides an `ICollector<T>` or `IAsyncCollector<T>` bound to the table, where `T` specifies the schema of the entities you want to add. Typically, type `T` derives from `TableEntity` or implements `ITableEntity`, but it doesn't have to. The `partitionKey`, `rowKey`, `filter`, and `take` properties are not used in this scenario.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Storage table input binding

The Azure Storage table input binding enables you to use a storage table in your function.

The Storage table input to a function uses the following JSON objects in the `bindings` array of function.json:

```
{
    "name": "<Name of input parameter in function signature>",
    "type": "table",
    "direction": "in",
    "tableName": "<Name of Storage table>",
    "partitionKey": "<PartitionKey of table entity to read - see below>",
    "rowKey": "<RowKey of table entity to read - see below>",
    "take": "<Maximum number of entities to read in Node.js - optional>",
    "filter": "<OData filter expression for table input in Node.js - optional>",
    "connection": "<Name of app setting - see below>",
}
```

Note the following:

- Use `partitionKey` and `rowKey` together to read a single entity. These properties are optional.
- `connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, the standard editor in the **Integrate** tab configures this app setting for you when you create a Storage account or selects an existing one. You can also configure this app setting manually.

Input usage

In C# functions, you bind to the input table entity (or entities) 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 input binding. In Node.js functions, you access the input table entity (or entities) using `context.bindings.<name>`.

The input data can be deserialized in Node.js or C# functions. The deserialized objects have `RowKey` and `PartitionKey` properties.

In C# functions, you can also bind to any of the following types, and the Functions runtime will attempt to deserialize the table data using that type:

- Any type that implements `ITableEntity`
- `IQueryable<T>`

Input sample

Supposed you have the following function.json, which uses a queue trigger to read a single table row. The JSON specifies `PartitionKey` `RowKey`. `"rowKey": "{queueTrigger}"` indicates that the row key comes from the queue message string.

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnection",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "personEntity",
      "type": "table",
      "tableName": "Person",
      "partitionKey": "Test",
      "rowKey": "{queueTrigger}",
      "connection": "MyStorageConnection",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

See the language-specific sample that reads a single table entity.

- C#
- F#
- Node.js

Input sample in C#

```
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 sample in F#

```
[<CLIMutable>]
type Person = {
  PartitionKey: string
  RowKey: string
  Name: string
}

let Run(myQueueItem: string, personEntity: Person) =
    log.Info(sprintf "F# Queue trigger function processed: %s" myQueueItem)
    log.Info(sprintf "Name in Person entity: %s" personEntity.Name)
```

Input sample in Node.js

```
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();
};
```

Storage table output binding

The Azure Storage table output binding enables you to write entities to a Storage table in your function.

The Storage table output for a function uses the following JSON objects in the `bindings` array of function.json:

```
{
    "name": "<Name of input parameter in function signature>",
    "type": "table",
    "direction": "out",
    "tableName": "<Name of Storage table>",
    "partitionKey": "<PartitionKey of table entity to write - see below>",
    "rowKey": "<RowKey of table entity to write - see below>",
    "connection": "<Name of app setting - see below>",
}
```

Note the following:

- Use `partitionKey` and `rowKey` together to write a single entity. These properties are optional. You can also specify `PartitionKey` and `RowKey` when you create the entity objects in your function code.
- `connection` must contain the name of an app setting that contains a storage connection string. In the Azure portal, the standard editor in the **Integrate** tab configures this app setting for you when you create a Storage account or selects an existing one. You can also configure this app setting manually.

Output usage

In C# functions, you bind to the table output 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 `paramName` is the name you specified in the output binding. In Node.js functions, you access the table output using `context.bindings.<name>`.

You can serialize objects in Node.js or C# functions. In C# functions, you can also bind to the following types:

- Any type that implements `ITableEntity`
- `ICollector<T>` (to output multiple entities. See sample.)
- `IAsyncCollector<T>` (async version of `ICollector<T>`)
- `CloudTable` (using the Azure Storage SDK. See sample.)

Output sample

The following *function.json* and *run.csx* example shows how to write multiple table entities.

```
{
  "bindings": [
    {
      "name": "input",
      "type": "manualTrigger",
      "direction": "in"
    },
    {
      "tableName": "Person",
      "connection": "MyStorageConnection",
      "name": "tableBinding",
      "type": "table",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

See the language-specific sample that creates multiple table entities.

- C#
- F#
- Node.js

Output sample in C#

```
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 sample in F#

```
[<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 sample in Node.js

```
module.exports = function (context) {

    context.bindings.outputTable = [];

    for (i = 1; i < 10; i++) {
        context.bindings.outputTable.push({
            PartitionKey: "Test",
            RowKey: i.toString(),
            Name: "Name " + i
        });

    context.done();
};
```

Sample: Read multiple table entities in C#

The following *function.json* and C# code example reads entities for a partition key that is specified in the queue message.

```
{
  "bindings": [
    {
      "queueName": "myqueue-items",
      "connection": "MyStorageConnection",
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in"
    },
    {
      "name": "tableBinding",
      "type": "table",
      "connection": "MyStorageConnection",
      "tableName": "Person",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The C# 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; }
}
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions timer trigger

1/17/2017 • 3 min to read • Edit on GitHub

This article explains how to configure and code timer triggers in Azure Functions. Azure Functions supports the trigger for timers. Timer triggers call functions based on a schedule, one time or recurring.

The timer trigger supports multi-instance scale-out. One single instance of a particular timer function is run across all instances.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Timer trigger

The timer trigger to a function uses the following JSON object in the `bindings` array of function.json:

```
{
    "schedule": "<CRON expression - see below>",
    "name": "<Name of trigger parameter in function signature>",
    "type": "timerTrigger",
    "direction": "in"
}
```

The value of `schedule` is a CRON expression that includes 6 fields: `{second} {minute} {hour} {day} {month} {day of the week}`. Many of the cron expressions you find online omit the `{second}` field. If you copy from one of them, you need to adjust for the extra `{second}` field. For specific examples, see Schedule examples below.

The default time zone used with the CRON expressions is Coordinated Universal Time (UTC). If you want your CRON expression to be based on another time zone, create a new app setting for your function app named `WEBSITE_TIME_ZONE`. Set the value to the the name of the desired time zone as shown in the Microsoft Time Zone Index.

For example, *Eastern Standard Time* is UTC-05:00. If you want your timer trigger to fire at 10:00 AM EST every day, your could use the following CRON expression which 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 * * *",
```

Schedule examples

Here are some samples of CRON expressions you can use for the `schedule` property.

To trigger once every 5 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",
```

Trigger 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
}
```

Trigger sample

Suppose you have the following timer trigger in the `bindings` array of function.json:

```
{
    "schedule": "0 */5 * * * *",
    "name": "myTimer",
    "type": "timerTrigger",
    "direction": "in"
}
```

See the language-specific sample that reads the timer object to see whether it's running late.

- C#
- F#
- Node.js

Trigger sample in C#

```
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}" );
}
```

Trigger sample in F#

```
let Run(myTimer: TimerInfo, log: TraceWriter ) =
    if (myTimer.IsPastDue) then
        log.Info("F# function is running late.")
    let now = DateTime.Now.ToLongTimeString()
    log.Info(sprintf "F# function executed at %s!" now)
```

Trigger sample in Node.js

```
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();
};
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Azure Functions Twilio output binding

1/17/2017 • 4 min to read • Edit on GitHub

This article explains how to configure and use Twilio bindings with Azure Functions.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first Azure Function
- Azure Functions developer reference
- C#, F#, or Node developer reference

Azure Functions supports Twilio output bindings to enable your functions to send SMS text messages with a few lines of code and a Twilio account.

function.json for Azure Notification Hub output binding

The function.json file provides the following properties:

- `name` : Variable name used in function code for the Twilio SMS text message.
- `type` : must be set to "*twilioSms*".
- `accountSid` : This value must be set to the name of an App Setting that holds your Twilio Account Sid.
- `authToken` : This value must be set to the name of an App Setting that holds your Twilio authentication token.
- `to` : This value is set to the phone number that the SMS text is sent to.
- `from` : This value is set to the phone number that the SMS text is sent from.
- `direction` : must be set to "*out*".
- `body` : This value can be used to hard code the SMS text message if you don't need to set it dynamically in the code for your function.

Example function.json:

```
{
  "type": "twilioSms",
  "name": "message",
  "accountSid": "TwilioAccountSid",
  "authToken": "TwilioAuthToken",
  "to": "+1704XXXXXXX",
  "from": "+1425XXXXXXX",
  "direction": "out",
  "body": "Azure Functions Testing"
}
```

Example C# queue trigger with Twilio output binding

Synchronous

This synchronous example code for an Azure Storage queue trigger uses an out parameter to send a text message to a customer who placed an order.

```
#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;
}
```

Asynchronous

This asynchronous example code for an Azure Storage queue trigger sends a text message to a customer who placed an order.

```
#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);
}
```

Example Node.js queue trigger with Twilio output binding

This Node.js example sends a text message to a customer who placed an order.

```
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();
};
```

Next steps

For information about other bindings and triggers for Azure Functions, see Azure Functions triggers and bindings developer reference

Create a function from the Azure portal

2/6/2017 • 4 min to read • Edit on GitHub

Overview

Azure Functions is an event-driven, compute-on-demand experience that extends the existing Azure application platform with capabilities to implement code triggered by events occurring in other Azure services, SaaS products, and on-premises systems. With Azure Functions, your applications scale based on demand and you pay only for the resources you consume. Azure Functions enables you to create scheduled or triggered units of code implemented in various programming languages. To learn more about Azure Functions, see the Azure Functions Overview.

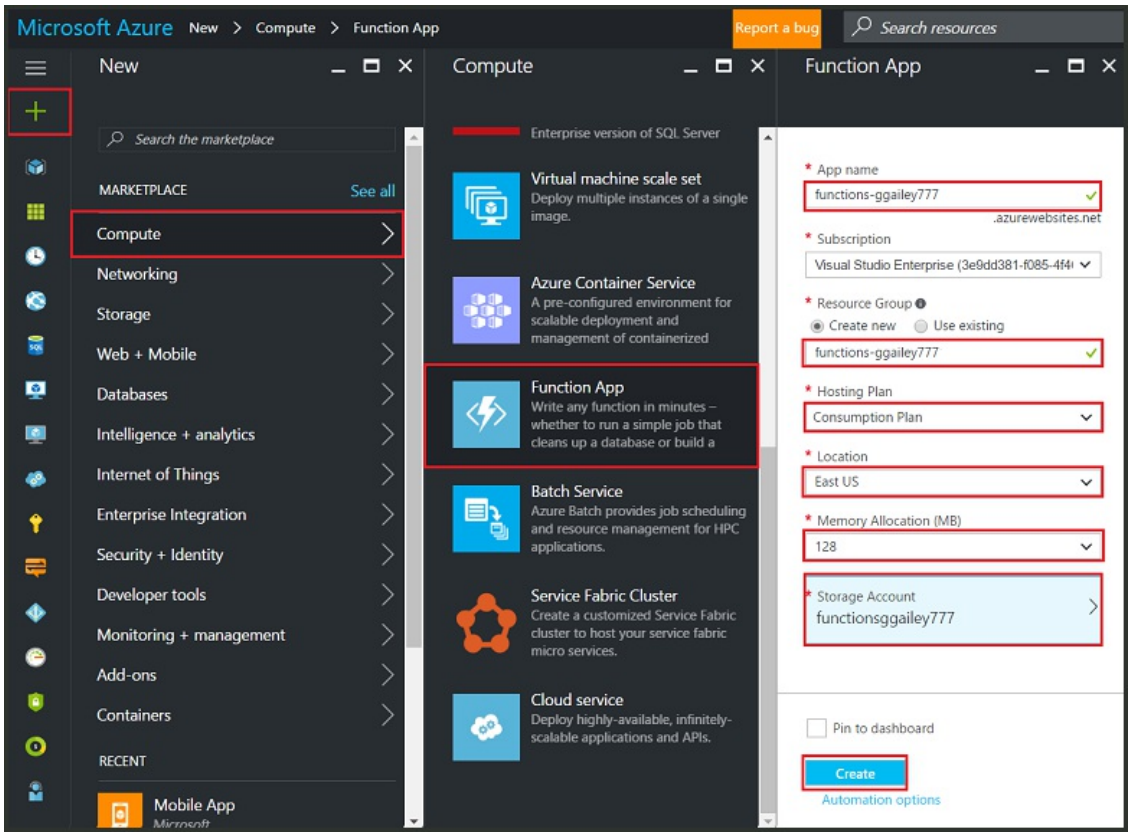
This topic shows you how to use the Azure portal to create a simple "hello world" Node.js Azure Function that is invoked by an HTTP-trigger. Before you can create a function in the Azure portal, you must explicitly create a function app in Azure App Service. To have the function app created for you automatically, see the other Azure Functions quickstart tutorial, which is a simpler quickstart experience and includes a video.

Create a function app

A function app hosts the execution of your functions in Azure. If you don't already have an Azure account, check out the Try Functions experience or create a free Azure acccount. Follow these steps to create a function app in the Azure portal.

1. Go to the Azure portal and sign-in with your Azure account.
2. Click **+New** > **Compute** > **Function App**, select your **Subscription**, type a unique **App name** that identifies your function app, then specify the following settings:

   - **Resource Group**: Select **Create new** and enter a name for your new resource group. You can also choose an existing resource group, however you may not be able to create a consumption-based App Service plan for your function app.
   - **Hosting plan**, which can be one of the following:
     - **Consumption plan**: The default plan type for Azure Functions. When you choose a consumption plan, you must also choose the **Location** and set the **Memory Allocation** (in MB). For information about how memory allocation affects costs, see Azure Functions pricing.
     - **App Service plan**: An App Service plan requires you to create an **App Service plan/location** or select an existing one. These settings determine the location, features, cost and compute resources associated with your app.
   - **Storage account**: Each function app requires a storage account. You can either choose an existing storage account or create one.



Note that you must enter a valid **App name**, which can contain only letters, numbers, and hyphens. Underscore (_) is not an allowed character.

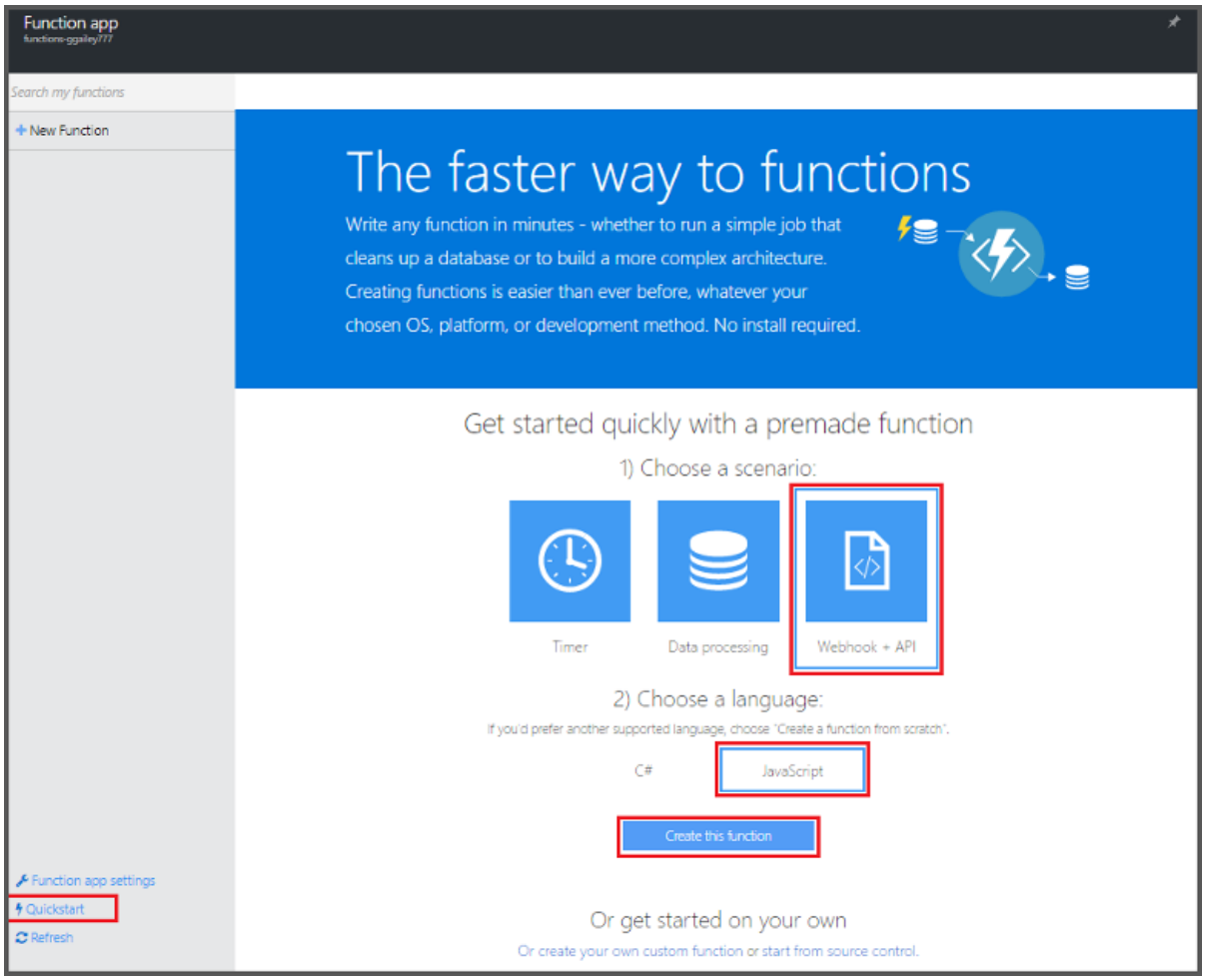
3. Click **Create** to provision and deploy the new function app.

Now that the function app is provisioned, you can create your first function.

Create a function

These steps create a function from the Azure Functions quickstart.

1. In the **Quickstart** tab, click **WebHook + API** and **JavaScript**, then click **Create a function**. A new predefined Node.js function is created.



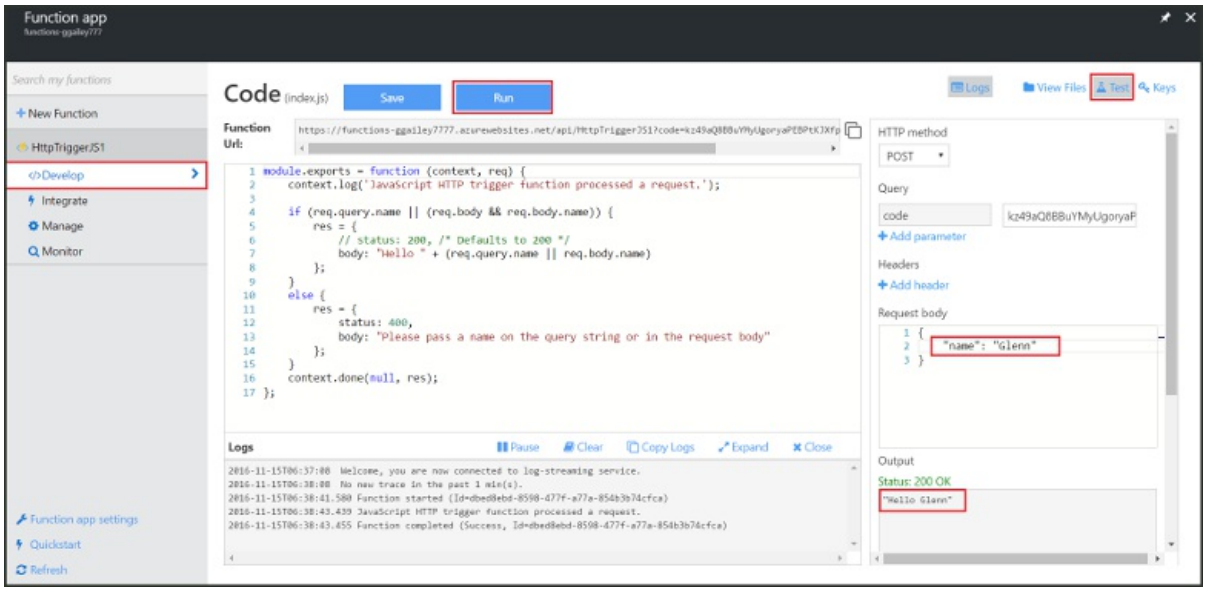
2. (Optional) At this point in the quickstart, you can choose to take a quick tour of Azure Functions features in the portal. After you have completed or skipped the tour, you can test your new function by using the HTTP trigger.

Test the function

Since the Azure Functions quickstarts contain functional code, you can immediately test your new function.

1. In the **Develop** tab, review the **Code** window and notice that this Node.js code expects an HTTP request with a *name* value passed either in the message body or in a query string. When the function runs, this value is returned in the response message.

2. Click **Test** to display the built-in HTTP test request pane for the function.



3. In the **Request body** text box, change the value of the *name* property to your name, and click **Run**. You see that execution is triggered by a test HTTP request, information is written to the logs, and the "hello" response is displayed in the **Output**.

4. To trigger execution of the same function from an HTTP testing tool or from another browser window, copy the **Function URL** value from the **Develop** tab and paste it into the tool or browser address bar. Append the query string value `&name=yourname` to the URL and execute the request. Note that the same information is written to the logs and the same string is contained in the body of the response message.

Next steps

This quickstart demonstrates a simple execution of a basic HTTP-triggered function. To learn more about using Azure Functions in your apps, see the following topics:

- Best Practices for Azure Functions
- 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.
- How to scale Azure Functions
  Discusses service plans available with Azure Functions, including the Consumption hosting plan, and how to choose the right plan.
- What is Azure App Service?
  Azure Functions uses the Azure App Service platform for core functionality like deployments, environment variables, and diagnostics.

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions`. - Visit Stack Overflow

Testing Azure Functions

2/6/2017 • 12 min to read • Edit on GitHub

Overview

This topic demonstrates the various ways to test functions, which includes the following general approaches:

- HTTP-based tools, such as cURL, Postman, and even a web browser for web-based triggers.
- Storage explorer to test Azure Storage-based triggers.
- Test tab in the Functions portal.
- Timer-triggered function.
- Testing application or framework.

All of the testing methods shown use an HTTP trigger function that accepts input through either a query string parameter or the request body. You will create this function in the first section.

Create a function for testing

For most of this tutorial, we will use a slightly modified version of the HttpTrigger JavaScript function template that is available when creating a new function. You can review the Create your first Azure Function tutorial if you need help creating a new function. Just 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 will update the code to also allow you to provide the name and an address as JSON content in the request body. Then the function will echo 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 of 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 use only query string parameters.

To test the function we defined above, copy the **Function Url** from the portal. It will have 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 using 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 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 while 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

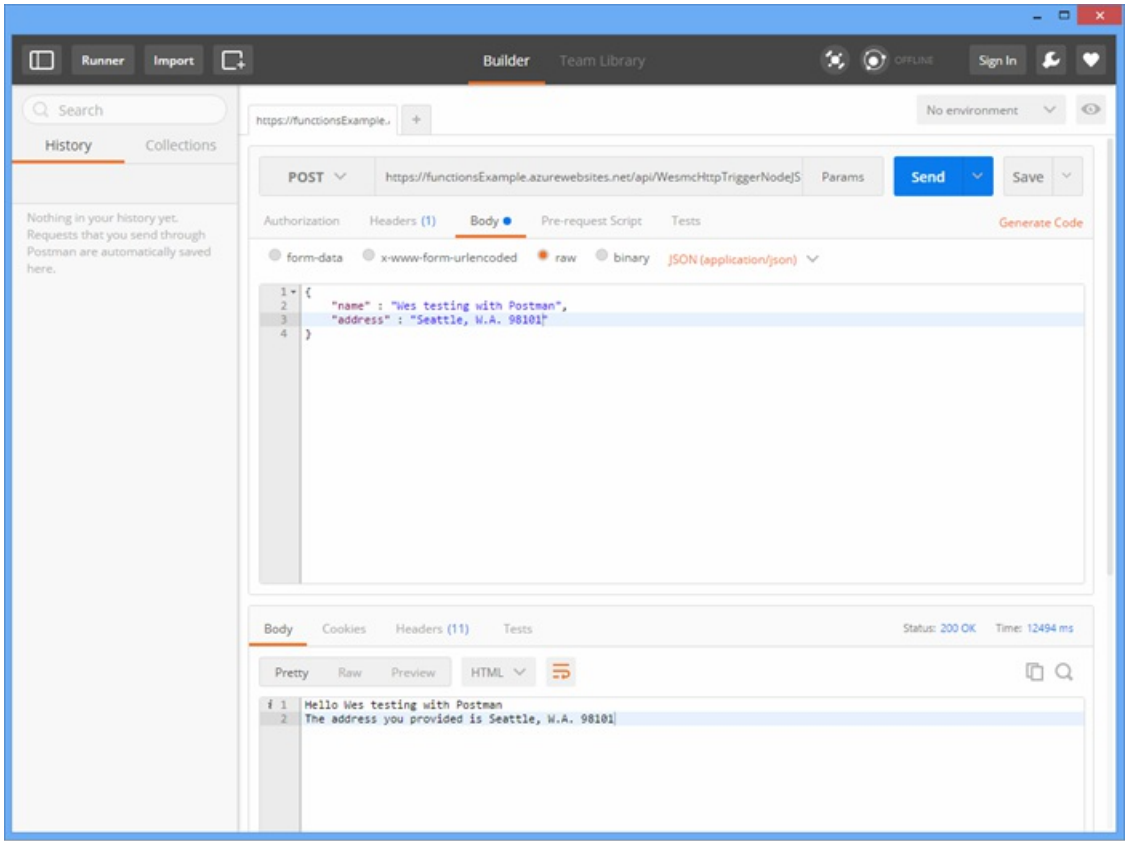
To test the function with a request body in Postman:

1. Launch Postman from the **Apps** button in the upper left of 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 JSON request body similar to the following:

```
{
    "name" : "Wes testing with Postman",
    "address" : "Seattle, W.A. 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 while 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 testing software, it's not necessary to look any further than the command-line to help debug your application, this is no different with 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 above, copy the **Function URL** from the portal. It will have the following form:

```
https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code>
```

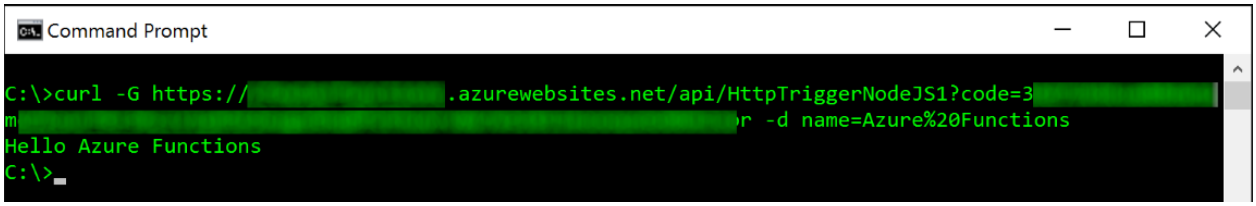
This is the URL for triggering your function, we can test this by using the cURL command on the command-line to make a GET (`-G` or `--get`) request against the function:

```
curl -G https://<Your Function App>.azurewebsites.net/api/<Your Function Name>?code=<your access code>
```

This particular example above 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:



In the portal **Logs** window, output similar to the following is logged while 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-b706-482fcd1b8f7a)
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-b706-482fcd1b8f7a)
```

Test a blob trigger using Storage Explorer

You can test a blob trigger function using Microsoft Azure Storage Explorer.

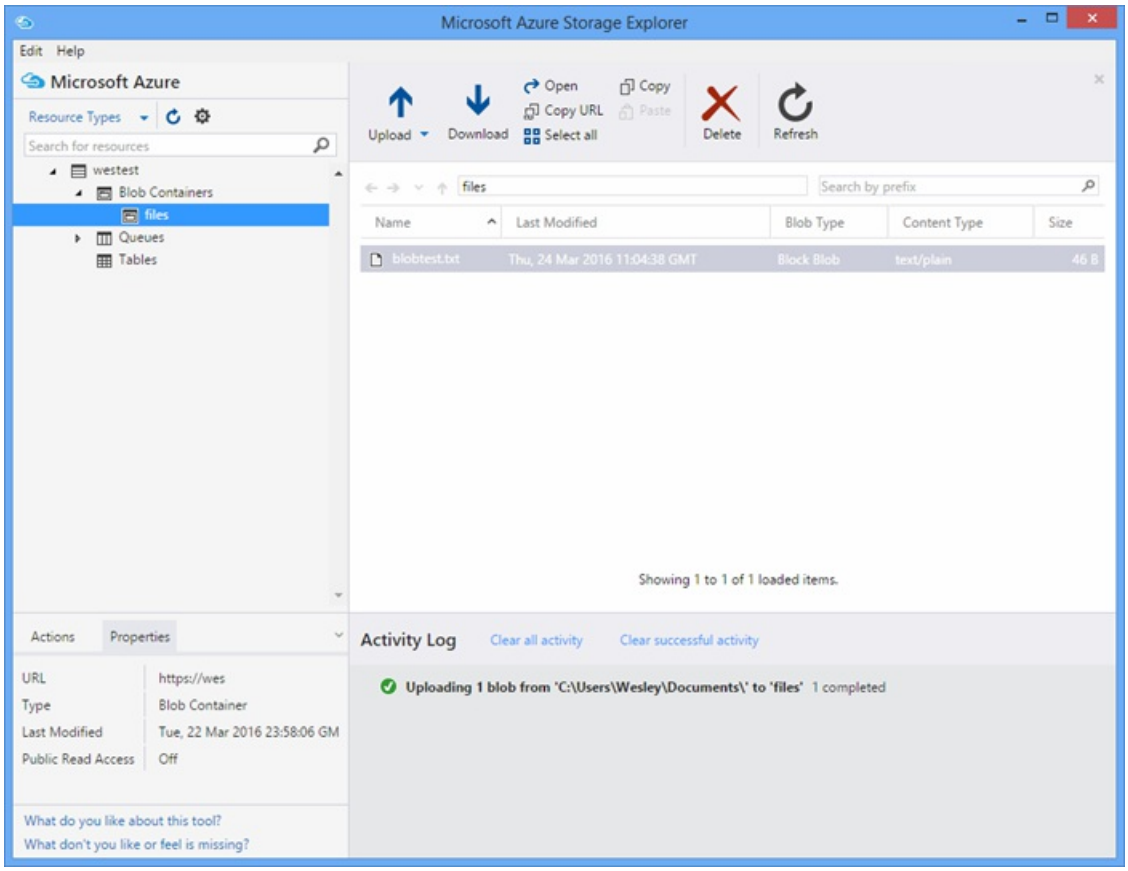
1. In the Azure Portal for your Functions app, create a new C#, F# or JavaScript blob trigger function. Set the path to monitor to the name of your blob container. For example:

   ```
   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 Microsoft Azure Storage Explorer and connect to the blob container in the storage account being monitored.
5. Click the **Upload** button and upload the text file.



The default blob trigger function code will report 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 which will allow you to do some limited testing. You can provide a request body using the run 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 while 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 truly tested with the tools mentioned previously. For example, a queue trigger function which runs when a message is dropped into Azure Queue Storage. You could always write code to drop a message into your queue and an example of this in a console project is provided below. However, there is another approach you can use to test with functions directly.

You could use a timer trigger configured with a queue output binding. That timer trigger code could then write the test messages to the queue. This section will walk through through an example.

For more in-depth information on using bindings with Azure Functions, see the Azure Functions developer reference.

Create queue trigger for testing

To demonstrate this approach, we will first create a queue trigger function that we want to test for a queue named `queue-newusers`. This function will process name and address information for a new user dropped into Azure queue storage.

> **NOTE**
> If you use a different queue name, make sure the name you use conforms to the Naming Queues and MetaData rules. Otherwise, you will get a HTTP Status code 400 : Bad Request.

1. In the Azure Portal for your Functions app, click **New Function** > **QueueTrigger - C#**.
2. Enter the queue name to be monitored by the queue function

   ```
   queue-newusers
   ```

3. Click the **+** (add) button to select or create the storage account you want to use. Then click **Create**.
4. Leave this portal browser window opened 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 will execute testing 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 the **+ New Output** button. Then click **queue** and the **Select** button.
5. Note the name you use for the **queue message object** you will use this in the timer function code.

```
myQueue
```

6. Enter the queue name where the message will be sent:

```
queue-newusers
```

7. Click the **+** (add) 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 above. 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 C# timer function will execute every 30 seconds if you used the example cron expression. The logs for the timer function will 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 will see the 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

There will some cases where you need to create an external application or framework to test your functions.

Test a 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=sc1wt62opn7k9buhrm8jpds4ikxvvj42m5ojdt0p91lz5jnhfr2c74ipoujyq26wab3wk5gkfbt9&" + nameQueryString,
  path: "/api/HttpTriggerNodeJS2?
code=sc1wt62opn7k9buhrm8jpds4ikxvvj42m5ojdt0p91lz5jnhfr2c74ipoujyq26wab3wk5gkfbt9",
  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 while 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 could test a queue trigger by using code to drop a message in your queue. The following example code is based off 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.
- This code accepts the name and address for a new user as command-line arguments during runtime. Pass a `name` and `address` as parameters to the app. For example,
  `C:\myQueueConsoleApp\test.exe "Wes testing queues" "in a console app"`

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 will see the 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)
```

How to code and test Azure functions locally

1/17/2017 • 1 min to read • Edit on GitHub

NOTE

The Azure Functions local development and tooling experience is currently in preview and the experience will be significantly improved before the final release. The easiest way to run the Azure Functions host locally is use the Azure Functions CLI and install it from npm. Currently, only Windows is supported. Full documentation is coming soon. To provide feedback or file bugs on the Azure Functions CLI, please file an issue in the azure-webjobs-sdk-script GitHub repo.

Best Practices for Azure Functions

2/6/2017 • 3 min to read • Edit on GitHub

Overview

This article provides a collection of best practices for you to consider when implementing function apps. Keep in mind that your Azure Function App is an Azure App Service. So those best practices would apply.

Avoid large long running functions

Large long running functions can cause unexpected timeout issues. A function can be large because of many Node.js dependencies. Importing these dependencies can cause increased load times resulting in unexpected timeouts. Node.js dependencies could be explicitly loaded by multiple `require()` statements in your code. They could also be implicit based on a single module loaded by your code that has its own internal dependencies.

Whenever possible refactor large functions into smaller function sets that work together and return fast responses. For example, a webhook or HTTP trigger function might require an acknowledgment response within a certain time limit. 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. It is common for webhooks to require an immediate response.

Cross function communication.

When integrating 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.

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.

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:

```
#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 Task.Result

Asynchronous programming is a recommended best practice. However, always avoid referencing the `Task.Result` property. This approach essentially does a busy-wait on a lock of another thread. Holding a lock creates the potential for deadlocks.

Next steps

For more information, see the following resources:

- Azure Functions developer reference
- Azure Functions C# developer reference
- Azure Functions F# developer reference
- Azure Functions NodeJS developer reference

Use Azure Functions to perform a scheduled clean-up task

1/17/2017 • 3 min to read • Edit on GitHub

This topic shows you how to use Azure Functions to create a new function in C# that runs based on an event timer to clean-up rows in a database table. The new function is created based on a pre-defined template in the Azure Functions portal. To support this scenario, you must also set a database connection string as an App Service setting in the function app.

Prerequisites

Before you can create a function, you need to have an active Azure account. If you don't already have an Azure account, free accounts are available.

This topic demonstrates a Transact-SQL command that executes a bulk cleanup operation in table named *TodoItems* in a SQL Database. This same TodoItems table is created when you complete the Azure App Service Mobile Apps quickstart tutorial. You can also use a sample database If you choose to use a different table, you will need to modify the command.

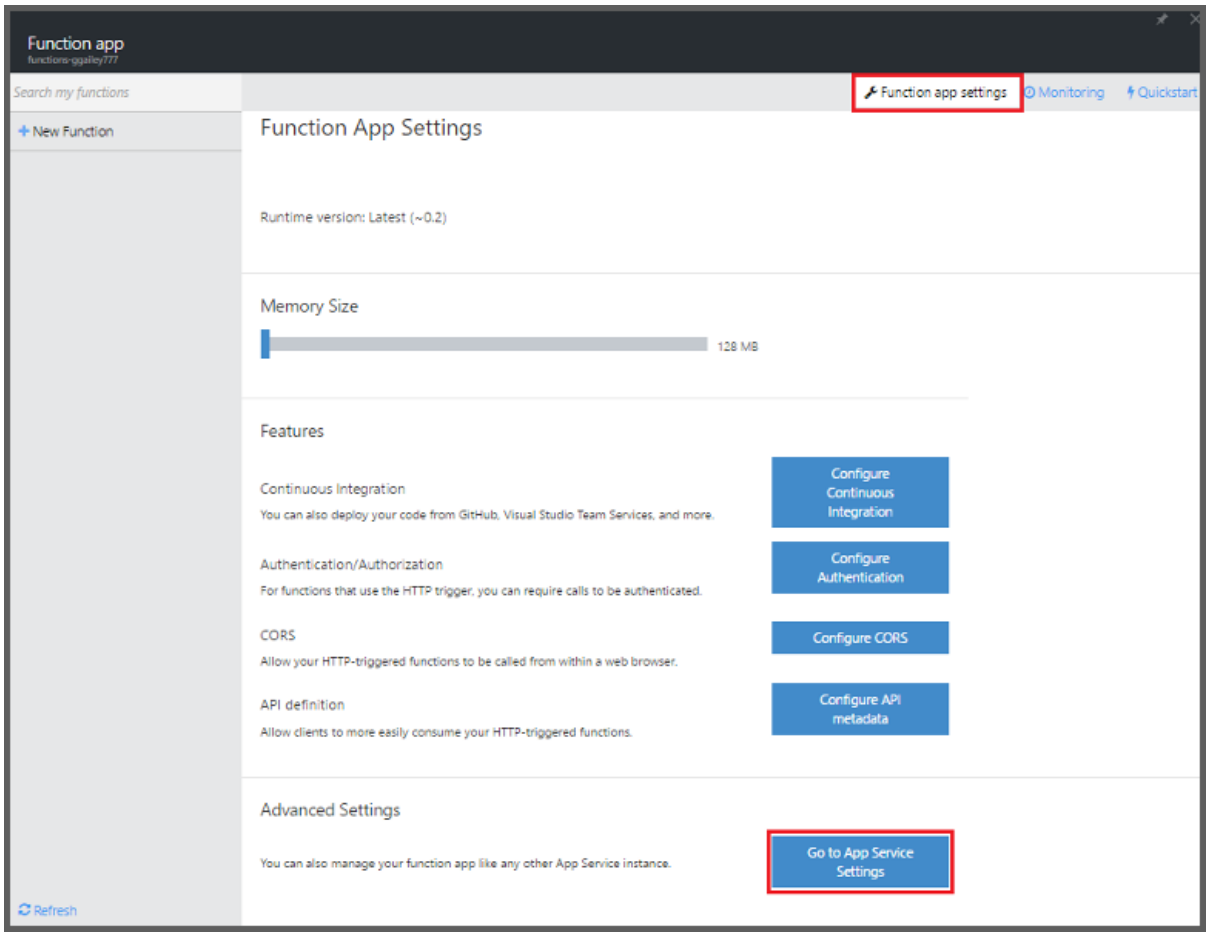
You can get the connection string used by a Mobile App backend in the portal under **All settings** > **Application settings** > **Connection strings** > **Show connection string values** > **MS_TableConnectionString**. You can also get the connection string direct from a SQL Database in the portal under **All settings** > **Properties** > **Show database connection strings** > **ADO.NET (SQL authentication)**.

This scenario uses a bulk operation against the database. To have your function process individual CRUD operations in a Mobile Apps table, you should instead use Mobile Table binding.

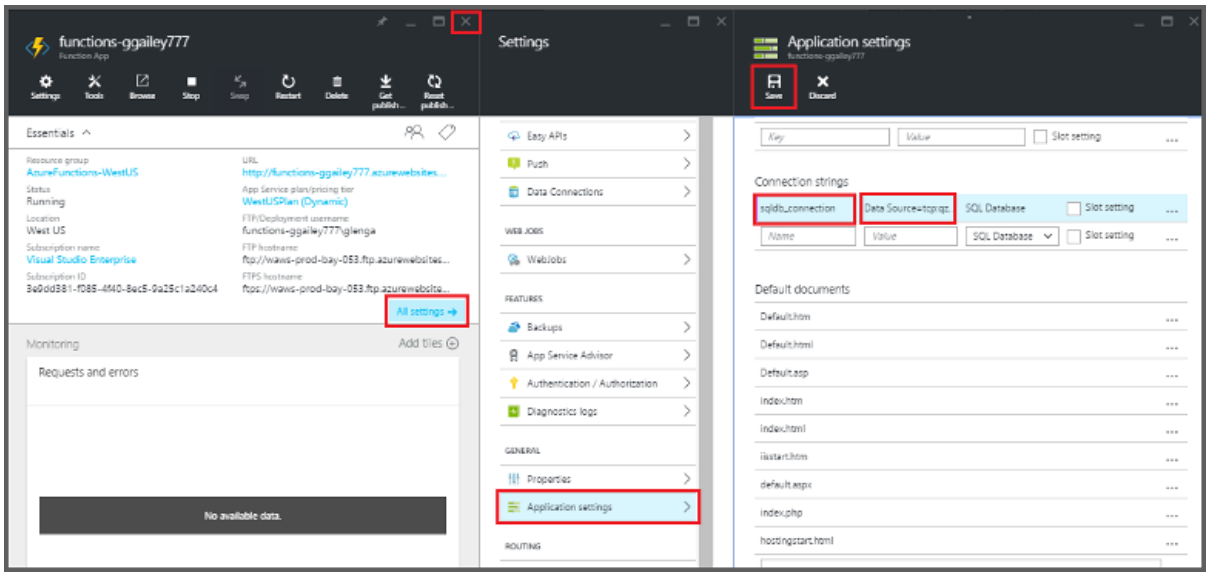
Set a SQL Database connection string in the function app

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. This prevents accidental disclosure when your function code ends-up in a repo somewhere.

1. Go to the Azure Functions portal and sign-in with your Azure account.
2. If you have an existing function app to use, select it from **Your function apps** then click **Open**. To create a new function app, type a unique **Name** for your new function app or accept the generated one, select your preferred **Region**, then click **Create + get started**.
3. In your function app, click **Function app settings** > **Go to App Service settings**.



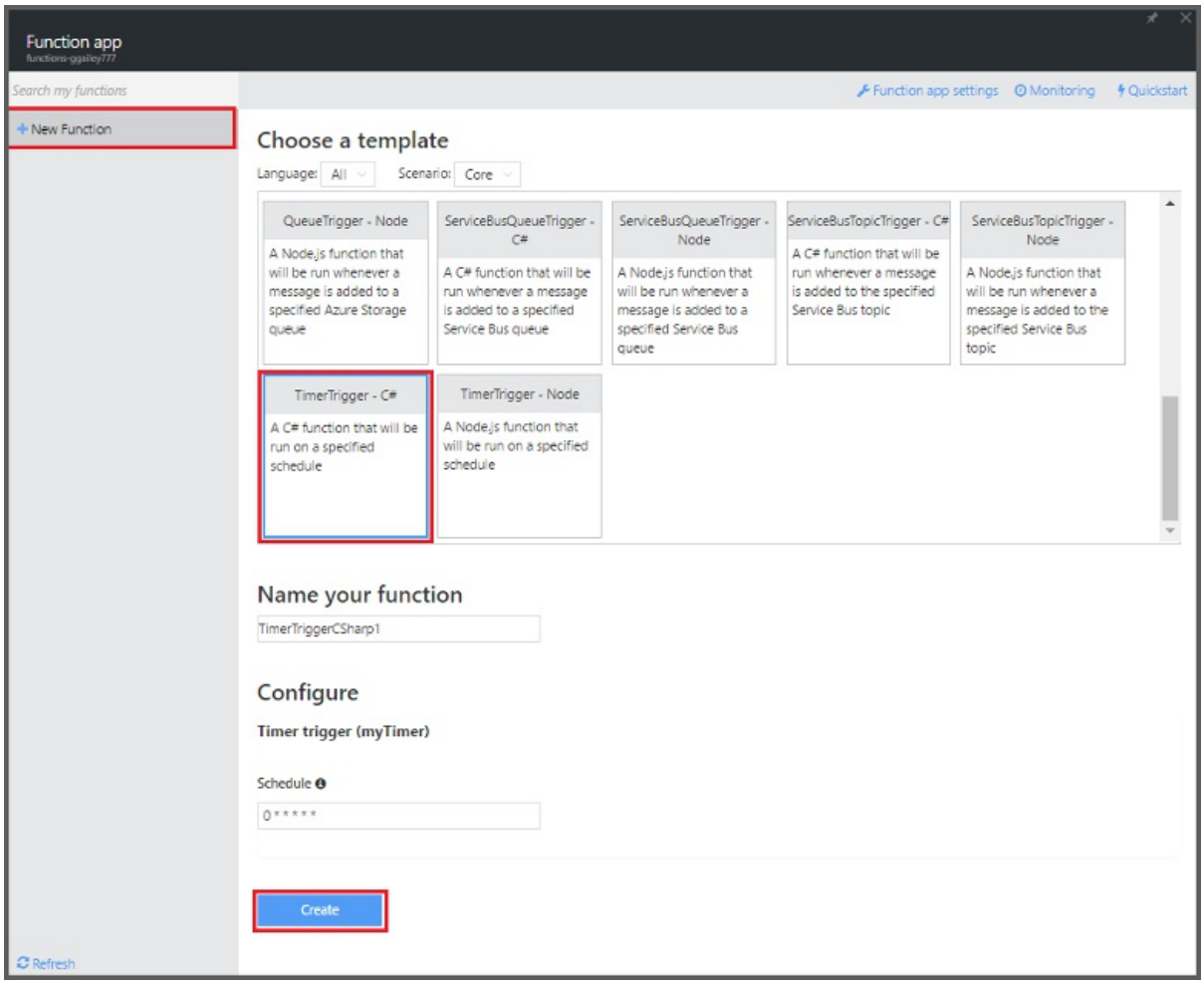
4. In your function app, click **All settings**, scroll down to **Application settings**, then under **Connection strings** type `sqldb_connection` for **Name**, paste the connection string into **Value**, click **Save**, then close the function app blade to return to the Functions portal.



Now, you can add the C# function code that connects to your SQL Database.

Create a timer-triggered function from the template

1. In your function app, click **+ New Function** > **TimerTrigger - C#** > **Create**. This creates a function with a default name that is run on the default schedule of once every minute.



2. In the **Code** pane in the **Develop** tab, 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 = "DELETE from dbo.TodoItems WHERE Complete='True'";
        using (SqlCommand cmd = new SqlCommand(text, conn))
        {
            // Execute the command and log the # rows deleted.
            var rows = await cmd.ExecuteNonQueryAsync();
            log.Info($"{rows} rows were deleted");
        }
    }
}
```

5. Click **Save**, watch the **Logs** windows for the next function execution, then note the number of rows deleted from the TodoItems table.

6. (Optional) Using the Mobile Apps quickstart app, mark additional items as "completed" then return to the **Logs** window and watch the same number of rows get deleted by the function during the next execution.

Next steps

See these topics for more information about Azure Functions.

- 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.
- How to scale Azure Functions
  Discusses service plans available with Azure Functions, including the Consumption plan, and how to choose the right plan.

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions`. - Visit Stack Overflow

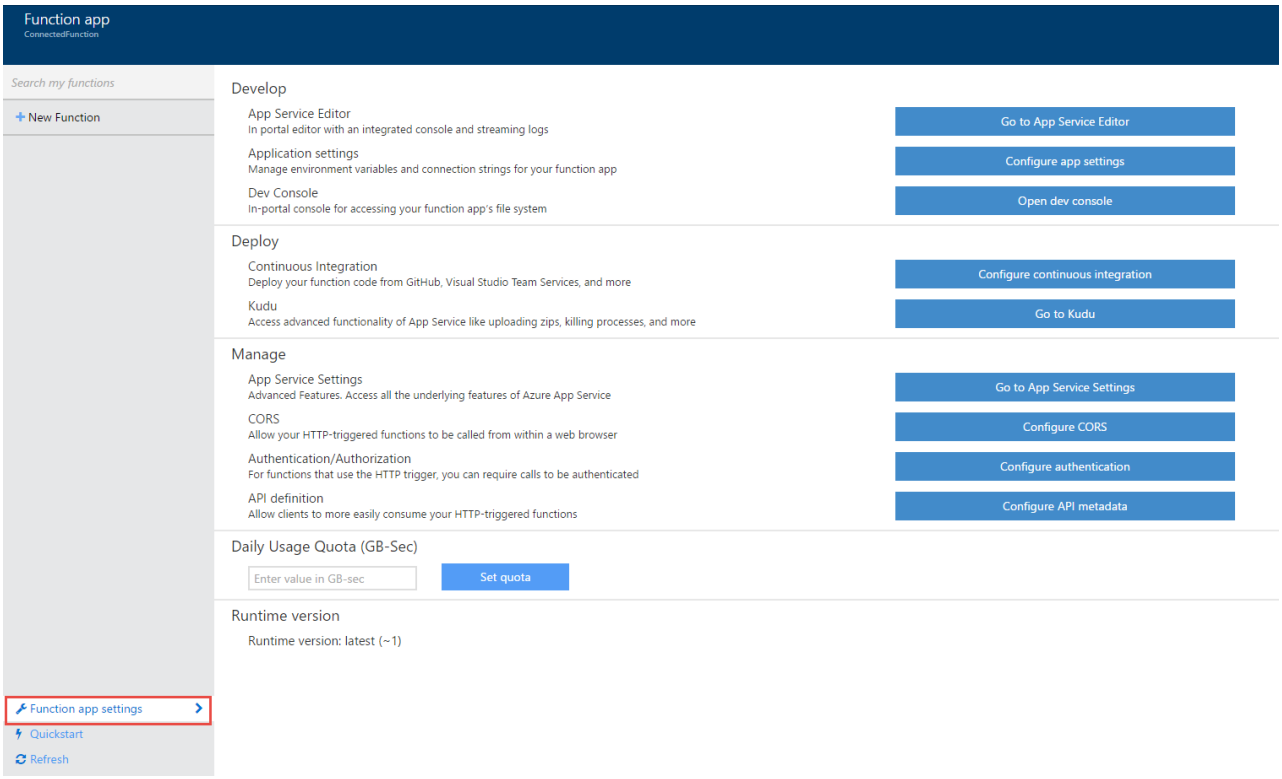
How to configure Azure Function app settings

1/17/2017 • 4 min to read • Edit on GitHub

Settings overview

You can manage Azure Function Apps settings by clicking the **Function App Settings** link in the bottom-left corner of the portal. Azure function app settings apply to all functions in the app.

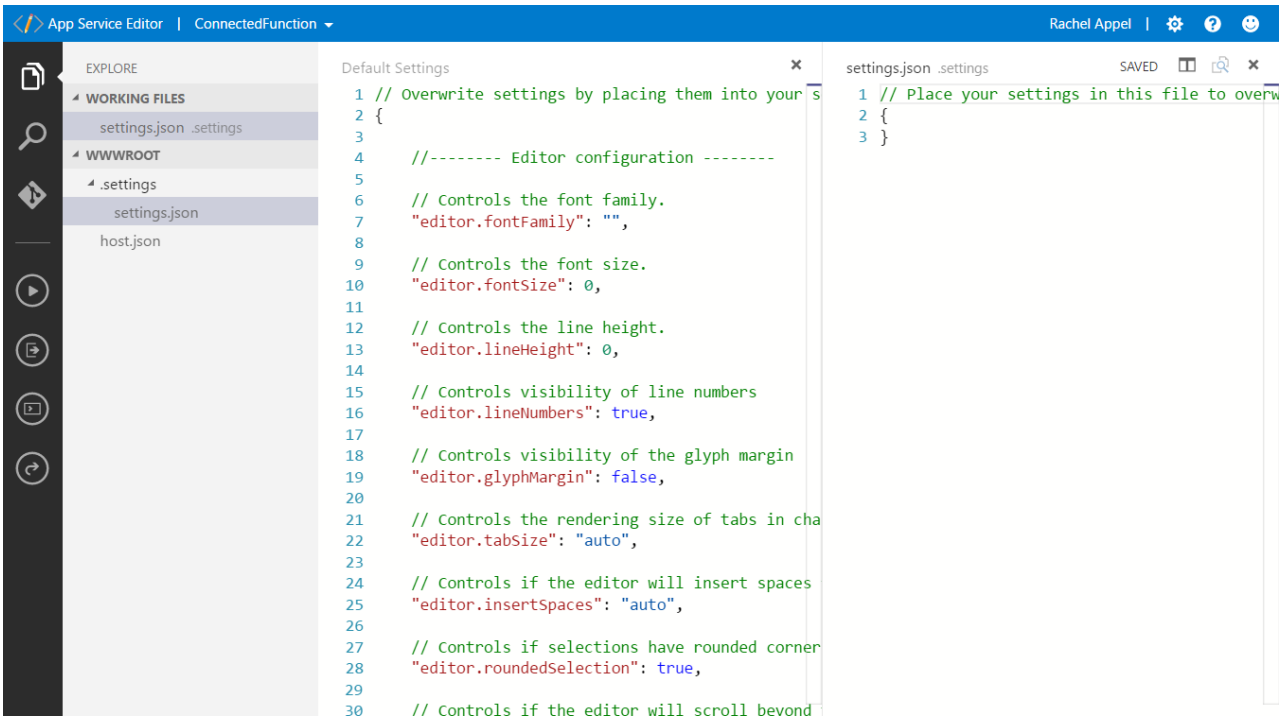
1. Go to the Azure portal and sign-in with your Azure account.
2. Click **Function App Settings** in the bottom-left corner of the portal. This action reveals several configuration options to choose from.



Develop

App Service Editor

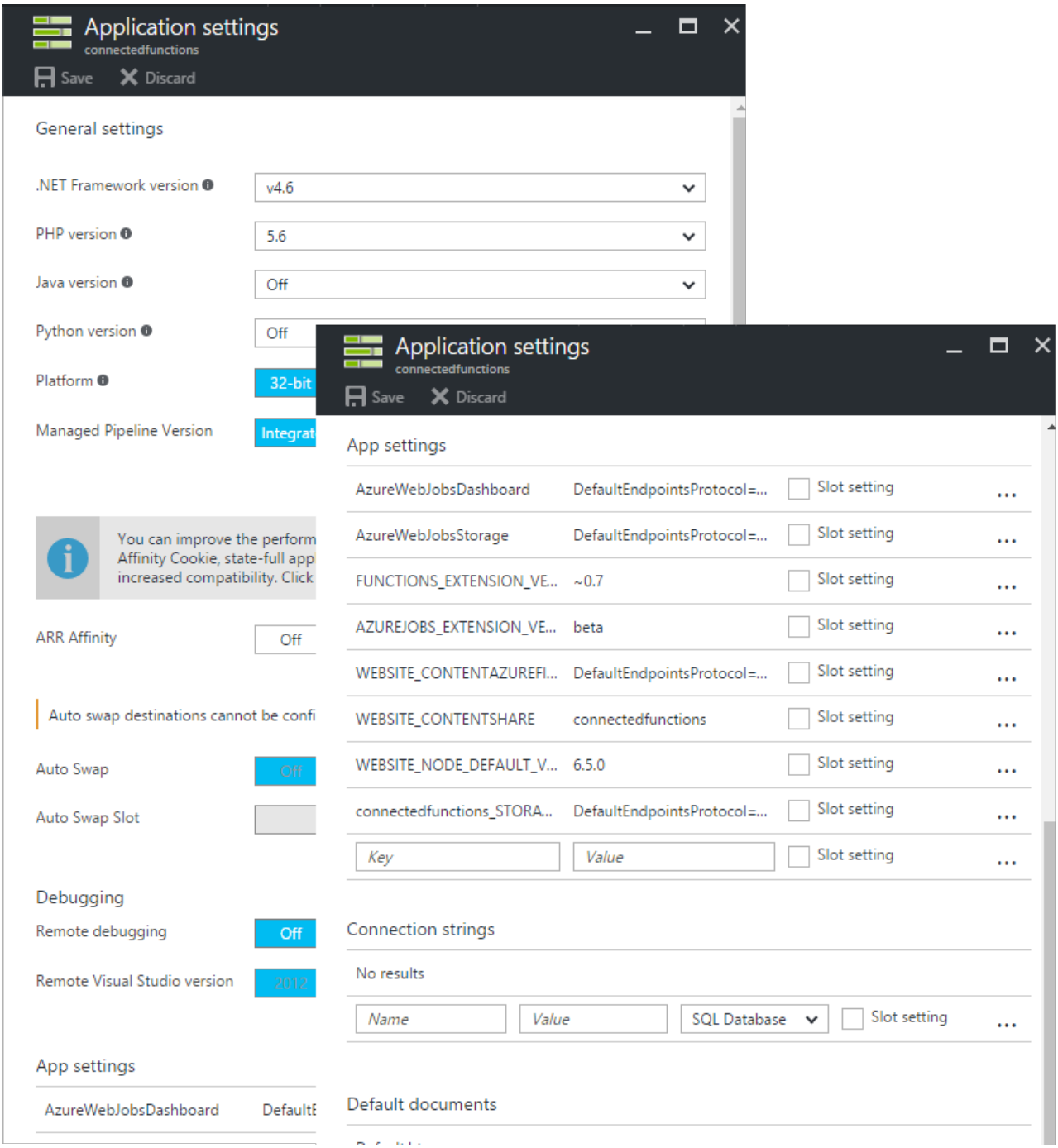
The App Service Editor is an advaned in-portal editor that you can use to modify Json configuration files and code files alike. Choosing this option launches a separate browser tab with a basic editor. This enables you to integrate with Github, run and debug code, and modify function app settings.



Application settings

Manage environment variables, Framework versions, remote debugging, app settings, connection strings, default docs, etc. These settings are specific to your Function App.

To configure app settings, click the **Configure App Settings** link.

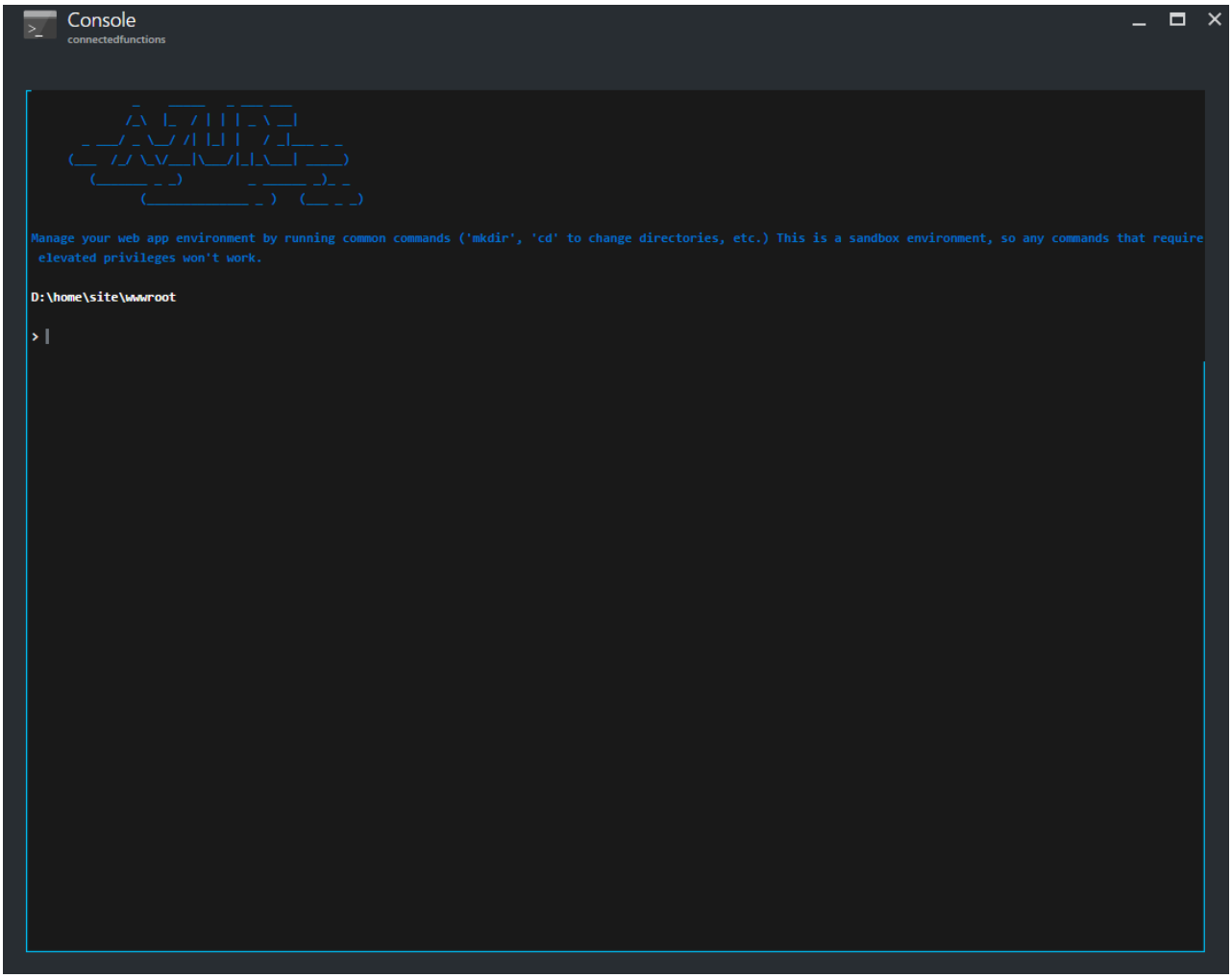


Dev console

You can execute DOS style commands with the Azure functions in-portal console. Common commands include directory and file creation and navigation, as well as executing batch files and scripts.

> **NOTE**
> You can upload scripts, but first you must configure an FTP client in the Azure Function's **Advanced Settings**.

To open the In-portal console, click **Open dev console**.



NOTE

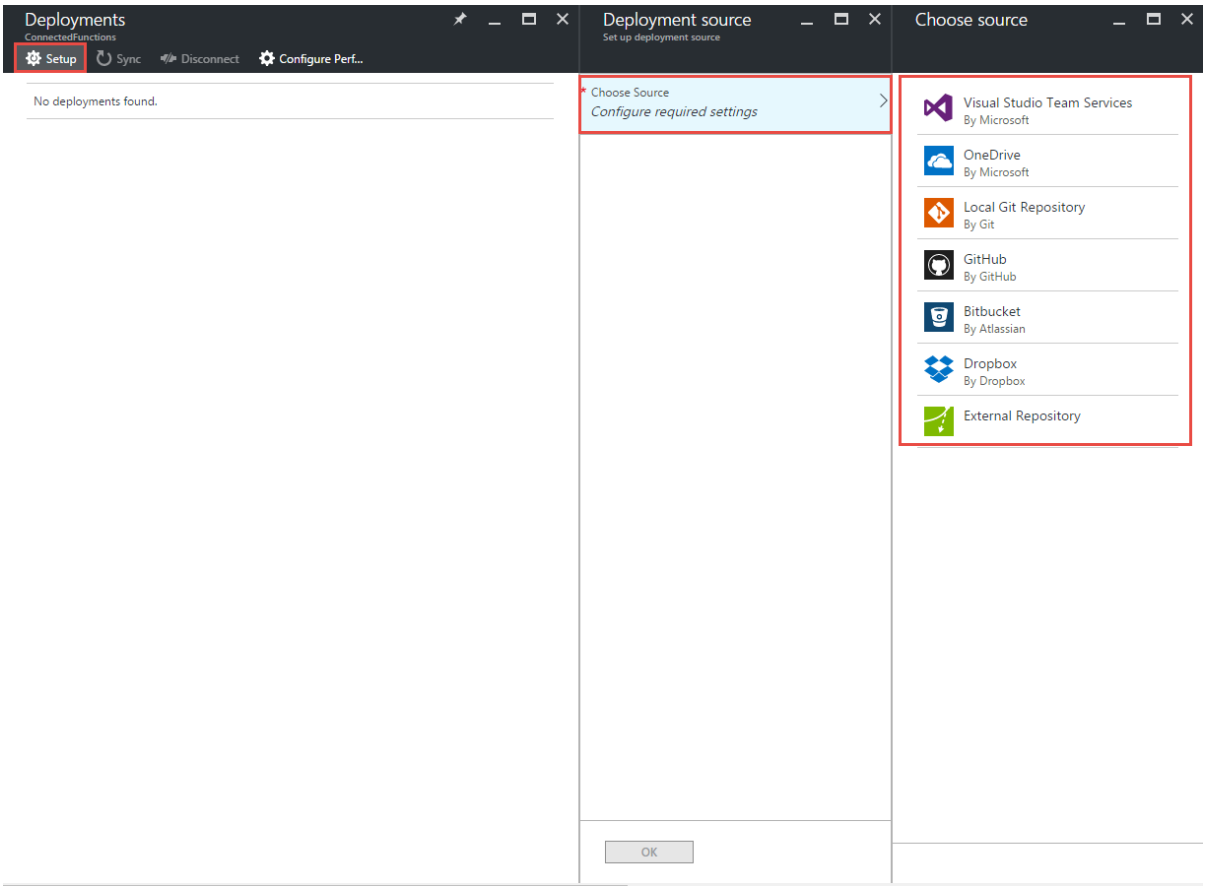
Working in a console with ASCII art like that makes you look cool.

Deploy

Continuous integration

You can integrate your Function App with GitHub, Visual Studio Team Services, and more.

1. Click the **Configure continuous integration** link. This opens a **Deployments** pane with options.
2. Click **Setup** in the **Deployments** pane to reveal a **Deployment Source** pane with one option: Click **Choose Source** to show available sources.
3. Choose any of the deployment sources available: Visual Studio Team Services, OneDrive, Local Git Repository, GitHub, Bitbucket, DropBox, or an External Repository by clicking it.



4. Enter your credentials and information as prompted by the various deployment sources. The credentials and information requested may be slightly different depending on what source you have chosen.

Once you have setup CI, connected code you push to the configured source is automatically deployed to this function app.

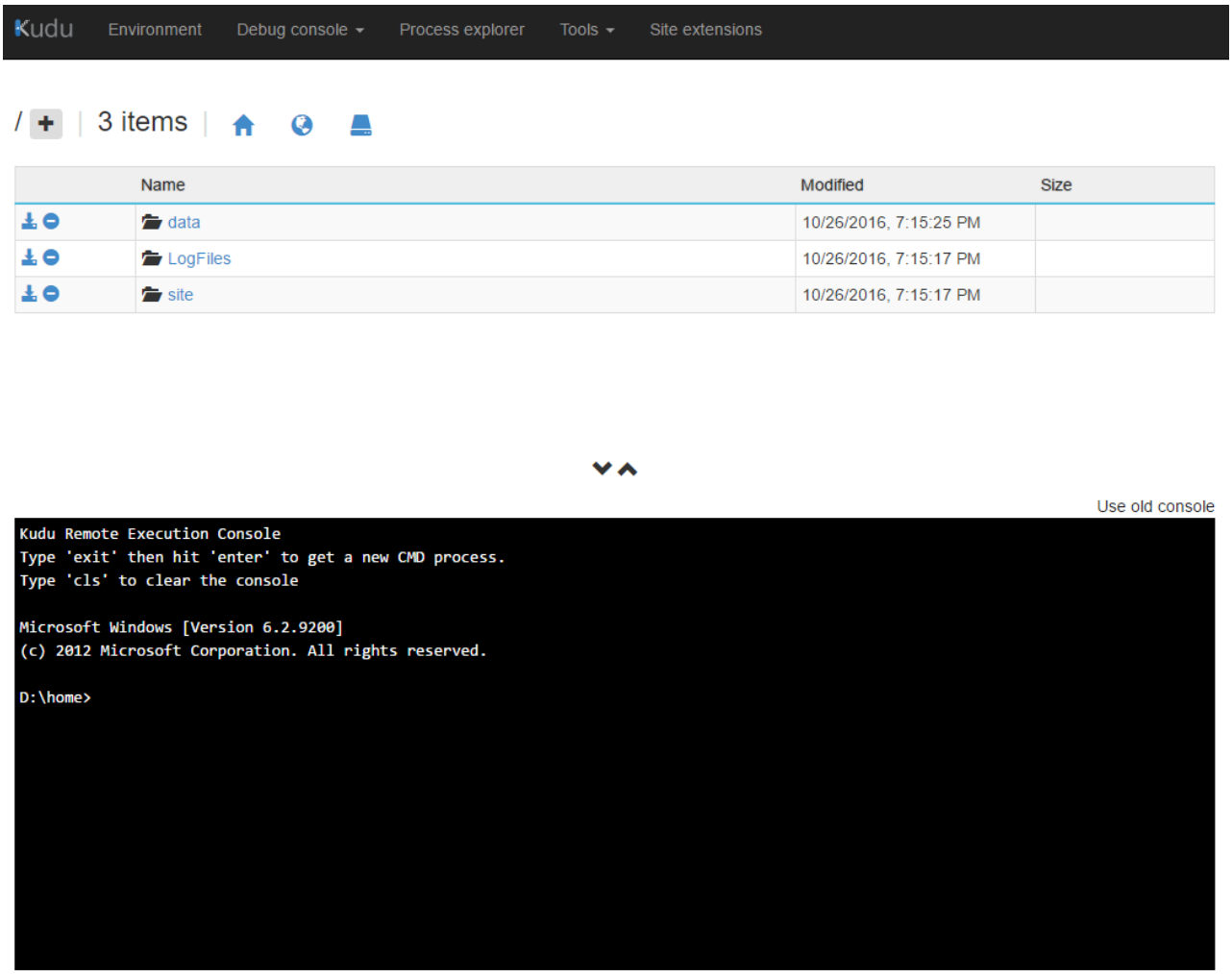
Kudu

Kudu allows you to access advanced administrative features of a Function App.

To open Kudu, click **Go to Kudu**. This action opens an entirely new browser window with the Kudu web admin.

> **NOTE**
>
> You can alternatively launch **Kudu** by inserting "scm" into your function's URL, as shown here:
>
> `https://<YourFunctionName>.scm.azurewebsites.net/`

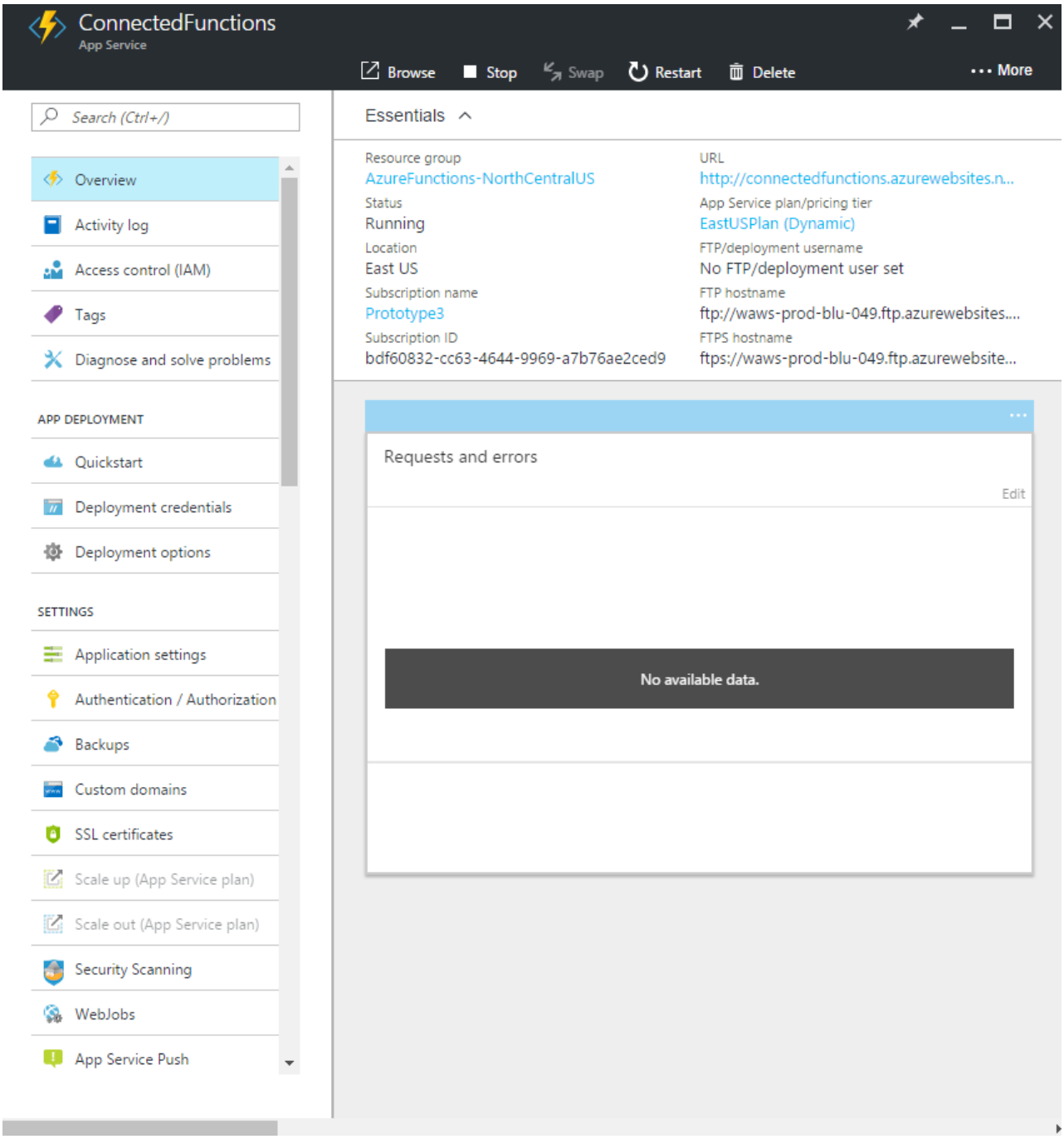
From the Kudu webpage, you can view and manage system information, app settings, environment variables, HTTP headers, server variables, and more.



Manage: App Service settings

Manage your function app like any other App Service instance. This option gives you access to all the previously discussed settings, plus several more.

To open advanced settings, click the **Advanced Settings** link.

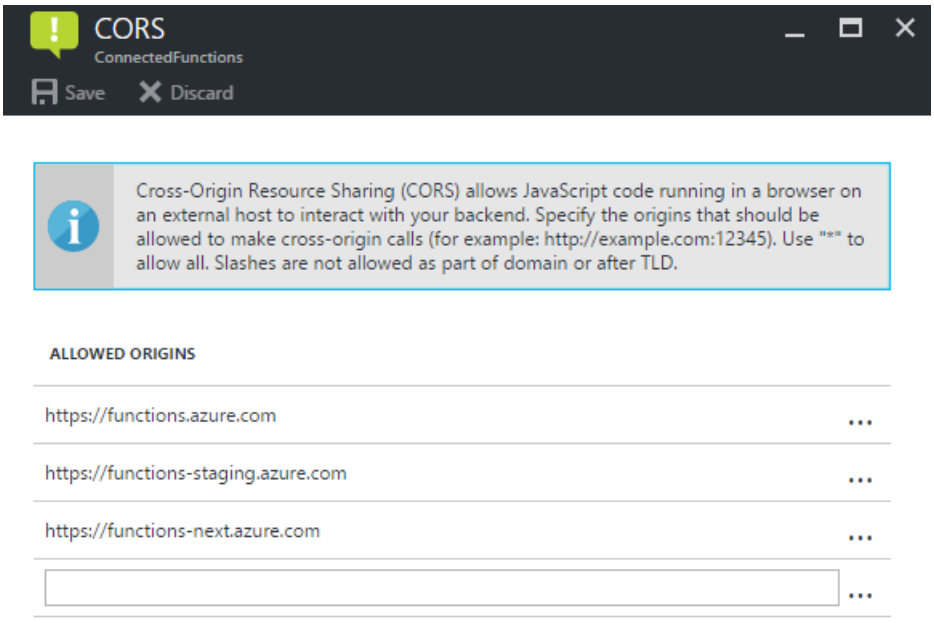


For details on how to configure each App Service setting, see Configure Azure App Service Settings.

Manage: CORS

Normally, for security reasons, calls to your hosts (domains) from external sources, such as Ajax calls from a browser, are not allowed. Otherwise, malicious code could be sent to and executed on the backend. The safest route then is to blacklist all sources of code, except for a few of your own trusted ones. You can configure which sources you accept calls from in Azure functions by configuring Cross-Origin Resource Sharing (CORS). CORS allows you to list domains that are the source of JavaScript that can call functions in your Azure Function App.

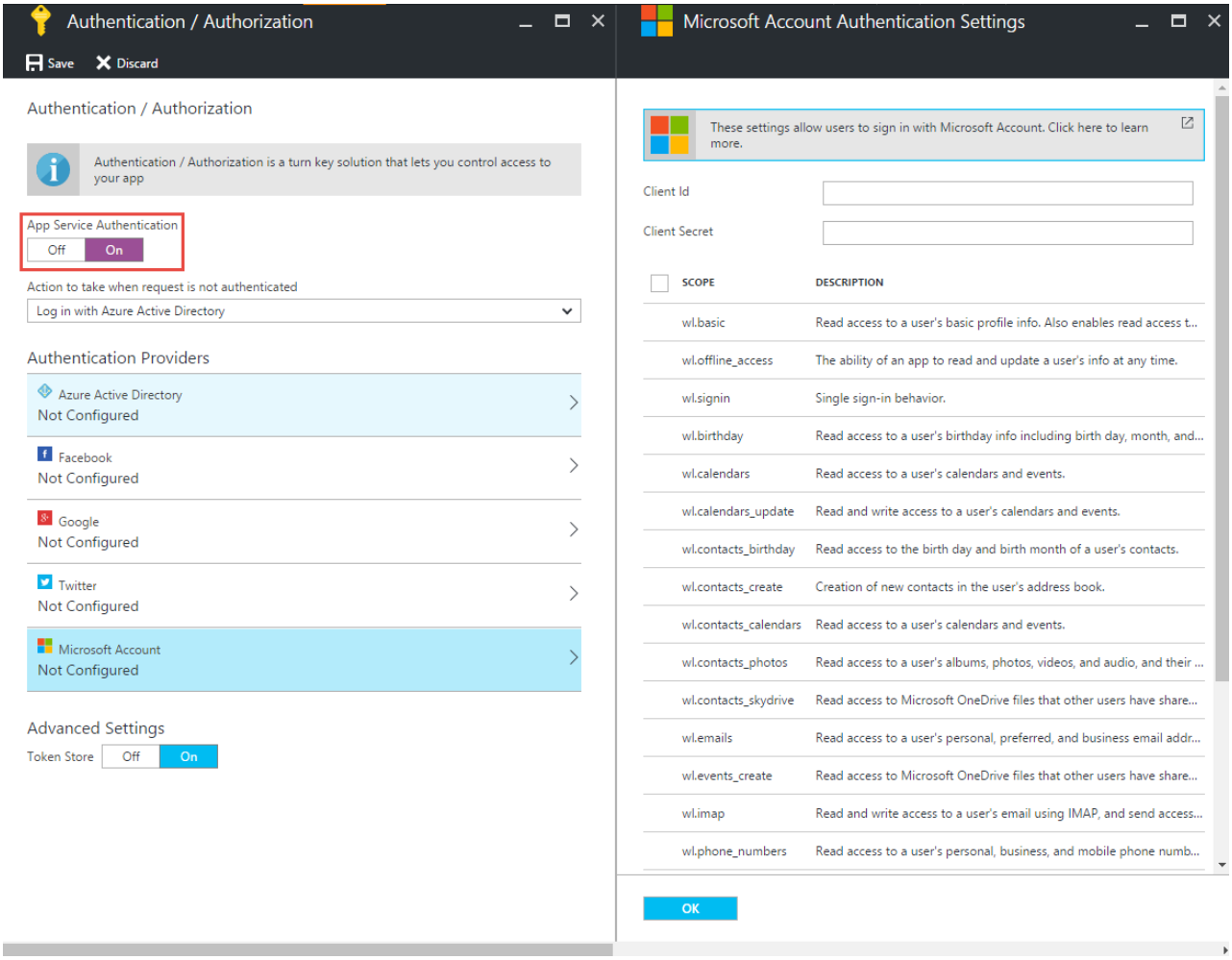
1. To configure CORS, click the **Configure CORS** link.
2. Enter the domains that you want to whitelist.



Manage: Authentication/authorization

For functions that use an HTTP trigger, you can require calls to be authenticated.

1. To configure authentication click the **Configure authentication** link.
2. Toggle the **App Service Authentication** button to **On**.



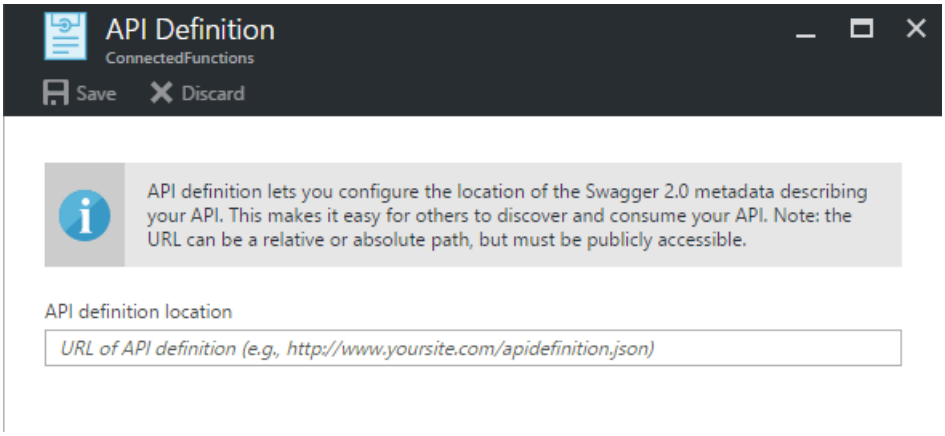
Most authentication providers ask for an API Key/Client ID and a Secret; however, both the Microsoft Account and Facebook options also allow you to define scopes (specific authorization credentials). Active Directory has several express or advanced configuration settings you can set.

For details on configuring specific authentication providers, see Azure App Service authentication overview.

Manage: API definition

Allow clients to more easily consume your HTTP-triggered functions.

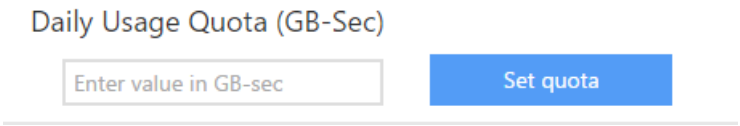
1. To set up an API, click **Configure API metadata**.
2. Enter the URL that points to a Swagger json file.



For more information on creating API definitions with Swagger, visit Get Started with API Apps, ASP.NET, and Swagger in Azure.

Daily Usage Quota

Azure Functions enables you to predictably limit platform usage by setting a daily spending quota. Once the daily spending 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. The unit of the spending quota is the unit of billing: GB-s (gigabyte-seconds), please refer to the Azure Functions pricing page for details on the billing model.



Next steps

Need some help?

Post questions in the Azure forums. - Visit MSDN

Tag questions with the keyword `azure-functions` . - Visit Stack Overflow

Continuous deployment for Azure Functions

1/17/2017 • 5 min to read • Edit on GitHub

Azure Functions makes it easy to configure continuous deployment for your function app. Functions uses Azure App Service integration with BitBucket, Dropbox, GitHub, and Visual Studio Team Services (VSTS) to enable a continuous deployment workflow where Azure pulls updates to your functions code when they are published to one of these services. 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
- Git local repo
- Git external repo
- GitHub
- Mercurial external repo
- 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. This folder structure is essentially your site code.

The code for all of the functions in a given function app lives in a root folder that contains a host configuration file and one or more subfolders, each of which contain 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
```

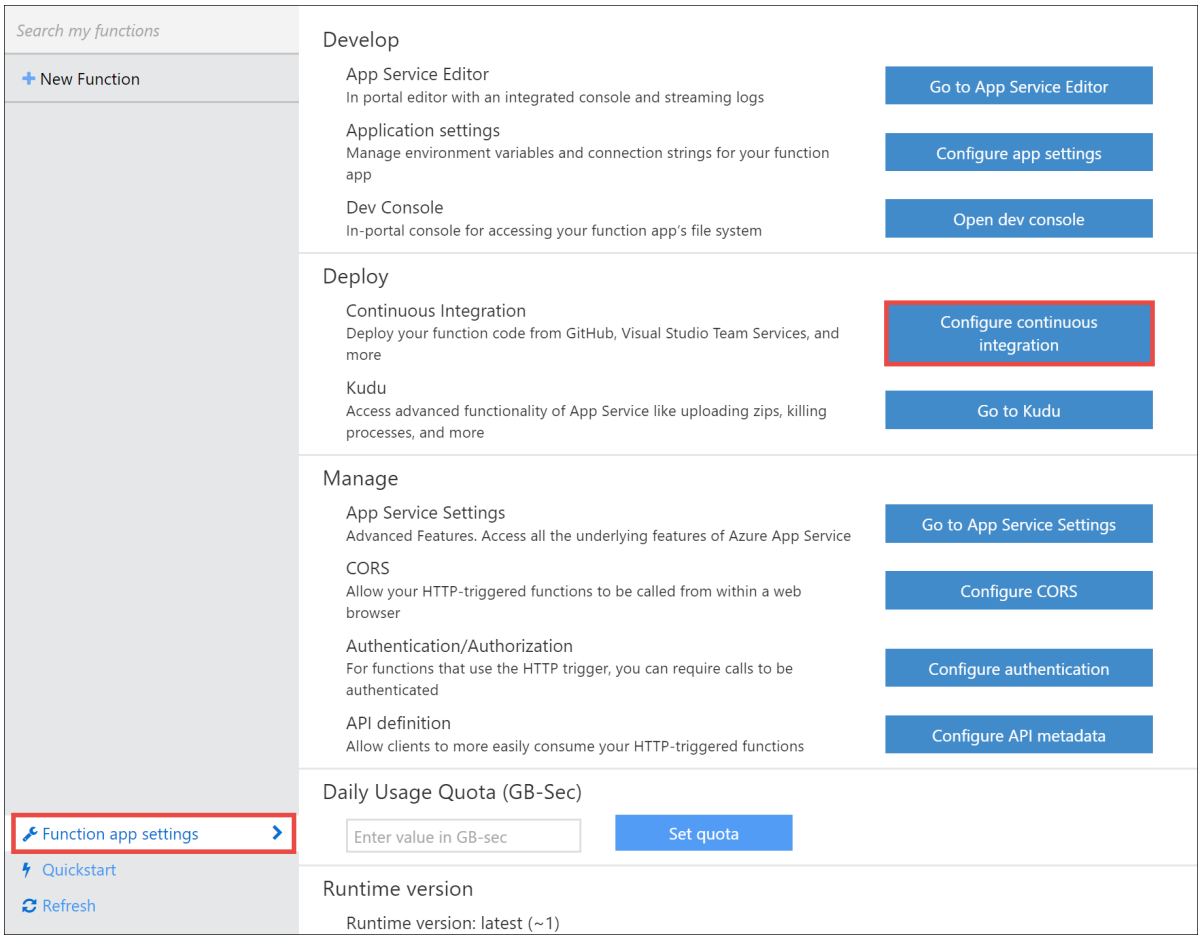
The *host.json* file contains some runtime-specific configuration and sits in the root folder of the function app. For information on settings that are available, see host.json in the WebJobs.Script repository wiki.

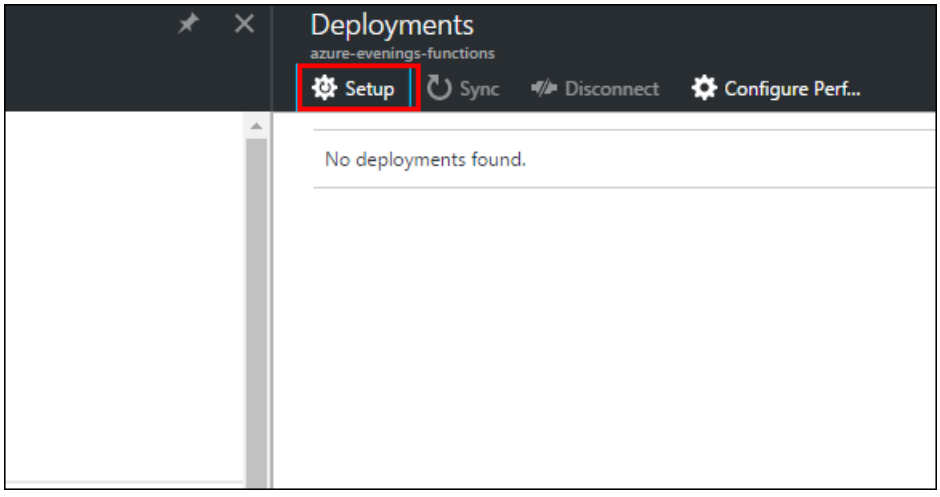
Each function has a folder that contains one or more code files, the function.json configuration and other dependencies.

Set up continuous deployment

Use the following procedure to configure continuous deployment for an existing function app:

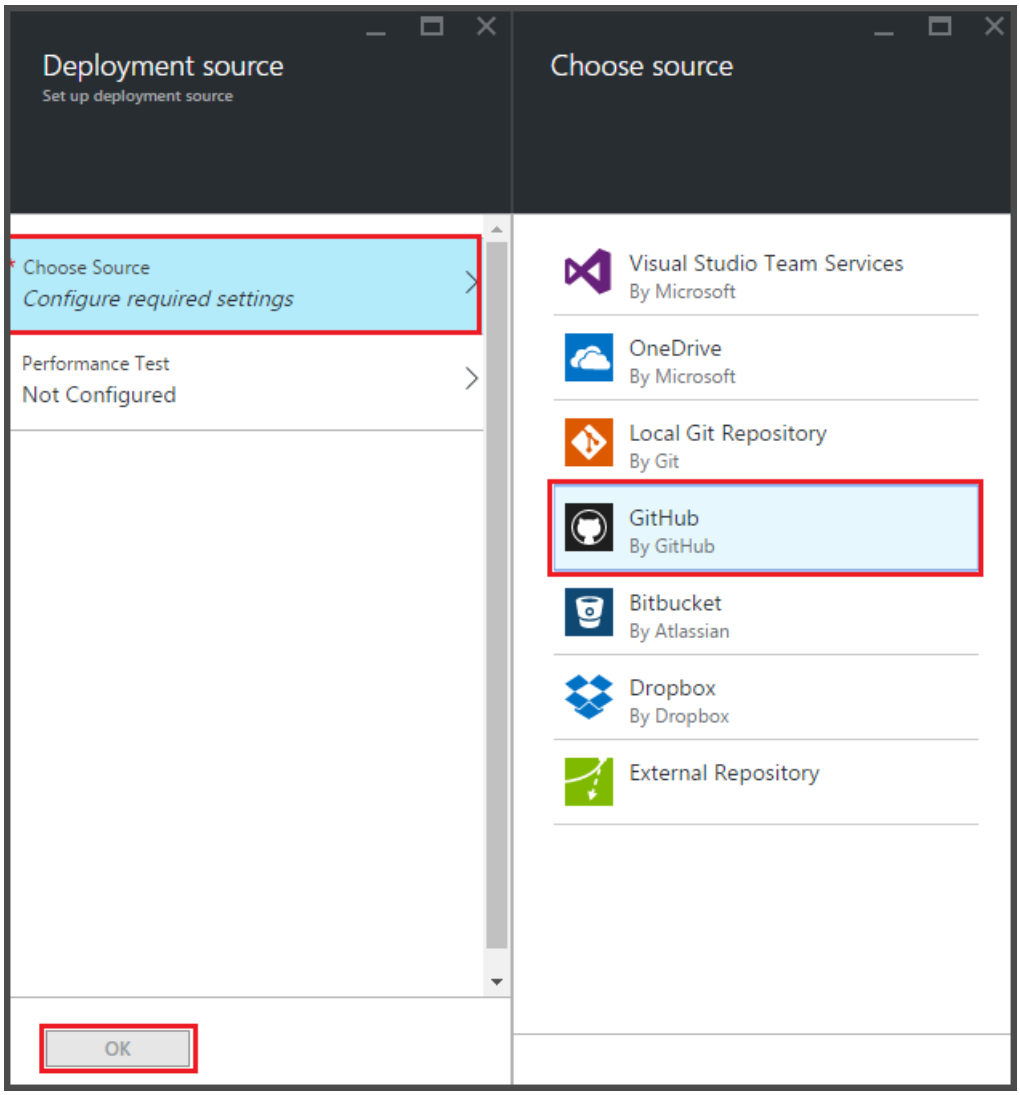
1. In your function app in the Azure Functions portal, click **Function app settings** > **Configure continuous integration** > **Setup**.





You can also get to the Deployments blade from the Functions quickstart by clicking **Start from source control**.

2. 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 changes files 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:

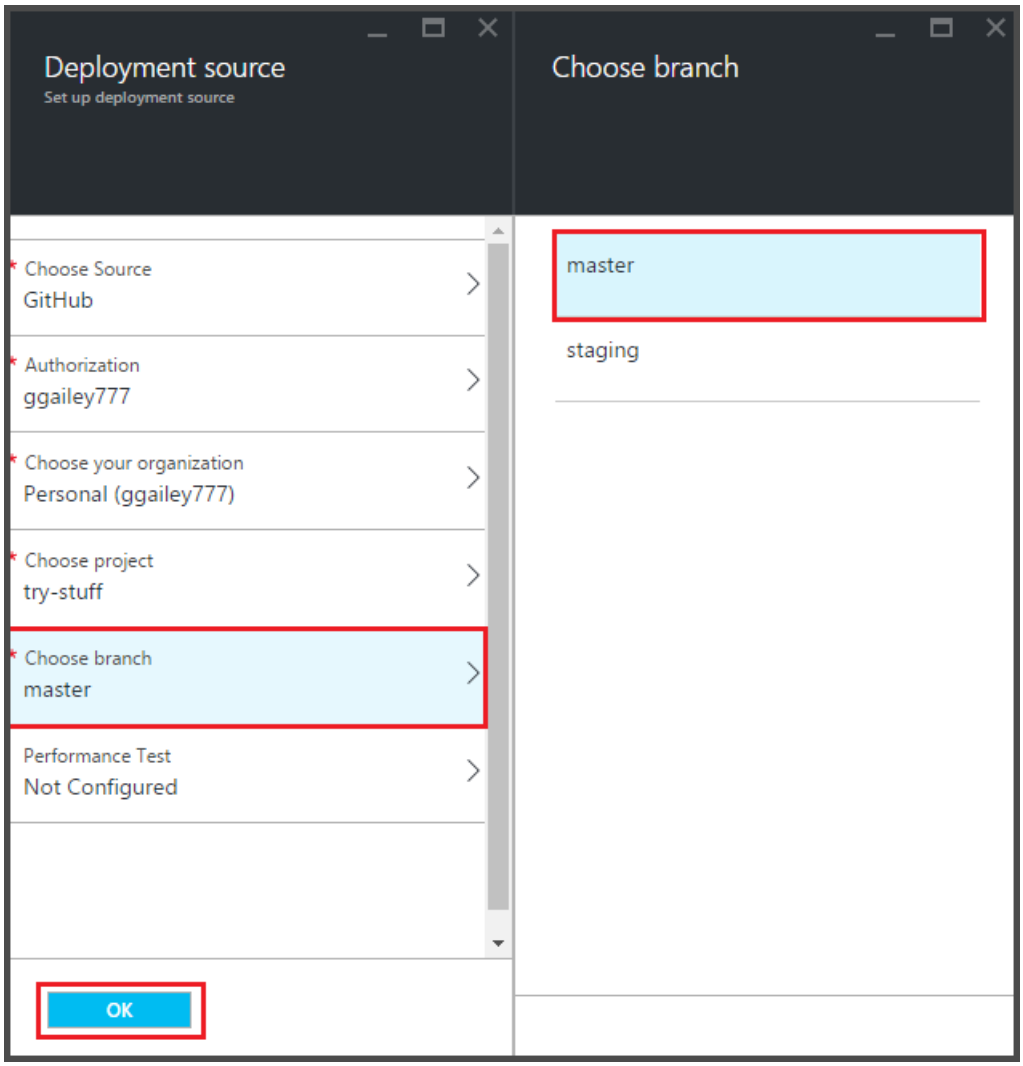
- Create a staging deployment
- Move existing functions to continuous deployment

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 above steps in **Set up continuous deployment** and set the deployment branch to the master branch of your GitHub repo.



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 will trigger 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.

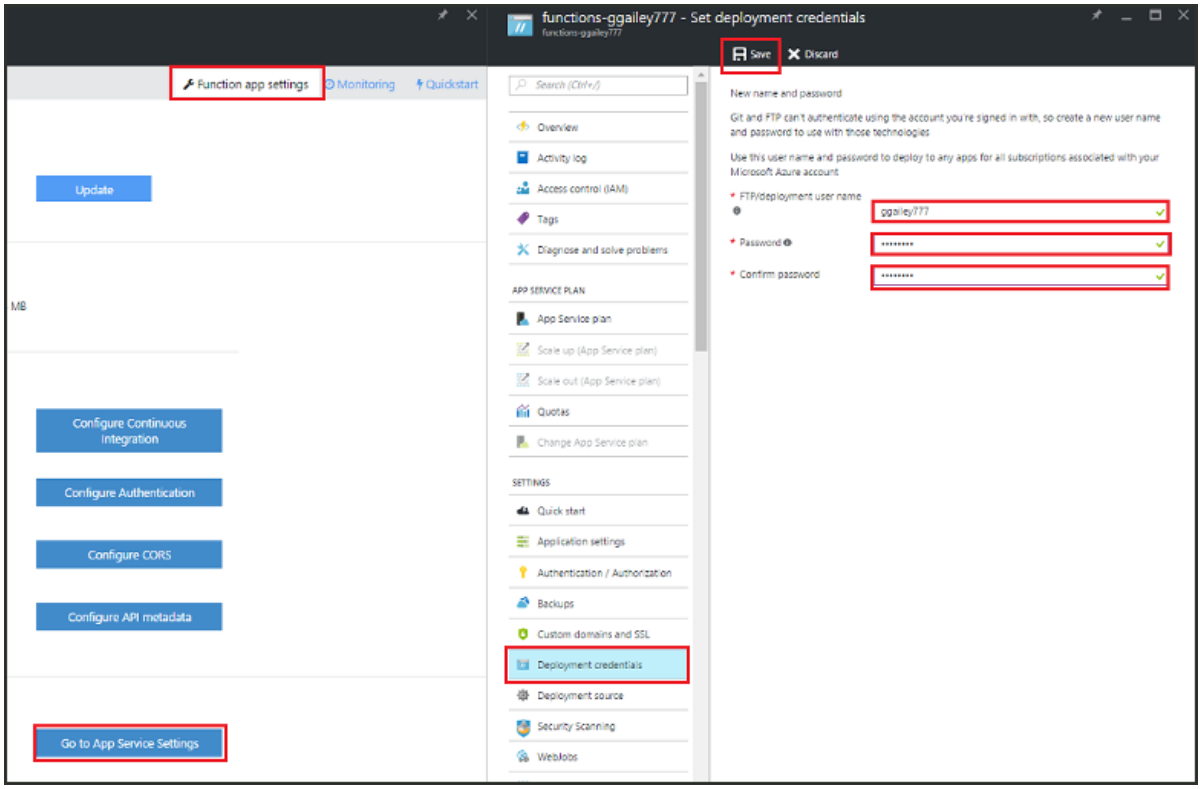
- How to: Configure deployment credentials
- How to: Download files using FTP
- How to: download files using the local Git repository

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, which you can do from the portal. Credentials are set at the Function app level.

1. In your function app in the Azure Functions portal, click **Function app settings** > **Go to App Service settings** > **Deployment credentials**.

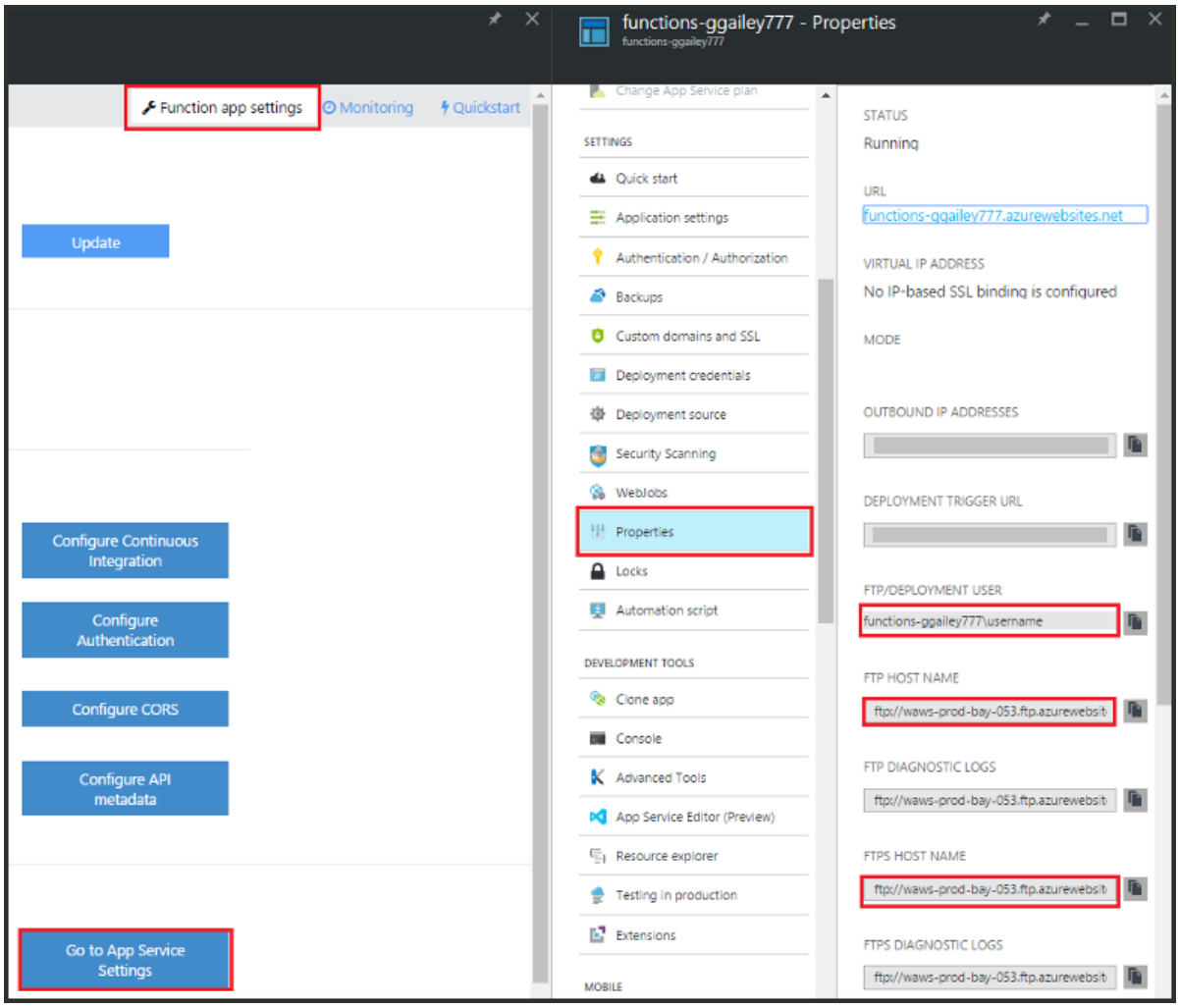


2. Type in a username and password, then click **Save**. You can now use these credentials to access your function app from FTP or the built-in Git repo.

How to: Download files using FTP

1. In your function app in the Azure Functions portal, click **Function app settings** > **Go to App Service settings** > **Properties** and 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, in order to provide proper context for the FTP server.



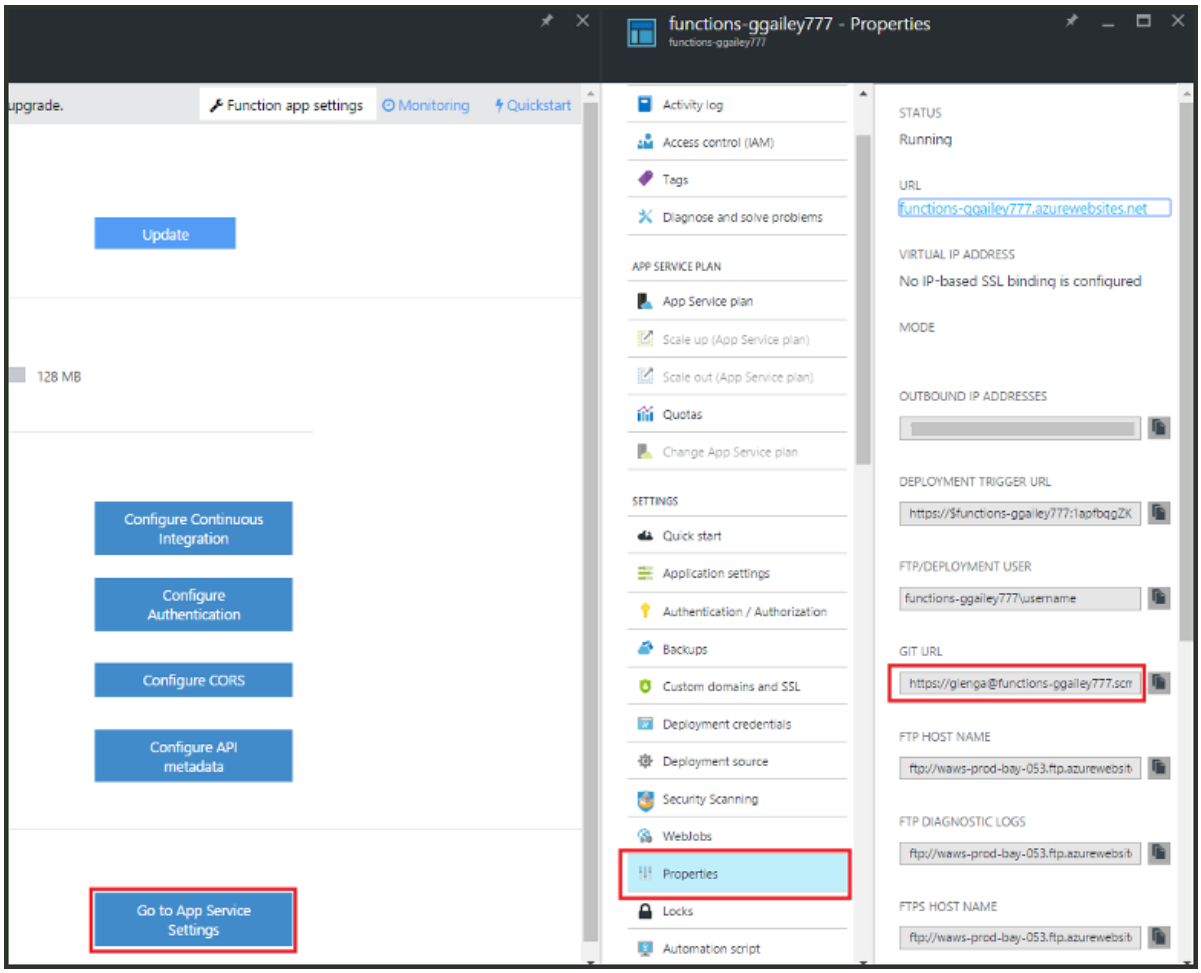
2. From your FTP client, use the connection information you gathered to connect to your app and download the source files for your functions.

How to: download files using the local Git repository

1. In your function app in the Azure Functions portal, click **Function app settings** > **Configure continuous integration** > **Setup**.

2. In the Deployments blade, click **Choose source**, **Local Git repository**, then click **OK**.

3. Click **Go to App Service settings** > **Properties** and note the value of Git URL.



4. Clone the repo on your local machine using a Git-aware command line or your favorite Git tool. The Git clone command looks like the following:

```
git clone https://username@my-function-app.scm.azurewebsites.net:443/my-function-app.git
```

5. 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 the username and password for your function app deployment.

Automate resource deployment for your Azure Functions app

2/7/2017 • 5 min to read • Edit on GitHub

You can use an Azure Resource Manager template to deploy an Azure Functions app. Learn how to define the baseline resources that are required for an Azure Functions app, and the parameters that are specified during deployment. Depending on the triggers and bindings in your Functions app, for a successful Infrastructure as Code configuration of your application, you might need to deploy additional resources.

For more information about creating templates, see Authoring Azure Resource Manager templates.

For examples of complete templates, see Create a Consumption plan-based Azure Functions app and Create an App Service plan-based Azure Functions app.

Required resources

You can use the examples in this article to create a baseline Azure Functions app. You'll need these resources for your app:

- Azure Storage account
- Hosting plan (Consumption plan or Azure App Service plan)
- Functions app (`type`: **Microsoft.Web/Site**, `kind`: **functionapp**)

Parameters

You can use Azure Resource Manager to define parameters for values that you want to specify when your template is deployed. A template's **Parameters** section has all the parameter values. Define parameters for values that vary based either on the project you are deploying, or on the environment you are deploying to.

Variables are useful for values that don't change based on an individual deployment, and for parameters that require transformation before being used in a template (for example, to pass validation rules).

When you define parameters, use the **allowedValues** field to specify which values a user can provide during deployment. To assign a value to the parameter, if no value is provided during deployment, use the **defaultValue** field.

An Azure Resource Manager template uses the following parameters.

appName

The name of the Azure Functions app that you want to create.

```
"appName": {
    "type": "string"
}
```

location

Where to deploy the Functions app.

NOTE

To inherit the location of the Resource Group, or if a parameter value isn't specified during a PowerShell or Azure CLI deployment, use the **defaultValue** parameter. If you deploy your app from the Azure portal, select a value in the **allowedValues** parameter drop-down box.

TIP

For an up-to-date list of regions where you can use Azure Functions, see Products available by region.

```
"location": {
    "type": "string",
    "allowedValues": [
        "Brazil South",
        "East US",
        "East US 2",
        "Central US",
        "North Central US",
        "South Central US",
        "West US",
        "West US 2"
    ],
    "defaultValue": "[resourceGroup().location]"
}
```

sourceCodeRepositoryURL (optional)

```
"sourceCodeRepositoryURL": {
    "type": "string",
    "defaultValue": "",
    "metadata": {
    "description": "Source code repository URL"
}
```

sourceCodeBranch (optional)

```
    "sourceCodeBranch": {
      "type": "string",
      "defaultValue": "master",
      "metadata": {
        "description": "Source code repository branch"
      }
    }
```

sourceCodeManualIntegration (optional)

```
"sourceCodeManualIntegration": {
    "type": "bool",
    "defaultValue": false,
    "metadata": {
        "description": "Use 'true' if you are deploying from the base repo. Use 'false' if you are deploying
from your own fork. If you use 'false', make sure that you have Administrator rights in the repo. If you get an
error, manually add GitHub integration to another web app, to associate a GitHub access token with your Azure
subscription."
    }
}
```

Variables

Azure Resource Manager templates use variables to incorporate parameters, so you can use more specific settings in your template.

In the next example, to meet Azure storage account naming requirements, we use variables to apply Azure Resource Manager template functions to convert the entered **appName** value to lowercase.

```
"variables": {
    "lowerSiteName": "[toLower(parameters('appName'))]",
    "storageAccountName": "[concat(variables('lowerSiteName'))]"
}
```

Resources to deploy

Storage account

An Azure storage account is required for an Azure Functions app.

```
{
    "type": "Microsoft.Storage/storageAccounts",
    "name": "[variables('storageAccountName')]",
    "apiVersion": "2015-05-01-preview",
    "location": "[variables('storageLocation')]",
    "properties": {
        "accountType": "[variables('storageAccountType')]"
    }
}
```

Hosting plan: Consumption vs. App Service

In some scenarios, you might want your functions scaled on-demand by the platform, also called fully managed scaling (by using a Consumption hosting plan). Or, you might choose user-managed scaling for your functions. In user-managed scaling, your functions run 24/7 on dedicated hardware (by using an App Service hosting plan). The number of instances can be set manually or automatically. Your choice of hosting plan might be based on available features in the plan, or on architecting by cost. To learn more about hosting plans, see Scaling Azure Functions.

Consumption plan

```
{
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2015-04-01",
    "name": "[variables('hostingPlanName')]",
    "location": "[resourceGroup().location]",
    "properties": {
        "name": "[variables('hostingPlanName')]",
        "computeMode": "Dynamic",
        "sku": "Dynamic"
    }
}
```

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
    }
}
```

Functions app (a site)

After you've selected a scaling option, create a Functions app. The app is the container that holds all your functions.

A Functions 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 create a successful Infrastructure as Code configuration for 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 using **siteConfig**. It's important to set these configurations at a top level because they convey information to the Azure 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 scenarios, your Functions app and functions need to be deployed *before* **config/appSettings** is applied. In those cases, for example, in 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

- PowerShell
- Azure CLI
- Azure portal
- REST API

Deploy to Azure button

Replace `<url-encoded-path-to-azuredeploy-json>` with a URL-encoded version of the raw path of your `azuredeploy.json` file in GitHub.

Markdown

```
[![Deploy to Azure](http://azuredeploy.net/deploybutton.png)]
(https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-to-azuredeploy-json>)
```

HTML

```
<a href="https://portal.azure.com/#create/Microsoft.Template/uri/<url-encoded-path-to-azuredeploy-json>"
target="_blank"><img src="http://azuredeploy.net/deploybutton.png"></a>
```

Next steps

Learn more about how to develop and configure Azure Functions.

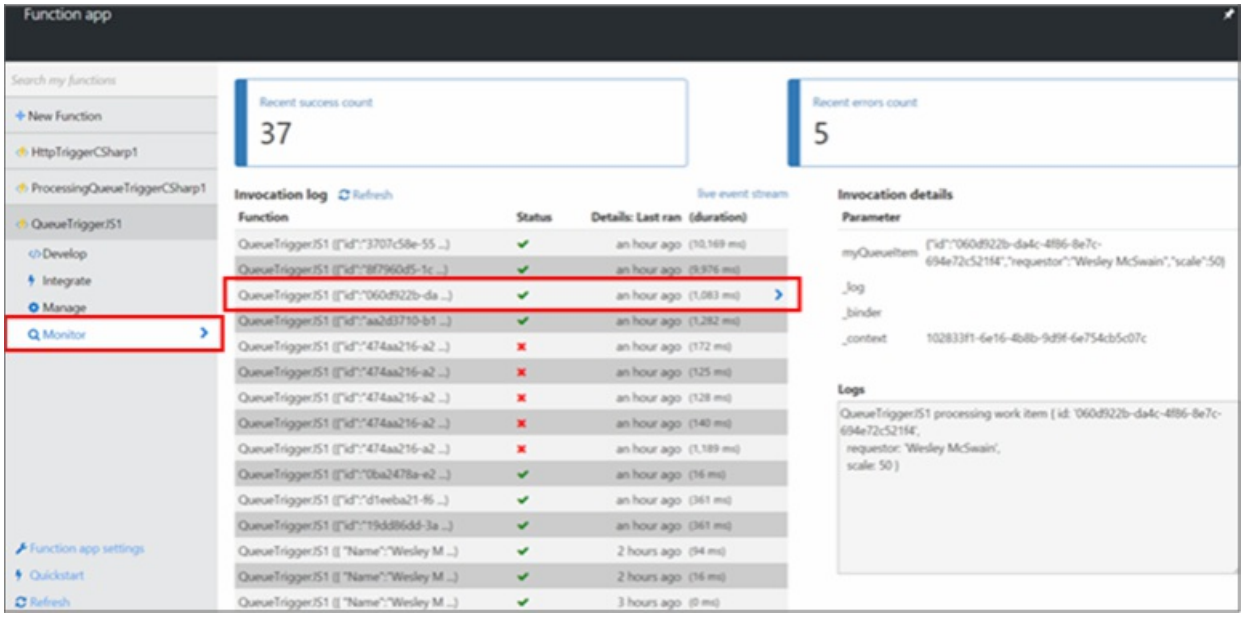
- Azure Functions developer reference
- How to configure Azure Functions app settings
- Create your first Azure function

Monitoring Azure Functions

1/17/2017 • 2 min to read • Edit on GitHub

Overview

The **Monitor** tab for each function allows you to review each execution of a function.

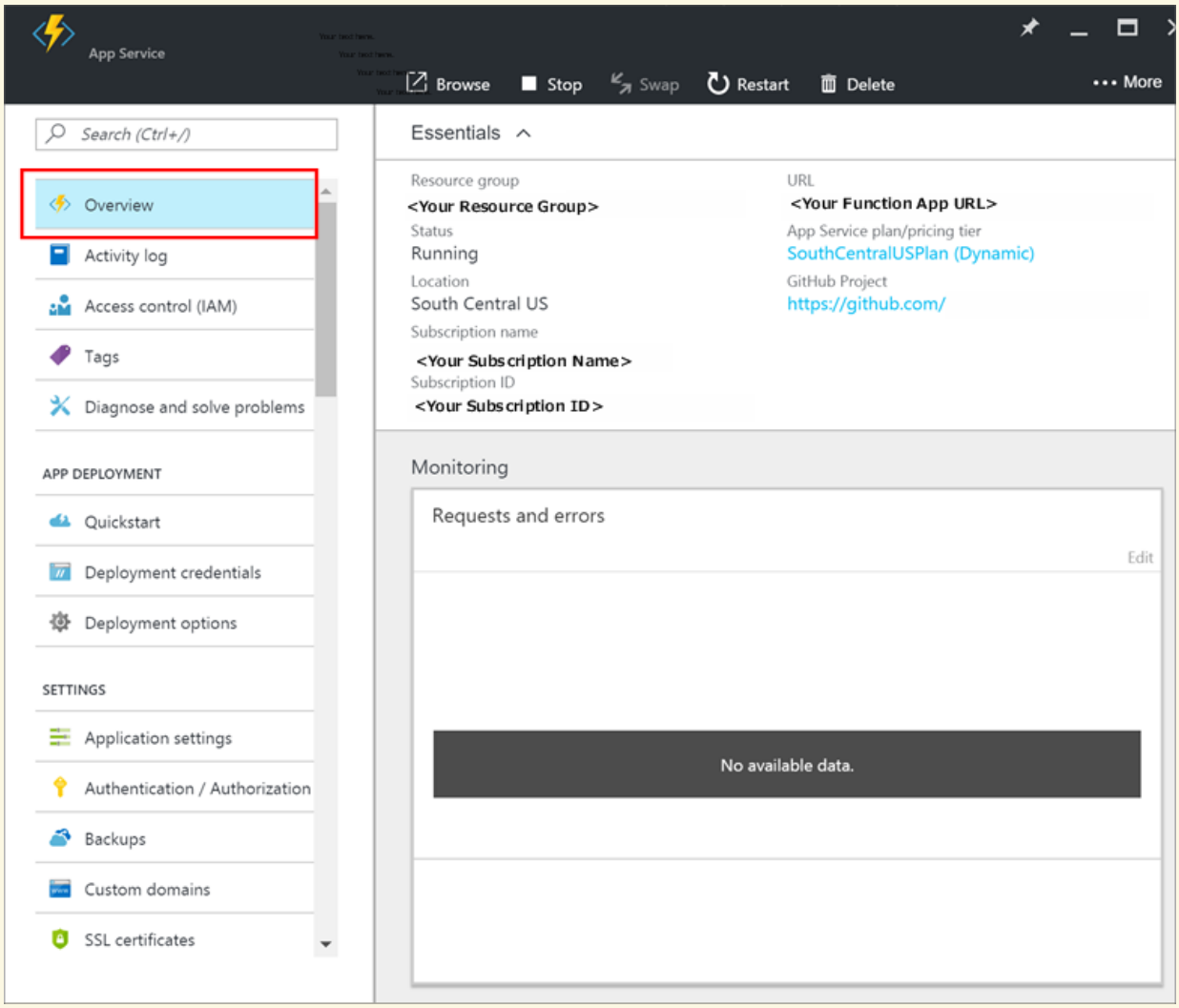


Clicking an execution allows you to review the duration, input data, errors, and associated log files. This is useful debugging and performance tuning your functions.

IMPORTANT

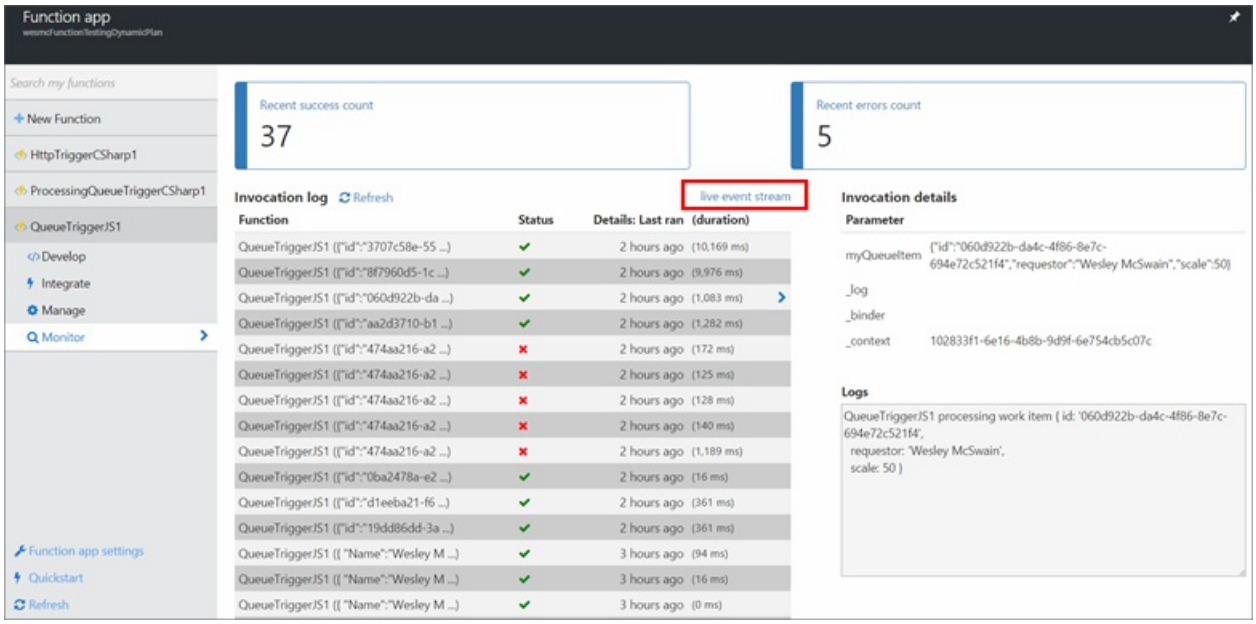
When using the Consumption hosting plan for Azure Functions, the **Monitoring** tile in the Function App overview blade will not show any data. This is because the platform dynamically scales and manages compute instances for you, so these metrics are not meaningful on a Consumption plan. To monitor the usage of your Function Apps, you should instead use the guidance in this article.

The following screen-shot shows an example:

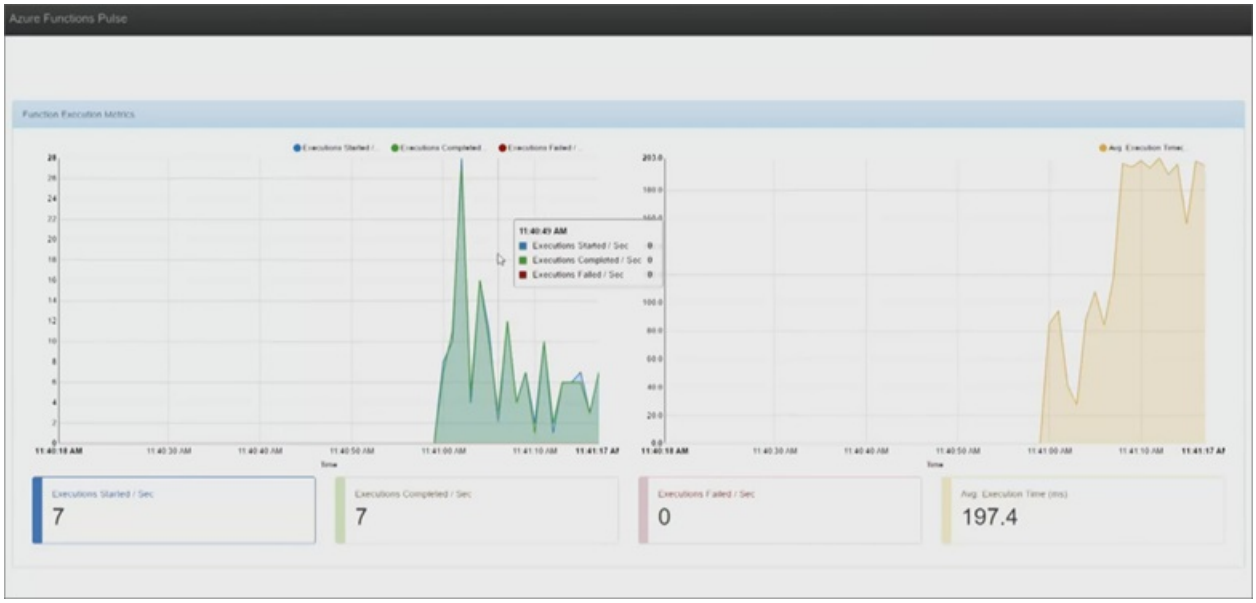


Real-time monitoring

Real-time monitoring is available by clicking **live event stream** as shown below.



The live event stream will be graphed in a new browser tab as shown below.



> **NOTE**
>
> There is a known issue that may cause your data to fail to be populated. If you experience this, you may need to close the browser tab containing the live event stream and then click **live event stream** again to allow it to properly populate your event stream data.

The live event stream will graph the following statistics for your function:

- Executions started per second
- Executions completed per second
- Executions failed per second
- Average execution time in milliseconds.

These statistics are real-time but the actual graphing of the execution data may have around 10 seconds of latency.

Monitoring log files from a command line

You can stream log files to a command line session on a local workstation using the Azure Command Line Interface (CLI) or PowerShell.

Monitoring function app log files with the Azure CLI

To get started, install the Azure CLI

Log into your Azure account using the following command, or any of the other options covered in, Log in to Azure from the Azure CLI.

```
azure login
```

Use the following command to enable Azure CLI Service Management (ASM) mode:.

```
azure config mode asm
```

If you have multiple subscriptions, use the following commands to list your subscriptions and set the current subscription to the subscription that contains your function app.

```
azure account list
azure account set <subscriptionNameOrId>
```

The following command will stream the log files of your function app to the command line:

```
azure site log tail -v <function app name>
```

Monitoring function app log files with PowerShell

To get started, install and configure Azure PowerShell.

Add your Azure account by running the following command:

```
PS C:\> Add-AzureAccount
```

If you have multiple subscriptions, you can list them by name with the following command to see if the correct subscription is the currently selected based on `IsCurrent` property:

```
PS C:\> Get-AzureSubscription
```

If you need to set the active subscription to the one containing your function app, use the following command:

```
PS C:\> Get-AzureSubscription -SubscriptionName "MyFunctionAppSubscription" | Select-AzureSubscription
```

Stream the logs to your PowerShell session with the following command:

```
PS C:\> Get-AzureWebSiteLog -Name MyFunctionApp -Tail
```

For more information refer to How to: Stream logs for web apps.

Next steps

For more information, see the following resources:

- Testing a function
- Scale a function