

AZURE FUNCTIONS

SUCCINCTLY

BY **ED FREITAS**

Azure Functions Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2018 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	6
About the Author	8
Acknowledgments	9
Introduction	10
Chapter 1 Fundamentals	11
Overview	11
Serverless overview	11
Real-world example.....	12
How Azure Functions can help us	13
Typical use cases.....	15
Summary.....	15
Chapter 2 Creating a Function App	17
Quick intro	17
Creating an Azure function app	17
Function app dashboard.....	20
Creating a function	21
Testing a function	25
Azure Storage Explorer	27
Sharing code between functions.....	30
Using shared code	36
Adding a message to the queue	38
Adding a new queue trigger function	43
Testing the queue trigger function	48
The static Run method	50

Summary.....	50
Chapter 3 Metadata, Blob, and Timer Triggers	51
Quick intro	51
Queue metadata.....	51
A blob-triggered function	54
Understanding CRON expressions.....	64
A timer-triggered function	67
Summary.....	73
Chapter 4 Working with HTTP Triggers.....	74
Quick intro	74
Webhooks	74
Generic webhook triggers.....	75
Webhook function URL.....	77
Creating an Azure activity alert.....	78
Updating the function code	82
Testing the webhook function	83
Webhook blob output.....	85
Code summary	89
Summary.....	95

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Face-book to help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas is a consultant on software development applied to customer success, mostly related to financial process automation, accounts payable processing, and data extraction.

He really likes Azure and enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family.

You can reach him at <https://edfreitas.me>.

Acknowledgments

Many thanks to all the people who contributed to this book. All the amazing [Syncfusion](#) team that helped this book become a reality—especially Tres Watkins, Darren West, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Graham High from Syncfusion and [James McCaffrey](#) from [Microsoft Research](#). Thank you all.

This book is dedicated to my mother, father, brother, sister, my nephews, and to *Lala*, whose love and support is always there for me, and to my little princess angel *Mi Chelin*, who inspire me every day and light up my path ahead—God bless you all, always.

Introduction

[Azure Functions](#) is a new and super cool technology from [Microsoft Azure](#) that opens the door to a new style of programming coined *serverless*, which allows developers to move rapidly, scale with ease, and keep costs to a minimum.

At the core of Azure Functions is the concept of events and code. Basically, you supply some code, which is usually a single function written in C# or JavaScript, and you tell Azure Functions which event should trigger the execution of the function.

For instance, you may have an Azure function that is triggered every hour, and what triggers the function is a scheduler. This is particularly useful when you're running a batch process.

Another type of event that can trigger an Azure function is new data becoming available. This could be a new message appearing in a queue, a new file being uploaded to Azure Blob Storage, or an email being sent.

Another example of an event that can trigger a function is a webhook. Whenever someone calls a specific URL, you might then require a specific function to execute; this could be a webhook **POST** callback from an e-commerce provider like [Shopify](#), triggered when a product is sold online.

If you've used Azure or any other cloud provider, you are probably thinking that this can already be done—so why bother with Azure Functions at all? With other Azure services such as Virtual Machines, Cloud Services, Web Apps, and Web Jobs, we can already implement event listeners for all these examples.

The biggest benefit and difference of using Azure Functions is that we don't actually have to worry about all the supporting code that needs to happen before an event is triggered. This is taken care of for us by the Azure Functions framework, so we can simply focus on writing our business logic inside a self-contained function.

Therefore, Azure Functions allows us to build great solutions without worrying about provisioning and maintaining servers, especially when the workload grows.

Azure Functions provides a fully managed computing platform with high reliability and security. It scales on demand, so we get the resources we need when we need them. It's an event-driven and serverless computing platform and experience.

Throughout this book we'll explore the fundamentals of Azure Functions step by step with examples that should be easy to follow. These examples help you easily get acquainted with this fascinating and trending technology.

The figures used in this book were the most recent taken at the time of writing, directly from the Azure Portal. Microsoft regularly updates the Azure UI, so it's possible that some of the screens will look slightly different by the time you read this; nevertheless, you should be able to find your way around very easily.

Chapter 1 Fundamentals

Overview

Azure Functions is one of the latest additions to the Azure App Service family, which offers developers a simplified programming model.

To develop Azure functions, all we must do is write code that responds to the event we're interested in, whether that's an HTTP request or a queue message. All the code we will write connects these events to the code that handles them and is abstracted away from us (handled by the framework).

As we'll see shortly, this provides a very lightweight development platform that is particularly suited for a fast style of development where you just focus on the code that meets your business requirements, eliminating a lot of overhead.

Another great benefit of Azure Functions compared to Virtual Machines or Web Applications is that there is no need to have at least one dedicated server running constantly. Instead, you only pay when your function runs. If your function is listening on a queue and no messages ever arrive, you won't have to pay anything. Isn't that great?

Azure Functions will automatically scale the number of servers running your functions to meet demand. If there's no demand, then there might be no servers running your code. However, the framework can spin one up very quickly when required.

As a result, the pay-as-you-go pricing model can result in dramatic cost savings.

Serverless overview

Microsoft is increasingly using the term “serverless” in their marketing efforts for services like Azure Functions. The term has caught on over the last couple of years, and you can find many cloud vendors using it.

In one sense, this term isn't truly accurate—there are, of course, servers required to run the Azure Functions framework and the code that runs on it. However, one of the key ideas behind serverless is that you basically delegate the overall management and maintenance of your servers to a third-party provider—in this case, Microsoft Azure—so that you can focus purely on business requirements.

In a typical serverless architecture, you would choose to rely on a third-party platform or back-end service—for instance, by using Azure Cosmos DB as your NoSQL cloud database—instead of creating your own virtual machine and installing a database server on it.

The great thing about serverless architecture is that there is a growing number of third-party services that meet many of the common needs of modern cloud applications, whether it's

logging, sending emails, performing full-text search of documents, or accepting online payments.

These third-party services cannot do everything on their own, so there will be a need to write some of your own back-end code. This is where services like Azure Functions fit nicely into serverless.

Instead of provisioning a whole server to run a website or host some background processes, you simply tell Azure Functions which events you need to respond to and what you want to do when those events are triggered, and then you let the framework worry about how many servers you need. This lightweight model is also referred to as function as a service (FaaS).

It's important to note that on its own, Azure Functions isn't a serverless framework, but it fulfills the FaaS part of serverless. Therefore, it can be used in conjunction with other cloud offerings and services to create a full serverless architecture.

Let's consider a very simple real-world example.

Real-world example

Some time ago, I needed to create a relatively simple website to sell a software utility I'd made, which was called Pluglock.

The whole idea behind this tool was that it would use your laptop power cable as a security device. If the utility was running in secure mode and the power cable was unplugged, the tool would sound an alarm, block the volume, lock the session, and prevent anyone except the authorized Windows user from logging back in. After the authorized user logged back in, the computer would be restored to a "safe" state, turn off the alarm, and let the user work again.

I created the tool for myself, as I used to travel quite a bit and found it annoying that every time I was in an airport and needed to run to the toilet, I had to pack my laptop and take it with me. So I thought to myself, why not create a tool that could protect my laptop while I went to the toilet, leaving it plugged in?

Imagine what a surprise someone with thoughts of unplugging my laptop would run into and be exposed to the eyes of everyone else, alerted by an alarm at full volume—not a pleasant scene for that person!

The website was simply static HTML, and it had a Buy button that integrated with a payment provider (which isn't that complicated, technically speaking).

However, I also needed to handle the webhook callback from the payment provider whenever someone bought the tool. So, what I did was add a web method API to my server to handle that, which needed to generate a license file and send an email in order to provide that information to the customer.

This required the web server to post messages to queues. Then, I needed to add code that would listen to these queues, which ended up on the web server as well. This resulted in more and more code being added to the web server.

Furthermore, the tool also allowed the user to opt-in and report certain things, such as location and tracking information (IP address, latitude, and longitude). This called a webhook, and it also needed to check whether the license was valid.

Finally, a process that ran overnight would summarize all the activity and send a detailed report via email to me.

So what started out to be a very simple website with a Buy button had grown into a full-blown back-end web application with a lot of responsibilities.

There was a point in time when I was thinking of moving from static HTML to something like WordPress, which would make the editing of the website content easier. However, it was soon clear that it wasn't possible to migrate it to WordPress because there was already so much back-end .NET code that it would require a complete rewrite in PHP, just so WordPress would be able to support the same functionality I had developed.

This type of scenario probably sounds very familiar to many developers out there—you end up with one server doing everything as a monolith, which is incredibly hard to scale.

The figure that follows represents the process I have just described.

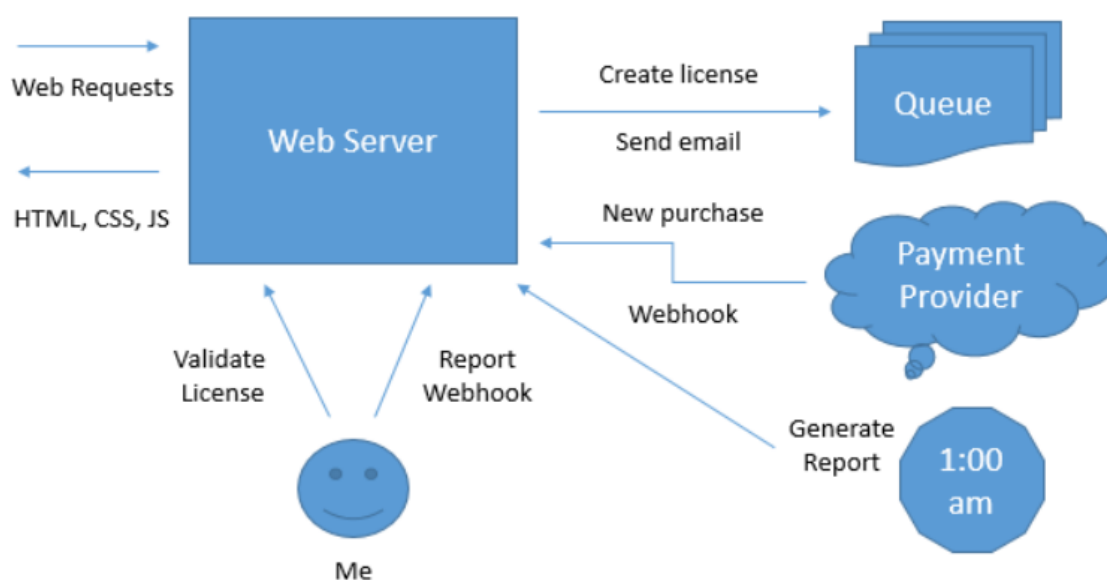


Figure 1-a: Monolithic Architecture

How Azure Functions can help us

In the real-world example I just described, we witnessed what is known as a *monolithic architecture*—which obviously doesn't scale well at all.

Let's see how a serverless architecture and Azure Functions could help us. What we can do is break up some or most of the parts of our monolithic architecture into smaller bits.

Let's say the webhook for new purchases is going to be handled by an Azure function that posts a message into a queue, and then that message is handled by another function that generates the license file and puts it into a blob.

That blob then creates triggers for another Azure function that emails the license to the customer. As you might have guessed, the reporting webhook is another Azure function that writes a row into Azure Table storage.

The license validation API is another Azure function that performs a database lookup, as well as the nightly process that produces a report.

We still have our web server, which could even be replaced with some static content hosted in blob storage. However, what we've managed to do is decompose our monolithic architecture into a set of loosely coupled functions, which you could even think of as nanoservices.

These functions could be wrapped up nicely into a single Azure Functions app, given that they belong together and share configuration settings and local resources.

Here's how our monolithic architecture would look using Azure Functions instead.

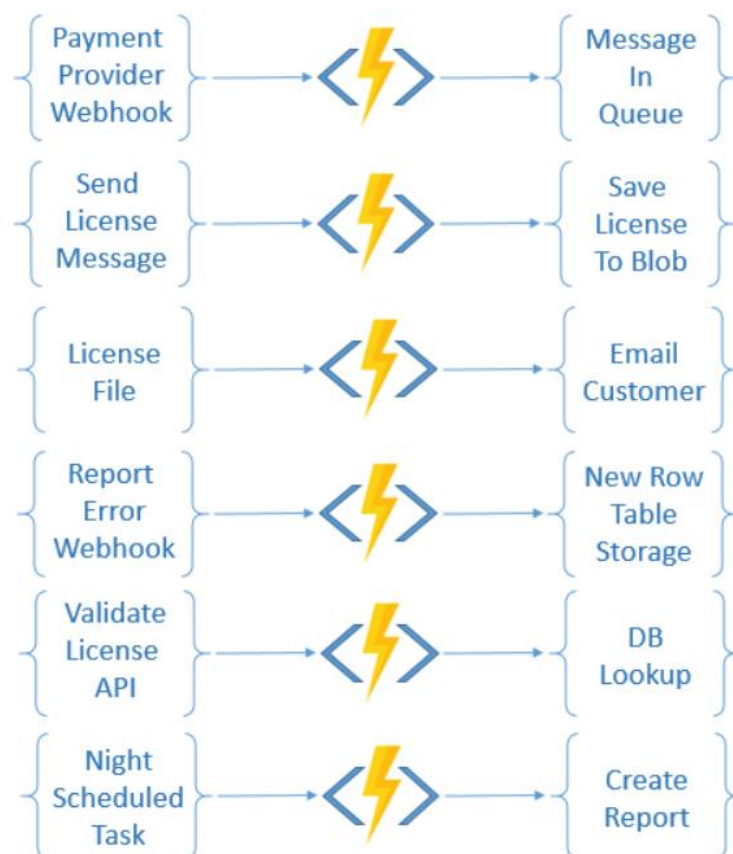


Figure 1-b: Serverless Architecture Using Azure Functions

As you can see, this is much easier to scale than the monolith we previously described, given that each of the steps in the process is a self-contained Azure Function.

Typical use cases

Now that we've seen how using Azure Functions can simplify an app's architecture, what sort of applications are Azure Functions and serverless good for?

Azure Functions is not necessarily right for every scenario, but it makes a lot of sense and adds value in specific cases, such as:

- **Experimentation and rapid prototyping:** Since it only takes a few minutes to get up and running using Azure Functions, it is possible to have a back end for a fully functioning prototype in no time.
- **Automating development processes:** Using Azure Functions is a very handy way of automating some of your internal development processes. A good example is that many software teams use Slack for communication, and it might be convenient to integrate it with your build server so that notifications can be sent to team members when a build occurs in case of errors or exceptions. Given that Slack has a rich ecosystem and extensibility model, all you need is a webhook that it can call to integrate it with Azure Functions.
- **Decomposing and extending applications:** Just like we saw previously, Azure Functions is ideal for simplifying monolithic systems.
- **Independent scaling:** Azure Functions is great if you have a number of queue handlers that all need different scaling requirements. If you are managing all the servers yourself, then this can suddenly turn into a headache. However, if you're using Azure Functions, you simply hand that problem over to the framework and let it solve the problem for you.
- **Integrating systems:** One of the most exciting and useful features of Azure Functions is the ability to integrate various systems. Sometimes you might need to create an intermediate adapter that connects two systems together—for instance, Twitter and Dropbox. Using Azure Functions is a great way to do that.
- **Going serverless:** Finally, it's all about not having to worry about managing servers and infrastructure, so using a series of loosely coupled Azure Functions that communicate together is a great way to simplify a back end.

Summary

We've covered a bit of ground in this quick introductory chapter by explaining the fundamental aspects of Azure Functions and providing an overview of a serverless architecture.

Serverless describes a style of programming where the servers, scaling, and even the wiring of the events are transparently managed for you, so you can focus purely on writing the code you need to solve your business needs.

An Azure function is nothing more than a small piece of code that responds to an event, so in other words: *Azure function = Event + Code*.

The Azure Functions framework is built on top of other solid Azure products, like Azure Web Apps and WebJobs.

One of the great things about Azure Functions is its flexibility—it doesn't force you to go all-in on a serverless architecture. Instead, it can be used alongside a more traditional architecture—and let's not forget the ability to do rapid prototyping.

In the next chapter, we'll be creating our first function application, which will allow us to group various Azure Functions together, and dive straight into code. Finally, some fun!

Chapter 2 Creating a Function App

Quick intro

Before closing the previous chapter, I promised that we were going to dive straight into the fun part, which is writing code. However, before we do that, let's quickly set the stage for what we will be doing throughout this chapter.

Assuming you have already signed up for an [Azure subscription](#)—which is very easy to do, and you even [get free credits from Microsoft](#) to begin with—we'll start off by creating our first Azure function app.

We'll check out some of the most relevant options for what we'll be doing, available on the Azure Functions section of the Azure Portal.

To understand how this works, we'll have a quick look behind the scenes using a handy tool provided by Azure and see what's inside the storage and queues that Azure provides.

Creating an Azure function app

To create an Azure function app, first make sure you're logged in to the Azure Portal, and then click **Create a resource**. Next, click **Serverless Function App**, as shown in the following figure.

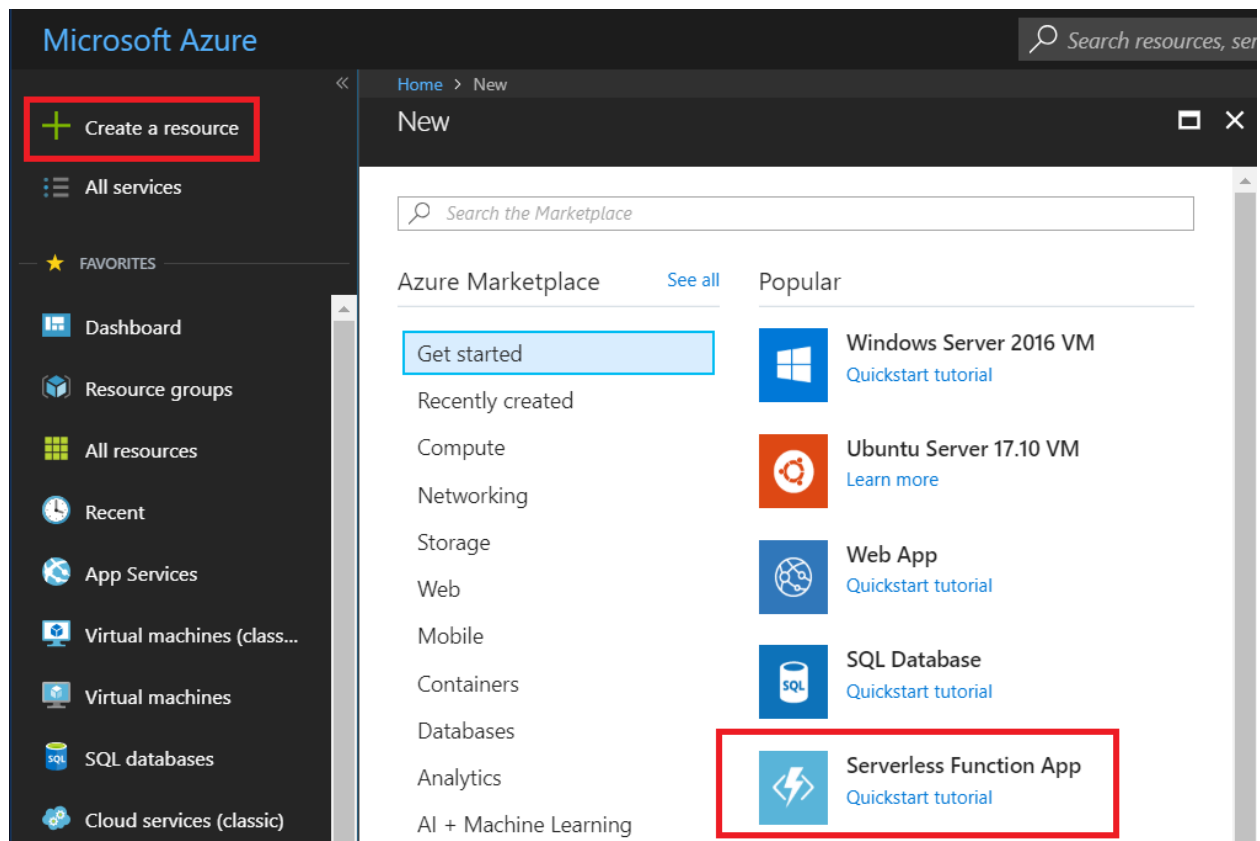


Figure 2-a: The Serverless Function App Resource on Azure

Notice that there's a **Quickstart tutorial** link just below the **Serverless Function App** link. You can click the tutorial link to access the documentation, which is always an excellent resource to have at hand.

Once you've clicked the **Serverless Function App** link, you'll be presented with the following screen, where you can enter the required details to create the function app.

Microsoft Azure

Home > New > Function App

Function App

Create

* App name
FunctionAppSuccinctly ✓
.azurewebsites.net

* Subscription
Visual Studio Dev Essentials

* Resource Group ⓘ
☒ Create new ☐ Use existing
FunctionAppSuccinctly ✓

* OS
Windows Linux (Preview)
Docker

* Hosting Plan ⓘ
Consumption Plan

* Location
Central US

* Storage ⓘ
☒ Create new ☐ Use existing
functionappsuccabfe

☒ Pin to dashboard

Create Automation options

Figure 2-b: Creating a Function App on Azure

You'll need to enter an **App name** that is unique. The Azure Portal will indicate whether or not the name has already been used. You'll see a green check mark if the name is available, or an X if it is not.

You'll also need to specify a **Resource Group**. You can create a new one or use an existing one.

There's also an option to select the **OS** type. I usually choose Windows, as it includes both a .NET and JavaScript runtime stack, which means that you can either choose to write your Azure function app using C# or JavaScript.

As for the **Hosting Plan**, you can choose from the **Consumption Plan** or use the **App Service Plan**. The former lets you pay per execution, while the latter allocates resources based on your app's predefined capacity with predictable costs and scale.

In my case, I'll be using this Azure function app just for demos, so I've selected the **Consumption Plan**.

You also need to choose a **Location**. I usually choose **East US** or **Central US** as just a personal preference, but you can choose any you wish, one that is closest to your actual physical location, or one that best serves the needs of your product or service (closest to where most of your customers or users might be, for example).

It's also necessary to select a **Storage** option. In my case, I've chosen the **Create new** option.

Finally, I've chosen to turn the **Application Insights** option off. Once all these options and requirements have been selected, click the **Create** button to finalize the creation of the Azure function app.

Function app dashboard

Now that our function app has been created, let's have a quick look at what Azure has prepared for us, which can be seen in the following figure.

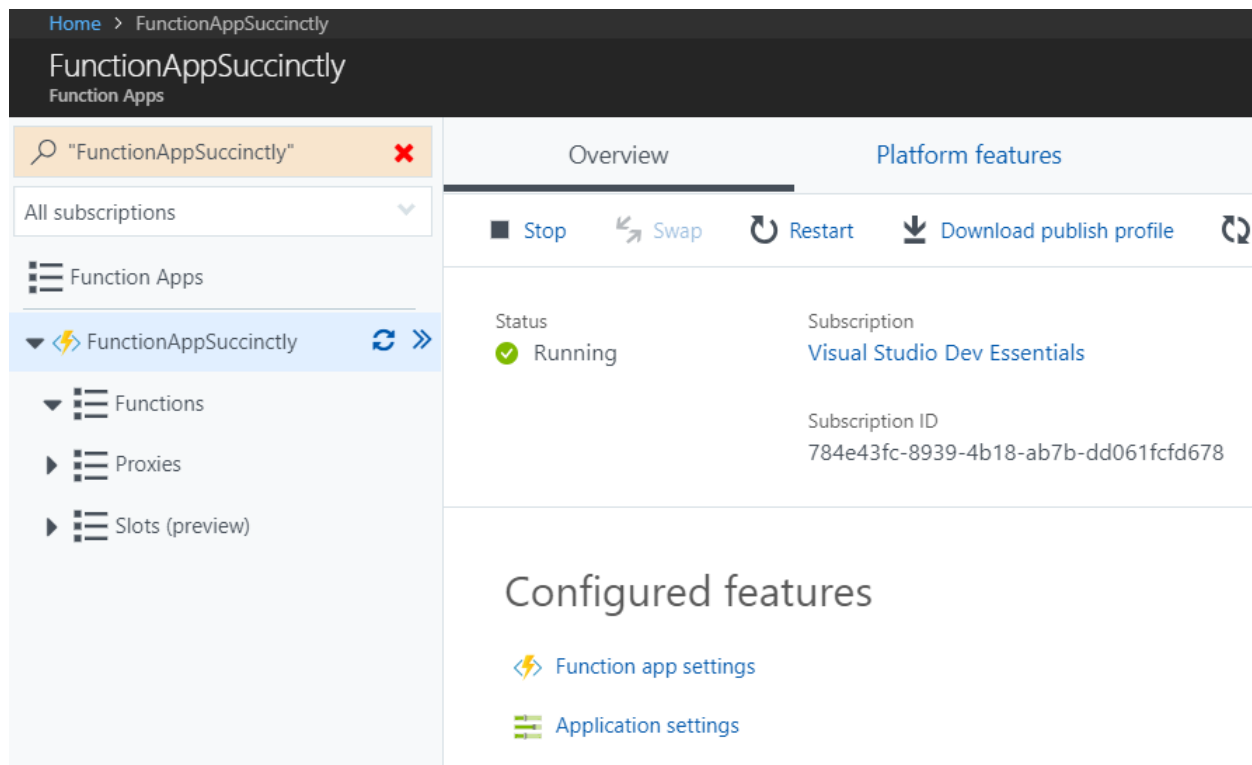


Figure 2-c: The Newly Created Function App

We won't go through all the features that this Azure Functions app dashboard contains. The Azure Portal user interface is frequently updated, and most of the items and labels are self-descriptive and easy to understand. Instead, we'll just focus our attention on the crucial items and the ones we need as we progress.

It's important to understand that from the moment the Azure function app is created, it's essentially active, meaning that it can execute code, but this doesn't mean that it is incurring costs.

On the dashboard, it is possible to start, stop, and restart the Azure function as well as set a broad range of options and properties—from application settings, to deployment options, authentication, and more.

After an Azure function app has been created, by default it is optimized to use the best possible settings given the original requirements you defined. That means there's no need to stress about tuning your app options. They're probably as good as they can already be.

What is important is how to start using it, and to do that, we need to create our first function.

Creating a function

To create a new function, all we need to do is click the **+** button next to **Functions**, as you can see in the following figure.

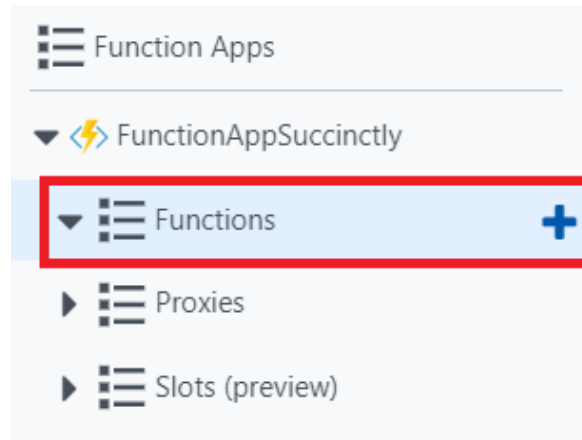


Figure 2-d: Creating a New Function

Once you've clicked the + button, you'll see a screen that is self-explanatory and easy to navigate. Let's have a look.

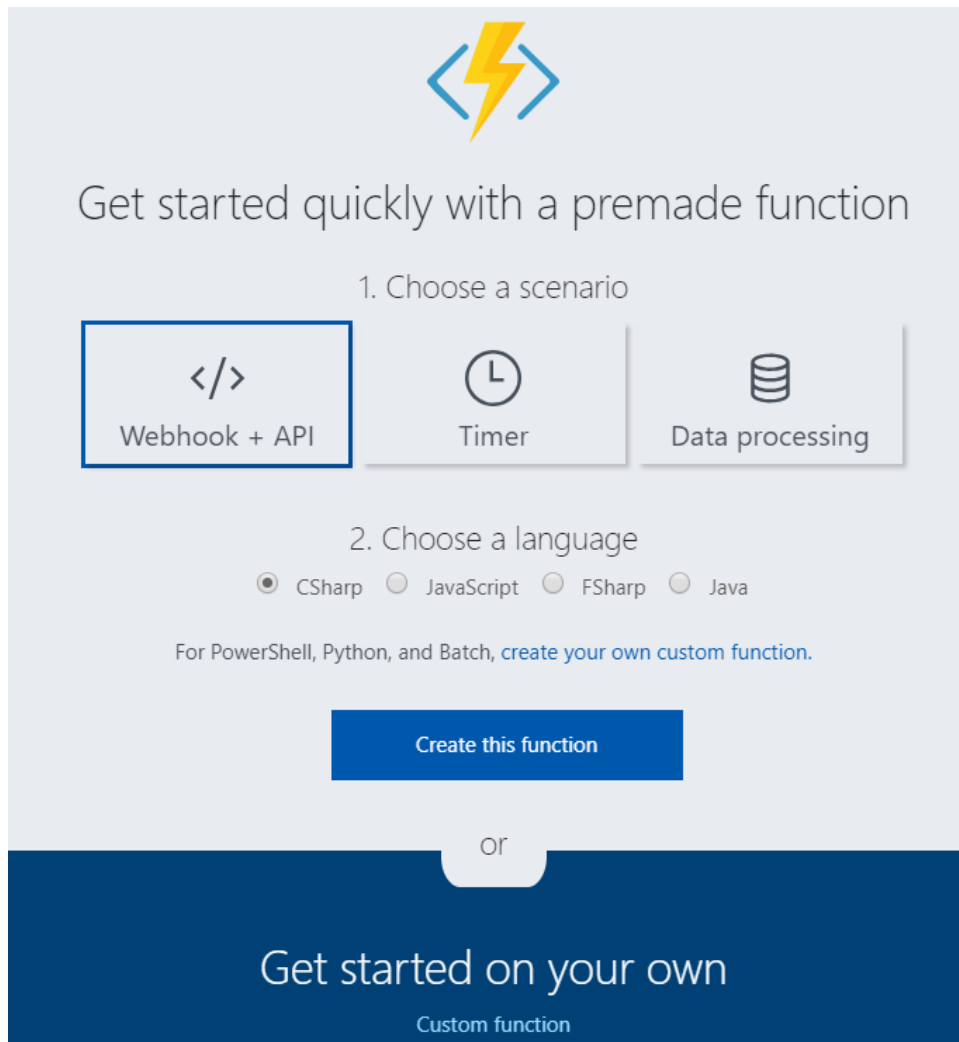


Figure 2-e: Functions—Getting Started Screen

In a screen that looks like the one shown in Figure 2-e, we can easily choose the type of function that we want to create. Let's start off by creating a simple webhook using C# (the default option). So, go ahead and click **Create this function**.

Once we have done that, we get the code shown in Listing 2-a.

Listing 2-a: The Default Webhook Azure Function Code

```
using System.Net;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,
TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    // parse query parameter
    string name = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
        .Value;

    if (name == null)
    {
        // Get request body
        dynamic data = await req.Content.ReadAsAsync<object>();
        name = data?.name;
    }

    return name == null
        ? req.CreateResponse(HttpStatusCode.BadRequest,
            "Please pass a name on the query string or in the request body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
}
```

Here's how it looks on the Azure Portal.



Figure 2-f: The Default Webhook Azure Function Code

What does this code do? Let's explore it briefly by dissecting each of the main parts of this function.

The first part of the Azure function is responsible for parsing the query parameters that are passed to the function itself.

Listing 2-b: Parsing the Query Parameters

```
// parse query parameter
string name = req.GetQueryNameValuePairs()
    .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
    .Value;
```

Basically, **req** is an instance of the **HttpRequestMessage** class. The **GetQueryNameValuePairs** method is executed, which is responsible for getting the parameters passed to the Azure function.

Then using LINQ, the value of the first parameter **name** is returned if it is found within the list of parameters.

In short, what this piece of code does is retrieve the value of the **name** parameter, which is passed to the Azure function.

The next part is shown in the following code snippet.

Listing 2-c: Parsing the Request Body

```
if (name == null)
{
    // Get request body
    dynamic data = await req.Content.ReadAsAsync<object>();
    name = data?.name;
}
```

All this really does is retrieve the request body content by invoking the **req.Content.ReadAsAsync** method and assign the result to a **dynamic** variable. This variable is then used to obtain the **name** property (in case **data** is not **null**), thus the syntax **data?.name** (where the **?** indicates that the object can be **null**).

Within the two previous snippets of code—the one that parses the query parameters, and the one that gets the body request—the code is doing the same thing, which is returning the value of a parameter called **name**.

Why do that at all? Surely it would be sufficient to just inspect the query parameters or the request body, right? The thing is, webhooks don't often only pass along query parameters or a request body—they might pass both. This is the reason why both options are used within the Azure function.

We now come to the last bit of the Azure function, which simply executes the **req.CreateResponse** method in order to create an HTTP response with the value of the **name** parameter.

Although it might appear that a lot happens inside this function, not much is going on besides reading the value of a **name** parameter (either from the function's query parameters or from the request body), which is sometimes referred to as the *payload* of the webhook, and then simply returning a response with the value **read**.

All this predefined function gives us is some boilerplate code with certain guidelines that we can use to write our own code.

Now that we've created our first function and explored what it does, let's have a look at how we can test it using the Azure Portal.

Testing a function

To test the function that we just created within the Azure Portal, all we really need to do is open the **Test** pane on the right-hand side of the screen to check what parameters we can pass on to the function, and then click the **Run** button at the top.

Here's where we can find the Test pane, as shown in the following figure.

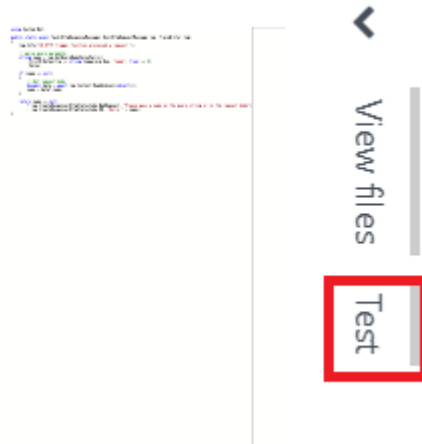


Figure 2-g: The Test Pane Tab

Here's where we can find the **Run** button. See the following screenshot.



Figure 2-h: The Run Button

Before clicking **Run**, let's open the **Test** pane to see what we have available for testing.

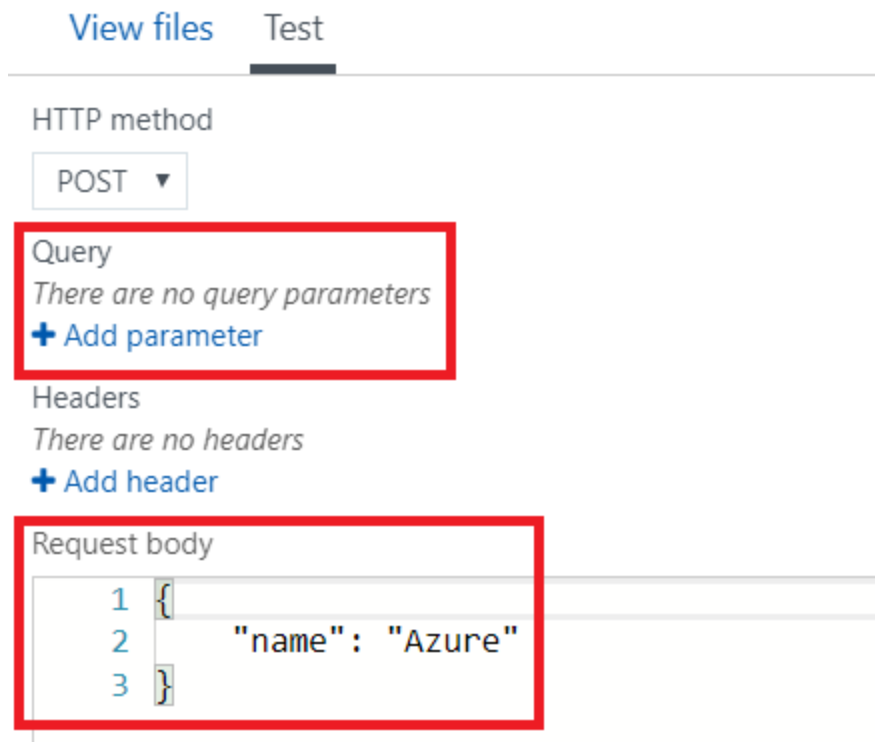


Figure 2-i: The Opened Test Panel

As you can see, I've highlighted the **Query** parameters section and the **Request body** section.

By default, the Azure Portal provides a JSON object with the **name** parameter and a value that can be used for testing.

To quickly test the function, all we need to do is click **Run**. After doing that, we can see the following result on the Test pane.

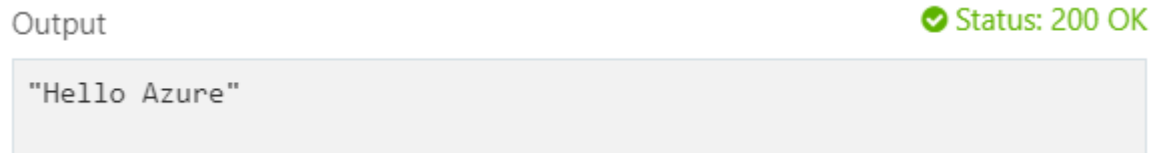


Figure 2-j: Testing Results

As we can see, the function has returned the correct string based on the input **name** parameter provided to the function through the request body.

The great thing about the Test pane is that we can also test our function by passing query parameters and header values.

Now that we've briefly seen how to test an Azure function, let's explore how we can create queues and blobs by using the Azure Storage Explorer.

Azure Storage Explorer

One very useful tool when working with Azure Functions is the Microsoft Azure Storage desktop application, which is available for [free](#).

So, go ahead and download it. Once that's done, double-click the executable, and you'll see an installation screen as follows.

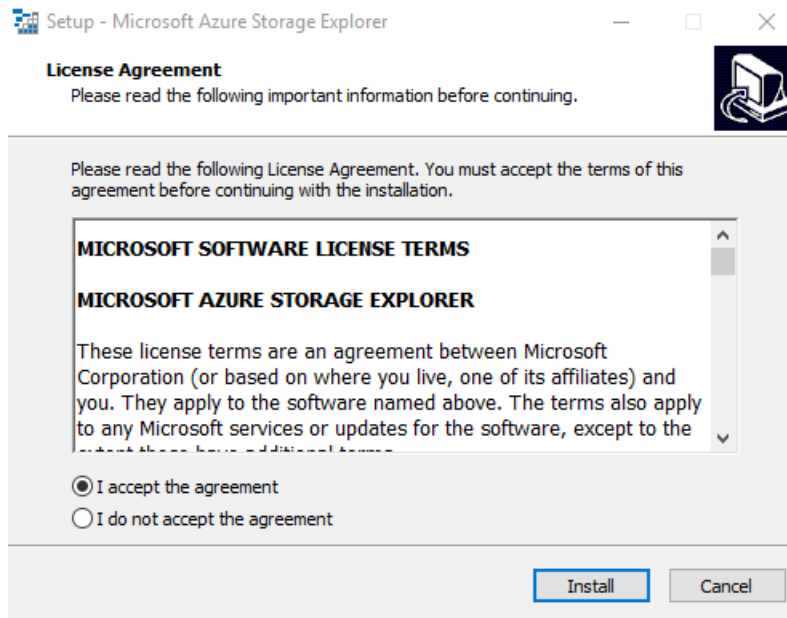


Figure 2-k: Azure Storage Explorer Installer

The installation itself is very simple and straightforward, so there should be no difficulty installing this tool. At the end of the installation process, you'll be asked to launch the application.

When you run the application for the first time, you'll be asked to sign in with your Microsoft account. Once you've done that, you'll see the following.

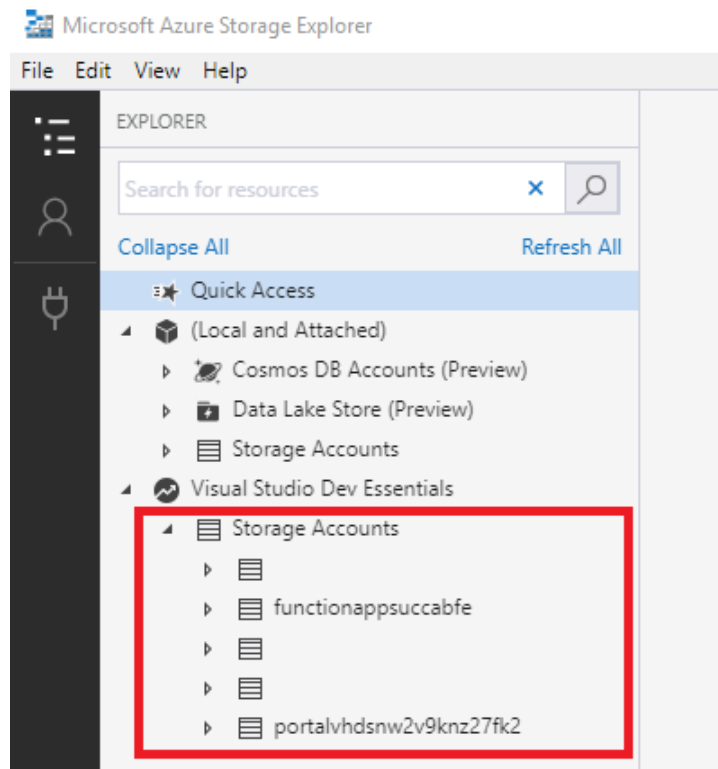


Figure 2-l: Storage Accounts in Storage Explorer

As you can see, these are all the Azure Storage accounts that exist within my Azure subscription. It might look different on your system.

When we created our Azure Functions application, a new storage account was automatically deployed as well, which in my case corresponds to the item **functionappsuccabfe** as shown in the previous screenshot. (The trailing “abfe” in the identifier represents four hex values, and it's just a coincidence in my case that all four digits were letters).

Let's expand the storage account to see what we have available.

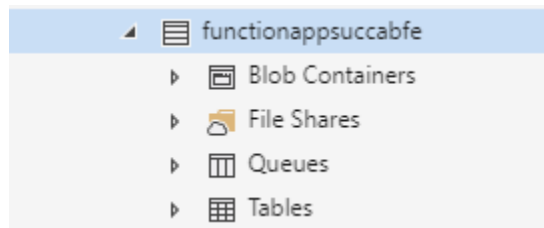


Figure 2-m: Items of an Azure Storage Account

We can see that the storage account is divided into various storage types: blob containers, file shares, queues, and tables.

We will use a couple of blob containers and a single queue for the examples that follow.

Let's create the items that we need, starting with a new blob container. We can do this by simply right-clicking **Blob Containers** and choosing **Create Blob Container**. Next, give it a name. I'll call mine **hello-requests**.

Once the blob container has been created, we can go into the pane on the right and manipulate the blobs within the blob container. For example, we can upload files into the blob storage or examine existing items in the blob container.

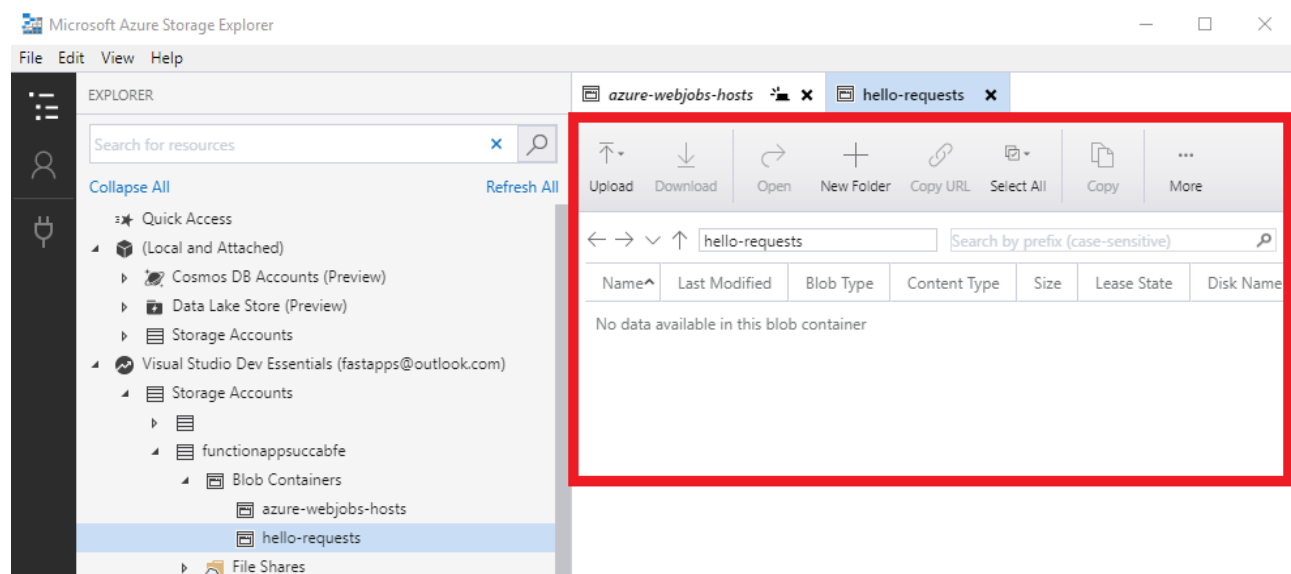


Figure 2-n: The Newly Created Blob Container

With our first blob created, let's repeat the same steps to add a second blob, which I'm going to call **receipts** (we'll use this later).

Let's go ahead and create a queue. We can do this by right-clicking **Queues** and naming it **hello-creation-requests**. We should now see the following items.

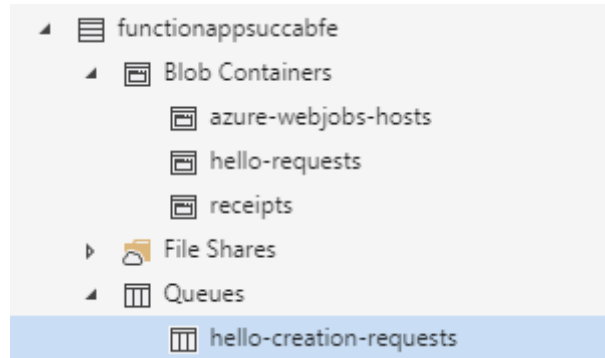


Figure 2-o: Items Created—Storage Account

Just like with blob containers, we can use the right pane within the Azure Storage Explorer application to add new items to the queue and to delete queues.

Sharing code between functions

Now that we've seen how to use the Azure Storage Explorer to create blob and queue containers, which is a fundamental aspect of interacting with Azure Functions, it's also important that we look at how we can share code between various functions within the same Azure Functions application.

In the examples that follow, we are going to be sharing a few classes between all the functions of our application. The best way to do this is to create a shared code function that doesn't execute, but instead, hosts the code that we want to share with the rest of the functions within the Azure Functions application.

So, going back to the Azure Portal, open the Azure Functions application we previously created and create a new function by clicking the **+** button, as highlighted in the following figure.

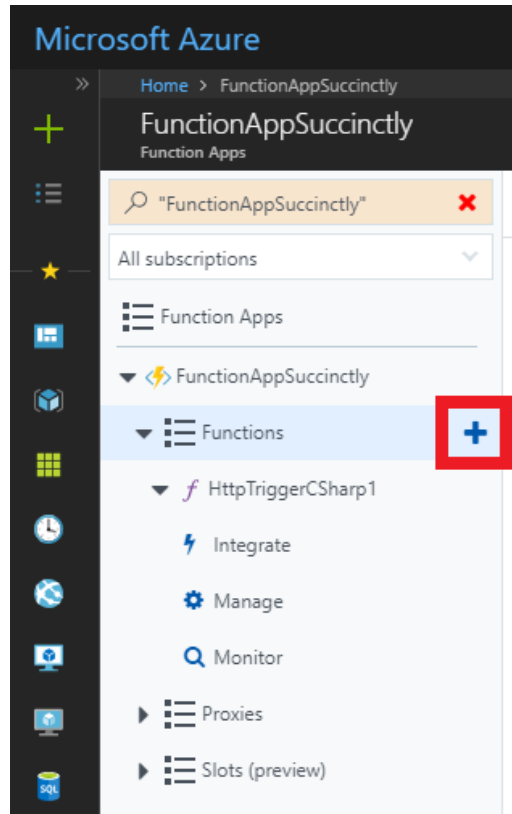


Figure 2-p: Create a New Function

This time we are going to select **Manual trigger** from the list of available function templates. You might need to scroll down through the list of available function templates to find it.

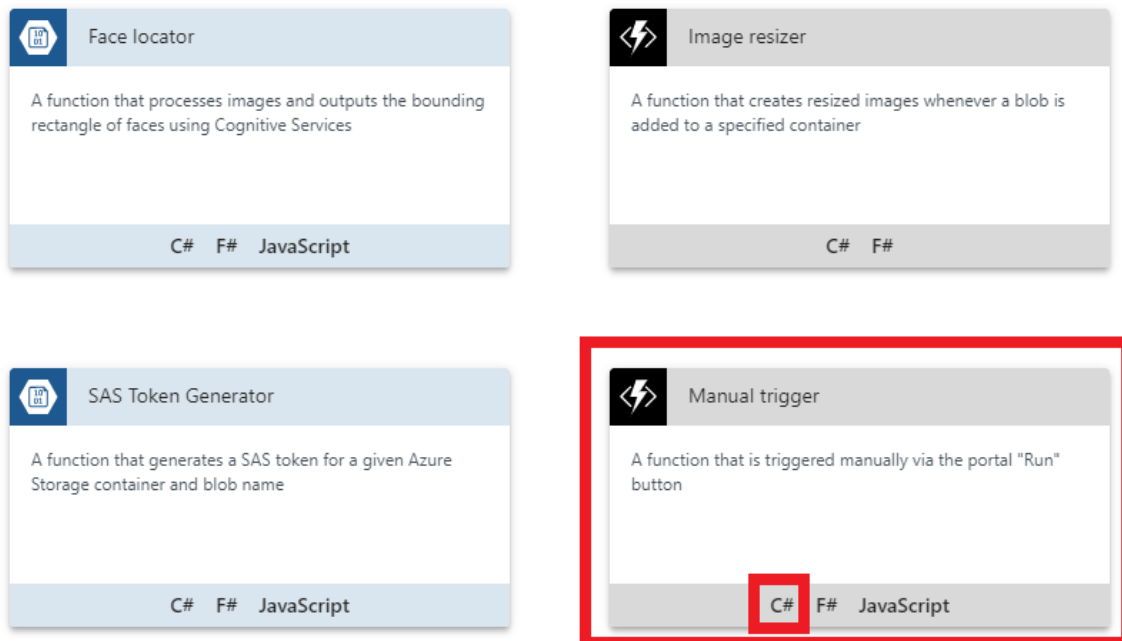
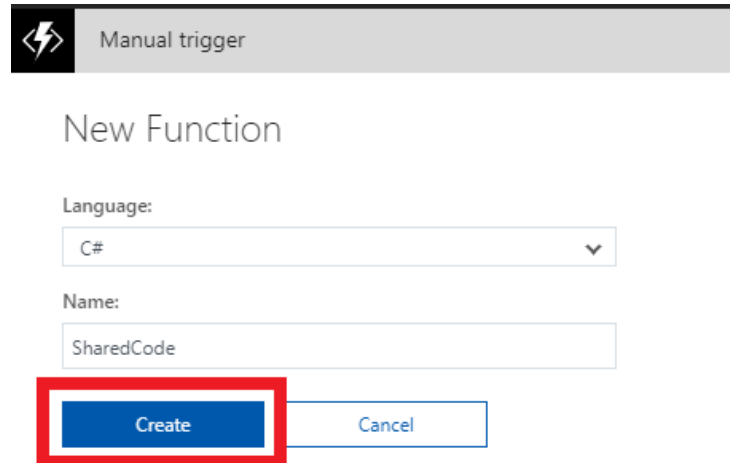


Figure 2-q: The Manual Trigger Function Template

You can create the function according to your programming language preference. In my case, I'll click the **C#** option.

Once you've clicked the option, you'll be asked to give the function a name, as shown in Figure 2-r. I've called the function **SharedCode**. After naming it, click **Create** to create the function.



The screenshot shows the 'New Function' dialog in the Azure Portal. At the top, there's a tab labeled 'Manual trigger' with a lightning bolt icon. Below this, the title 'New Function' is displayed. There are two input fields: 'Language:' with a dropdown menu showing 'C#' and a small downward arrow, and 'Name:' with a text box containing 'SharedCode'. At the bottom, there are two buttons: 'Create' (highlighted with a red rectangle) and 'Cancel'.

Figure 2-r: New Function Dialog

In principle, we now have a container that will host our shared code. By default, the following code has been added to the function by the Azure Portal.

Listing 2-d: Default Manual Trigger Function Code

```
using System;

public static void Run(string input, TraceWriter log)
{
    log.Info(
        $"C# manually triggered function called with input: {input}");
}
```

With this in place, click the **View files** tab on the right-hand side of the screen. Now, let's add our first class by clicking **Add** and name our file **CreateHelloRequest.csx** (where the .csx extension indicates a C# script, introduced with the release of the Roslyn compiler).

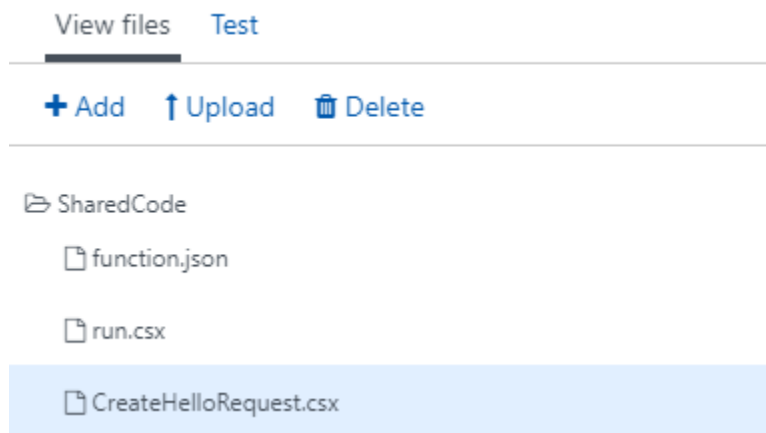


Figure 2-s: CreateHelloRequest.csx under View files

In the middle of the Azure Portal screen, you should see a blank area where we can add some code to **CreateHelloRequest.csx**.

I'll go ahead and add the following boilerplate code to save you some time.

Listing 2-e: CreateHelloRequest Code

```
public class CreateHelloRequest
{
    public string Number;
    public string FirstName;

    public override string ToString() => $"{FirstName} {Number}";
}
```

This class represents a request to create a greeting for a given phone number and a given first name. This will ultimately result in a text message being sent to the number with a hello message using the first name.

Before doing anything else, make sure you click **Save** to update **CreateHelloRequest.csx**.

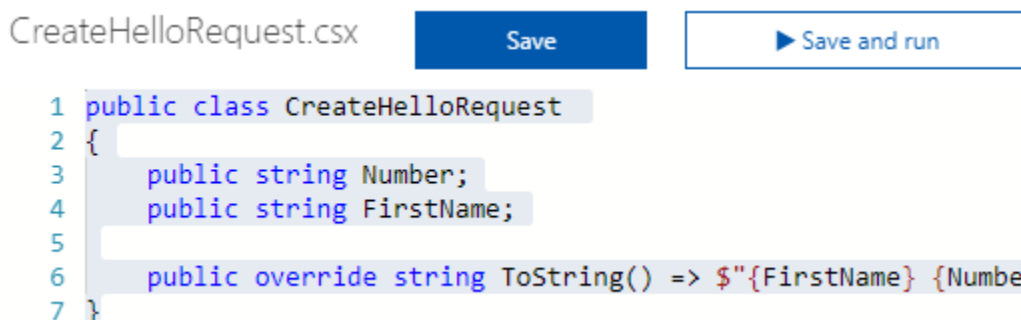


Figure 2-t: The CreateHelloRequest Code

Let's go back to the **View files** pane to create a new file called **HelloRequest.csx** and paste the following code. Make sure you save the file.

Listing 2-f: HelloRequest.csx Code

```
public class HelloRequest
{
    public string Number;
    public string Message;

    public override string ToString() => $"{Number} {Message}";
}
```

Here we've overwritten the **ToString** method to return the **Number** to which we want to send the request and the **Message** we want to send.

Next, on the **View files** pane, let's create another file called **MsgSentConfirmation.csx** and paste the following code. Don't forget to save the file.

Listing 2-g: MsgSentConfirmation.csx Code

```
public class MsgSentConfirmation
{
    public string Number;
    public string Message;
    public string ReceiptId;

    public override string ToString() =>
        $"{ReceiptId} {Number} {Message}";
}
```

This class is going to act as an audit log and confirm that a message was sent to a specific **Number** with a specific **Message**, and we'll also keep the **ReceiptId** of the message gateway. What we are doing here is simulating an SMS gateway.

With this sharing code approach, it's important to tell our Azure Functions app to monitor the **SharedCode** folder so that anytime changes occur to classes within this folder, any function that uses these classes can be also updated.

In order to do this, we need to edit the **host.json** file within the **App Service Editor**. This can be found under **Platform features**, as shown in the following screenshot.

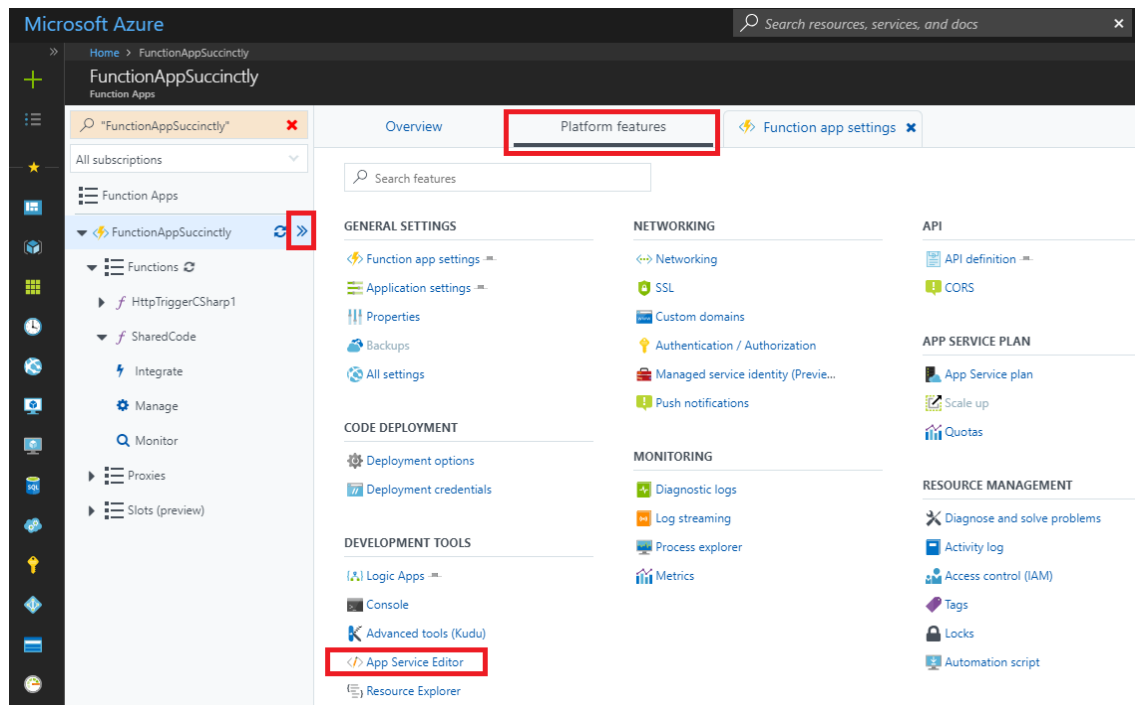


Figure 2-u: Function Apps Platform Features

Once you've clicked the **App Service Editor** option, a new tab should open in your browser, and you'll be presented with a screen like the following one. Here you'll be able to find the **host.json** file under the **SharedCode** item, as shown in the following figure.

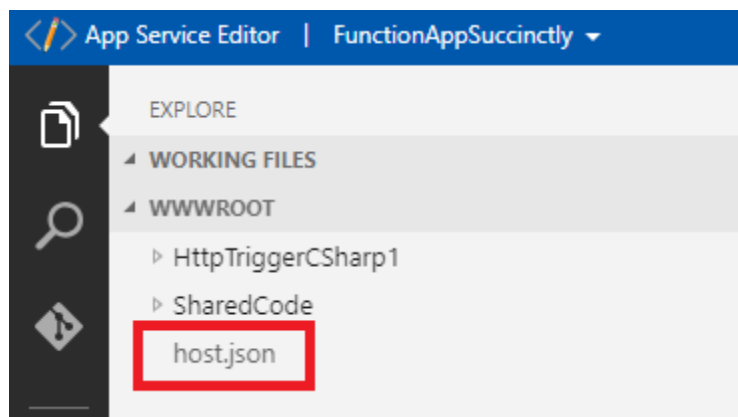


Figure 2-v: App Service Editor

Once you've clicked the **host.json** file, you'll be presented with an editor. Let's go ahead and update this file. There is a setting we need to add to monitor the **SharedCode** folder, as follows.

Listing 2-h: The Updated host.json File

```
{
  "watchDirectories": ["SharedCode"]
}
```

Notice how the **host.json** file is automatically saved after you add the entry.



Figure 2-w: The host.json File Automatically Saved

The interesting thing about the **watchDirectories** setting is that if there is any change on any of the files contained within the **SharedCode** folder, the functions will get recompiled, which is also cool and handy.

With this done, let's head back to the original tab where we have opened the Azure Portal and create a new manually triggered function. We'll use this to reference the shared code function we just created.

Using shared code

Just like we did for our shared code function, let's go ahead and create a new function by using the **Manual trigger** function template.

I'll call this function **ManualTrigger**, but feel free to use any other name. As its name suggests, you can execute a manually triggered function from within the Azure Portal by clicking the **Run** button.

With the function created, let's go ahead and modify the default code that was provided out of the box.

The first thing we are going to do is add a reference to one of the shared classes from the shared code function we previously created. To do this, we can use the special **#load** directive, and between quotation marks, indicate the name of the file that we want to load.

We can achieve this by modifying **run.csx** as follows.

Listing 2-i: Adding a Shared Code Class

```
#load "..\SharedCode\CreateHelloRequest.csx"  
  
using System;  
  
public static void Run (string input, TraceWriter log)  
{  
    log.Info(  
        $"C# manually triggered function called with input: {input}");  
}
```

Notice how we need to go up one folder level by using two periods right after the first quotation mark on the file path, and then finally go down into the **SharedCode** folder to load the **CreateHelloRequest.csx** file.

Now that we have a reference to the **CreateHelloRequest** class, let's make the necessary modifications to put this functionality to work and test it out.

To achieve this, the second modification we need to apply to the previous code is change the **string** parameter of the **Run** method with a **CreateHelloRequest** parameter instead.

Listing 2-j: Adding a CreateHelloRequest Parameter

```
#load "..\SharedCode\CreateHelloRequest.csx"

using System;

public static void Run (CreateHelloRequest input, TraceWriter log)
{
    log.Info(
        $"C# manually triggered function called with input: {input}");
}
```

If we now click **Save** and add a JSON object to test with, passing the parameters required by the **CreateHelloRequest** class, we get the following results.

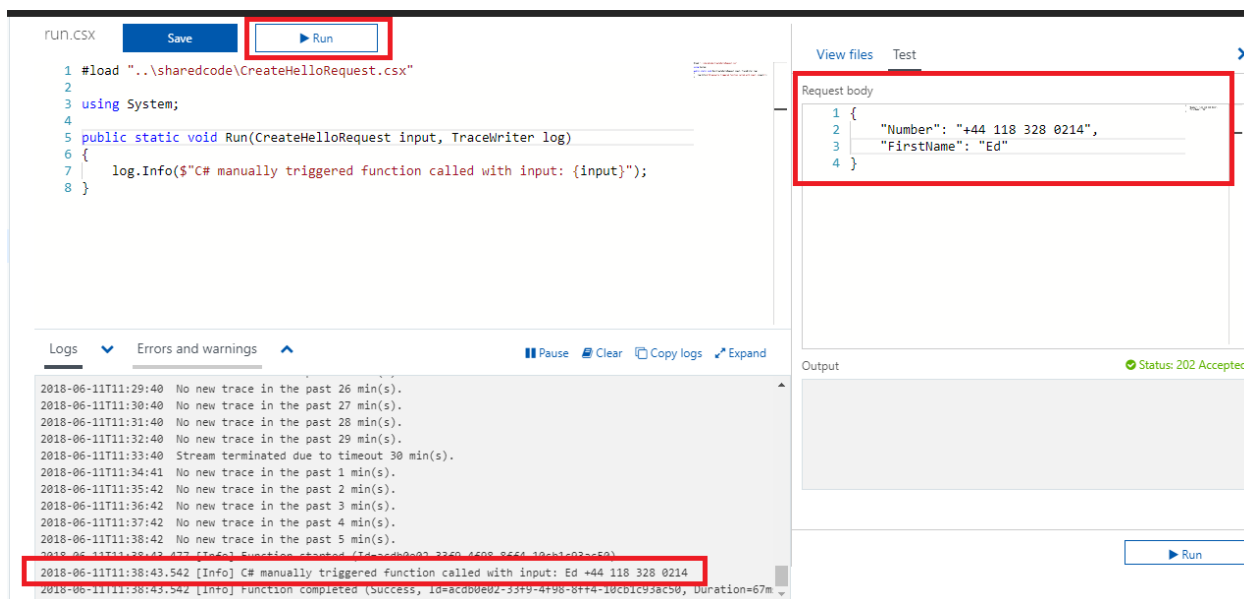


Figure 2-x: Testing with the CreateHelloRequest Parameters

As you can see, this works—nothing extraordinary, as the **Run** method is simply printing out the values passed through the **CreateHelloRequest** class. The interesting thing is that we were able to easily use a class that has been defined in another Azure function by simply adding a reference to it and passing the class name as a parameter.

Adding a message to the queue

In the real world, what we've just done—simply printing out values to the console—is not particularly useful, although it is valuable for testing.

What we really should aim to achieve is to do something with the data we receive through the **Run** method. When we run this function, we want to add a new message to the queue that we created previously using Azure Storage Explorer.

To do this, we need to go over to the **Integrate** option under the name of our Azure function and add a new output of type **Azure Queue Storage**, as follows.

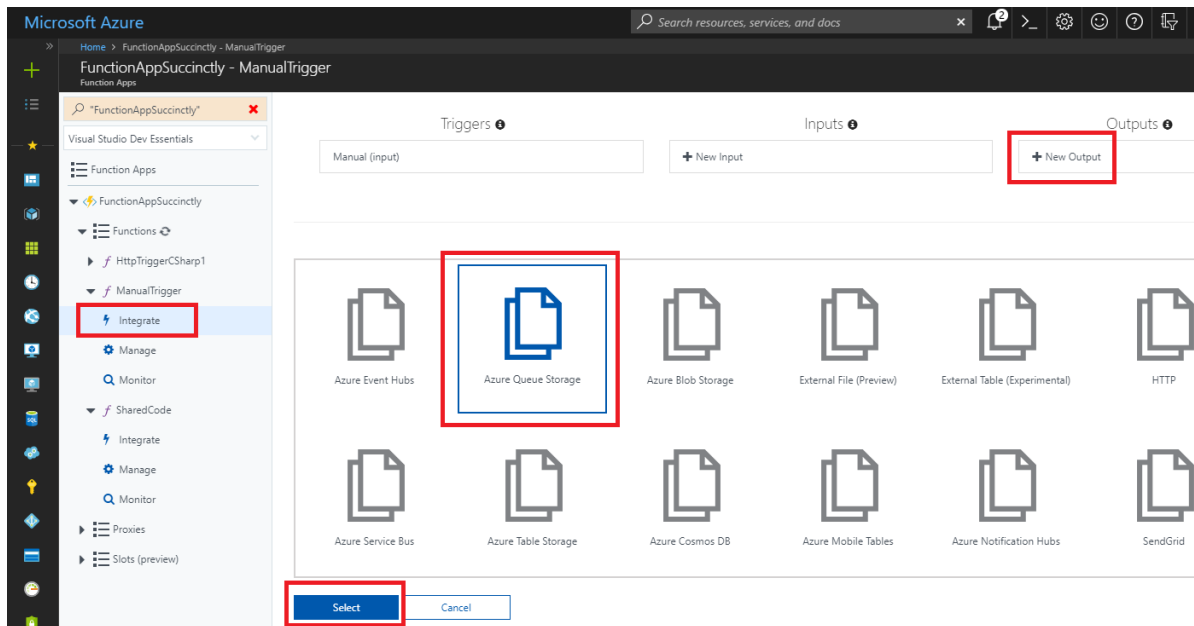


Figure 2-y: Adding an Azure Queue Storage as an Output Type (Step 1)

Once the **Azure Queue Storage** option has been selected, Azure asks us to add some additional details to finish configuring our queue storage. After clicking **Select**, you'll see the following screen.

Azure Queue Storage output

Message parameter name ⓘ
outputQueueItem

☐ Use function return value

Storage account connection ⓘ show value
AzureWebJobsDashboard new

Save Cancel

Queue name ⓘ
hello-creation-requests

Figure 2-z: Adding an Azure Queue Storage as an Output Type (Step 2)

We can leave the **Message parameter name** with the default value; what we are really interested in changing is the **Queue name**. Here we need to use the same name we used when we created our queue using Azure Storage Explorer, which in my case was **hello-creation-requests**, as shown in Figure 2-z.

We also need to create a new **Storage account connection**. We can do this by clicking the **new** link next to the **Storage Account connection** drop-down list, as shown in Figure 2-z. After that, we'll be asked to select the storage account we want to use.

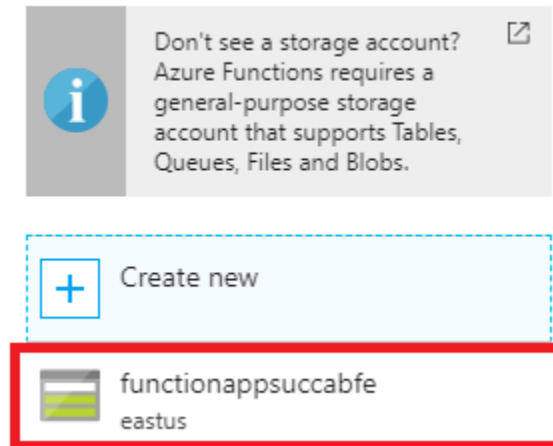


Figure 2-aa: Selecting a Storage Account

Once the storage account has been selected, it will appear as the selected option under **Storage account connection**. Select the storage account and click **Save**. The account connection will automatically have **_STORAGE** added to the end of its name.



Figure 2-ab: Save the Azure Queue Storage Output

The **Azure Queue Storage output** will be shown at the top of the **Outputs** list, as highlighted in the following screenshot.

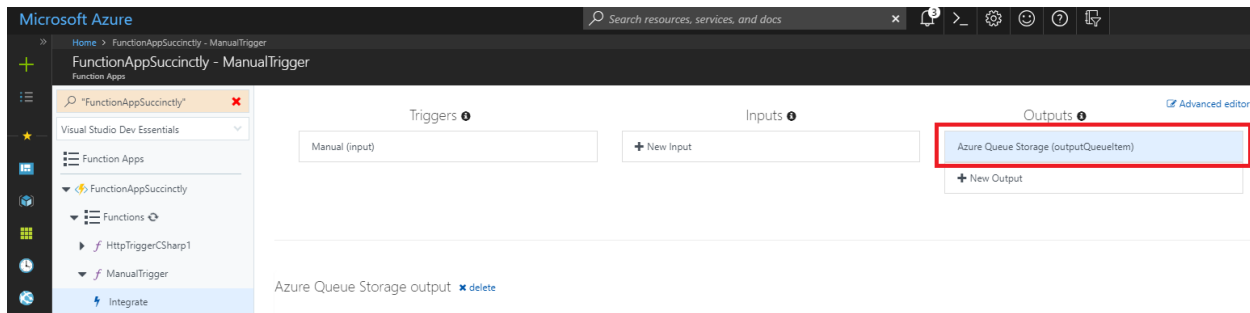


Figure 2-ac: Output List

Now that we have the output defined to an Azure Queue Storage, we need to go back to our code and modify the **Run** method. We need to add an extra parameter of type **CreateHelloRequest**, as shown in the following code listing.

Listing 2-k: Adding a *CreateHelloRequest* Parameter

```
#load "..\SharedCode\CreateHelloRequest.csx"

using System;

public static void Run (CreateHelloRequest input, TraceWriter log,
                        out CreateHelloRequest outputQueueItem)
{
    log.Info(
        $"C# manually triggered function called with input: {input}");
    outputQueueItem = input;
}
```

Notice how we named the new **out** parameter **outputQueueItem** to match the binding we recently created (see Figure 2-z).

Also notice that we've had to assign to **outputQueueItem** the value of **input**; otherwise, we would get a compilation error if we were to run this method. This is because **outputQueueItem** has been defined as an **out** parameter, which means that a value must be assigned to it before the method finalizes.

If we now run this method and then open the Azure Storage Explorer application we previously installed, we should be able to see an entry added to our **hello-creation-requests** queue.

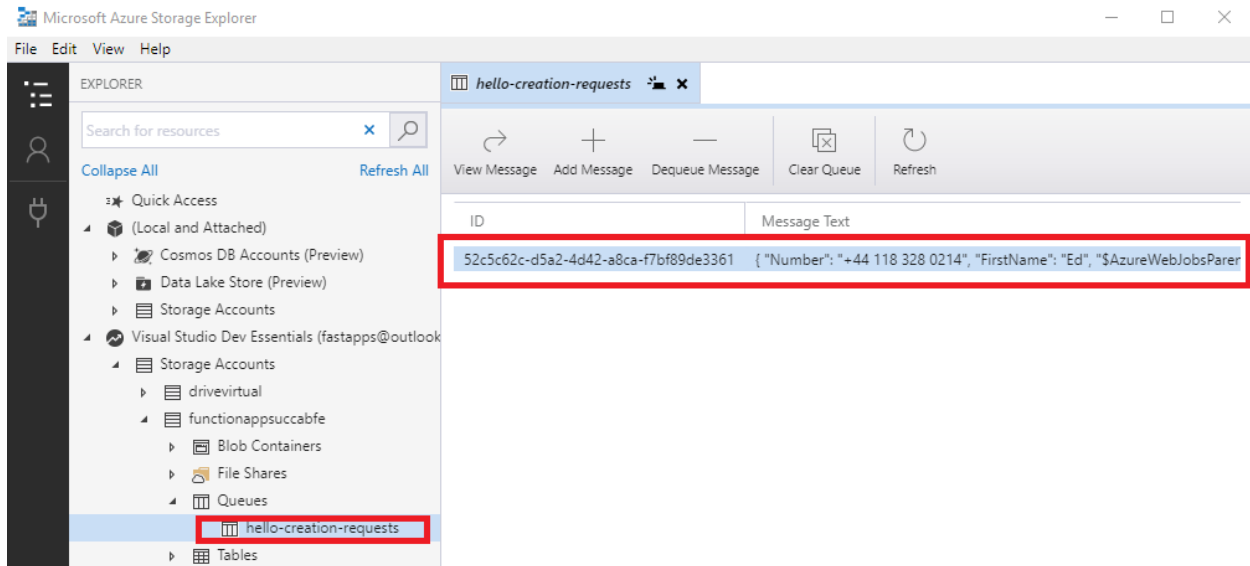


Figure 2-ad: Entry Added to the Azure Queue Storage

We can now see how the pieces are starting to fit together. If you double-click this queue entry, you'll be able to see its content.

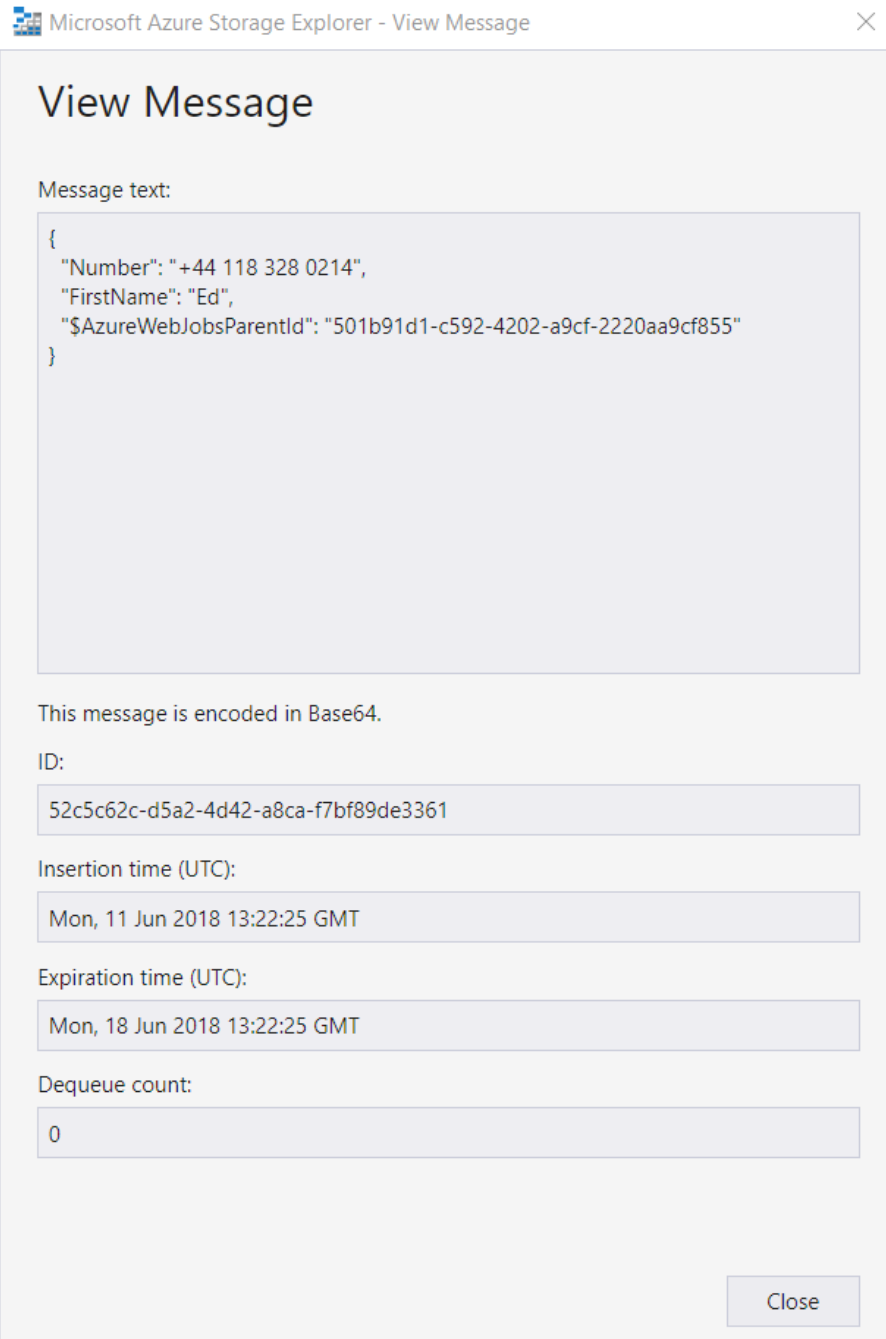


Figure 2-ae: Details of the Entry Added to the Azure Queue Storage

As you can see, the details have been saved into the queue as they were written—this is what we were expecting.

Now that we have this function that we can manually execute to add messages to the queue, let's create a new function with a queue trigger to monitor this queue and automatically execute the function whenever a new message is added to the queue.

Adding a new queue trigger function

Let's go ahead and create a new function by clicking the **+** button, as shown in [Figure 2-p](#). This time, however, we are going to choose the **Queue trigger** template using C#, as shown in the following figure.

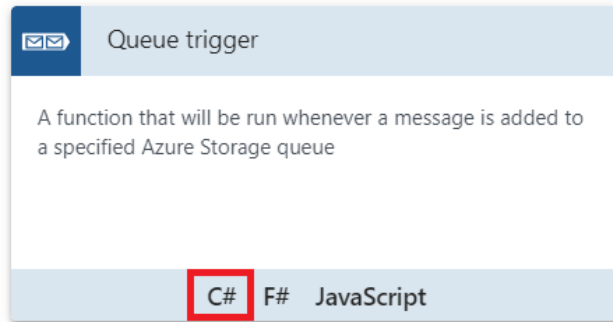


Figure 2-af: The Queue trigger Template Using C#

Once you've clicked the C# option, you'll be presented with a screen where you'll be asked to enter the **Name** of the function and the **Queue name**, and to select the **Storage account connection**. In my case, I've specified the following details.

A screenshot of the 'New Function' configuration screen. The title bar says 'Queue trigger' with an envelope icon. The main heading is 'New Function'. Below it, there is a 'Language:' dropdown menu with 'C#' selected. A red box highlights the 'Name:' field with the value 'QueueTrigger'. Below that, the section 'Azure Queue Storage trigger' contains a 'Queue name' field with the value 'hello-creation-requests', also highlighted with a red box. Below that, a 'Storage account connection' dropdown menu is highlighted with a red box, showing the value 'functionappsuccabfe_STORAGE'. To the right of this dropdown are links for 'new' and 'show value'. At the bottom, there are two buttons: 'Create' (blue) and 'Cancel' (white with blue border).

Figure 2-ag: The Queue Trigger Function Details

Notice how I've specified **hello-creation-requests** as the **Queue name** and selected the same **Storage account connection** we previously used.

Once you've specified the details, click **Create** to finalize the creation of the function.

With this done, any time we get a new message on the **hello-creation-requests** queue, this function will automatically execute, which is what we want.

Now that the function has been created, let's have a look at its **Run** method, as follows.

Listing 2-1: The Run Method of the Queue Trigger Function

```
using System;

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

We can see that there's a **myQueueItem** parameter, which we haven't seen before. When this function is being triggered (when a new message being added to the queue), **myQueueItem** will contain the value of the message that triggered this function.

What this function is going to do next is take the **CreateHelloRequest** message from the queue, examine the current time, and create a hello greeting. Then, it is going to use the first name data from the message. Once the greeting has been created, it will place it on a blob within blob storage.

To do this, let's add a new output of type of Blob Storage. Go to the **Integrate** option below the **QueueTrigger** function, and then click **New Output**.

Then, click **Azure Blob Storage** and select this option.

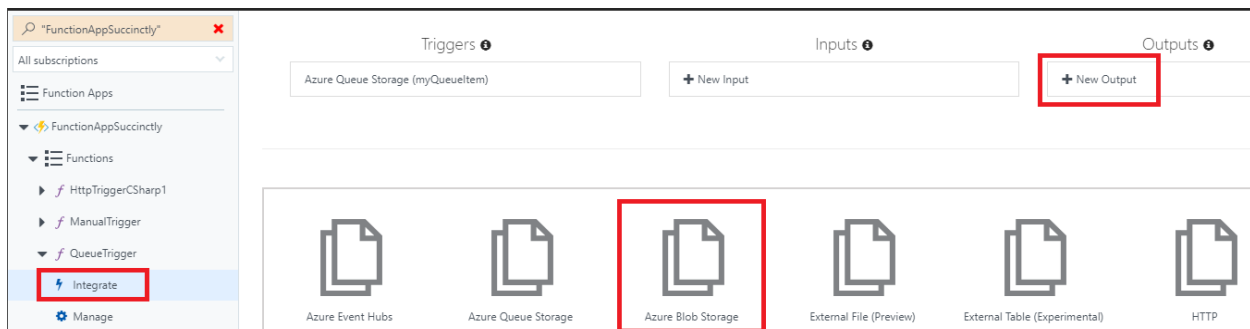


Figure 2-ah: Adding a New Blob Storage Output

In the screen that follows, let's leave the parameter name as **outputBlob** (the default) and change the name of the blob to **hello-requests**, which we previously created.

We're also going to choose the **Storage account connection** and set it to the same one we've been using so far. We can see this as follows.

Azure Blob Storage output

Blob parameter name ⓘ
outputBlob

☐ Use function return value

Storage account connection ⓘ show value
functionappsuccabfe_STORAGE new

Path ⓘ
hello-requests/{rand-guid}

Save Cancel

Figure 2-ai: Blob Storage Output Setup

Don't forget to save the changes. Then, we'll head back to the **run.csx** code, and add an additional parameter—**outputBlob**—that binds this to a new blob storage output.

I'm now going to paste a method called **GenerateHello**—which takes a first name and then generates a message based on the time of the day (it just appends the first name passed into the **GenerateHello** returned string).

Listing 2-m: The Updated *run.csx* for *QueueTrigger*

```
using System;

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

private static string GenerateHello (string firstName)
{
    string hello;
    int hourOfDay = DateTime.Now.Hour;

    if (hourOfDay <= 12)
        hello = "The Morning...";
    else if (hourOfDay <= 18)
        hello = "The Afternoon...";
    else
```

```

        hello = "The Evening...";

        return $"{hello} {firstName}";
    }

```

All this looks good so far, and the code is straightforward. However, we still need to add the references to the **CreateHelloRequest** and **HelloRequest** classes from our shared code. We can do this as follows.

Listing 2-n: References to the Shared Code

```

#load "..\SharedCode\CreateHelloRequest.csx"
#load "..\SharedCode\HelloRequest.csx"

```

Now the **run.csx** code should look as follows.

Listing 2-o: The Updated run.csx Code

```

#load "..\SharedCode\CreateHelloRequest.csx"
#load "..\SharedCode\HelloRequest.csx"

using System;

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

private static string GenerateHello (string firstName)
{
    string hello;
    int hourOfDay = DateTime.Now.Hour;

    if (hourOfDay <= 12)
        hello = "The Morning...";
    else if (hourOfDay <= 18)
        hello = "The Afternoon...";
    else
        hello = "The Evening...";

    return $"{hello} {firstName}";
}

```

We had to add these references, as it's an instance of the **HelloRequest** class that we are going to output to the blob storage.

So far so good, but have you noticed that we still need to do something to our code?

For this to work, we need to change the type of **myQueueItem** from **string** to **CreateHelloRequest**.

In addition to that, we'll also need to create an instance of **HelloRequest** and assign the **Number** field of this object. As for the **Message** field, we'll assign it the value returned from the **GenerateHello** method.

As an input to the **GenerateHello** method, we'll pass the **firstName** from the incoming queue message.

We'll still need to assign an output value to the output blob parameter—we'll have to serialize **HelloRequest** as JSON and pass this out as a JSON string. To be able to do this, we'll have to reference the **Newtonsoft.Json** library and add a **using** directive to the **Newtonsoft.Json** namespace.

Then, we can assign the value to the **outputBlob** parameter. We'll need to use the **SerializeObject** method from JSON.NET by passing in the **HelloRequest** object.

Let's see what all these changes look like. I've emphasized them in bold in the following code listing so they're easy to spot.

Listing 2-p: The Final run.csx for QueueTrigger

```
#load "..\SharedCode\CreateHelloRequest.csx"
#load "..\SharedCode\HelloRequest.csx"

// In a .csx script file, the #r syntax is used to reference a .NET
// assembly
#r "Newtonsoft.Json"

using System;
using Newtonsoft.Json;

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

    var helloRequest = new HelloRequest
    {
        Number = myQueueItem.Number,
        Message = GenerateHello(myQueueItem.FirstName)
    };

    outputBlob = JsonConvert.SerializeObject(helloRequest);
}

private static string GenerateHello (string firstName)
{
```

```

string hello;
int hourOfDay = DateTime.Now.Hour;

if (hourOfDay <= 12)
    hello = "The Morning...";
else if (hourOfDay <= 18)
    hello = "The Afternoon...";
else
    hello = "The Evening...";

return $"{hello} {firstName}";
}

```

It's always recommended that you click **Save** after you've made any changes to the code. Not only will this effectively save the recent changes, but also compile the code and highlight any potential errors.

If the compilation has not returned any errors (as would be expected if the steps have been followed), then it's now time to test the code.

Testing the queue trigger function

To test the queue trigger function that we've just created, let's head back to the previously created manual trigger function.

Use the same test data we used before, and then click the **Run** button within the **Test** pane.

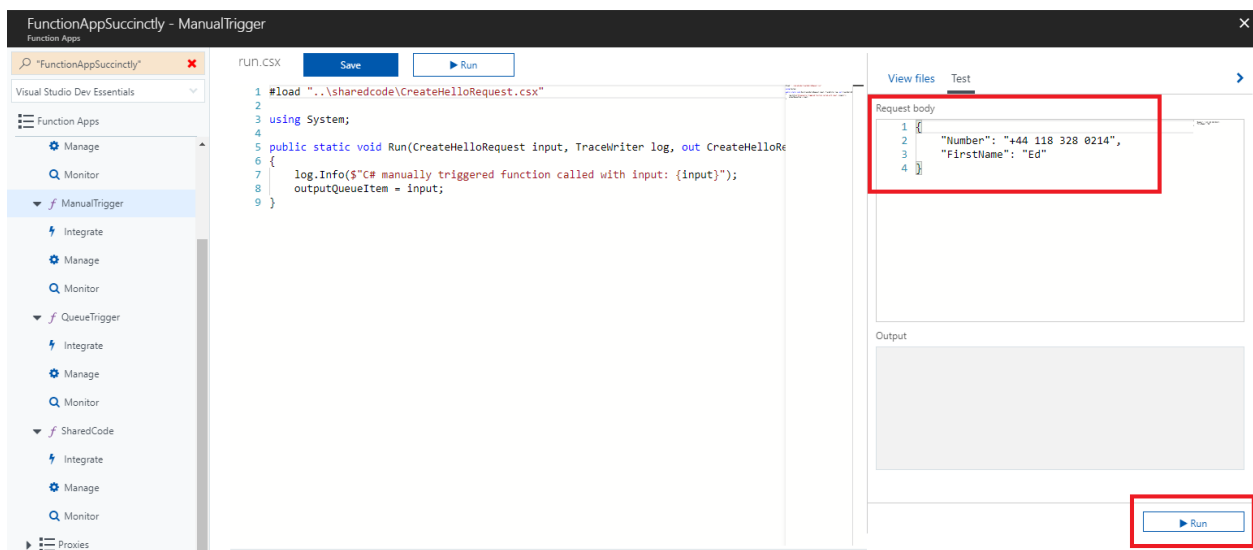


Figure 2-aj: The Manual Trigger Function

After running this function, go to the Azure Storage Explorer and refresh the content of the **hello-requests** blob. You'll notice that a new item has been added.

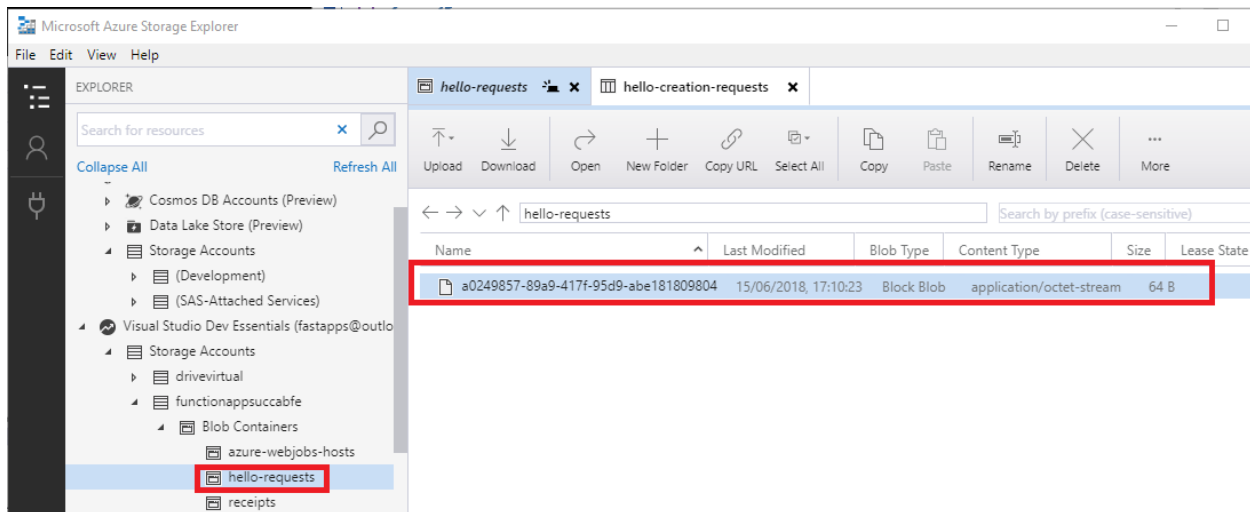


Figure 2-ak: Item Added to the Blob

If you now navigate to the **hello-creation-requests** queue, you'll see that the queue item was removed. After we ran the manual trigger function, an entry was added to the **hello-creation-requests** queue, and this triggered the execution of the queue trigger function, which read (and removed) the queue entry, and then created the **hello-requests** blob entry.

To see the actual value written to the blob, simply double-click the blob entry and open it with a text editor if prompted. See the following figure.

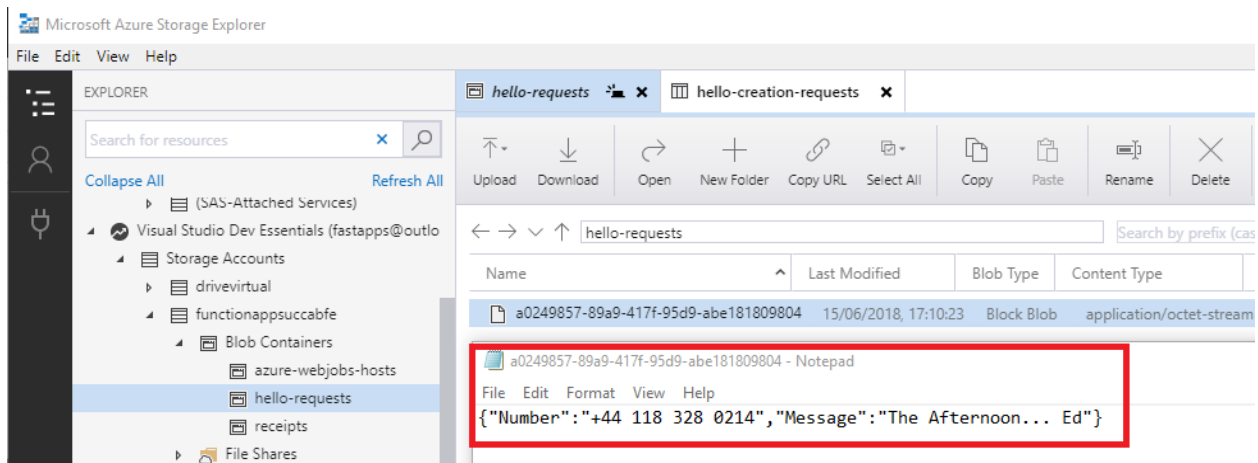


Figure 2-al: The Blob Entry Content

We can clearly see that the **GenerateHello** method did its job, as it returned the correct concatenated string—in my case, **“The Afternoon... Ed”**.

The static Run method

Something very important that you might not have even thought about is that the **Run** method of an Azure function is **static**.

This might seem like something trivial, but it is not. Let me explain why. When I was testing the interaction between the manual trigger and queue trigger functions, I accidentally removed the **static** keyword from the **Run** method of the queue trigger function.

The consequence of this innocent mistake—which took me a while to figure out (and which I still haven't been able to find properly documented anywhere)—was that the queue trigger function was not actually able to execute at all.

The difficulty of troubleshooting this was that everything looked like it was correctly set up, but it was difficult to determine whether the actual **Run** method within the queue trigger function was doing something.

This is so very important: always make sure your **Run** method is correctly marked as **static** to avoid losing time and running into unexpected situations.

Summary

It's been a long but interesting chapter—I couldn't have made it any shorter, even if I wanted to—and we managed to cover quite a lot of ground.

The main goal of this chapter was to walk you through all the steps required to get a small, but workable scenario where a couple of Azure functions work together, one triggering the other by sharing some common code, and using essential Azure components such as blobs and queues to establish that interaction.

I think we achieved this objective and managed to glue all these pieces together and build something precise, while keeping things as simple as possible.

In the next chapter, we are going to explore blob and timer triggers in more detail. One of the things we'll do is to create a function that has a blob trigger, which will notice that we're writing messages and execute the function.

Before we dive into those details, we'll also have a quick look at queue trigger metadata.

Chapter 3 Metadata, Blob, and Timer Triggers

Quick intro

At this point, we know how to create an Azure function application that uses various functions that can share code and can be either manually triggered or triggered when a new item arrives in an Azure queue.

There are though extra parameters that are quite useful when working with queues, such as **ExpirationTime**, **InsertionTime**, **NextVisibleItem**, **DequeueCount**, and **QueueTrigger**. This type of information is known as metadata.

In this chapter, we'll dig a little bit into metadata and explore how we can use blob storage and timers as triggers for Azure functions.

Building upon the Azure function application we previously created, we are going to learn how we can take the input from a blob container by using a blob trigger that will automatically know when a new blob is written.

We'll also learn how to execute a function on a specified schedule using a timer trigger. Shortly after reviewing the basics of metadata, we'll start off by creating a blob-triggered function in the Azure Portal and access the blob's metadata.

In the body of the blob-triggered function, we'll perform some processing, which will include writing out to another blob container.

An interesting aspect about blob triggers is that we can also specify different paths. This means that we could trigger a function only if a new blob is added with a specified extension.

Finally, we'll create a new type of function which will be timer-based. We'll learn how to specify how often the function executes by using CRON expressions. Timer-triggered functions are a great way to do some housekeeping, so we'll use them to delete blobs in the blob storage that are older than a specified time range.

Let's get started.

Queue metadata

Let's head over to the **Integrate** option of our **QueueTrigger** function so we can explore the type of metadata we have available.

By default, the metadata information is not visible, so to see it we must click the **Documentation** link that appears on the **Integrate** page.

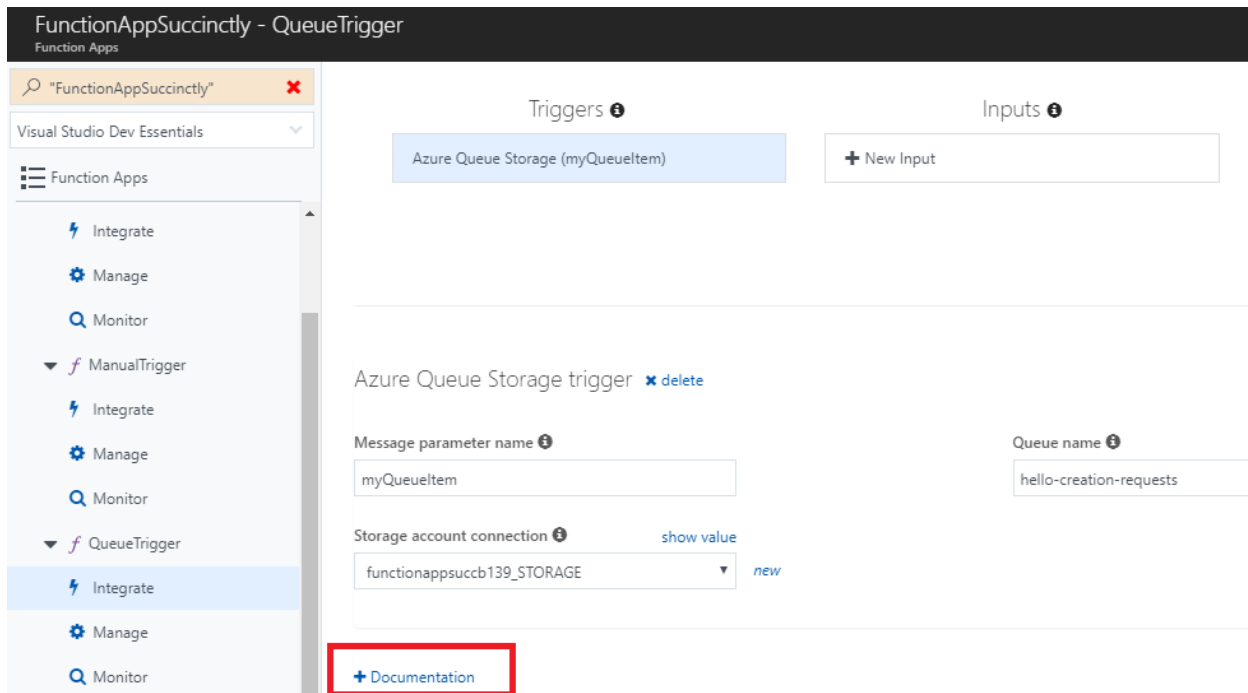


Figure 3-a: The Documentation Link under the Integrate View

Once you have clicked the **Documentation** link, you'll probably have to scroll down a bit to find the metadata details we are looking for. An example follows.

Settings for storage queue trigger

- **name** : The variable name used in function code for the queue or the queue message.
- **queueName** : The name of the queue to poll. For queue naming rules, see [Naming Queues and Metadata](#).
- **connection** : The name of an app setting that contains a storage connection string. If you leave **connection** empty, the trigger will work with the default storage connection string for the function app, which is specified by the AzureWebJobsStorage app setting.
- **type** : Must be set to `queueTrigger`.
- **direction** : Must be set to `in`.

Additional metadata for Storage Queue trigger

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)

C# types for Storage Queue trigger

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

- `Object` (from JSON)
- `String`
- `Byte array`
- `CloudQueueMessage` (C#)

Figure 3-b: Metadata Documentation

Metadata details can be added as new parameters to the **Run** method with their specific names.

Let's see how we can modify the **Run** method of the **QueueTrigger** function to add some new parameters, such as the **InsertionTime** and **Id** of a queue item.

To do this, click the **QueueTrigger** item so that the code for the **Run** method opens.

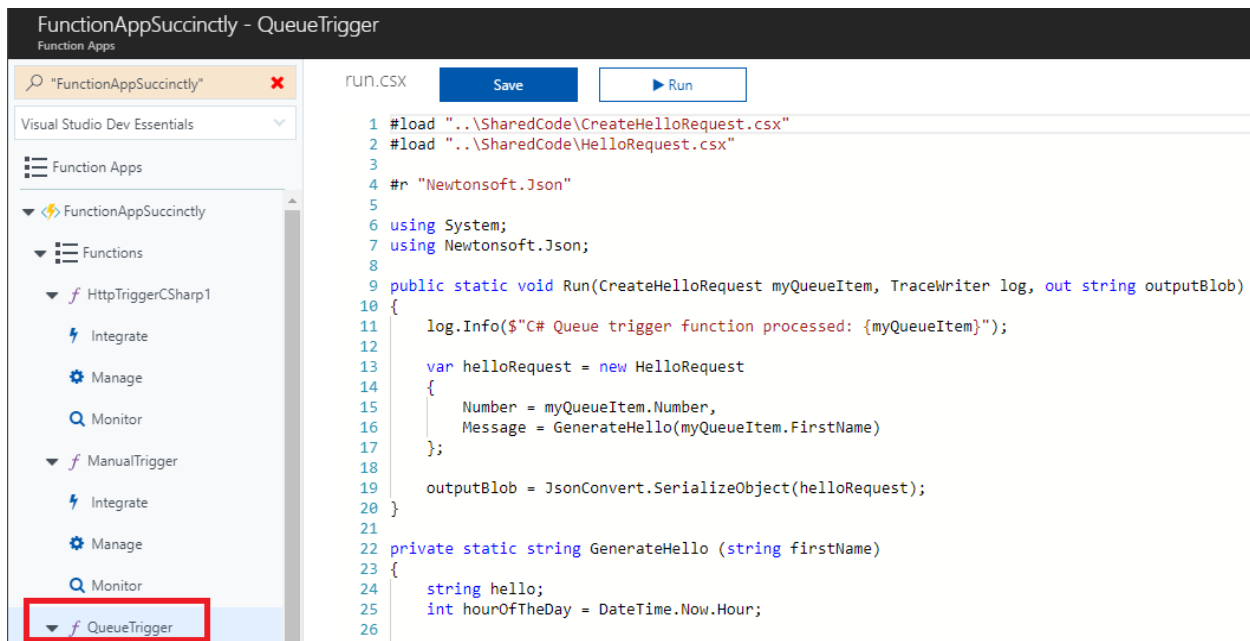


Figure 3-c: The QueueTrigger Run Method

From this code, we can start to add new parameters after the **outputBlob** variable.

The first parameter I'm going to add is called **insertionTime**, of type **DateTimeOffset**. The second parameter that I'll be adding is called **id**, and it is of type **string**.

The modified code of the **Run** method follows, with the additional parameters highlighted in bold.

Listing 3-a: The Updated run.csx for QueueTrigger

```
#load "..\SharedCode\CreateHelloRequest.csx"
#load "..\SharedCode\HelloRequest.csx"

#r "Newtonsoft.Json"

using System;
using Newtonsoft.Json;

public static void Run (CreateHelloRequest myQueueItem, TraceWriter log,
    out string outputBlob,
    DateTimeOffset insertionTime,
    string id)
{
    log.Info(
        $"C# Queue trigger function processed: {myQueueItem}");

    log.Info($"InsertionTime: {insertionTime}");
    log.Info($"Id: {id}");
}
```

```

var helloRequest = new HelloRequest
{
    Number = myQueueItem.Number,
    Message = GenerateHello(myQueueItem.FirstName)
};

outputBlob = JsonConvert.SerializeObject(helloRequest);
}

private static string GenerateHello (string firstName)
{
    string hello;
    int hourOfDay = DateTime.Now.Hour;

    if (hourOfDay <= 12)
        hello = "The Morning...";
    else if (hourOfDay <= 18)
        hello = "The Afternoon...";
    else
        hello = "The Evening...";

    return $"{hello} {firstName}";
}

```

Now when this trigger executes, it's going to give us additional information about the message that triggered this function. What we'll do with this information is output it as log messages to the Azure Portal console.

Once the changes have been entered, click **Save** to save the changes and make sure that there are no compilation errors.

As you can see, it's very easy to add metadata parameters to the **Run** method, as it might be useful to have extra details about the operation being performed.

A blob-triggered function

Let's go back to the Azure Portal and create a new function by clicking the **+** button. This time, let's select the **Blob trigger** option with **C#**, as shown in the following figure.

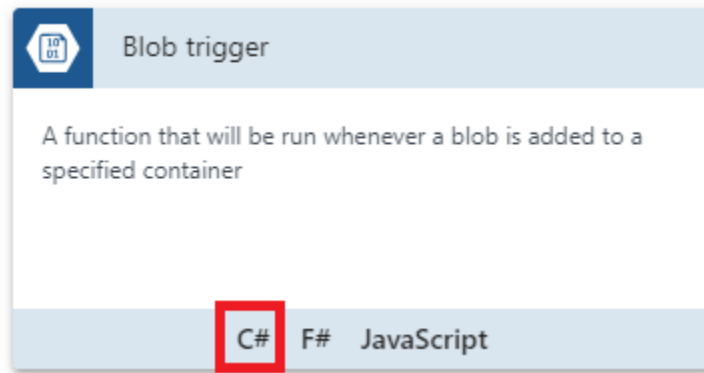


Figure 3-d: The Blob trigger Option with C#

Once we've selected that option, we'll be asked to enter some details for our blob-triggered function. We can see these in the following screenshot.

A screenshot of the 'New Function' configuration screen. The title 'New Function' is at the top. Below it, the 'Language:' dropdown is set to 'C#'. The 'Name:' field contains 'BlobTrigger' and is highlighted with a red box. Below that, the 'Azure Blob Storage trigger' section has a 'Path' field with the value 'hello-requests/{name}' and an information icon, also highlighted with a red box. The 'Storage account connection' dropdown is set to 'functionappsuccb139_STORAGE' and is highlighted with a red box. To the right of this dropdown are links for 'show value' and 'new'. At the bottom, there are 'Create' and 'Cancel' buttons.

Figure 3-e: Blob-triggered Function Details

Notice that the function name has been set to **BlobTrigger**, the path points to the **hello-requests** blob we previously created, and the **Storage account connection** has been set accordingly.

You might have noticed that the name of the **Storage account connection** looks slightly different than it did in previous screenshots. This is because when doing my own tests, I removed the previous connection and created a new one, which is perfectly valid. However, you might not have had the need to do that, so you'll probably have the same **Storage account connection** you were previously using.

Notice the format of the Azure Blob Storage trigger path—it is set to **hello-requests/{name}**. The **name** element inside the curly braces will give us the name of the blob that triggered this function. This **name** element is also bound to a **string** parameter in our function's input.

With all the details entered, click **Create** to finalize the creation of the blob-triggered function.

Once created, we'll be presented with the default code of **run.csx** as follows.

Listing 3-b: The Default run.csx for BlobTrigger

```
public static void Run(Stream myBlob, string name, TraceWriter log)
{
    log.Info($"C# Blob .. function Processed blob{Name: {name} ..");
}
```

Notice how the second parameter of the **Run** method is called **name** and it represents the **name** element that was specified in the Azure Blob Storage trigger path.

With this done, let's go over to the **Integrate** option of our **BlobTrigger** function and click **New Output**. Select the **Azure Blob Storage** option.

We'll leave the **Blob parameter name** as is and change the name of the blob container in **Path** to **receipts/{rand-guid}**, which is the name of a blob container we previously created (and have not used yet).

Make sure to select the correct **Storage account connection** as well. We can see these changes as follows.

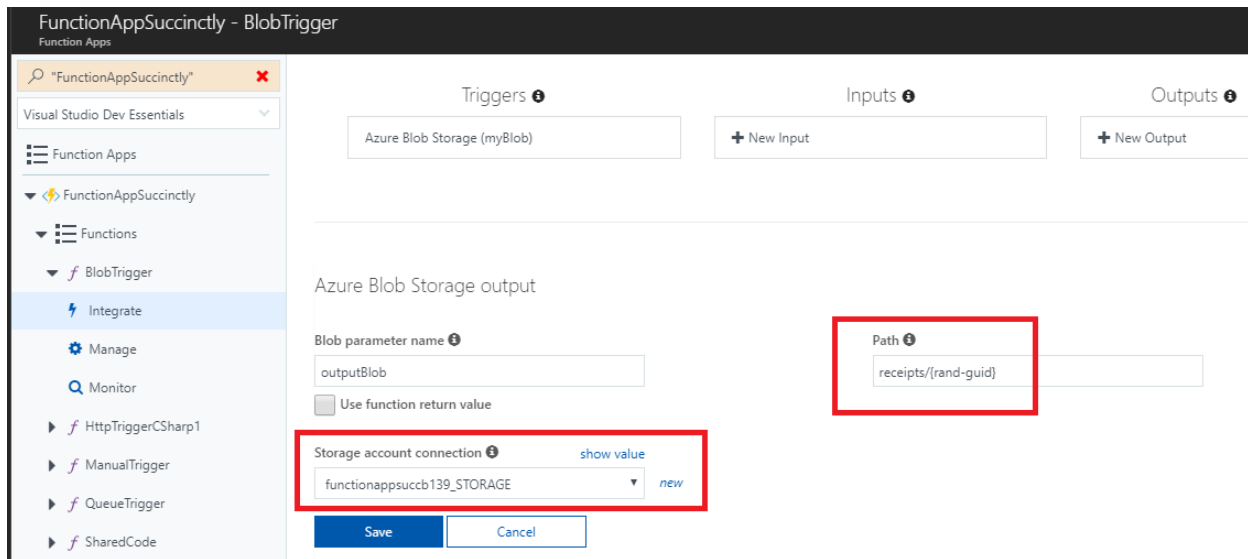


Figure 3-f: Creating a Blob Storage Output

Notice that we're using the **{rand-guid}** syntax, which means that we'll get a random GUID assigned to the file name.

With this done, let's head back to our default **run.csx** code and add the output blob parameter. We can do this as follows.

Listing 3-c: The Updated **run.csx** for **BlobTrigger**

```
public static void Run(Stream myBlob, string name,
    TraceWriter log, Stream outputBlob)
{
    log.Info($"C# Blob .. function Processed blob\Name: {name} ..");
}
```

If we want to get additional metadata about the input blob, we can change the type from **Stream** to **CloudBlockBlob**.

In order to have access to this **CloudBlockBlob** type, we'll need to add a reference to the **Microsoft.WindowsAzure.Storage** assembly, and also add a **using** statement.

We'll use JSON as well, so we'll need to add a reference to **Newtonsoft.Json** and add the corresponding **using** directive.

We're also going to be making use of a couple of classes within the **SharedCode** function we previously wrote. We can add these with the **load** directive. I'll remove the log line, as we won't need it.

So, let's update the **run.csx** code to reflect these changes, as follows.

Listing 3-d: Updated run.csx

```
#load "..\SharedCode\HelloRequest.csx"
#load "..\SharedCode\MsgSentConfirmation.csx"

#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name,
                      TraceWriter log, Stream outputBlob)
{
}
```

Because we've changed the incoming blob to type **CloudBlockBlob**, we can get additional metadata information.

Let's make use of this metadata. We can do this by simply printing to the log. We can access various metadata properties, such as the **Name** of the blob, the **StorageUri** of the incoming blob, and even the blob's container metadata.

The job that this blob-triggered function is going to perform can be split into three parts. The first part will be to download the blob content from the blob storage.

To do that, we are going to invoke a method called **GetHelloRequest** and pass it **CloudBlockBlob**. Then, we'll simply write the message returned by **GetHelloRequest**.

So, let's update **run.csx** to reflect these changes.

Listing 3-e: Updated run.csx

```
#load "..\SharedCode\HelloRequest.csx"
#load "..\SharedCode\MsgSentConfirmation.csx"

#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name,
                      TraceWriter log, Stream outputBlob)
{
    log.Info($"Metadata Name: {myBlob.Name}");
    log.Info($"Metadata StorageUri: {myBlob.StorageUri}");
    log.Info($"Metadata Container: {myBlob.Container.Name}");
}
```

```

    HelloRequest helloRequest = GetHelloRequest(myBlob);
    log.Info($"Hello Request: {helloRequest}");
}

```

Notice that if you click **Save**, the previous code will fail to compile because the **GetHelloRequest** method is invoked but it doesn't actually exist yet.

Now let's create the **GetHelloRequest** method, which is going to download the blob into a memory stream and use the **Newtonsoft.Json** assembly to deserialize the content into a **HelloRequest** object, which is returned to the caller.

This is how the code of *run.csx* looks with the **GetHelloRequest** method added.

Listing 3-f: Updated run.csx

```

#load "..\SharedCode\HelloRequest.csx"
#load "..\SharedCode\MsgSentConfirmation.csx"

#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name,
                      TraceWriter log, Stream outputBlob)
{
    log.Info($"Metadata Name: {myBlob.Name}");
    log.Info($"Metadata StorageUri: {myBlob.StorageUri}");
    log.Info($"Metadata Container: {myBlob.Container.Name}");

    HelloRequest helloRequest = GetHelloRequest(myBlob);
    log.Info($"Hello Request: {helloRequest}");
}

public static HelloRequest GetHelloRequest(CloudBlockBlob blob)
{
    HelloRequest helloRequest;

    using (var ms = new MemoryStream())
    {
        blob.DownloadToStream(ms);
        ms.Position = 0;

        using (var res = new StreamReader(ms))
        {
            using (var jtr = new JsonTextReader(res))
            {

```

```

        var s = new JsonSerializer();
        helloRequest = s.Deserialize<HelloRequest>(jtr);
    }
}

return helloRequest;
}

```

I mentioned before that this blob-triggered function is going to perform a job that can be split into three parts. We've just done the first part, which is to call and execute the **GetHelloRequest** method.

The second part is to send a text message using the returned **helloRequest** object. To do this, we are going to call a method named **SendMessage**, which will return a **Guid** from a simulated messaging gateway. We'll need to pass the **helloRequest** object as a parameter.

Let's create the **SendMessage** method—the following listing is the updated **run.csx** code.

Listing 3-g: Updated run.csx

```

#load "..\SharedCode\HelloRequest.csx"
#load "..\SharedCode\MsgSentConfirmation.csx"

#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name,
    TraceWriter log, Stream outputBlob)
{
    log.Info($"Metadata Name: {myBlob.Name}");
    log.Info($"Metadata StorageUri: {myBlob.StorageUri}");
    log.Info($"Metadata Container: {myBlob.Container.Name}");

    HelloRequest helloRequest = GetHelloRequest(myBlob);
    log.Info($"Hello Request: {helloRequest}");

    string id = SendMessage(helloRequest);
}

public static string SendMessage(HelloRequest req)
{
    // We simulate sending SMS with req and returning a unique GUID
    return Guid.NewGuid().ToString();
}

```

```

public static HelloRequest GetHelloRequest(CloudBlockBlob blob)
{
    HelloRequest helloRequest;

    using (var ms = new MemoryStream())
    {
        blob.DownloadToStream(ms);
        ms.Position = 0;

        using (var res = new StreamReader(ms))
        {
            using (var jtr = new JsonTextReader(res))
            {
                var s = new JsonSerializer();
                helloRequest = s.Deserialize<HelloRequest>(jtr);
            }
        }
    }

    return helloRequest;
}

```

We now have two parts finished, and we are just missing the last stage. Finally, let's create a message confirmation and write it to the blob storage.

To do that, let's create an instance of the **MsgSentConfirmation** class from the **SharedCode** function.

We'll need to set the **ReceiptId** property to the value received from the simulated messaging gateway and set the **Number** and **Message** properties from the information we'll read from the blob storage.

Finally, we want to upload this **MsgSentConfirmation** object to the blob storage.

We can do this by creating a new method called **UploadMsg** that is going to use JSON.NET to serialize the **MsgSentConfirmation** object and write it to the blob storage.

Let's go ahead and add this code to **run.csx**. These changes are indicated in bold in the following listing.

Listing 3-h: Updated run.csx

```

#load "..\SharedCode\HelloRequest.csx"
#load "..\SharedCode\MsgSentConfirmation.csx"

#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

```

```

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name,
                      TraceWriter log, Stream outputBlob)
{
    log.Info($"Metadata Name: {myBlob.Name}");
    log.Info($"Metadata StorageUri: {myBlob.StorageUri}");
    log.Info($"Metadata Container: {myBlob.Container.Name}");

    HelloRequest helloRequest = GetHelloRequest(myBlob);
    log.Info($"Hello Request: {helloRequest}");

    string id = SendMessage(helloRequest);

    var confirm = new MsgSentConfirmation
    {
        ReceiptId = id,
        Number = helloRequest.Number,
        Message = helloRequest.Message
    };

    UploadMsg(confirm, outputBlob);
}

public static void UploadMsg(MsgSentConfirmation confirm,
                             Stream outputBlob)
{
    using (var w = new StreamWriter(outputBlob))
    {
        using (var jw = new JsonTextWriter(w))
        {
            JsonSerializer s = new JsonSerializer();
            s.Serialize(jw, confirm);
            jw.Flush();
        }
    }
}

public static string SendMessage(HelloRequest req)
{
    // We simulate sending SMS with req and returning a unique GUID
    return Guid.NewGuid().ToString();
}

public static HelloRequest GetHelloRequest(CloudBlockBlob blob)
{
    HelloRequest helloRequest;
}

```

```

using (var ms = new MemoryStream())
{
    blob.DownloadToStream(ms);
    ms.Position = 0;

    using (var res = new StreamReader(ms))
    {
        using (var jtr = new JsonTextReader(res))
        {
            var s = new JsonSerializer();
            helloRequest = s.Deserialize<HelloRequest>(jtr);
        }
    }

    return helloRequest;
}

```

As you can see, it's not complicated at all. After adding the code to the Azure Portal, click **Save** so that all changes are saved, and check if there are any compilation errors.

In order to test this, let's go back to the **ManualTrigger** function. Change the input test data and then click **Run** to execute it.

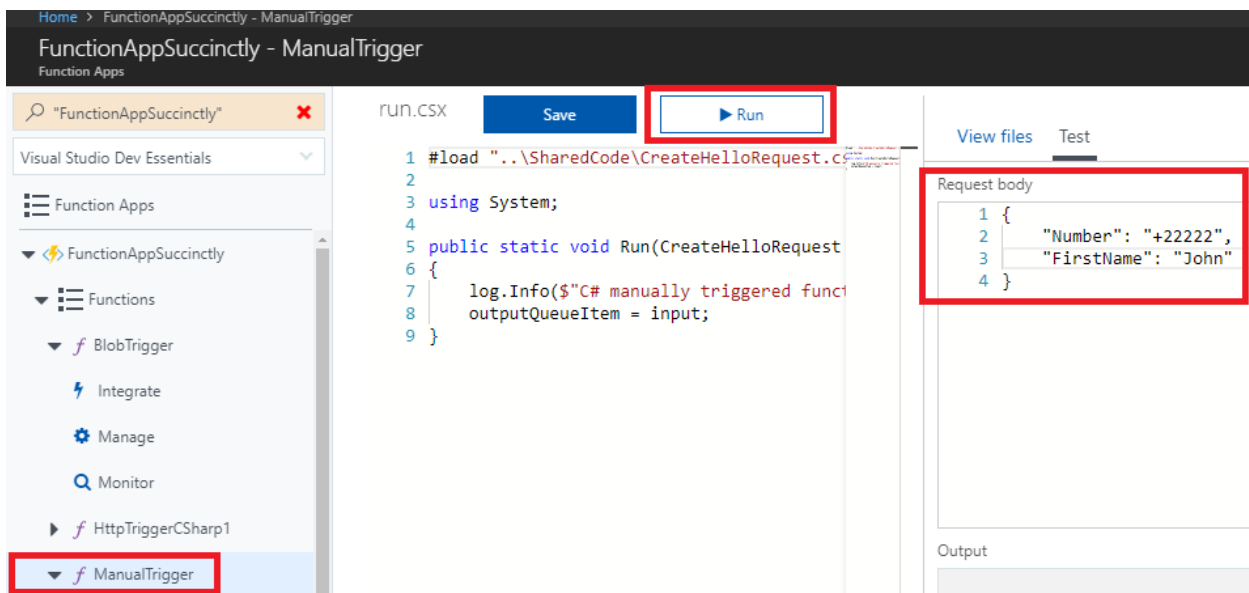


Figure 3-g: Running the ManualTrigger Function

When this executes, it is going to go through a series of functions. When writing to the queue, our queue trigger will pick up the message and write it to the blob storage. Then, the blob trigger should receive this and the **BlobTrigger** function should execute. The final result should be available on the **receipts** blob container.

Go to the Azure Storage Explorer to verify the result.

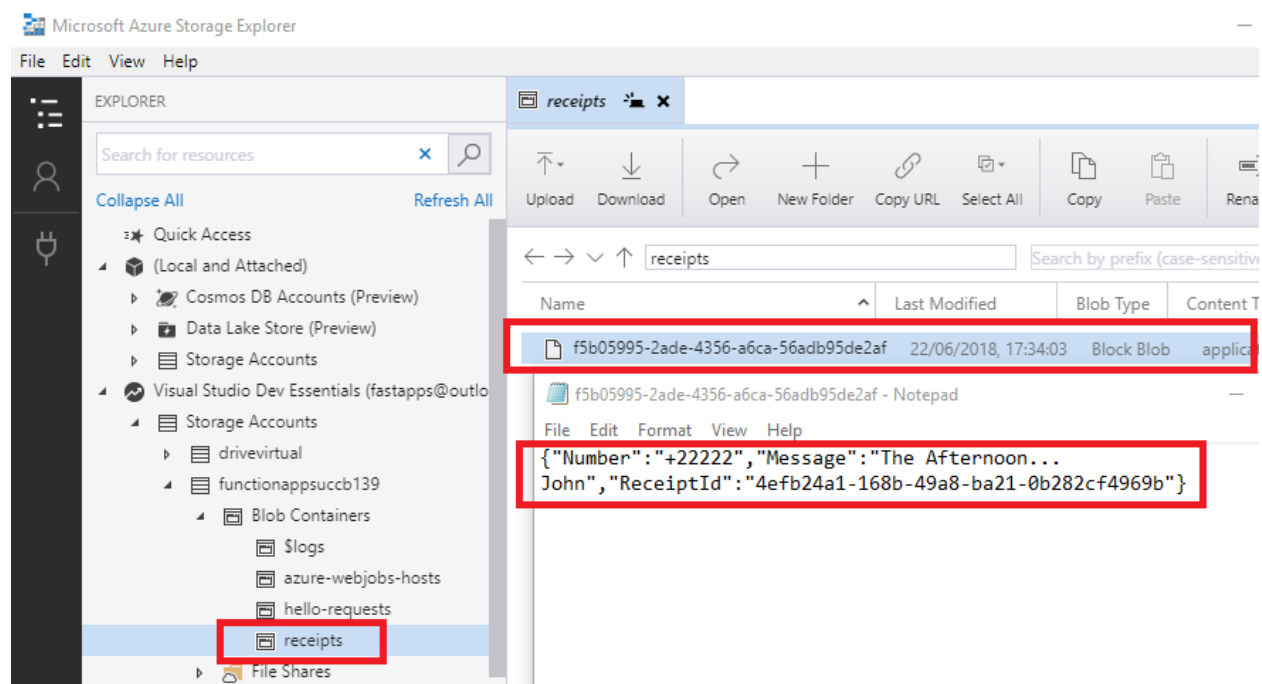


Figure 3-h: The Resultant Receipts Blob

Now we have a working solution made up of several Azure functions that work together by communicating through queues and blobs.

If we keep executing the manually triggered function, we might end up with a lot of items in the blobs we are using, so it would be handy to have a function to clean them up after a given period of time. This can be achieved by using a function that executes periodically and removes older blobs.

Understanding CRON expressions

Before we create a timer-triggered function, it's important to understand how CRON expressions work. When creating a timer-triggered function, we'll need to specify a CRON expression, which determines how often or when the trigger will execute.

With Azure Functions, specifying a CRON expression consists of six parts, as shown in the following figure.



Figure 3-i: Parts of a CRON Expression

A CRON expression consists of the second, minute, hour, day, month, and the day of the week that indicate when the task will execute.

In addition to each of these values, there are a number of special characters that we can use inside the expression.

If we have a list of values, we can use a comma (,). If we want to create step values—something that reoccurs periodically—we can use a forward slash (/).

An asterisk (*), which is similar to a wildcard character, indicates that there can be any value. A hyphen (-) indicates a range of values.

Let's have a look at an example of a CRON expression.



Figure 3-j: Example of a CRON Expression

In this example, the CRON expression will execute at 9:28 pm at 35 seconds, every first day of every month, independently of the day of the week that day is.

Imagine that instead, we want to execute this task every day in March (note that the month is zero-based), and not only on the first day of the month. We could do this as follows.

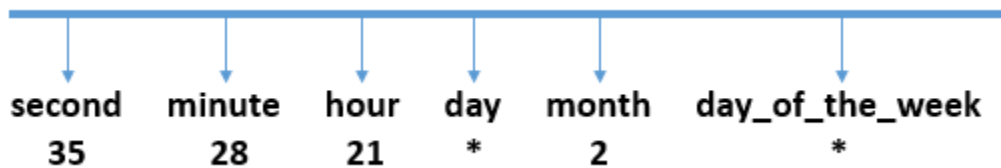


Figure 3-k: Another Example of a CRON Expression

We can also create schedules that reoccur by using step values. The following is an example that indicates the task will be triggered once every 28 minutes, every day—instead of specifically on minute 28.

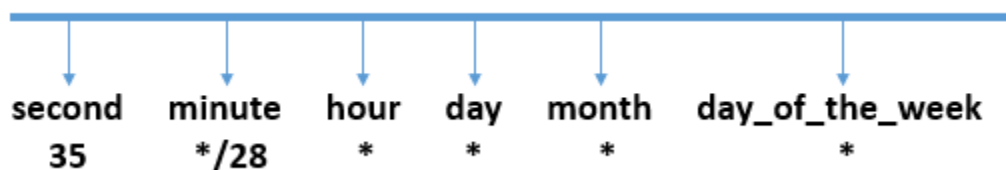


Figure 3-l: Another Example of a CRON Expression

We can also indicate specific days, months, or days of the week to limit the execution of this task to those specific dates. We can do this as follows.

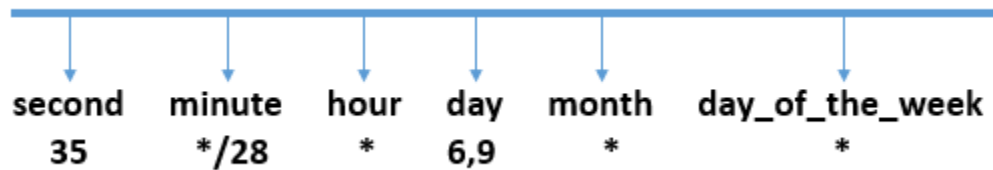


Figure 3-m: Another Example of a CRON Expression

In this example, the task will be executed every 28 minutes, on second 35 of every hour, but only on day 6 and 9 of every month, independently of the day of the week it is.

If you would like to limit this task to execute on specific days of the week, say on Saturdays and Sundays, when users are not in the office, it could be done as follows.



Figure 3-n: Another Example of a CRON Expression

In the previous example, the value 0 of day_of_the_week represents Sundays, and 6 indicates Saturdays. As you might have guessed, 1 indicates Monday, 2 indicates Tuesday, and so on.

It's also possible to specify ranges. A range from Monday to Wednesday inclusive would be as follows.



Figure 3-o: Another Example of a CRON Expression

As you can see, the possibilities are endless. Now that we understand how CRON expressions work, let's go back to the Azure Portal and create a new function that uses a time trigger.

A timer-triggered function

Just like we've seen previously in the Azure Portal, click the **+** button next to **Functions** to create a new Azure function.

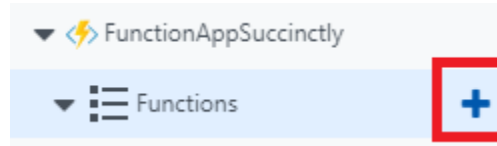


Figure 3-p: The + Button to Add a New Azure Function

When prompted, choose the **Timer trigger** template with **C#**.

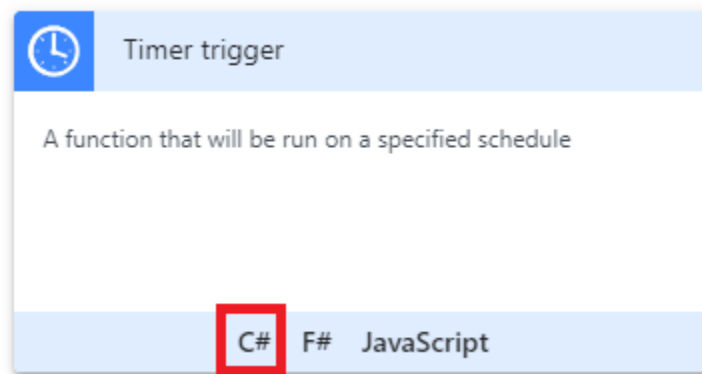


Figure 3-q: The Timer trigger Template

After clicking the **C#** option, a dialog box will appear where we can indicate the **Name** of the Azure function and the execution **Schedule**.

By default, the CRON expression is set to run every five minutes, but we can modify it to run at a different schedule. In my case, I've set it to execute every 10 minutes instead. We can see this as follows.

The screenshot shows the 'New Function' wizard in the Azure portal. At the top, there's a blue header with a clock icon and the text 'Timer trigger'. Below this, the title 'New Function' is centered. The 'Language' dropdown is set to 'C#'. The 'Name' field contains 'TimerTrigger'. Under the 'Timer trigger' section, the 'Schedule' field is highlighted with a red rectangle and contains the cron expression '0 */10 * * * *'. Below the schedule field are two buttons: a blue 'Create' button and a light blue 'Cancel' button.

Figure 3-r: Creating the Timer trigger Function

Once the **Name** of the function and the **Schedule** have been specified, click **Create** to finish the creation of the function.

With the timer-triggered function created, notice how in the **run.csx** code the first parameter is set to use the **TimerInfo** object. You can see this in the following listing.

Listing 3-i: run.csx for the Timer-Triggered Function

```
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer triggered function executed at: {DateTime.Now}");
}
```

By using the **TimerInfo** object, we can access information about the timer that's executing this function.

In order to see some of these timer details, let's add some additional log information to our **Run** method. We can see this on the listing that follows.

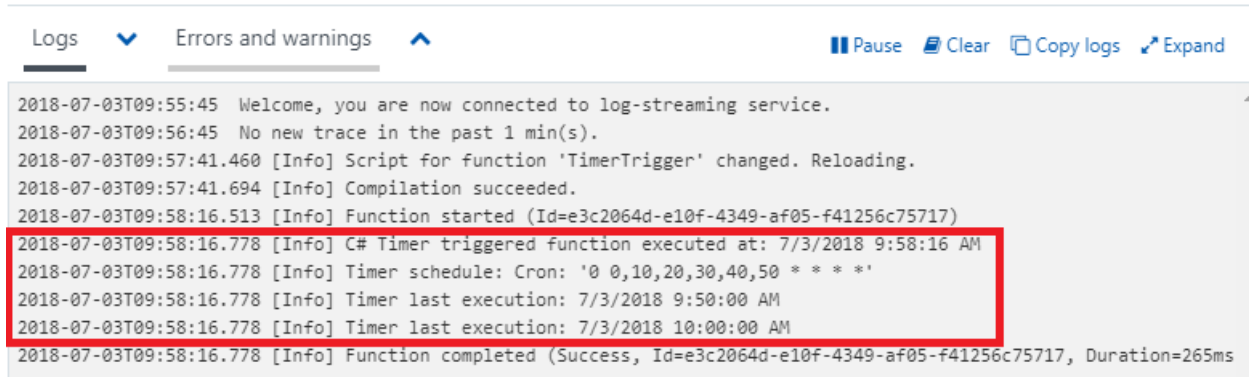
Listing 3-j: The Modified run.csx for the Timer-Triggered Function

```
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer triggered function executed at: {DateTime.Now}");

    log.Info($"Timer schedule: {myTimer.Schedule}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Last}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Next}");
}
```

Notice how I've added the details of the **Last** and **Next** schedule execution to the output. If you actually execute the function by clicking **Run**, you'll be able to see the following results on the logger.



```
2018-07-03T09:55:45 Welcome, you are now connected to log-streaming service.
2018-07-03T09:56:45 No new trace in the past 1 min(s).
2018-07-03T09:57:41.460 [Info] Script for function 'TimerTrigger' changed. Reloading.
2018-07-03T09:57:41.694 [Info] Compilation succeeded.
2018-07-03T09:58:16.513 [Info] Function started (Id=e3c2064d-e10f-4349-af05-f41256c75717)
2018-07-03T09:58:16.778 [Info] C# Timer triggered function executed at: 7/3/2018 9:58:16 AM
2018-07-03T09:58:16.778 [Info] Timer schedule: Cron: '0 0,10,20,30,40,50 * * * *'
2018-07-03T09:58:16.778 [Info] Timer last execution: 7/3/2018 9:50:00 AM
2018-07-03T09:58:16.778 [Info] Timer last execution: 7/3/2018 10:00:00 AM
2018-07-03T09:58:16.778 [Info] Function completed (Success, Id=e3c2064d-e10f-4349-af05-f41256c75717, Duration=265ms)
```

Figure 3-s: The Results of Executing the Timer-Triggered Function

Given that this is a timer-triggered function, the execution of the function will be done automatically, but since I was a bit impatient to sit around and wait for 10 minutes to let the function execute on its own, I clicked the **Run** button to show you these results.

Notice that as part of the results, we can see the long form of the CRON expression, which specifies how this function will automatically execute.

In the background, while I have been busy writing this, the timer has already executed automatically, and the results can be seen in the following screenshot.

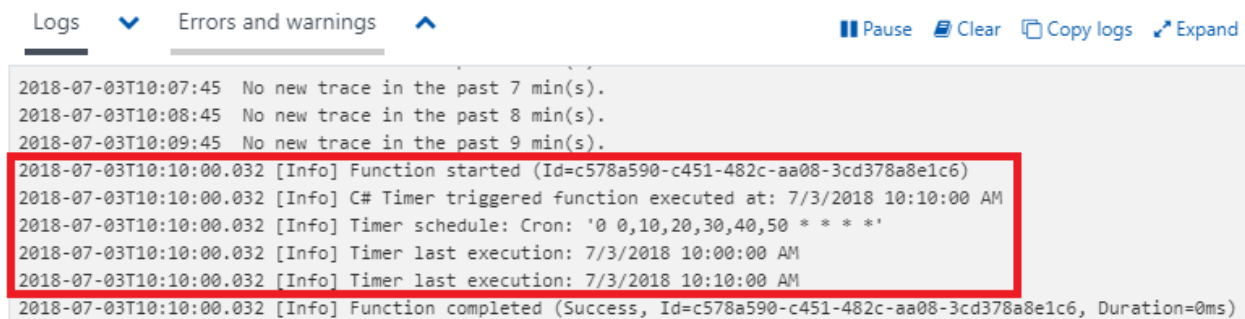
The screenshot shows the 'Logs' tab in the Azure portal. The log entries are as follows:
2018-07-03T10:07:45 No new trace in the past 7 min(s).
2018-07-03T10:08:45 No new trace in the past 8 min(s).
2018-07-03T10:09:45 No new trace in the past 9 min(s).
2018-07-03T10:10:00.032 [Info] Function started (Id=c578a590-c451-482c-aa08-3cd378a8e1c6)
2018-07-03T10:10:00.032 [Info] C# Timer triggered function executed at: 7/3/2018 10:10:00 AM
2018-07-03T10:10:00.032 [Info] Timer schedule: Cron: '0 0,10,20,30,40,50 * * * *'
2018-07-03T10:10:00.032 [Info] Timer last execution: 7/3/2018 10:00:00 AM
2018-07-03T10:10:00.032 [Info] Timer last execution: 7/3/2018 10:10:00 AM
2018-07-03T10:10:00.032 [Info] Function completed (Success, Id=c578a590-c451-482c-aa08-3cd378a8e1c6, Duration=0ms)
The log entries from 'Function started' to the second 'Timer last execution' are highlighted with a red rectangle.

Figure 3-t: Automatic Execution of the Timer-Triggered Function

Now that we've seen how this function works, let's go ahead and make it more interesting. So, let's modify the **run.csx** code and add a reference to the **WindowsAzure.Storage** and **System.Configuration** assemblies with their respective namespaces.

Listing 3-k: The Modified run.csx for the Timer-Triggered Function

```
#r "Microsoft.WindowsAzure.Storage"
#r "System.Configuration"

using Microsoft.WindowsAzure.Storage.Blob;
using Microsoft.WindowsAzure.Storage;
using System.Configuration;
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer triggered function executed at: {DateTime.Now}");

    log.Info($"Timer schedule: {myTimer.Schedule}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Last}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Next}");
}
```

Next, we want to get access to the connection string to our storage account. To get this, we need to read it out from the app settings.

You'll notice that in the code that follows, I'll reference the string **functionappsuccb139_STORAGE**, which is the same connection that we've used in the blob and queue triggers.

Once that's been done, we need to get access to our **receipts** blob container so that we can check for any blobs that are over a specified age.

Let's update **run.csx** to reflect this.

Listing 3-l: The Modified run.csx for the Timer-Triggered Function

```
#r "Microsoft.WindowsAzure.Storage"
#r "System.Configuration"

using Microsoft.WindowsAzure.Storage.Blob;
using Microsoft.WindowsAzure.Storage;
using System.Configuration;
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer triggered function executed at: {DateTime.Now}");

    log.Info($"Timer schedule: {myTimer.Schedule}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Last}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Next}");

    string conn =
        ConfigurationManager.AppSettings["functionappsucb139_STORAGE"];

    CloudStorageAccount storageAccount = CloudStorageAccount.Parse(conn);
    CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
    CloudBlobContainer container =
        blobClient.GetContainerReference("receipts");
}
```

With this code in place, we can now specify that any blob over five minutes old needs to be deleted. This way, we have a clean system.

Next, we need to loop through all the **CloudBlockBlobs** objects within the **receipts** blob container and verify whether or not they are too old. If they are too old, we should delete them.

Let's modify our code to reflect this.

Listing 3-m: The Modified run.csx for the Timer-Triggered Function

```
#r "Microsoft.WindowsAzure.Storage"
#r "System.Configuration"

using Microsoft.WindowsAzure.Storage.Blob;
using Microsoft.WindowsAzure.Storage;
using System.Configuration;
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer triggered function executed at: {DateTime.Now}");
```

```

log.Info($"Timer schedule: {myTimer.Schedule}");
log.Info($"Timer last execution: {myTimer.ScheduleStatus.Last}");
log.Info($"Timer last execution: {myTimer.ScheduleStatus.Next}");

string conn =
    ConfigurationManager.AppSettings["functionappsuccb139_STORAGE"];

CloudStorageAccount storageAccount = CloudStorageAccount.Parse(conn);
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
CloudBlobContainer container =
    blobClient.GetContainerReference("receipts");

DateTime oldestTime =
    DateTime.Now.Subtract(TimeSpan.FromMinutes(5));

log.Info($"Checking for old receipts");

foreach(CloudBlockBlob blob in
    container.ListBlobs().OfType<CloudBlockBlob>())
{
    var isOld = blob.Properties.LastModified < oldestTime;
    if (isOld)
    {
        log.Info($"Blob deleted: {blob.Name}");
        blob.Delete();
    }
}
}

```

Let's save this code to check that there are no compilation errors.

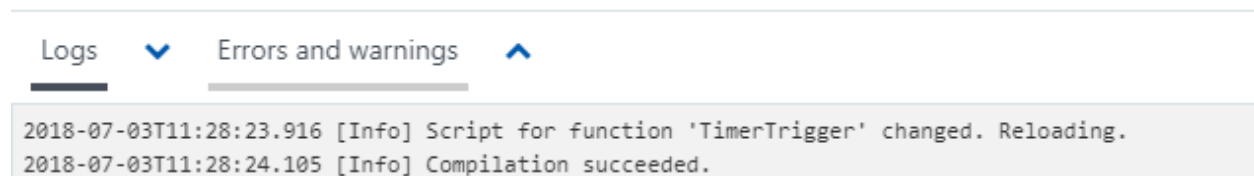
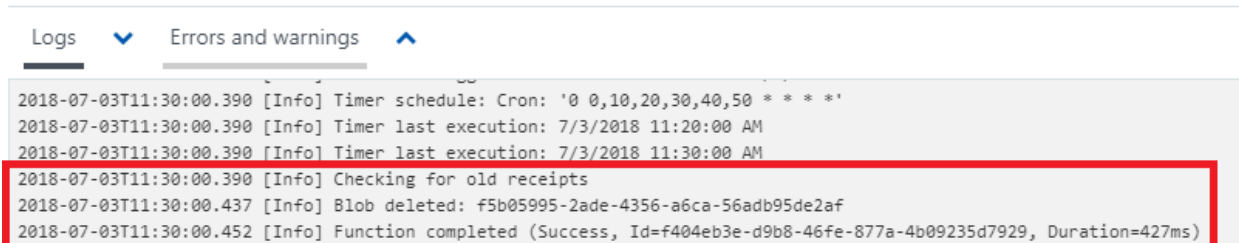


Figure 3-u: Compilation Succeeded

Awesome, no compilation errors. Now all we need to do is wait for the timer to execute so it can do its job.

If you now check the **receipts** blob container using the Azure Storage Explorer application, you might still find a blob there. If so, check the blob container again in 10 minutes, and you'll see that the blob has been deleted.

Here's the output from my tests.



```
Logs ▾ Errors and warnings ▲
2018-07-03T11:30:00.390 [Info] Timer schedule: Cron: '0 0,10,20,30,40,50 * * * *'
2018-07-03T11:30:00.390 [Info] Timer last execution: 7/3/2018 11:20:00 AM
2018-07-03T11:30:00.390 [Info] Timer last execution: 7/3/2018 11:30:00 AM
2018-07-03T11:30:00.390 [Info] Checking for old receipts
2018-07-03T11:30:00.437 [Info] Blob deleted: f5b05995-2ade-4356-a6ca-56adb95de2af
2018-07-03T11:30:00.452 [Info] Function completed (Success, Id=f404eb3e-d9b8-46fe-877a-4b09235d7929, Duration=427ms)
```

Figure 3-v: Output with a Blob Deletion Entry

In my case, the function automatically executed and deleted the blob entry that existed on the **receipts** blob container.

Summary

We started this chapter by exploring function metadata, and then we created a new blob-triggered function.

When we created the blob-triggered function, we set the path to **hello-requests** and used the special name inside braces, and we saw that the name of the blob is populated into the function parameter called **name**.

We also changed the incoming parameter from **stream** to **CloudBlockBlob**, and we saw how to access metadata about the incoming blob, such as the blob name.

We added some code to simulate sending a text message, and then wrote out a new blob that represented a message receipt.

After completing the blob-triggered function, we explored the basics of CRON expressions, which govern the execution of timer-triggered functions, and created a function that executes every ten minutes. This function does some housekeeping and removes blobs that are more than five minutes old.

As you've seen, this has been a very interesting chapter where we explored a variety of useful things to do with Azure Functions.

Next, we'll have a look at HTTP-triggered functions, which is a way of allowing the outside world to execute code within an Azure Functions application.

We'll learn how to create an HTTP trigger that posts some data and writes it to the queue that we have already set up. Sounds exciting, right?

Chapter 4 Working with HTTP Triggers

Quick intro

In addition to triggering functions manually, on a timer basis, or in response to a queue or blob events, we can also trigger them through HTTP calls.

In this chapter, we are going to explore how to create an HTTP webhook function that responds to an alert that's triggered when a new Azure resource group has been created.

We'll check how we can output that interaction to the log and write it to a blob entry. Let's get started.



Note: The code samples for this chapter are available for download on [GitHub](#).

Webhooks

In essence, webhooks allow us to get notifications from external third-party systems when events occur. The advantage of using webhooks is that rather than having to periodically poll third-party systems for changes, we can get those third-party systems to inform us about these changes when they happen.

For example, a webhook could be used if we have our own system and an external system—which could be of different types, such as an online shop or a code repository—and we wanted to get notifications whenever a customer buys a product.

To do this, our system can define an HTTP endpoint, which needs to be set up in such a way that it can respond correctly to the data that the third-party system passes to it.

We can then configure the third-party system to be aware of our HTTP endpoint. So, when a customer buys a product, the third-party system will call our HTTP endpoint, passing the sales info of the product sold.

Our system can then respond to this data in whichever way it needs to. In general, Azure Functions supports two major types of webhooks:

- **Generic webhook:** Not limited to any specific third-party service.
- **Specific trigger types:** Integrate with specific systems, so it's possible to have various webhooks for different types of systems.

Now that we know the two fundamental types of webhooks, let's go over to the Azure Portal to create a generic webhook trigger.

Generic webhook triggers

In the Azure Portal, click the **+** button to create a new Azure function. Then, select the **Generic webhook** template with **C#**.

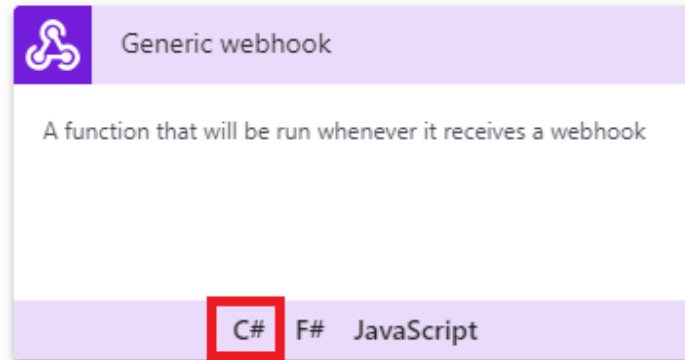


Figure 4-a: The Generic webhook Template

Once you have clicked the **C#** option, the following dialog box will appear. Here we can specify a **Name** and click **Create**.

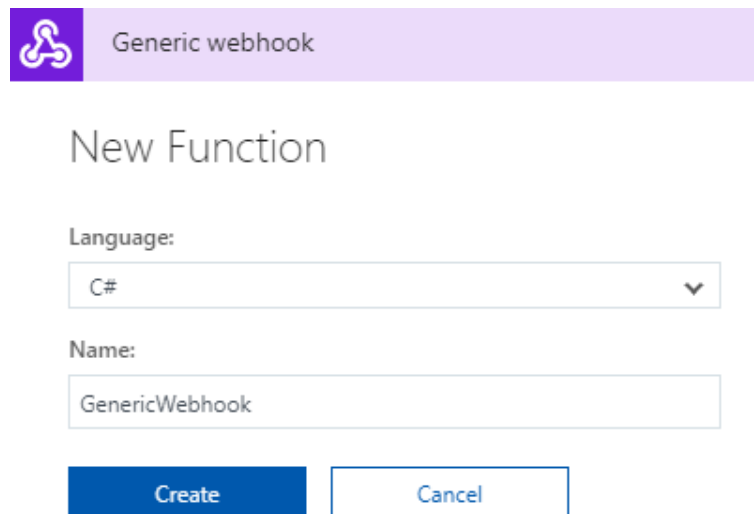


Figure 4-b: Creating a C# Generic Webhook

Once created, the code within **run.csx** will look as follows.

Listing 4-a: The run.csx for the Generic Webhook Function

```
#r "Newtonsoft.Json"

using System;
using System.Net;
```

```

using Newtonsoft.Json;

public static async Task<object> Run(HttpRequestMessage req, TraceWriter
log)
{
    log.Info($"Webhook was triggered!");
    string jsonContent = await req.Content.ReadAsStringAsync();
    dynamic data = JsonConvert.DeserializeObject(jsonContent);

    if (data.first == null || data.last == null)
    {
        return req.CreateResponse(HttpStatusCode.BadRequest, new
        {
            error =
                "Please pass first/last properties in the input object"
        });
    }
    return req.CreateResponse(HttpStatusCode.OK, new
    {
        greeting = $"Hello {data.first} {data.last}!"
    });
}

```

Now let's go to the **Integrate** option of this function—notice that the configuration looks very similar to the one from an HTTP trigger function.

HTTP trigger [delete](#)

Allowed HTTP methods ⓘ
All methods ▼

Request parameter name ⓘ
req

Mode ⓘ
Webhook ▼

Route template ⓘ
Route template

Webhook type ⓘ
Generic JSON ▼

Figure 4-c: Generic Webhook Function Configuration

Note that the function **Mode** setting is set to **Webhook**, and the **Webhook type** is set to **Generic JSON**.

If you expand the **Webhook type**, we can see that we also have options for **GitHub** and **Slack**.



Figure 4-d: Webhook type Options

This means that besides being able to create a **Generic webhook**, we can also create webhooks that integrate with **GitHub** or **Slack**.

Webhook function URL

As we'll see shortly, we'll need the public URL of the Azure function we've just created for the example we'll be using. You can obtain the function URL by clicking the **Get function URL** link, as shown in the following figure.

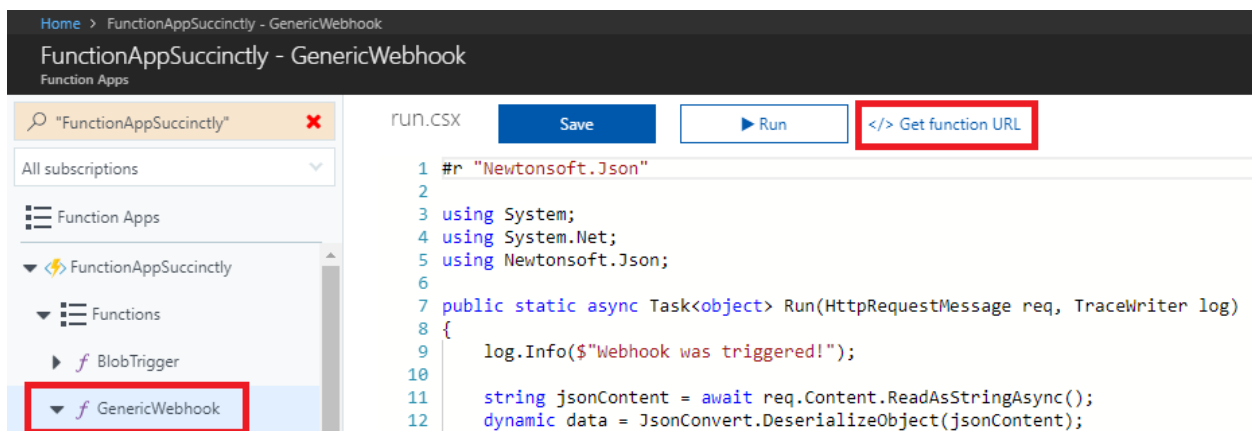


Figure 4-e: The Get function URL Link

Once you have clicked **Get function URL**, the following dialog box will appear, where we can copy the function URL.

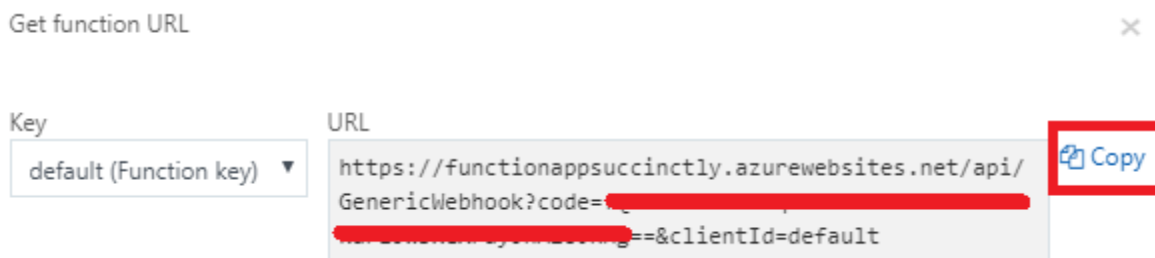


Figure 4-f: The Function URL

Copy it to the system clipboard, as we'll need it shortly.

Creating an Azure activity alert

The purpose of the **Generic webhook** function is that it can be triggered by an alert raised by the Azure Monitor service when an Azure resource group is added to your subscription.

So, in the Azure Portal, navigate to the **Monitor Service**, choose **Alerts (classic)**, and then click **Add activity log alert**, as shown in the following figure.

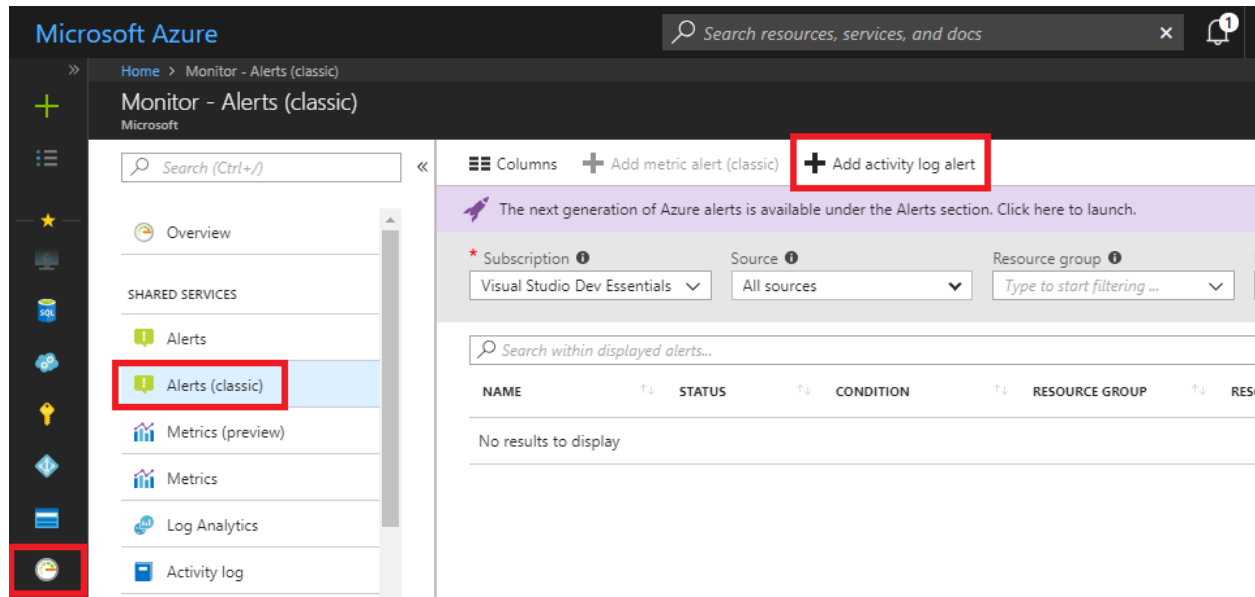


Figure 4-g: Add activity log alert

Once you've clicked **Add activity log alert**, we'll have to enter some information to create the alert. First, we'll provide some basic details, such as the **Activity log alert name**, **Subscription**, and **Resource group**.

For the **Activity log alert name**, you can choose any name you want. I've selected the **FunctionAppSuccinctly** as a **Resource group**, which I already have available within my Azure subscription.

The screenshot shows a dialog box titled 'Add activity log alert'. It contains four main input fields: 1. 'Activity log alert name' with the value 'resource-alert' and a green checkmark icon. 2. 'Description' which is an empty text box. 3. 'Subscription' with a dropdown menu showing 'Visual Studio Dev Essentials'. 4. 'Resource group' with a dropdown menu showing 'FunctionAppSuccinctly'.

Figure 4-h: Adding Basic Info for Activity Log Alert

After entering these basic details, we'll need to specify the **Criteria** details, as shown in the following figure.

Criteria

* Event category ⓘ Administrative ▼

Resource type ⓘ Resource groups (Microsoft.Resources/subscriptions/resourceGroups) ▼

Resource group ⓘ All ▼

Resource ⓘ All ▼

Operation name ⓘ Create Resource Group (subscriptions/resourceGroups) ▼

Level ⓘ Informational ▼

Status ⓘ Succeeded ▼

Event initiated by ⓘ

Figure 4-i: Adding Criteria Info for Activity Log Alert

The **Criteria** section is actually the most important part needed to create an alert. For the **Event category**, it's important to select the **Administrative** option.

The **Resource type** parameter has been set to the **Resource groups** option. The **Resource group** parameter needs to be set to **All**.

The **Resource** parameter has been set to **All**. The **Operation name** parameter has been set to **Create Resource Group**.

The **Level** parameter has been set to **Informational**—this is because we only want to get alerts that are informational, and not errors or warnings.

After specifying the **Criteria** data, we have to also specify the **Alert via** parameters, which we can see as follows.

Alert via

Action group ☒ New ☐ Existing

* Action group name ⓘ function-webhook ✓

* Short name ⓘ funcwebhook ✓

Figure 4-j: Adding Alert via Info for Activity Log Alert

I've selected the **New** option for the **Action group** parameter—this is because we don't have an existing one.

As for the **Action group name** and **Short name**, I've entered two names that indicate exactly what we are creating.

With all these details entered, we've reached the most important part of the activity alert—the **Actions** section. It's the most important section because it is how we are going to link the activity alert with the webhook function we created.

To enter the data required in the **Actions** section, we need to use the URL of the webhook function we created, which you should have already copied.

Under **ACTION NAME**, enter a valid name. I've entered the value **CallWebhook**. For the **ACTION TYPE**, I've selected **Webhook** from the list of options available.

Once you have selected the option **Webhook**, you'll be presented with an **Edit details** option, which we'll have to click to enter the **URI** of the webhook function. This can be seen in the following screenshot.

The screenshot shows the 'Add activity log alert' interface with a 'Webhook' configuration pane open on the right. The main pane has a breadcrumb trail: 'Home > Monitor - Alerts (classic) > Add activity log alert > Webhook'. The 'Action group' is set to 'New'. The 'Action group name' is 'function-webhook' and the 'Short name' is 'funcwebhook'. The 'Actions' section has a table with columns: ACTION NAME, ACTION TYPE, STATUS, and DETAILS. The first row shows 'CallWebhook' as the action name, 'Webhook' as the action type, and a green checkmark in the status column. A red box highlights the 'CallWebhook' and 'Webhook' cells. Below the table, a red message says 'Please configure the action by clicking the link.' and a red box highlights the 'Edit details' link. The 'Webhook' pane on the right has a title bar 'Webhook' and a close button. It contains the text 'Send a notification to this URI when an alert fires.' and a red box around the 'URI' field with the value 'https://functionappsuccinctly.azurewebsites.net...'. A blue 'OK' button is at the bottom of the pane.

ACTION NAME	ACTION TYPE	STATUS	DETAILS
CallWebhook	Webhook	✓	Edit details

Figure 4-k: Adding a Webhook in Adding an Activity Log Alert

Once you've entered the **URI**, click **OK**.

Finally, you'll return to the main **Add activity log alert** pane to finalize the creation of the activity alert. Click **OK**.

Alert via

Action group ☒ New ☐ Existing

* Action group name ? ✓

* Short name ? ✓

Actions

ACTION NAME	ACTION TYPE	STATUS	DETAILS
<input type="text" value="CallWebhook"/> ✓	<input type="text" value="Webhook"/> ▼		Edit details
<input type="text" value="Unique name for the ac..."/>	<input type="text" value=""/>		

? It can take up to 5 minutes for an Activity log alert to become active.

[Privacy Statement](#)

[Pricing](#)

OK

Figure 4-l: Finalizing the Activity Log Alert

Now you'll see the resource alert we just created under the **Alerts (classic)** list as follows.

Subscription ⓘ

Visual Studio Dev Essentials ▾

Source ⓘ

All sources ▾

Resource group ⓘ

Type to start filtering ... ▾

Resource type ⓘ

0 selected ▾

Resource ⓘ

Type to start filtering ... ▾

🔍 Search within displayed alerts...

NAME	STATUS	CONDITION	RESOURCE GROUP	RESOURCE	LAST FIRED
resource-alert	Active	category equals Admini...	FunctionAppSuccinctly	-	-

Figure 4-m: The Alerts (classic) List

The webhook is now called when a resource group is created in our Azure subscription.

Next, we need to update the code so our function can handle the JSON log data in the body of the request.

Updating the function code

Let's update our **run.csx** code so we can process the data whenever there's an activity alert triggered after a new resource group is created.

Listing 4-b: The Updated run.csx for the Generic Webhook Function

```
#r "Newtonsoft.Json"

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

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

    // Get the activityLog from the JSON in the message body.
    string jsonContent = await req.Content.ReadAsStringAsync();

    JToken activityLog = JObject.Parse(jsonContent.ToString())
        .SelectToken("data.context.activityLog");

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

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

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

Once you've made these changes to the code, click **Save** to save the changes and make sure there are no compilation errors.

Let's analyze this code. The first thing that happens is the function reads the **Content** property as a **string**, received through the **HttpRequestMessage** object.

That **content** value is then parsed in order to retrieve the **data.context.activityLog** object from the activity alert.

Next, we need to check if the **activityLog** object actually is the resource group. This is done by checking if the **resourceType** property of the **activityLog** object contains this string value **Microsoft.Resources/subscriptions/resourceGroups**.

If the **activityLog** object is not of the correct **resourceType**, then an error is returned as the response of the **Run** method.

If the **activityLog** object is of the correct type, then we can retrieve the details of this object through its properties, such as **resourceGroupName**, **subStatus**, and **submissionTimestamp**, and output this to the log.

Testing the webhook function

Now that we have the code ready, let's test things out. We can do this by creating a new resource group within the Azure Portal.

We can do this by clicking the **Add** button under **Resource groups**, as shown in the following figure.

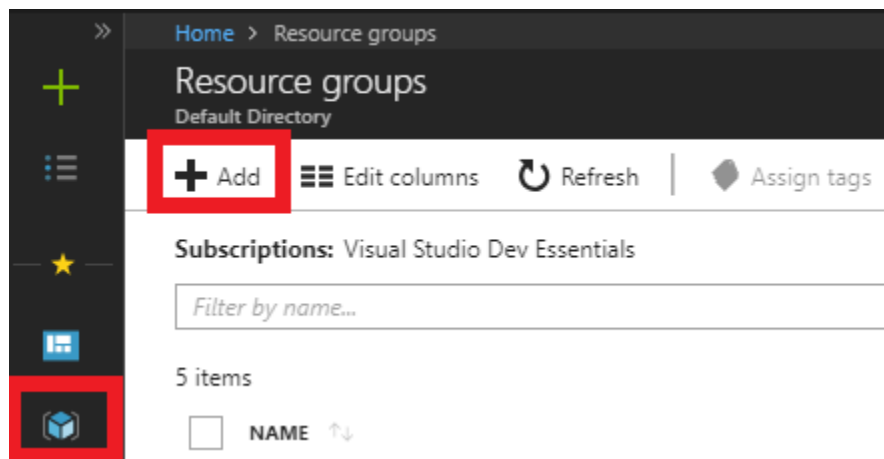


Figure 4-n: Creating a New Resource Group

Once you have clicked **Add**, you'll be asked to enter some details about the resource group, which can be seen as follows.

Resource group
Create an empty resource group

* Resource group name
webhook-resource

* Subscription
Visual Studio Dev Essentials

* Resource group location
East US

Create

Figure 4-o: New Resource Group Details

Basically, all that needs to be added is the **Resource group name**, and unless you want to select a different **Subscription** or a different **Resource group location** than the default one, you can click **Create**.

However, before you click **Create**, open a new browser tab and on the Azure Portal, go to the **GenericWebhook** function and make sure that the code is visible, and also that the **Logs** view is just below the code.

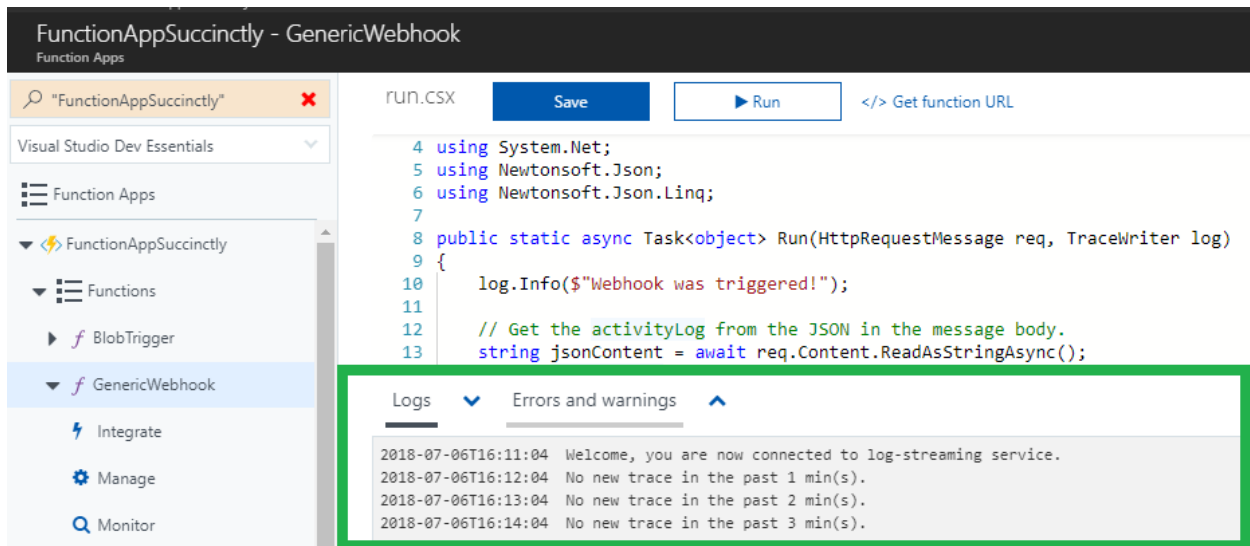


Figure 4-p: The Log View for the GenericWebhook Function

Now, go back to your other tab—the one where you have your **Resource group** ready to be created—and click **Create**. Then, on the **GenericWebhook** tab, within the **Logs** view, you should be able to see the function output a few seconds later. Here's what it looked like when I ran it.



Figure 4-q: The Log View for the GenericWebhook Function

Notice how the webhook function was triggered, and that the resource group created has been correctly identified and written to the log.

Webhook blob output

Now that we have achieved this milestone, it's time to modify our webhook function to output the same result, but to a blob rather than to the log. This is actually more useful, but a bit trickier.

To do that, let's modify the webhook function by setting the same output we previously defined for the QueueTrigger function.

We can do this by going into the **Integrate** section of our webhook function and creating a new **Azure Blob Storage output** type as follows.

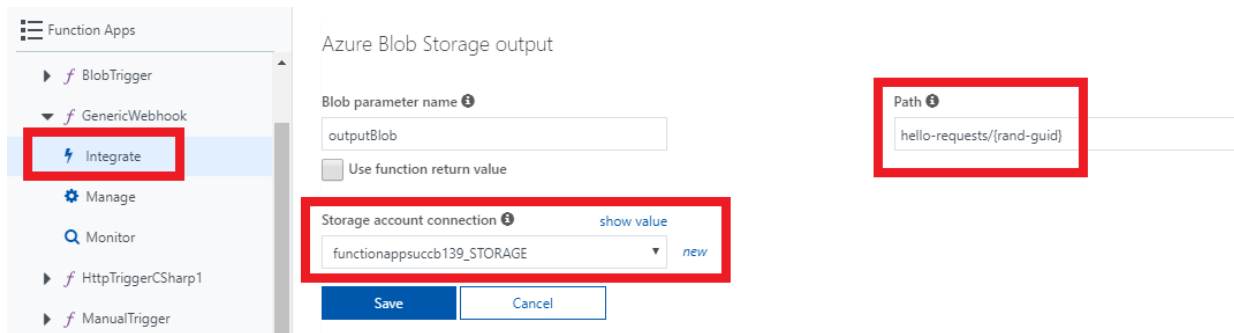


Figure 4-r: Creating a Blob Output for the Webhook Function

Make sure you specify the correct **Path** value by setting it to the **hello-requests** blob we've always used, and select the correct **Storage account connection**. You can leave the **Blob parameter name** as the default option.

Once you have made these modifications, click **Save**. Now it's time to make modifications to our **run.csx** code. Here's where things get interesting.

I've placed the updated code in the following listing. Please have a good look at it. You'll notice some significant differences, which I've highlighted in bold.

Listing 4-c: The Updated run.csx for the Generic Webhook Function

```
#load "..\SharedCode\HelloRequest.csx"
#r "Newtonsoft.Json"

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

public static object Run(HttpRequestMessage req, TraceWriter log,
    out string outputBlob)
{
    outputBlob = string.Empty;
    string jsonContent = string.Empty;
    JObject activityLog = null;

    log.Info($"Webhook was triggered!");

    Task.Run(async () => {
        jsonContent = await req.Content.ReadAsStringAsync();
        activityLog = JObject.Parse(jsonContent.ToString())
            .SelectToken("data.context.activityLog");
    });
}
```

```

        if (activityLog == null ||
            !string.Equals((string)activityLog["resourceType"],
                "Microsoft.Resources/subscriptions/resourceGroups"))
        {
            log.Error("An error occurred");
            activityLog = null;
        }
    });

    Thread.Sleep(500);

    if (activityLog == null)
        return req.CreateResponse(HttpStatusCode.BadRequest, new
            {
                error = "Unexpected payload or wrong alert received."
            });
    else
    {
        var helloRequest = new HelloRequest
        {
            Number = (string)activityLog["resourceGroupName"],
            Message =
                string.Format("Resource group '{0}' was {1} on {2}.",
                    (string)activityLog["resourceGroupName"],
                    ((string)activityLog["subStatus"]).ToLower(),
                    (DateTime)activityLog["submissionTimestamp"])
        };

        outputBlob = JsonConvert.SerializeObject(helloRequest);
    }

    return req.CreateResponse(HttpStatusCode.OK);
}

```

Let's analyze this code to understand what is going on.

First, we invoke the **HelloRequest** class from the **ShareCode** function we previously wrote. We do this because we'll use the **HelloRequest** class as a way to write to the blob container.

Next, we've added the **System.Threading** namespace, because we'll need to pause the execution of the code for 500 milliseconds (half a second) to retrieve the alert activity details, which are obtained from the call to the **req.Content.ReadAsStringAsync** method.

Notice that the **Run** method was previously marked with the **async** keyword, and now it is no longer used. This is because in order to write to a blob container, the **Run** method cannot be **async**—and it is also necessary to have the **outputBlob** added as a parameter. In other words, having an **async Run** method with an **out string outputBlob** parameter is incompatible.

However, we still need to be able to manage having a **Run** method with some async functionality—which is responsible for invoking the **req.Content.ReadAsStringAsync** method and getting the resource group data—and at the same time, be able to return the **outputBlob** parameter, which is required to write to the blob container.

To combine these two functionalities, I've had to wrap the call to the **req.Content.ReadAsStringAsync** method, invoked through **Task.Run**, around an **async** anonymous function.

Because the execution of the code within **Task.Run** is asynchronous and the rest of the code that follows is synchronous, we have to wait until the data from the created resource group is available before we can write it to the blob container. This is why we invoke **Thread.Sleep**: to give the resource data enough time to become available, following the execution of the **async** code.

Once the resource group data is available (after **Thread.Sleep** is done), we can write this result to the blob container by creating an instance of the **HelloRequest** class, serializing it, and assigning it to the **outputBlob** variable.

Like always, click **Save** to save the changes and make sure there are no compilation errors.

To test if this works, create a new **resource group**, exactly like we did previously. Then, using the **Microsoft Azure Storage Explorer**, go to the **hello-requests** blob container and look for the most recent blob entry. The following screenshot was taken when I tested this.

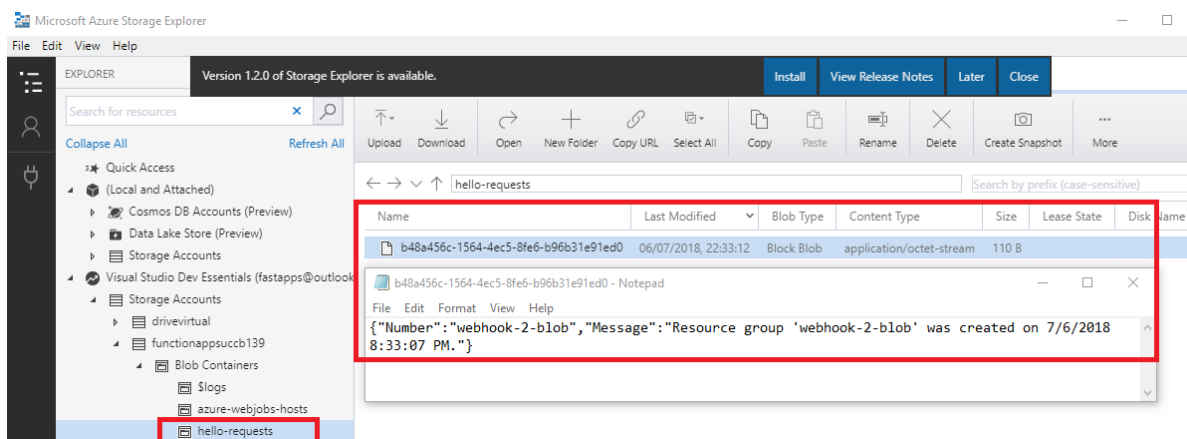


Figure 4-s: The Resource Group Written to the Blob Container

You can clearly see that the blob entry was successfully written to the blob container when the new resource group was created.

In my opinion, this is a really cool example, and shows the possibilities of what can be done with Azure Functions when integrating different services.

Code summary

With our webhook example finished and our objective accomplished, I think it is worthwhile to review the code for each of the Azure functions that we've developed and written throughout this book.

The following listing is the code for each of the functions we created.

Listing 4-d: The Final BlobTrigger run.csx Code

```
#load "..\SharedCode\HelloRequest.csx"
#load "..\SharedCode\MsgSentConfirmation.csx"

#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name,
                      TraceWriter log, Stream outputBlob)
{
    log.Info($"Metadata Name: {myBlob.Name}");
    log.Info($"Metadata StorageUri: {myBlob.StorageUri}");
    log.Info($"Metadata Container: {myBlob.Container.Name}");

    HelloRequest helloRequest = GetHelloRequest(myBlob);
    log.Info($"Hello Request: {helloRequest}");

    string id = SendMessage(helloRequest);

    var confirm = new MsgSentConfirmation
    {
        ReceiptId = id,
        Number = helloRequest.Number,
        Message = helloRequest.Message
    };

    UploadMsg(confirm, outputBlob);
}

public static void UploadMsg(MsgSentConfirmation confirm, Stream
outputBlob)
{
    using (var w = new StreamWriter(outputBlob))
    {
        using (var jw = new JsonTextWriter(w))
        {
            JsonSerializer s = new JsonSerializer();
```

```

        s.Serialize(jw, confirm);
        jw.Flush();
    }
}

public static string SendMessage>HelloRequest req)
{
    // We simulate sending SMS with req and returning a unique GUID
    return Guid.NewGuid().ToString();
}

public static>HelloRequest GetHelloRequest(CloudBlockBlob blob)
{
   >HelloRequest helloRequest;

    using (var ms = new MemoryStream())
    {
        blob.DownloadToStream(ms);
        ms.Position = 0;

        using (var res = new StreamReader(ms))
        {
            using (var jtr = new JsonTextReader(res))
            {
                var s = new JsonSerializer();
                helloRequest = s.Deserialize>HelloRequest>(jtr);
            }
        }
    }

    return helloRequest;
}

```

Listing 4-e: The Final GenericWebhook run.csx Code

```

#load "..\SharedCode\HelloRequest.csx"
#r "Newtonsoft.Json"

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

public static object Run(HttpRequestMessage req, TraceWriter log,
                        out string outputBlob)

```

```

{
    outputBlob = string.Empty;
    string jsonContent = string.Empty;
    JToken activityLog = null;

    log.Info($"Webhook was triggered!");

    Task.Run(async () => {
        jsonContent = await req.Content.ReadAsStringAsync();
        activityLog = JObject.Parse(jsonContent.ToString())
            .SelectToken("data.context.activityLog");

        if (activityLog == null ||
            !string.Equals((string)activityLog["resourceType"],
                "Microsoft.Resources/subscriptions/resourceGroups"))
        {
            log.Error("An error occurred");
            activityLog = null;
        }
    });

    Thread.Sleep(500);

    if (activityLog == null)
        return req.CreateResponse(HttpStatusCode.BadRequest, new
        {
            error =
                "Unexpected message payload or wrong alert received."
        });
    else {
        var helloRequest = new HelloRequest
        {
            Number = (string)activityLog["resourceGroupName"],
            Message = string.
                Format("Resource group '{0}' was {1} on {2}.",
                    (string)activityLog["resourceGroupName"],
                    ((string)activityLog["subStatus"]).ToLower(),
                    (DateTime)activityLog["submissionTimestamp"]);
        };

        outputBlob = JsonConvert.SerializeObject(helloRequest);
    }

    return req.CreateResponse(HttpStatusCode.OK);
}

```

Listing 4-f: The Final ManualTrigger run.csx Code

```
#load "..\SharedCode\CreateHelloRequest.csx"

using System;

public static void Run(CreateHelloRequest input, TraceWriter log,
    out CreateHelloRequest outputQueueItem)
{
    log.Info(
        $"C# manually triggered function called with input: {input}");
    outputQueueItem = input;
}
```

Listing 4-g: The Final QueueTrigger run.csx Code

```
#load "..\SharedCode\CreateHelloRequest.csx"
#load "..\SharedCode\HelloRequest.csx"

#r "Newtonsoft.Json"

using System;
using Newtonsoft.Json;

public static void Run (CreateHelloRequest myQueueItem, TraceWriter log,
    out string outputBlob,
    DateTimeOffset insertionTime,
    string id)
{
    log.Info(
        $"C# Queue trigger function processed: {myQueueItem}");

    log.Info($"InsertionTime: {insertionTime}");
    log.Info($"Id: {id}");

    var helloRequest = new HelloRequest
    {
        Number = myQueueItem.Number,
        Message = GenerateHello(myQueueItem.FirstName)
    };

    outputBlob = JsonConvert.SerializeObject(helloRequest);
}

private static string GenerateHello (string firstName)
{
    string hello;
    int hourOfDay = DateTime.Now.Hour;
```

```

    if (hourOfDay <= 12)
        hello = "The Morning...";
    else if (hourOfDay <= 18)
        hello = "The Afternoon...";
    else
        hello = "The Evening...";

    return $"{hello} {firstName}";
}

```

Listing 4-h: The Final SharedCode run.csx Code

```

using System;

public static void Run(string input, TraceWriter log)
{
    log.Info(
        $"C# manually triggered function called with input: {input}");
}

```

Listing 4-i: The Final CreateHelloRequest.csx Code

```

public class CreateHelloRequest
{
    public string Number;
    public string FirstName;

    public override string ToString() => $"{FirstName} {Number}";
}

```

Listing 4-j: The Final HelloRequest.csx Code

```

public class HelloRequest
{
    public string Number;
    public string Message;

    public override string ToString() => $"{Number} {Message}";
}

```

Listing 4-k: The Final MsgSentConfirmation.csx Code

```

public class MsgSentConfirmation
{
    public string Number;
    public string Message;
    public string ReceiptId;

    public override string ToString() =>
        $"{ReceiptId} {Number} {Message}";
}

```

Listing 4-1: The Final TimerTrigger run.csx Code

```

#r "Microsoft.WindowsAzure.Storage"
#r "System.Configuration"

using Microsoft.WindowsAzure.Storage.Blob;
using Microsoft.WindowsAzure.Storage;
using System.Configuration;
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer triggered function executed at: {DateTime.Now}");

    log.Info($"Timer schedule: {myTimer.Schedule}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Last}");
    log.Info($"Timer last execution: {myTimer.ScheduleStatus.Next}");

    string conn =
        ConfigurationManager.AppSettings["functionappsucb139_STORAGE"];

    CloudStorageAccount storageAccount = CloudStorageAccount.Parse(conn);
    CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
    CloudBlobContainer container =
        blobClient.GetContainerReference("receipts");

    DateTime oldestTime =
        DateTime.Now.Subtract(TimeSpan.FromMinutes(5));

    log.Info($"Checking for old receipts");

    foreach(CloudBlockBlob blob in
        container.ListBlobs().OfType<CloudBlockBlob>())
    {
        var isOld = blob.Properties.LastModified < oldestTime;
        if (isOld)
        {

```

```
        log.Info($"Blob deleted: {blob.Name}");
        blob.Delete();
    }
}
```

That's all the code we developed throughout this book.

Summary

We've reached the end of this chapter, and also this book. We've had a good introduction on how to get started with Azure Functions by creating a function app, which included several functions that work seamlessly together.

We've barely scratched the surface of what is possible with this amazing technology, so there is still a lot to explore and learn. I'd like to invite you to expand your curiosity and dig a bit further into other Azure Functions-related topics, such as:

- Service bus triggers.
- Event hub triggers.
- Developing and debugging Azure functions in Visual Studio.
- Chaining functions together.
- Durable Azure functions.
- Azure Functions with Logic Apps.
- Monitoring workflows and error-handling.

As you can see, there's a wealth of Azure Functions-related topics to be explored.

The number of applications that Azure Functions can be used for is practically unlimited—not only is it a versatile technology, but it is also one with lots of useful applications in real-world scenarios.

In my opinion, Azure Functions is a great complement to other technologies, such as Azure Logic Apps, which I also encourage you to explore as it's a great way to connect multiple systems together.

Finally, the core principle of Azure Functions is to make monolithic systems easier to decouple, simplify processes, and provide a mechanism that allows unrelated systems to interact with each other—all without having to worry about server provisioning, security, or infrastructure.

Thanks for reading and taking the time to explore Azure Functions. I hope that this book serves as a source of inspiration, so you can continue to expand your learning path into this amazing technology.

All the best,

Ed