Azure Serverless Computing Cookbook

Second Edition

Build and monitor Azure applications hosted on serverless architecture using Azure Functions

Packt
www.packt.com

Praveen Kumar Sreeram

Azure Serverless Computing Cookbook
Second Edition

Build and monitor Azure applications hosted on serverless architecture using Azure Functions

Praveen Kumar Sreeram

BIRMINGHAM - MUMBAI

Azure Serverless Computing Cookbook
Second Edition

Commissioning Editor: Vijin Boricha
Acquisition Editor: Shrilekha Inani
Content Development Editor: Nithin George Varghese
Technical Editor: Komal Karne
Copy Editor: Safis Editing
Project Coordinator: Drashti Panchal
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Tom Scaria
Production Coordinator: Aparna Bhagat

First published: August 2017
Second edition: November 2018

Production reference: 1301118

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78961-526-5

www.packtpub.com

It would have not been possible to complete the book without the support of my best half, my wife, Haritha, and my cute little angel, Rithwika Sreeram.

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Praveen Kumar Sreeram works as an Azure architect, trainer in a leading MNC. He has over 14 years of experience in the field of development, analysis, design, and delivery of applications, including custom web development using ASP.NET and MVC to building mobile apps using Xamarin for domains such as insurance, telecom, and wireless expense management. He has been recognized twice as the MVP by one of the leading social community websites, CSharpCorner. He is an avid blogger who writes about his learning at his personal blog, called `Praveen Kumar Sreeram`. His current focus is on analyzing business problems and providing technical solutions for various projects related to Microsoft Azure and .NET Core. His twitter ID is `@PrawinSreeram`.

First of all, my thanks go to the Packt Publishing team, including Shrilekha Inani, Nithin George Varghese, and Komal Karne.

I would like to thank my grandma, Neelavatamma; dad, Kamalakar and mom, Seetha; for being in my life and giving me courage all the time.

I would like to express my deepest gratitude to Bhagyamma (my grandmother), Kamala Kumar (my maternal uncle), his brothers, and rest of the family, who have been supporting me all the time.

About the reviewers

Kasam Shaikh, Microsoft Azure enthusiast, is a seasoned professional with a "Could be" attitude, with 10 years of industry experience working as a cloud architect with one of the leading IT companies in Mumbai, India. He is a certified Azure architect, and has been recognized as an MVP by a leading online community, and is also a global AI speaker. He has authored books on Azure Cognitive, Azure Bots, and Microsoft Bot Frameworks. He leads the Azure India (AZINDIA) community, the fastest growing online community for learning Azure. He is also a founder of the Dear Azure website.

First and foremost, I would like to thank the Almighty Allah, my family, and especially my better half, for motivating me throughout this process. I am highly grateful to Packt Publishing for believing in me and for considering me for this opportunity.

Michael Sync (**Soe Htike**) is a senior engineer working at Readify in Australia. He was an MVP (Microsoft Valuable Professional) for 7 years. He participated as a speaker, mentor, and helper at several community events. He is also an author and a tech book reviewer. He has published two books with Manning and has reviewed several books for a number of publishers. He has been working in the software industry for more than 16 years.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Microsoft provides a solution to easily run small segments of code in the cloud with Azure Functions. Azure Functions provides solutions for processing data, integrating systems, and building simple APIs and microservices.

The book starts with intermediate-level recipes on serverless computing, along with some use cases on the benefits and key features of Azure Functions. Then, we'll deep dive into the core aspects of Azure Functions, such as the services it provides, how you can develop and write Azure Functions, and how to monitor and troubleshoot Azure Functions.

Moving on, you'll get practical recipes on integrating DevOps with Azure Functions, and providing continuous deployment with Azure DevOps (formerly Visual Studio Team Services). The book also provides hands-on steps and tutorials based on real-world serverless use cases to guide you through configuring and setting up your serverless environments with ease. Finally, you'll see how to manage Azure Functions, providing enterprise-level security and compliance to your serverless code architecture.

You will also learn how to quickly build applications that are reliable and durable using Durable Functions, with an example of a very common real-time use case.

By the end of this book, you will have all the skills required to work with serverless code architectures, providing continuous delivery to your users.

Who this book is for

If you are a cloud administrator, architect, or developer who wants to build scalable systems and deploy serverless applications with Azure Functions, then the *Azure Serverless Computing Cookbook* is for you.

What this book covers

`Chapter 1`, *Developing Cloud Applications Using Function Triggers and Bindings*, goes through how the Azure Functions runtime provides templates that can be used to quickly integrate different Azure services for your application needs. It reduces all of the plumbing code so that you can focus on just your application logic. In this chapter, you will learn how to build web APIs and bindings related to Azure Storage Services.

`Chapter 2`, *Working with Notifications Using the SendGrid and Twilio Services*, deals with how communication is one of the most critical aspects of any business requirement. In this chapter, you will learn how easy it is to connect your business requirements written in Azure Functions with the most popular communication services, such as SendGrid (for email) and Twilio (for SMS).

`Chapter 3`, *Seamless Integration of Azure Functions with Azure Services*, discusses how Azure provides many connectors that you could leverage to integrate your business applications with other systems pretty easily. In this chapter, you will learn how to integrate Azure Functions with cognitive services and Logic Apps.

`Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools for Azure Functions*, teaches you how to develop Azure Functions using Visual Studio, which provides you with many features such as Intellisense, local and remote debugging, and most of the regular development features.

`Chapter 5`, *Exploring Testing Tools for the Validation of Azure Functions*, helps you to understand different tools and processes that help you streamline your development and quality control processes. You will also learn how to create loads using Azure DevOps (formerly VSTS) load testing, and you'll look at how to monitor the performance of Azure Functions using the reports provided by Application Insights. Finally, you will also learn how to configure alerts that notify you when your apps are not responsive.

`Chapter 6`, *Monitoring and Troubleshooting Azure Serverless Services*, teaches you how to continuously monitor applications, analyze the performance, and review the logs to understand whether there are any issues that end users are facing. Azure provides us with multiple tools to achieve all the monitoring requirements, right from the development and maintenance stages of the application.

`Chapter 7`, *Developing Reliable Serverless Applications Using Durable Functions*, shows you how to develop long-running, stateful solutions in serverless environments using Durable Functions, which has advanced features that have been released as an extension to Azure Functions.

`Chapter 8`, *Bulk Import of Data Using Azure Durable Functions and Cosmos DB*, teaches you how to leverage Azure Durable Functions to read and import the data from the Blob storage and dump the data into Cosmos DB.

`Chapter 9`, *Implementing Best Practices for Azure Functions*, teaches a few of the best practices that you should follow in order to improve performance and security while working in Azure Functions.

`Chapter 10`, *Configuring of Serverless Applications in the Production Environment*, demonstrates how to deploy a function app in an efficient way and copy/move the configurations in a smarter way so as to avoid human error. You will also learn how to configure a custom domain that you could share with your customers or end users instead of the default domain that is created as part of provisioning the function app.

`Chapter 11`, *Implementing and Deploying Continuous Integration Using Azure DevOps*, helps you learn how to implement continuous integration and delivery of your Azure Functions code with the help of Visual Studio and Azure DevOps.

To get the most out of this book

Prior knowledge and hands-on experience with the core services of Microsoft Azure is required.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Azure-Serverless-Computing-Cookbook-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789615265_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

Any command-line input or output is written as follows:

```
docker tag functionsindocker
cookbookregistry.azurecr.io/functionsindocker:v1
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "During the installation, choose **Azure development** in the **Workloads** section."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to help you increase your knowledge of it.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Developing Cloud Applications Using Function Triggers and Bindings

In this chapter, we will cover the following recipes:

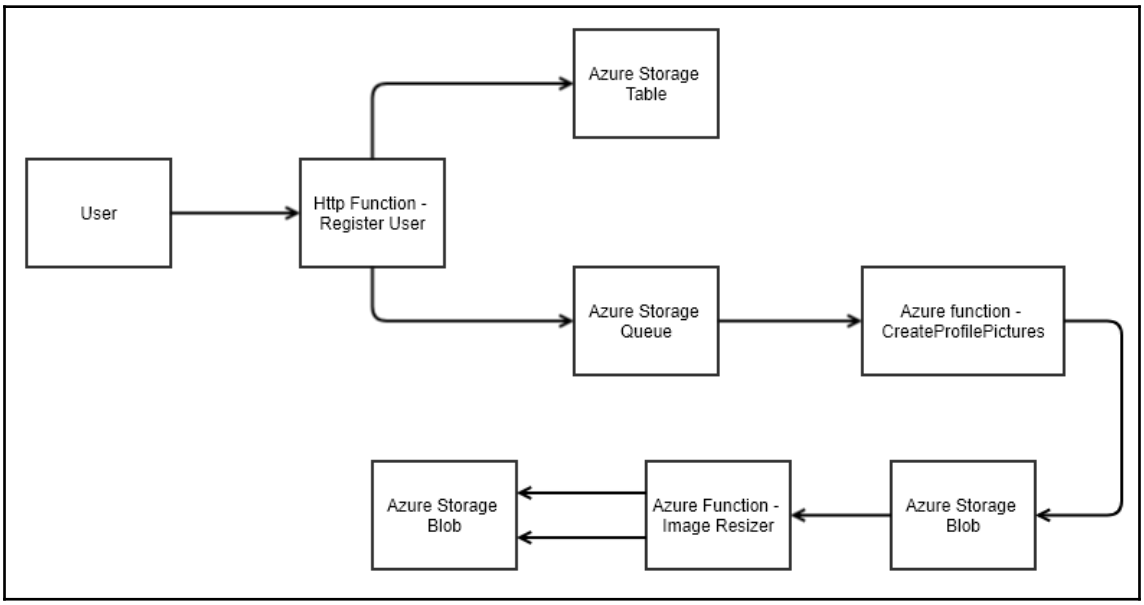
- Building a backend Web API using HTTP triggers
- Persisting employee details using Azure Storage table output bindings
- Saving the profile images to Queues using Queue output bindings
- Storing the image in Azure Blob Storage

Introduction

Every software application needs backend components that are responsible for taking care of the business logic and storing the data in some kind of storage, such as databases and filesystems. Each of these backend components could be developed using different technologies. Azure serverless technology also allows us to develop these backend APIs using Azure Functions.

Azure Functions provide many out-of-the-box templates that solve most common problems, such as connecting to storage, building Web APIs, and cropping images. In this chapter, we will learn how to use these built-in templates. Along with learning the concepts related to Azure serverless computing, we will also try to implement a solution to a basic domain problem of creating components required for any organization to manage the internal employee information.

The following is a simple diagram that helps you understand what we will achieve in this chapter:



Building a backend Web API using HTTP triggers

We will use Azure serverless architecture to build a Web API using HTTP triggers. These HTTP triggers could be consumed by any frontend application that is capable of making HTTP calls.

Getting ready

Let's start our journey of understanding Azure serverless computing using Azure Functions by creating a basic backend Web API that responds to HTTP requests:

- Refer to `https://azure.microsoft.com/en-in/free/?wt.mc_id=AID607363_SEM_8y6Q27AS` for creating a free Azure Account.
- Visit `https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-function-app-portal` to understand the step-by-step process of creating a function app, and `https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function` to create a function. While creating a function, a Storage Account is also created for storing all the files. Remember the name of the Storage Account, as it will be used later in the other chapters.
- Once you create the Function App, please go through the basic concepts of Triggers and Bindings which are the core concepts of how Azure Functions work. I highly recommend you to go through the `https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings` article before you proceed.

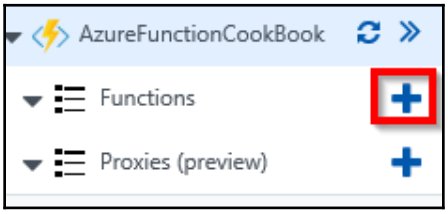
We will be using C# as the programming language throughout the book. Most of the functions are developed using Azure Functions V2 run-time. However, there are a few recipes which are not yet supported in V2 run-time which is mentioned in the respective recipe. Hopefully, by the time you are read this book, Microsoft will have made those features available for V2 run-time as well.

How to do it…

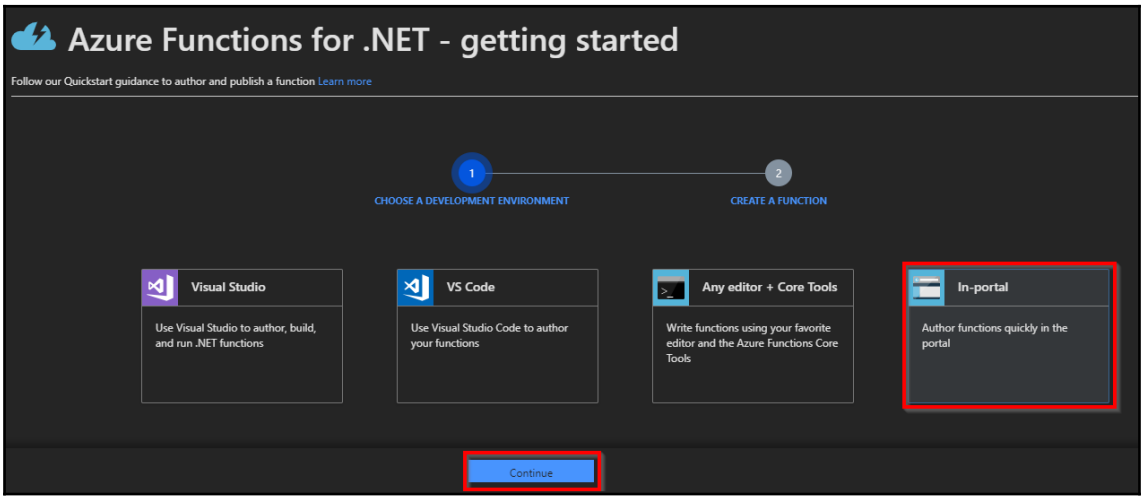
Perform the following steps:

1. Navigate to the **Function App** listing page by clicking on the **Function Apps** menu, which is available on the left hand side.

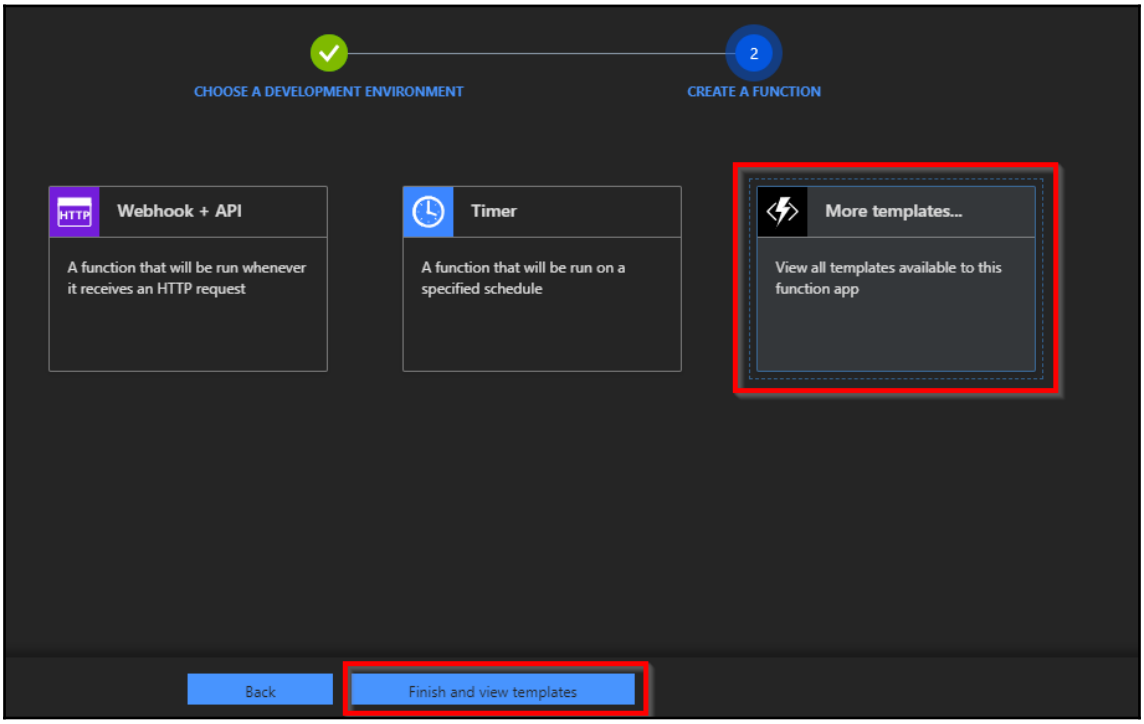
2. Create a new function by clicking on the + icon:



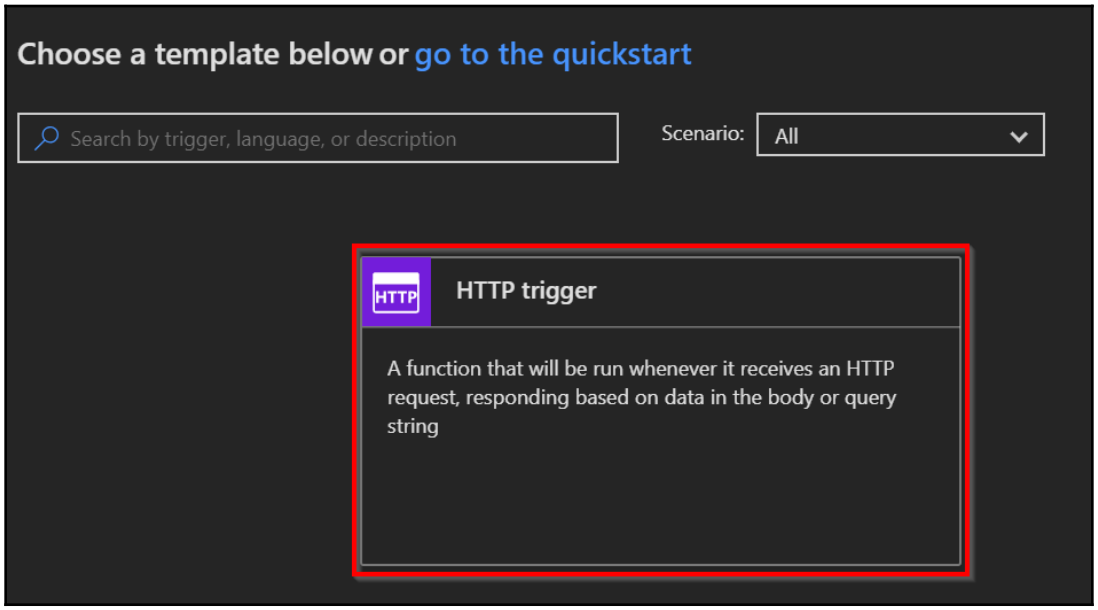
3. You will see the **Azure Functions for .NET - getting started** page where you will prompted to choose the type of tools you would like to use. You can choose the one you are interested in. For the initial few chapters, we will use the **In-portal** option where you can quickly create Azure Functions right from the portal without any tools. Later, in the other chapters, we will use Visual Studio and Azure Functions Core Tools to create the Functions.:



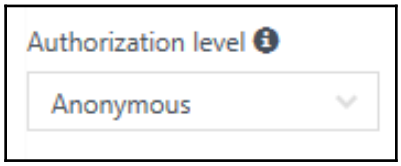
4. In the next step, select **More templates** and click on **Finish and view templates** as shown in the following screenshot:



5. In the **Choose a template below or go to the quickstart** section, choose **HTTP trigger** to create a new HTTP trigger function:



6. Provide a meaningful name. For this example, I have used `RegisterUser` as the name of the Azure Function.
7. In the **Authorization level** drop-down, choose the **Anonymous** option. We will learn more about the all the authorization levels in `Chapter 9`, *Implementing Best Practices for Azure Functions*:

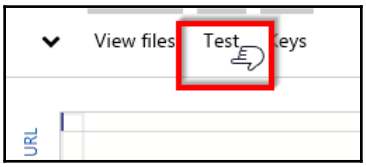


8. Click on the **Create** button to create the HTTP trigger function.

9. As soon as you create the function, all the required code and configuration files will be created automatically and the `run.csx` file will be opened for you to edit the code. Remove the default code and replace it with the following code. I have added two parameters (`firstname` and the `lastname`) that would be displayed in the output when the HTTP Trigger is triggered:

```
#r "Newtonsoft.Json"

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

public static async Task<IActionResult> Run(
                    HttpRequest req,
                    ILogger log)
{
  log.LogInformation("C# HTTP trigger function processed a
request.");
  string firstname=null,lastname = null;
  string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();

 dynamic inputJson = JsonConvert.DeserializeObject(requestBody);
 firstname = firstname ?? inputJson?.firstname;
 lastname = inputJson?.lastname;

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

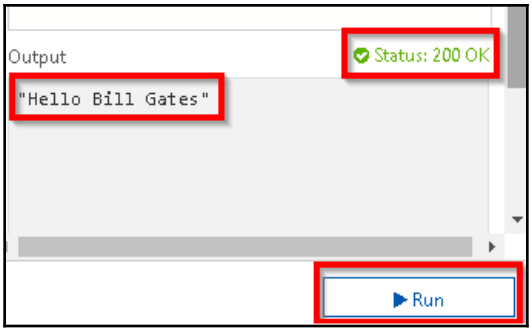
10. Save the changes by clicking on the **Save** button available just above the code editor.
11. Let's try to test the `RegisterUser` function using the **Test** console. Click on the **Test** tab to open the **Test** console:



12. Enter the values for `firstname` and `lastname` in the **Request body** section:

Make sure you select **POST** in the **HTTP method** drop-down.

13. Once you have reviewed the input parameters, click on the **Run** button available at the bottom of the **Test** console:



14. If the input request workload is passed correctly with all the required parameters, you will see a **Status 200 OK,** and the output in the **Output** window will be as shown in the preceding screenshot.

How it works…

We have created the first basic Azure Function using HTTP triggers and made a few modifications to the default code. The code just accepts the `firstname` and `lastname` parameters and prints the name of the end user with a `Hello {firstname} {lastname}` message as a response. We also learned how to test the HTTP trigger function right from the Azure Management portal.

For the sake of simplicity, I didn't perform validations of the input parameter. Make sure that you validate all the input parameters in your applications running on your production environment.

See also

The *Enabling authorization for function apps* recipe in `Chapter 9`, *Implementing Best Practices for Azure Functions*

Persisting employee details using Azure Storage table output bindings

In the previous recipe, you learned how to create an HTTP trigger and accept the input parameters. Let's now work on something interesting, that is, where you store the input data into a persistent medium. Azure Functions supports us to store data in many ways. For this example, we will store the data in Azure Table storage.

Getting ready

In this recipe, you will learn how easy it is to integrate an HTTP trigger and the **Azure Table storage** service using output bindings. The Azure HTTP trigger function receives the data from multiple sources and stores the user profile data in a storage table named `tblUserProfile`. We will follow the pre-requisites listed below:

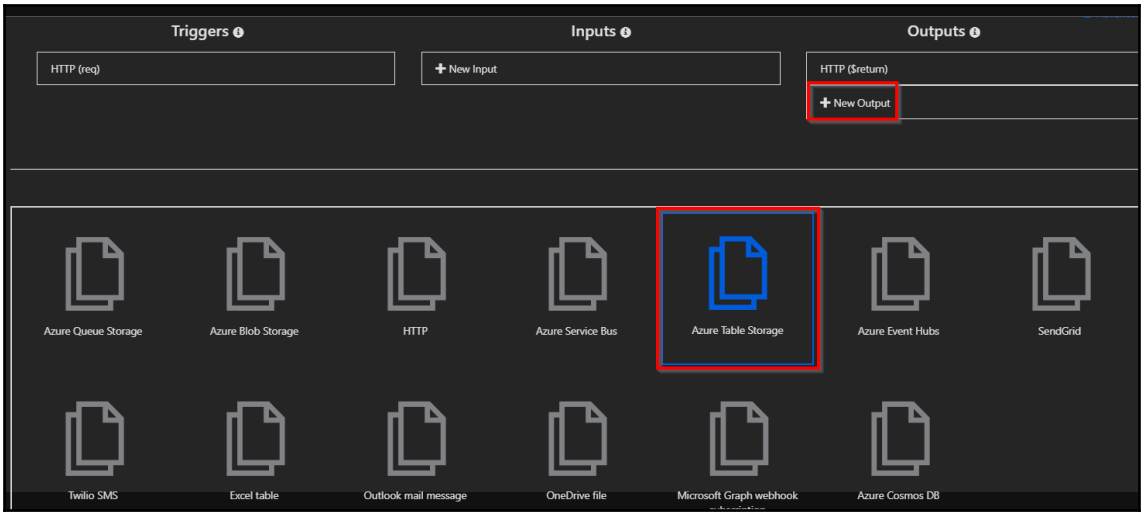
- For this recipe, we will use the same HTTP trigger that we created in our previous recipe.

- We will be using **Azure Storage Explorer**, which is a tool that helps us to work with the data stored in the Azure Storage account. You can download it from `http://storageexplorer.com/`.
- You can learn more about how to connect to the Storage Account using Azure Storage Explorer at `https://docs.microsoft.com/en-us/azure/vs-azure-tools-storage-manage-with-storage-explorer`.

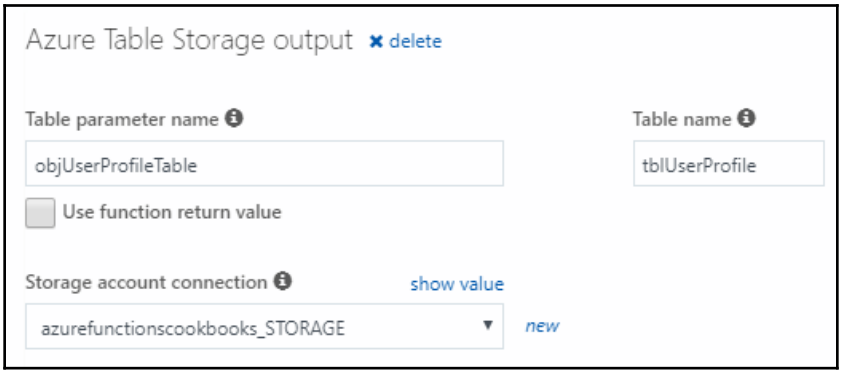
How to do it...

Perform the following steps:

1. Navigate to the **Integrate** tab of the `RegisterUser` HTTP trigger function.
2. Click on the **New Output** button, select **Azure Table Storage**, then click on the **Select** button:



3. You will be prompted to install the bindings, click on Install which would take a few minutes. Once the bindings are installed, choose the following settings of the Azure Table storage output bindings:
    - **Table parameter name**: This is the name of the parameter that you will be using in the `Run` method of the Azure Function. For this example, provide `objUserProfileTable` as the value.
    - **Table name**: A new table in Azure Table storage will be created to persist the data. If the table doesn't exist already, Azure will automatically create one for you! For this example, provide `tblUserProfile` as the table name.
    - **Storage account connection**: If you don't see the **Storage account connection** string, click on **new** (as shown in the following screenshot) to create a new one or to choose an existing storage account.
    - The Azure Table storage output bindings should be as shown in the following screenshot:



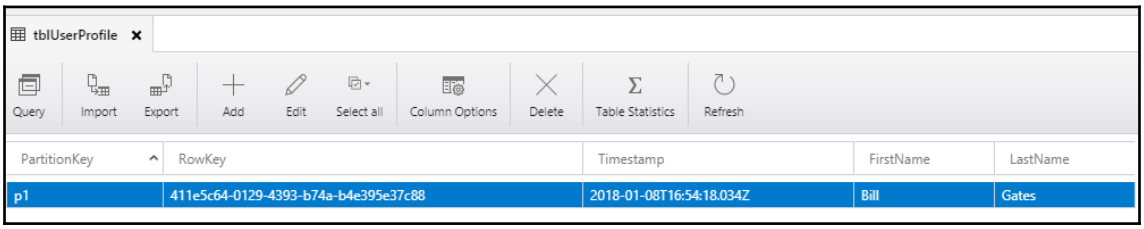
4. Click on **Save** to save the changes.
5. Navigate to the code editor by clicking on the function name and paste in the following code. The following code accepts the input passed by the end user and saves it in the Table Storage:

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

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Table;
```

```
public static async Task<IActionResult> Run(
 HttpRequest req,
 CloudTable objUserProfileTable,
 ILogger log)
{
  log.LogInformation("C# HTTP trigger function processed a
request.");
  string firstname=null,lastname = null;
  string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
 dynamic inputJson = JsonConvert.DeserializeObject(requestBody);
 firstname = firstname ?? inputJson?.firstname;
 lastname = inputJson?.lastname;
 UserProfile objUserProfile = new UserProfile(firstname, lastname);
 TableOperation objTblOperationInsert =
TableOperation.Insert(objUserProfile);
 await objUserProfileTable.ExecuteAsync(objTblOperationInsert);
 return (lastname + firstname) != null
 ? (ActionResult)new OkObjectResult($"Hello, {firstname + " " +
lastname}")
 : new BadRequestObjectResult("Please pass a name on the query" +
"string or in the request body");
}

class UserProfile : TableEntity
 {
    public UserProfile(string firstName,string lastName)
    {
       this.PartitionKey = "p1";
       this.RowKey = Guid.NewGuid().ToString();
       this.FirstName = firstName;
       this.LastName = lastName;
    }
    UserProfile() { }
    public string FirstName { get; set; }
    public string LastName { get; set; }
 }
```

6. Let's execute the function by clicking on the **Run** button of the **Test** tab by passing the `firstname` and `lastname` parameters in the **Request body**:

7. If everything went well, you should get a **Status 200 OK** message in the **Output** box as shown in the preceding screenshot. Let's navigate to Azure Storage Explorer and view the table storage to see whether the table named `tblUserProfile` was created successfully:



How it works...

Azure Functions allows us to easily integrate with other Azure services just by adding an output binding to the trigger. For this example, we have integrated the HTTP trigger with the Azure Storage table binding and also configured the Azure Storage account by providing the storage connection string and the Azure Storage table name in which we would like to create a record for each of the HTTP requests received by the HTTP trigger.

We have also added an additional parameter for handling the table storage, named `objUserProfileTable`, of the `CloudTable` type, to the `Run` method. We can perform all the operations on the Azure Table storage using `objUserProfileTable`.

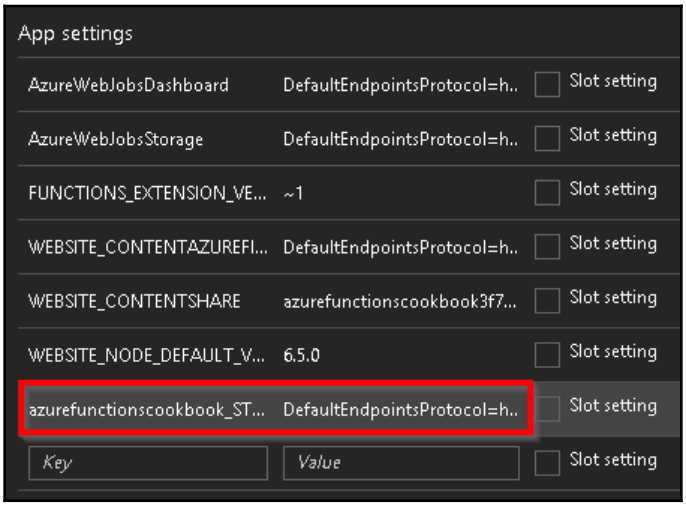
The input parameters are not validated in the code sample. However, in your production environment, it's important that you should validate them before storing them in any kind of persisting medium.

We have also created an `UserProfile` object, and filled it with the values received in the request object, and then passed it to a table operation.

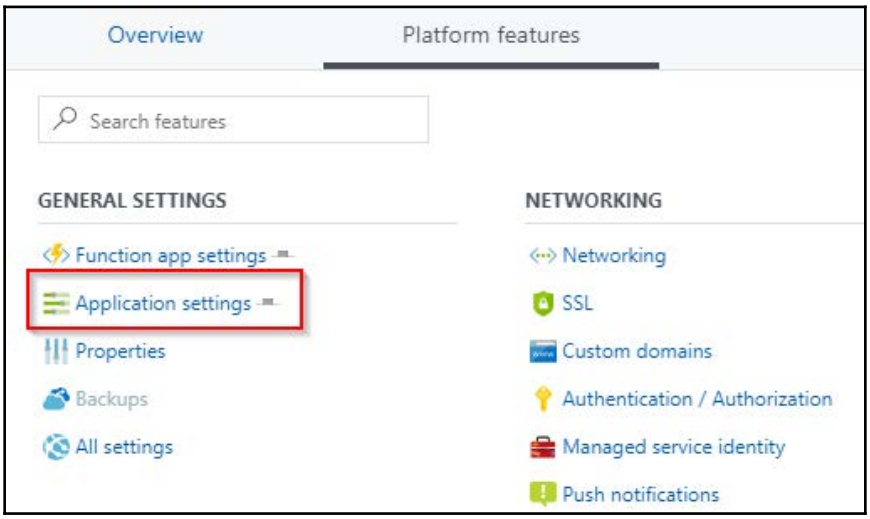
You can learn more about handling operations on the Azure Table storage service at `https://docs.microsoft.com/en-us/azure/storage/storage-dotnet-how-to-use-tables`.

Understanding storage connection

When you create a new storage connection (refer to the *step 3* of the *How to do it...* section of this recipe), a new **App settings** will be created:



You can navigate to the **App settings** by clicking on the **Application settings** menu available in the **GENERAL SETTINGS** section of the **Platform features** tab:



What is the Azure Table storage service?

The Azure Table storage service is a NoSQL key-value persistent medium for storing semi-structured data.

You can learn more about it at `https://azure.microsoft.com/en-in/services/storage/tables/`.

Partition key and row key

The primary key of the Azure Table storage table has two parts:

- **Partition key**: Azure Table storage records are classified and organized into partitions. Each record located in a partition will have the same partition key (`p1` in our example).
- **Row key**: A unique value should be assigned to each of the rows.

There's more...

The following is the very first lines of the code in this recipe:

```
#r "Newtonsoft.json"
#r "Microsoft.WindowsAzure.Storage"
```

The preceding lines of code instructs the runtime function to include a reference to the specified library to the current context.

Saving the profile images to Queues using Queue output bindings

In the previous recipe, you learned how to receive two string parameters, `firstname` and `lastname`, in the **Request body,** and store them in Azure Table storage. In this recipe, let's add a new parameter named `ProfilePicUrl` for the profile picture of the user that is publicly accessible via the internet. In this recipe, you will learn how to receive a URL of an image and save the URL in the Blob container of an Azure Storage account.

You might be thinking that the `ProfilePicUrl` input parameter could have been used to download the picture from the internet in the previous recipe, *Persisting employee details using Azure Storage table output bindings.* We didn't do it because the size of the profile pictures might be huge with the modern technology and so the processing of images on the fly in the HTTP requests might hinder the performance of the overall application. For that reason, we will just grab the URL of the profile picture and store it in Queue, and later we can process the image and store it in the Blob.

Getting ready

We will be updating the code of the `RegisterUser` function that we used in the previous recipes.

How to do it…

Perform the following steps:

1. Navigate to the **Integrate** tab of the `RegisterUser` HTTP trigger function.
2. Click on the **New Output** button, select **Azure Queue Storage**, and then click on the **Select** button.
3. Provide the following parameters in the **Azure Queue Storage output** settings:
    - **Message parameter name**: Set the name of the parameter to `objUserProfileQueueItem`, which will be used in the `Run` method
    - **Queue name**: Set the value of the Queue name as `userprofileimagesqueue`
    - **Storage account connection**: Make sure that you select the right storage account in the **Storage account connection** field
4. Click on **Save** to the create the new output binding.
5. Navigate back to the code editor by clicking on the function name (`RegisterUser` in this example) or the `run.csx` file and make the changes marked bold in the below code:

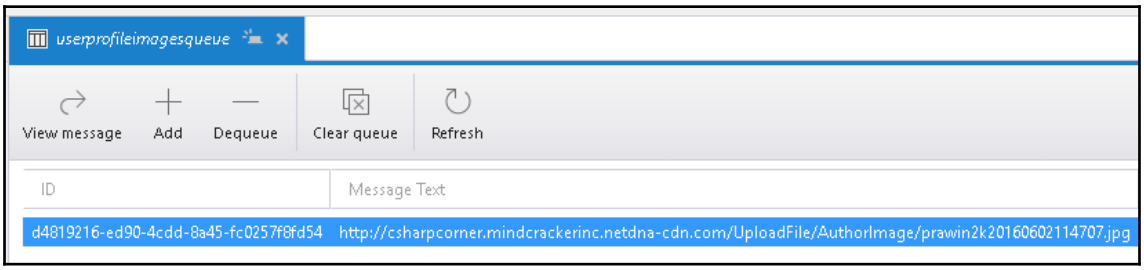
```
public static async Task<IActionResult> Run(
   HttpRequest req,
   CloudTable objUserProfileTable,
   IAsyncCollector<string> objUserProfileQueueItem,
   ILogger log)
   {
```

```
        ....
        string firstname= inputJson.firstname;
        string profilePicUrl = inputJson.ProfilePicUrl;
        await objUserProfileQueueItem.AddAsync(profilePicUrl);
        ....
        objUserProfileTable.Execute(objTblOperationInsert);
    }
```

6. In the preceding code, we have added Queue Output Bindings, by adding the `IAsyncCollecter` parameter to the `Run` method and just passing the required message to the `AddAsync` method, the output bindings will take care of saving the `ProfilePicUrl` into the **Queue**. Now, Click on **Save** to save the code changes in the code editor of the `run.csx` file.
7. Let's test the code by adding another parameter, `ProfilePicUrl`, to the **Request body** and then click on the **Run** button in the **Test** tab of the Azure Function code editor window. The image used in the following JSON might not exist when you are reading this book. So, make sure that you provide a valid URL of the image:

```
{
 "firstname": "Bill",
 "lastname": "Gates",
 "ProfilePicUrl":"https://upload.wikimedia.org/wikipedia/
commons/1/19/Bill_Gates_June_2015.jpg"
}
```

8. If everything goes fine you will see the **Status : 200 OK** message, then the image URL that you have passed as an input parameter in the **Request body** will be created as a Queue message in the Azure Storage Queue service. Let's navigate to Azure Storage Explorer, and view the Queue named `userprofileimagesqueue`, which is the Queue name that we provided in *step 3*. The following is a screenshot of the Queue message that was created:



How it works…

In this recipe, we added Queue message output binding and made the following changes to the code:

- Added a new parameter named `out string objUserProfileQueueItem`, which is used to bind the URL of the profile picture as a Queue message content
- Used the `AddAsync` method of the `IAsyncCollector` to the the `Run` method which saves the profile URL to the Queue as a Queue message.

Storing the image in Azure Blob Storage

In the previous recipe, we stored the image URL in the queue message. Let's learn how to trigger an Azure Function (Queue Trigger) when a new queue item is added to the Azure Storage Queue service. Each message in the Queue is the URL of the profile picture of a user, which will be processed by the Azure Functions and will be stored as a Blob in the Azure Storage Blob service.

Getting ready

In the previous recipe, we learned how to create Queue output bindings. In this recipe, you will grab the URL from the Queue, create a byte array, and then write it to a Blob.

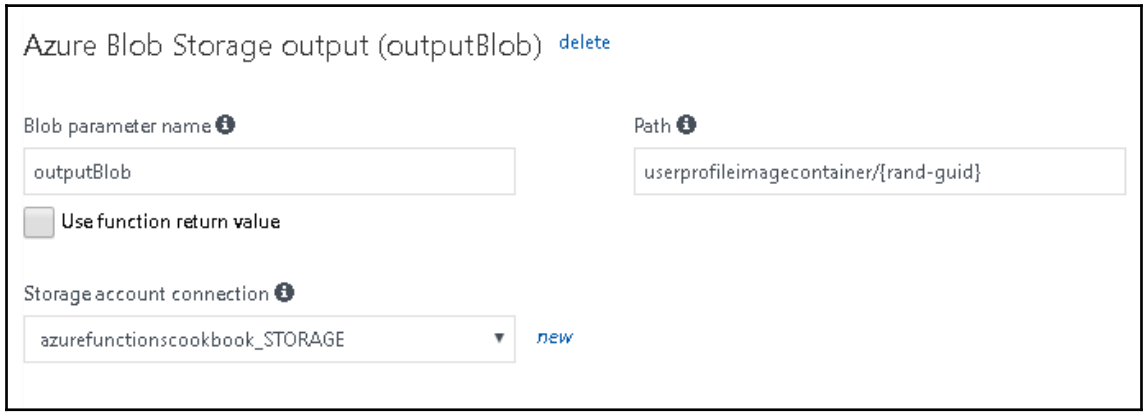
This recipe is a continuation of the previous recipes. Make sure that you have implemented them.

How to do it...

Perform the following steps:

1. Create a new Azure Function by choosing **Azure Queue Storage Trigger** from the templates.

2. Provide the following details after choosing the template:
    - **Name your function:** Provide a meaningful name, such as `CreateProfilePictures`.
    - **Queue name**: Name of the Queue `userprofileimagesqueue`. This will be monitored by the Azure Function. Our previous recipe created a new item for each of the valid requests coming to the HTTP trigger (named `RegisterUser`) into the `userprofileimagesqueue` Queue**.** For each new entry of a queue message to this Queue storage, the `CreateProfilePictures` trigger will be executed automatically.
    - **Storage account connection**: Connection of the storage account where the Queues are located.
3. Review all the details, and click on **Create** to create the new function.
4. Navigate to the **Integrate** tab, click on **New Output**, choose **Azure Blob Storage**, and then click on the **Select** button.
5. In the **Azure Blob Storage output** section, provide the following:
    - **Blob parameter name**: Set it to `outputBlob`
    - **Path**: Set it to `userprofileimagecontainer/{rand-guid}`
    - **Storage account connection**: Choose the storage account where you would like to save the Blobs and click on the **Save** button:

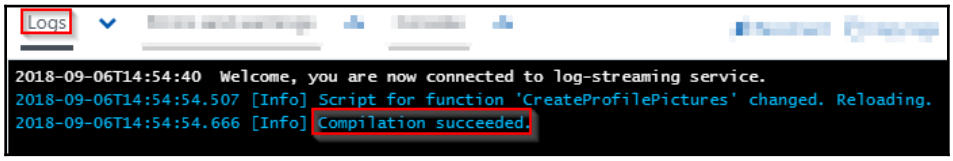


6. Click on the **Save** button to save all the changes.

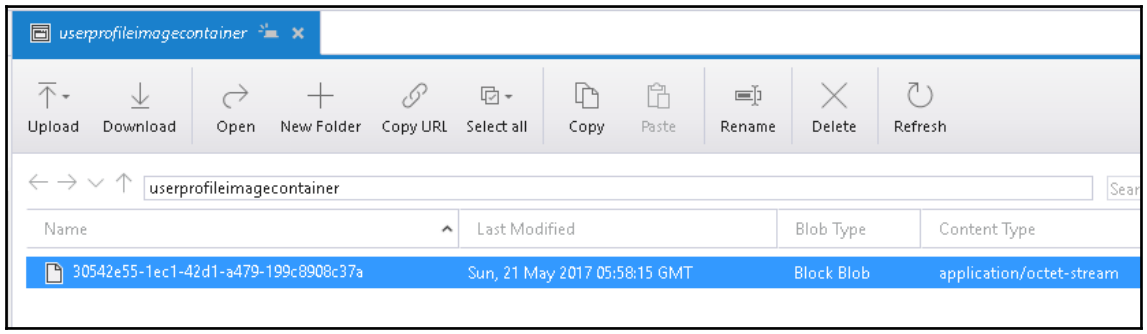
7. Replace the default code of the `run.csx` file of the `CreateProfilePictures` function with the following code. The following code grabs the URL from the Queue, creates a byte array, and then writes it to a Blob:

```
        using System;
        public static void Run(Stream outputBlob,string myQueueItem,
         TraceWriter log)
        {
           byte[] imageData = null;
using (var wc = new System.Net.WebClient())
{
imageData = wc.DownloadData(myQueueItem);
}
outputBlob.WriteAsync(imageData,0,imageData.Length);
        }
```

8. Click on the **Save** button to save the changes. Make sure that there are no compilation errors in the **Logs** window:



9. Let's go back to the `RegisterUser` function and test it by providing the `firstname`, `lastname`, and `ProfilePicUrl` fields as we did in the *Saving the profile images to Queues using Queue output bindings* recipe.
10. Navigate to the Azure Storage Explorer, and look at the `userprofileimagecontainer` Blob container. You will find a new Blob:



11. You can view the image in any tool (such as MS Paint or Internet Explorer).

How it works...

We have created a Queue trigger that gets executed as and when a new message arrives in the Queue. Once it finds a new Queue message, it reads the message, and as we know the message is a URL of a profile picture. The function makes a web client request, downloads the image data in the form of a byte array, and then writes the data into the Blob which is configured as an output Blob.

There's more...

The `rand-guid` parameter will generate a new GUID and is assigned to the Blob that gets created each time the trigger is fired.

It is mandatory to specify the Blob container name in the `Path` parameter of the Blob storage output binding while configuring the Blob storage output. Azure Functions creates one automatically if it doesn't exist.

You can use Queue messages only when you would like to store messages that are up to 64 KB. If you would like to store messages greater than 64 KB, you need to use the Azure Service Bus.

2
Working with Notifications Using the SendGrid and Twilio Services

In this chapter, we will look at the following:

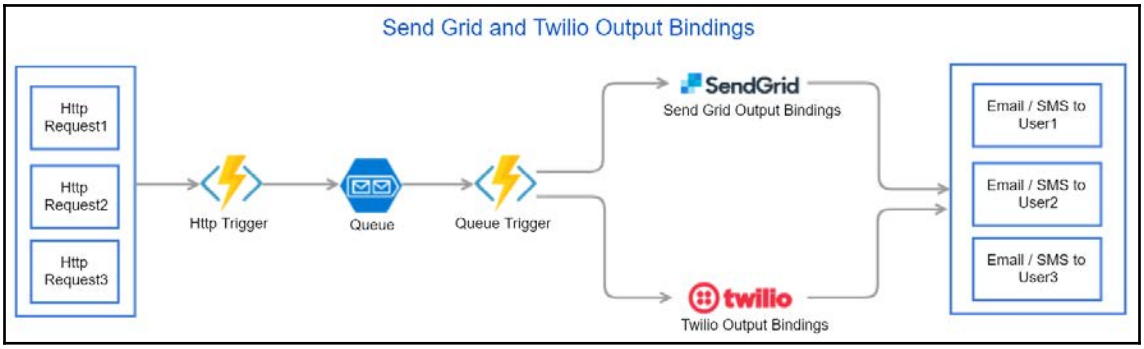
- Sending an email notification to the administrator of a website using the SendGrid service
- Sending an email notification dynamically to the end user
- Implementing email logging in Azure Blob Storage
- Modifying the email content to include an attachment
- Sending an SMS notification to the end user using the Twilio service

Introduction

For every business application to run its business operations smoothly, one of the key features is to have a reliable communication system between the business and its customers. The communication channel might be two-way, either by sending a message to the administrators managing the application or sending alerts to the customers via emails or SMS to their mobile phones.

Azure can integrate with two popular communication services: SendGrid for emails, and Twilio for working with SMS. In this chapter, we will be using both of these communication services to learn how to leverage their basic services to send messages between business administrators and end users.

Below is the architecture that we will be using for utilizing Send Grid and Twilio output bindings with Http Trigger and Queue Trigger.



Sending an email notification to the administrator of a website using the SendGrid service

In this recipe, you will learn how to create a **SendGrid** output binding and send an email notification, containing static content, to the website administrator. Our use case only involves one administrator, so we will be hardcoding the email address of the administrator in the **To address** field of the **SendGrid output (message)** binding.

Getting ready

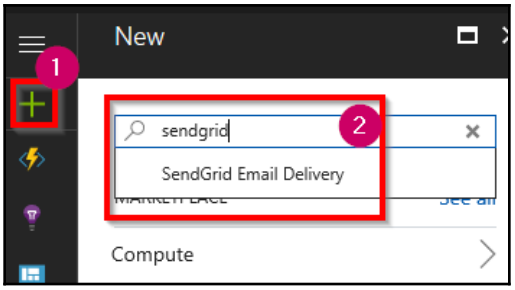
We will perform the following steps before moving on to the next section:

1. Creating a SendGrid account API key from the Azure Management portal
2. Generating an API key from the SendGrid portal
3. Configuring the SendGrid API key with the Azure Function app

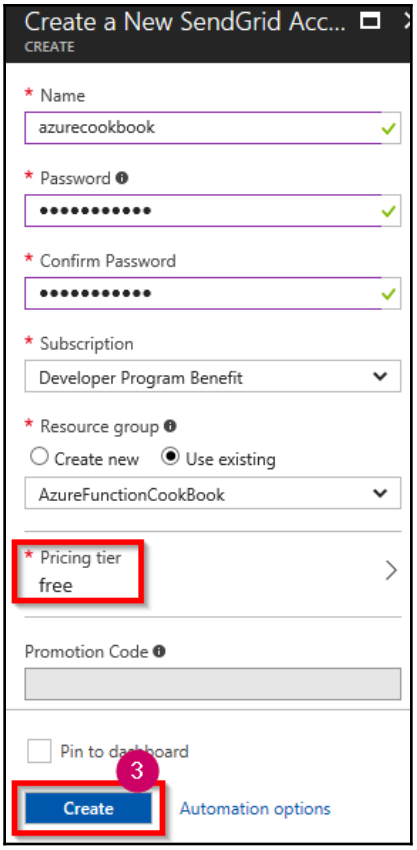
Creating a SendGrid account

Perform the following steps:

1. Navigate to Azure Management portal and create a **SendGrid Email Delivery** account by searching for it in the Marketplace, as shown in the following screenshot:

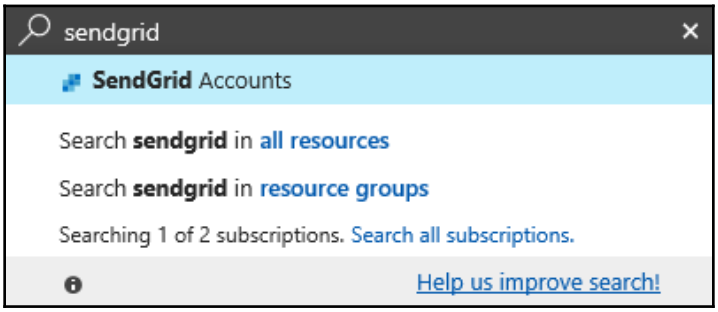


2. In the **SendGrid Email Delivery** blade, click on the **Create** button to navigate to **Create a New SendGrid Account**. Select **free** in the **Pricing tier** options, provide all the other details, and then click on the **Create** button, as shown in the following screenshot:

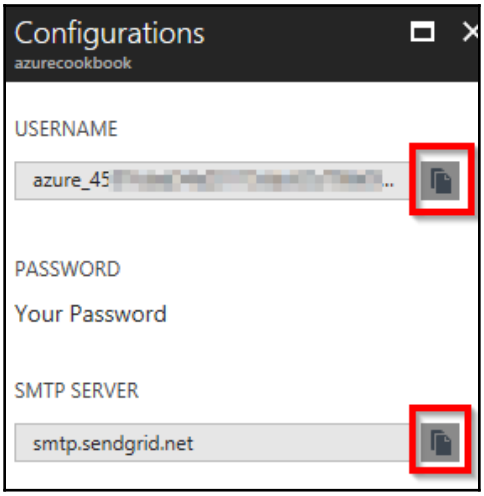


At the time of writing, the Send Grid free account allows us to send 25,000 free emails per month. If you would like to send more emails, then you can get a Silver S2-level account, where you would have 100,000 emails per month.

3. Once the account is created successfully, navigate to **SendGrid Accounts**. You can use the search box available on the top, as shown in the following screenshot:



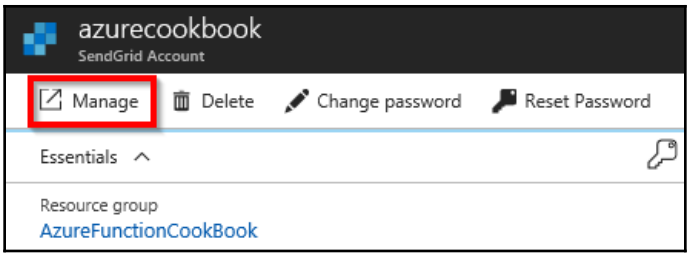
4. Navigate to **Settings**, choose **Configurations**, and grab **USERNAME** and **SMTP SERVER** from the **Configurations** blade, as shown in the following screenshot:



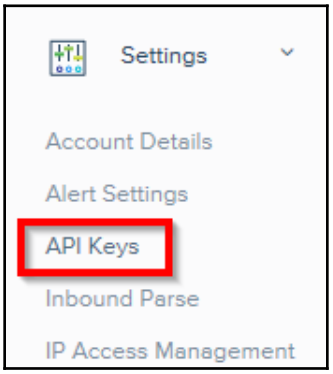
Generating an API key from the SendGrid portal

Perform the following steps:

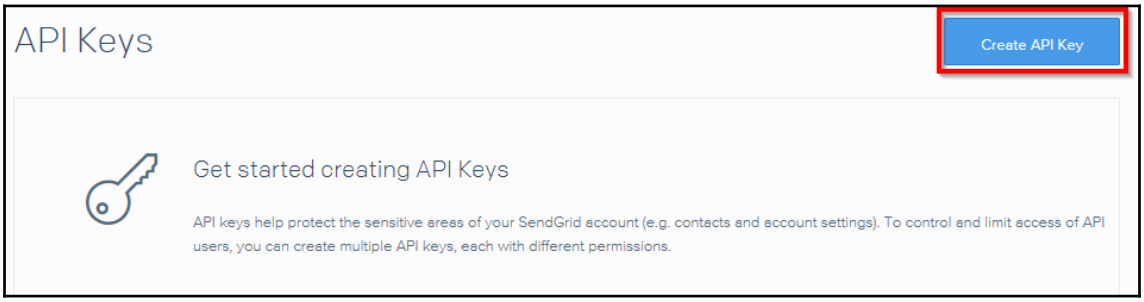
1. In order to utilize the SendGrid account in the Azure Functions runtime, we need to provide the SendGrid API key as an input for the Azure Functions. You can generate an API key from the SendGrid portal. Let's navigate to the SendGrid portal by clicking on the **Manage** button in the **Essentials** blade of **SendGrid Account**, as shown in the following screenshot:



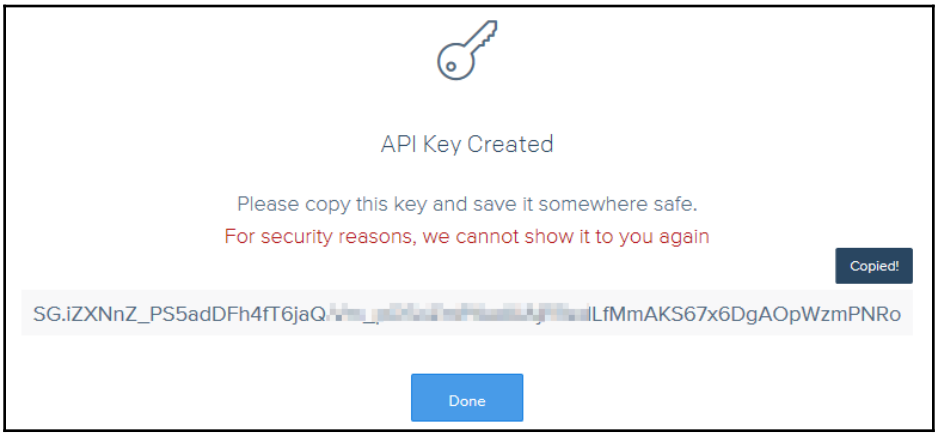
2. In the SendGrid portal, click on **API Keys** under the **Settings** section of the left-hand side menu, as shown in the following screenshot:



3. In the **API Keys** page, click on **Create API Key**, as shown in the following screenshot:



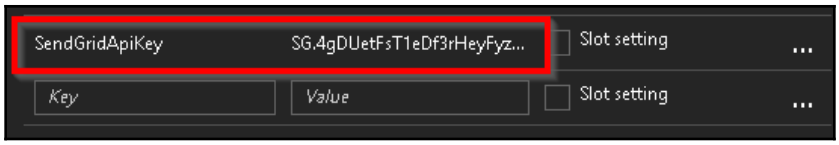
4. In the **Create API Key** popup, provide a name and choose **API Key Permissions**, and then click on the **Create & View** button.
5. After a moment, you will be able to see the API key. Click on the key to copy it to the clipboard, as shown in the following screenshot:



Configuring the SendGrid API key with the Azure Function app

Perform the following steps:

1. Create a new **App settings** configuration in the Azure Function app by navigating to the **Application settings** blade, under the **Platform features** section of the function app, as shown in the following screenshot:



2. Click on the **Save** button after adding the **App settings** from the preceding step.

How to do it...

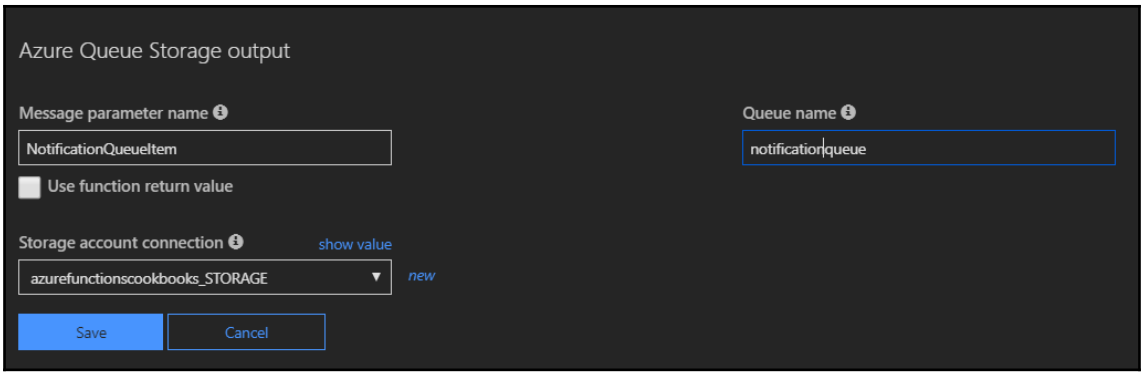
In this section, we will perform the following.

1. Create Storage Queue binding to the HTTP Trigger
2. Create Queue Trigger to process the message of the HTTP Trigger
3. Create SendGrid output binding to the Queue Trigger
4. Create Twilio output binding to the Queue Trigger

Create Storage Queue binding to the HTTP Trigger

Perform the following steps:

1. Navigate to the **Integrate** tab of the `RegisterUser` function and click on the **New Output** button to add a new output binding.
2. Choose **Azure Queue Storage** and click on **Select** button to add the binding and provide the values shown below and then click on **Save** button. Please make of the Queue name (in this case `notificationqueue`) which will be used in a moment.

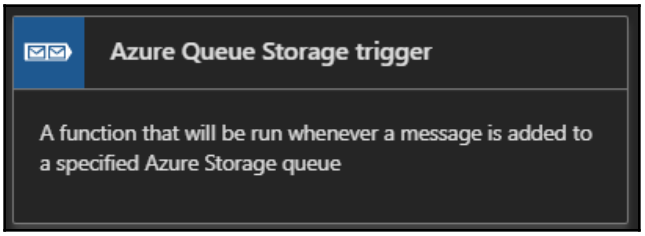


3. Navigate to the **Run** method of the `RegisterUser` function and make the following highlighted changes. We added another Queue output binding and add an empty message to trigger the Queue Trigger function. For now, we didn't add any message to the queue. We will make changes to the `NotificationQueueItem.AddAsync("");` method in the upcoming recipe of the chapter.

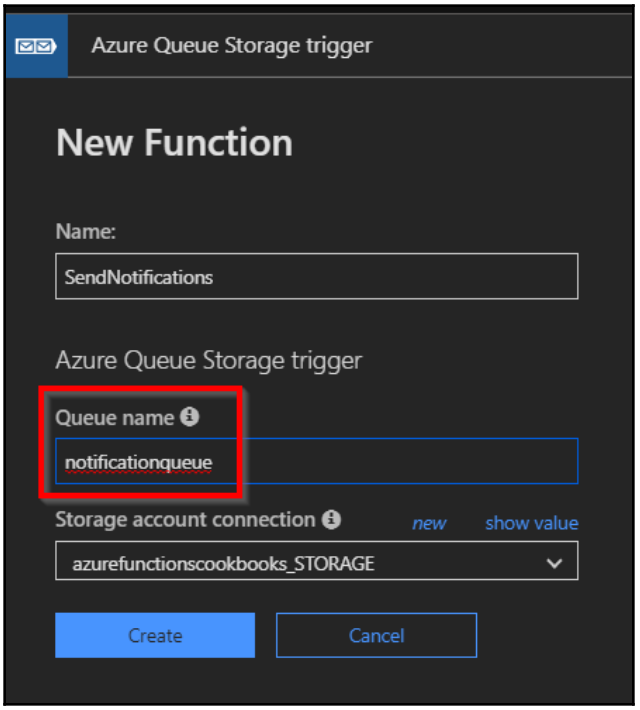
```
public static async Task<IActionResult> Run(
 HttpRequest req,
 CloudTable objUserProfileTable,
 IAsyncCollector<string> objUserProfileQueueItem,
 IAsyncCollector<string> NotificationQueueItem,
 ILogger log)
{
     log.LogInformation("C# HTTP trigger function processed a
request.");
    string firstname=null,lastname = null;
     ...
     ...
     await NotificationQueueItem.AddAsync("");
return (lastname + firstname) != null
        ? (ActionResult)new OkObjectResult($"Hello, {firstname + "
" + lastname}")
        : new BadRequestObjectResult("Please pass a name on the
query" +
         "string or in the request body");
}
```

Create Queue Trigger to process the message of the HTTP Trigger

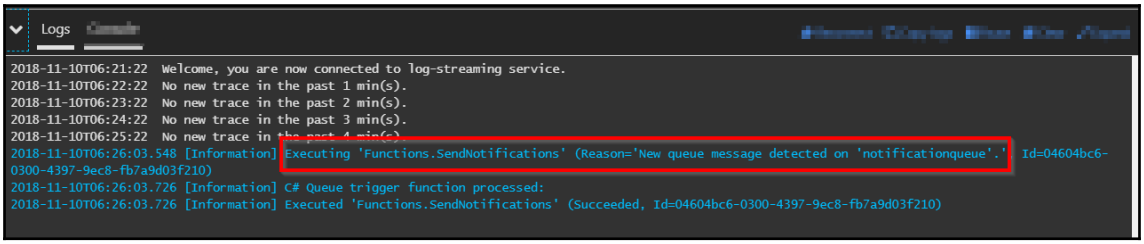
1. Create a **Azure Queue Storage Trigger** by choosing the template shown below.



2. In the next step, provide all the name of the Queue Trigger and provide the name of the queue which needs to be monitored for sending the notifications. Once you provide all the details, click on Create button to create the Function.



3. After creating the Queue Trigger function, let's run the `RegisterUser` function to see if the Queue trigger is getting invoked. Open the `RegisterUser` function in a new tab and test it by clicking on the **Run** button. In the Logs window of the `SendNotifications` tab, you should see something as shown as follows:

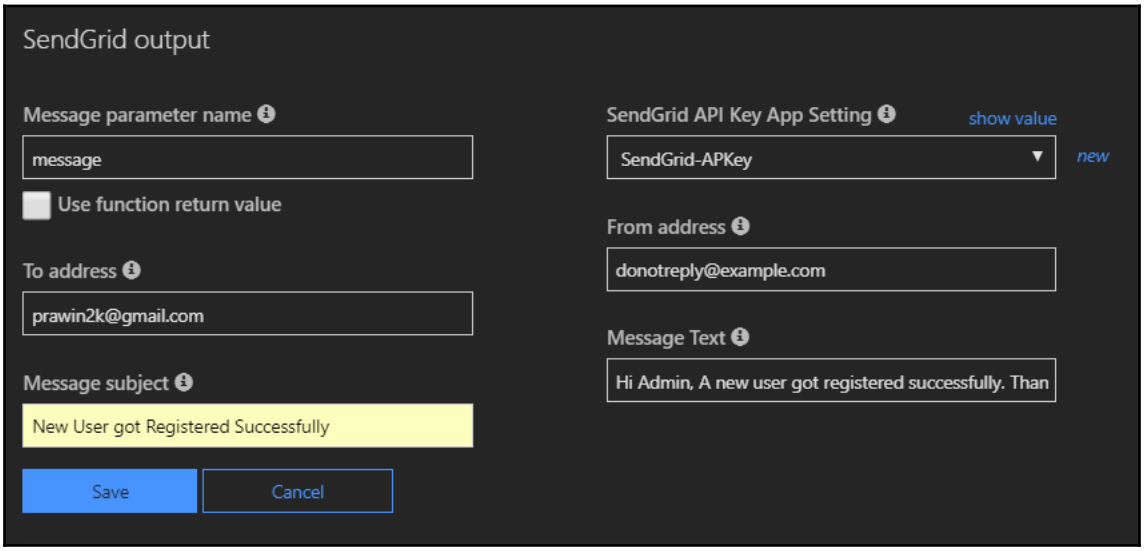


Once we ensure that the Queue Trigger is working as expected, let's create the **SendGrid** bindings to send the email.

Create SendGrid output binding to the Queue Trigger

1. Navigate to the **Integrate** tab of the `SendNotifications` function and click on the **New Output** button to add a new output binding.
2. Choose the **SendGrid** binding and click on the **Select** button to add the binding.
3. The next step is to install the **SendGrid** extensions. Click on the **Install** button to install the extensions. It might take a few minutes to install the extensions.
4. Provide the following parameters in the **SendGrid output (message)** binding:
    - **Message parameter name**: Leave the default value, which is `message`. We will be using this parameter in the `Run` method in a moment.
    - **SendGrid API Key**: Choose the **App settings** key that you created in **Application settings** for storing the Send Grid API Key.
    - **To address**: Provide the email address of the administrator.
    - **From address**: Provide the email address from where you would like to send the email. It might be something like `donotreply@example.com`.
    - **Message subject**: Provide the subject that you would like to have displayed in the email subject.
    - **Message Text**: Provide the email body text that you would like to have in the email body.

5. This is how the **SendGrid output (message)** binding should look like after providing all the fields:



6. Once you review the values, click on **Save** to save the changes.
7. Navigate to the `Run` method of the `SendNotifications` functions and make the following changes:
    - Add a new reference for SendGrid, along with the `SendGrid.Helpers.Mail` namespace.
    - Add a new out parameter message of the `SendGridMessage` type.
    - Create an object of the `SendGridMessage` type. We will look at how to use this object in the next recipe.
8. The following is the complete code of the `Run` method:

```
#r "SendGrid"
using System;
using SendGrid.Helpers.Mail;

public static void Run(string myQueueItem,out SendGridMessage
message, ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed:
{myQueueItem}");
    message = new SendGridMessage();
}
```

9. Now, let's test the functionality of sending the email by navigating to the `RegisterUser` function and submitting a request with the some test values, as follows:

```
{
    "firstname": "Bill",
    "lastname": "Gates",
"ProfilePicUrl":"https://upload.wikimedia.org/
 wikipedia/commons/thumb/1/19/
 Bill_Gates_June_2015.jpg/220px-
 Bill_Gates_June_2015.jpg"
}
```

How it works...

The aim of this recipe is to send a notification via email to the administrator, updating them that a new registration was created successfully.

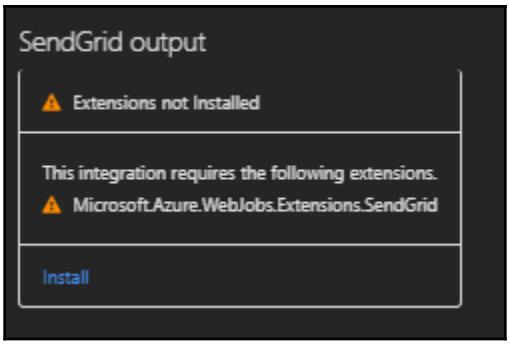
We have used one of the Azure Function output bindings, named `SendGrid`, as a **Simple Mail Transfer Protocol** (**SMTP**) server for sending our emails, by hardcoding the following properties in the **SendGrid output (message)** bindings:

- The "from" email address
- The "to" email address
- The subject of the email
- The body of the email

The **SendGrid output (message)** bindings will use the API key provided in **App settings** to invoke the required APIs of the `SendGrid` library in order to send the emails.

There's more

While, adding the SendGrid bindings, you will be prompted to install the Extensions as shown below.



In case, if you don't see them, please delete the output binding and re-create the same. You could also manually install the extensions by going through the instructions mentioned in the `https://docs.microsoft.com/en-us/azure/azure-functions/install-update-binding-extensions-manual` article

Sending an email notification dynamically to the end user

In the previous recipe, we hardcoded most of the attributes related to sending an email to an administrator, as there was just one administrator. In this recipe, we will modify the previous recipe to send a `Thank you for registration` email to the users themselves.

Getting ready

Make sure that the following are configured properly:

- The SendGrid account has been created and an API key is generated in the SendGrid portal.
- An **App settings** configuration is created in the **Application settings** of the function app.
- The **App settings** key is configured in the **SendGrid output (message)** bindings.

How to do it...

In this recipe, we will update the code in the `run.csx` file of the following Azure Functions

- `RegisterUser`
- `SendNotifications`

Accept the new email Parameter in the RegisterUser function

Perform the following steps:

1. Navigate to the `RegisterUser` function, in the `run.csx` file, add a new string variable that accepts a new input parameter, named `email`, from the `request` object, as follows. Also, note that we are serializing the `UserProfile` object and storing the JSON content to the Queue message:

```
string firstname=null,lastname = null, email = null;
...
...
string email = inputJson.email;
...
...
UserProfile objUserProfile = new UserProfile(firstname,
lastname,email);
...
...
await
NotificationQueueItem.AddAsync(JsonConvert.SerializeObject(objUserProfile))
;
```

2. Update the following highlighted code to the `UserProfile` class and click on the **Save** button to save the changes:

```
public class UserProfile : TableEntity
{
    public UserProfile(string firstname,string lastname, string
     profilePicUrl,string email)
    {
       ....
       ....
       this.ProfilePicUrl = profilePicUrl;
       this.Email = email;
```

```
    }
      ....
      ....
    public string ProfilePicUrl {get; set;}
    public string Email  { get; set; }
}
```

Retrieve the UserProfile information in the SendNotifications trigger

Perform the following steps

1. Navigate to the `SendNotifications` function, in the `run.csx` file, add `NewtonSoft.Json` reference and also the namespace.
2. The Queue Trigger will receive the input in the form of JSON string. We will use `JsonConvert.Deserializeobject` method to convert the string into a dynamic object so that we can retrieve the individual properties. Replace the existing code with the following code where we are dynamically populating the properties of the `SendGridMessage` from the code.

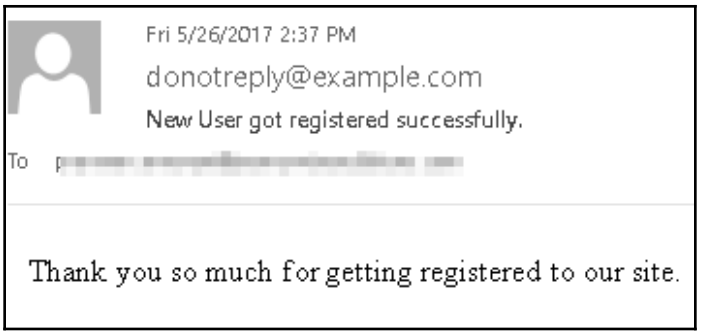
```
#r "SendGrid"
#r "Newtonsoft.Json"
using System;
using SendGrid.Helpers.Mail;
using Newtonsoft.Json;

public static void Run(string myQueueItem,
                       out SendGridMessage message,
                       ILogger log)
{
 log.LogInformation($"C# Queue trigger function processed:

 {myQueueItem}");
 dynamic inputJson = JsonConvert.DeserializeObject(myQueueItem);
 string FirstName=null, LastName=null, Email = null;
 FirstName=inputJson.FirstName;
 LastName=inputJson.LastName;
 Email=inputJson.Email;
 log.LogInformation($"Email{inputJson.Email}, {inputJson.FirstName
+ " " + inputJson.LastName}");
 message = new SendGridMessage();
 message.SetSubject("New User got registered successfully.");
 message.SetFrom("donotreply@example.com");
 message.AddTo(Email,FirstName + " " + LastName);
```

```
 message.AddContent("text/html", "Thank you " + FirstName + " " +
LastName +" so much for getting registered to our site.");
}
```

3. Let's run a test by adding a new input field email to the test request payload, shown as follows:

```
{
    "firstname": "Praveen",
    "lastname": "Sreeram",
    "email":"example@gmail.com",
    "ProfilePicUrl":"A Valid url here"
}
```

4. This is the screenshot of the email that I have received:

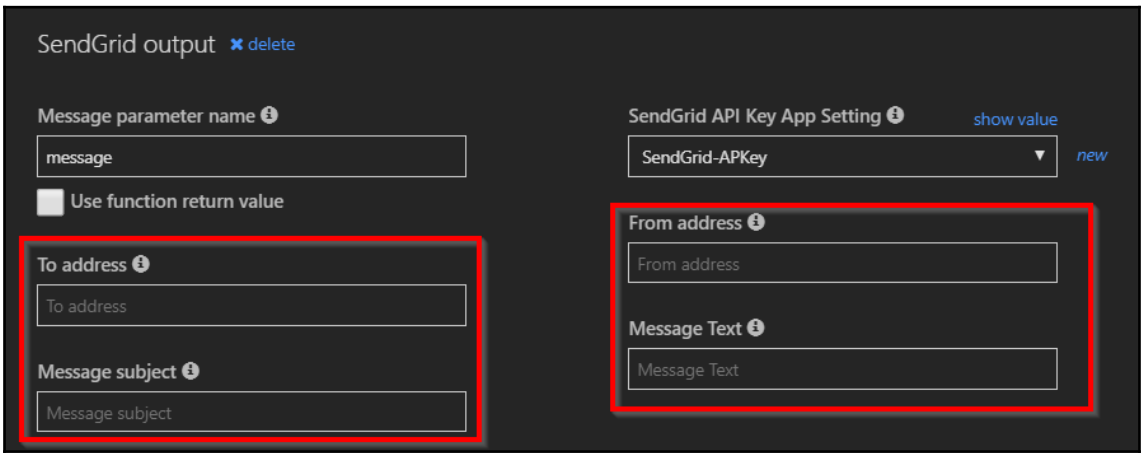


How it works...

We have updated the code of the `RegisterUser` function to accept another new parameter, named `email`.

The function accepts the `email` parameter and sends the email to the end user using the SendGrid API. We have also configured all the other parameters, such as the `From` address, `Subject`, and body (content) in the code, so that it is customized dynamically based on the requirements.

We can also clear the fields in the **SendGrid output** bindings, as shown in the following screenshot:



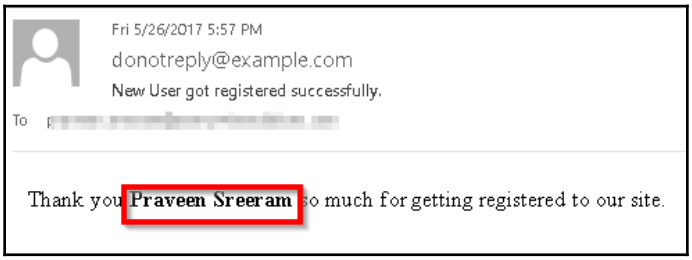
The values specified in the code will take precedence over the values specified in the preceding step.

There's more...

You can also send HTML content in the body to make your email look more attractive. The following is a simple example, where I have just applied a bold (<b>) tag to the name of the end user:

```
message.AddContent("text/html", "Thank you <b>" + FirstName + "</b><b> " +
LastName +" </b>so much for getting registered to our site.");
```

The following is the screenshot of the email, with my name in bold:



Implementing email logging in Azure Blob Storage

Most of the business applications of automated emails are likely to involve sending emails containing notifications, alerts, and so on, to the end user. At times, users might complain that they haven't received any email, even though we don't see any error in the application while sending such notification alerts.

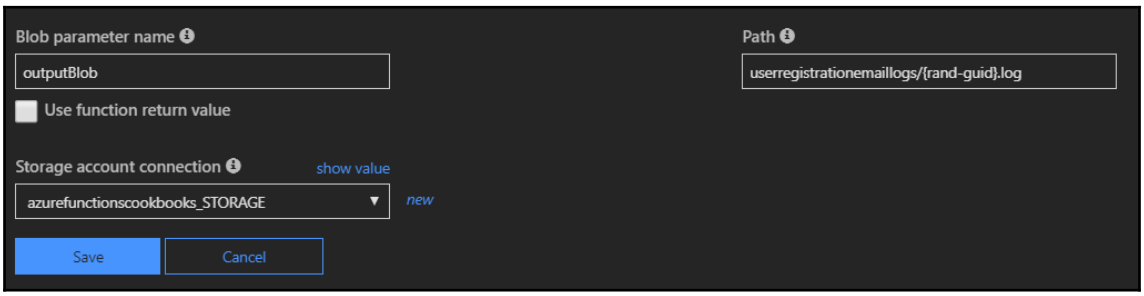
There might be multiple reasons why users might not have received the email. Each of the email service providers has different spam filters that might block the emails from the end user's inbox. But these emails might have some important information that the users might need. It makes sense to store the email content of all the emails that are sent to the end users, so that we can retrieve the data at a later stage, for troubleshooting any unforeseen issues.

In this recipe, you will learn how to create a new email log file with the `.log` extension for each new registration. This log file can be used as a redundancy for the data stored in the Table storage. You will also learn how to store the email log files as a Blob in a storage container, alongside the data entered by the end user during registration.

How to do it...

Perform the following steps:

1. Navigate to the **Integrate** tab of the `SendNotifications` function, click on **New Output**, and choose **Azure Blob Storage**. It would prompt you to install the Storage Extensions, please install the extensions to continue forward.
2. Provide the required parameters in the **Azure Blob Storage output** section, as shown in the following screenshot. Note the `.log` extension in the **Path** field:

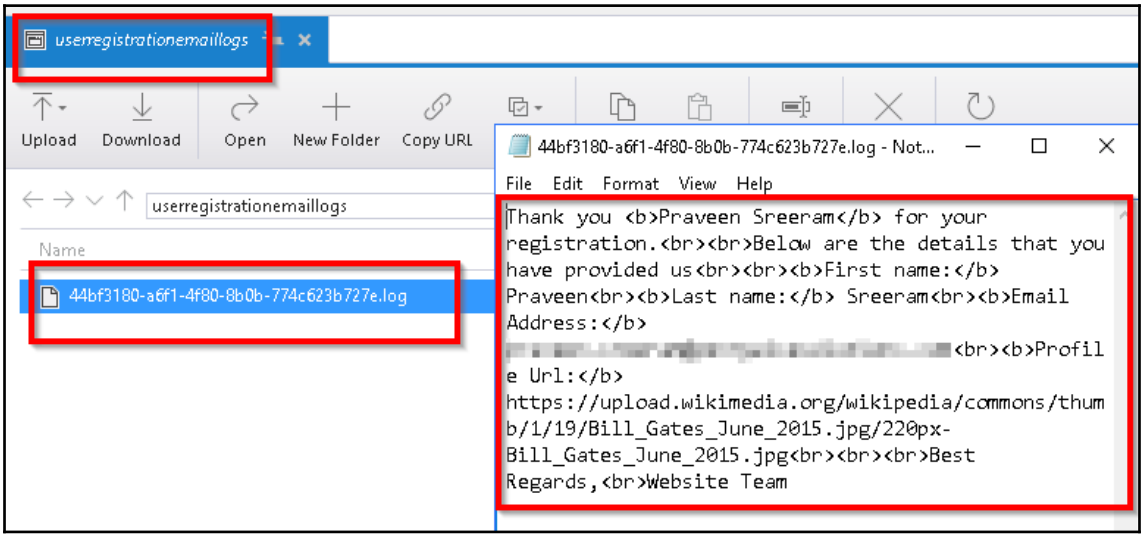


3. Navigate to the code editor of the `run.csx` file of the `SendNotifications` function and make the following changes:
    1. Add a new parameter `outputBlob` of the `TextWriter` type to the `Run` method.
    2. Add a new string variable named `emailContent`. This variable is used to frame the content of the email. We will also use the same variable to create the log file content that is finally stored in the blob.
    3. Frame the email content by appending the required static text and the input parameters received in **Request body**, as follows:

```
public static void Run(string myQueueItem,
                       out SendGridMessage message,
                       TextWriter outputBlob,
                       ILogger log)
....
....
string FirstName=null, LastName=null, Email = null;
string emailContent;
....
....
emailContent = "Thank you <b>" + FirstName + " " +
    LastName +"</b> for your registration.<br><br>" +
    "Below are the details that you have provided

 us<br> <br>"+ "<b>First name:</b> " +
    FirstName + "<br>" + "<b>Last name:</b> " +
    LastName + "<br>" + "<b>Email Address:</b> " +
    inputJson.Email + "<br><br> <br>" + "Best
Regards," + "<br>" + "Website Team";
message.AddContent(new
 Content("text/html",emailContent));
outputBlob.WriteLine(emailContent);
```

4. Run a test using the same request payload that we used in the previous recipe.

5. After running the test, the log file will be created in the container named `userregistrationemaillogs`:



How it works...

We have created new Azure Blob output bindings. As soon as a new request is received, the email content is created and written to a new `.log` file (note that you can use any other extension as well) that is stored as a Blob in the container specified in the **Path** field of the output bindings.

Modifying the email content to include an attachment

In this recipe, you will learn how to send a file as an attachment to the registered user. In our previous recipe, we created a log file of the email content. We will send the same file as an attachment to the email. However, in real-world applications, you might not intend to send log files to the end user. For the sake of simplicity, we will send the log file as an attachment.

At the time of writing, SendGrid recommends that the size of the attachment shouldn't exceed 10 MB, though technically, your email can be as large as 20 MB.

Getting ready

This is a continuation of the previous recipe. If you are reading this first, make sure to go through the previous recipes of this chapter beforehand.

How to do it...

We will need to perform the following steps before moving to the next section:

1. Make the changes to the code to create a log file with the RowKey of the table. We will achieve this using the `IBinder` interface.
2. Send this file as an attachment to the email.

Customizing the log file name using IBinder interface

Perform the following steps:

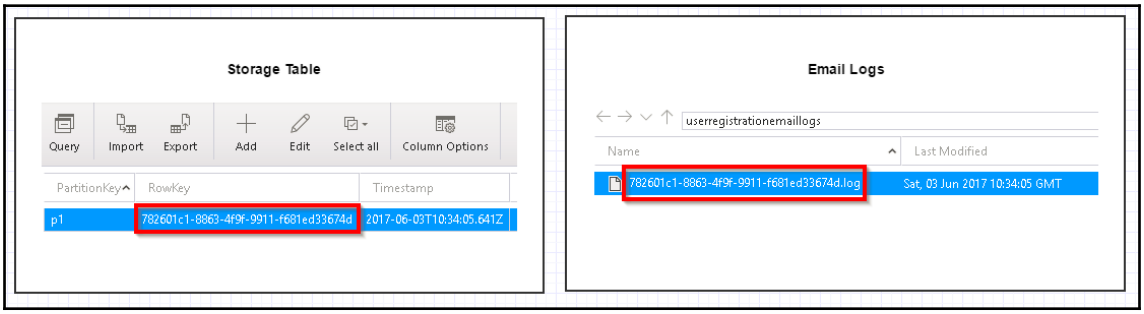
1. Navigate to the `run.csx` file of the `SendNotifications` function.
2. Remove the `TextWriter` object and replace it with the variable binder of the `IBinder` type. The following is the new signature of the `Run` method with the changes highlighted:

```
#r "SendGrid"
#r "Newtonsoft.Json"
#r "Microsoft.Azure.WebJobs.Extensions.Storage"

using System;
using SendGrid.Helpers.Mail;
using Newtonsoft.Json;
using Microsoft.Azure.WebJobs.Extensions.Storage;

public static void Run(string myQueueItem,
                       out SendGridMessage message,
                       IBinder binder,
                       ILogger log)
```

3. We have removed the `TextWriter` object, the `outputBlob.WriteLine(emailContent);` function will no longer work. Let's replace it with the following piece of code:

```
using (var emailLogBloboutput = binder.Bind<TextWriter>(new
 BlobAttribute($"userregistrationemaillogs/
 {objInsertedUser.RowKey}.log")))
{
   emailLogBloboutput.WriteLine(emailContent);
}
```

4. Let's run a test using the same request payload that we used in the previous recipes.
5. You can see the email log file that is created using the RowKey of the new record stored in the Azure Table storage, as shown in the following screenshot:



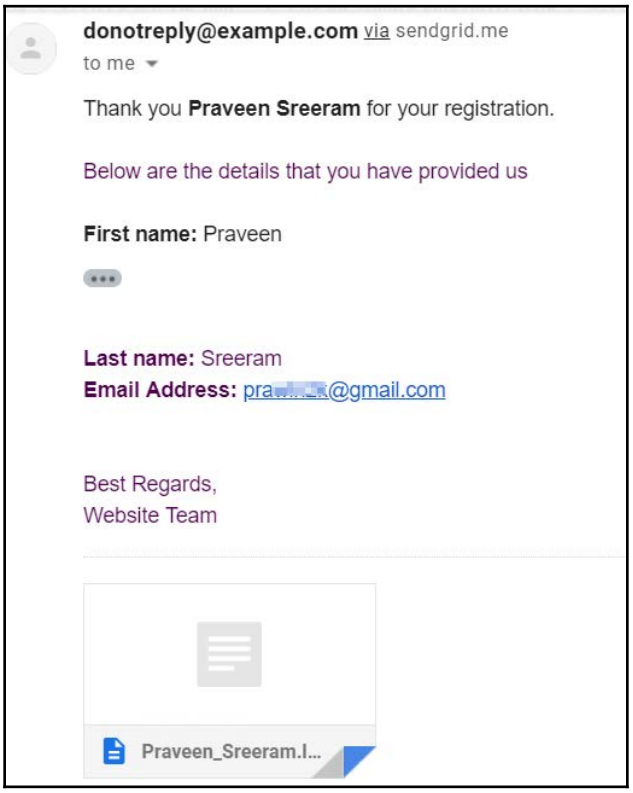
Adding an attachment to the email

Perform the following steps:

1. Add the following code to the `Run` method of the `SendNotifications` function, and save the changes by clicking on the **Save** button:

```
        message.AddAttachment(FirstName +"_"+LastName+".log",
System.Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(emailConte
nt)),
                                "text/plain",
                                "attachment",
                                 "Logs"
                                  );
```

2. Run a test using the same request payload that we have used in the previous recipes.
3. This is the screenshot of the email, along with the attachment:



You can learn more about the Send Grid API at `https://sendgrid.com/docs/API_Reference/api_v3.html`

Sending an SMS notification to the end user using the Twilio service

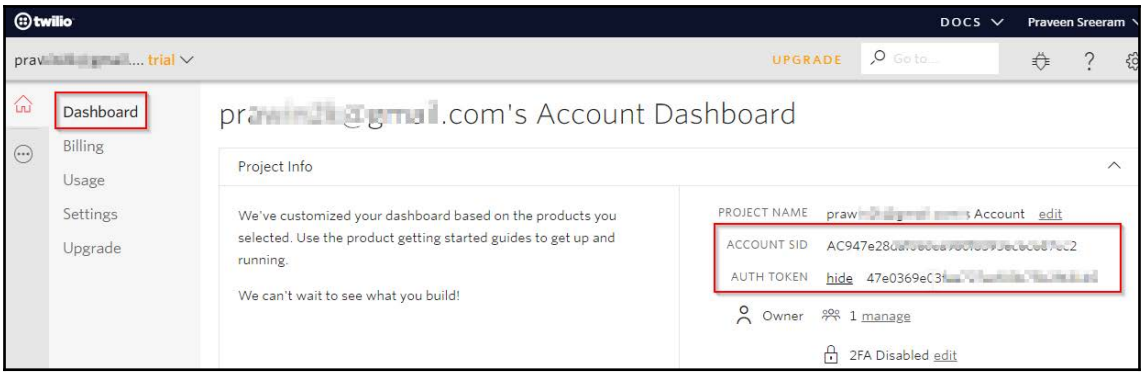
In most of the previous recipes of this chapter, we have worked with SendGrid triggers to send emails in different scenarios. In this recipe, you will learn how to send notifications via SMS, using one of the leading cloud communication platforms, named Twilio.

You can also learn more about Twilio at `https://www.twilio.com/`.

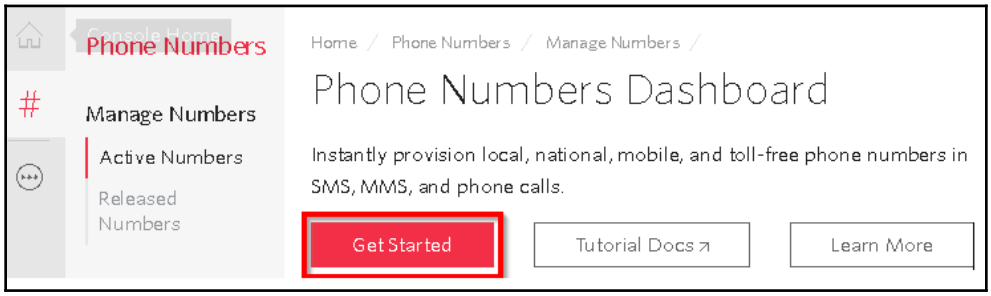
Getting ready

In order to use the **Twilio SMS output (objsmsmessage)** binding, we need to do the following:

1. Create a trial Twilio account at `https://www.twilio.com/try-twilio`.
2. After successful creation of the account, grab the **ACCOUNT SID** and **AUTH TOKEN** from the Twilio **Dashboard**, as shown in the following screenshot. We will create two **App settings** in the **Application settings** blade of the function app for both of these settings:

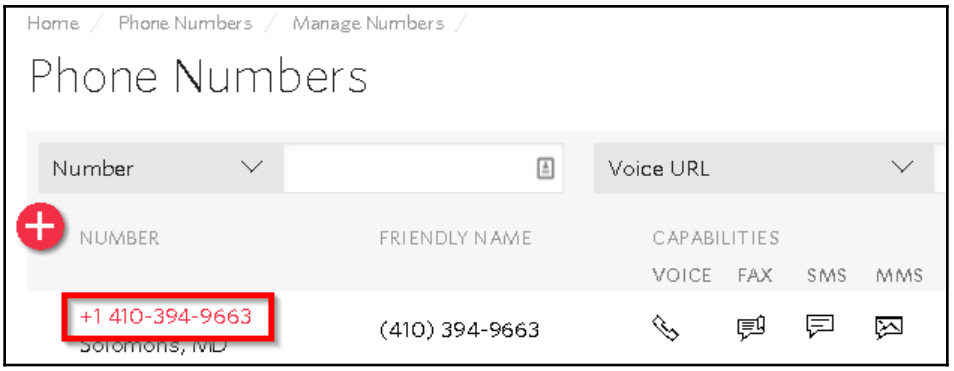


3. In order to start sending messages, you need to create an active number within Twilio, which will be used as the **From number** that you will use for sending the SMS. You can create and manage numbers in the **Phone Numbers Dashboard**. Navigate to `https://www.twilio.com/console/phone-numbers/incoming` and click on the **Get Started** button, as shown in the following screenshot:

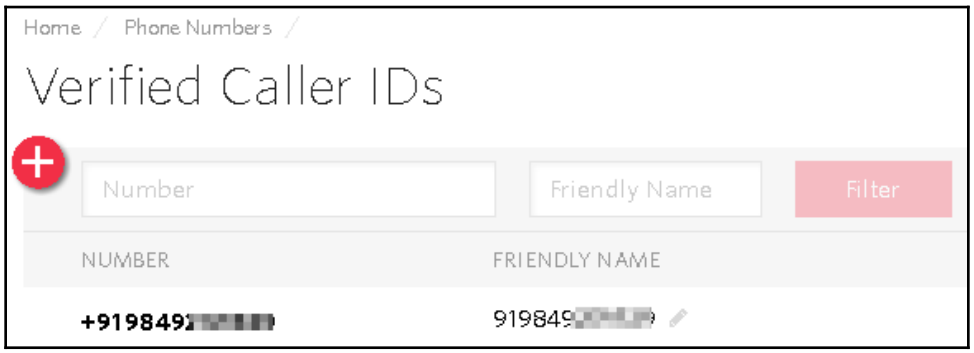


4. On the **Get Started with Phone Numbers** page, click on **Get your first Twilio phone number**, as shown in the following screenshot:

5. Once you get your number, it will be listed as follows:



6. The final step is to verify a number to which you would like to send an SMS. You can have only one number in your trial account. You can verify a number on Twilio's Verified page, available at `https://www.twilio.com/console/phone-numbers/verified`. The following is a screenshot of the list of verified numbers:



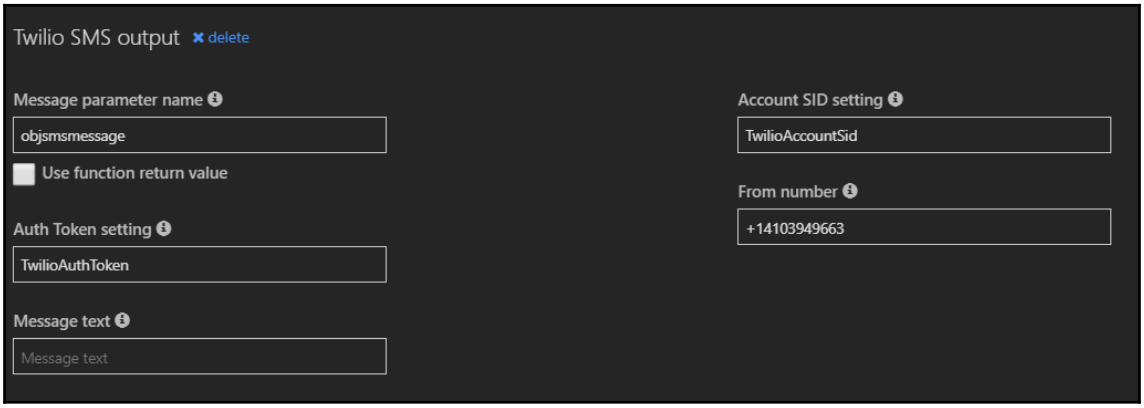
How to do it...

Perform the following steps:

1. Navigate to the **Application settings** blade of the function app and add two keys to store **TwilioAccountSID** and **TwilioAuthToken**, shown as follows:

2. Go the **Integrate** tab of the `SendNotifications` function, click on **New Output**, and choose **Twilio SMS**.

3. Click on **Select** and provide the following values to the **Twilio SMS output** bindings. Please install the extensions of Twilio. The **From number** is the one that is generated in the Twilio portal, which we discussed in the *Getting ready* section of this recipe:

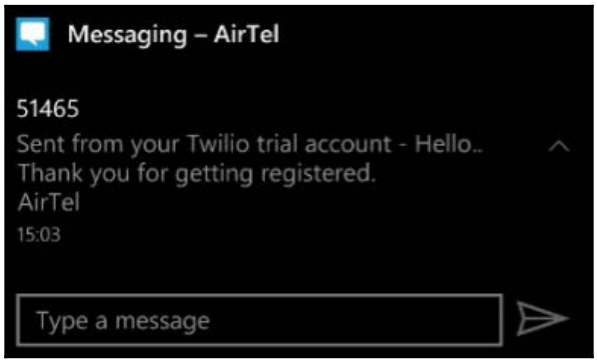


4. Navigate to the code editor and add the following lines of code, highlighted in bold. In the below code, I have hardcoded the **To number**. However, in real-world scenarios, you would dynamically receive the end user's mobile number and send the SMS via code:

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

...
...
using Microsoft.Azure.WebJobs.Extensions.Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;
public static void Run(string myQueueItem,
                out SendGridMessage message,
                IBinder binder,
                out CreateMessageOptions objsmsmessage,
                ILogger log)
 ...
 ...
 ...
 message.AddAttachment(FirstName +"_"+LastName+".log",
System.Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(emailConte
nt)),
```

```
                                "text/plain",
                                "attachment",
                                 "Logs"
                                  );

        objsmsmessage = new CreateMessageOptions(new PhoneNumber("+91
98492*****"));
        objsmsmessage.Body = "Hello.. Thank you for getting registered.";
        }
```

5. Now, do a test run of the `RegisterUser` function using the same request payload.
6. The following is the screenshot of the SMS that I have received:



How it works...

We have created a new Twilio account and copied the account ID and app key into the **App settings** of the Azure Function app. These two settings will be used by the function app runtime in order to connect to the Twilio API to send the SMS.

For the sake of simplicity, I have hardcoded the phone number in the output bindings. However, in real-world applications, you would send the SMS to the phone number provided by the end users.

You can go through the video `https://www.youtube.com/watch?v=ndxQXnoDIj8` to view a working implementation.

3
Seamless Integration of Azure Functions with Azure Services

In this chapter, we will cover the following recipes:

- Using Cognitive Services to locate faces in images
- Azure SQL Database interactions using Azure Functions
- Monitoring tweets using Logic Apps and notifying users when a popular user tweets
- Integrating Logic Apps with serverless functions
- Auditing Cosmos DB data using change feed triggers

Introduction

One of the major goals of Azure Functions is to enable developers to just focus on developing application requirements and logic and abstract everything else.

As a developer or business user, you cannot afford to invent and develop your own applications from scratch for each of your business needs. You would first need to research the existing systems and see whether they fit your business requirements. Often, it would not be easy to understand the APIs of the other systems and integrate them, as they will have been developed by someone else.

Azure provides many connectors that you can leverage to integrate your business applications with other systems pretty easily.

In this chapter, we will learn how easy it is to integrate the different services that are available in the Azure ecosystem.

Using Cognitive Services to locate faces in images

In this recipe, you will learn how to use the Computer Vision API to detect faces within an image. We will be locating faces, and capturing their coordinates, and saving them in different areas of Azure Table Storage based on gender.

Getting ready

To get started, we need to create a Computer Vision API and configure its API keys so that Azure Functions (or any other program) can access it programmatically.

Make sure that you have Azure Storage Explorer installed and have also configured to access the storage area where you are uploading the blobs.

Creating a new Computer Vision API account

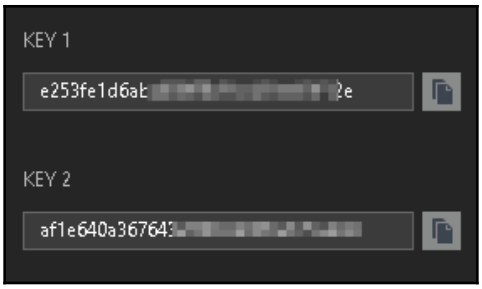
Perform the following steps:

1. Search for `computer vision` and click on **Create**.
2. The next step is to provide all the details to create a Computer Vision API account. At the time of writing this, the Computer Vision API has just two pricing tiers. I went for the free one, **F0**, which allows 20 API calls per minute and is limited to 5,000 calls each month.

Configuring application settings

Perform the following steps:

1. Once the Computer Vision API account is generated, you can navigate to the **Keys** blade and grab any of the following keys:



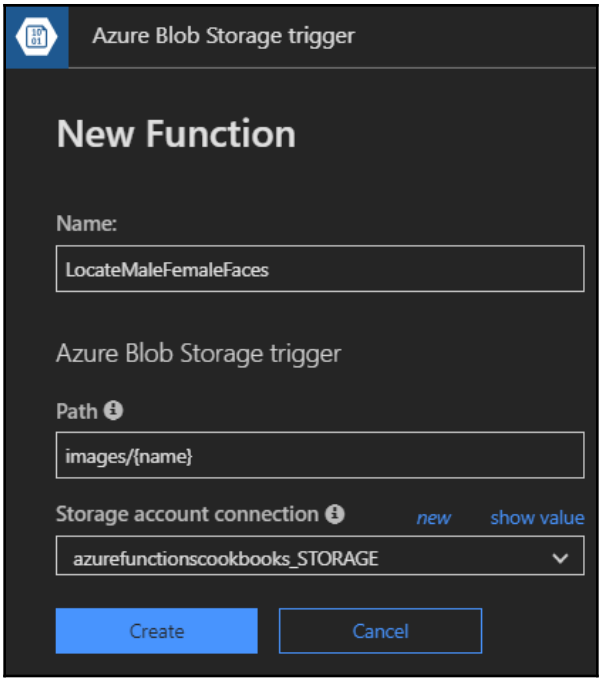
2. Navigate to your Azure functions app, configure **Application settings** with the name `Vision_API_Subscription_Key`, and use any of the preceding keys as the value. This key will be used by the Azure Functions Runtime to connect to and consume the Computer Vision Cognitive Services API.
3. Make a note of the location where you are creating the computer vision service. In my case, I have chosen **West Europe**. It is important when you are passing the images to the Cognitive Services API to ensure that the endpoint of the API starts with the location name. It would be something like this: `https://westeurope.api.cognitive.microsoft.com/vision/v1.0/analyze?visualFeatures=Faces&language=en`.

How to do it...

Perform the following steps:

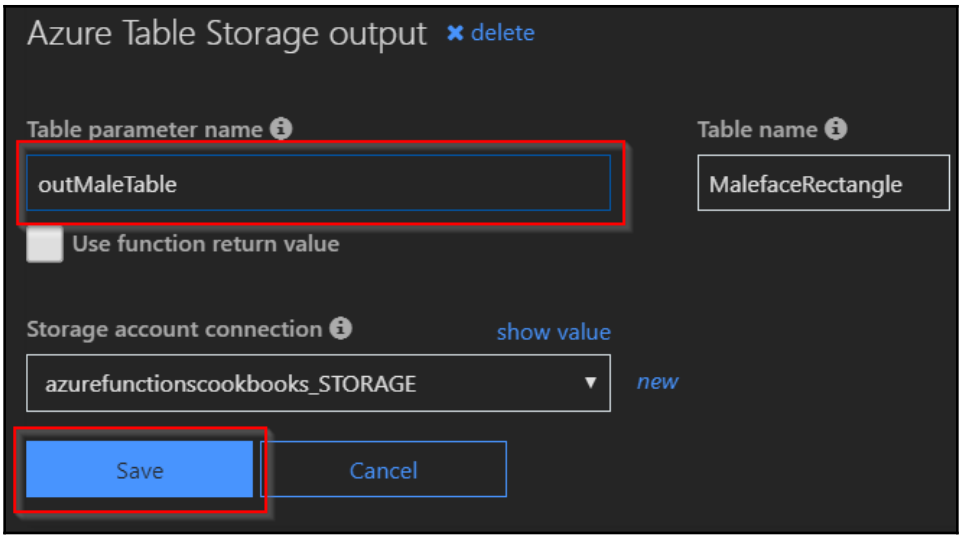
1. Create a new function using one of the default templates named **Azure Blob Storage Trigger**.

2. Next, you need to provide the name of the Azure Function along with the **Path** and **Storage account connection**. We will upload a picture to the **Azure Blob Storage trigger (image)** container (mentioned in the **Path** parameter in the following screenshot) at the end of this section:

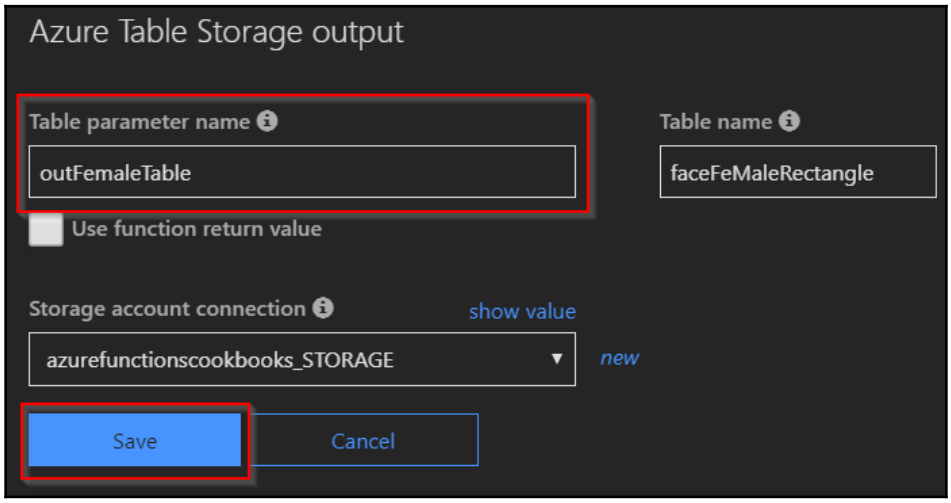


Note that while creating the function, the template creates one **Blob Storage Table output** binding and allows us to provide the name of the **Table name** parameter. However, we cannot assign the name of the parameter while creating the function. We will be able to change it after it is created. Once you have reviewed all the details, click on the **Create** button to create the Azure Function.

3. Once the function is created, navigate to the **Integrate** tab, click on **New Output** and choose **Azure Table Storage**, then click on the **Select** button. Provide the parameter values and then click on the **Save** button, as shown in the following screenshot:



4. Let's create another **Azure Table Storage output** binding to store all the information for women by clicking on the **New Output** button in the **Integrate** tab, selecting **Azure Table Storage**, and clicking on the **Select** button. This is how it looks after providing the input values:

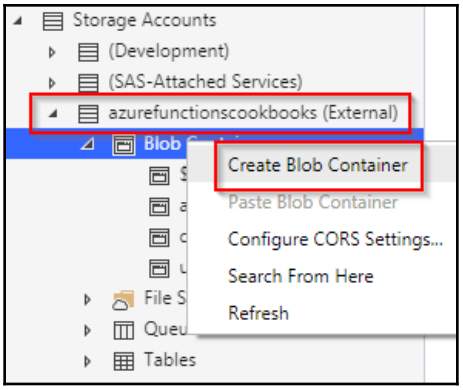


5. Once you have reviewed all the details, click on the **Save** button to create the **Azure Table Storage output** binding and store the details about women.
6. Navigate to the code editor of the `Run` method of the `LocateMaleFemaleFaces` function, then add the `outMaleTable` and `outFemaleTable` parameters. The following code grabs the image stream uploaded to the blob, which is then passed as an input to Cognitive Services, which returns a JSON with all the face information. Once the face information, including coordinates and gender details, is received, we store the face coordinates into the respective Table Storage using the table output bindings:

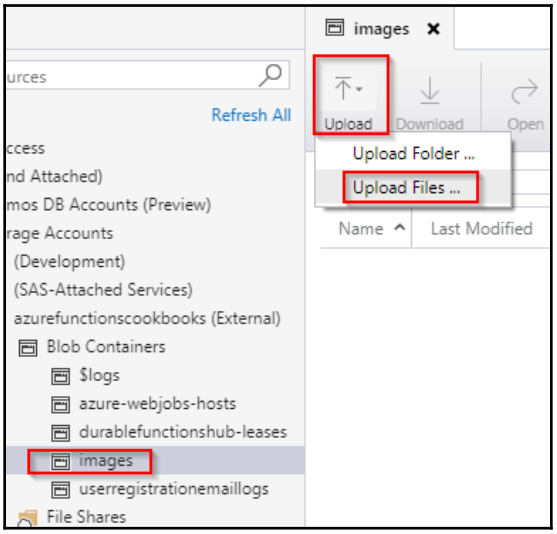
```
#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Newtonsoft.Json;
using Microsoft.WindowsAzure.Storage.Table;
using System.IO;
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
public static async Task Run(Stream myBlob,
                        string name,
                        IAsyncCollector<FaceRectangle> outMaleTable,
                        IAsyncCollector<FaceRectangle>
outFemaleTable,
                        ILogger log)
{
    log.LogInformation($"C# Blob trigger function Processed blob\n
Name:{name} \n Size: {myBlob.Length} Bytes");
    string result = await CallVisionAPI(myBlob);
    log.LogInformation(result);
    if (String.IsNullOrEmpty(result))
    {
        return;
    }
    ImageData imageData =
JsonConvert.DeserializeObject<ImageData>(result);
    foreach (Face face in imageData.Faces)
    {
        var faceRectangle = face.FaceRectangle;
        faceRectangle.RowKey = Guid.NewGuid().ToString();
        faceRectangle.PartitionKey = "Functions";
        faceRectangle.ImageFile = name + ".jpg";
        if(face.Gender=="Female")
           {
              await outFemaleTable.AddAsync(faceRectangle);
           }
```

```
            else
            {
               await outMaleTable.AddAsync(faceRectangle);
            }
    }
}
static async Task<string> CallVisionAPI(Stream image)
{
    using (var client = new HttpClient())
    {
        var content = new StreamContent(image);
        var url =
"https://westeurope.api.cognitive.microsoft.com/vision/v1.0/analyze
?visualFeatures=Faces&language=en";
        client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-
Key",
Environment.GetEnvironmentVariable("Vision_API_Subscription_Key"));
        content.Headers.ContentType = new
MediaTypeHeaderValue("application/octet-stream");
        var httpResponse = await client.PostAsync(url, content);

        if (httpResponse.StatusCode == HttpStatusCode.OK)
        {
            return await httpResponse.Content.ReadAsStringAsync();
        }
    }
    return null;
}
public class ImageData
{
    public List<Face> Faces { get; set; }
}
public class Face
{
    public int Age { get; set; }
    public string Gender { get; set; }
    public FaceRectangle FaceRectangle { get; set; }
}
public class FaceRectangle : TableEntity
{
    public string ImageFile { get; set; }
    public int Left { get; set; }
    public int Top { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
}
```

7. Let's add a condition (highlighted in **bold** in the code mentioned in *step 10*) to check the gender and, based on the gender, store the information in the respective Table Storage.
8. Create a new blob container named `images` using Azure Storage Explorer, as shown in the following screenshot:



9. Let's upload a picture with male and female faces to the container named `images` using Azure Storage Explorer, as shown here:

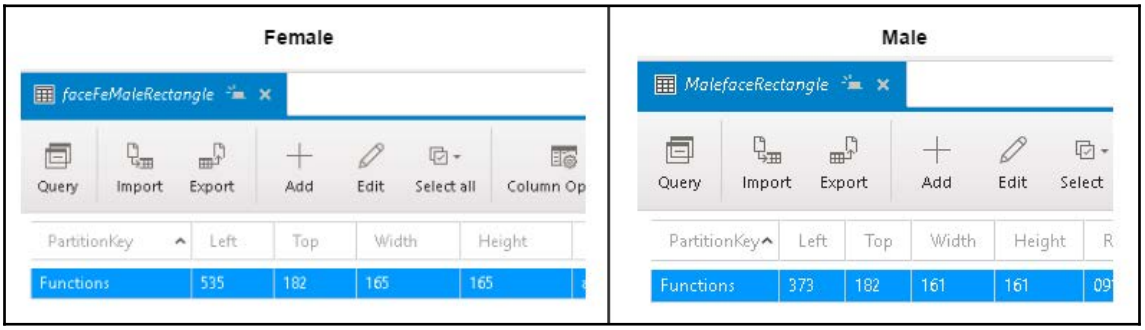


10. The function gets triggered as soon as you upload an image. This is the JSON that was logged in the **Logs** console of the function:


```
{
   "requestId":"483566bc-7d4d-45c1-87e2-6f894aaa4c29",
    "metadata":{  },
   "faces":[
   {
      "age":31,
      "gender":"Female",
      "faceRectangle":{
       "left":535,
       "top":182,
       "width":165,
       "height":165
      }
   },
   {
      "age":33,
      "gender":"Male",
      "faceRectangle":{
      "left":373,
      "top":182,
      "width":161,
      "height":161
      }
   }
 ]
}
```


If you are a frontend developer with expertise in HTML5 and canvas-related technologies, you can even draw squares that locate the faces in image using the information provided by Cognitive Services.

11. The function has also created two different Azure Table Storage tables, as shown here:



How it works...

We first created a Table Storage output binding for storing details about all the men in the photos. Then, we created another Table Storage output binding to store the details about all the women.

While we do use all the default code that the Azure Functions template provides to store all the face coordinates in a single table, we just made a small change to check whether the person in the photo is male or female and store the data based on the result.

Note that the APIs aren't 100% accurate in identifying the correct gender. So, in your production environments, you should have a fallback mechanism to handle such situations.

There's more...

The default code that the template provides invokes the Computer Vision API by passing the image that we have uploaded to the blob storage. The face locator templates invoke the API call by passing the `visualFeatures=Faces` parameter, which returns information about the following:

- Age
- Gender
- Coordinates of the faces in the picture

You can learn more about the Computer Vision API at `https://docs.microsoft.com/en-in/azure/cognitive-services/computer-vision/home`.

Use the `Environment.GetEnvironmentVariable("KeyName")` function to retrieve the information stored in application settings. In this case, the `CallVisionAPI` method uses the function to retrieve the key, which is essential for making a request to Microsoft Cognitive Services.

It's a best practice to store all keys and other sensitive information in application settings.

`ICollector` and `IAsyncCollector` are used for bulk insertion of data.

Azure SQL Database interactions using Azure Functions

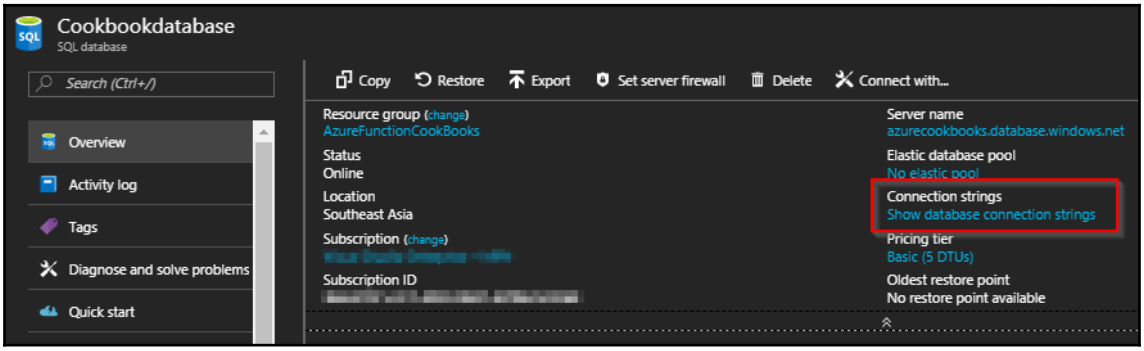
So far, you have learned how to store data in Azure Storage services, such as blobs, queues, and tables. All these storage services are great for storing non-structured or semi-structured data. However, we might need to store data in relational database management systems, such as an Azure SQL Database.

In this recipe, you will learn how to utilize the ADO.NET API to connect to a SQL Database and insert JSON data into a table named `EmployeeInfo`.

Getting ready

Navigate to the Azure portal and do the following:

1. Create a logical SQL Server with the name of your choice. It is recommended that you create it in the same resource group where you have your Azure Functions.
2. Create an Azure SQL Database named `Cookbookdatabase` by choosing **Blank database** in the **Select source** drop-down of the **SQL Database** blade while creating the database.
3. Create a firewall rule for your IP address by clicking on the **Set Firewall rule** button in the **Overview** blade so that you can connect to the Azure SQL Databases using **SQL Server Management Studio** (**SSMS**). If you don't have SSMS, install the latest version of SSMS. You can download it from `https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms`.
4. Click on the **Show database connection strings** link in the **Essentials** blade of SQL Database, as shown in the following screenshot:



5. Copy the connection string from the following blade. Make sure that you replace the `your_username` and `your_password` templates with your actual username and password:

6. Open your SSMS and connect to the Azure logical SQL Server that you created in the previous steps.
7. Once you're connected, create a new table named `EmployeeInfo` using the following schema:

```
CREATE TABLE [dbo].[EmployeeInfo](
[PKEmployeeId] [bigint] IDENTITY(1,1) NOT NULL,
[firstname] [varchar](50) NOT NULL,
[lastname] [varchar](50) NULL,
[email] [varchar](50) NOT NULL,
[devicelist] [varchar](max) NULL,
CONSTRAINT [PK_EmployeeInfo] PRIMARY KEY CLUSTERED
(
[PKEmployeeId] ASC
)
)
```

How to do it...

Perform the following steps:

1. Navigate to your function app, create a new HTTP trigger using the **HttpTrigger-CSharp** template, choose **Authorization level** as **Anonymous**.
2. Navigate to the code editor of `run.csx` in the `SaveJSONToAzureSQLDatabase` function and replace the default code with the following. The following code grabs the data that is passed to the HTTP trigger and inserts it into the database using the ADO.Net API:

```
#r "Newtonsoft.Json"
#r "System.Data"

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


public static async Task<IActionResult> Run(HttpRequest req,
ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a
request.");
    string firstname,lastname, email, devicelist;

    string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    firstname = data.firstname;
    lastname = data.lastname;
    email = data.email;
    devicelist = data.devicelist;

    SqlConnection con =null;
           try
           {
              string query = "INSERT INTO EmployeeInfo
(firstname,lastname, email, devicelist) " + "VALUES
(@firstname,@lastname, @email, @devicelist) ";
              con = new
SqlConnection("Server=tcp:azurecookbooks.database.windows.net,1433;
Initial Catalog=Cookbookdatabase;Persist Security Info=False;User
ID=username;Password=password;MultipleActiveResultSets=False;Encryp
```

```
t=True;TrustServerCertificate=False;Connection Timeout=30;");
                SqlCommand cmd = new SqlCommand(query, con);
                cmd.Parameters.Add("@firstname", SqlDbType.VarChar,
                 50).Value = firstname;
                cmd.Parameters.Add("@lastname", SqlDbType.VarChar,
50)
                 .Value = lastname;
                cmd.Parameters.Add("@email", SqlDbType.VarChar, 50)
                 .Value = email;
                cmd.Parameters.Add("@devicelist", SqlDbType.VarChar)
                 .Value = devicelist;
                con.Open();
                cmd.ExecuteNonQuery();
            }
            catch(Exception ex)
            {
                log.LogInformation(ex.Message);
            }
            finally
            {
                if(con!=null)
                {
                   con.Close();
                }
            }

    return (ActionResult)new OkObjectResult($"Successfully inserted
the data.");
}
```

Note that you need to validate each and every input parameter. For the sake of simplicity, the code that validates the input parameters is not included. Make sure that you validate each and every parameter before you save it into your database. It also a good practice to store the connection string in **Application settings**.

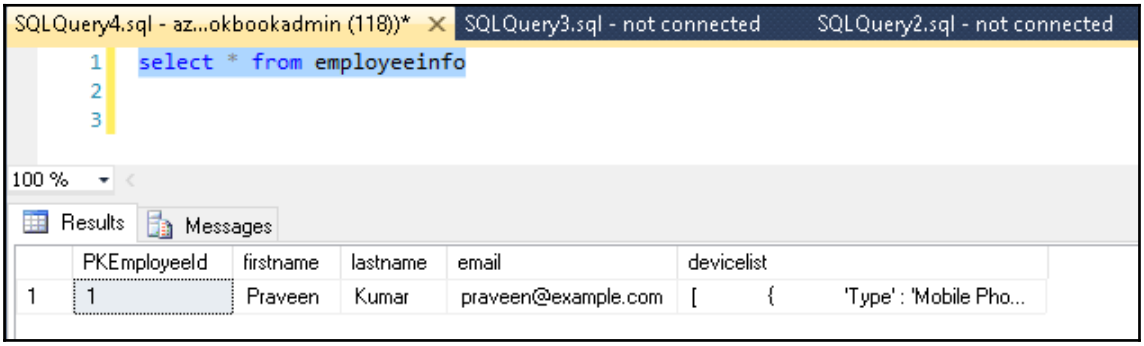
3. Let's run the HTTP trigger using the following test data right from the **Test** console of Azure Functions:

```
{
            "firstname": "Praveen",
            "lastname": "Kumar",
            "email": "praveen@example.com",
            "devicelist":
                "[
                    {
                      'Type' : 'Mobile Phone',
```

```
                    'Company':'Microsoft'
                },
                {
                    'Type' : 'Laptop',
                    'Company':'Lenovo'
                }
            ]"
}
```

Note that you need to validate each and every input parameter. For the sake of simplicity, the code that validates the input parameters is not included. Make sure that you validate each and every parameter before you save it into your database.

A record was inserted successfully, as shown in the following screenshot:



How it works...

The goal of this recipe was to accept input values from the user and save them to a relational database where the data could be retrieved later for operational purposes. For this, we used Azure SQL Database, a relational database offering also known as **database as a service** (**DBaaS**). We have created a new SQL Database and created firewall rules that allow us to connect remotely from the local development workstation using SSMS. We have also created a table named `EmployeeInfo`, which can be used to save data.

We have developed a simple program using the ADO.NET API that connects to the Azure SQL Database and inserts data into the `EmployeeInfo` table.

Monitoring tweets using Logic Apps and notifying users when a popular user tweets

One of my colleagues, who works for a social grievance management project, is responsible for monitoring the problems that users post on social media platforms, such as Facebook, Twitter, and so on. He was facing the problem of continuously monitoring the tweets posted on his customer's Twitter handle with specific hashtags. His main job was to respond quickly to the tweets by users with a huge follower count, say, users with more than 50,000 followers. So, he was looking for a solution that kept monitoring a particular hashtag and alerted him whenever an user with more than 50,000 followers tweets so that he can quickly have his team respond to that user.

Note that for the sake of simplicity, we will have the condition to check for 200 followers instead of 50,000 followers.

Before I knew about Azure Logic Apps, I thought it would take a few weeks to learn about, develop, test, and deploy such a solution. Obviously, it would take a good amount of time to learn, understand, and consume the Twitter (or any other social channel) API to get the required information and build an end-to-end solution that solves the problem.

Fortunately, after learning about Logic Apps and its out-of-the-box connectors, it hardly takes 10 minutes to design a solution for the problem that my friend had.

In this recipe, you will learn how to design a Logic App that integrates with Twitter (for monitoring tweets) and Gmail (for sending emails).

Getting ready

We need to have the following to work with this recipe:

- A valid Twitter account
- A valid Gmail account

When working with the recipe, we will need to authorize Azure Logic Apps to access your accounts.

How to do it...

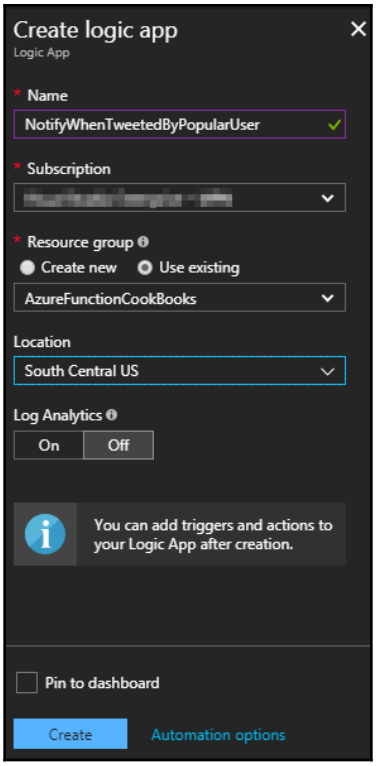
We will go through the following steps:

1. Create a new Logic App
2. Design the Logic app with Twitter and Gmail connectors
3. Test the Logic App by tweeting the tweets with the specific hashtag

Creating a new Logic App

Perform the following steps:

1. Log in to the Azure Management portal, search for `logic apps`, and select **Logic App**.
2. In the **Create logic app** blade, once you have provided the **Name**, **Resource group**, **Subscription**, and **Location**, click on the **Create** button to create the Logic App:

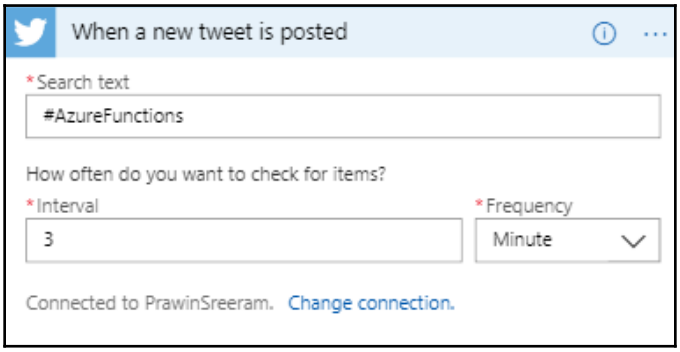


Designing the Logic App with Twitter and Gmail connectors

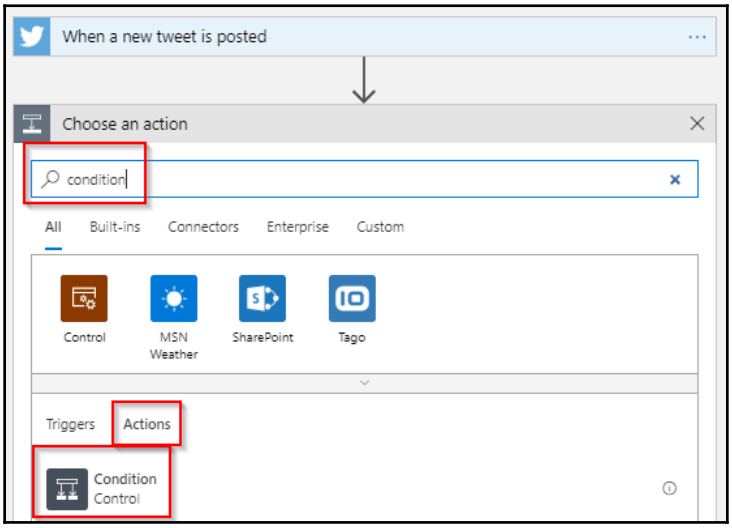
Perform the following steps:

1. After the Logic App is created, navigate to the **Logic app designer** and choose **Blank logic app**.
2. Next, you will be prompted to choose **Connectors.** In the **Connectors** list, click on **Twitter**. Then, you will be prompted to connect to Twitter by providing your Twitter account credentials. If you have already connected, it will directly show you the list of **Triggers** associated with the Twitter connector, as shown in the following screenshot:

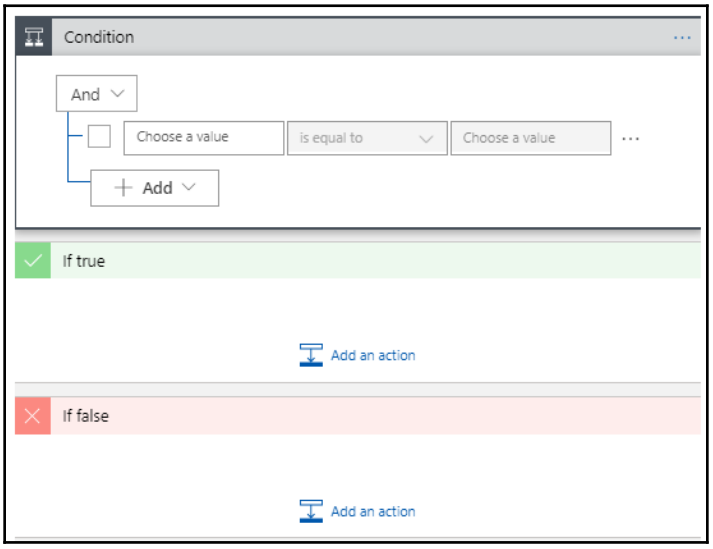
3. Once you have clicked on the **Twitter** trigger, you will be prompted to provide **Search text** (for example, hashtags and keywords) and the **Frequency** at which you would like the Logic App to poll the tweets. This is how it looks after you provide the details:



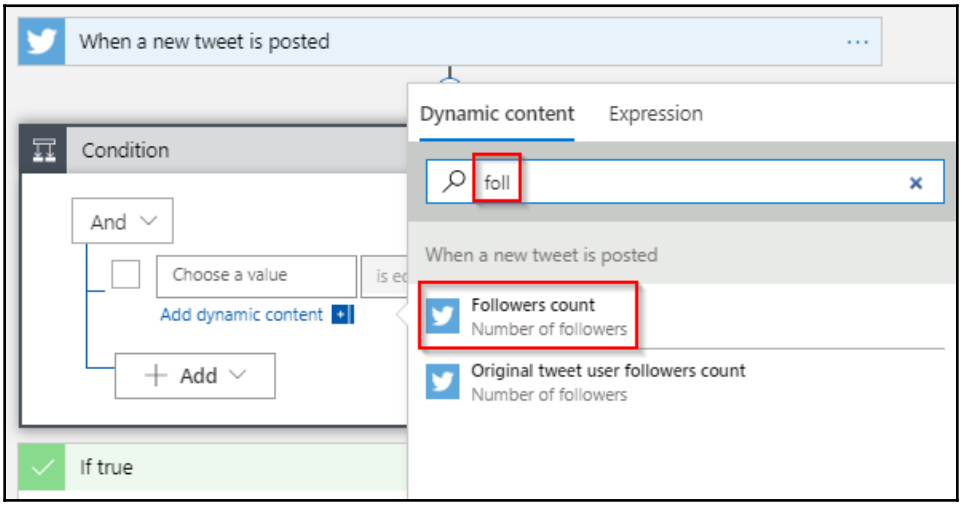
4. Let's add a new condition by clicking on **Next Step**, searching for `condition`, and selecting **Condition** action, as shown in the following screenshot:



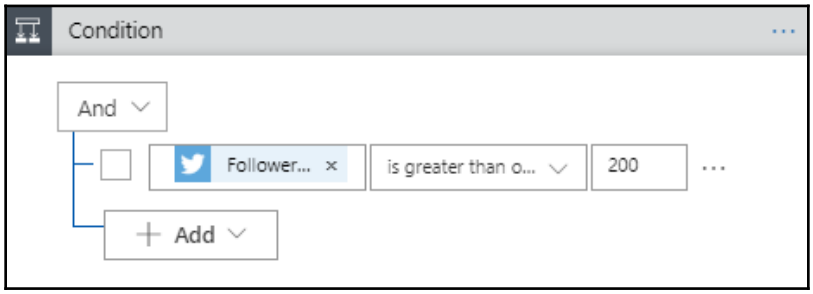
5. From the previous instruction, the following screen will be displayed, where you can choose the values for the condition and choose what you would like to add when the condition evaluates to true or false:



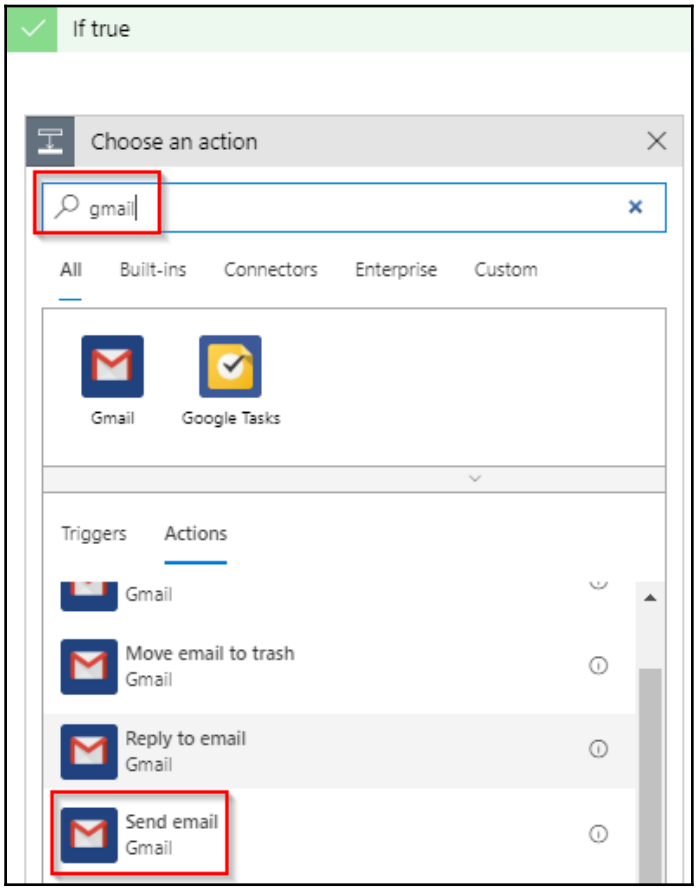
6. When you click on the **Choose a value** input field, you will get all the parameters on which you could add a condition; in this case, we need to choose **Followers count**, as shown in the following screenshot:



7. Once you choose the **Followers Count** parameter, you create a condition (**Followers count is greater than or equal to 200**), as shown in the following screenshot:

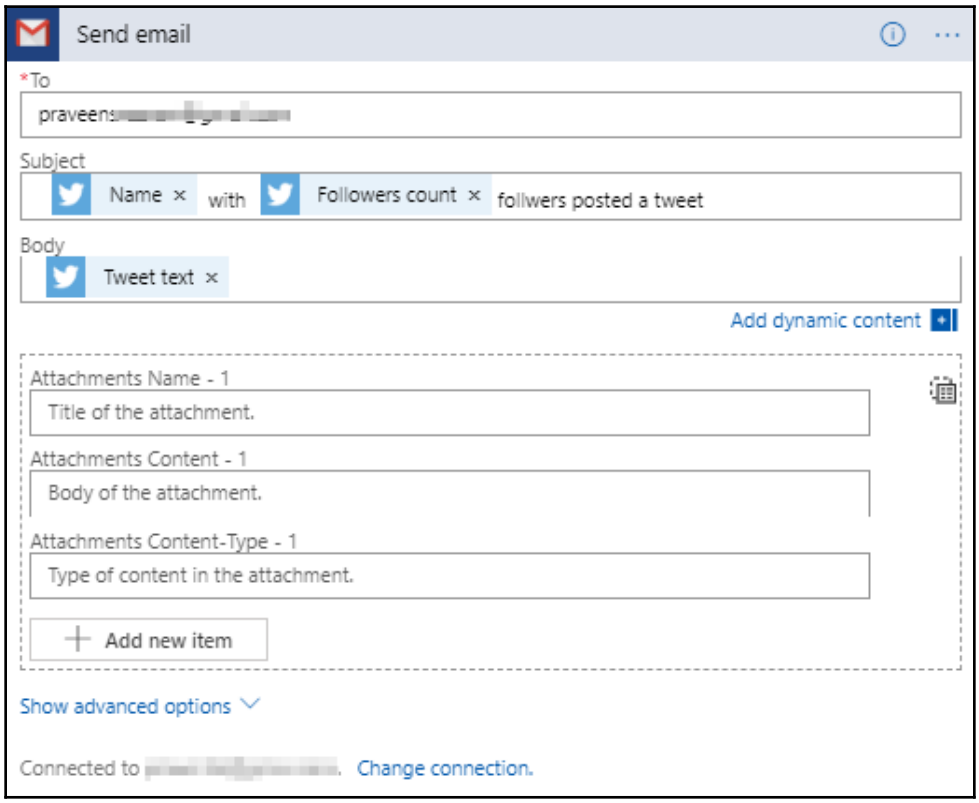


8. In the **If true** section of the preceding **Condition**, search for Gmail connection and select **Gmail** | **Send email**, as shown in the following screenshot:



9. It will ask you to log in if you haven't already. Provide your credentials and authorize Azure Logic Apps to access your Gmail account.

10. Once you have authorized, you can frame your email with **Add dynamic content** with the Twitter parameters, as shown in the following screenshot. If you don't see **Followers count**, click on the **Show more** link:

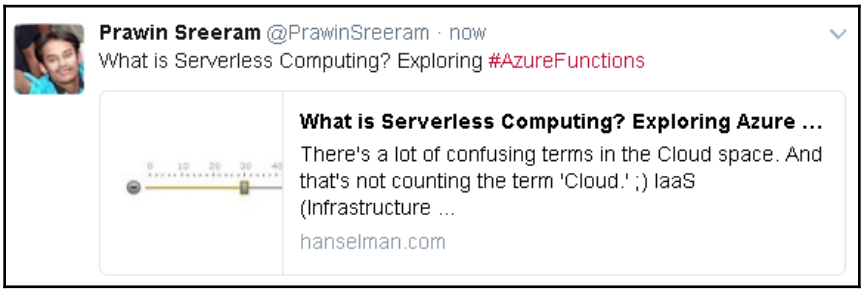


11. Once you are done, click on the **Save** button.

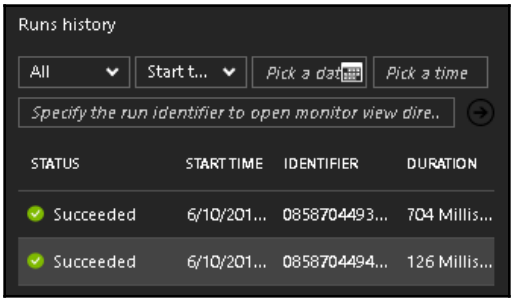
Testing the Logic App functionality

Perform the following steps:

1. Let's post a tweet on Twitter with the hashtag `#AzureFunctions`, as shown in the following screenshot:



2. After a minute or so, the Logic App should have been triggered. Let's navigate to the **Overview** blade of the Logic App and view **Runs history**:



3. Yay! It has triggered twice and I have received the emails. One of them is shown in the following screenshot:

How it works...

You have created a new Logic App and have chosen the Twitter connector to monitor the tweets posted with the hashtag `#AzureFunctions` once a minute. If there are any tweets with that hashtag, it checks whether the follower count is greater than or equal to 200. If the follower count meets the condition, then a new action is created with a new Gmail connector that is capable of sending an email with the dynamic content being framed using the Twitter connector parameters.

Integrating Logic Apps with serverless functions

In the previous recipe, you learned how to integrate different connectors using Logic Apps. In this recipe, we will implement the same solution that we implemented in the previous recipe by just moving the conditional logic that checks the followers count to Azure Functions.

Getting ready

Before moving further, we will perform the following steps:

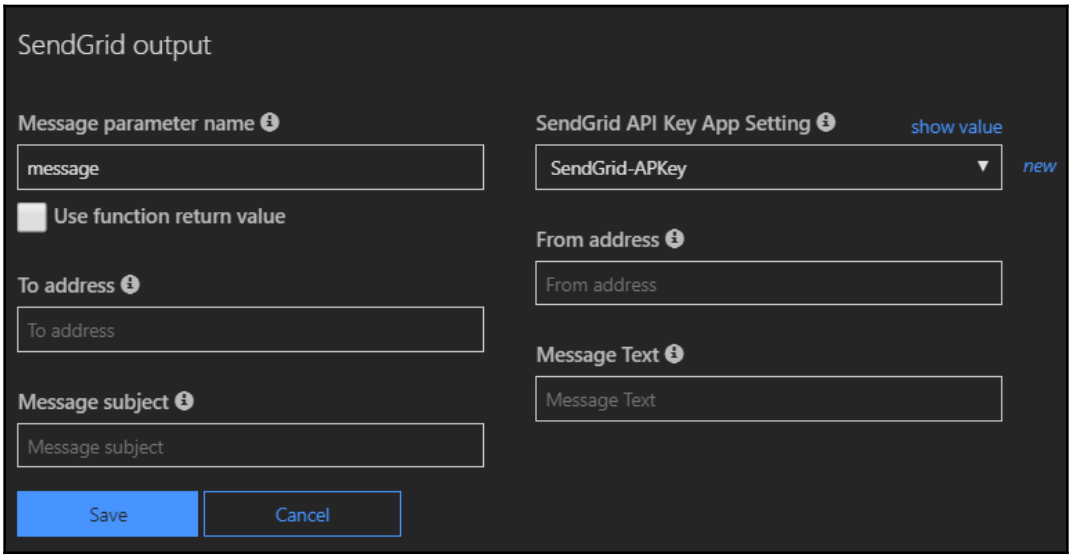
1. Create a SendGrid account (if not created already), grab the SendGrid API key, and create a new key in the **Application settings** of the function app.
2. Install Postman to test the HTTP trigger**.** You can download the tool from `https://www.getpostman.com/`.

How to do it...

Perform the following steps:

1. Create a new function, by choosing the HTTP trigger, and name it `ValidateTwitterFollowerCount`.

2. Navigate to the **Integrate** tab and add a new output binding, **SendGrid**, by clicking on the **New Output** button:

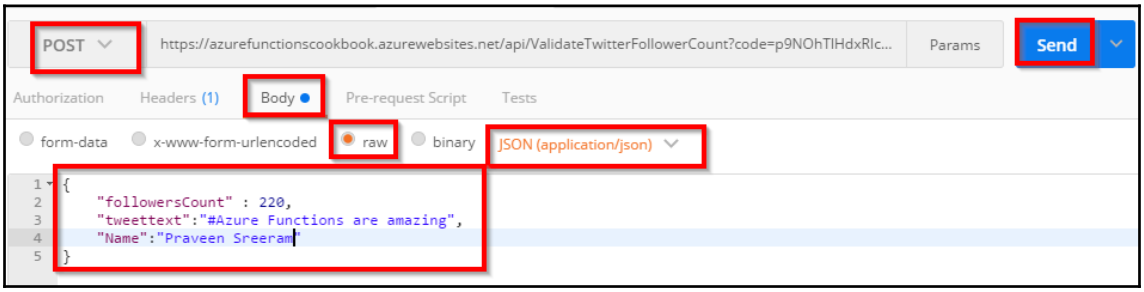


3. Replace the default code with the following and click on **Save**. The following code just checks the followers count and if it is greater than 200, then it sends out an email:

```
#r "Newtonsoft.Json"
#r "SendGrid"
using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using SendGrid.Helpers.Mail;
public static async Task<IActionResult> Run(HttpRequest req,
IAsyncCollector<SendGridMessage> messages, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a
request.");

    string name = req.Query["name"];

    string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    string strTweet = "";
    SendGridMessage message = new SendGridMessage();
```

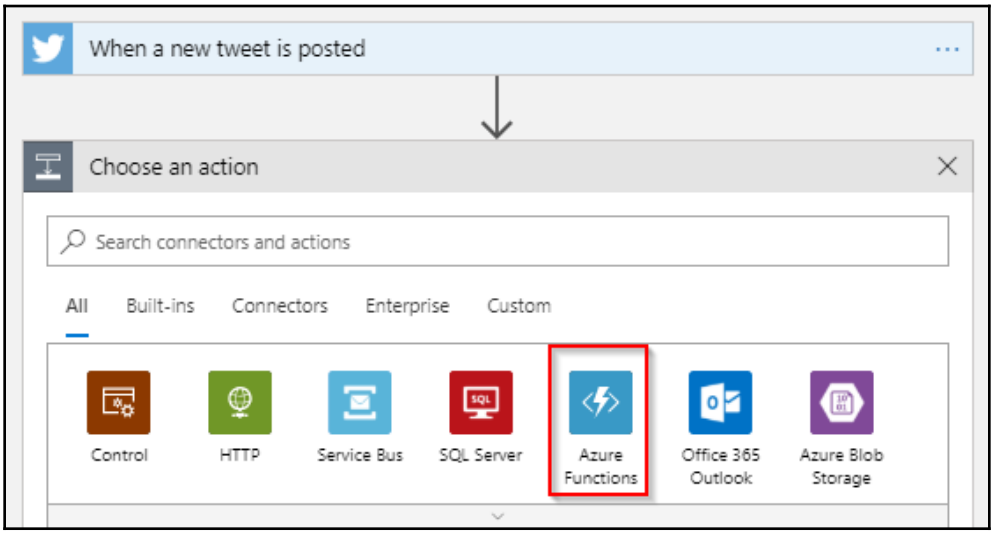
```
        if(data.followersCount >= 200)
        {
            strTweet = "Tweet Content" + data.tweettext;
            message.SetSubject($"{data.Name} with {data.followersCount}
    followers has posted a tweet");
            message.SetFrom("donotreply@example.com");
            message.AddTo("prawin2k@gmail.com");
            message.AddContent("text/html", strTweet);

        }
        else
        {
            message = null;
        }
        await messages.AddAsync(message);
         return (ActionResult)new OkObjectResult($"Hello");
    }
```

4. Test the function using Postman by choosing the parameters highlighted in the following screenshot. In the next steps, after we integrate the `ValidateTwitterFollowerCount` Azure Function, all the following input parameters, such as `followersCount`, `tweettext`, and `Name`, will be posted by the Twitter connector of the Logic App:

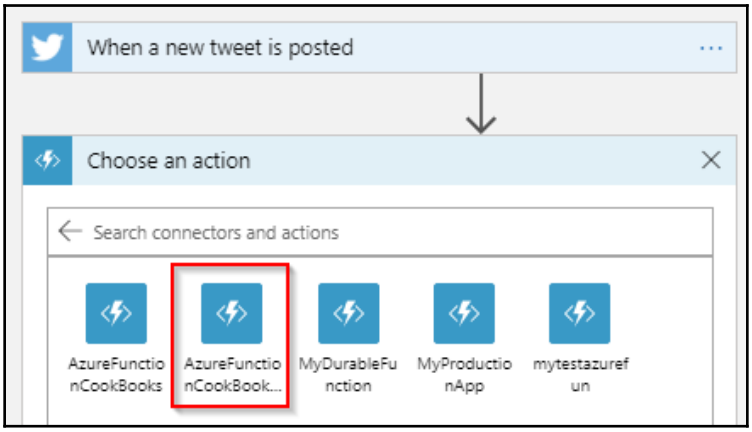


5. Create a new Logic App, named `NotifywhenTweetedbyPopularUserUsingFunctions`.
6. Start designing the app with the **Blank logic app** template, choose the **Twitter** connector, and configure **Search text**, **Frequency**, and **Interval**.

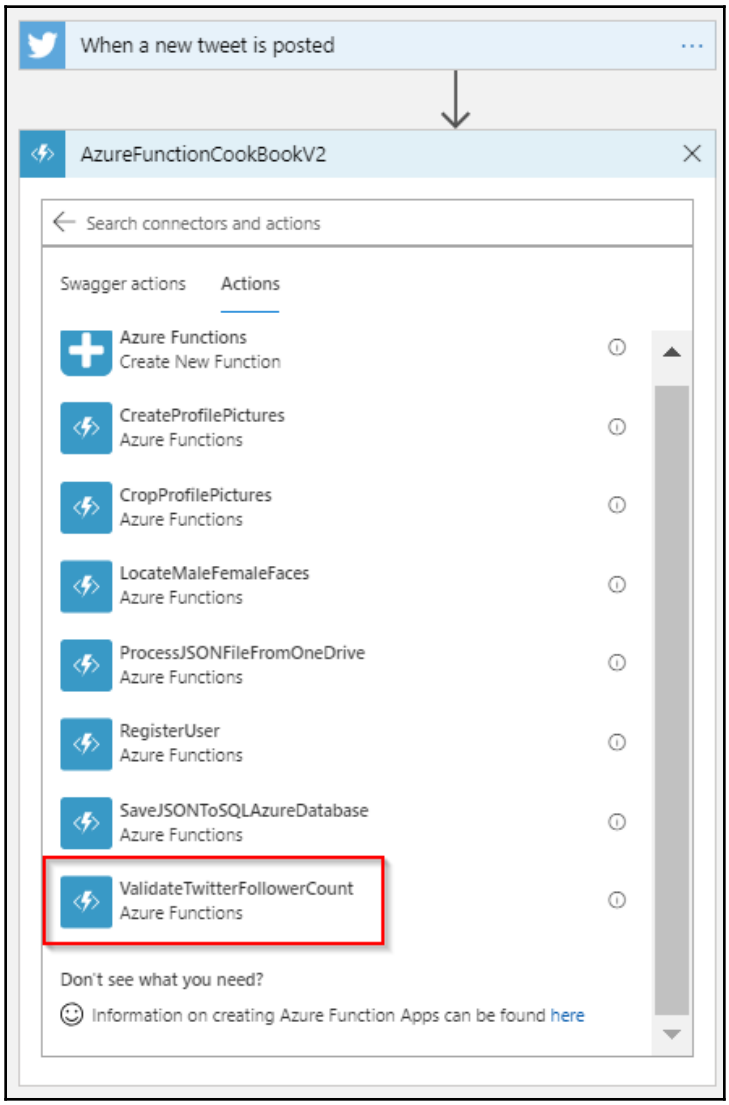
7. Click on **New step** to add an action. In the **Choose an action** section, choose **Azure Functions** as a connector, as shown in the following screenshot:



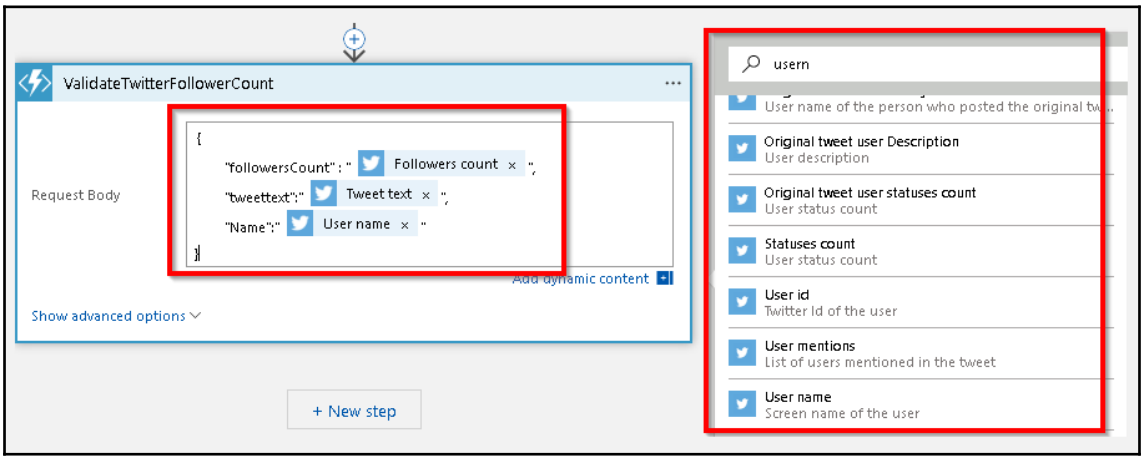
8. Clicking on **Azure Functions** will list all the available Azure function apps, as shown in the following screenshot:



9. Click on the function app in which you have created the `ValidateTwitterFollowerCount` function. Now select the `ValidateTwitterFollowerCount` function, as shown in the following screenshot:



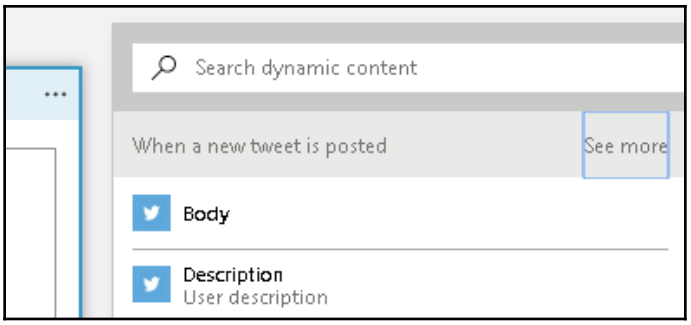
10. In the next step, you need to prepare the JSON input that needs to be passed from the Logic App to the `ValidateTwitterFollowerCount` HTTP trigger function that we developed. Let's frame input JSON in the same way that we did when testing the HTTP trigger function using Postman, as shown in the following screenshot (the only difference is that the values, such as `followersCount`, `Name`, and `tweettext`, are dynamic now):



11. Once you have reviewed all the parameters that the `ValidateTwitterFollowerCount` function expects, click on the **Save** button to save the changes.
12. You can wait for a few minutes or post a tweet with the hashtag that you have configured in the **Search text** input field.

There's more...

If you don't see the intended dynamic parameter, click on the **See more** button, as shown in the following screenshot:



In the `ValidateTwitterFollowerCount` Azure Function, we have hardcoded the follower count threshold as `200`. It's good practice to store these values as configurable items by storing them in **Application settings**.

See also

See the *Sending an email notification to the end user dynamically* recipe in `Chapter 2`, *Working with Notifications Using the SendGrid and Twilio Services*.

Auditing Cosmos DB data using change feed triggers

Many of you might have already heard about Cosmos DB, as it has become very popular and many organizations are using it because of the features it provides.

In this recipe, we will learn about integrating serverless Azure Functions with a serverless NoSQL database in Cosmos DB. You can read more about Cosmos DB at `https://docs.microsoft.com/en-us/azure/cosmos-db/introduction`.

It might often be necessary to keep change logs of fields, attributes, documents, and more for auditing purposes. In the world of relational databases, you might have seen developers using triggers or stored procedures to implement this kind of auditing functionality, where you write code to store data into a separate audit table.

In this recipe, we will learn how easy it is to achieve the preceding use case and audit Cosmos DB collections by writing a simple function that gets triggered whenever there is a change in a document of a Cosmos DB collection.

In the world of relational databases, a collection is the same as a table and a document is the same as a record.

Getting ready

In order to get started, we need to first do the following:

- Create a Cosmos DB account
- Create a new Cosmos DB collection where you can store data in the form of documents

Creating a new Cosmos DB account

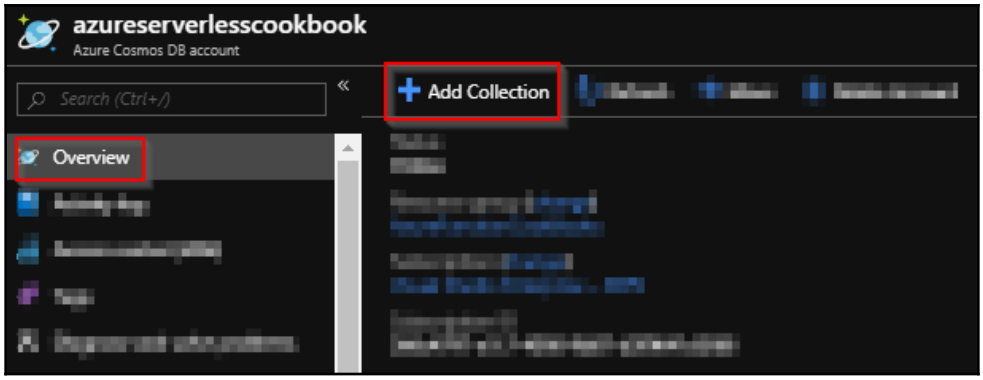
Navigate to the Azure portal and create a new Cosmos DB account. You would need to provide the following:

- A valid subscription and a resource group.
- A valid account name. This will create an endpoint at `<<accountname>>.document.azure.com`.
- An API—set this as SQL. This will ensure that you can write queries in SQL. Feel free to try out other APIs.

Creating a new Cosmos DB collection

Perform the following steps:

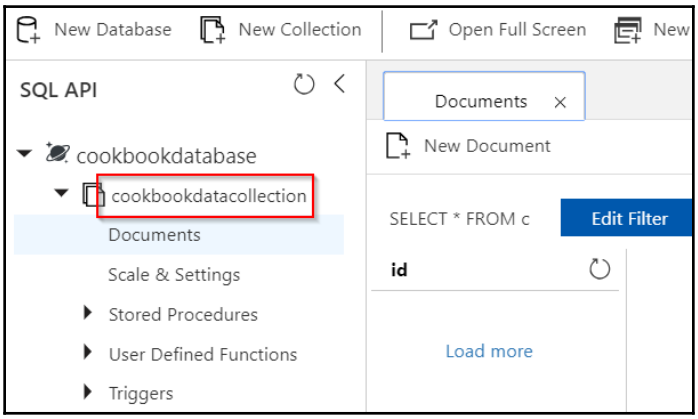
1. Once the account is created, you need to create a new database and a collection. We can create both of them in a single step right from the portal.
2. Navigate to the **Overview** tab and click on the **Add Collection** button to create a new collection:



3. You will now be navigated to the **Data Explorer** tab automatically, where you will be prompted to provide the following details:

Field Name	Value	Comment
Database id	`cookbookdatabase`	This is a container of multiple Cosmos DB collections.
Collection id	`cookbookdatacollection`	This is the name of the collection where you will be storing the data.
Storage capacity	`Fixed (10 GB)`	Depending on your production workloads, you might have to go with **Unlimited**, as you may get partitions otherwise.
Throughput (400 - 10,000 RU/s)	`400`	This is the capacity of your Cosmos DB Collection. The performance of the reads and writes on the collection depends on the throughput that you configure when provisioning the collection.

4. Next, click on the **OK** button to create the collection. If everything went well, you will see something like the following in the **Data Explorer** tab of the Cosmos DB account:

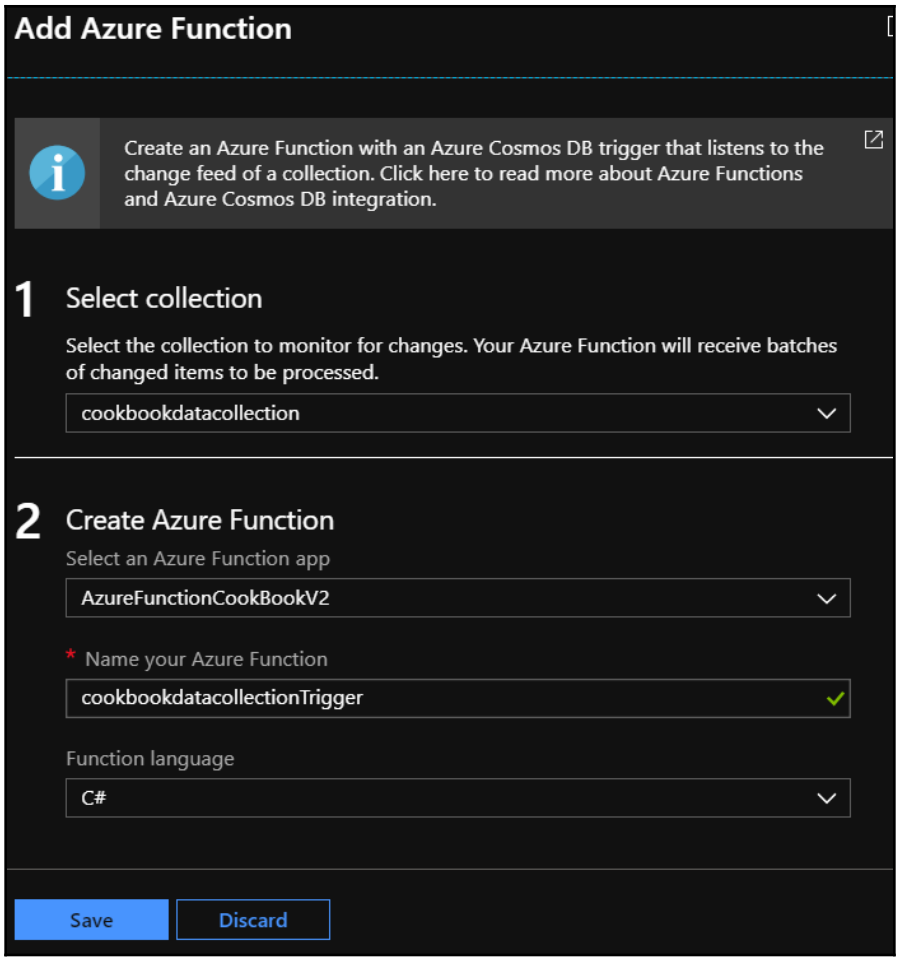


We have successfully created a Cosmos DB account and a collection. Let's now learn how to integrate the collection with a new Azure Function and see how to trigger it whenever there is a change in the Cosmos DB collection.

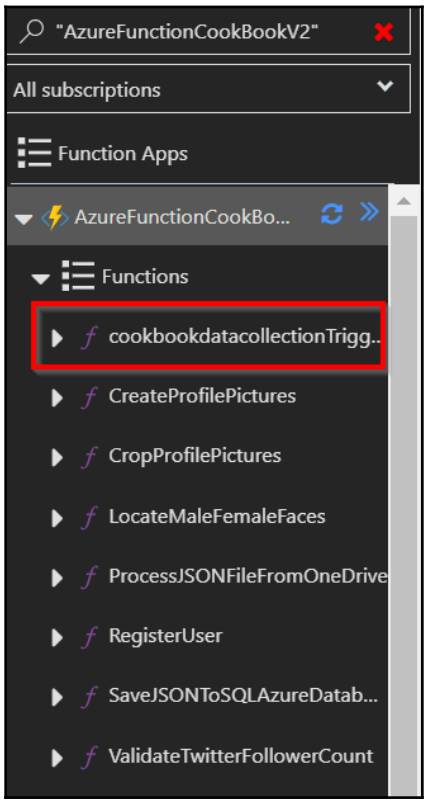
How to do it...

Perform the following steps:

1. Navigate to the Cosmos DB Account and click on the **Add Azure Function** menu in the **All settings** blade of the Cosmos DB account.
2. You will now be taken to the **Add Azure Function** blade, where you will choose the Azure function app in which you would like to create a new function (Cosmos DB trigger) to be triggered whenever a change in the collection happens. Here is how it looks:


Add Azure Function
Create an Azure Function with an Azure Cosmos DB trigger that listens to the change feed of a collection. Click here to read more about Azure Functions and Azure Cosmos DB integration.
1 Select collection
Select the collection to monitor for changes. Your Azure Function will receive batches of changed items to be processed.
cookbookdatacollection
2 Create Azure Function
Select an Azure Function app
AzureFunctionCookBookV2
* Name your Azure Function
cookbookdatacollectionTrigger
Function language
C#
Save
Discard

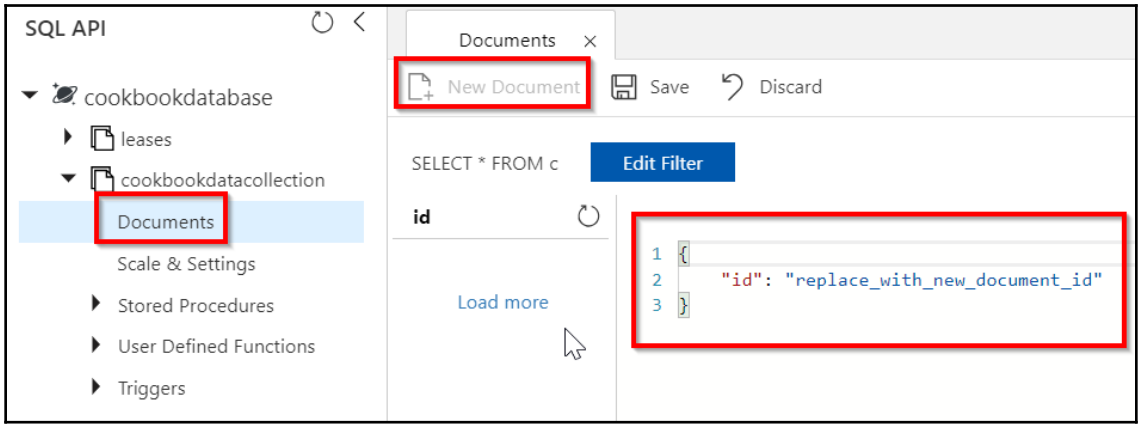
3. Once you have reviewed the details, click on the **Save** button (shown in the previous screenshot) to create the new function, which will be triggered for every change that is made in the collection. Let's quickly navigate to the Azure function app (in my case, it is `AzureFunctionCookBookV2`) and see whether the new function with the name `cookbookdatacollectionTrigger` has been created. Here is a view of my function app:



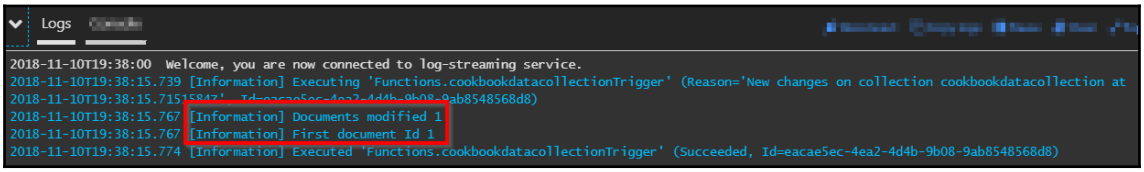
4. Replace the default code with the following code of the Azure Functions Cosmos DB trigger, which gets a list of all the documents that were updated. The following code just prints the count of documents that were updated and the `id` of the first document in the **Logs** console:

```
#r "Microsoft.Azure.DocumentDB.Core"
using System;
using System.Collections.Generic;
using Microsoft.Azure.Documents;

public static void Run(IReadOnlyList<Document> input, ILogger log)
{
    if (input != null && input.Count > 0)
    {
        log.LogInformation("Documents modified " + input.Count);
        log.LogInformation("First document Id " + input[0].Id);
    }
}
```

5. Now the integration of the Cosmos DB collection and the Azure Function is complete. Let's now add a new document to the collection and see how the trigger gets fired in action. Open a new tab (leaving the `cookbookdatacollectionTrigger` tab open in the browser), navigate to the collection, and create a new document by clicking on the **New Document** button, as shown in the following screenshot:



6. Once you have replaced the default JSON (which just has an `id` attribute) with the JSON that has the required attributes, click on the **Save** button to save the changes and quickly navigate to the other browser tab, where you have the Azure Function open, and view the logs to see the output of the function. The following is how my logs look, as I just added a value to the `id` attribute of the document. It might look different for you, depending on your JSON structure:

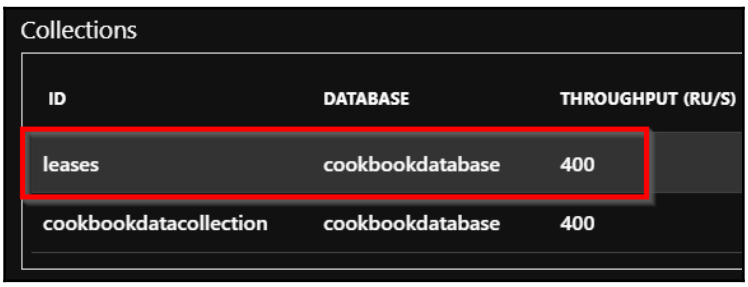


How it works...

In order to integrate Azure Functions with Cosmos DB, we first created a Cosmos DB account and created a database and a new collection within it. Once the collection was created, we integrated it from within the Azure portal by clicking on the **Add Azure Function** button, which is available at the Cosmos DB account level. We have chose the required function app in which we wanted to create a Cosmos DB trigger. Once the integration was complete, we created a sample document in the Cosmos DB collection and then verified that the function was triggered automatically for all the changes (all reads and writes but not deletes) that we make on the collection.

There's more...

When you integrate Azure Functions to track Cosmos DB changes, it will automatically create a new collection named **leases**, as shown here. Be aware that this is an additional cost as the cost in Cosmos DB is based on the **request units** (**RUs**) that are allocated for each collection:



It's important to note that the the Cosmos DB trigger wouldn't be triggered (at the time of writing) for any deletes in the collection. It is only triggered for create and updates to documents in a collection. If it is important for you to track deletes, then you need to do **soft deletes**, which means setting an attribute such as `isDeleted` to true for records that are deleted by the application and based on the value of the `isDeleted` attribute, implementing your custom logic in the Cosmos DB trigger.

The integration that we have done between Azure Functions and Cosmos DB uses Cosmos DB change feeds. You can learn more about change feeds here: `https://docs.microsoft.com/en-us/azure/cosmos-db/change-feed`.

Don't forget to delete the Cosmos DB account and its associated collections if you think you won't use them anymore, because the collections are charged based on the RUs allocated even if you are not actively using them.

If you are not able to run this Azure Function or you get error saying that Cosmos DB extensions are not installed, then try creating a new Azure Cosmos DB trigger, which should then prompt installation.

4
Understanding the Integrated Developer Experience of Visual Studio Tools

In this chapter, we will cover the following:

- Creating a function app using Visual Studio 2017
- Debugging C# Azure Functions on a local staged environment using Visual Studio 2017
- Connecting to the Azure Storage cloud from the local Visual Studio environment
- Deploying the Azure Function app to Azure Cloud using Visual Studio
- Debugging live C# Azure Function, hosted on the Microsoft Azure Cloud environment, using Visual Studio
- Deploying Azure Functions in a container

Introduction

In all of our previous chapters, we looked at how to create Azure Functions right from the Azure Management Portal. Here are a few of the features:

- You can quickly create a function just by selecting one of the built-in templates provided by the Azure Function Runtime
- Developers need not worry about writing the plumbing code and understanding how the frameworks work
- Configuration changes can be made right within the UI using the standard editor

In spite of all the advantages mentioned, developers might not find it comfortable if they became used to working with their favorite **Integrated Development Environments** (**IDEs**) a long time ago. So, the Microsoft team has come up with some tools that help developers integrate them into Visual Studio so that they can leverage some of the critical IDE features that accelerate their development efforts. Here are few of them:

- Developers benefit from IntelliSense support
- You can debug code line by line
- Quickly view the values of the variables while you are debugging the application
- Integration with version control systems such as Azure DevOps (earlier, this was called **Visual Studio Team Services** (**VSTS**))

Currently, at the time of writing, the Visual Studio tools for the function supports debugging only for C#. In the future, Microsoft is likely to come up with all these cool features for other languages.

You will learn some of these aforementioned features in this chapter, and see how to integrate code with Azure DevOps in `Chapter 11`, *Implementing and Deploying Continuous Integration Using Azure DevOps*.

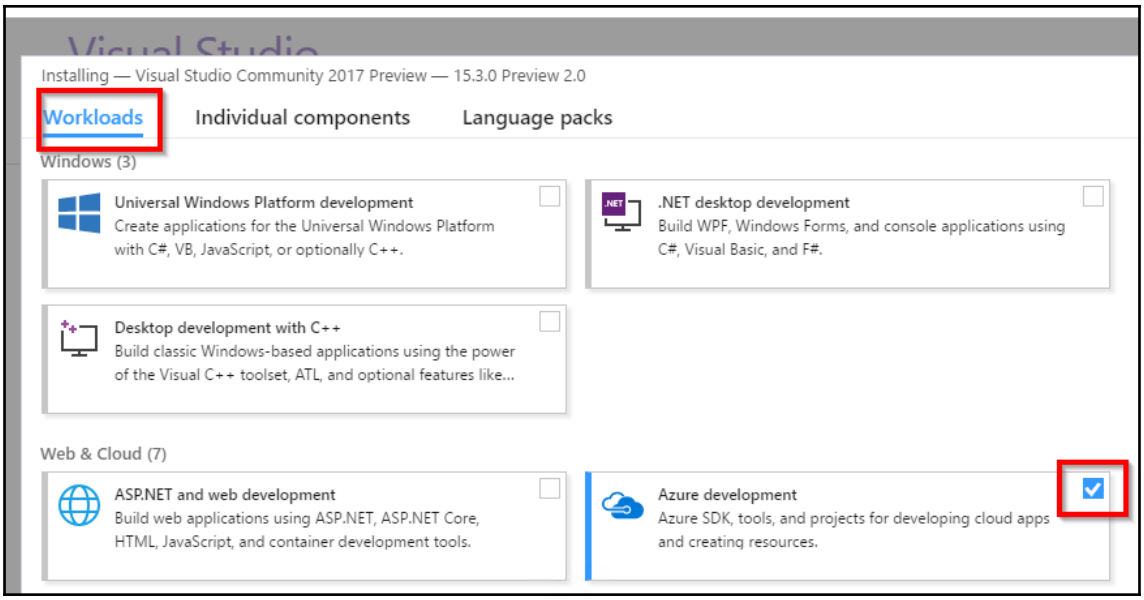
Creating a function app using Visual Studio 2017

In this recipe, you will learn how to create an Azure Function for your favorite IDE in Visual Studio 2017.

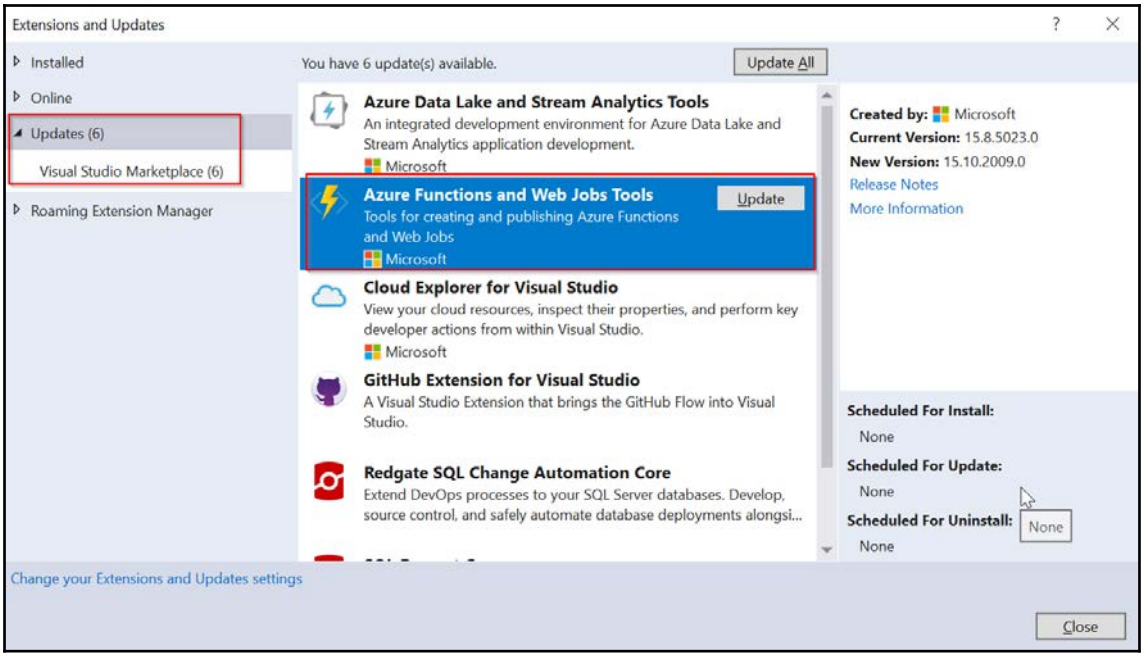
Getting ready

You need to download and install the following tools and software:

1. Download the latest version of Visual Studio 2017. You can download it from `https://visualstudio.microsoft.com/downloads/`.
2. During the installation, choose **Azure development** in the **Workloads** section, as shown in the following screenshot, and then click on the **Install** button:



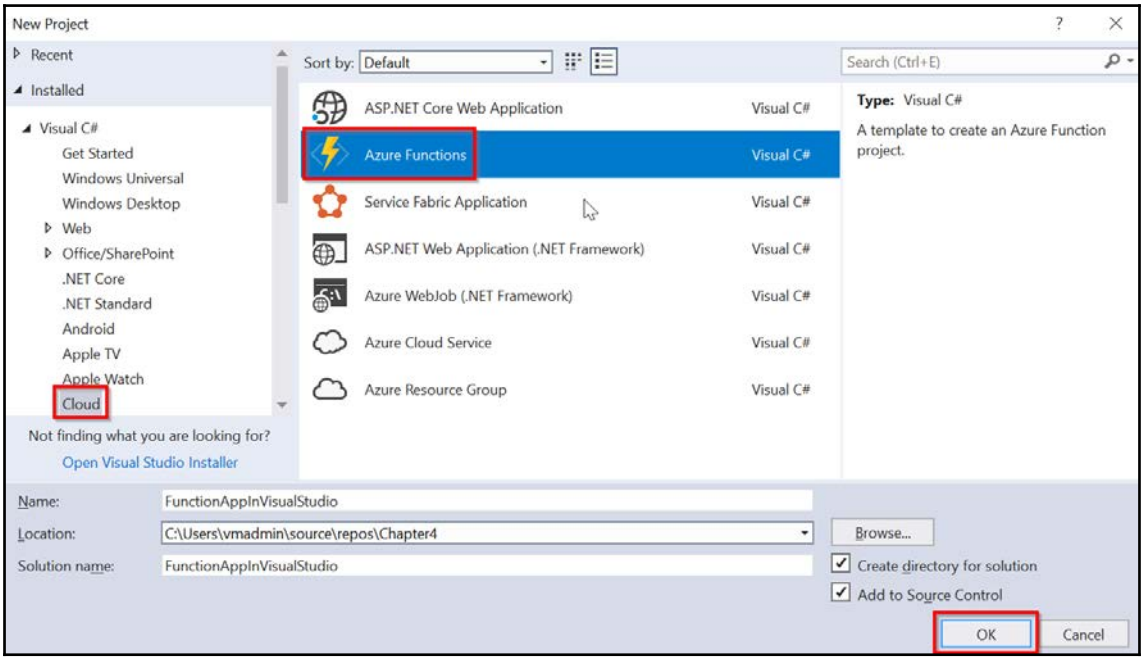
3. Navigate to **Tools** | **Extensions and Updates** and see if there are any updates to Visual Studio tools for Azure Functions, as shown in the following screenshot:



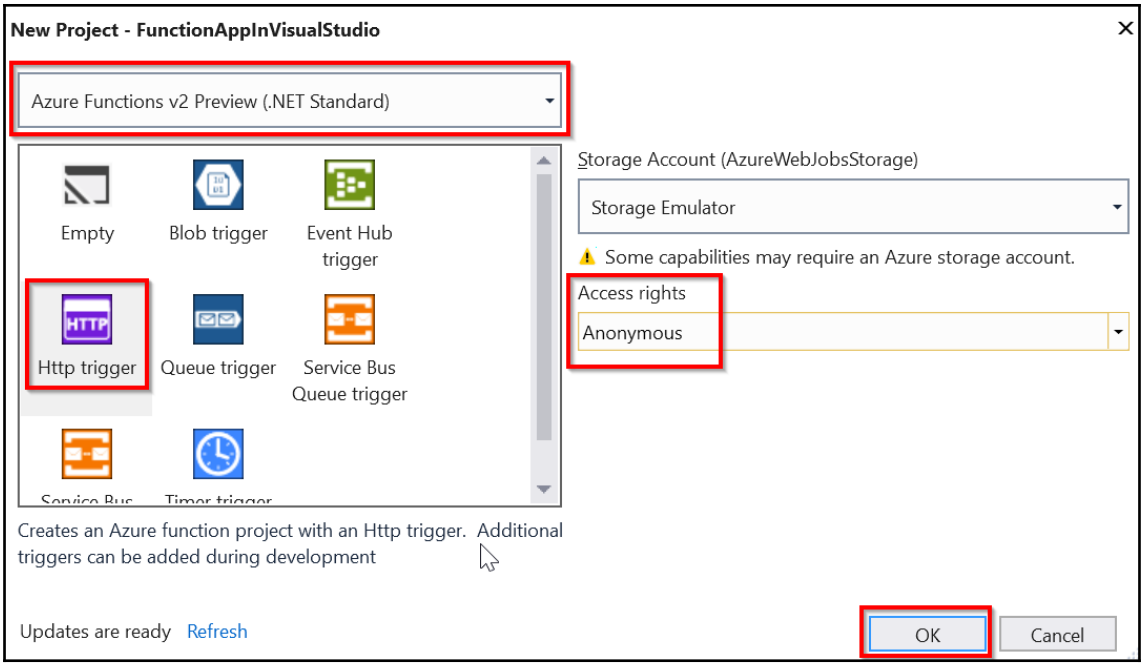
How to do it...

Perform the following steps:

1. Open Visual Studio, choose **File**, and then click on **New Project**. In the **New Project** dialog box, in the **Installed** templates, under **Visual C#**, select **Cloud**, and then select the **Azure Functions** template:

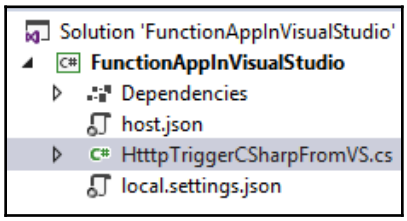


2. Provide the name of the function app. Click on the **OK** button to go to the next step. As shown in the following screenshot, choose **Azure Funtions v2 (.NET Core)** from the drop-down menu, then select **Http trigger**, and click on the **OK** button:



3. We have successfully created the Azure Function App, along with an HTTP Trigger (which accepts web requests and sends a response to the client), with the name `Function1`. Feel free to change the default name of the function app, and also make sure to build the application to download the required NuGet packages, if any.

4. After you create a new function, a new class will also be created, as shown in the following screenshot:



We have now successfully created a new HTTP-triggered function app using Visual Studio 2017.

How it works...

Visual Studio tools for Azure Functions allow developers to use their favorite IDE, which they may have been using for ages. Using the Azure Function tools, you can use the same set of templates that the Azure Management Portal provides in order to quickly create and integrate with the cloud services without writing any (or minimal) plumbing code.

The other advantage of using Visual Studio tools for functions is that you don't need to have a live Azure subscription. You can debug and test Azure Functions right in your local development environment. Azure CLI and related utilities provide us with all the required assistance to execute Azure Functions.

There's more...

One of the most common problems that developers face while developing any application on their local environment is that "*everything works fine on my local machine but not on the production environment*." Developers need not worry about this in the case of Azure Functions. The Azure Functions runtime provided by the Azure CLI tools is exactly the same as the runtime available on Azure Cloud.

Note that you can always use and trigger an Azure service running on the cloud, even when you are developing Azure Functions locally.

Debugging C# Azure Functions on a local staged environment using Visual Studio 2017

Once the basic setup of our function creation is complete, the next step is to start working on developing the application as per your needs. Developing code on a daily basis is not at all a cake walk; developers end up facing all kinds of technical issues. They need tools to help them identify the root cause of the problem and fix it to make sure they are delivering the solution. These tools include debugging tools that help developers step into each line of the code to view the values of the variable and objects and get a detailed view of the exceptions.

In this recipe, you will learn how to configure and debug an Azure Function in a local development environment within Visual Studio.

Getting ready

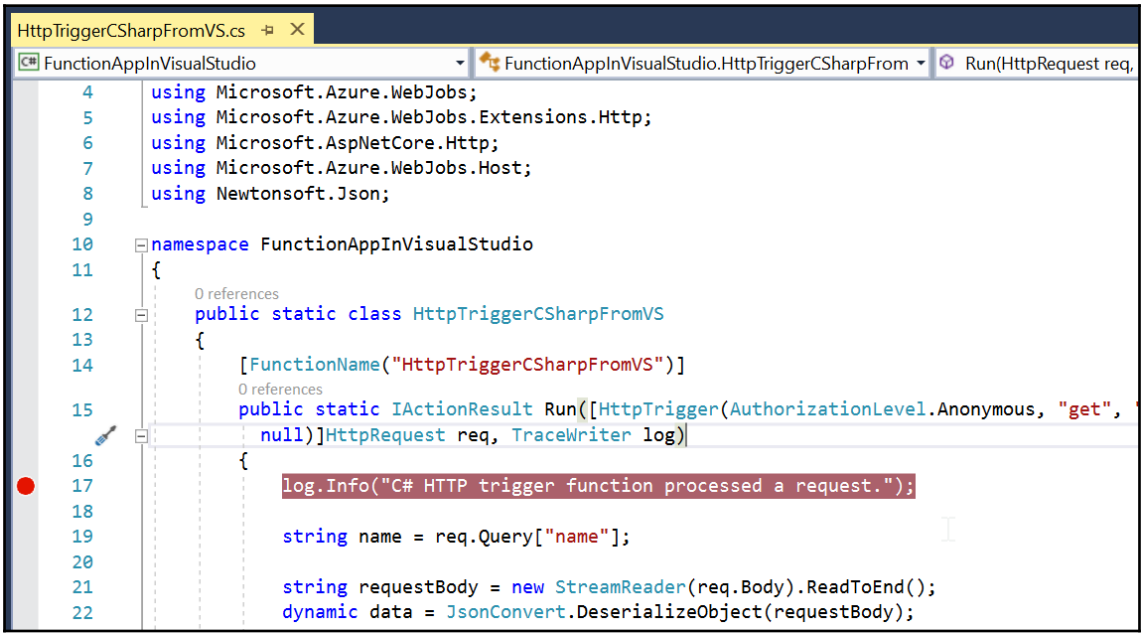
Download and install Azure CLI (if you don't have these tools installed, note that Visual Studio will automatically download them when you run your functions from Visual Studio).

How to do it...

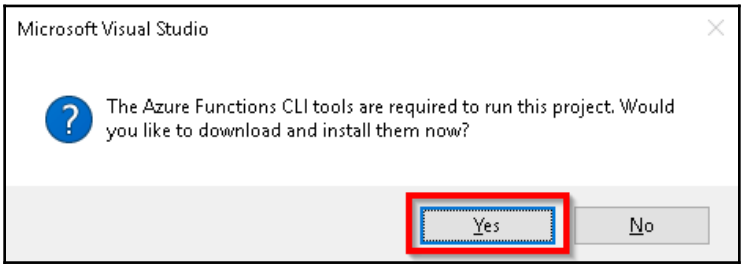
Perform the following steps:

1. In our previous recipe, we created the `HTTPTrigger` function using Visual Studio. Let's build the application by clicking on **Build** and then clicking on **Build Solution**.

2. Open the `HTTPTriggerCSharpFromVS.cs` file and create a breakpoint by pressing the *F9* key, as shown in the following screenshot:



3. Press the *F5* key to start debugging the function. When you press *F5* for the first time, Visual Studio prompts you to download Visual Studio CLI tools if they aren't already installed. These tools are essential for executing an Azure Function in Visual Studio:

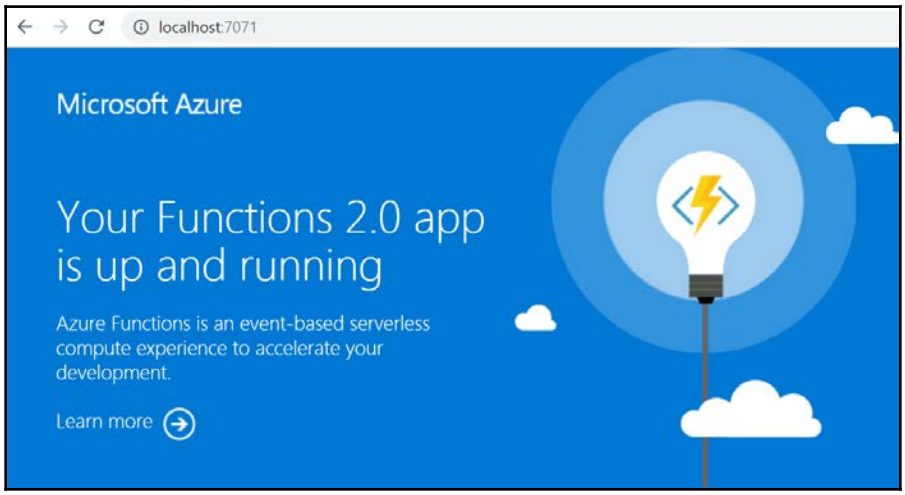


The Azure Function CLI has now been renamed to Azure Function Core Tools. You can learn more about them at `https://www.npmjs.com/package/azure-functions-core-tools`.

4. Clicking on **Yes** in the preceding screenshot will start downloading the CLI tools. The download and installation of the CLI tools will take a few minutes.
5. After the Azure Function CLI tools are installed successfully, a job host will be created and started. It starts monitoring requests on a specific port for all the functions of our function app. The following is the screenshot that shows that the job host has started monitoring the requests to the function app:

```
C:\Users\vmadmin\AppData\Local\AzureFunctionsTools\Releases\2.8.1\cli\func.exe
info: Host.Startup[0]
      Reading host configuration file 'C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio
ualStudio\bin\Debug\netstandard2.0\host.json'
info: Host.Startup[0]
      Host configuration file read:
      {}
[9/23/2018 11:35:17 AM] Initializing Host.
[9/23/2018 11:35:17 AM] Host initialization: ConsecutiveErrors=0, StartupCount=1
[9/23/2018 11:35:17 AM] Starting JobHost
[9/23/2018 11:35:17 AM] Starting Host (HostId=vm2017-1799987705, InstanceId=2e6439d4-91e0-44a1-b7fd-a05
n=2.0.12115.0, ProcessId=9760, AppDomainId=1, Debug=False, FunctionsExtensionVersion=)
[9/23/2018 11:35:17 AM] Generating 1 job function(s)
[9/23/2018 11:35:17 AM] Found the following functions:
[9/23/2018 11:35:17 AM] FunctionAppInVisualStudio.HttpTriggerCSharpFromVS.Run
[9/23/2018 11:35:17 AM]
[9/23/2018 11:35:17 AM] Host initialized (608ms)
[9/23/2018 11:35:17 AM] Host started (624ms)
[9/23/2018 11:35:17 AM] Job host started
Hosting environment: Production
Content root path: C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisual
netstandard2.0
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Listening on http://0.0.0.0:7071/
Hit CTRL-C to exit...

Http Functions:

        HttpTriggerCSharpFromVS: http://localhost:7071/api/HttpTriggerCSharpFromVS
```

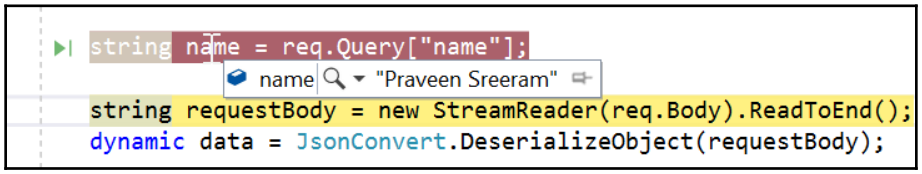
6. Let's try to access the function app by making a request to `http://localhost:7071` in your favorite browser:



7. Now key in the complete URL of our HTTP trigger in the browser. It should look like this: `http://localhost:7071/api/HttpTriggerCsharpFromVS?name=Praveen Sreeram`.
8. After typing the correct URL of the Azure Function, as soon as we hit the *Enter* key in the address bar of the browser, the Visual Studio debugger hits the debugging point (if you have one), as shown in the following screenshot:

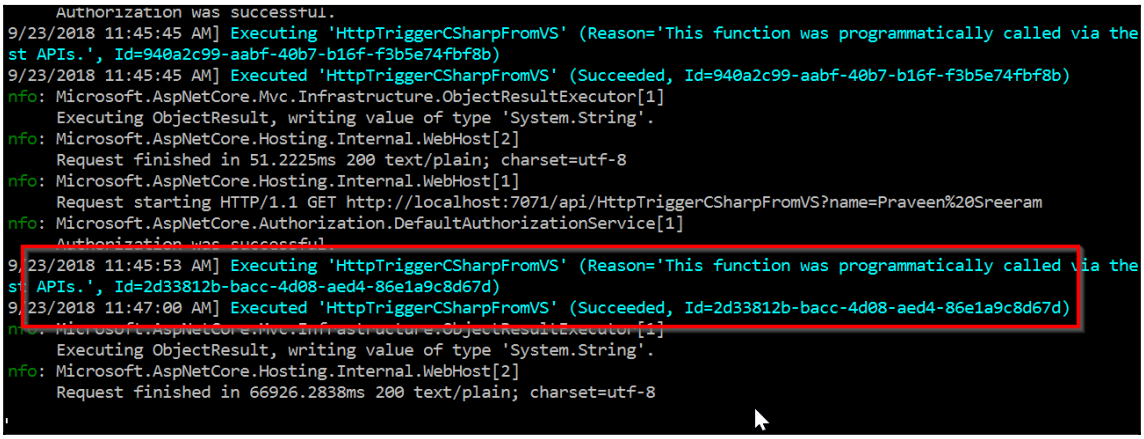
```
10  namespace FunctionAppInVisualStudio
11  {
        0 references
12      public static class HttpTriggerCSharpFromVS
13      {
14          [FunctionName("HttpTriggerCSharpFromVS")]
            0 references
15          public static IActionResult Run([HttpTrigger(AuthorizationLevel.Anonym
              null)]HttpRequest req, TraceWriter log)
16          {
17              log.Info("C# HTTP trigger function processed a request.");
18
19              string name = req.Query["name"];
20
21              string requestBody = new StreamReader(req.Body).ReadToEnd();
22              dynamic data = JsonConvert.DeserializeObject(requestBody);
23              name = name ?? data?.name;
```

9. You can also view the data of your variables, as shown in the following screenshot:



10. Once you complete the debugging, you can click on the *F5* key to complete the execution process, after which you will see the output response in the browser, as shown in the following screenshot:

11. The function execution log will be seen in the job host console, as shown in the following screenshot:



12. You can add more Azure Functions to the function app, if required. In the next recipe, we will look at how to connect to the Azure Storage cloud from the local environment.

How it works...

The job host works as a server that listens to a specific port. If there are any requests to that particular port, it automatically takes care of executing the requests and sends a response.

The job host console provides you with the following details:

- The status of the execution, along with the request and response data
- The details about all the functions available in the function app

There's more...

Using Visual Studio, you can directly create precompiled functions, which means that when you build your functions, Visual Studio creates a `.dll` file that can be referenced in other applications, just as you do for your regular classes. The following are two of the advantages of using precompiled functions:

- Precompiled functions have better performance, as they aren't required to be compiled on the fly
- You can convert your traditional classes into Azure Functions easily, and refer them in other applications seamlessly

Connecting to the Azure Storage cloud from the local Visual Studio environment

In both of the previous recipes, you learned how to create and execute Azure Functions in a local environment. We triggered the function from a local browser. However, in this recipe, you will learn how to trigger an Azure Function in your local environment when an event occurs in Azure. For example, when a new Blob is created in a Azure Storage account, you can have your function triggered on your local machine. This helps developers test their applications upfront, before deploying them to the production environment.

Getting ready

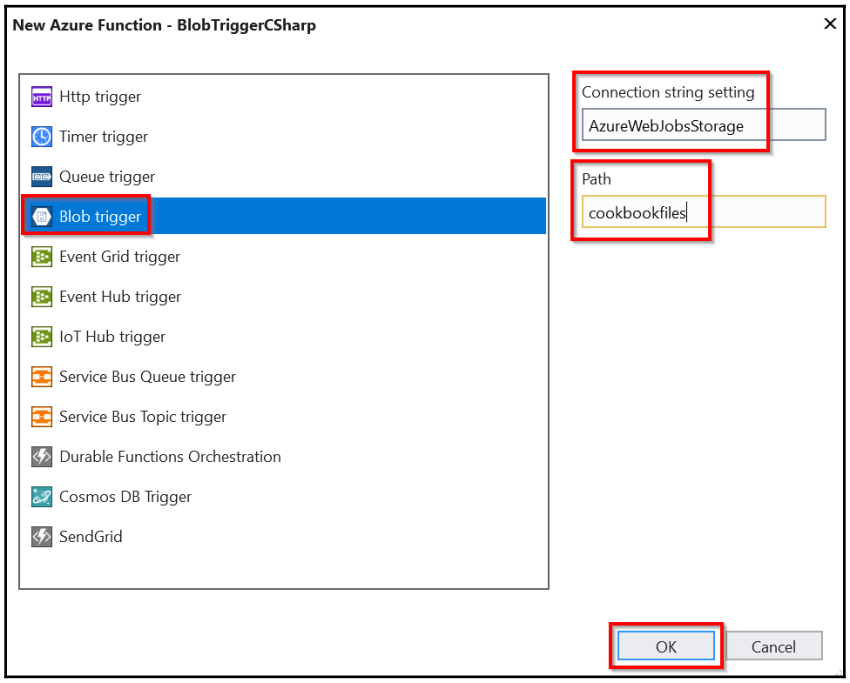
Perform the following prerequisites:

1. Create a storage account, and then a Blob container named `cookbookfiles`, in Azure.
2. Install Microsoft Azure Storage Explorer from `http://storageexplorer.com/`.

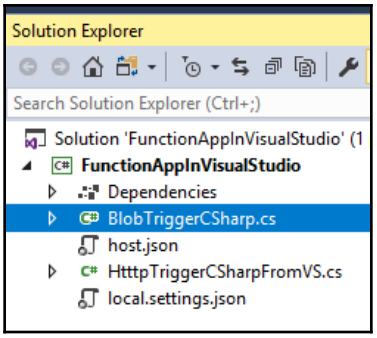
How to do it...

Perform the following steps:

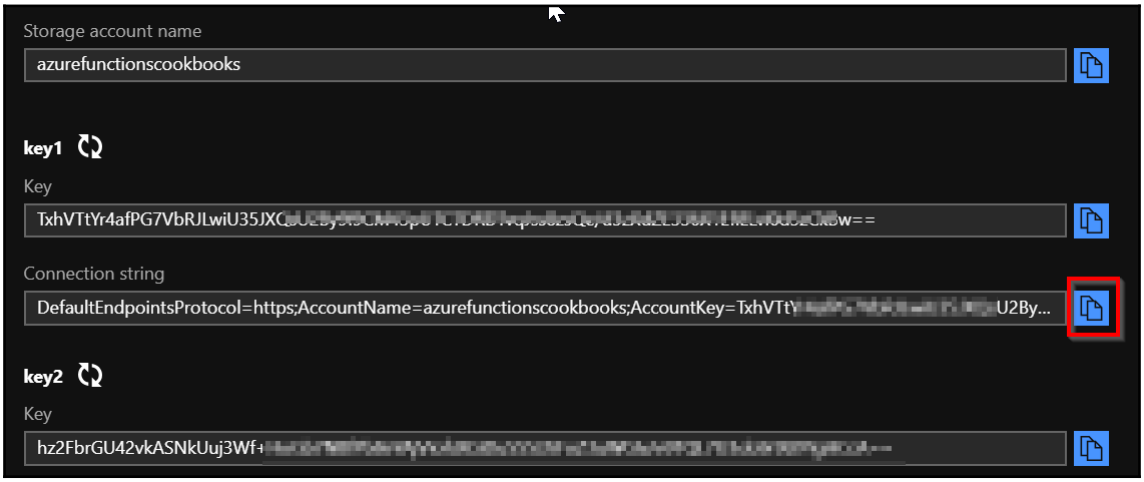
1. Open the `FunctionAppInVisualStudio` Azure Function app in Visual Studio, and then add a new function by right-clicking on the `FunctionAppInVisualStudio` project. Click on **Add** | **New Azure Function**, which opens a popup. Here, for the name field, enter `BlobTriggerCSharp` and then click on the **Add** button.
2. This opens another popup, where you can provide other parameters, as shown in the following screenshot:



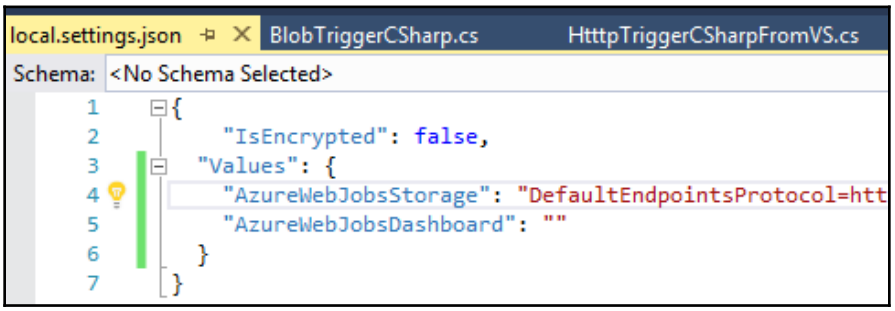
3. In the storage account connection settings, provide `AzureWebJobsStorage` as the name of the connection string, and also provide the name of the Blob container (in my case, it is `cookbookfiles`) in the **Path** input field, then click on the **OK** button to create the new Blob trigger function. A new Blob trigger function gets created, as shown in the following screenshot:



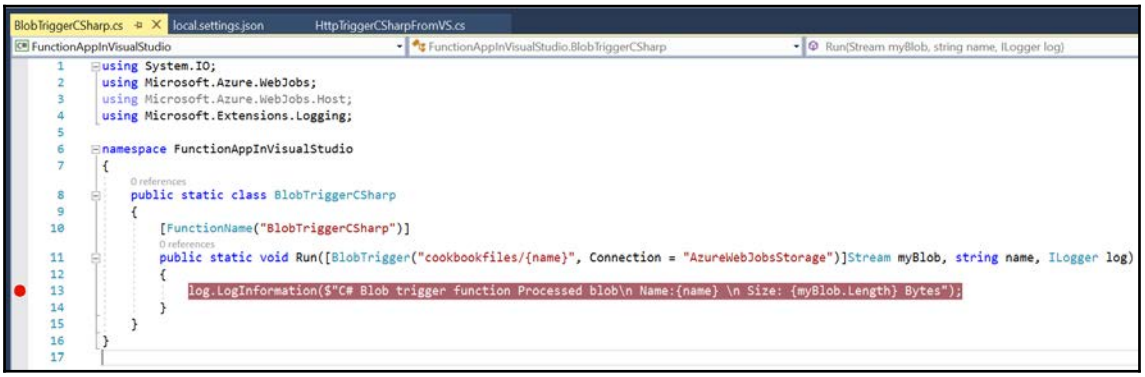
4. If you remember the *Building a backend Web API using HTTP triggers* recipe from `Chapter 1`, *Developing Cloud Applications Using Function Triggers and Bindings*, the Azure Management Portal allowed us to choose between a new or existing storage account. However, the preceding dialog box is not connected to your Azure subscription. So, you need to navigate to the storage account and copy the connection string, which can be found in the **Access Keys** blade of the storage account in the Azure Management Portal, as shown in the following screenshot:



5. Paste the connection string in the `local.settings.json` file, which is in the `root` folder of the project. This file is created when you create the function app. After you add the connection string to the key named `AzureWebJobsStorage`, the `local.settings.json` file should look like that shown in the following screenshot :



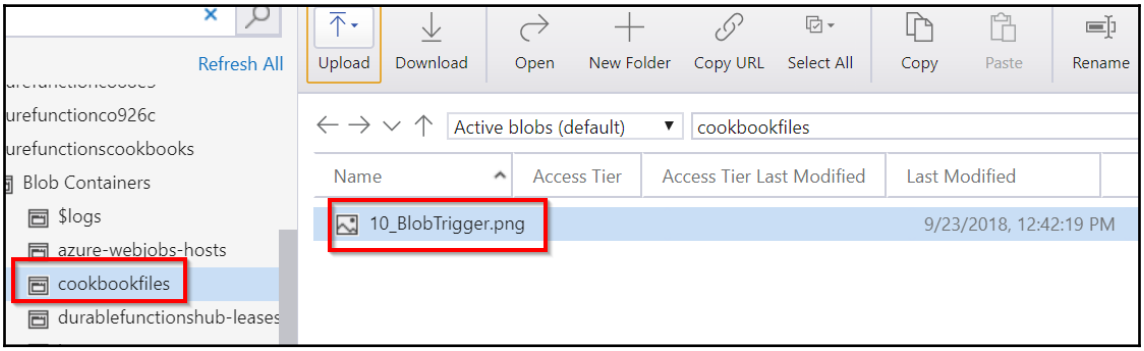
6. Open the `BlobTriggerCSharp.cs` file and create a breakpoint, as shown in the following screenshot:



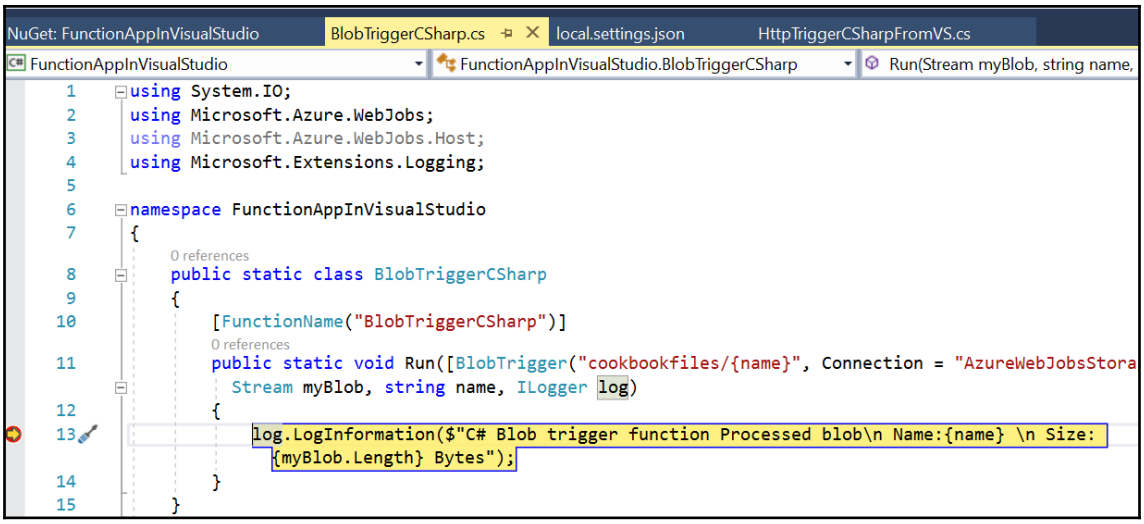
7. Now press the *F5* key to start the job host, as shown in the following screenshot:

```
[9/23/2018 12:39:21 PM] Initializing Host.
[9/23/2018 12:39:21 PM] Host initialization: ConsecutiveErrors=0, StartupCount=1
[9/23/2018 12:39:21 PM] Starting JobHost
[9/23/2018 12:39:21 PM] Starting Host (HostId=vm2017-1799987705, InstanceId=35c7497e-6bb8-4454-
n=2.0.12115.0, ProcessId=13672, AppDomainId=1, Debug=False, FunctionsExtensionVersion=)
[9/23/2018 12:39:22 PM] Generating 2 job function(s)
[9/23/2018 12:39:22 PM] Found the following functions:
[9/23/2018 12:39:22 PM] FunctionAppInVisualStudio.BlobTriggerCSharp.Run
[9/23/2018 12:39:22 PM] FunctionAppInVisualStudio.HttpTriggerCSharpFromVS.Run
[9/23/2018 12:39:22 PM]
[9/23/2018 12:39:22 PM] Host initialized (502ms)
[9/23/2018 12:39:25 PM] Host started (3363ms)
[9/23/2018 12:39:25 PM] Job host started
Listening on http://0.0.0.0:7071/
Hit CTRL-C to exit...
Hosting environment: Production
```

8. I have added a new Blob file using Azure Storage Explorer, as shown in the following screenshot:



9. As soon as the Blob has been added to the specified container (in this case, it is `cookbookfiles`), which is sitting in the cloud in a remote location, the job host running in my local machine detects that a new Blob has been added and the debugger hits the function, as shown in the following screenshot:



How it works...

In this `BlobTriggerCSharp` class, the `Run` method has the WebJobs attributes with a connection string (in this case, it is `AzureWebJobsStorage`). This instructs the runtime to refer to the Azure Storage connection string in the local settings configuration file with the key named after the `AzureWebJobsStorage` connection string. When the job host starts running, it uses the connection string and keeps an eye on the storage accounts containers that we have specified. Whenever a new Blob is added or updated, it automatically triggers the Blob trigger in the current environment.

There's more...

When you create Azure Functions in the Azure Management Portal, you need to create triggers and output bindings in the **Integrate** tab of each Azure Function. However, when you create a function from the Visual Studio 2017 IDE, you can just configure WebJobs attributes to achieve this.

You can learn more about WebJobs attributes at `https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-get-started`.

Deploying the Azure Function app to Azure Cloud using Visual Studio

So far, our function app is just a regular application within Visual Studio. To deploy the function app along with its functions, we need to either create the following new resources, or select existing ones to host the new function app:

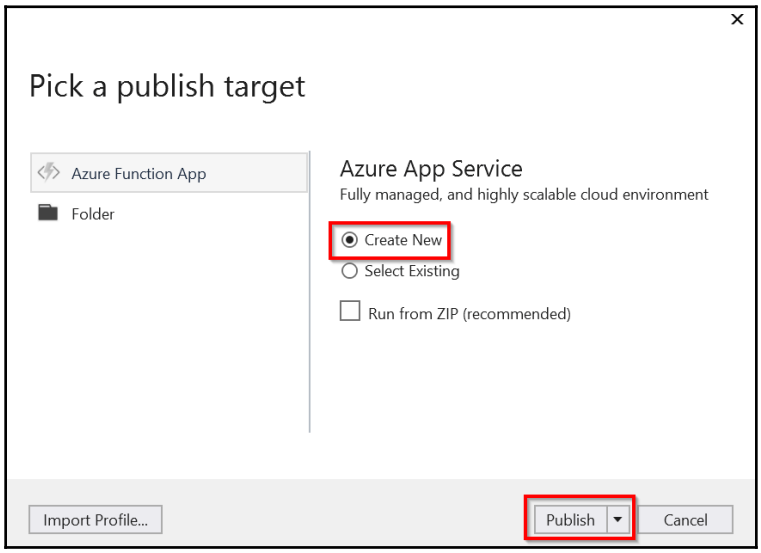
- The resource group
- The App Service plan
- The Azure Function app

You can provide all these details directly from Visual Studio without opening the Azure Management Portal. You will learn how to do that in this recipe.

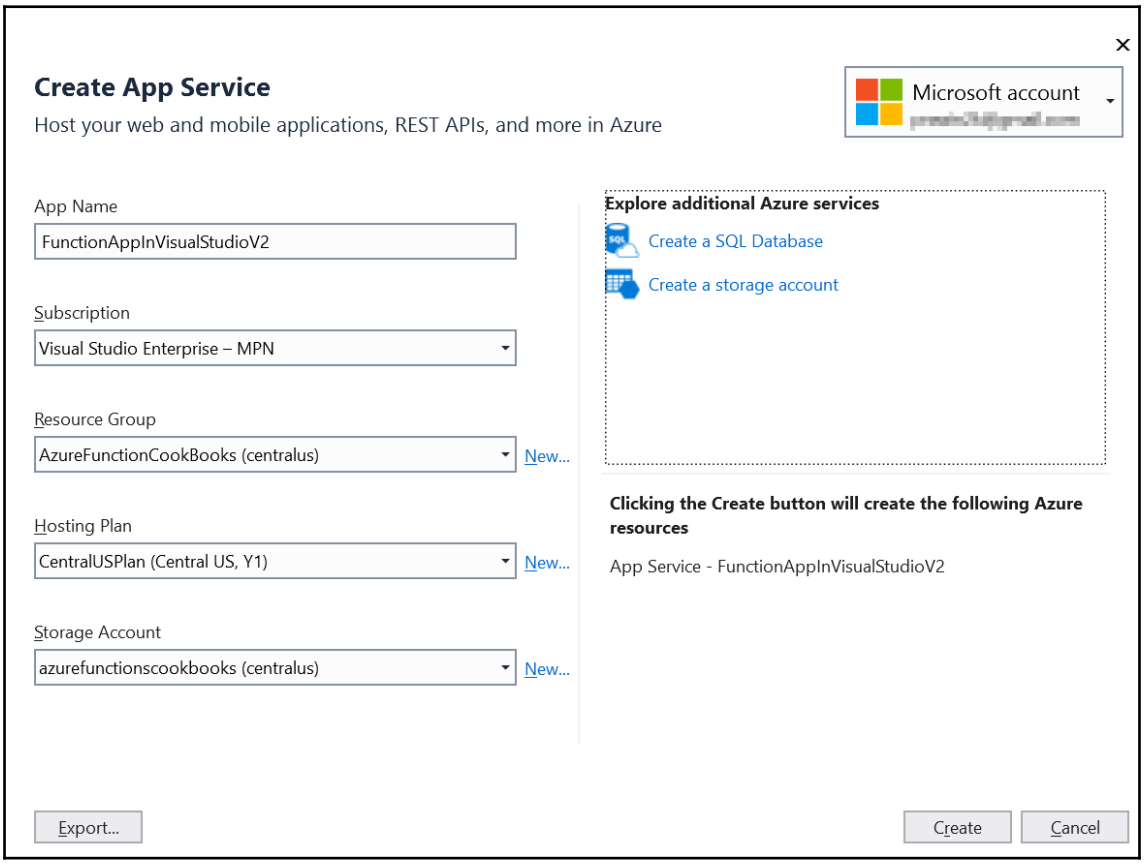
How to do it...

Perform the following steps:

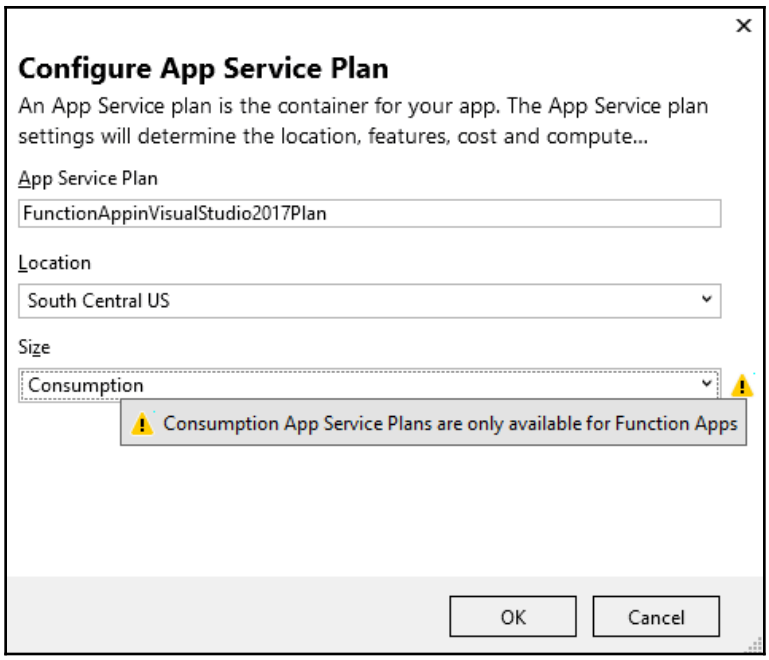
1. Right-click on the project and then click on the **Publish** button to open the publish window.
2. In the **Publish** window, choose the **Create New** option and click on the **Publish** button, as shown in the following screenshot:



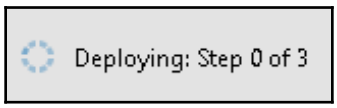
3. In the **Create App Service** window, you can choose from existing resources, or click on the **New** button to choose the new **Resource Group**, the App Service plan, and the **Storage Account**, as shown in the following screenshot:



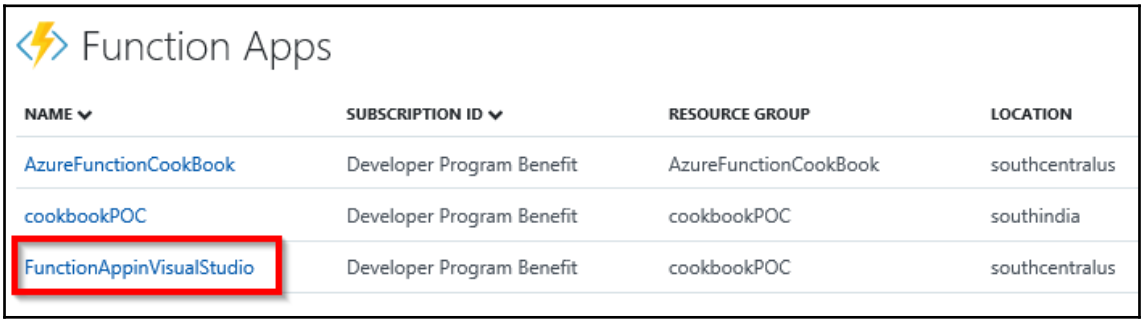
4. In most of the cases, you are best off going with the **Consumption** plan for hosting the Azure Functions, unless you have a strong reason not to, and would prefer to utilize one of your existing App Services. To choose the **Consumption** plan, you need to click on the **New** button that is available for the App Service plan, as shown in the preceding screenshot. Select **Consumption** in the **Size** drop-down menu, then click on the **OK** button, as shown in the following screenshot:



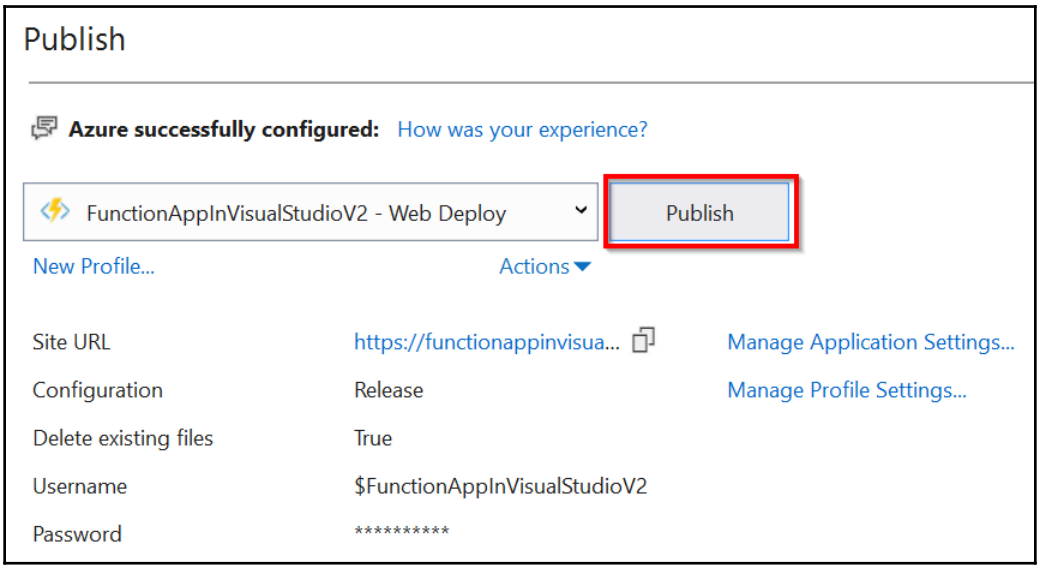
5. After reviewing all the information, click on the **Create** button of the **Create App Service** window. This should start deploying the services to Azure, as shown in the following screenshot:



6. If everything goes fine, you can view the newly created function app in the Azure Management Portal, as shown in the following screenshot:



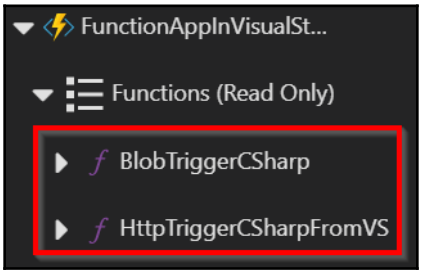
7. Hold on! Our job in Visual Studio is not yet done. We have just created the required services in Azure right from the Visual Studio IDE. Our next job is to publish the code from the local workstation to the Azure cloud. As soon as the deployment is complete, you will be taken to the web deploy step, as shown in the screenshot. Click on the **Publish** button to start the process of publishing the code:



8. After a few seconds, you should see something similar to the following screenshot in the **Output** window of your Visual Studio instance:

```
Publish Started
FunctionAppInVisualStudio -> C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio
  \FunctionAppInVisualStudio.dll
FunctionAppInVisualStudio -> C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio
Updating file (FunctionAppInVisualStudioV2\bin\extensions.json).
Updating file (FunctionAppInVisualStudioV2\BlobTriggerCSharp\function.json).
Updating file (FunctionAppInVisualStudioV2\FunctionAppInVisualStudio.deps.json).
Updating file (FunctionAppInVisualStudioV2\host.json).
Updating file (FunctionAppInVisualStudioV2\HttpTriggerCSharpFromVS\function.json).
Publish Succeeded.
Publish completed.
```

9. That's it. We have completed the deployment of your function app and its functions to Azure right from your favorite development IDE, Visual Studio. You can review the function deployment in the Azure Management Portal. Both Azure Functions were created successfully, as shown in the following screenshot:



There's more...

Azure Functions that are created from Visual Studio 2017 are precompiled, which means that you deploy the `.dll` files from Visual Studio 2017 to Azure. Therefore, you cannot edit the functions' code in Azure after you deploy them. However, you can make changes to the configurations, such as changing the Azure Storage connection string, the container path, and so on. We will look at how to do this in the next recipe.

Debugging a live C# Azure Function, hosted on the Microsoft Azure Cloud environment, using Visual Studio

In one of the previous recipes, *Connecting to the Azure Storage cloud from the local Visual Studio environment*, you learned how to connect the cloud storage account from the local code. In this recipe, you will learn how to debug the live code running in the Azure Cloud environment. We will be performing the following steps in the `BlobTriggerCSharp` function of the `FunctionAppinVisualStudio` function app:

- Changing the path of the container in the Azure Management Portal to that of the new container
- Opening the function app in Visual Studio 2017
- Attaching the debugger from within Visual Studio 2017 to the required Azure Function
- Creating a Blob in the new storage container
- Debugging the application after the breakpoints are hit

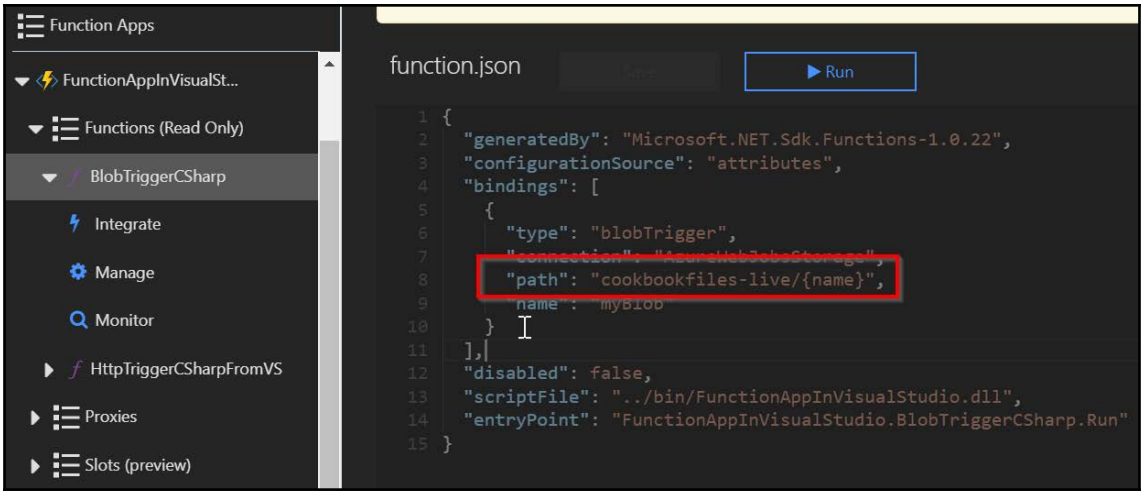
Getting ready

Create a container named `cookbookfiles-live` in the storage account. We will be uploading a Blob to this container.

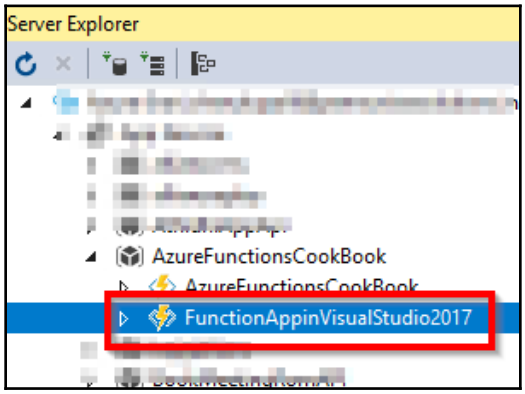
How to do it...

Perform the following steps:

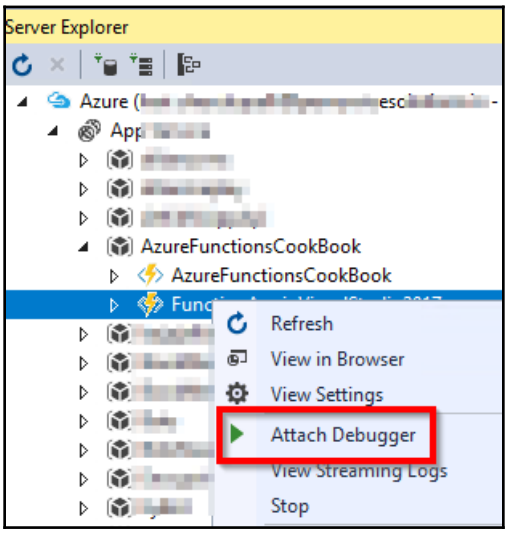
1. Navigate to the `BlobTriggerCSharp` function in the Azure Management Portal and change the path of the `path` variable to point to the new container `cookbookfiles-live`. Then, re-publish it. It should look something like that shown in the following screenshot:



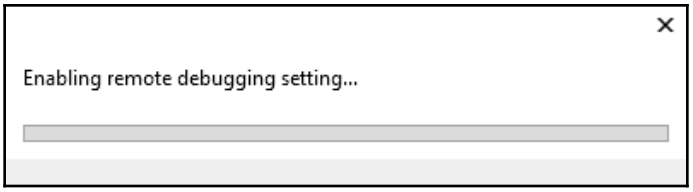
2. Open the function app in Visual Studio 2017. Open **Server Explorer** and navigate to your Azure Function; in this case, `FunctionAppinVisualStudio2017`, as shown in the following screenshot:



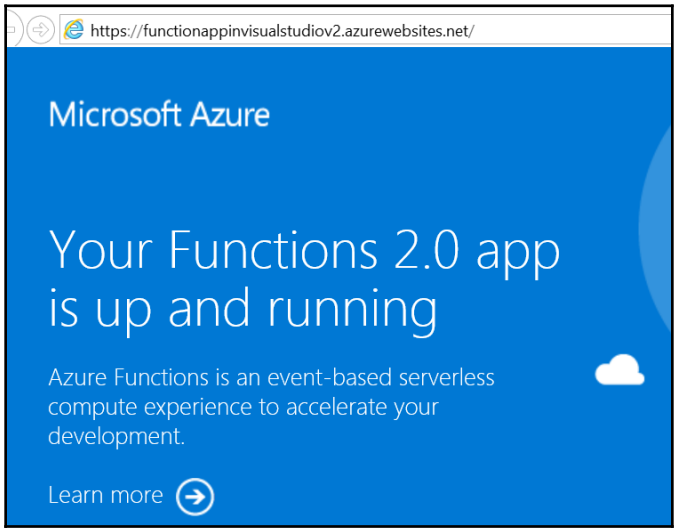
3. Right-click on the function and click on **Attach Debugger**, as shown in the following screenshot:



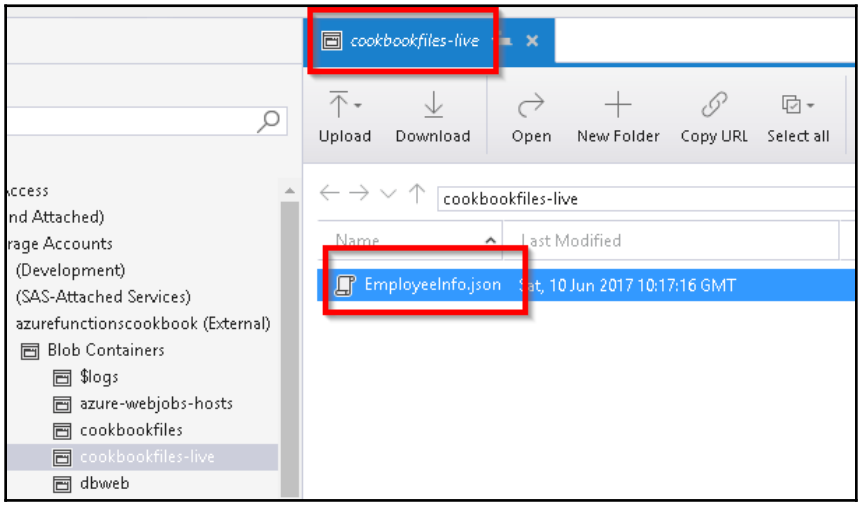
4. Visual Studio will take some time to enable remote debugging, as shown in the following screenshot:



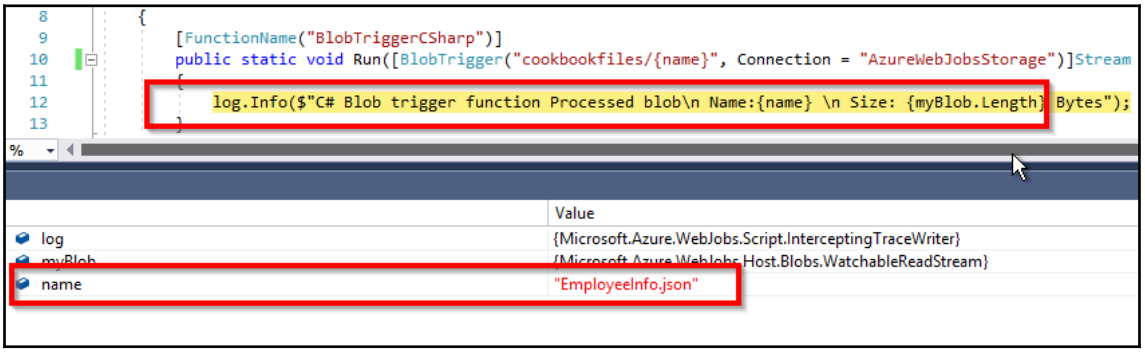
5. The function app URL will be opened in the browser, as shown in the following screenshot, indicating that our function app is running:



6. Navigate to **Storage Explorer** and upload a new file (in this case, I uploaded `EmployeeInfo.json`) to the `cookbookfiles-live` container, as shown in the following screenshot:



7. After a few moments, the debug breakpoint will be hit, shown as follows, where you can view the filename that has been uploaded:



Deploying Azure Functions in a container

By now, you might have understood the major use case of why one might use Azure Functions. Yes—when developing a piece of code and deploying it in a serverless environment, where a developer or administrator doesn't need to worry about the provisioning and scaling of instances for hosting your server-side applications.

Making all of the features of serverless could only be achieved when you create your Function App, by choosing the **Consumption** plan in the **Hosting Plan** drop-down menu.

By looking at the title of this recipe, you might already be wondering why and how deploying an Azure Function to a Docker container would help. Yes, the combination of Azure Functions and Docker Container might not make sense, as you would lose all the serverless benefits of Azure Functions when you deploy to Docker. However, there might be some customers whose existing workloads might be in some cloud (be it public or private), but now they want to leverage some of the Azure Function triggers and related Azure Services, and so would want to deploy the Azure Functions as a Docker image. This recipe deals with how to implement this.

Getting ready

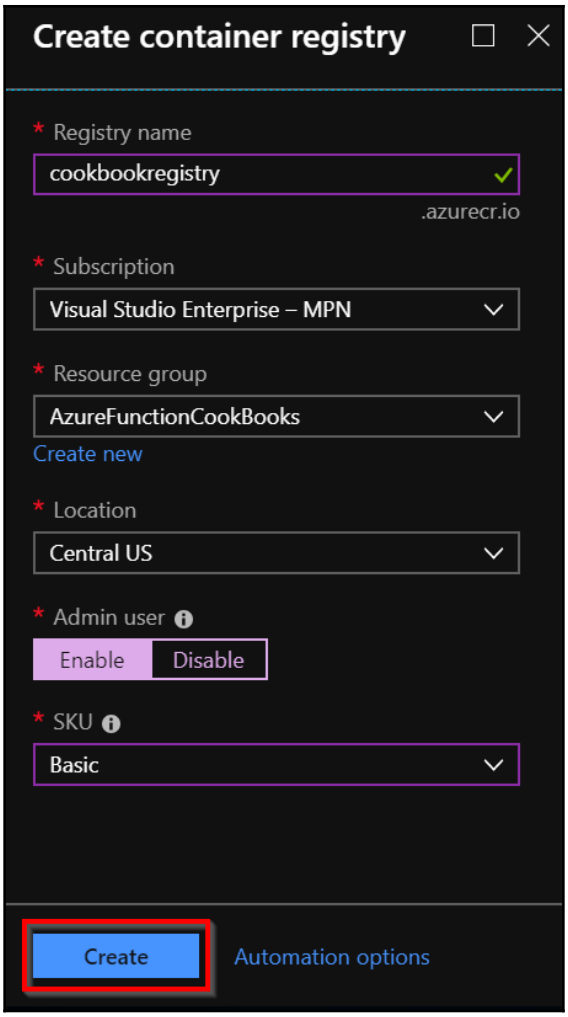
The following are the prerequisites for getting started with this recipe:

- Please install Azure CLI from `https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest`
- You can download Docker from `https://store.docker.com/editions/community/docker-ce-desktop-windows`. Ensure that you install the version of Docker that is compatible with the operating system of your development environment.
- Also required is basic knowledge of Docker and its commands, in order to build and run Docker images. You can go through the official Docker documentation to learn this, if you don't already have this knowledge.
- Create an **Azure Container Registry** (**ACR**) with the following steps. This can be used as a repository for all of the Docker images.

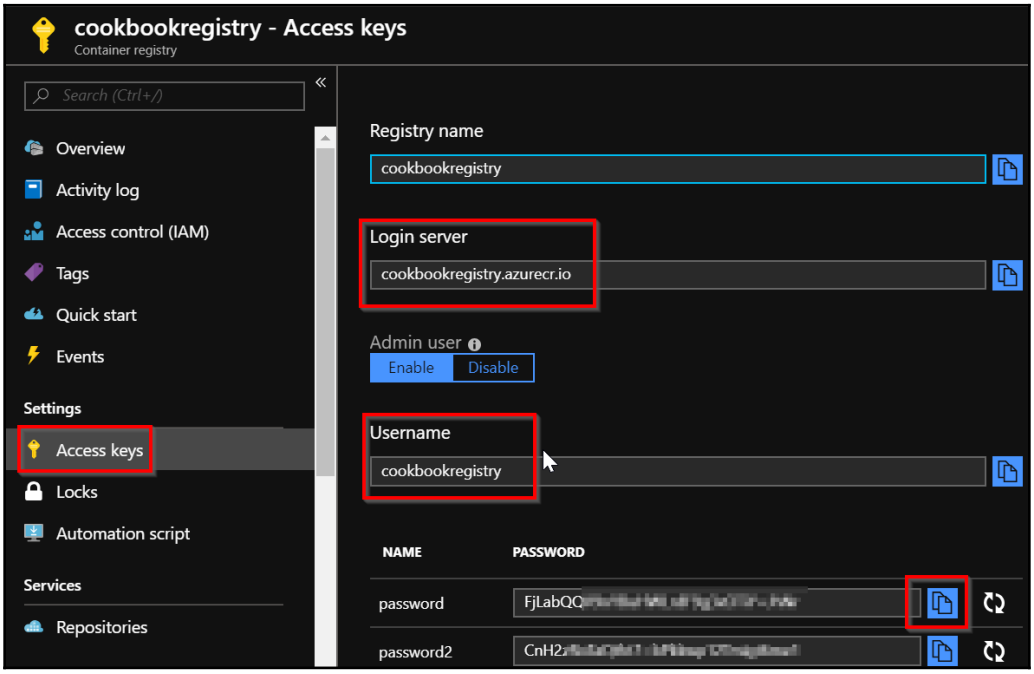
Creating an ACR

Perform the following steps:

1. Create a new ACR by providing the following details, as shown in the following screenshot:


Create container registry
* Registry name
cookbookregistry
.azurecr.io
* Subscription
Visual Studio Enterprise – MPN
* Resource group
AzureFunctionCookBooks
Create new
* Location
Central US
* Admin user
Enable
Disable
* SKU
Basic
Create
Automation options

2. Once the ACR is successfully created, navigate to the **Access Keys** blade and make a note of the **Login server**, **Username**, and **password**, which are highlighted in the following screenshot. You will be using them later in this recipe:



How to do it...

In the first three chapters, we created both the function app and the functions right within the portal. And, so far in this chapter, we have created the function app and the functions in the Visual Studio itself.

Before we start, let's make a small change to the `HTTPTrigger`, so that we understand that the code is running from Docker, as highlighted in the following image. To do this, I have just added a `From Docker` message to the output, as follows:

```
[FunctionName("HttpTriggerCSharpFromVS")]
0 references
public static IActionResult Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]HttpRequest req)
{
    //log.Info("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

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

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

Creating a Docker image for the function app

Perform the following steps:

1. The first step in creating a Docker image is to create a Dockerfile in our Visual Studio project. Create a `.Dockerfile` with the following contents:

```
FROM microsoft/azure-functions-dotnet-core2.0:2.0
 COPY ./bin/Release/netstandard2.0 /home/site/wwwroot
```

2. Then, navigate to Command Prompt and run the following Docker command, `docker build -t functionsindocker .`, taking care not to miss the period at the end of the command, to create a Docker image. Once you execute the `docker build` command, you should see something similar to that shown in the following screenshot:

```
C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>docker build -t functionsindocker .
Sending build context to Docker daemon  36.51MB
Step 1/2 : FROM microsoft/azure-functions-dotnet-core2.0:2.0
 ---> 35c818e033e2
Step 2/2 : COPY . .
 ---> b81847dc3c70
Successfully built b81847dc3c70
Successfully tagged functionsindocker:latest
```

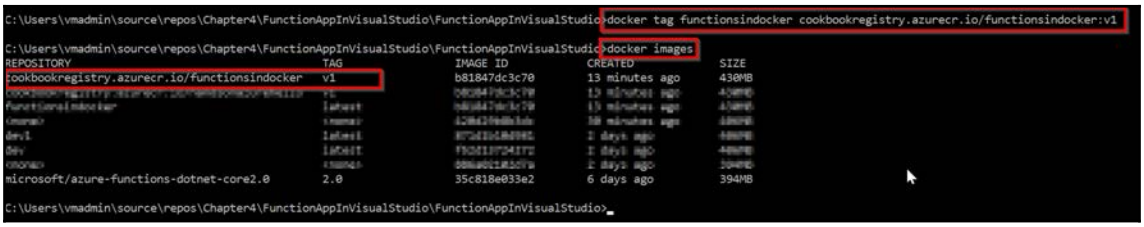
3. Once the image is successfully created, the next step is to run the Docker image on a specific port. Run the command to execute it. You should see something like the following screenshot:

4. Verify that everything is working fine in the local environment by navigating to the local host with the right port, as shown in the following screenshot.

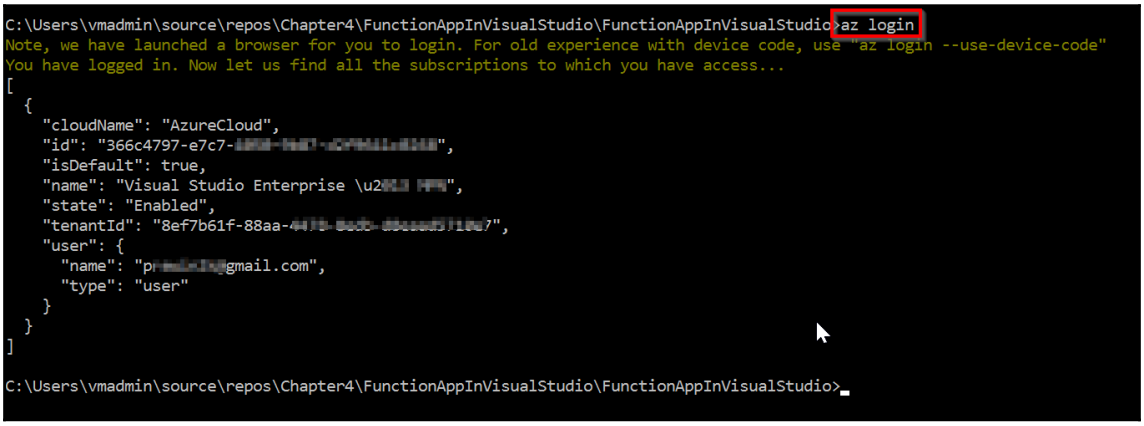
Pushing the Docker image to the ACR

Perform the following steps:

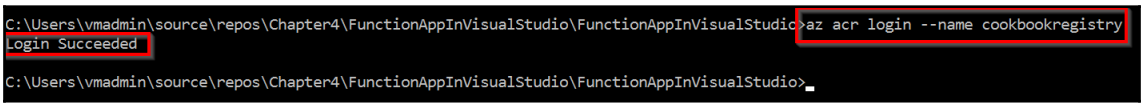
1. The first step is to ensure that we provide a valid tag to the image using the `docker tag functionsindocker cookbookregistry.azurecr.io/functionsindocker:v1` command. Running this command won't provide any output. However, to view our changes, let's run the `docker images` command, as shown in the following screenshot:



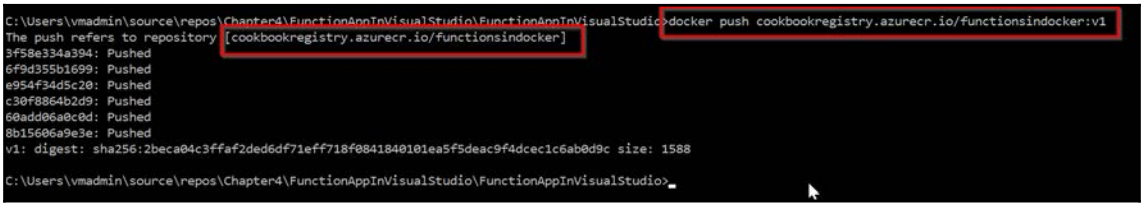
2. In order to push the image to ACR, you need to authenticate yourself to Azure. For this, you can use Azure CLI commands. Let's log in to Azure using the `az login` command. Running this command will open a browser and authenticate your credentials, as shown in the following screenshot:



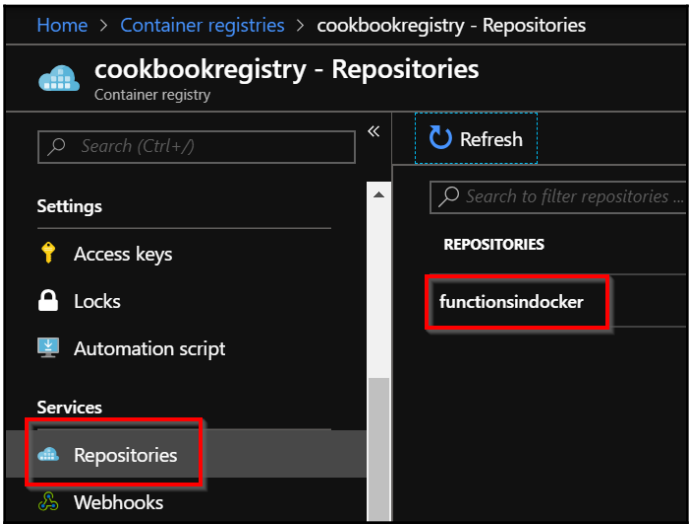
3. The next step is to authenticate yourself to the ACR using the `az acr login --name cookbookregistry` command. Replace the ACR name (in my case, it is `cookbookregistry`) with the one you have created:



4. Once you authenticate yourself, you can push the image to ACR by running the `docker push cookbookregistry.azurecr.io/functionsindocker:v1` command, as shown in the following screenshot:



5. Let's navigate to ACR in the Azure portal and review if our image was pushed to it properly in the **Repositories** blade, as shown in the following screenshot:

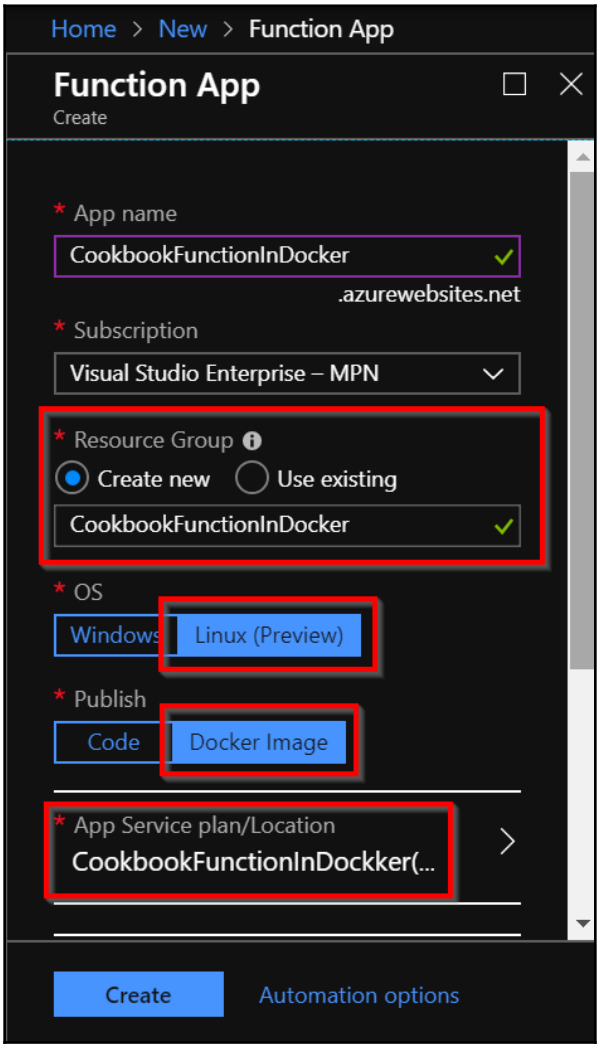


We have successfully created an image and pushed it to ACR. Now, it's time to create the Azure Function, and refer the Docker image that was pushed to the ACR.

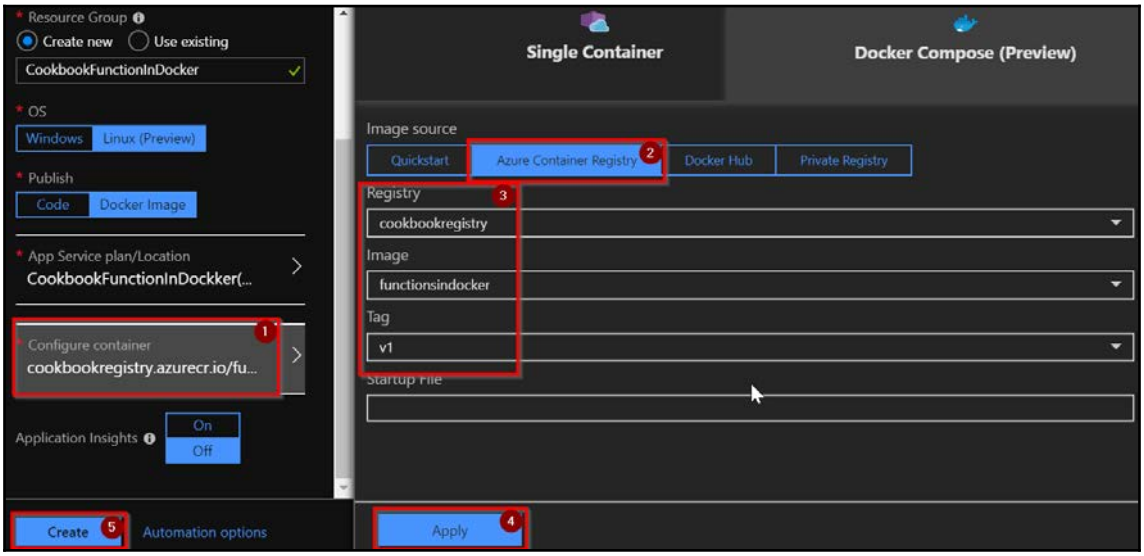
Creating a new function app with Docker

Perform the following steps:

1. Navigate to the **New** | **Function App** blade and provide the following information:



2. Now, choose **Linux (Preview)** in the **OS** field, and choose **Docker Image** in the **Publish** field, and then click on the **App Service Plan/Location**, as shown in the preceding screenshot. Here, choose to create a new **Basic** App Service plan.
3. The next and most important step is to refer the image that we have pushed to the ACR. This can be done by clicking on the **Configure container** button and choosing the **Azure Container Registry**, then choosing the correct image, as shown in the following screenshot:

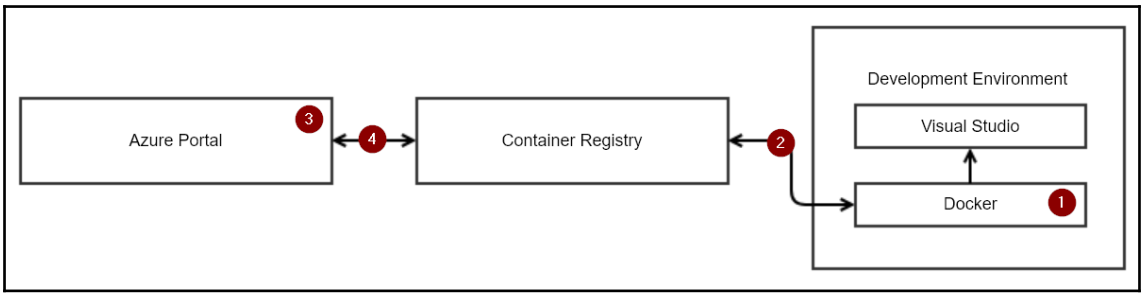


4. Once you review all the details, click on the **Create** button to create the function app.
5. That's it. We have created a Function App that could let us deploy the Docker images by linking it to the image hosted in the Azure Container Registry. Let's quickly test the `HttpTrigger` by navigating to the HTTP endpoint in the browser. The following is the output of the Azure Function.

How it works...

In the first three chapters, we created both the function app and the functions right within the portal. By contrast, so far in this chapter, we have created the function app and the functions in Visual Studio itself.

In this recipe, we have done the following:



The numbered points in this diagram refer to the following steps:

1. Create a Docker image of the function app that we created in this chapter using Visual Studio.
2. Push the Docker image to the ACR
3. From the portal, create a new function app, choosing to publish the executable package as a Docker image
4. Attach the Docker image from the ACR (from *step 2* in the preceding guide) to the Azure Function (from *step 3* in the preceding guide)

5
Exploring Testing Tools for the Validation of Azure Functions

In this chapter, we will explore different ways of testing the Azure Functions in more detail with the following recipes:

- Testing Azure Functions:
    - Testing HTTP triggers using Postman
    - Testing the Blob trigger using Microsoft Storage Explorer
    - Testing the Queue trigger using the Azure Management Portal
- Testing an Azure Function on a staged environment using deployment slots
- Load testing Azure Functions using Azure DevOps
- Creating and testing Azure Function locally using Azure CLI tools
- Testing and validating Azure Function responsiveness using Application Insights
- Developing unit tests for Azure Functions with HTTP triggers

Introduction

In the previous chapters, you learned how to develop Azure Functions and where they are useful, and looked at validating the functionality of those functions.

In this chapter, we will start looking at ways of testing different Azure Functions. This includes, for example, running tests of HTTP trigger functions using Postman and using Microsoft Storage Explorer to test Azure Blob triggers, Queue triggers, and other storage-service-related triggers. You will also learn how to perform a simple load test on an HTTP trigger, to help you understand how the serverless architecture works by provisioning the instances in the backend, without developers needing to worry about the scaling settings on different factors. The Azure Function runtime will automatically take care of scaling the instances.

You will also learn how to set up a test that checks the availability of our functions by continuously pinging the application endpoints on a predefined frequency from multiple locations.

Testing Azure Functions

The Azure Function runtime allows us to create and integrate many Azure services. At the time of writing, there are more than 20 types of Azure Functions you can create. In this recipe, you will learn how to test the most common Azure Functions, listed as follows:

- Testing HTTP triggers using Postman
- Testing the Blob trigger using Microsoft Storage Explorer
- Testing the Queue trigger using the Azure Management portal

Getting ready

Install the following tools, if you haven't already done so:

- **Postman**: You can download this from `https://www.getpostman.com/`
- **Microsoft Azure Storage Explorer**: You can download this from `http://storageexplorer.com/`

You can use Storage Explorer to connect to your storage accounts and view all the data available from different storage services, such as Blobs, Queues, Tables, and Files. You can also create, update, and delete them right from the Storage Explorer.

How to do it...

In this section, we will create three Azure Functions, using the default templates available in the Azure Management portal, and then test them with different tools.

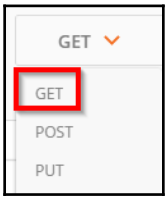
Testing HTTP triggers using Postman

Perform the following steps:

1. Create an HTTP trigger function that accepts the `Firstname` and `Lastname` parameters and sends them in the response. Once it is created, make sure you set **Authorization Level** as **Anonymous**.
2. Replace the default code with the following. Note that for the sake of simplicity, I have removed the validations. In real-time applications, you need to validate each and every input parameter:

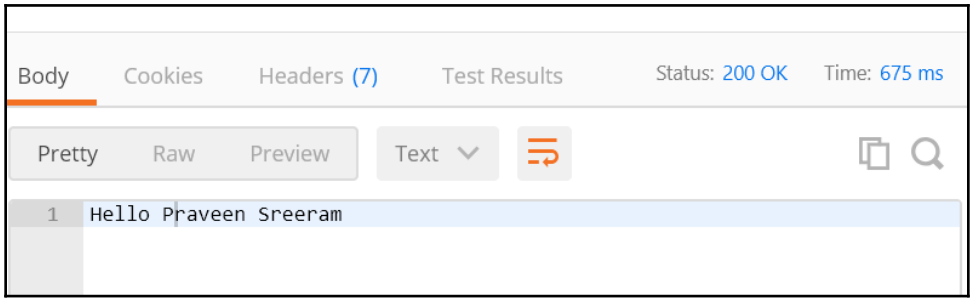
```
#r "Newtonsoft.Json"

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

public static async Task<IActionResult> Run(HttpRequest req,
ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a
request.");

    string firstname=req.Query["firstname"];
  string lastname=req.Query["lastname"];

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

        return (ActionResult)new OkObjectResult($"Hello, {firstname
+ " " + lastname}");
}
```

3. Open the Postman tool and complete the following:
    1. The first step is to choose the type of HTTP method with which you would like to make the HTTP request. As our function accepts most of the methods by default, choose the **GET** method, shown as follows:



    2. The next step is to provide the URL of the HTTP trigger. Note that you would need to replace `<HttpTriggerTestUsingPostman>` with your actual `HttpTrigger` function name, shown as follows:

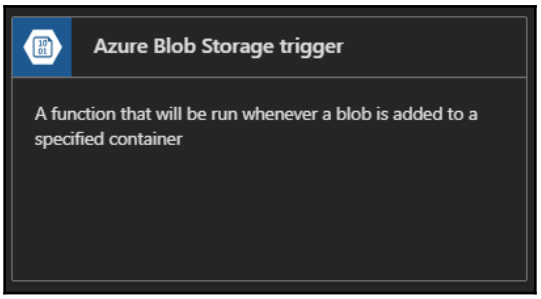
    3. Click on the **Send** button to make the request. If you have provided all the details expected by the API, then you would see a **Status: 200 OK**, along with the response, as shown here:



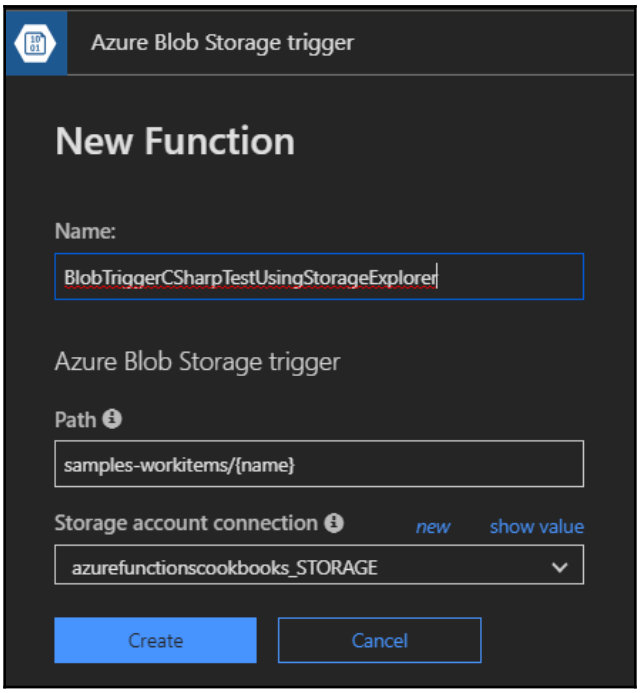
Testing a Blob trigger using Microsoft Storage Explorer

Perform the following steps:

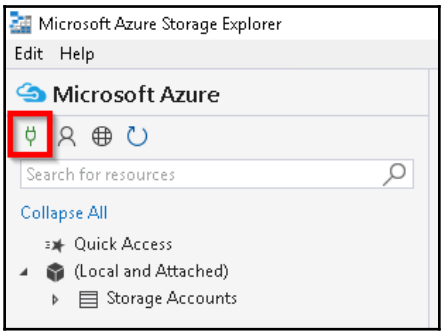
1. Create a new Blob trigger by choosing the **Azure Blob Storage trigger** template, as shown here:



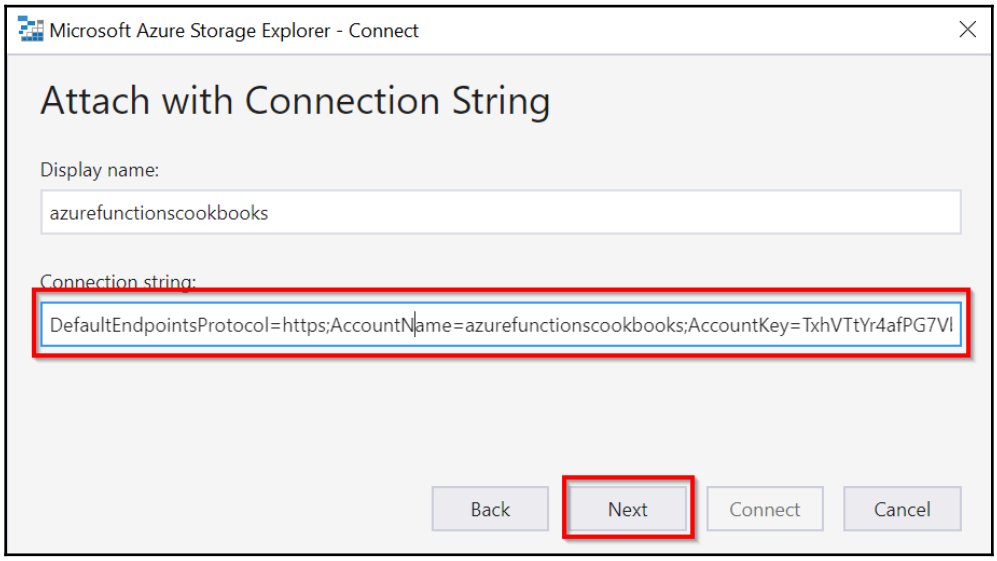
2. Once you click on the template in the previous screenshot, it will prompt you to provide a storage account and a container where you will store the Blob, shown as follows:



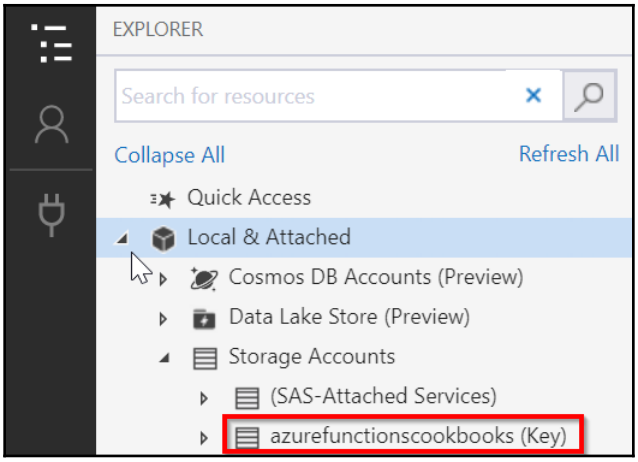
3. Let's connect to the storage account that we will be using in this recipe. Open **Microsoft Azure Storage Explorer** and click on the button that is highlighted in the following screenshot to connect to Azure Storage:



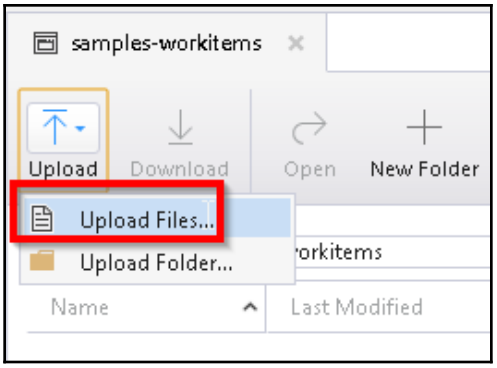
4. You will be prompted to enter various details, including storage connection string, **shared access signature** (**SAS**), and your **account key**. For this recipe, let's use the storage connection string. Navigate to **Storage Account**, copy the connection string in the **Access Keys** blade, and paste it in the **Microsoft Azure Storage Explorer - Connect** popup, shown as follows:



5. Clicking on the **Next** button in the preceding screenshot will take you to the **Connection Summary** window, displaying the account name and other related details for confirmation. Click on the **Connect** button to connect to the chosen Azure Storage account.
6. As shown in the following screenshot, you are now connected to the Azure Storage account, where you can manage all your Azure Storage services:



7. Now, let's create a storage container named `samples-workitems`. Right-click on the `Blob Containers` folder and click on **Create Blob Container** to create a new Blob container named `samples-workitems`. Then click on the **Upload files** button, as shown in the following screenshot:



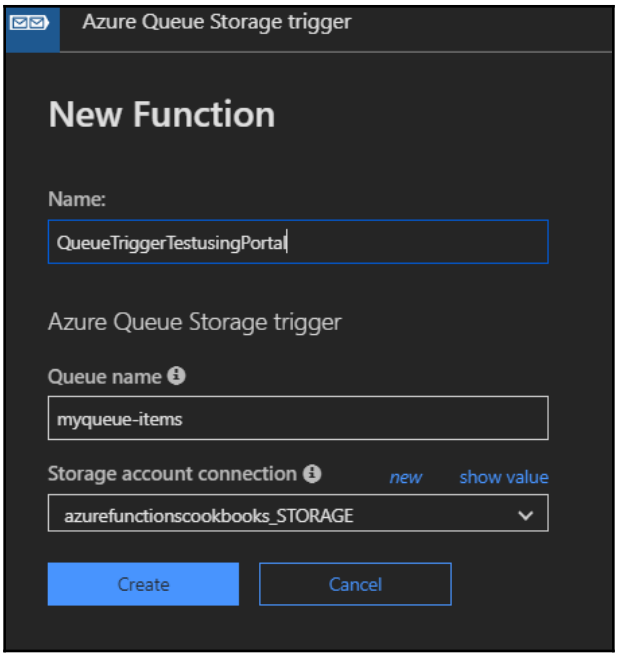
8. In the **Upload Files** window, choose a file that you would like to upload and then click on the **Upload** button.
9. Immediately navigate to the Azure Function code editor and look at the **Logs** window, as shown in the following screenshot. The log shows the Azure Function getting triggered successfully:

```
2018-09-27T02:52:35  Welcome, you are now connected to log-streaming service.
2018-09-27T02:53:35  No new trace in the past 1 min(s).
2018-09-27T02:54:35  No new trace in the past 2 min(s).
2018-09-27T02:55:35  No new trace in the past 3 min(s).
2018-09-27T02:56:35  No new trace in the past 4 min(s).
2018-09-27T02:57:35  No new trace in the past 5 min(s).
2018-09-27T02:58:35  No new trace in the past 6 min(s).
2018-09-27T02:59:15.577 [Info] Function started (Id=a07da295-103d-4bad-94b1-0113389bfffa)
2018-09-27T02:59:15.638 [Info] C# Blob trigger function Processed blob
 Name:1_BlobTemplate.png
 Size: 12254 Bytes
2018-09-27T02:59:15.638 [Info] Function completed (Success, Id=a07da295-103d-4bad-94b1-0113389bfffa, Duration=72ms)
```

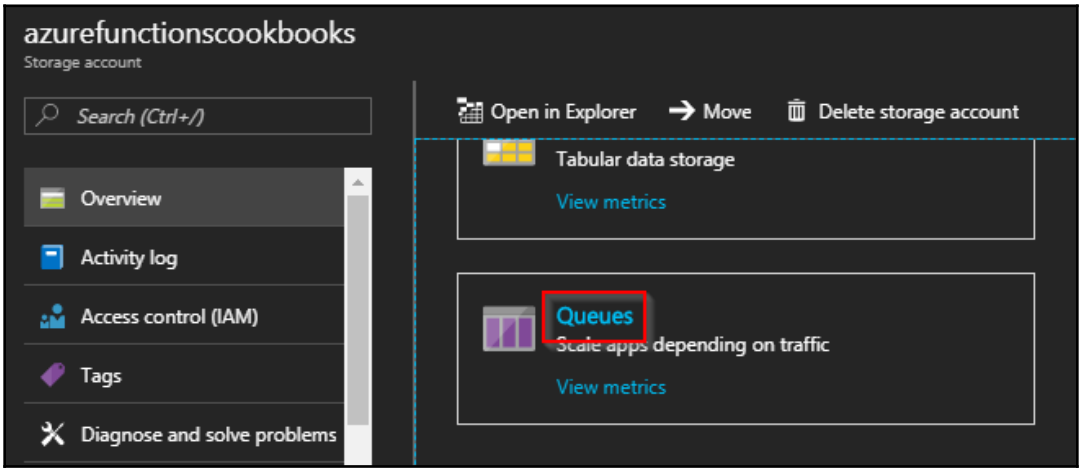
Testing the Queue trigger using the Azure Management portal

Perform the following steps:

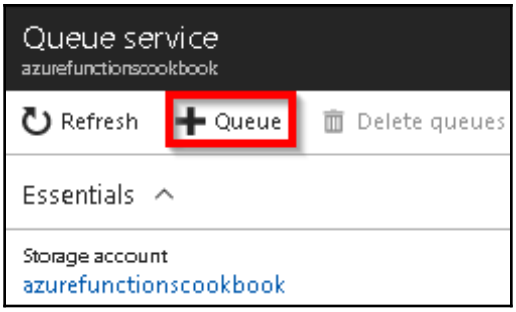
1. Create a new **Azure Storage Queue trigger** named `QueueTriggerTestusingPortal`, as shown in the following screenshot. Take note of the Queue name, `myqueue-items`, as we need to create a Queue service with the same name later using the Azure Management portal:



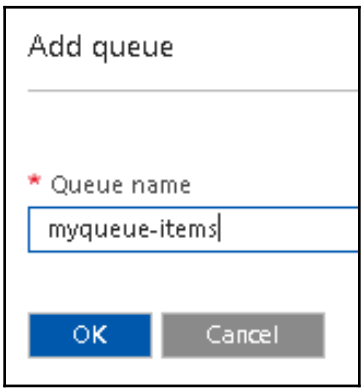
2. Navigate to the storage account's **Overview** blade and click on **Queues**, as shown in the following screenshot:



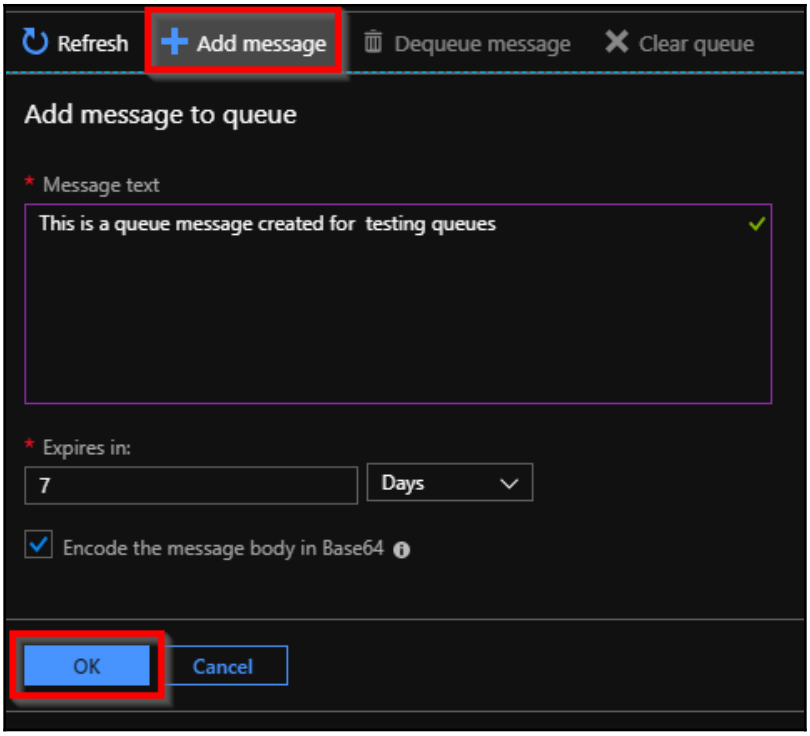
3. In the **Queue service** blade, click on **Queue** to add a new Queue:



4. Provide `myqueue-items` as the **Queue name** in the **Add queue** popup, as shown in the following screenshot. This was the same name we used while creating the Queue trigger. Click on **OK** to create the Queue service:



5. Now we need to create a Queue message. In the Azure Management portal, click on the `myqueue-items` Queue service to navigate to the **Messages** blade. Click on the **Add message** button, as shown in the following screenshot, and then provide a Queue message text. Lastly, click on **OK** to create the Queue message:

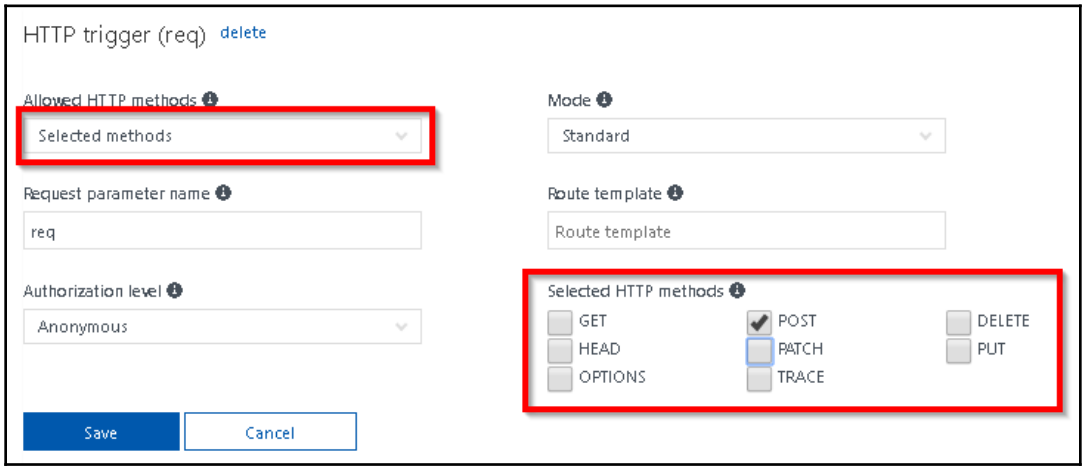


6. Immediately navigate to the `QueueTriggerTestusingPortal` Queue trigger, and view the **Logs** blade. Here, you can find out how the Queue function was triggered, as shown in the following screenshot:

```
2018-09-27T03:01:11  Welcome, you are now connected to log-streaming service.
2018-09-27T03:02:11  No new trace in the past 1 min(s).
2018-09-27T03:03:11  No new trace in the past 2 min(s).
2018-09-27T03:03:37.216 [Info] Function started (Id=935287b1-e661-4249-85ef-6010ab20459a)
2018-09-27T03:03:37.247 [Info] C# Queue trigger function processed: This is a queue message created for  testing
queues
2018-09-27T03:03:37.247 [Info] Function completed (Success, Id=935287b1-e661-4249-85ef-6010ab20459a, Duration=28ms)
```

There's more...

For all your HTTP triggers, if you would like to allow your API consumers to only use the `POST` method, then you can restrict it as such by choosing **Selected methods** and choosing only **POST** in **Selected HTTP methods**, as shown in the following screenshot:



Testing an Azure Function on a staged environment using deployment slots

In general, every application needs pre-production environments, such as staging, beta, and so on, in order to review functionalities before publishing them for the end users.

Though the pre-production environments are great and help multiple stakeholders validate the application's functionality against business requirements, there are some pain points in managing and maintaining them. The following are a few of them:

- We would need to create and use a separate environment for our pre-production environments
- Once everything is reviewed in pre-production and the IT Ops team gets the go-ahead, there would be a bit of downtime in the production environment while deploying the code base of new functionalities

All the preceding limitations can be covered in Azure Functions, using a feature called **slots** (these are called **deployment slots** in App Service environments). Using slots, you can set up a pre-production environment where you can review all the new functionalities and promote them (by swapping, which we will discuss in a moment) to the production environment seamlessly and whenever you need.

How to do it...

Perform the following steps:

1. Create a new function app named `MyProductionApp`.
2. Create a new HTTP trigger and name it `MyProd-HttpTrigger1`. Replace the last line with the following:

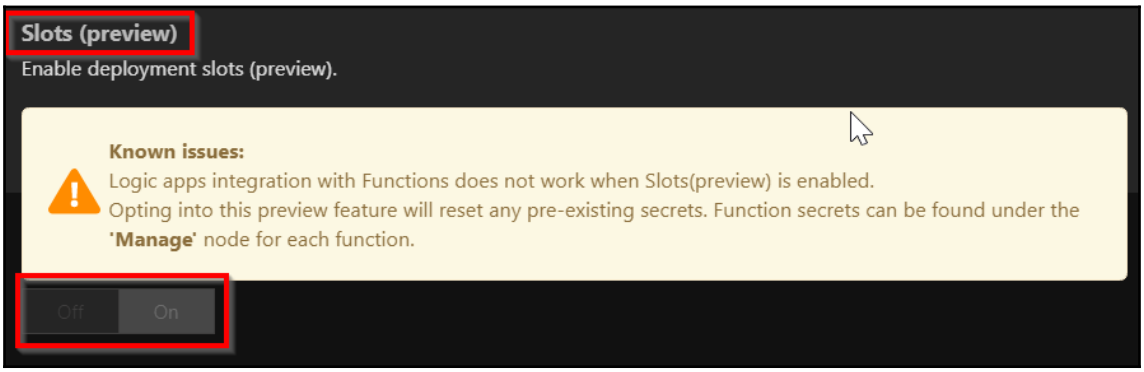
```
         return name != null
        ? (ActionResult)new OkObjectResult("Welcome to MyProd-HttpTrigger1
of Production App")
        : new BadRequestObjectResult("Please pass a name on the query
string or in the request body");
```

3. Create another new HTTP trigger and name it `MyProd-HttpTrigger2`. Use the same code that you used for `MyProd-HttpTrigger1`—just replace the last line with the following:

```
return name != null
        ? (ActionResult)new OkObjectResult("Welcome to MyProd-
HttpTrigger2 of Production App")
        : new BadRequestObjectResult("Please pass a name on the
query string or in the request body");
```

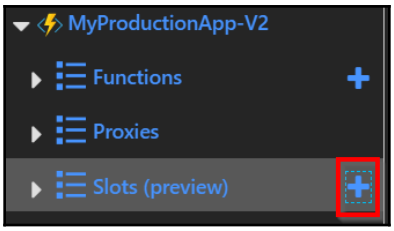
4. Assume that both the functions of the function app are live on your production environment with the URL `https://<<functionappname.azurewebsites.net>>`.
5. Now, the customer has requested us to make some changes to both functions. Instead of directly making the changes to the functions of your production function app, you might need to create a slot.

6. Hold on! Before you can create a slot, you first need to enable the feature by navigating to the **Function app settings** under the **General Settings** of the **Platform features** tab of the function app. Once you click on the **Function app settings**, a new tab will be opened where you can enable the **Slots (preview)**, as shown in the following screenshot:

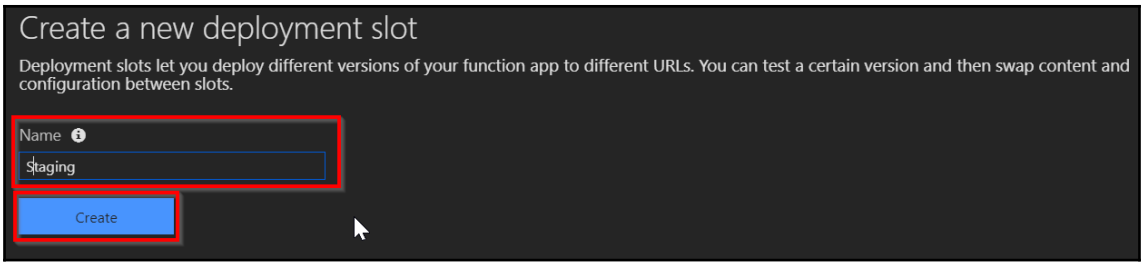


The slots feature is currently in preview. By the time you read this, it might be **Generally Available** (**GA**). It is not recommended to use this feature in production workloads until it goes GA

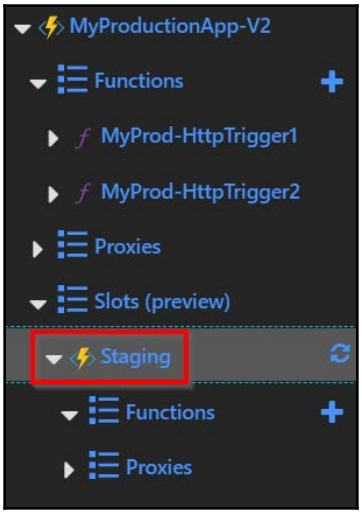
7. Click the **On** button in the **Slots (preview)** section highlighted in the preceding screenshot. As soon as you turn it on, the slots section will be hidden, as it is a one-time setting. Once it's enabled, you cannot disable it.
8. OK, let's create a new slot with all the functions that we have in our function app, named `MyProductionApp`.
9. Click on the **+** icon, available near the **Slots (preview)** section, as shown in the following screenshot:



10. It prompts you to enter a name for the new slot. Provide a meaningful name, something such as `Staging`, as shown in the following screenshot:



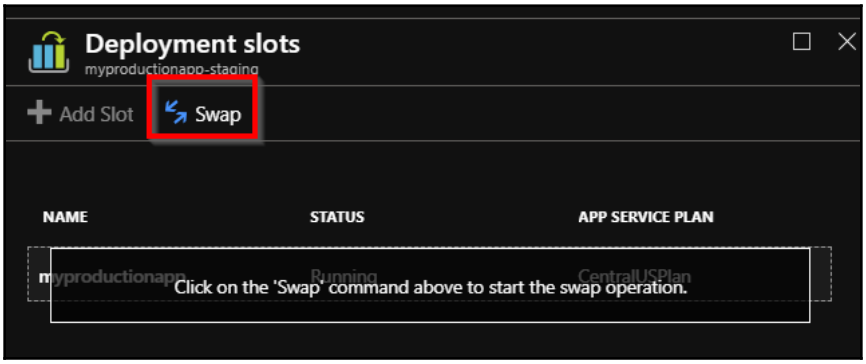
11. Once you click on **Create**, a new slot will be created, as shown in the following screenshot. In case, if you see the Functions as read only, you can make them read-write in the **Function App Settings**.



The URL for the slot will be `https://<<functionappname>>-<<Slotname>>.azurewebsites.net>>`. Each slot within a function app will have a different URL.

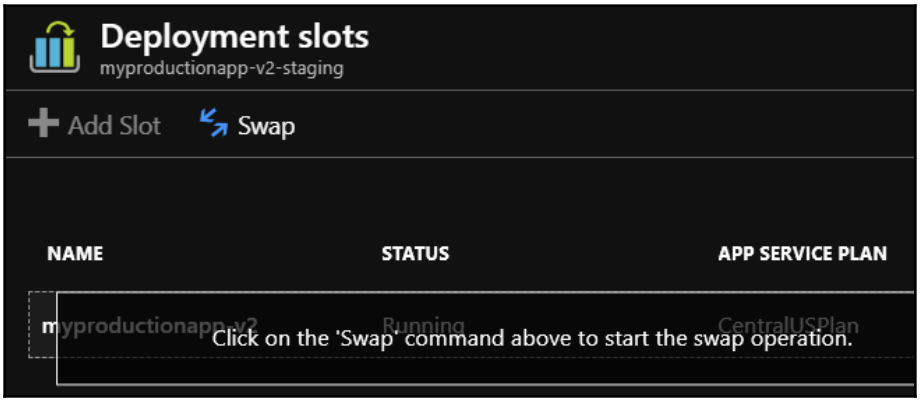
12. To make a staged environment complete, you need to copy all the Azure Functions from the production environment (in this case, the `MyProductionApp` app) to the new staged slot named `Staging`. Create two HTTP triggers and copy both the functions' code (`MyProd-HttpTrigger1` and `MyProd-HttpTrigger2`) from `MyProductionApp` to the new `Staging` slot. Basically, you need to copy all the functions to the new slot manually.

13. Change the `production` string to `staging` in the last line of both the functions in the **Staging** slot. This is useful for testing the output of the swap operation:

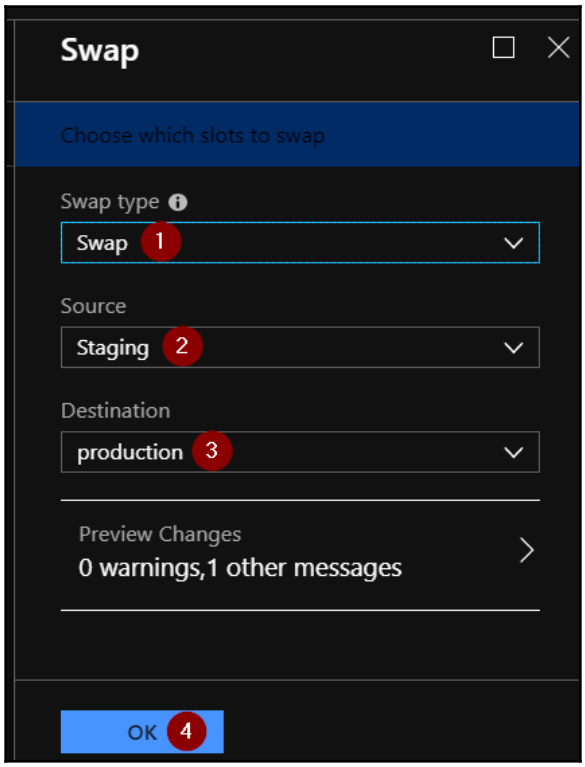


Note that, in all the slots that you create as a pre-production app, you need to make sure that you use the same function names as you have in your production environment.

14. Click on the **Swap** button, available in the **Deployment slots** blade, as shown in the following screenshot:

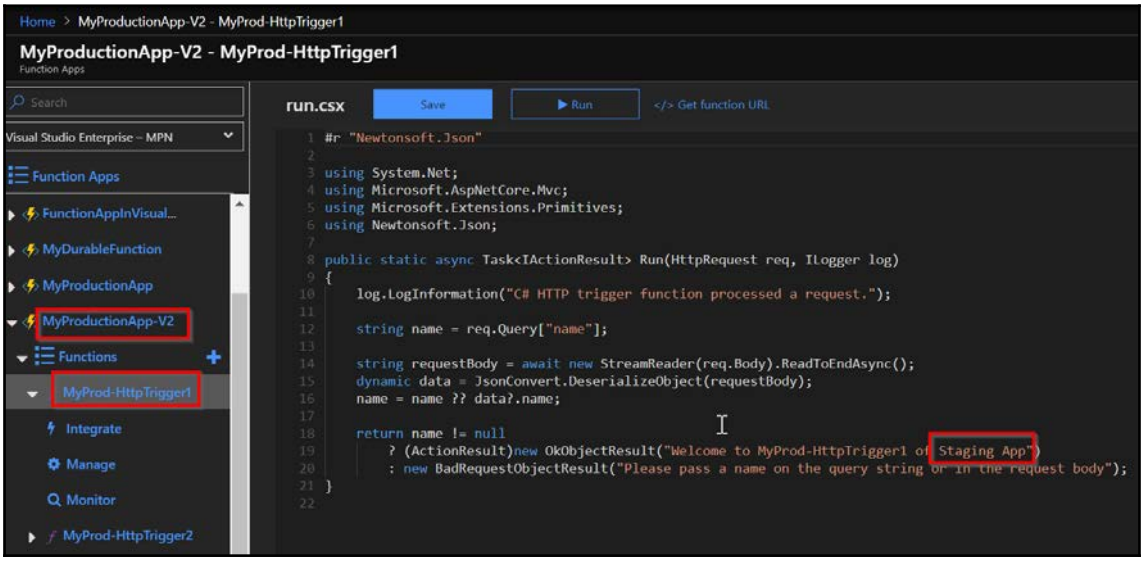


15. In the **Swap** blade, you need to choose the following:
    - **Swap Type**: Choose the **Swap** option.
    - **Source**: Choose the slot that you would like to move to production. In this case, we're swapping **Staging** in general, but you can even swap across non-production slots.
    - **Destination**: Choose the **production** option, as shown in the following screenshot:

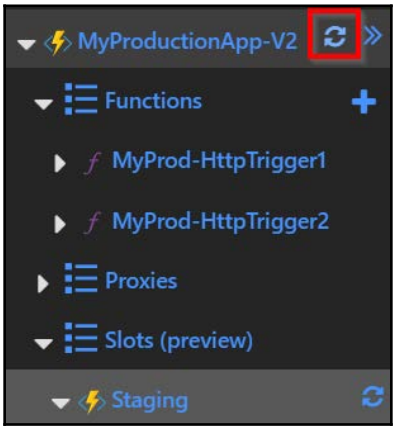


16. Once you review the settings, click on the **OK** button of the preceding step. It will take a few moments to swap the functions. A progress bar will appear, as shown in the following screenshot:

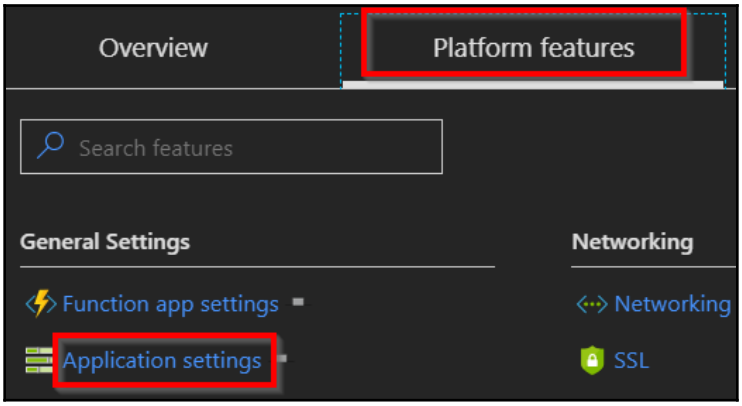
17. After a minute or two, the staging and production slots get swapped. Let's review the `run.csx` script files of the production:



18. If you don't see any changes, click on the refresh button of the function app, as shown in the following screenshot:



19. Make sure that the **Application settings** and **Database Connection Strings** are marked as **Slot Setting** (slot-specific). Otherwise, **Application settings** and **Database Connection Strings** will also get swapped, which could cause unexpected behavior. You can mark any of these settings as such from **Platform features**, as shown in the following screenshot:



20. Clicking on the **Application settings** will take you to the following blade, where you can mark any setting as a **SLOT SETTING**:

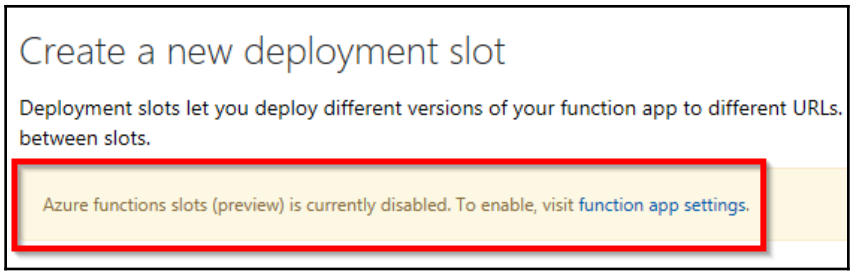
All the functions taken in the recipe are HTTP triggers; note that you can have any kind of triggers in the function app. The deployment slots are not limited to HTTP triggers.

You can have multiple slots for each of your function apps. The following are a few of the examples:

- Alpha
- Beta
- Staging

There's more...

If you try to create a slot without enabling the feature of Deployment Slots, you will see something similar to what is shown in the following screenshot:



You need to have all the Azure Functions in each of the slots that you would like to swap with your production function app:

- Slots are specific to the function app, but not to the individual function.
- Once you enable the slots features, all the keys will be regenerated, including the master. Be cautious if you have already shared the keys of the functions with third parties. If you had already shared them and enabled the slots, all the existing integrations with the old keys will not work.

In general, if you are using App Services and would like to create deployment slots, you need to have your App Service plan in either one of the Standard or Premium tiers. However, you can create slots for the function app even if it is under **Consumption** (or dynamic) plans.

Load testing Azure Functions using Azure DevOps

Every application needs to perform well in terms of performance. It's everyone's responsibility within the team that the application is performing well. In this recipe, you will learn how to create a load on the Azure Functions using the **Load Test** tool provided by **Azure DevOps** (formerly known as VSTS). This recipe will also help you understand how the auto-scaling of instances works in the serverless environment, without the developers or architect needing to worry about the instances that are responsible for serving the requests.

Getting ready

Create an Azure DevOps account at `https://visualstudio.microsoft.com/`. We will be using the **Load Test** tool of Azure DevOps to create URL-based load testing.

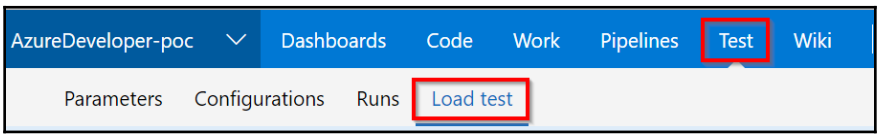
How to do it...

Perform the following steps:

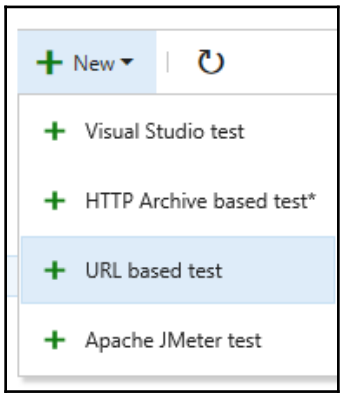
1. Create a new HTTP trigger, named `LoadTestHttpTrigger`, with **Authorization Level** set to **Anonymous**.
2. Replace the default code in `run.csx` with the following:

```
using System.Net;
using Microsoft.AspNetCore.Mvc;
public static async Task<IActionResult> Run(HttpRequest req,
ILogger log)
{
 System.Threading.Thread.Sleep(2000);
 return (ActionResult)new OkObjectResult($"Hello");
}
```

3. The preceding code is self-explanatory. In order to make the load test interesting, let's simulate some processing load by adding a wait time of two seconds, using `System.Threading.Thread.Sleep(2000);`.
4. Copy the function URL by clicking on the **</> Get function URL** link on the right-hand side of the `run.csx` code editor.
5. Navigate to the **Load test** tab of the Azure DevOps account. You can find it under the **Test** menu after you log in to Azure DevOps:

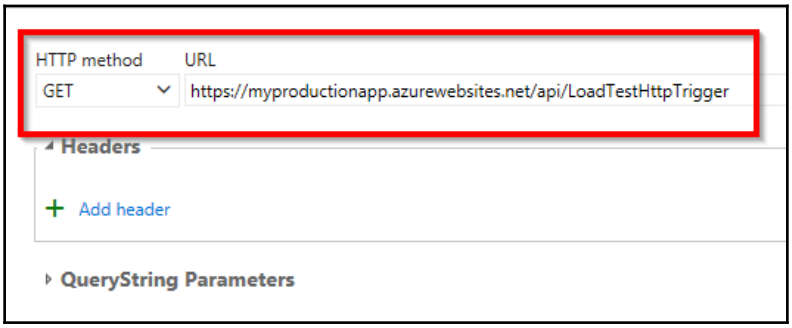


6. Click on the **New** link and select **URL based test**, as shown in the following screenshot:

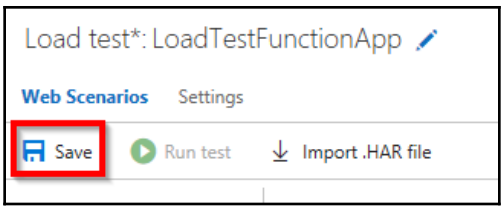


7. In the **Web Scenarios** tab, provide a meaningful name for the load test, as shown in the following screenshot:

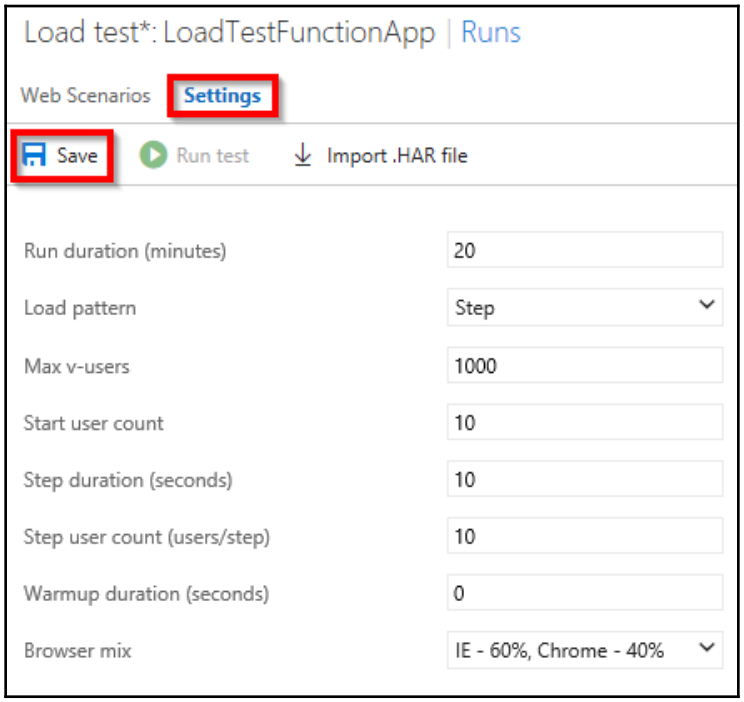
8. Paste the HTTP trigger URL that you copied in *step 4* into the **URL** input field, as shown in the following screenshot:



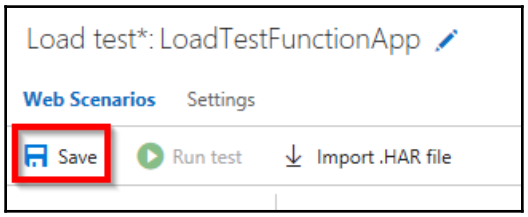
9. Now, click on the **Save** button to save the load test:



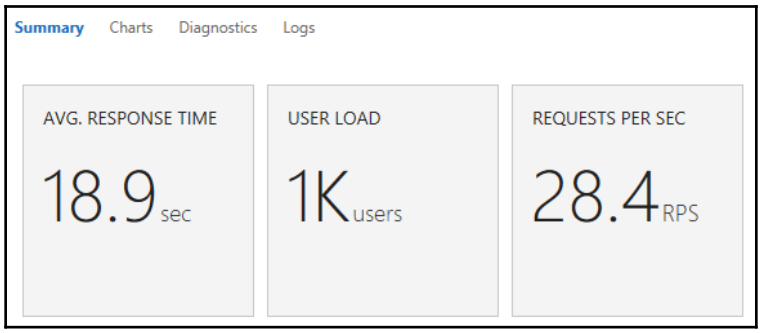
10. The next step is to provide details about the load that we would like to create on the Azure Function. As shown in the following screenshot, click on **Settings** and provide the details about the load test that you would like, depending on your requirements:



11. Once you provide all your details for the load test, click on **Save**. Once you save the test, the **Run test** button will be enabled, as shown in the following screenshot:



12. Click on **Run test** to start the load test. As the run duration of our load test is 20 minutes, it would take 20 minutes to complete the load test. Once the load is complete, Azure DevOps provides us with the performance reports, shown as follows:
    - **Summary report**: This provides us the average response time of the HTTP trigger for the load of **1K users**.



There's more...

We can also look at how Azure scales out the instances automatically behind the scenes in the **Live Metrics Stream** tab of **Application Insights**. The following screenshot shows the instance IDs and the health of the virtual machines that are allocated automatically, based on the load on the Azure serverless architecture. You will learn how to integrate Application Insights with Azure Functions in `Chapter 6`, *Monitoring and Troubleshooting Azure Serverless Services*

See also

The *Monitoring Azure Functions using Application Insights* recipe in `Chapter 6`, *Monitoring and Troubleshooting Azure Serverless Services*, contains more information on this topic.

Creating and testing Azure Functions locally using Azure CLI tools

Most of the recipes that you have learned so far have been created using either the browser or Visual Studio **Integrated Development Environment** (**IDE**).

Azure also provides us with tools for developers that love working with the command line. These tools allow us to create Azure resources with simple commands right from the command line. In this recipe, you will learn how to create a new function app, and also understand how to create a function and deploy it to the Azure Cloud right from the command line.

Getting ready

Perform the following steps:

1. Download and install Node.js from `https://nodejs.org/en/download/`.
2. Download and install Azure CLI tools from `https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest`.

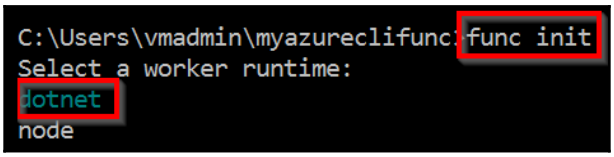
How to do it...

Perform the following steps:

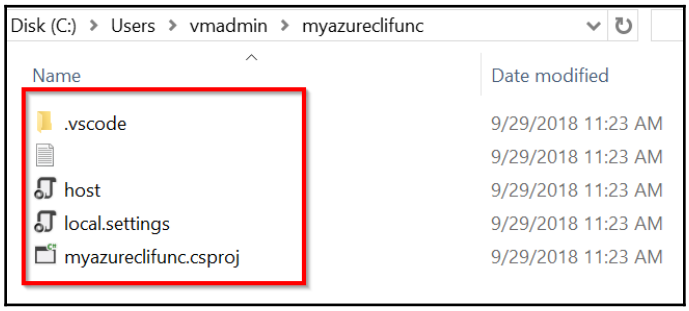
1. Once the Azure Functions Core Tools are ready, run the following command to create a new function app:

```
func init
```

You will get the following output after executing the preceding command:



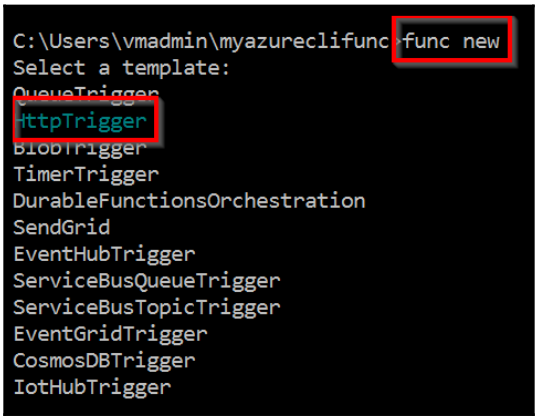
In the preceding screenshot, **dotnet** is selected by default. Pressing *Enter* will create the required files, as shown in the following screenshot:



2. Run the following command to create a new HTTP trigger function within the new function app that we have created:

```
func new
```

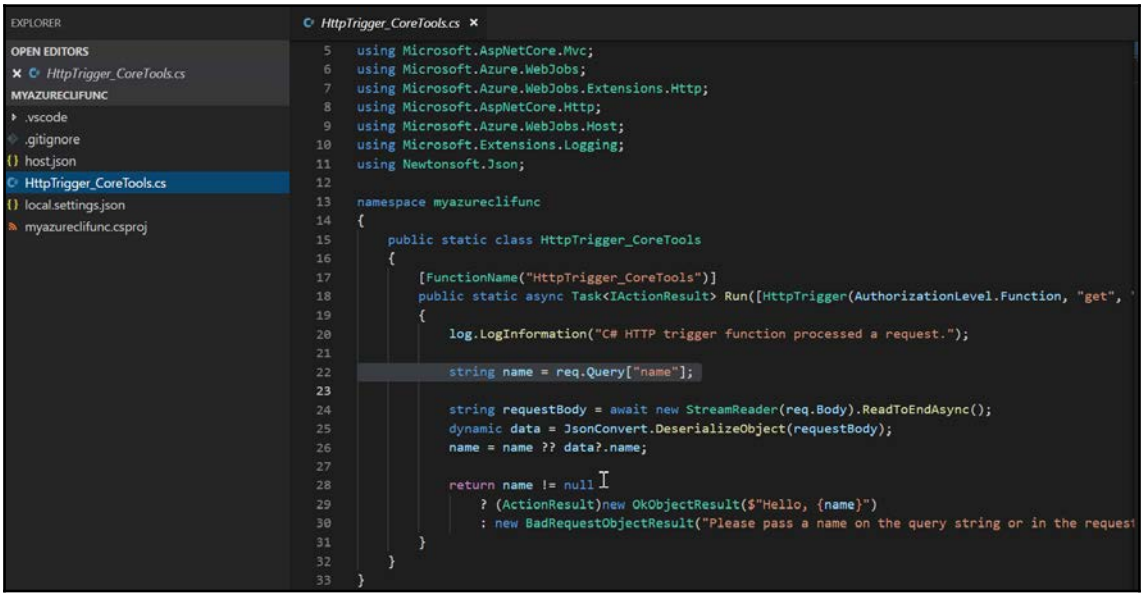
You will get the following output after executing the preceding command:



3. As shown in the preceding screenshot, you will be prompted to choose the function template. For this recipe, I have chosen `HttpTrigger`. Choose `HttpTrigger` by using the down arrow. You can choose the Azure Function type based on your requirements. You can navigate between the options using the up/down arrows on your keyboard.
4. The next step is to provide a name for the Azure Function that you are creating. Provide a meaningful name and press *Enter*, as shown in the following screenshot:

```
C:\Users\vmadmin\myazureclifunc>func new
Select a template: Function name: HttpTrigger-CoreTools
HttpTrigger-CoreTools

The function "HttpTrigger-CoreTools" was created successfully from the "HttpTrigger" template.

C:\Users\vmadmin\myazureclifunc>
```

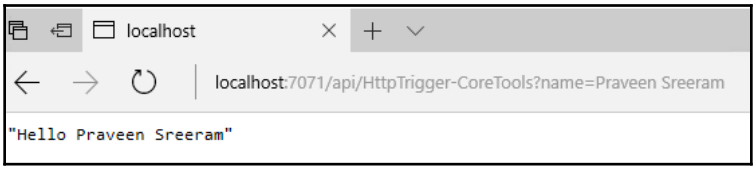
5. You can use your favorite IDE to edit the Azure Function code. In this recipe, I am using Visual Studio Code to open the `HttpTrigger` function, as shown in the following screenshot:



6. Let's test the Azure Function right from your local machine. For this, we need to start the Azure Function host by running the following command:

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

7. Once the host is started, you can copy the URL and test it in your browser, along with a query string parameter name, as shown in the following screenshot:



Testing and validating Azure Function responsiveness using Application Insights

Any application is only useful for any business if it is up and running. Applications might go down for multiple reasons; the following are a few of them:

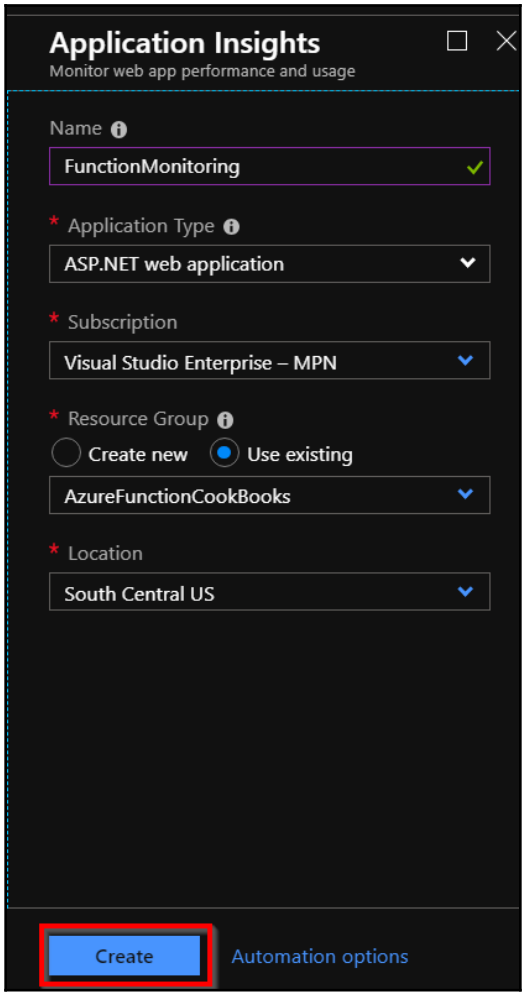
- Any hardware failures, such as a server crash, bad hard disk, or any other hardware issue—even an entire data center might go down, although this would be very rare
- There might be software errors because of bad code or a deployment error
- The site might receive unexpected traffic and the servers may not be capable of handling this traffic
- There might be cases where your application is accessible from one country, but not from others

It would be really helpful to get a notification when our site is not available or not responding to user requests. Azure provides a few tools to help by alerting us if the website is not responding or is down. One of them is Application Insights. You will learn how to configure Application Insights to ping our Azure Function app every minute, and set it to alert us if the function is not responding.

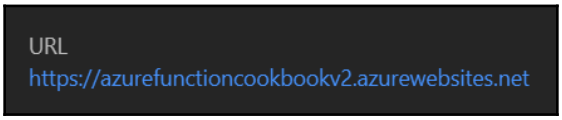
Getting ready

Perform the following steps:

1. Navigate to the Azure Management portal, search for **Application Insights**, then click on the **Create** button, and provide all the required details, as shown in the following screenshot:



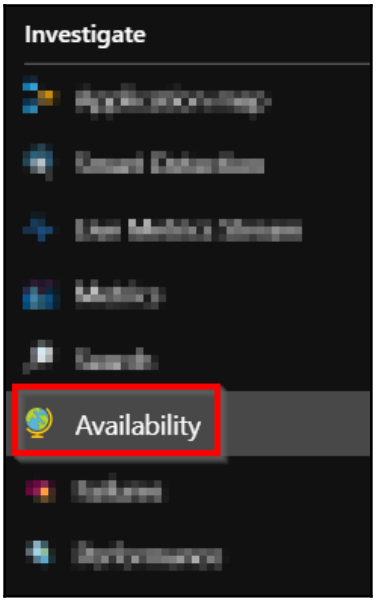
2. Navigate to your function app's **Overview** blade and grab the function app **URL**, as shown in the following screenshot:



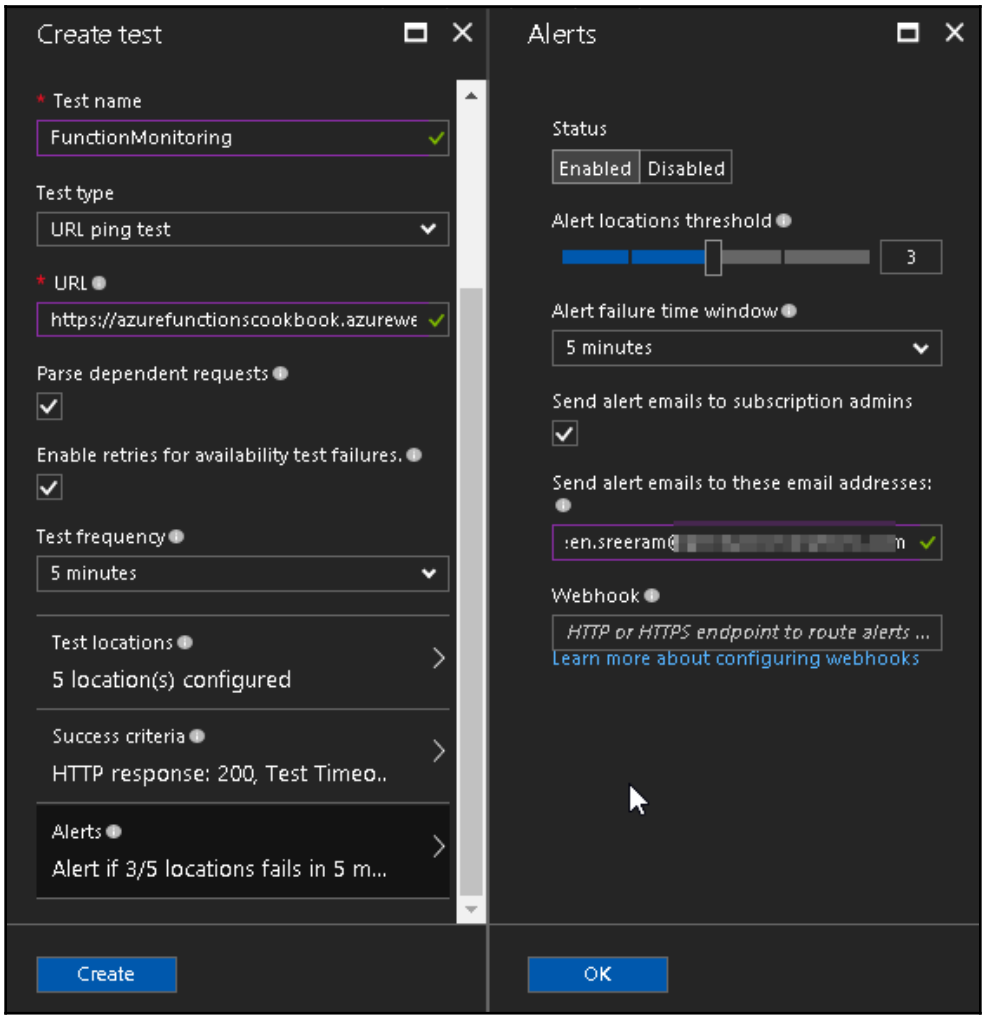
How to do it...

Perform the following steps:

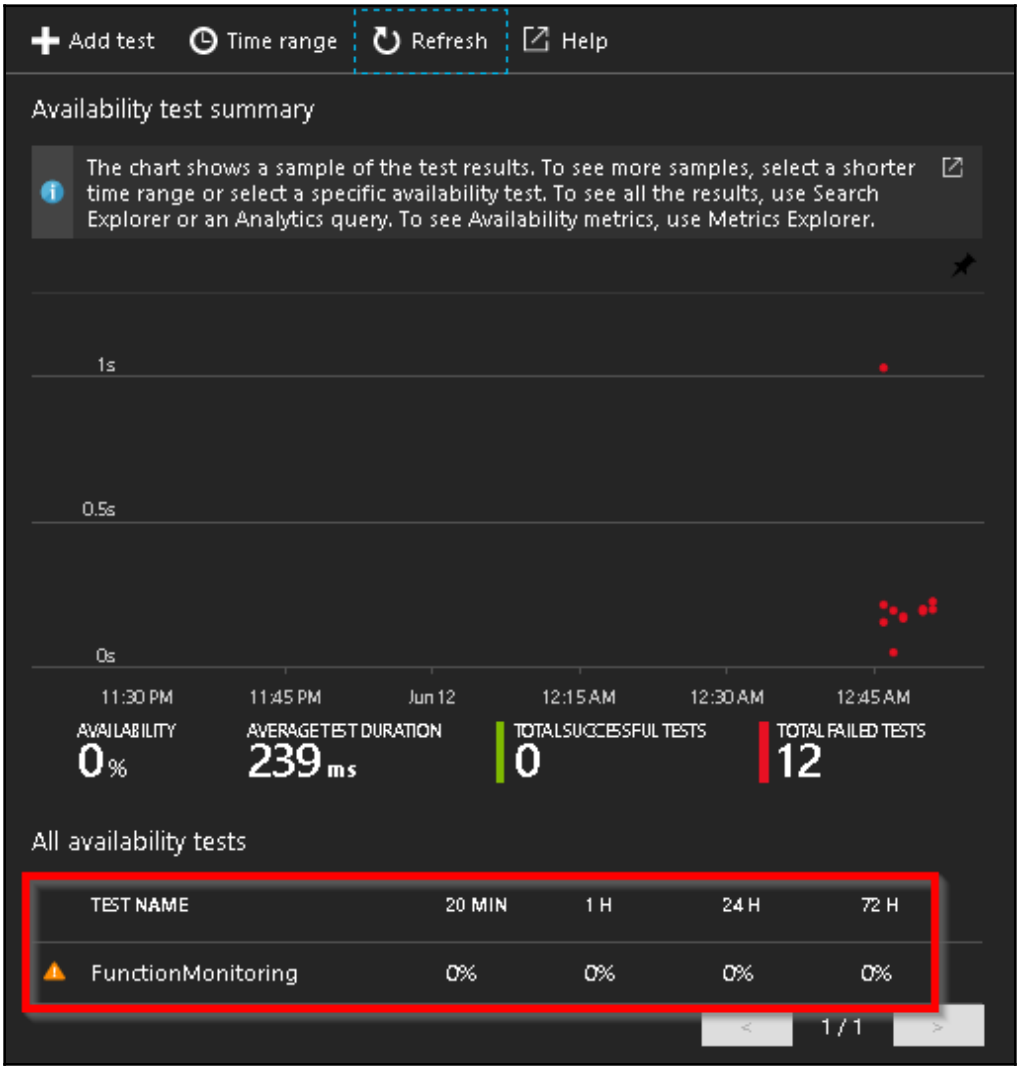
1. Navigate to the **Availability** blade and click on the **Add test** button, as shown in the following screenshot:



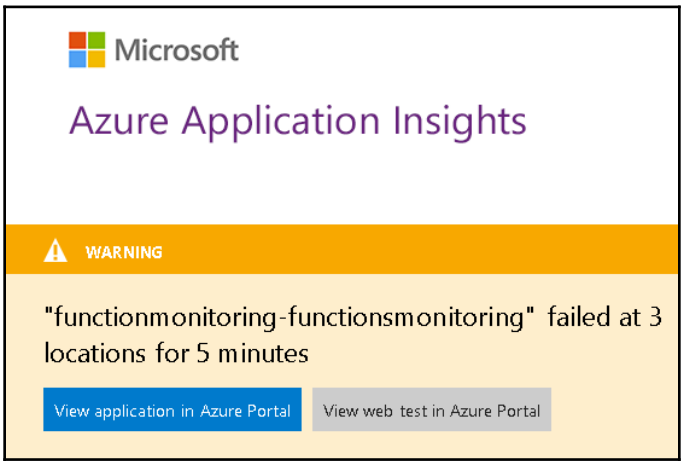
2. In the **Create test** blade, enter a meaningful name for your requirement and paste the function app URL, which you noted down in *step 2* of the preceding *Getting ready* section, in the **URL** field of the **Create test** blade. In the **Alerts** blade, provide a valid email address in the **Send alert emails to these email addresses:** field, to which an alert should be sent if the function is not available or not responding:



3. Click on **OK** in the **Alerts** blade, and then click on the **Create** button of the **Create test** blade to create the test, as shown in the following screenshot, in the **All availability tests** section:



4. In order to test the functionality of this alert, let's stop the function app by clicking on the **Stop** button, found in the **Overview** tab of the function app.
5. When the function app was stopped, Application Insights will try to access the function URL using the ping test. The response code will not be `200`, as the app was stopped, which means the test failed and a notification should have been sent to the configured email, as shown in the following screenshot:



How it works...

We have created an `Availability` test, where our function app will be pinged once every five minutes from a maximum of five different locations across the world. You can configure them in the **Test Location** tab of the **Create test** blade while creating the test. The default criterion of the ping is to check if the response code of the URL is `200`. If the response code is not `200`, then the test has failed, and an alert is sent to the configurable email address.

There's more...

You can use a multi-step web test (using the **Test Type** option in the **Create test** blade) if you would like to test a page or functionality that requires navigating to multiple pages.

Developing unit tests for Azure Functions with HTTP triggers

So far, we have created multiple Azure Functions and validated their functionality using different tools. The functionalities of the functions that we have developed so far is pretty simple and straightforward; however, in your real-world applications, it won't be that simple—there will likely be many changes to the code that we initially created. It's good practice to write automated unit tests that can help us in testing the functionality of our Azure Functions. Every time we run these automated unit tests, we can test all the various paths within the code.

In this recipe, we will learn how to use the basic HTTP trigger, and see how easy it is to write automated unit test cases for this using Visual Studio Test Explorer and Moq (an open source framework available as a NuGet package).

Getting ready

We will be using the Moq mocking framework to unit test our Azure Function. Having a basic working knowledge of Moq is a requirement for this recipe. If you need to, you can learn more about Moq at `https://github.com/moq/moq4/wiki`.

In order to make the Unit Test Case simple, I have commented out the lines of code that reads the data from the Post parameters to the Run method of **HTTPTriggerCSharpFromVS** HTTPTrigger as shown below with bold highlighted.

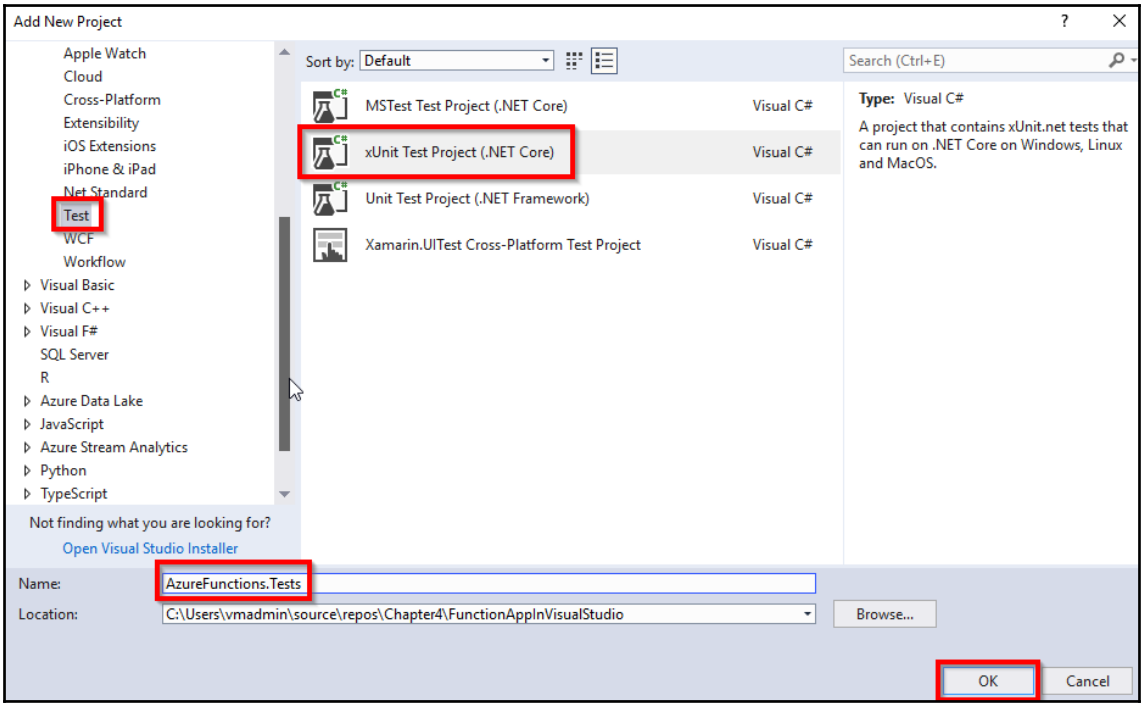
```
[FunctionName("HTTPTriggerCSharpFromVS")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route
= null)] HttpRequest req,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a
request.");

            string name = req.Query["name"];

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

            return name != null
```

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

How to do it...

Perform the following steps:

1. Create a new unit testing project by right-clicking on the solution and then clicking on **Add New Project**. In the **Add New Project** window, choose **Test** in the list of **Project Type** and choose **xUnit Test Project(.NET Core)** in the list of projects, as shown here:



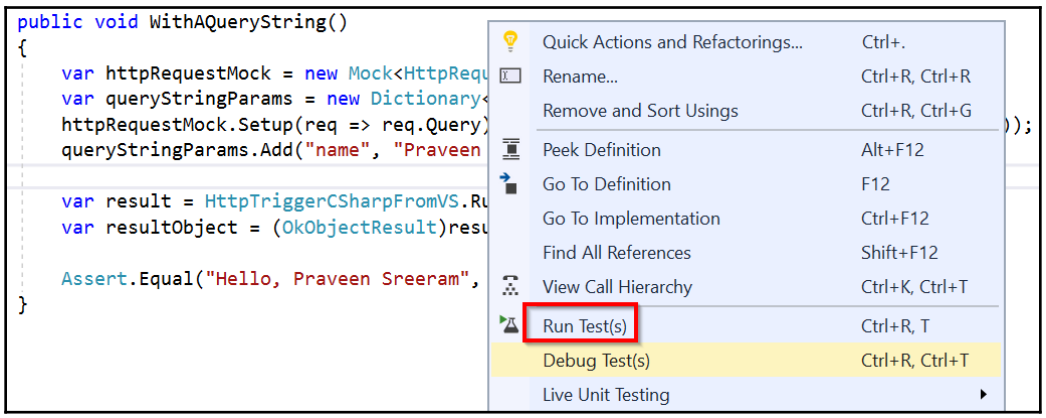
2. Ensure that you have chosen the **xUnit Test Project(.NET Core)** in the `Package Manager` console and run the following commands:
    - Install the Moq NuGet package using the `Install-Package Moq` command
    - Install the ASP.NET Core package using the `Install-Package Microsoft.AspNetCore` command

3. In the unit test project, we also need the reference to the Azure Function that we want to run the unit tests. Add a reference to the `FunctionAppInVisualStudio` application so that we can call the HTTP trigger's `Run` method from our unit tests.
4. Add all the required namespaces to the `Unit Test` class, as follows: and replace the default code with the following code. The following code mocks the requests, creates a Query string collection with a key named `name`, assigns a value of `Praveen Sreeram`, executes the function, gets the response, and then compares the response value with an expected value:

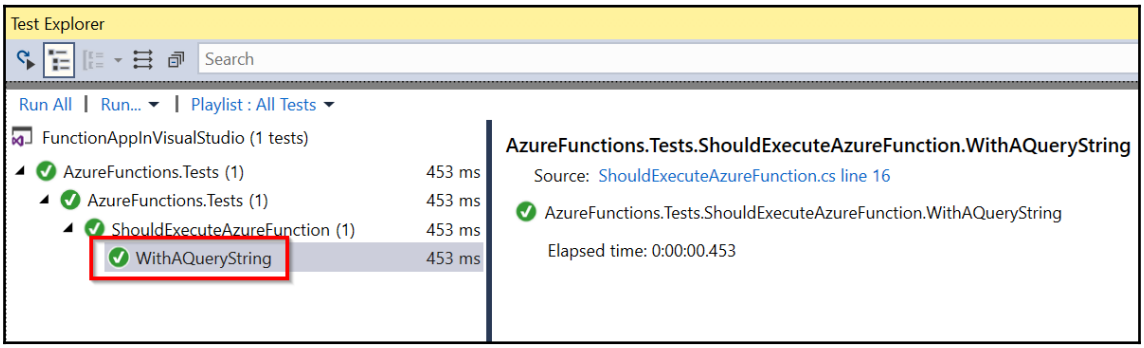
```
using FunctionAppInVisualStudio;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http.Internal;
using Microsoft.Extensions.Primitives;
using Moq;
using System;
using System.Collections.Generic;
using Xunit;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace AzureFunctions.Tests
{
    public class ShouldExecuteAzureFunctions
    {
        [Fact]
        public async Task WithAQueryString()
        {
            var httpRequestMock = new Mock<HttpRequest>();
            var LogMock = new Mock<ILogger>();
            var queryStringParams = new Dictionary<String,
StringValues>();
            httpRequestMock.Setup(req => req.Query).Returns(new
QueryCollection(queryStringParams));
            queryStringParams.Add("name", "Praveen Sreeram");
            var result = await
```

```
HTTPTriggerCSharpFromVS.Run(httpRequestMock.Object,LogMock.Object);
            var resultObject = (OkObjectResult)result;
            Assert.Equal("Hello, Praveen Sreeram",
resultObject.Value);
        }
    }
}
```

5. Now, right-click on the unit test and click on **Run test(s)**, as shown in the following screenshot:



If everything is set up correctly, your tests should pass, as shown in the following screenshot:



That's it. We have learnt how to write a basic unit test case for a HTTP Trigger.

6
Monitoring and Troubleshooting Azure Serverless Services

In this chapter, you will learn about the following:

- Troubleshooting your Azure Functions
- Integrating Azure Functions with Application Insights
- Monitoring your Azure Functions
- Pushing custom telemetry details to Application Insights Analytics
- Sending application telemetry details via email
- Integrating real-time Application Insights monitoring data with Power BI using Azure Functions

Introduction

Completing the development of a project and making an application live is not the end of the deployment story. We need to continuously monitor our applications, analyze their performance, and review their logs to understand whether there are any issues that end users are facing.

Azure provides us with multiple tools to meet all of our monitoring requirements, right from the development and maintenance stages.

In this chapter, you will learn how to utilize these tools and take any action necessary based on the information available.

Troubleshooting your Azure Functions

In this recipe, you will learn how to view the application logs of your function apps' using the log streaming feature of functions.

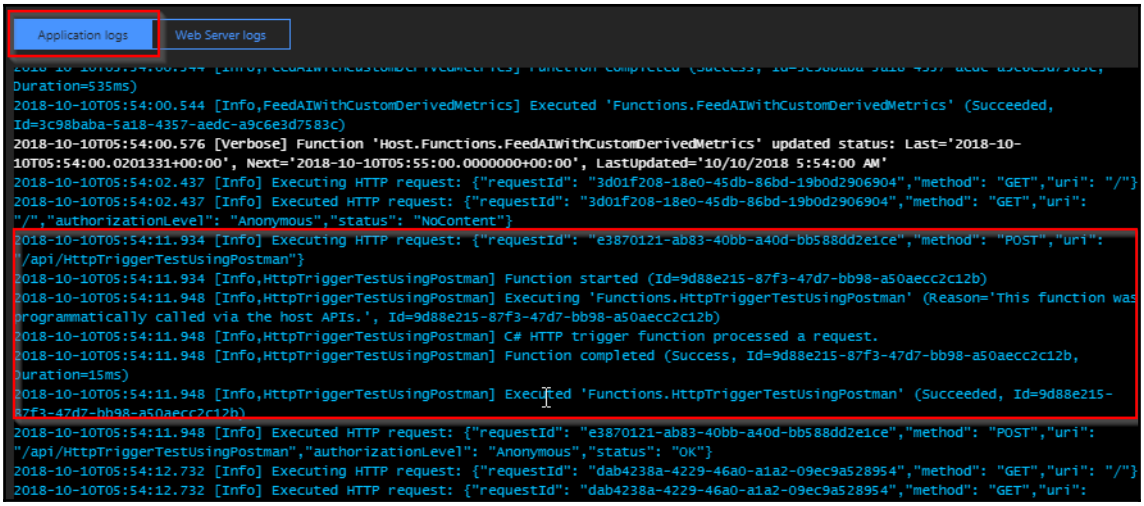
How to do it...

Once you are done with development and have tested your apps thoroughly in your local environment, you might want to deploy them to Azure. There might be cases where you face issues after deploying an application to Azure, as the environment is different. For example, a developer might have missed creating App Settings in the app. With a missing configuration key, your application might not work as expected, and it's not easy to troubleshoot the error. Fortunately, the Azure environment makes it easy with the Log Streaming feature. In this recipe, we will learn how to view real-time logs and also gain an understanding of how to use the **Diagnose and solve problems** feature.

Viewing real-time application logs

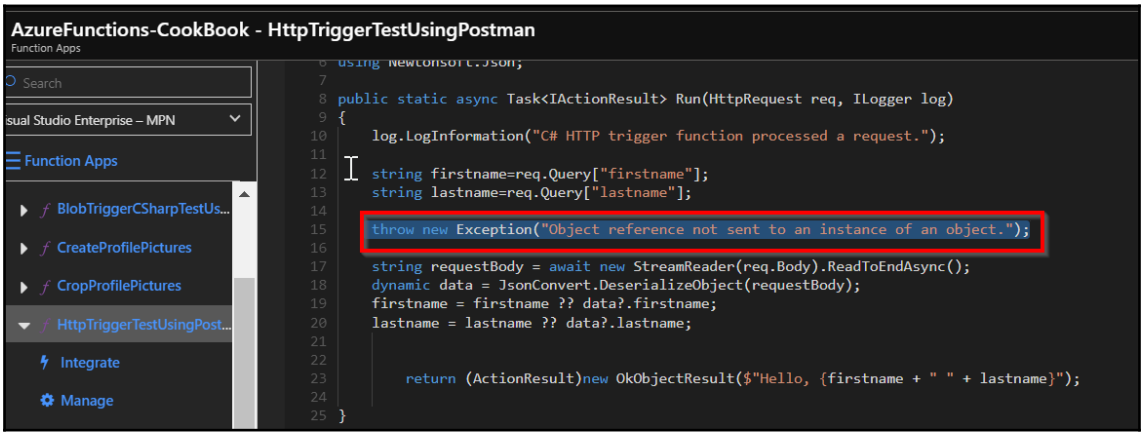
Perform the following steps:

1. Navigate to **Platform features** of the function app and click on the **Log Streaming** button, where you can view the **Application logs** , as shown in the following screenshot:

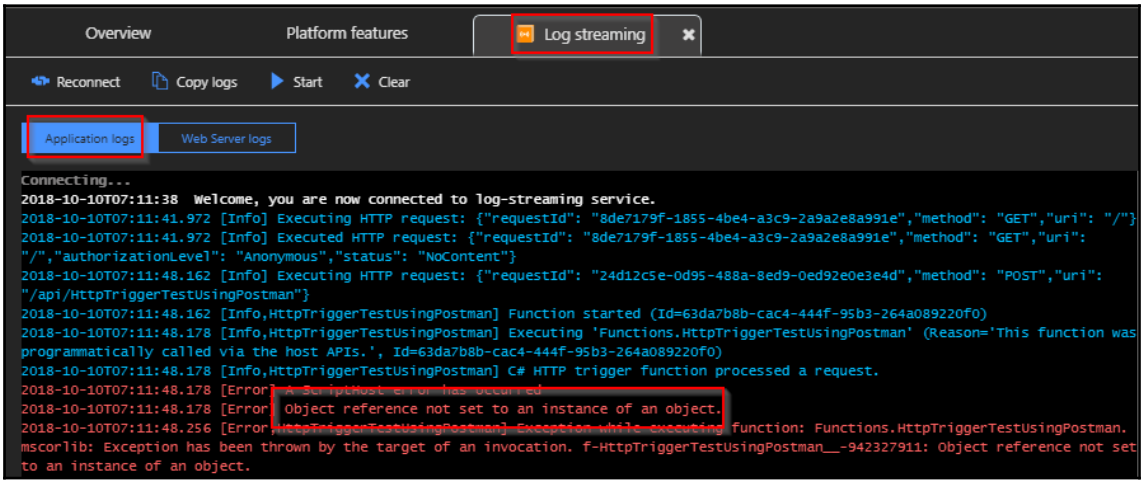


At the time of writing, web server logs provide no information relating to Azure Functions.

2. Let's open any of the Azure Functions that you added earlier in a new browser tab and add a line of code that causes an exception. To make it simple (and to just illustrate how application logs in log streaming work), I have added the following line to the simple HTTP trigger that I created earlier:



3. Subsequently, click on the **Save** button and then on the **Run** button. As expected, you will receive an exception, along with the message in the **Application logs** section shown in the following screenshot:

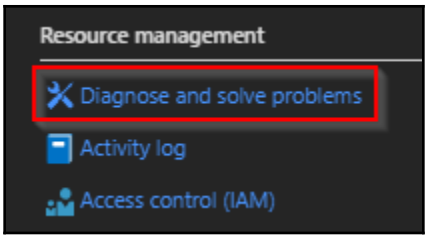


The log window shows errors only for that particular function, and not for the other functions associated with the function app. That is where log streaming application logs come in handy, which can be used across the functions of any given function app.

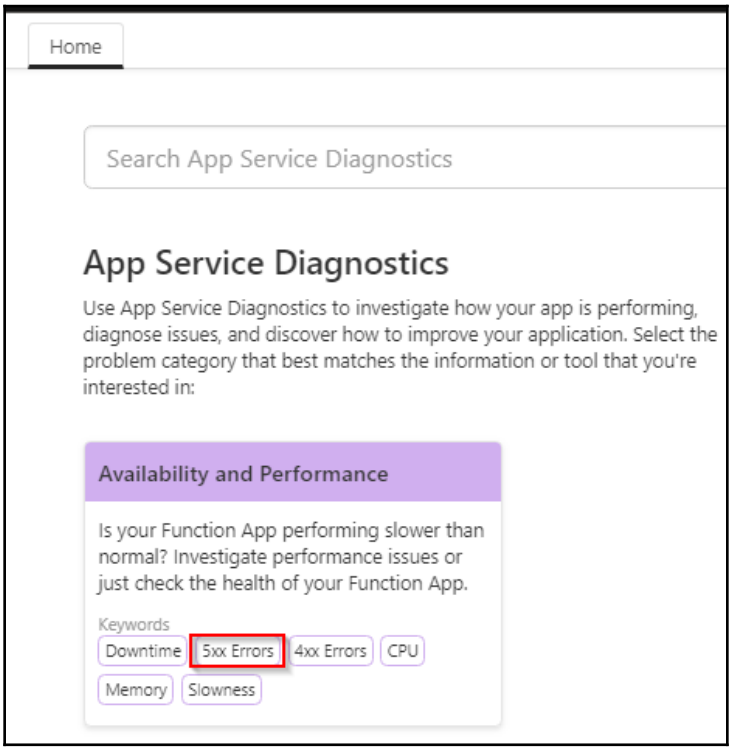
Diagnosing the entire function app

In the preceding section, we learned how to monitor application errors in real time, which will be helpful to quickly identify and fix any we come across. However, it is not always possible to monitor application logs and understand the errors that end users might be facing. Azure Functions provides another great tool, called **Diagnose and solve problems**:

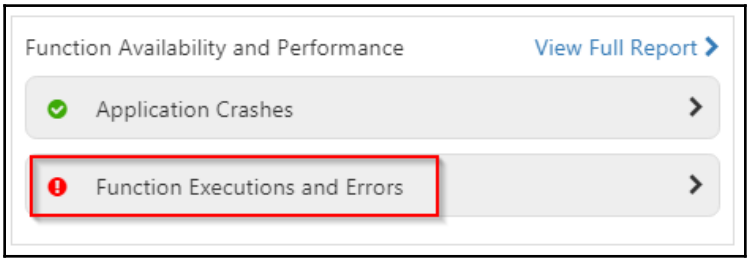
1. Navigate to **Platform features** and click on **Diagnose and solve problems**, as shown in the following screenshot:



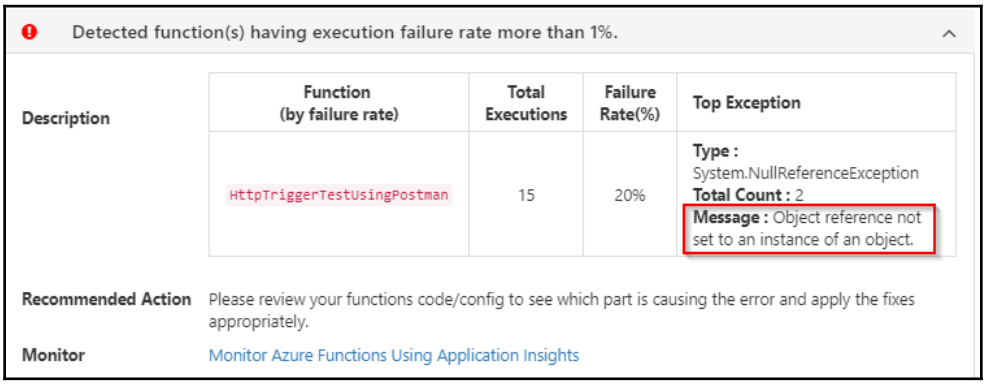
2. Soon after, you will be taken to another blade, where you can choose the right category for the problems that you are currently troubleshooting. Click on **5xx Errors** to view details about the exceptions that end users are facing, as shown in the following screenshot:



3. This will show you **Function Availability and Performance** and **Application Crashes**. Click on the **Function Availability and Performance** option to view the actual links, as shown in the following screenshot:



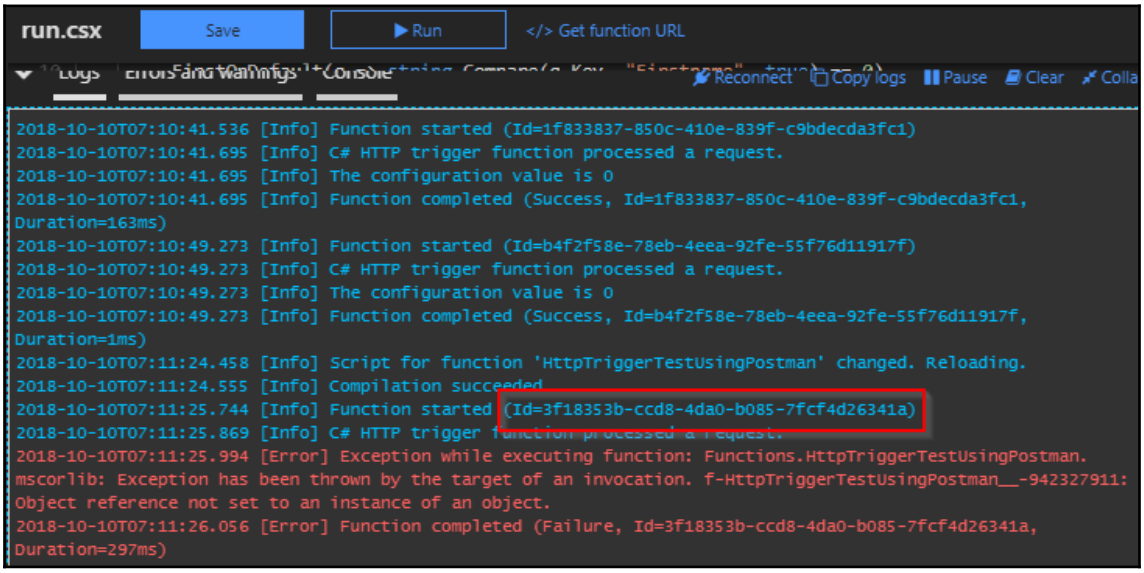
4. Click on **Function Executions and Errors** to view the detailed exceptions, as shown in the following screenshot:



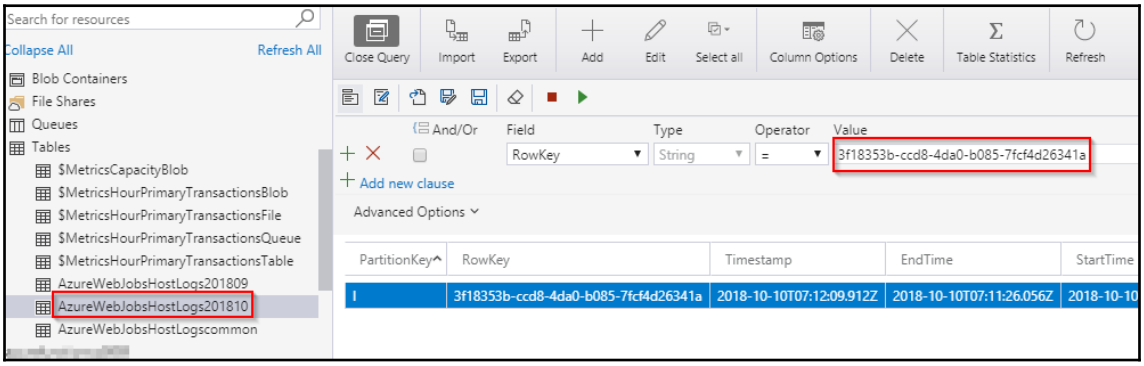
There's more...

Each function event is logged in an Azure Table Storage service. Every month, a table is created with the name `AzureWebJobsHostLogs<Year><Month>`.

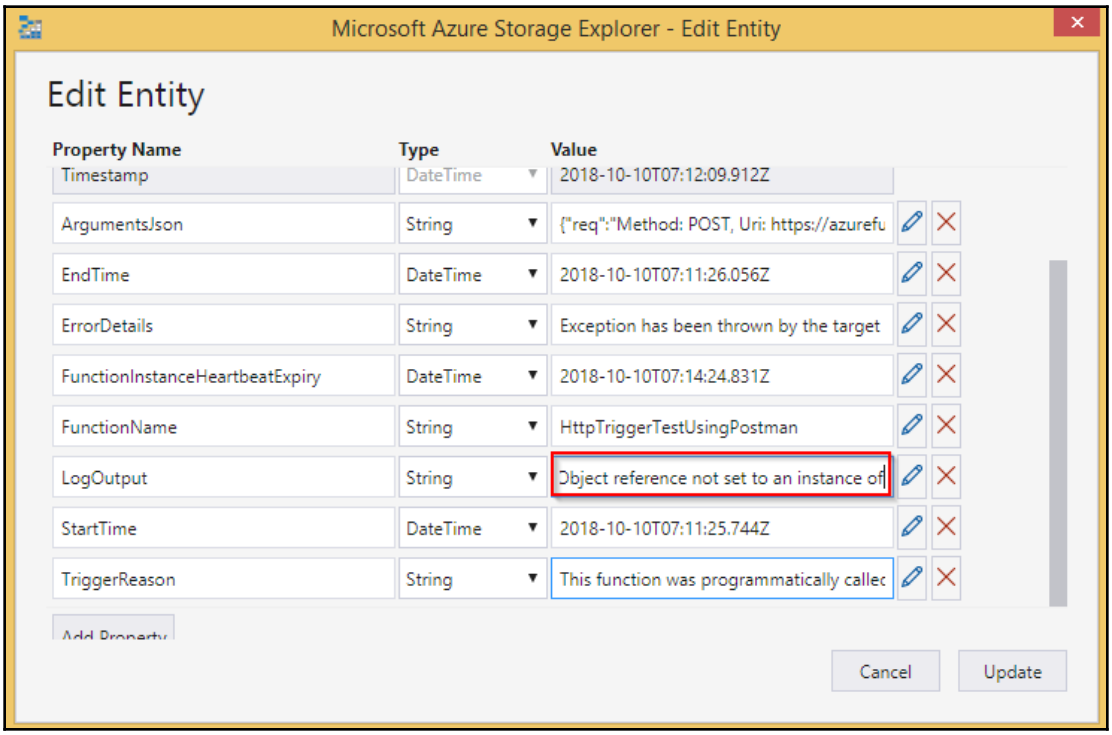
As part of troubleshooting, if you would like to get more details on any error, you first find the `Id` field in the **Invocation details** section, as shown in the following screenshot:



Look for that data in the **RowKey** column of the `AzureWebJobsHostLogs<year><month>` table, as shown in the following screenshot:



As shown in the preceding screenshot, you will get the log entry stored in the Table storage. Clicking on the row will open up the complete details of the error, as shown in the following screenshot:



Integrating Azure Functions with Application Insights

Application Insights is an application performance management service. Once you integrate Application Insights into your application, it will start sending telemetry data to your Application Insights account, hosted on the cloud. In this recipe, you will learn how simple it is to integrate Azure Functions with Application Insights.

Getting ready

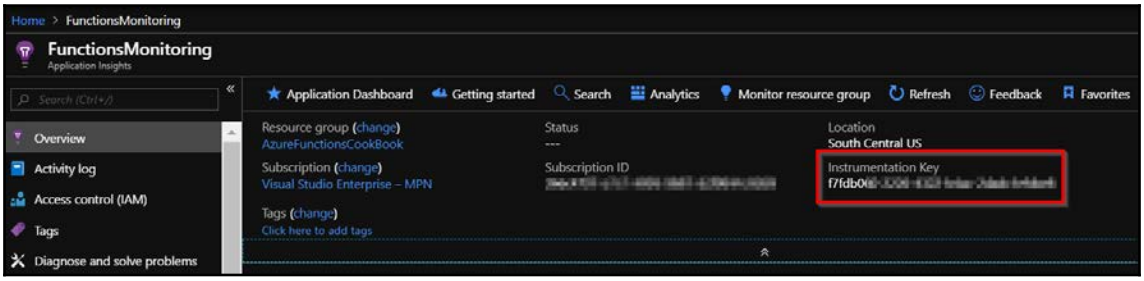
We created an Application Insights account in the *Testing and validating Azure Functions responsiveness using Application Insights* recipe of `Chapter 5`, *Exploring Testing Tools for the Validation of Azure Functions*. Create an account, if you haven't already done so, by taking the following steps:

1. Navigate to Azure Management portal, click on **Create a resource**, and then select **Management Tools**.
2. Choose **Application Insights** and provide all the required details. If you already created an Application Insights account in the previous recipe, you can ignore this step.

How to do it...

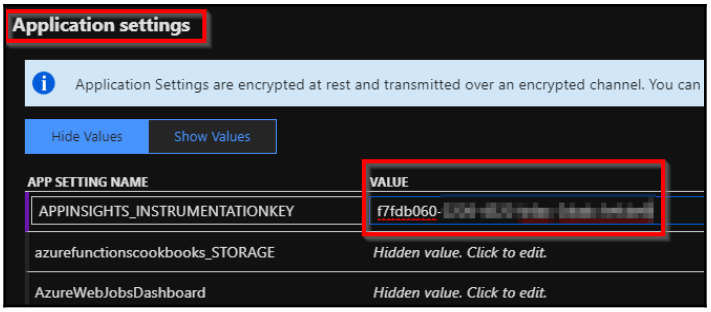
Perform the following steps:

1. Once the Application Insights account is created, navigate to the **Overview** tab and go to **Instrumentation Key**, as shown in the following screenshot:

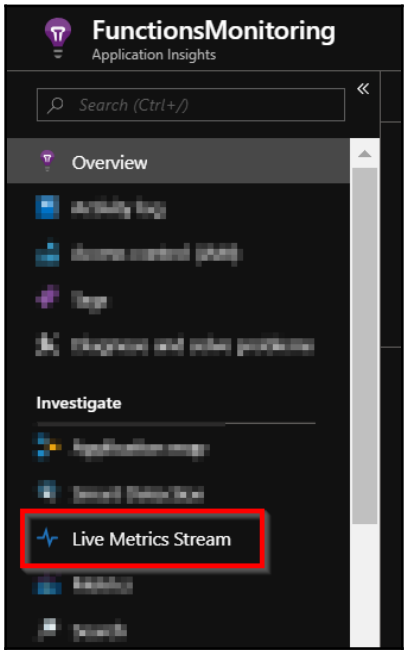


2. Navigate to **Function apps**, for which you would like to enable monitoring and go to **Application settings**.

3. Add a new key with the name `APPINSIGHTS_INSTRUMENTATIONKEY` and provide the instrumentation key that you copied from the Application Insights account, shown as follows, then click on **Save** to save the changes:

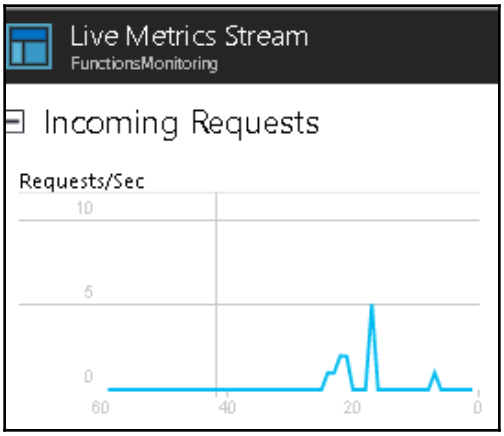


4. That's it; you can start utilizing all the features of Application Insights to monitor the performance of your Azure functions. Open **Application Insights** and the `RegisterUser` function in two different tabs to test how **Live Metrics Stream** works:
    - Open **Application Insights** and click on **Live Metrics Stream** in the first tab of your browser, as shown in the following screenshot:



- Open any of your Azure functions (in my case, I have opened HTTP trigger) in another tab and run a few tests to ensure that it emits some logs to Application Insights.

5. After you have completed those tests, go to the tab that has Application Insights. You should see the live traffic going to your function app, as shown in the following screenshot:



How it works...

We have created an Application Insights account. Once you integrate Application Insights' **Instrumentation Key** with Azure Functions, the runtime will take care of sending the telemetry data asynchronously to your Application Insights account.

There's more...

In **Live Metrics Stream**, you can also view all the instances, along with some other data, such as the number of requests per second handled by your instances.

Monitoring your Azure Functions

We now understand how to integrate Azure Functions with Application Insights. Let's now gain an understanding of how to view the logs that are written to Application Insights by Azure Functions code so that, as a developer, you can troubleshoot any exceptions that occur.

Let's make a small change to the HTTP trigger function and then run it a few times.

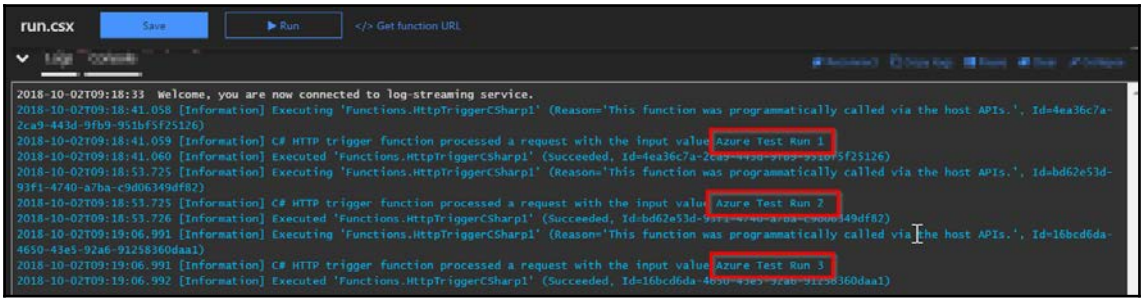
How to do it...

Perform the following steps:

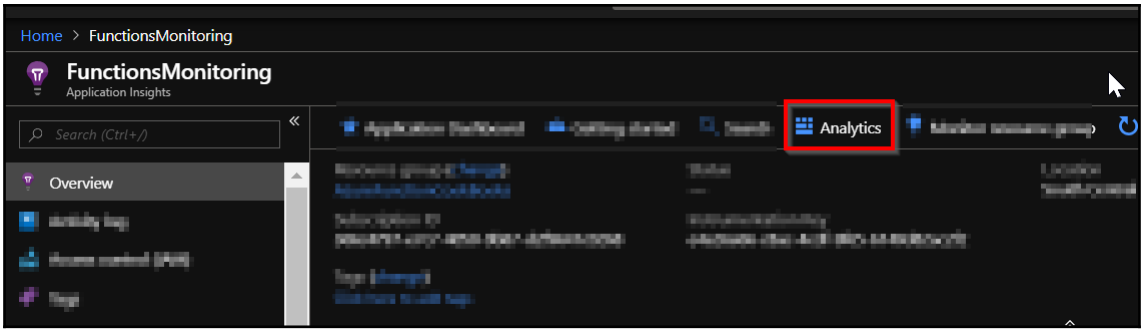
1. Navigate to the HTTP trigger that you created and replace the following code. I just moved the line of code that logs the information to the **Logs** console and added the `name` parameter at the end of the method:

```
public static async Task<IActionResult> Run(HttpRequest req,
ILogger log)
     {
          string name = req.Query["name"];
          string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
          dynamic data =
JsonConvert.DeserializeObject(requestBody);
          name = name ?? data?.name;
          log.LogInformation($"C# HTTP trigger function processed a
request with the input value {name}");
          return name != null
             ? (ActionResult)new OkObjectResult($"Hello, {name}")
             : new BadRequestObjectResult("Please pass a name on
the query string or in the request body");
}
```

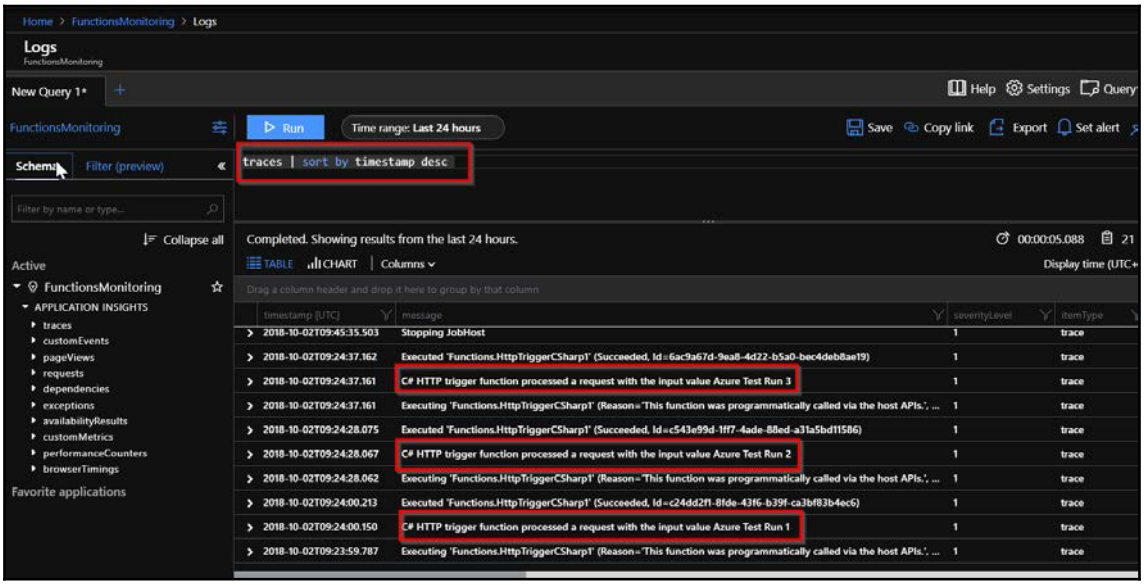
2. Now, run the function by providing the value for the `name` parameter with different values such as `Azure Test Run 1`, `Azure Test Run 2` and `Azure Test Run 3`. This is just for demo purposes. You can use any input you like. The **Logs** console will show the following output:



3. The logs in the preceding **Logs** console are only available when you are connected to the **Logs** console. You will not get them when you go offline. That's where Application Insights comes in handy. Navigate to the Application Insights instance that you have integrated with the Azure function app.
4. Click on the **Analytics** button that is available in the **Overview** tab of Application Insights, as shown in the following screenshot:



5. In the analytics query window, type the `traces | sort by timestamp desc` command, which returns all the traces sorted by date descending, as shown in the following screenshot:



How it works...

In HTTP trigger, we have added a `log` statement that displays the value of the `name` parameter that the user provides. We ran HTTP trigger a few times. After some time, click on the **Analytics** button in the Application Insights button, which opens the analytics window, where you can write queries to view the telemetry data that is being emitted by Azure Functions. All of this can be achieved without writing any custom code.

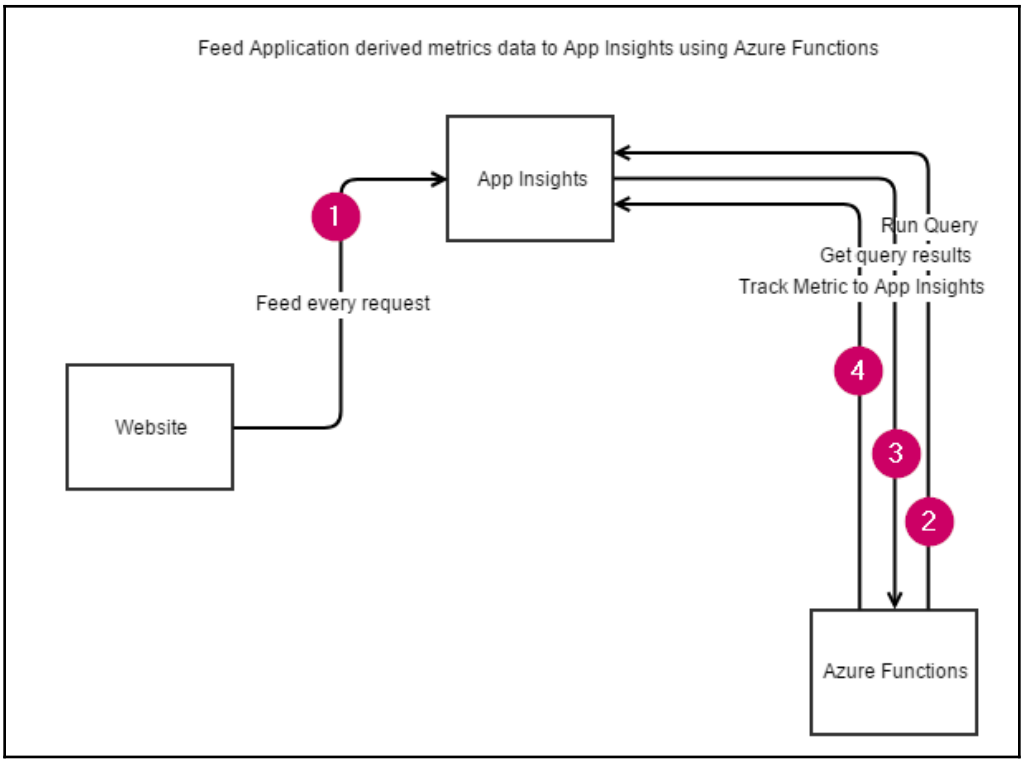
Pushing custom telemetry details to Application Insights Analytics

You have been asked by our customers to provide analytic reports for a derived metric within Application Insights. So, what is a derived metric? Well, by default, Application Insights provides you with many insights into metrics such as requests, errors, exceptions, and so on.

You can run queries on the information that Application Insights provides using its Analytics query language.

In this context, `requests per hour` is a derived metric, and if you would like to build a new report within Application Insights, then you will need to feed Application Insights about the new derived metric on a regular basis. Once you feed the required data regularly, Application Insights will take care of providing reports for your analysis.

We will be using Azure Functions that feed Application Insights with a derived metric named `requests per hour`:



For this example, we will develop a query using the Analytics query language for the `request per hour` derived metric. You can make changes to the query to generate other derived metrics for your requirements, say, *requests per hour for my campaign* or something similar to that.

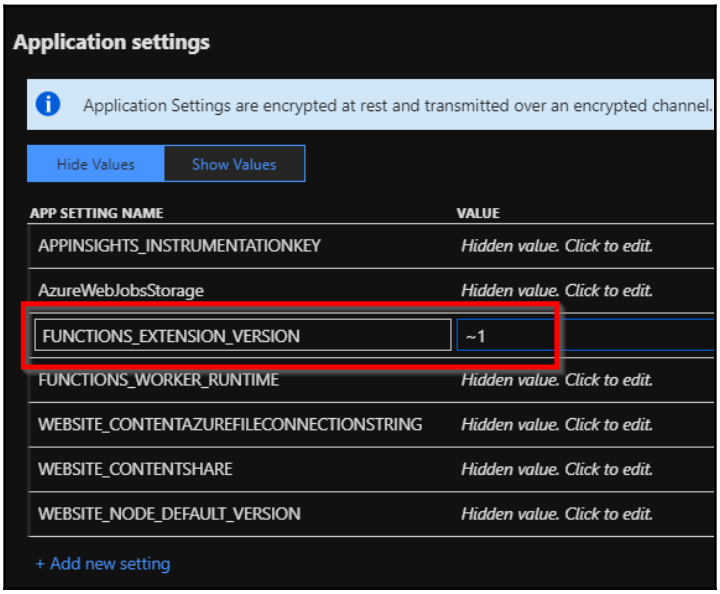
You can learn more about the Analytics query language at `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-analytics-reference`.

Getting ready

Perform the following prerequisite steps:

1. Create a new Application Insights account, if you don't have one already.
2. Make sure you have a running application that integrates with Application Insights. You can learn how to integrate your application with Application Insights at `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-asp-net`.

It is recommended to create this recipe and the following two in a separate Azure function app that is based on runtime v1, as these templates are not available in v2 yet. By the time, if you don't see the **Application Insights Scheduled Analytics** template, then change the Azure Functions runtime version to v1 by setting the value of `FUNCTIONS_EXTENSIONS_VERSION` to `~1`, as shown in the following screenshot, in the **Application settings** of Azure Functions:



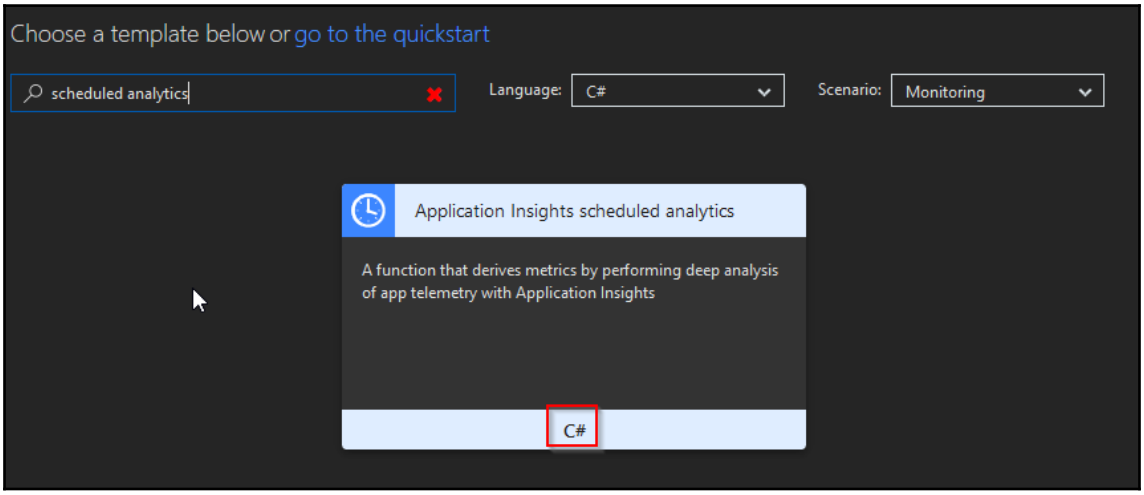
How to do it...

We will perform the following steps to push custom telemetry details to Application Insights Analytics.

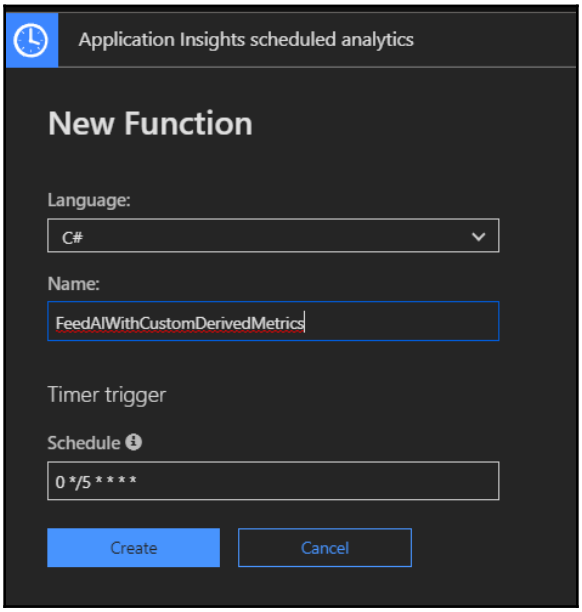
Creating an Application Insights function

Perform the following steps:

1. Create a new function template by choosing **Monitoring** in the **Scenario** drop-down, as shown in the following screenshot. You can also search for `scheduled analytics` to easily find the template:



2. Now, click on **C#** (shown in the preceding screenshot) and provide the name along with the schedule frequency at which the function needs to run:

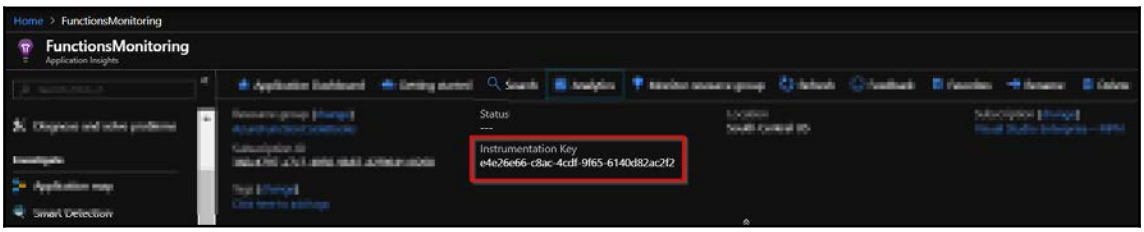


3. As shown in the preceding screenshot, click on the **Create** button to create the function.

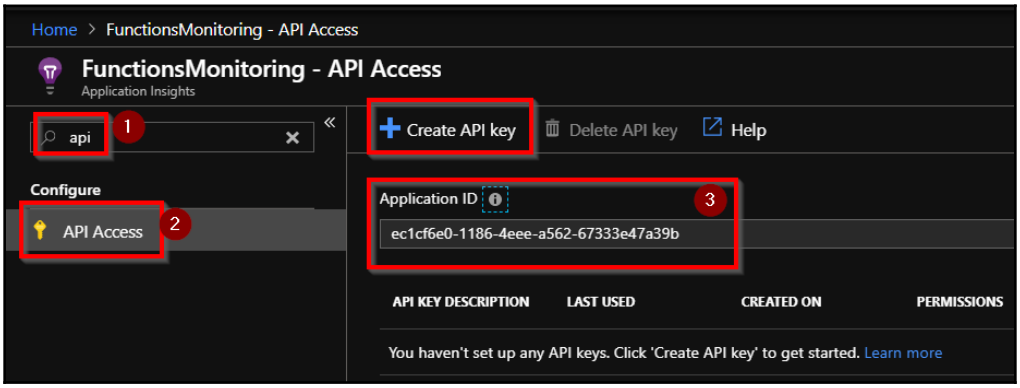
Configuring access keys

Perform the following steps:

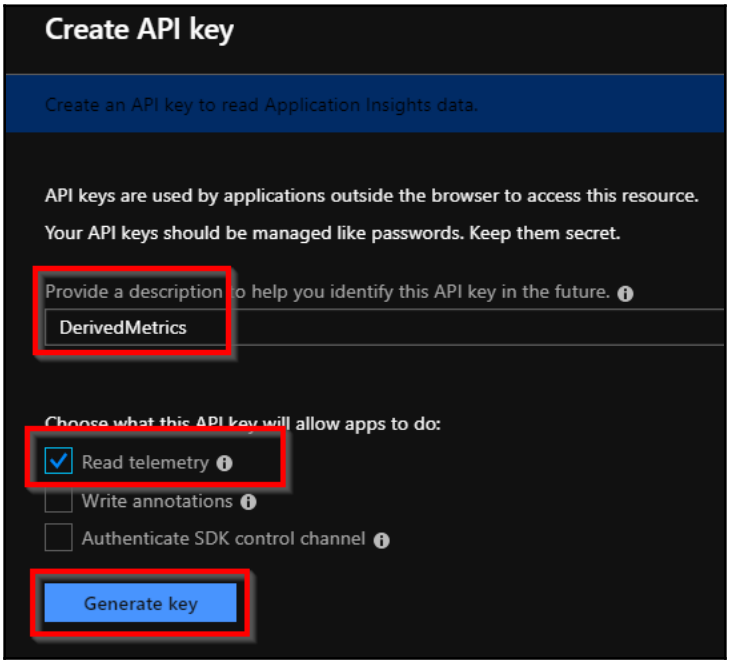
1. Navigate to Application Insights's **Overview** blade, as shown, and copy the **Instrumentation Key**. We will be using the **Instrumentation Key** to create an application setting named `AI_IKEY` in the function app:



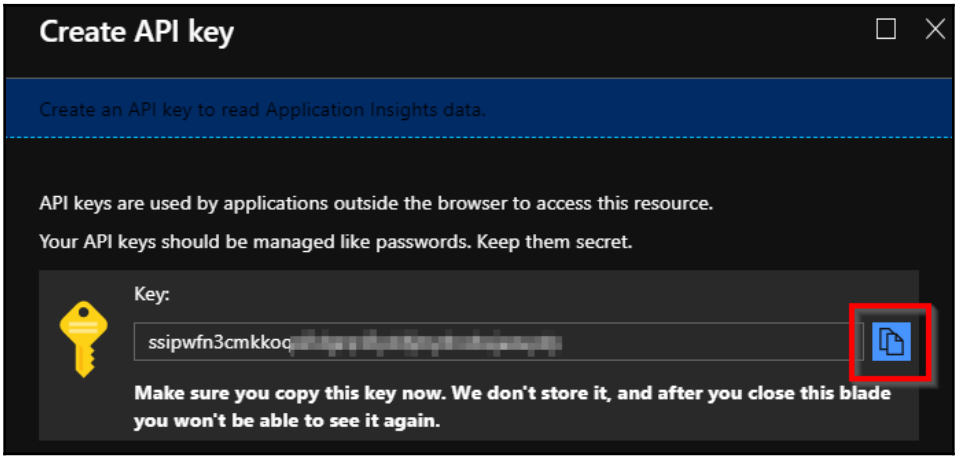
2. Navigate to the **API Access** blade and copy the `Application ID`. We will be using this `Application ID` to create a new app setting with the name `AI_APP_ID` in the function app:



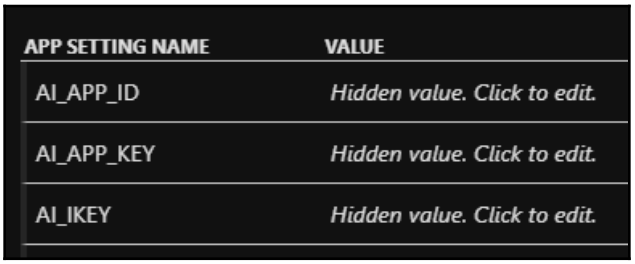
3. We also need to create a new API key. As shown in the preceding step, click on the **Create API key** button to generate the new API key, as shown in the following screenshot. Provide a meaningful name, check the **Read telemetry** data, and click on **Generate key**:



4. Soon after, you can view and copy the key, as shown in the following screenshot. We will be using this to create a new app setting with the name `AI_APP_KEY` in our function app:



5. Create all three app setting keys in the function app, as shown in the following screenshot. These three keys will be used in our Azure Function named `FeedAIwithCustomDerivedMetric`.



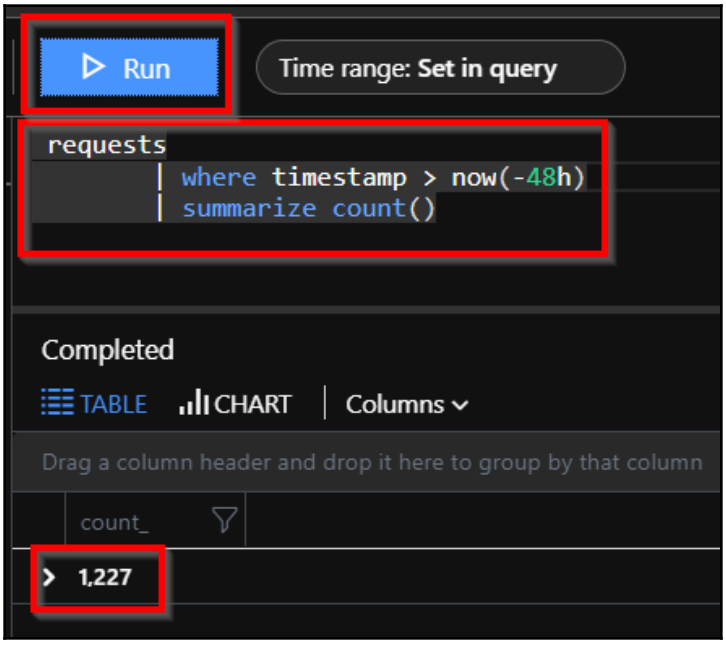
Integrating and testing an Application Insights query

Perform the following steps:

1. Now, it's time to develop a query that provides us with the `requests per hour` derived metric value. Navigate to the Application Insights **Overview** blade and click on the **Analytics** button.
2. You will be taken to the **Analytics** blade. Write the following query in the query tab. You can write your own query as per your requirements. Make sure that the query returns a scalar value:

```
requests
| where timestamp > now(-1h)
| summarize count()
```

3. Once you are done with your query, run it by clicking on the **Run** button to see the count of records, as shown in the following screenshot:

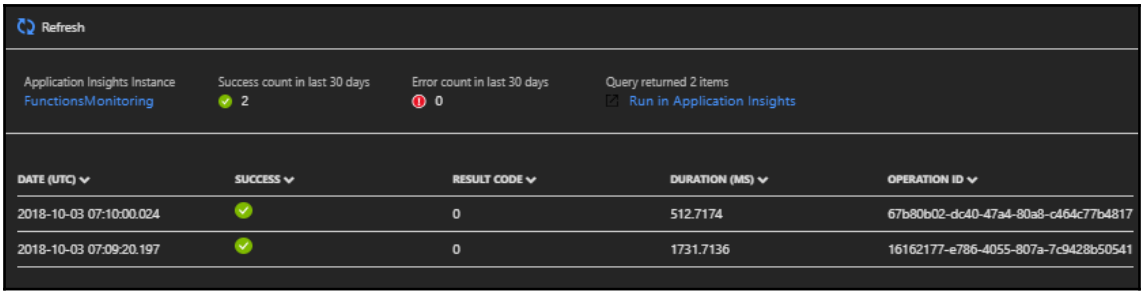


4. We are now ready with the required Application Insights query. Let's integrate the query with our `FeedAIwithCustomDerivedMetrics` function. Navigate to the Azure Functions code editor and make the following changes:
    1. Provide a meaningful name for your derived metric, in this case, `Requests per hour`.
    2. Replace the default query with the one that we have developed.
    3. Save the changes by clicking on the **Save** button:

```
29 public static async Task Run(TimerInfo myTimer, TraceWriter log)
30 {
31     if (myTimer.IsPastDue)
32     {
33         log.Warning($"[Warning]: Timer is running late! Last ran
34     }
35
36     // [CONFIGURATION_REQUIRED] update the query accordingly for
37     // be sure to run it against Application Insights Analytics
38     // output should be a number if sending derived metrics
39     // [Application Insights Analytics] https://docs.microsoft.com
40     await ScheduledAnalyticsRun(
41         name: "Requests per hour",
42         query: @"
43 requests
44 | where timestamp > now(-1h)
45 | summarize count()
46 ",
47         log: log
48     );
49 }
```

5. Let's do a quick test to see whether you have configured all three app settings and the query correctly. Navigate to the **Integrate** tab and change the run frequency to one minute, as shown in the following screenshot:

6. Now, let's navigate to the **Monitor** tab and see whether everything is working fine. If there are any problems, you will see an **X** mark in the **Status** column. View the error in the **Logs** section of the **Monitor** tab by clicking on the **Invocation log** entry:

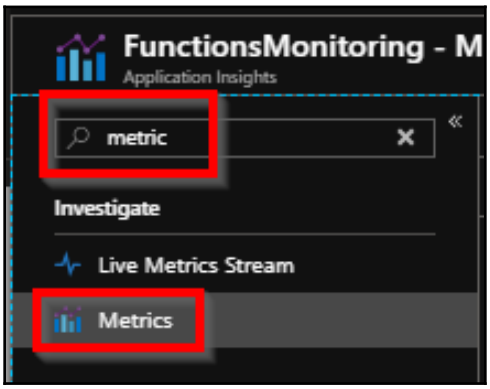


7. Once you have made sure that the function is running smoothly, revert the **Schedule** frequency to one hour.

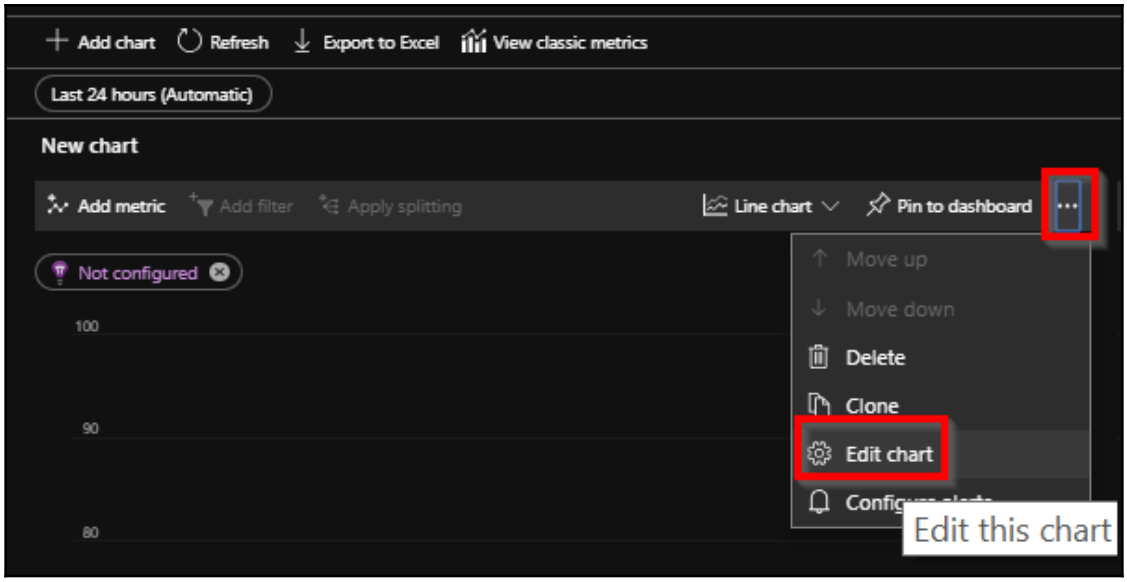
Configuring the custom derived metric report

Perform the following steps:

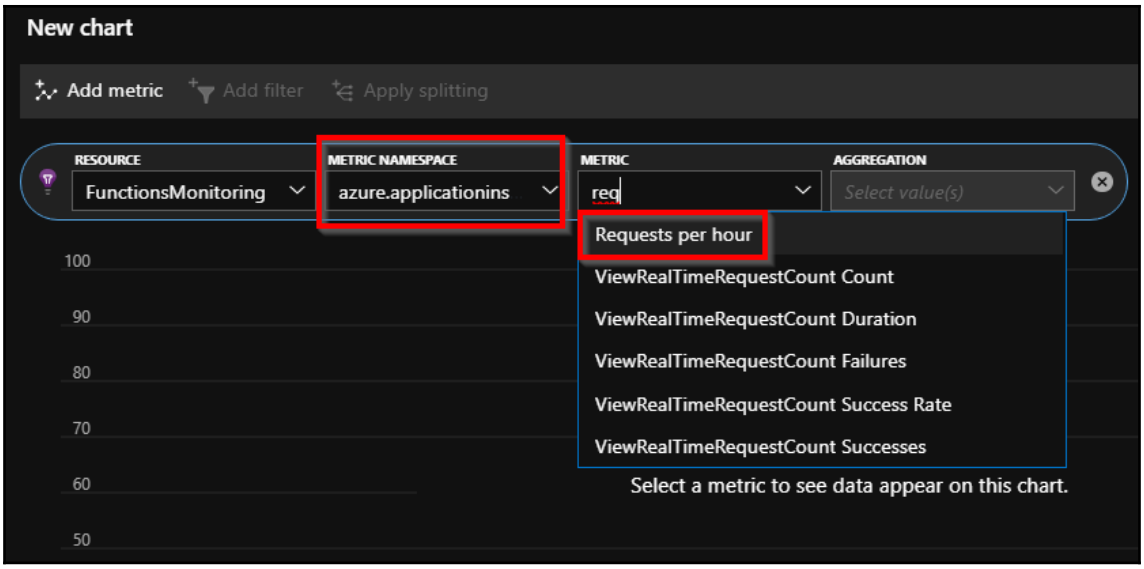
1. Navigate to the Application Insights's **Overview** tab and click on the **Metrics** menu, as shown in the following screenshot:



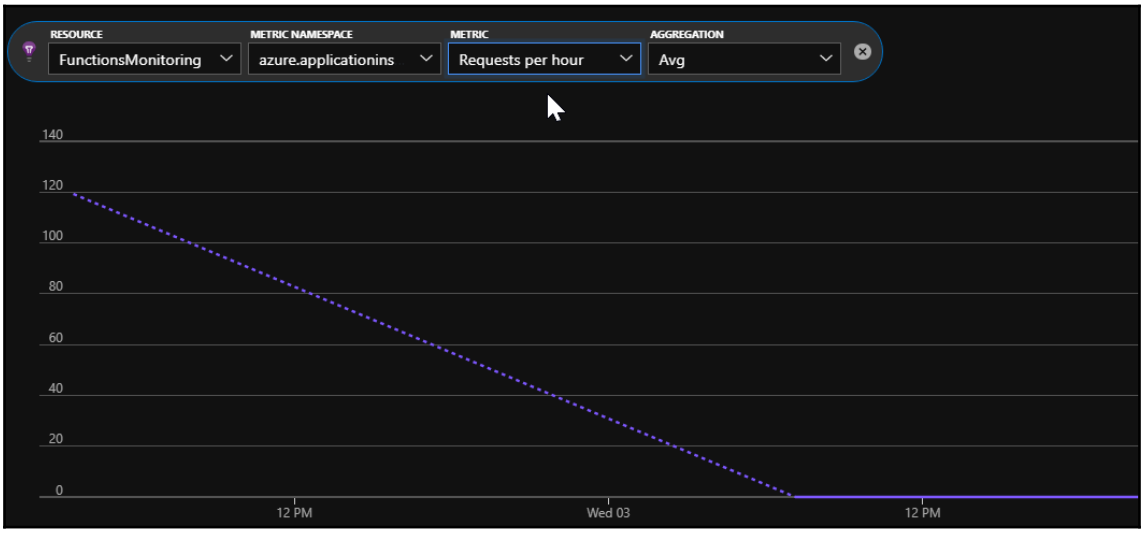
2. **Metrics Explorer** is where you will find all of your analytics regarding different metrics. In **Metrics Explorer**, click on the **Edit** button of any report to configure your custom metric, as shown in the following screenshot (or you can click on the **Add chart** button in the top-left of the following screenshot):



3. Thereafter, you will be taken to the **Chart details** blade, where you can configure your custom metric and all other details related to the chart. In the **METRIC NAMESPACE** drop-down, choose **azure.applicationinsghts**, as shown in the following screenshot and then choose the **Request per hour** custom metric that you created:



4. Subsequently, your chart will be created as shown in the following screenshot:



How it works...

This is how the entire process works:

- We created the Azure Function using the default **Application Insights Scheduled Analytics** template.
- We configured the following keys in the **Application settings** of the Azure function app:
    - Application Insights's **Instrumentation Key**
    - The application ID
    - The API access key
- The Azure function runtime automatically consumed the Application Insights API, ran the custom query to retrieve the required metrics, and performed the required operations to feed the derived telemetry data to Application Insights.
- Once everything in the Azure function was configured, we developed a simple query that pulled the request count of the last hour and fed it to Application Insights as a custom derived metric. This process repeated every hour.
- Later, we configured a new report using Application Insights **Metrics** with our custom derived metric.

Sending application telemetry details via email

One of the activities of your application, once live, will be to receive a notification email with details about health, errors, response time, and so on, at least once a day.

Azure Functions provides us with the ability to get all the basic details using a function template with code that is responsible for retrieving all the required values from Application Insights and the plumbing code of framing the email body and sending the email using SendGrid. We will look at how to do that in this recipe.

Getting ready

Perform the following prerequisite steps:

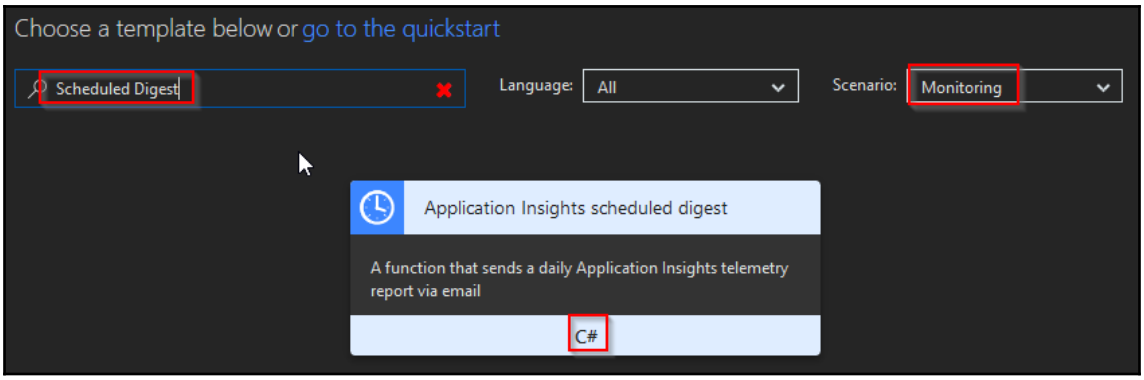
1. Create a new SendGrid account, if you have not yet created one, and get the SendGrid API key
2. Create a new Application Insights account, if you don't have one already
3. Make sure you have a running application that integrates with Application Insights

You can learn how to integrate your application with Application Insights at `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-asp-net`.

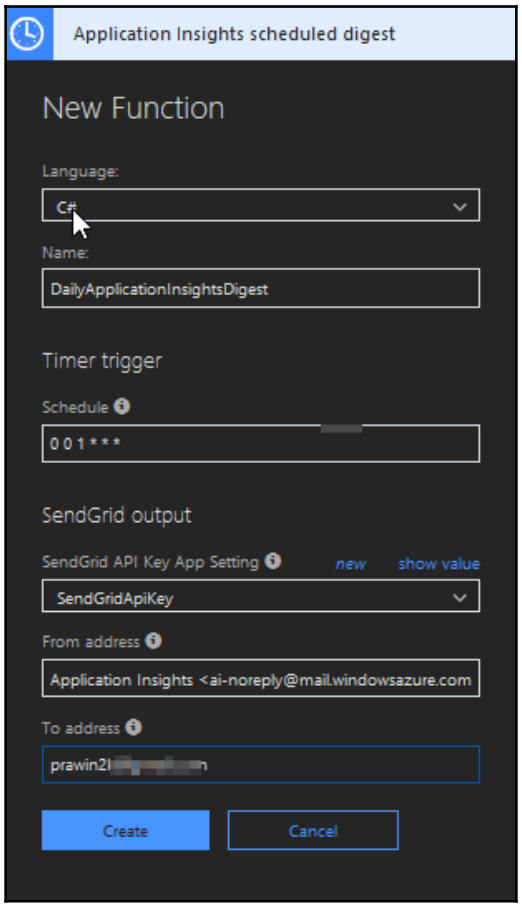
How to do it...

Perform the following steps:

1. Create a new function by choosing **Monitoring** in the **Scenario** drop-down and select the **Application Insights scheduled digest—C#** template, as shown in the following screenshot:



2. Soon after, you will be prompted to provide the name of the function, the scheduled frequency, and the **SendGrid API Key** for the SendGrid output binding, as shown in the following screenshot:



3. Next, click on the **Create** button shown in the previous step to create the new Azure Function. The template creates all the code that's required to query the data from Application Insights and sends an email to the person mentioned in the **To address** of the preceding screenshot.

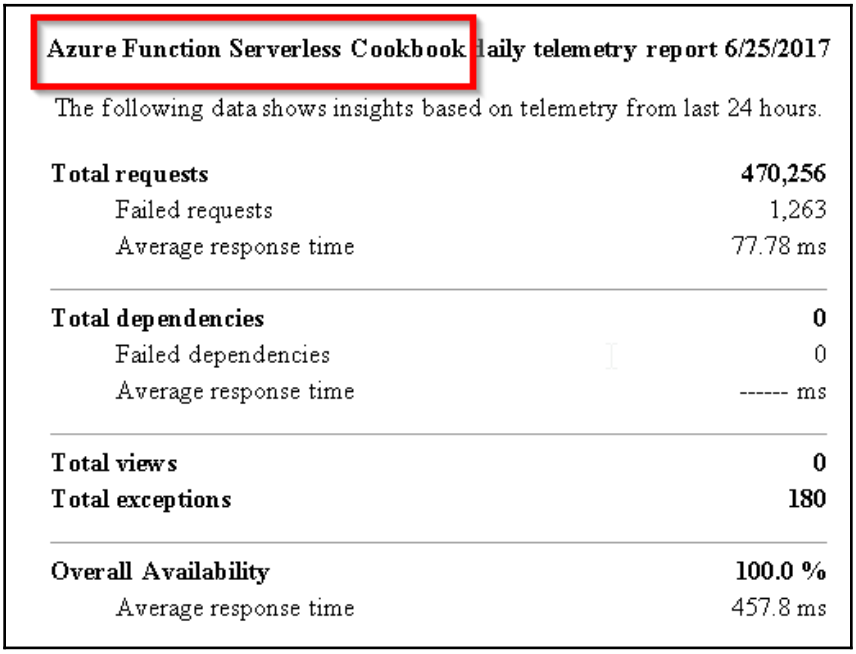
Make sure that you follow the steps in the *Configuring access keys* section of the *Pushing custom telemetry details to analytics of Application Insights* recipe to configure these access keys: Application Insights **Instrumentation Key**, the application ID, and the API access key.

4. Navigate to the `run.csx` function and change the app name to your application name, as shown in the following screenshot:

```
// [CONFIGURATION_REQUIRED] configure {appName} accordingly for your app/email
string appName = "Azure Function Serverless Cookbook";
```

5. If you've configured all the settings properly, you will start receiving emails based on the timer settings.
6. Let's do a quick test run by clicking on the **Run** button above the code editor, as shown in the following screenshot:

7. This is a screenshot of the email that I received after clicking on the **Run** button in the preceding screenshot:



How it works...

The Azure Function uses the Application Insights API to run all the Application Insights Analytics queries, retrieves all the results, frames the email body with all the details, and invokes the SendGrid API to send an email to the configured email account.

There's more...

Azure templates provide default code that has a few queries that are generally useful for monitoring application health. If you have any specific requirements for getting notification alerts, go ahead and add new queries to the `GetQueryString` method. In order to incorporate the new values, you will also need to change the `DigestResult` class and the `GetHtmlContentValue` function.

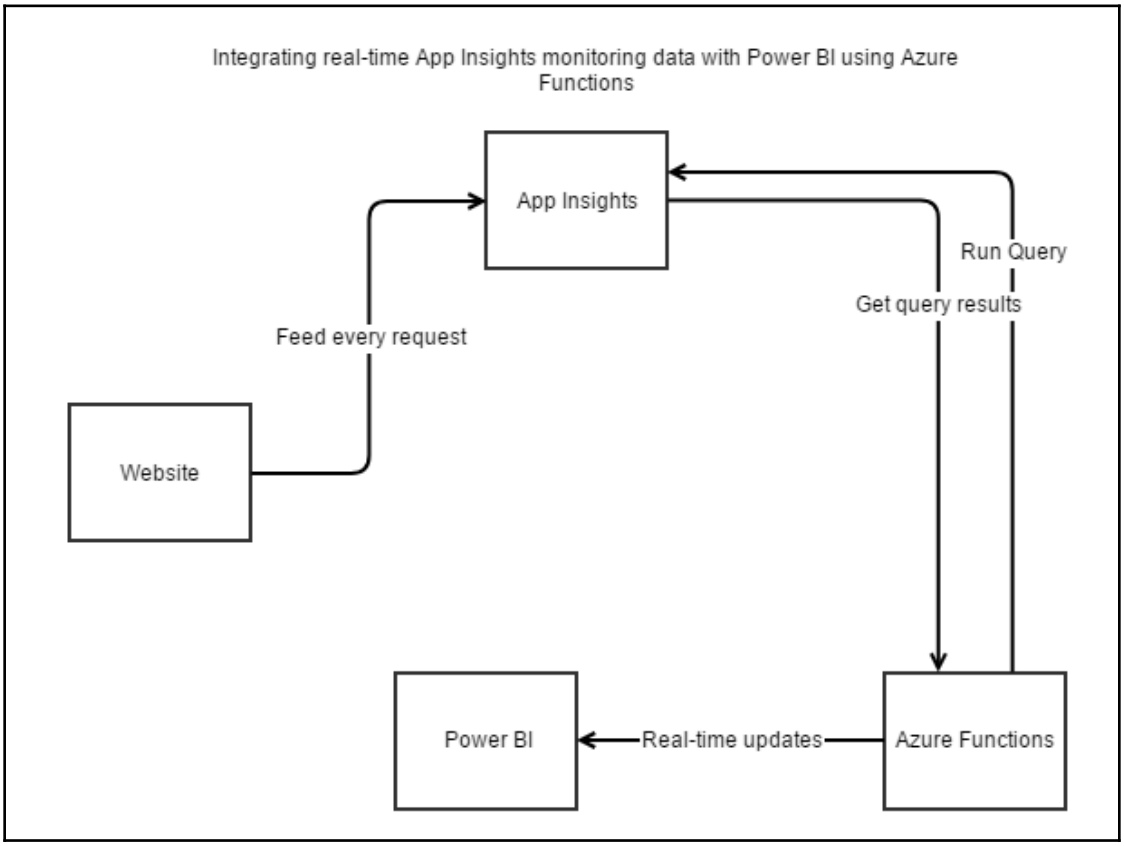
See also

The *Sending an email notification to the administrator of the website using the SendGrid service* recipe of `Chapter 2`, *Working with Notifications Using the SendGrid and Twilio Services*.

Integrating real-time Application Insights monitoring data with Power BI using Azure Functions

Sometimes, you will need to view real-time data about your application's availability or information relating to your application's health on a custom website. Retrieving information for Application Insights and displaying it in a custom report would be a tedious job, as you'd need to develop a separate website and build, test, and host it somewhere.

In this recipe, you will learn how easy is to view real-time health information for the application by integrating Application Insights and Power BI. We will be leveraging Power BI capabilities for the live streaming of data, and Azure timer functions to continuously feed health information to Power BI. This is a high-level diagram of what we will be doing in the rest of the recipe:



In this recipe, we will use the **Application Insights Power BI** template of a function app that's created using Azure Functions v1 runtime. The Azure Functions v2 runtime doesn't have it. If you are using the v2 runtime, you can simply create a Timer Trigger and follow the instructions in this recipe.

Getting ready

Perform the following prerequisites steps:

1. Create a Power BI account at `https://powerbi.microsoft.com/en-us/`.
2. Create a new Application Insights account, if you don't have one already.
3. Make sure that you have a running application that integrates with Application Insights. You can learn how to integrate your application with Application Insights at `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-asp-net`.

You need to use your work account to create a Power BI account. At the time of writing, it's not possible to create a Power BI account using a personal email address such as Gmail, Yahoo, and so on.

Make sure that you follow the steps in the *Configuring access keys* section of the *Pushing custom telemetry details to analytics of Application Insights* recipe to configure these access keys: Application Insights **Instrumentation Key**, the application ID, and the API access key.

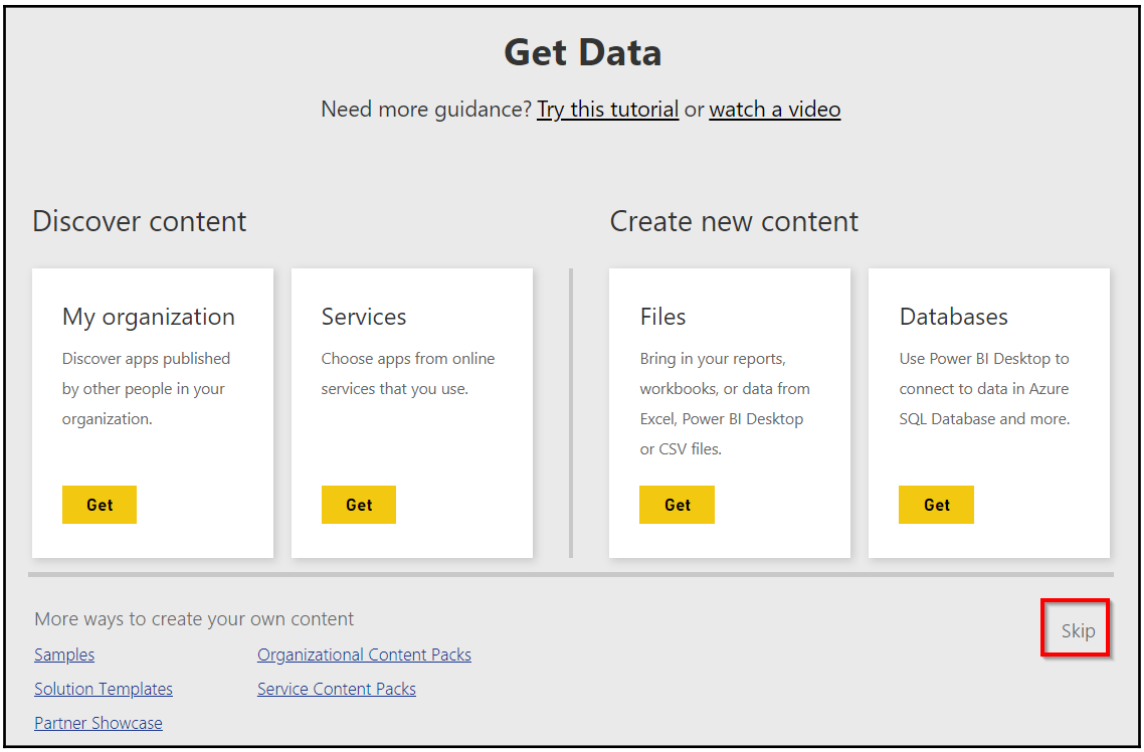
How to do it...

We will perform the following steps to integrate Application Insights and Power BI.

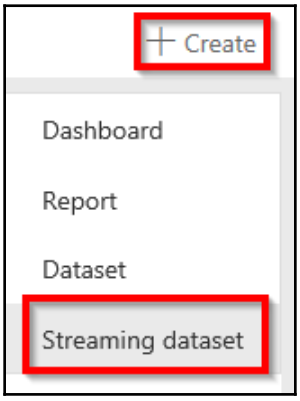
Configuring Power BI with a dashboard, a dataset, and the push URI

Perform the following steps:

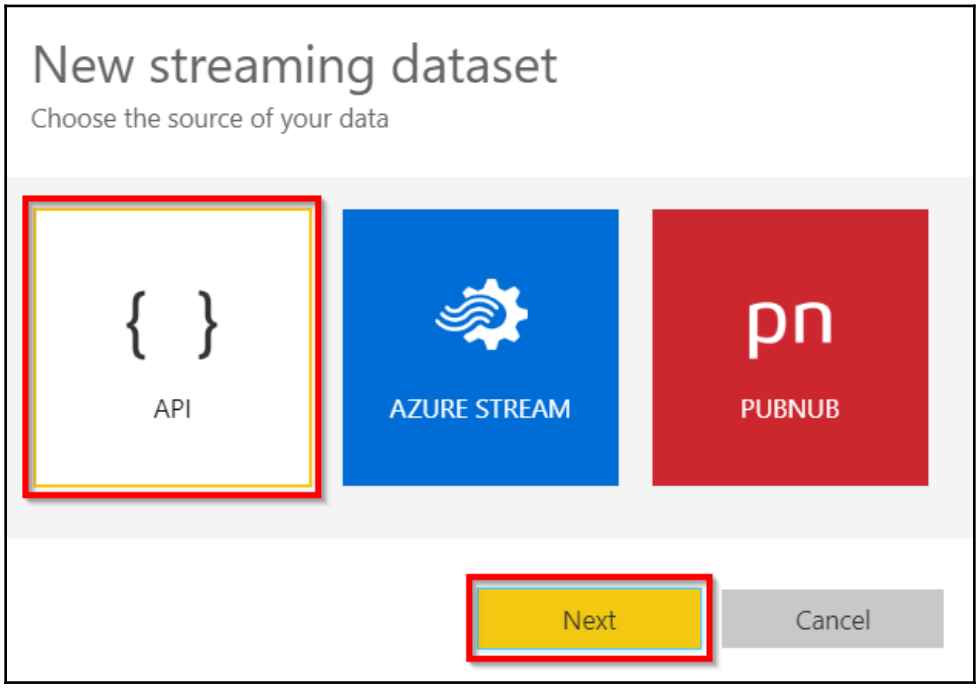
1. If you are using the Power BI portal for the first time, you might have to click on **Skip** on the welcome page, as shown in the following screenshot:



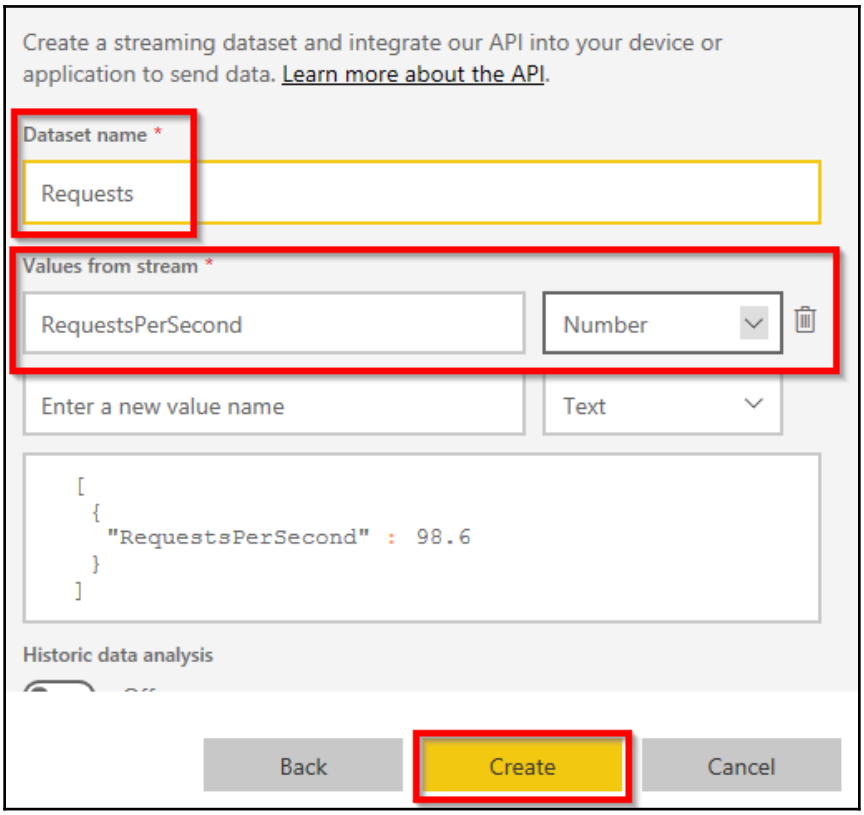
2. The next step is to create a streaming dataset by clicking on **Create** and then choosing **Streaming dataset**, as shown in the following screenshot:



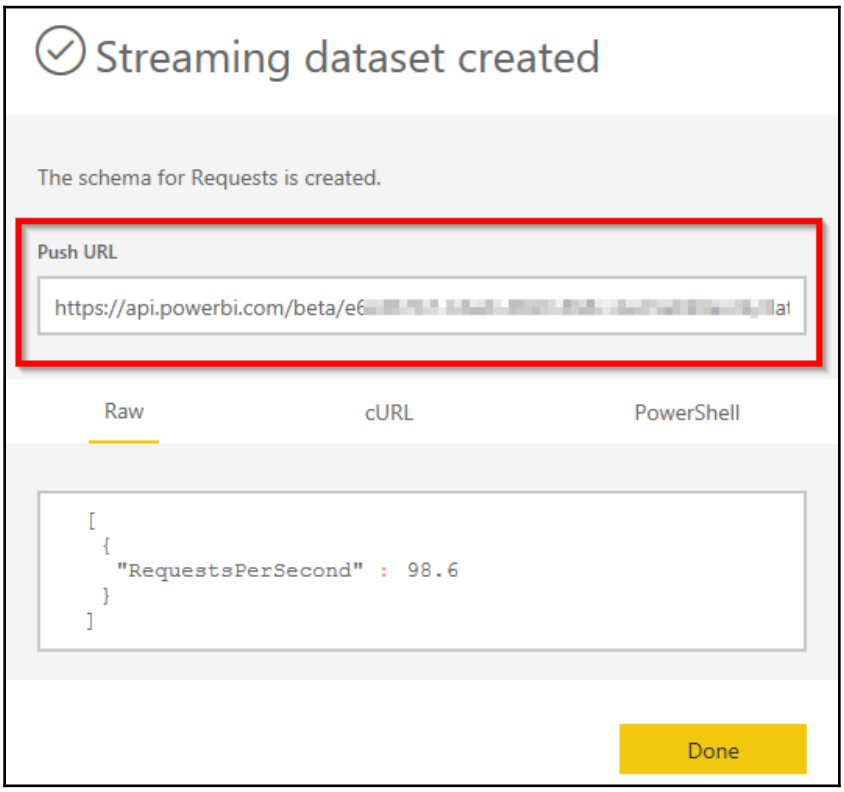
3. In the **New streaming dataset** step, select **API** and click on the **Next** button, as shown in the following screenshot:



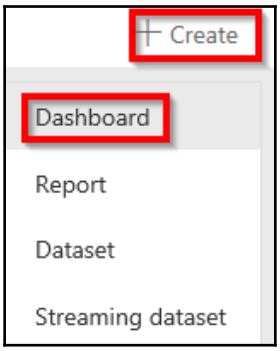
4. In the next step, you need to create the fields for the streaming dataset. Provide a meaningful name for the dataset and provide the values that you would like to push to Power BI. For this recipe, I have created a dataset with just one field, named `RequestsPerSecond`, of type **Number** and clicked on **Create**, as shown in the following screenshot:



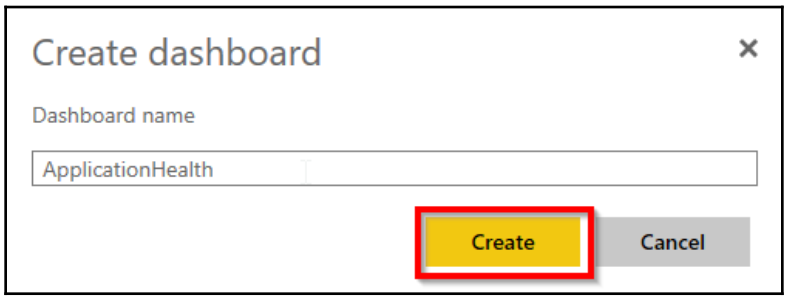
5. Once you create the dataset, you will be prompted with a **Push URL**, as shown in the following screenshot. You will use this **Push URL** in Azure Functions to push the `RequestsPerSecond` data every second (or according to your requirements) with the actual value of requests per second. Click on **Done**:



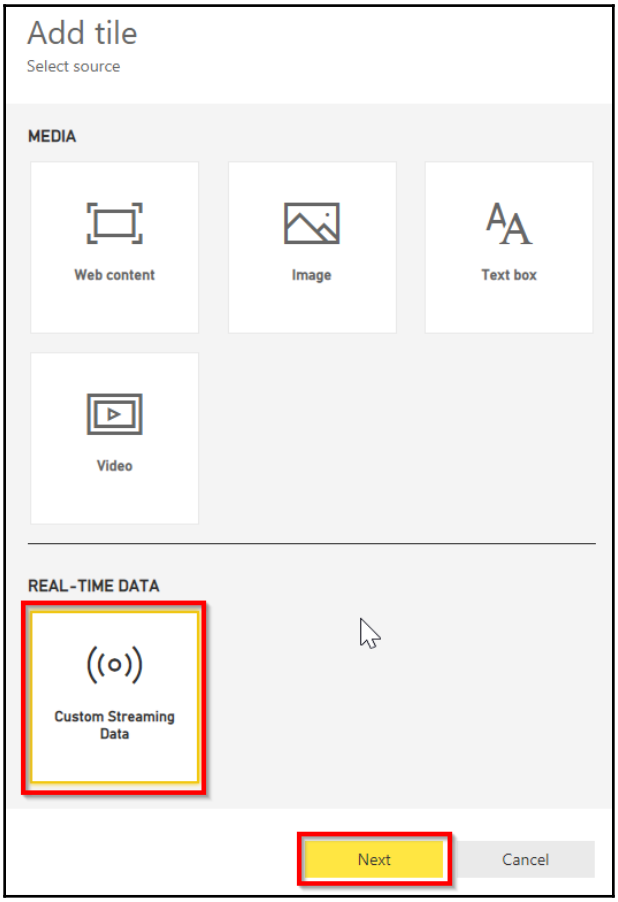
6. The next step is to create a dashboard with a tile in it. Let's create a new dashboard by clicking on **Create** and choosing **Dashboard**, as shown in the following screenshot:



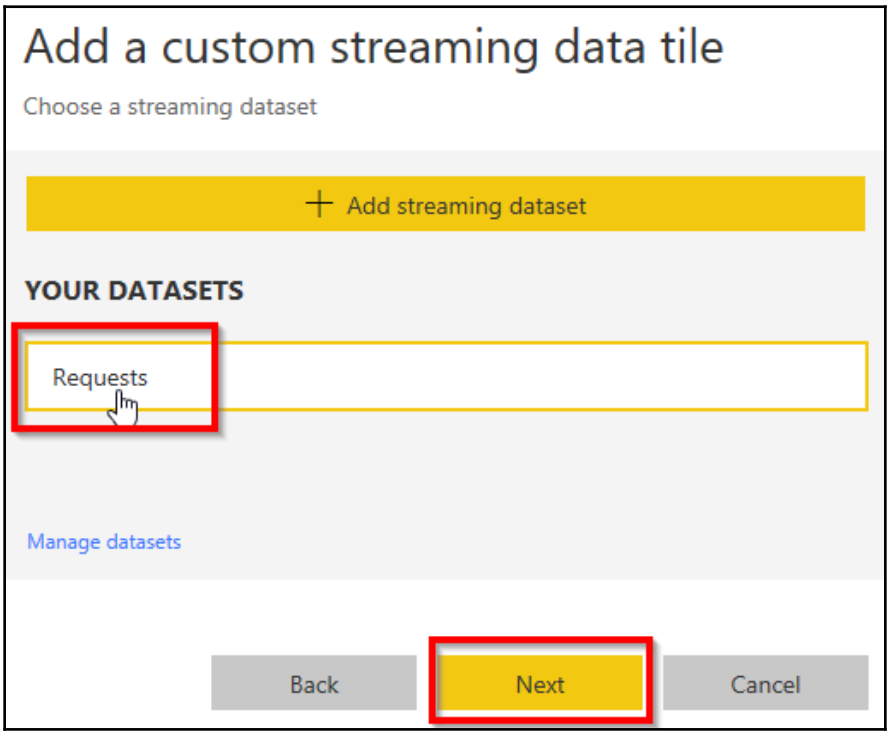
7. In the **Create dashboard** popup, provide a meaningful name and click on **Create**, as shown in the following screenshot, to create an empty dashboard:



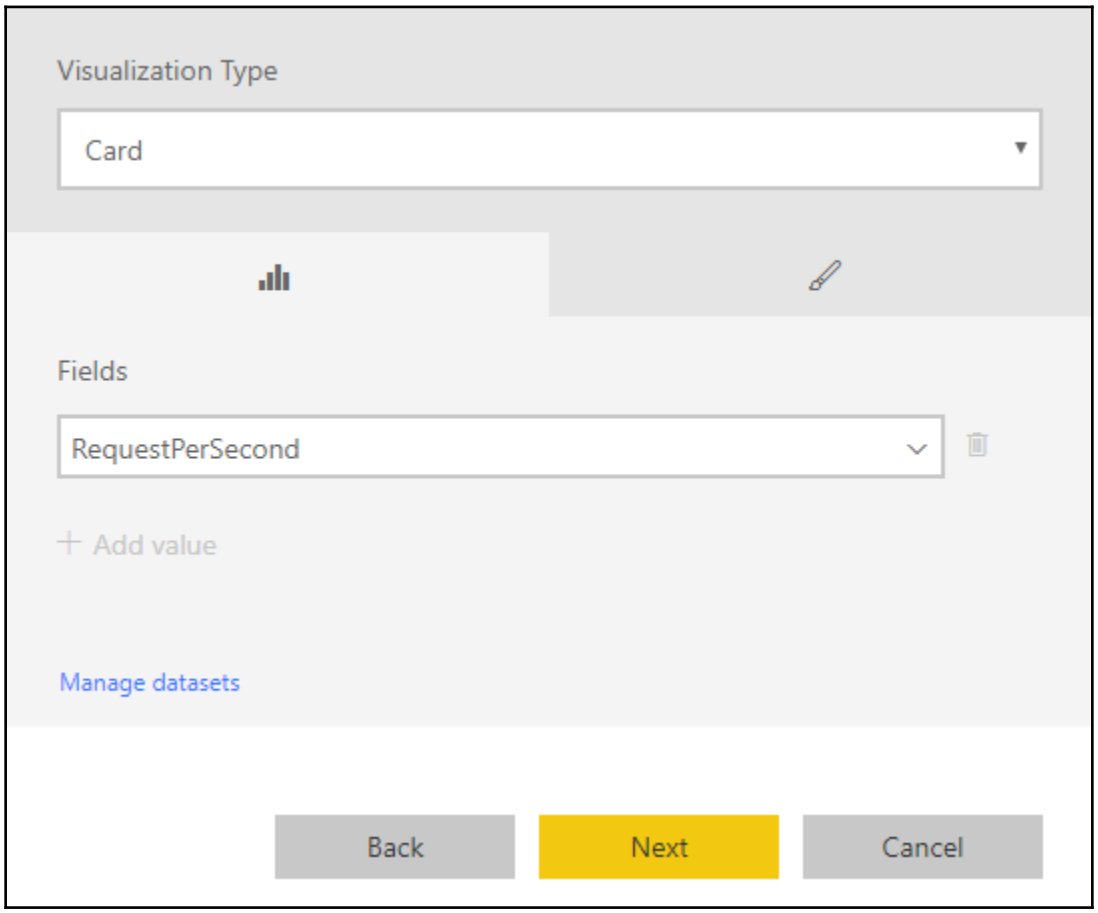
8. In the empty dashboard, click on the **Add tile** button to create a new tile. Clicking on **Add tile** will open a new popup, where you can select the data source from which the tile should be populated:



9. Select **Custom Streaming Data** and click on **Next**, as shown in the preceding screenshot. In the following step, select the **Requests** dataset and click on the **Next** button:



10. The next step is to choose **Visualization Type** (it is **Card** in this case) and select the fields from the data source, as shown in the following screenshot:

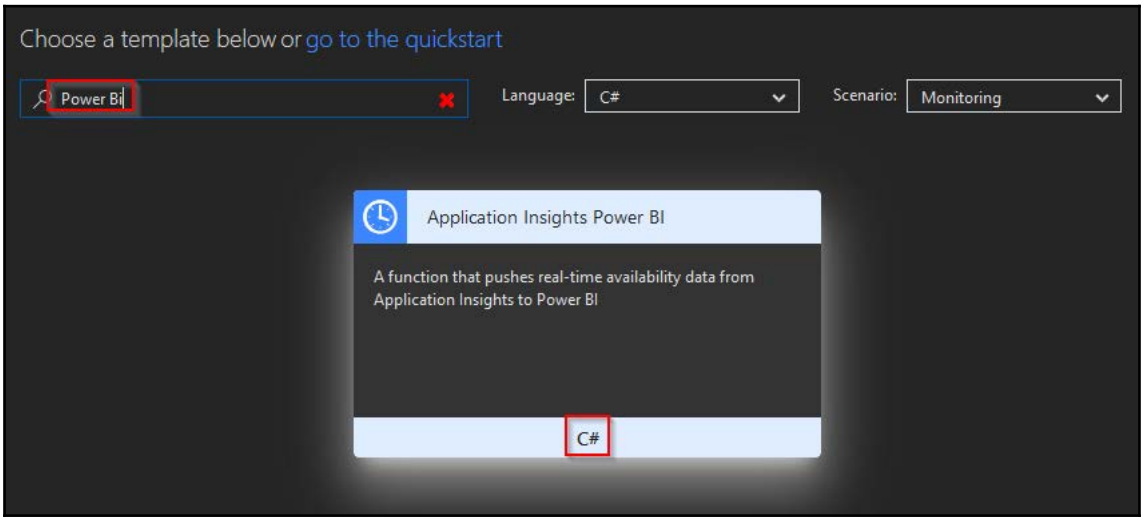


11. The final step is to provide a name for your tile. I have provided **RequestsPerSecond**. The name might not make sense in this case. But you are free to provide any name as per your requirements.

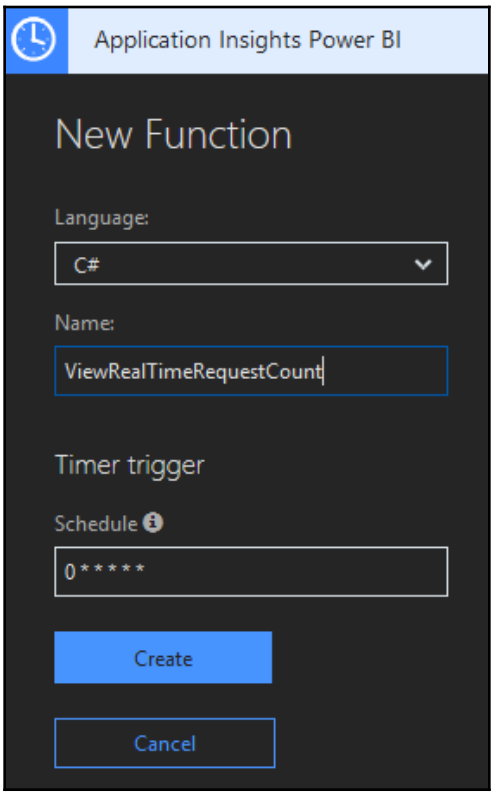
Creating an Azure Application Insights real-time Power BI – C# function

To create an Azure Application Insights real-time Power BI using the C# function, complete the following steps:

1. Navigate to Azure Functions and create a new function using the following template:



2. Click on **C#** in the preceding screenshot, provide the **Name** and click on the **Create** button, as shown in the following screenshot:



3. Replace the default code with the following code. Make sure that you configure the right value for which the analytics query should pull the data. In my case, I have provided five minutes (`5m`) in the following code:

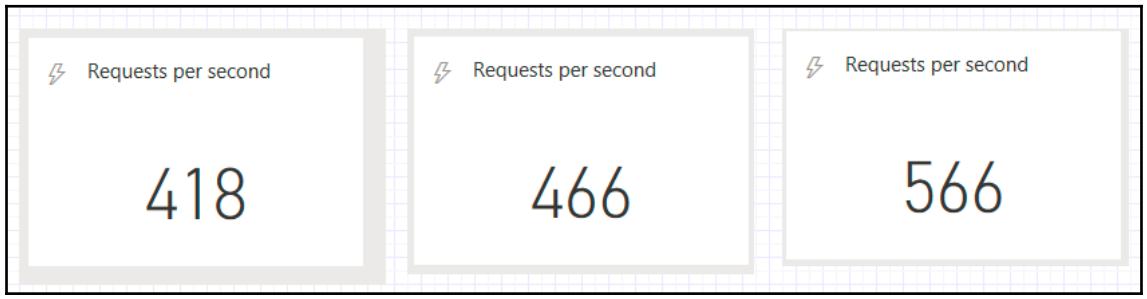
```
#r "Newtonsoft.Json"
using System.Configuration;
using System.Text;
using Newtonsoft.Json.Linq;
private const string AppInsightsApi =
 "https://api.applicationinsights.io/beta/apps";
private const string RealTimePushURL = "PastethePushURLhere";
private static readonly string AiAppId =
 ConfigurationManager.AppSettings["AI_APP_ID"];
private static readonly string AiAppKey =
 ConfigurationManager.AppSettings["AI_APP_KEY"];
```

```
public static async Task Run(TimerInfo myTimer, TraceWriter
 log)
{
   if (myTimer.IsPastDue)
   {
      log.Warning($"[Warning]: Timer is running late! Last ran
       at: {myTimer.ScheduleStatus.Last}");
   }
   await RealTimeFeedRun(
   query: @"
   requests
   | where timestamp > ago(5m)
   | summarize passed = countif(success == true),
     total = count()
   | project passed
   ",
   log: log
   );
   log.Info($"Executing real-time Power BI run at:
    {DateTime.Now}");
}

private static async Task RealTimeFeedRun( string query,
 TraceWriter log)
{
   log.Info($"Feeding Data to Power BI has started at:
    {DateTime.Now}");
   string requestId = Guid.NewGuid().ToString();
   using (var httpClient = new HttpClient())
   {
      httpClient.DefaultRequestHeaders.Add("x-api-key",
       AiAppKey);
      httpClient.DefaultRequestHeaders.Add("x-ms-app",
       "FunctionTemplate");
      httpClient.DefaultRequestHeaders.Add("x-ms-client-
       request-id", requestId);
      string apiPath = $"{AppInsightsApi}/{AiAppId}/query?
       clientId={requestId}&timespan=P1D&query={query}";
      using (var httpResponse = await
       httpClient.GetAsync(apiPath))
       {
          httpResponse.EnsureSuccessStatusCode();
          var resultJson = await
           httpResponse.Content.ReadAsAsync<JToken>();
          double result;
          if (!double.TryParse(resultJson.SelectToken
           ("Tables[0].Rows[0][0]")?.ToString(), out result))
          {
```

```
                    throw new FormatException("Query must result in a
                     single metric number. Try it on Analytics before
                     scheduling.");
                }
                string postData = $"[{{ "requests": "{result}"
                 }}]";
                log.Verbose($"[Verbose]: Sending data: {postData}");
                using (var response = await
                 httpClient.PostAsync(RealTimePushURL, new
                 ByteArrayContent(Encoding.UTF8.GetBytes(postData))))
                 {
                    log.Verbose($"[Verbose]: Data sent with response:
                     {response.StatusCode}");
                 }
            }
        }
    }
```

The preceding code runs an Application Insights analytics query that pulls data for the last five minutes (requests) and pushes the data to the Power BI push URL. This process repeats continuously based on the timer frequency that you have configured.

4. This is a screenshot that has a sequence of pictures showing the real-time data:



How it works...

We have created the following in this specific order:

1. A streaming dataset in the Power BI application
2. A dashboard and new tile that can display the values available in the streaming dataset
3. A new Azure Function that runs an Application Insights Analytics query and feeds data to Power BI using the push URL of the dataset
4. Once everything is done, we can view the real-time data in the Power BI's tile of the dashboard

There's more...

- Power BI allows us to create real-time data in reports in multiple ways. In this recipe, you learned how to create real-time reports using the streaming dataset. The other ways are with the push dataset and the PubNub streaming dataset. You can learn more about all three approaches at `https://powerbi.microsoft.com/en-us/documentation/powerbi-service-real-time-streaming/`.
- Be very careful when you would like real-time application health data. The Application Insights API has a rate limit. Take a look at `https://dev.applicationinsights.io/documentation/Authorization/Rate-limits` to understand more about API limits.

7
Developing Reliable Serverless Applications Using Durable Functions

In this chapter, you will learn the following:

- Configuring Durable Functions in the Azure Management portal
- Creating a Durable Functions hello world app
- Testing and troubleshooting Durable Functions
- Implementing multithreaded reliable applications using Durable Functions

Introduction

When working on developing modern applications that need to be hosted on the cloud, you need to make sure that the applications are stateless. Statelessness is an essential factor for developing cloud-aware applications. For example, you should avoid persisting any data in the resource that is specific to any **virtual machine** (**VM**) instance provisioned to any Azure Service (for example, app service, the API, and so on). If you do so, you cannot leverage some of the services, such as auto-scaling functionality, as the provisioning of instances is dynamic. If you depend on any VM-specific resources, you will end up facing troubles with unexpected behaviors.

Having said that, the downside of the previously mentioned approach is that you end up working on identifying ways of persisting data in different mediums, depending on your application architecture.

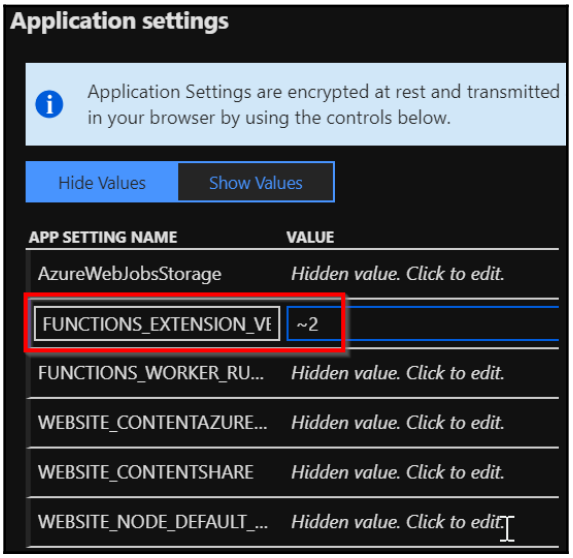
For more information about Durable Functions, check the official documentation, available at `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-overview`.

Configuring Durable Functions in the Azure Management portal

Azure has come up with a new way of handling statefulness in serverless architecture, along with other features, such as durability and reliability, in the form of Durable Functions. This is available as an extension to Azure Functions. In this chapter, we will start learning about Durable Functions.

Getting ready

Create a new function app if not already created. Ensure that the runtime version is `~2` in the **Application settings**, as shown in the following screenshot:

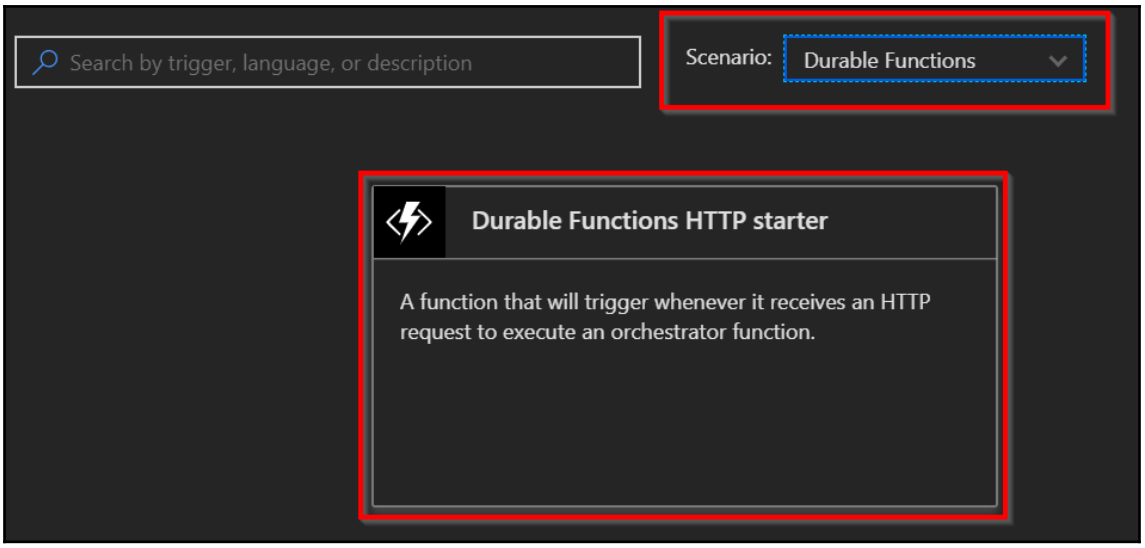


How to do it...

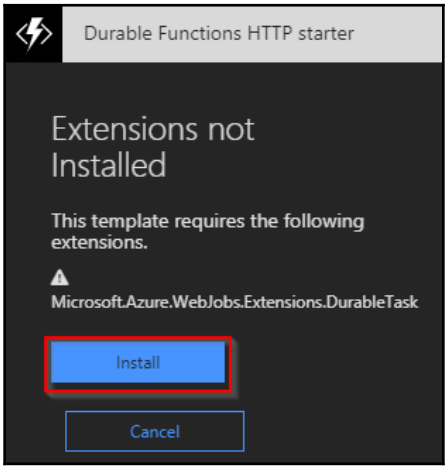
Perform the following steps:

1. Click on the + button to create a new function:

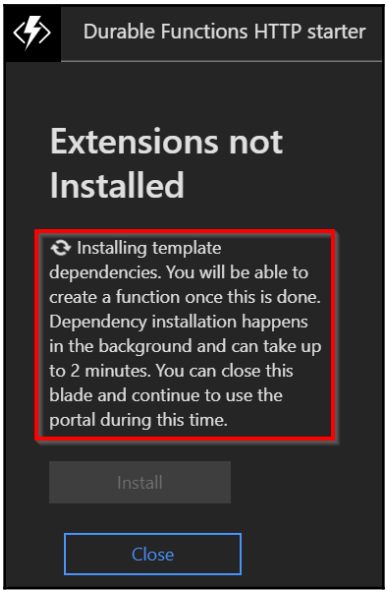
2. Create a new **Durable Functions HTTP starter** function by choosing **Durable Functions** in the **Scenario** drop-down menu, as shown in the following screenshot:



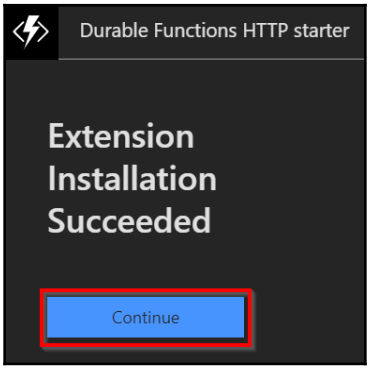
3. Subsequently, a new tab will open:



4. Click on the **Install** button, shown in the preceding step, to start installing the **DurableTask** extensions. It should take around two minutes to install the dependencies:



5. Once the process is complete, the following should appear:



There's more...

Currently, C# is the only language supported for developing Durable Functions. Support for other languages is in the preview phase.

Creating a Durable Function hello world app

Though the overall intention of this book is to have each recipe of every chapter solve at least one business problem, this recipe doesn't solve any real-time domain problems. Instead, it provides some quick-start guidance to help you understand more about Durable Functions and its components, along with the approach of developing Durable Functions. In the next chapter, we will learn how easy it is to develop a workflow-based application using Durable Functions.

Getting ready

We will perform the following steps before moving ahead:

1. Download and install Postman from `https://www.getpostman.com/`, if you haven't already installed it.
2. Read more about Orchestrator and activity trigger bindings at `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-bindings`.

How to do it...

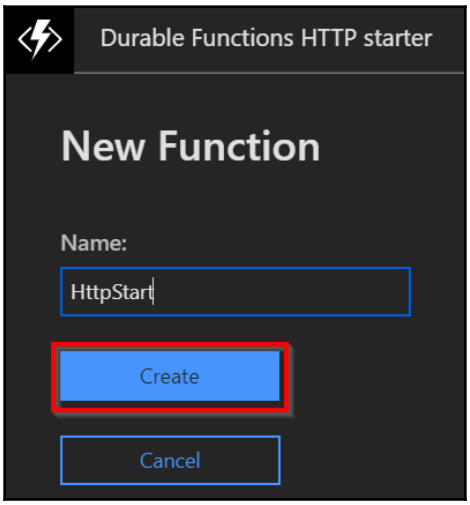
In order to develop Durable Functions, we need to create the following three functions:

- **Orchestrator client**: An Azure Function that can manage Orchestrator instances.
- **Orchestrator function**: The actual Orchestrator function allows the development of stateful workflows via code. This function can asynchronously call other Azure Functions (named **Activity functions**), and can even save the return values of those functions into local variables.
- **Activity functions**: These are the functions that will be called by the orchestrator functions.

Creating an HttpStart function in the Orchestrator client

Perform the following steps:

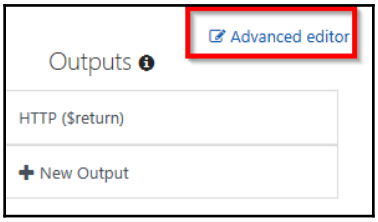
1. Create a new Durable Functions HTTP starter function by choosing **Durable Functions** in the **Scenario** drop-down menu, and click on the **Durable Functions HTTP starter**, which opens a new tab, shown as follows. Let's create a new HTTP function named `HttpStart`:



2. Soon after, you will be taken to the code editor. The following function is a HTTP trigger which accepts the name of the Function that needs to be executed along with the input. It uses the `StartNewAsync` method of the `DurableOrchestrationClient` object to start the Orchestration:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Newtonsoft.Json"

using System.Net;

public static async Task<HttpResponseMessage> Run(
HttpRequestMessage req,
DurableOrchestrationClient starter,
string functionName,
ILogger log)
{
    // Function input comes from the request content.
    dynamic eventData = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName,
    eventData);

    log.LogInformation($"Started orchestration with ID =
'{instanceId}'.");

    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

3. Navigate to the **Integrate** tab and click on **Advanced editor**, as shown in the following screenshot:



4. In the **Advanced editor**, the bindings should be similar to the following. If not, replace the default code with the following code:

```
{
   "bindings":
    [
       {
         "authLevel": "anonymous",
```

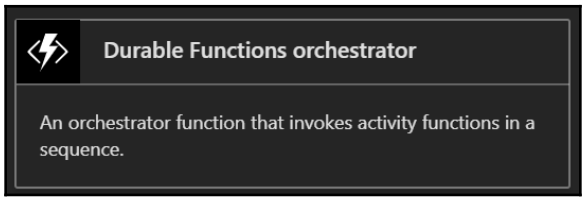
```
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "route": "orchestrators/{functionName}",
      "methods": [
            "post",
            "get"
          ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    },
    {
      "name": "starter",
      "type": "orchestrationClient",
      "direction": "in"
    }
  ]
}
```

The `HttpStart` function works like a gateway for invoking all the functions in the function app. Any request you make using the `https://<durablefunctionname>>.azurewebsites.net/api/orchestrators/{functionName}` URL format will be received by this `HttpStart` function. This function will take care of executing the `Orchestrator` function, based on the parameter available in the `{functionName}` route parameter. All of this is possible with the `route` attribute, defined in `function.json` of the `HttpStart` function.

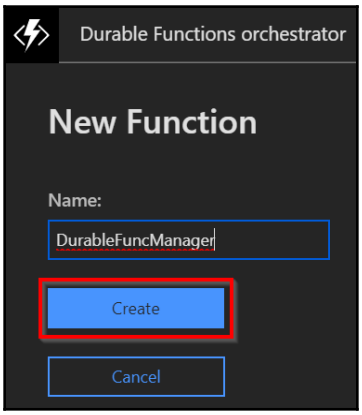
Creating the Orchestrator function

Perform the following steps:

1. Let's create an Orchestrator function by clicking on the **Durable Functions orchestrator** template, shown as follows:



2. Once you click on the **Durable Functions orchestrator** tile, you will be taken to the following tab where you provide the name of the function. Once you provide the name, click on the **Create** button to create the Orchestrator function:



3. In the `DurableFuncManager`, replace the default code with the following, and click on the **Save** button to save the changes: The following Orchestrator will call the Activity functions using the `CallActivityAsync` method of the `DurableOrchestraionContext` object:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
public static async Task<List<string>>
 Run(DurableOrchestrationContext context)
{
   var outputs = new List<string>();
   outputs.Add(await context.CallActivityAsync<string>
    ("ConveyGreeting", "Welcome Cookbook Readers"));
   return outputs;
}
```

4. In the **Advanced editor** of the **Integrate** tab, replace the default code with the following code:

```
{
   "bindings": [
   {
      "name": "context",
      "type": "orchestrationTrigger",
      "direction": "in"
   }
 ]
}
```

Creating an activity function

Perform the following steps:

1. Create a new function named `ConveyGreeting` using the **Durable Functions activity** template:

2. Replace the default code with the following code that just displays the name which is provided as input, and then click on the **Save** button to save the changes:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
public static string Run(string name)
{
   return $"Hello Welcome Cookbook Readers!";
}
```

3. In the **Advanced editor** of the **Integrate** tab, replace the default code with the following code:

```
{
   "bindings": [
   {
      "name": "name",
      "type": "activityTrigger",
      "direction": "in"
   }
   ]
}
```

In this recipe, we have created an Orchestration client, an Orchestrator function, and an activity function. We will learn how to test these in our next recipe.

How it works...

Let us take a look at the working of the recipe:

- We first developed the Orchestrator client (in our case, `HttpStart`), which is capable of creating Orchestrators using the `StartNewAsync` function of the `DurableOrchestrationClient` class. This method creates a new Orchestrator instance.
- Next, we developed the Orchestrator function—the most crucial part of Durable Functions. The following are a few of the most important core features of the Orchestrator context:
    - It can invoke multiple activity functions
    - It can save the output returned by an activity function and pass it to another activity function.
    - These Orchestrator functions are also capable of creating checkpoints that save execution points, so that if there is any problem with the VMs, then it can replace or resume service automatically
- And lastly, we developed the activity function, where we write most of the business logic. In our case, it's just returning a simple message.

There's more...

Durable Functions are dependent on the Durable Task framework. You can learn more about the Durable Task Framework at `https://github.com/Azure/durabletask`.

Testing and troubleshooting Durable Functions

In previous chapters, we have discussed various ways of testing the Azure Functions. We can test Durable Functions with the same set of tools. However, the approach to testing is entirely different, because of its features and the way it works.

In this recipe, we will learn a few of the essential things that one should be aware of while working with Durable Functions.

Getting ready

Download and install the following if you haven't installed them yet:

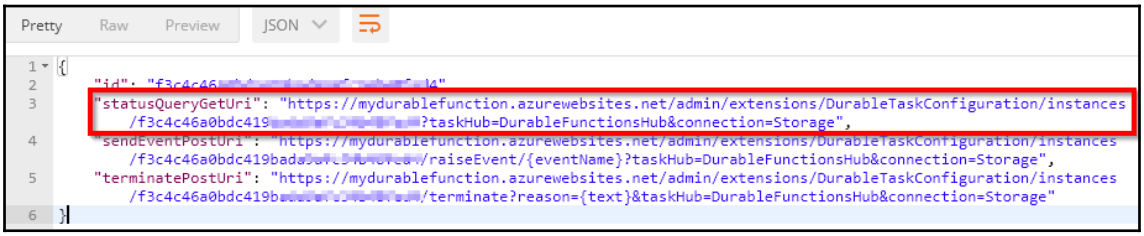
- The Postman tool, available from `https://www.getpostman.com`.
- Microsoft Azure Storage Explorer, available from `http://storageexplorer.com`.

How to do it...

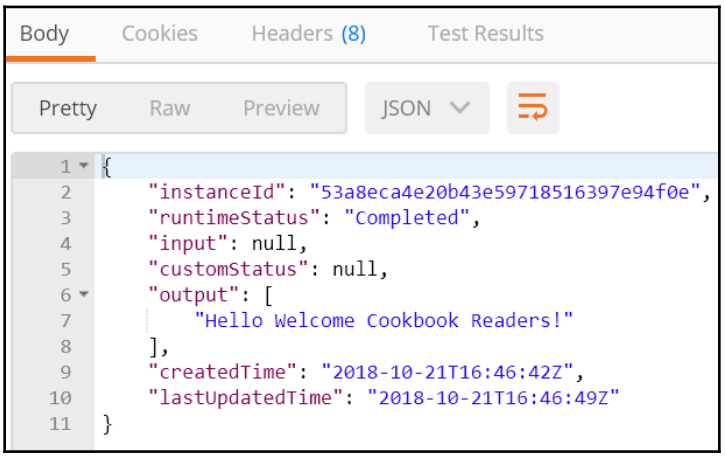
Perform the following steps:

1. Navigate to the code editor of the `HttpStart` function and grab the URL by clicking on **</>Get function URL**. Replace the `{functionName}` template value with `DurableFuncManager`.
2. Let's make a `POST` request using Postman:

3. Once you click on the **Send** button, you will get a response with the following:
    - The instance ID
    - The URL for retrieving the status of the function
    - The URL to send an event to the function
    - The URL to terminate the request



4. Click on `statusQueryGetUri` in the preceding step to view the status of the function. Clicking on the link in the preceding step will open the query in a new tab within the Postman tool. Once the new tab is opened, click on the **Send** button to get the actual output:



5. If everything goes well, we can see the `runtimeStatus` as `Completed` in Postman, as shown in the preceding screenshot. You will also get eight records in the table storage, where the execution history is stored, shown as follows:

EventType	ExecutionId
OrchestratorStarted	a426ec4dabe44527a6aeae14290802c1
ExecutionStarted	a426ec4dabe44527a6aeae14290802c1
TaskScheduled	a426ec4dabe44527a6aeae14290802c1
OrchestratorCompleted	a426ec4dabe44527a6aeae14290802c1
OrchestratorStarted	a426ec4dabe44527a6aeae14290802c1
TaskCompleted	a426ec4dabe44527a6aeae14290802c1
ExecutionCompleted	a426ec4dabe44527a6aeae14290802c1
OrchestratorCompleted	a426ec4dabe44527a6aeae14290802c1

6. If something has gone wrong, you can see the error message in the results column, which tells you in which function the error has occurred. Then, navigate to the **Monitor** tab of that function to see a detailed explanation of the error.

Implementing multithreaded reliable applications using Durable Functions

I have worked in a few of the applications where parallel execution is required to perform some computing tasks. The main advantage of this approach is that you get the desired output pretty quickly, depending on the sub-threads that you create. It could be achieved in multiple ways using different technologies. However, the challenge in these approaches is that, if something goes wrong in the middle of any of the sub-thread, it's not easy to self-heal and resume from where it was stopped. I'm sure many of you might have faced similar problems in your application, as it is a very common business case.

In this recipe, we will implement a simple way of executing a function in parallel with multiple instances using the Durable Functions for the following scenario.

Assume that we have five customers (whose IDs are 1, 2, 3, 4, and 5 respectively) who approached us to generate a huge number of barcodes (say around 50,000). It would take a lot of time to generate the barcodes as it would involve some image processing tasks. So, one simple way to quickly process the request is to use asynchronous programming by creating a thread for each of the customers and then executing the logic in parallel for each of them.

We will also simulate a simple use case to understand how the Durable Functions auto-heals when the VM on which they are hosted goes down or is restarted.

Getting ready

Install the following if you haven't installed them yet:

- The Postman tool, available from `https://www.getpostman.com/`.
- Microsoft Azure Storage Explorer, available from `http://storageexplorer.com/`.

How to do it...

In this recipe, we will create the following Azure Function triggers:

- One Orchestrator function, named `GenerateBARCode`

- Two activity trigger functions, as follows:
    - `GetAllCustomers`: To make it simple, this function just returns the array of customer IDs. In your real-world applications, you would have business logic that decides which customers are eligible and based that logic, you would return the eligible customer Ids.
    - `CreateBARCodeImagesPerCustomer`: This function doesn't actually create the barcode; rather, it just logs a message to the console, as our goal is to understand the features of Durable Functions. For each customer, we will randomly generate a number less than 50,000 and simply iterate through it.

Creating the Orchestrator function

Perform the following steps:

1. Create a new function named `GenerateBARCode` using the **Durable Functions Orchestrator template**. Replace the default code with the following, and click on the **Save** button to save the changes. The following code initially class the `GetAllCustomers` activity function, stores all the customer Ids in an array and then for each customer, it again calls another activity function that returns the number of Bar Codes that are generated. Finally, it waits till the Activity functions for all the customers gets completed an then returns the sum of all the Bar Codes that are generated for all the customers.

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
public static async Task<int> Run(
    DurableOrchestrationContext context)
{
    int[] customers = await
    context.CallActivityAsync<int[]>("GetAllCustomers",null);

    var tasks = new Task<int>[customers.Length];
    for (int nCustomerIndex = 0; nCustomerIndex < customers.Length;
nCustomerIndex++)
    {
      tasks[nCustomerIndex] = context.CallActivityAsync<int>
      ("CreateBARCodeImagesPerCustomer",
customers[nCustomerIndex]);
    }
    await Task.WhenAll(tasks);
    int nTotalItems = tasks.Sum(item => item.Result);
```

```
    return nTotalItems;
}
```

2. In the **Advanced editor** of the **Integrate** tab, replace the default code with the following code:

```
{
   "bindings": [
   {
      "name": "context",
      "type": "orchestrationTrigger",
      "direction": "in"
   }
   ]
}
```

Creating a GetAllCustomers activity function

Perform the following steps:

1. Create a new function named `GetAllCustomers` using the **Durable Functions Activity template**, replace the default code with the following code, and then click on the **Save** button to save the changes:

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
public static int[] Run(string name)
{
   int[] customers = new int[]{1,2,3,4,5};
   return customers;
}
```

2. In the **Advanced editor** of the **Integrate** tab, replace the default code with the following code:

```
{
   "bindings": [
   {
      "name": "name",
      "type": "activityTrigger",
      "direction": "in"
   }
   ]
}
```

Creating a CreateBARCodeImagesPerCustomer activity function

Perform the following steps:

1. Create a new function named `CreateBARCodeImagesPerCustomer` using the **Durable Functions Activity template**. Replace the default code with the following, and then click on the **Save** button to save the changes.

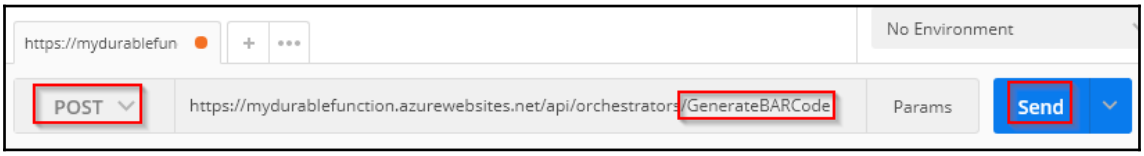
```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Microsoft.WindowsAzure.Storage"
using Microsoft.WindowsAzure.Storage.Blob;

public static async Task<int> Run(DurableActivityContext
 customerContext,ILogger log)
{
   int ncustomerId = Convert.ToInt32
    (customerContext.GetInput<string>());
   Random objRandom = new Random(Guid.NewGuid().GetHashCode());
   int nRandomValue = objRandom.Next(50000);
   for(int nProcessIndex = 0;nProcessIndex<=nRandomValue;
    nProcessIndex++)
   {
      log.LogInformation($" running for {nProcessIndex}");
   }
   return nRandomValue;
}
```

2. In the **Advanced editor** of the **Integrate** tab, replace the default code with the following code:

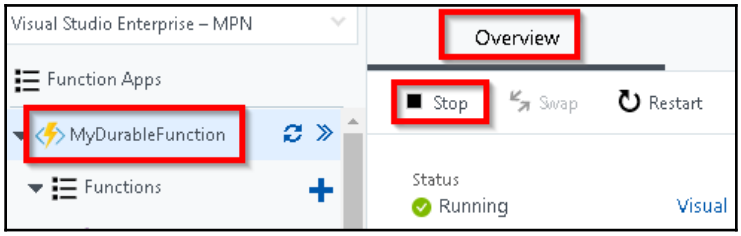
```
{
   "bindings": [
   {
      "name": "customerContext",
      "type": "activityTrigger",
      "direction": "in"
   }
   ]
}
```

3. Let's run the function using Postman. We will be stopping the App Service to simulate a restart of the VM where the function would be running, and to see how the Durable Function resumes from where it was paused.

4. Make a `POST` request using Postman, as shown as follows:



5. Once you click on the **Send** button, you will get a response with the status URL. Click on `statusQueryGetURi` to view the status of the function. Clicking on the `statusQueryGetURi` link will open it in a new tab within the Postman tool. Once the new tab is opened, click on the **Send** button to get the progress of the function.
6. While the function is running, let's navigate to the function app's **Overview** blade and stop the service by clicking on the **Stop** button:



7. The execution of the function will be stopped in the middle. Let's navigate to our storage account in Storage Explorer, and open the **DurableFunctionsHubHistory** table to see how much progress has been made:

EventType	ExecutionId	IsPlayed	_Timestamp	Input	Name
OrchestratorStarted	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:34:56.005Z		
ExecutionStarted	cf36ba2c05344390896fe2dcbb898189	true	2018-01-19T16:34:46.159Z	null	GenerateBARCode
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:35:06.009Z		GetAllCustomers
OrchestratorCompleted	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:35:06.010Z		

8. After some time—in my case, after just five minutes—go back to the **Overview** blade and start the function app service. You will notice that the Durable Function will resume from where it had stopped. We didn't write any code for this; it's an out-of-the-box feature. The following is the screenshot of the completed function:

EventType	ExecutionId	IsPlayed	_Timestamp	Input	Name
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:42:23.523Z		CreateBARCodeImagesPerCustomer
OrchestratorStarted	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:34:56.005Z		
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:35:06.009Z		GetAllCustomers
OrchestratorCompleted	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:35:06.010Z		
OrchestratorStarted	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:41:51.507Z		
TaskCompleted	cf36ba2c05344390896fe2dcbb898189	true	2018-01-19T16:41:50.516Z		
ExecutionStarted	cf36ba2c05344390896fe2dcbb898189	true	2018-01-19T16:34:46.159Z	null	GenerateBARCode
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:42:23.523Z		CreateBARCodeImagesPerCustomer
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:42:23.523Z		CreateBARCodeImagesPerCustomer
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:42:23.523Z		CreateBARCodeImagesPerCustomer
TaskScheduled	cf36ba2c05344390896fe2dcbb898189	false	2018-01-19T16:42:23.523Z		CreateBARCodeImagesPerCustomer

How it works...

Durable Functions allow us to develop reliable execution of our functions, which means that even if the VMs crashed or are restarted while the function is running, it automatically resumes back to its previous state automatically. It does so with the help of something called **checkpointing** and **replaying**, where the history of the execution is stored in the storage table.

You can learn more about this feature at `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-checkpointing-and-replay`.

There's more...

- In case you get a 404 Not Found response when you run the `statusQueryGetURi` URL, don't worry. It will take some time, but it will eventually work when you make a request later on.
- In order to view the execution history of your Durable Functions, navigate to the `DurableFunctionsHubHistory` table, which is located in the storage account that was created while creating the function app:

You can find the storage account name in the **Application settings**, as shown in the preceding screenshot.

8
Bulk Import of Data Using Azure Durable Functions and Cosmos DB

In this chapter, we will learn the following recipes:

- Uploading employee data into Blob Storage
- Creating a Blob trigger
- Creating the Durable Orchestrator and triggering it for each Excel import
- Reading Excel data using activity functions
- Auto-scaling Cosmos DB throughput
- Bulk inserting data into Cosmos DB

Introduction

In this chapter, we will develop a mini-project by taking a very common use case that solves the business problem of sharing data across different applications using Excel. We will use Durable Functions, which is an extension to Azure Functions that lets you write workflows by writing the minimum lines of code.

Here are the two core features of Durable Functions that we will be using in the recipes of this chapter:

- **Orchestrator**: Orchestrator is a function that is responsible for managing all activity triggers. It can be treated as a workflow manager that has multiple steps. Orchestrator is responsible for initiating the activity trigger, passing inputs to the activity trigger, getting the output, maintaining the state, and then passing the output of one activity trigger to another if required.
- **Activity trigger**: Each activity trigger can be treated as a workflow step that performs a function.

You can learn more about Durable Functions at `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-overview`.

Business problem

In general, every organization will definitely be using various applications hosted in multiple platforms across different data centers (either cloud or on-premises). Often, there will be requirements where the data from one application needs to be fed to another system. Usually, Excel spreadsheets (or in some cases, JSON or XML files) are used for exporting data from one application and importing it into another application.

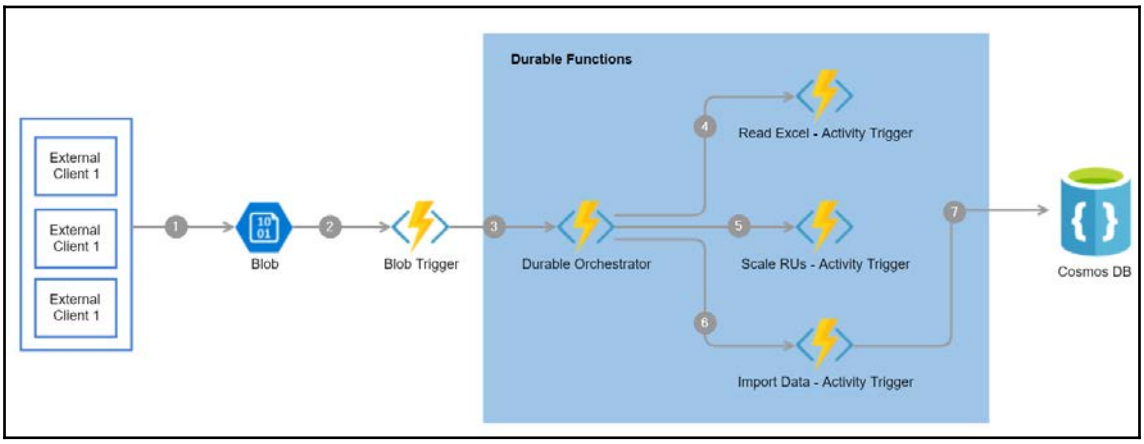
You might think that exporting an Excel file from one application to another would be an easy job, but if there are many applications that need to feed data to other applications, and on a weekly/monthly basis, then it would become very tedious and there is lot of scope for manual error. So, obviously the solution is to automate the process to the highest possible extent.

In this chapter, we will learn how to develop a durable solution based on serverless architecture using Durable Functions. If you have already read `Chapter 7`, *Developing Reliable Serverless Applications Using Durable Functions*, then you might have some basic knowledge of what Durable Functions are and how they work. In `Chapter 7`, *Developing Reliable Serverless Applications Using Durable Functions*, we implemented the solution from the portal. However, in this chapter, we will implement a mini-project using Visual Studio 2017 (preferably 15.5 or higher).

Before we start developing the project, let's try to understand the new serverless way of implementing the solution.

Durable serverless way of implementing an Excel import

This diagram shows all the steps required for building the solution using the serverless architecture:



1. External clients or applications upload an Excel file to Blob Storage
2. **Blob Trigger** gets triggered after the Excel file is uploaded successfully
3. Durable Orchestrator is started from **Blob Trigger**
4. Orchestrator invokes **Read Excel - Activity Trigger** to read the Excel content from Blob Storage
5. Orchestrator invokes **Scale RUs - Activity Trigger** to scale up the **Cosmos DB** collection's throughput so that it can accommodate the load
6. Orchestrator invokes **Import Data - Activity Trigger** to prepare the collection to bulk import data
7. Finally, **Import Data - Activity Trigger** loads the collection data into **Cosmos DB** collection using Cosmos DB output bindings

Uploading employee data into Blob Storage

In this recipe, we will develop a console application that is responsible for uploading the Excel sheet to Blob Storage.

Getting ready

Perform the following prerequisites:

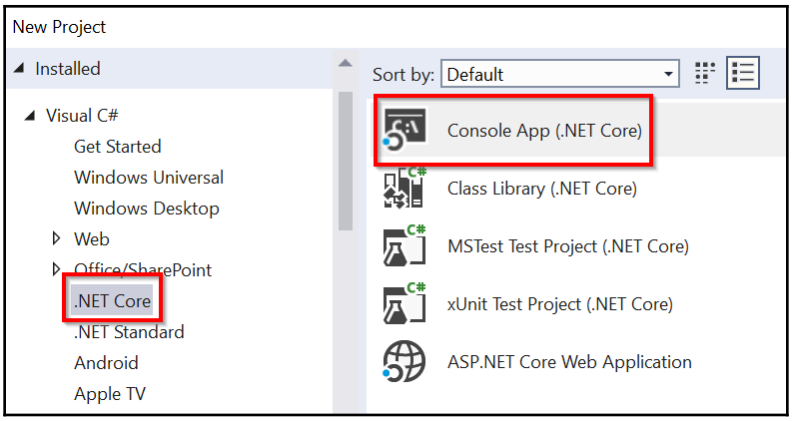
1. Install Visual Studio 2017 Version 15.5 or higher.
2. Create a storage account and create a blob container with the name `Excelimports`.
3. Create an Excel file with some employee data as shown in the following screenshot:

Emp Id	Name	Email	PhoneNumber
1	Vijay	Vijay@vijay.com	1111111111
2	Vivek	Vivek@vivek.com	2222222222
3	Vineesha	vineesha@vineesha.com	3333333333
4	Praveen	Praveen@praveen.com	4444444444
5	Nishit	nishit@nishit.com	5555555555
6	Ragi	ragi@ragi.com	6666666666
7	Uma	uma@uma.com	7777777777
8	Harsha	harsha@harsha.com	8888888888

How to do it...

Perform the following steps:

1. Create a new console app named `ExcelImport.Client` using Visual Studio, as shown in the following screenshot:



2. Once the project is created, execute the following commands in the NuGet package manager:

```
Install-Package Microsoft.Azure.Storage.Blob
Install-Package Microsoft.Extensions.Configuration
Install-Package Microsoft.Extensions.Configuration.FileExtensions
Install-Package Microsoft.Extensions.Configuration.Json
```

3. Add the following namespaces at the top of the `Program.cs` file:

```
using Microsoft.Extensions.Configuration;
 using Microsoft.WindowsAzure.Storage;
 using Microsoft.WindowsAzure.Storage.Blob;
 using System;
 using System.IO;
 using System.Threading.Tasks;
```

4. The next step is to develop the code in a function named `UploadBlob` that uploads the Excel file into the blob container that we have created. For the sake of simplicity, the following code uploads the Excel file from a hardcoded location. However, in a typical real-time application, this file would be uploaded by the end user via a web interface. Copy the following code and paste it in the `Program.cs` file of the `ExcelImport.Client` application:

```
private static async Task UploadBlob()
 {
 var builder = new ConfigurationBuilder()
 .SetBasePath(Directory.GetCurrentDirectory())
 .AddJsonFile("appsettings.json", optional: true, reloadOnChange:
true);
 IConfigurationRoot configuration = builder.Build();
CloudStorageAccount cloudStorageAccount =
```

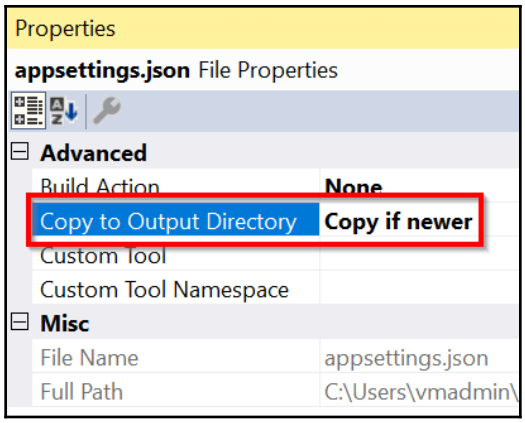
```
CloudStorageAccount.Parse(configuration.GetConnectionString("Storag
eConnection"));
CloudBlobClient cloudBlobClient =
cloudStorageAccount.CreateCloudBlobClient();
CloudBlobContainer ExcelBlobContainer =
cloudBlobClient.GetContainerReference("Excelimports");
 await ExcelBlobContainer.CreateIfNotExistsAsync();
CloudBlockBlob cloudBlockBlob =
ExcelBlobContainer.GetBlockBlobReference("EmployeeInformation" +
Guid.NewGuid().ToString());
await
cloudBlockBlob.UploadFromFileAsync(@"C:\Users\vmadmin\source\repos\
POC\ImportExcelPOC\ImportExcelPOC\EmployeeInformation.xlsx");
}
```

5. Now, copy the following code to the `Main` function. This piece of code just invokes the `UploadBlob` function, which internally is responsible for uploading the blob:

```
 {
 UploadBlob().Wait();
 }
 catch (Exception ex)
 {
 Console.WriteLine("An Error has occurred with the message" +
ex.Message);
 }
```

6. The next step is to create a configuration file named `appsettings.json`, which contains the storage account's connection string, as shown in the following screenshot:

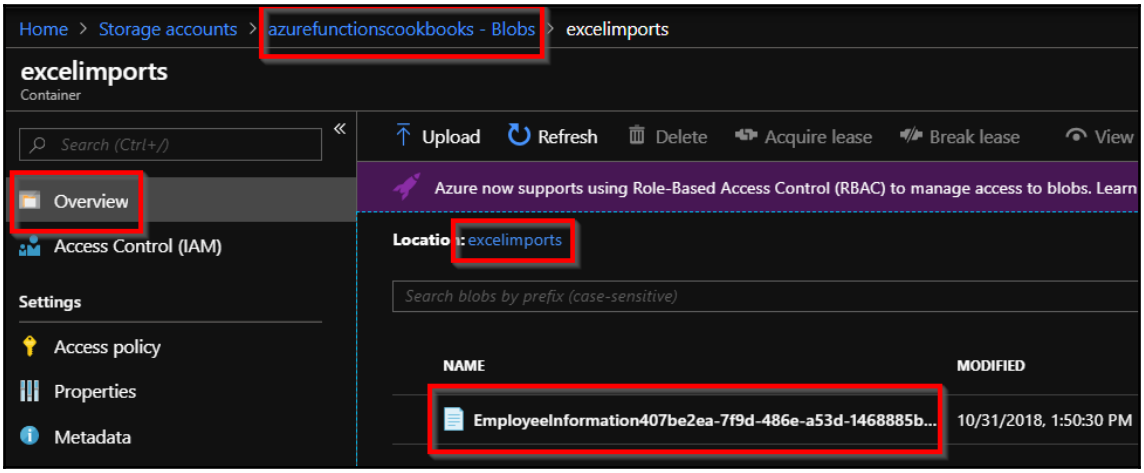
7. Go to the properties of the `appsettings.json` file and change **Copy to Output Directory** to the **Copy if newer** option, so that the properties can by read by the program as shown in the following screenshot:



8. Now, build the application and execute it. If you have configured everything, then you should see something as shown in the following screenshot:

```
Successfully Uploaded.
Press any key to continue . . .
```

9. Let's navigate to the storage account and go to the blob container named `Excelimports`, where you should see the Excel file that we have uploaded, as shown in the following screenshot:



That's it. We have developed an application that is responsible for uploading the blob.

How it works...

In this recipe, we have created a console application that uses storage assemblies to upload a blob (in our case, it is just an Excel file) to the designated blob container. Note that every time the application runs, a new file will get created in the blob container. In order to upload the Excel files with unique names, we are appending a GUID.

There's more...

Make a note of the naming conventions that should be followed while creating the blob container:

At the time of writing, this is the error message that the portal throws if you don't adhere the naming rules: **This name may only contain lowercase letters, numbers, and hyphens, and must begin with a letter or a number. Each hyphen must be preceded and followed by a non-hyphen character. The name must also be between 3 and 63 characters long**.

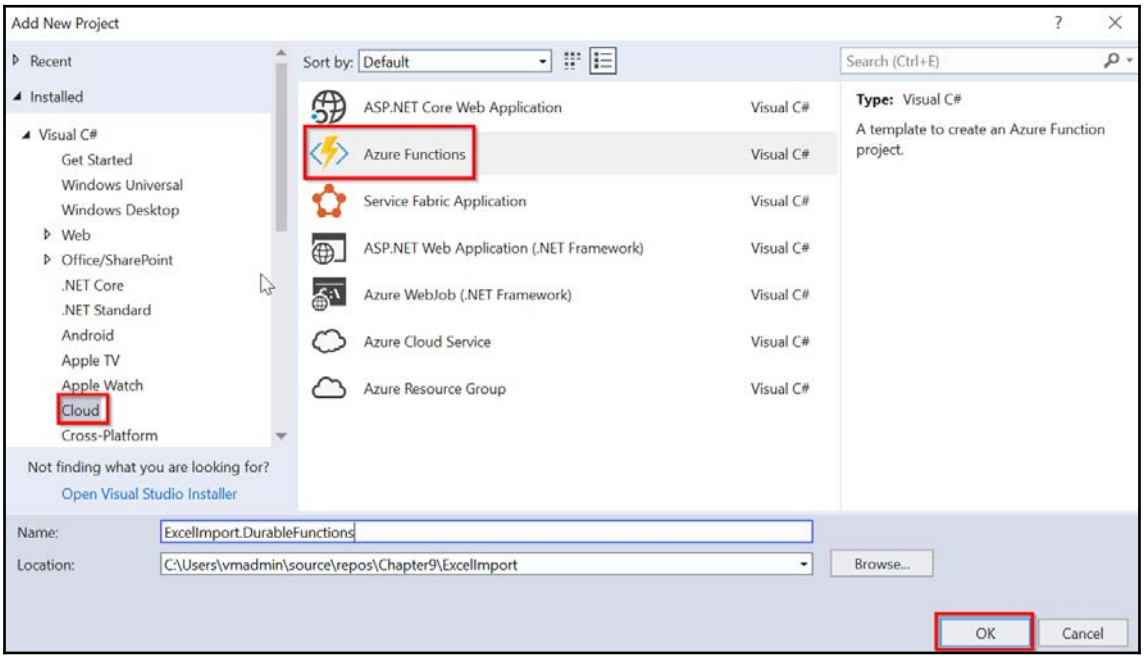
Creating a Blob trigger

In this recipe, we will create a function app with the Azure Functions V2 runtime and learn how to create a Blob trigger using Visual Studio, and we will also see how the Blob trigger gets triggered when the Excel file is uploaded successfully to the blob container.

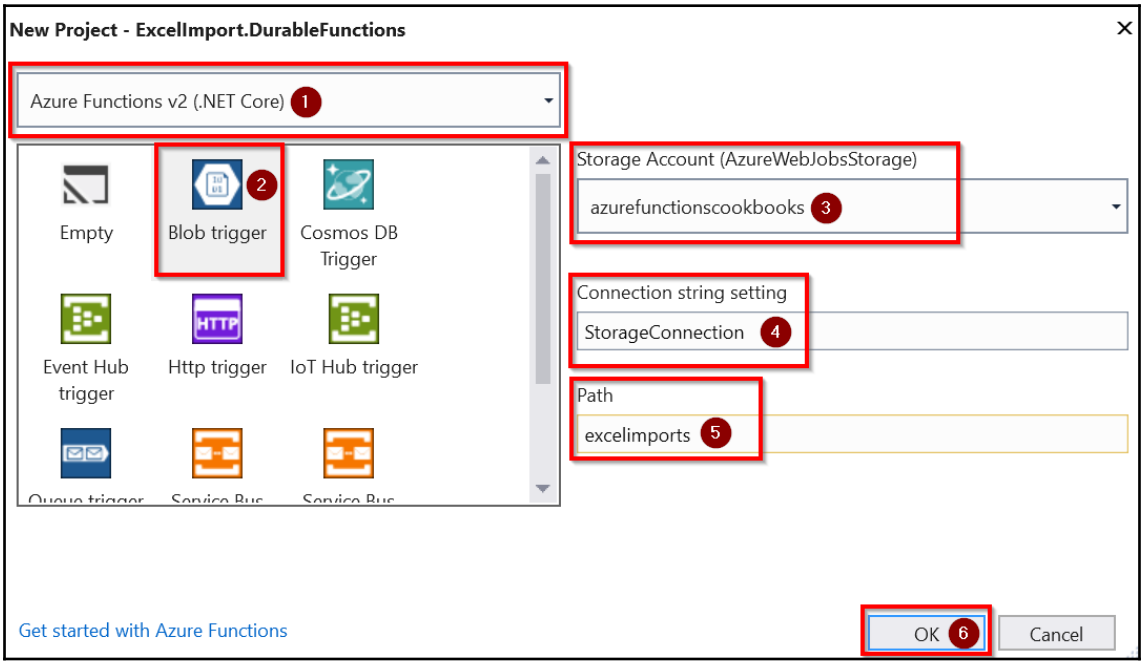
Getting ready

Perform the following prerequisites:

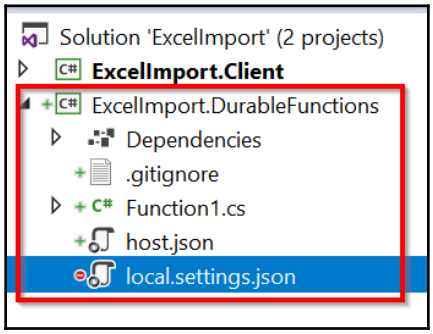
1. Add a new project named `ExcelImport.DurableFunctions` to the existing solution by choosing the **Azure Functions** template, as shown in the following screenshot:



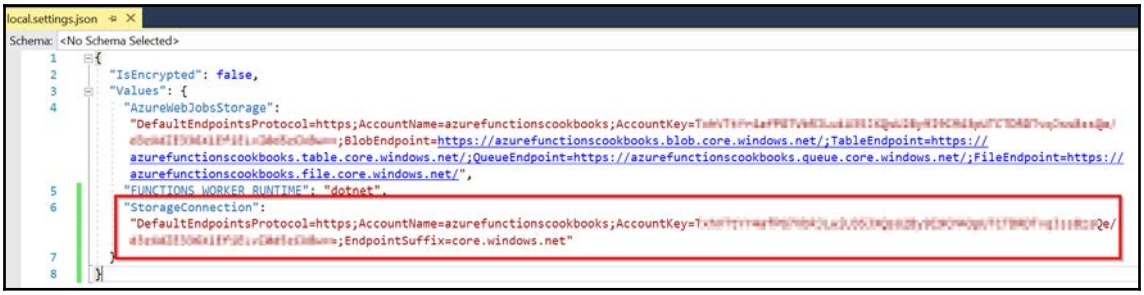
2. The next step is to choose the Azure Functions runtime as well as the trigger. Choose **Azure Functions v2 (.NET Core)**, choose **Blob trigger**, and provide the following:
    - **Storage Account (AzureWebJobsStorage)**: This is the name of the storage account in which our blob container resides
    - **Connection string setting**: This is the connection string key name that refers to the storage account
    - **Path**: This is the name of the blob container where the Excel files are being uploaded



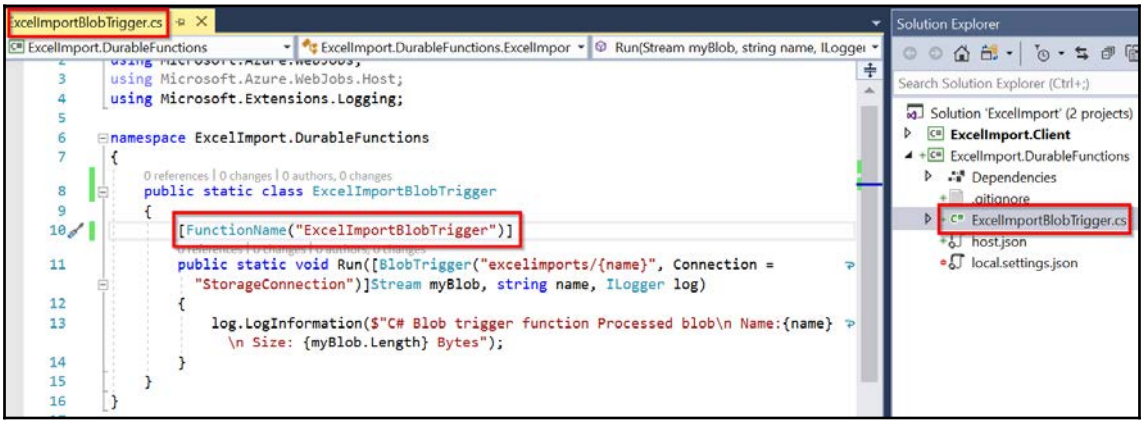
3. When you create the project, the structure should look something like the following screenshot:



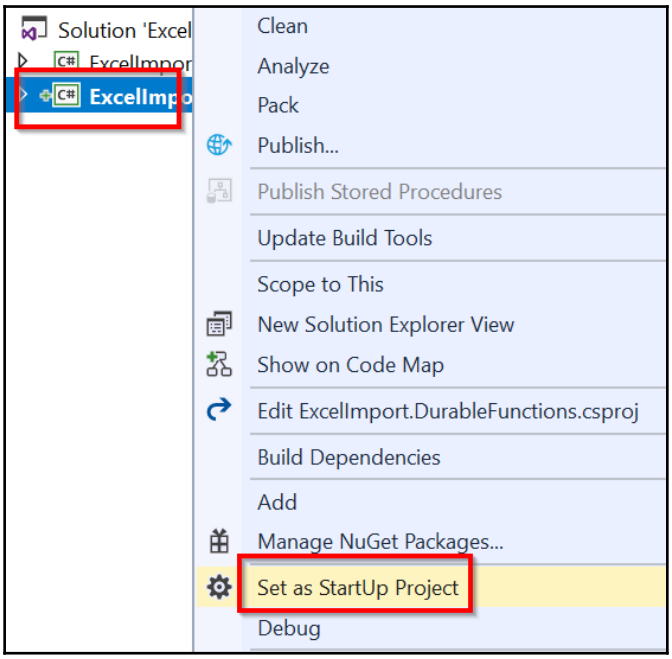
4. Let's add a connection string with the name `StorageConnection` (remember, we have used this in the connection string setting file in one of our earlier steps) to `local.settings.json`, as shown in the following screenshot:



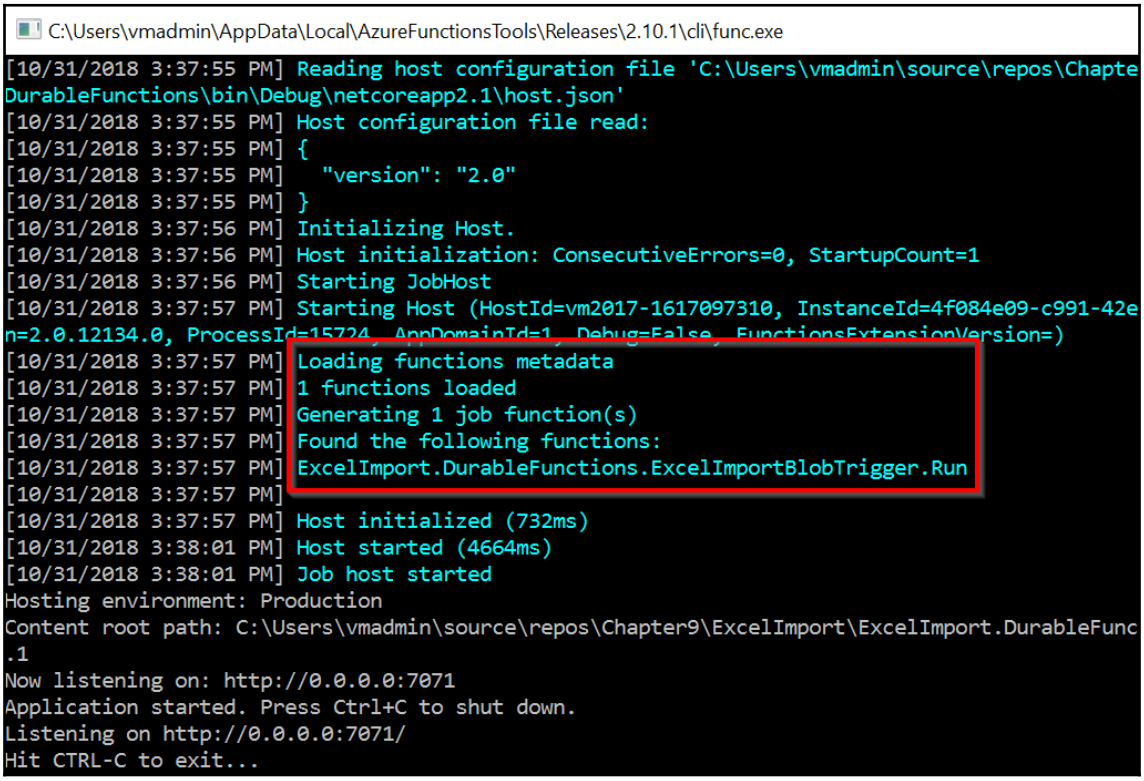
5. Now, open the `Function1.cs` file and rename it to `ExcelImportBlobTrigger`, and also replace `Function1` (the name of the function) with `ExcelImportBlobTrigger` (line 10), as shown in the following screenshot:



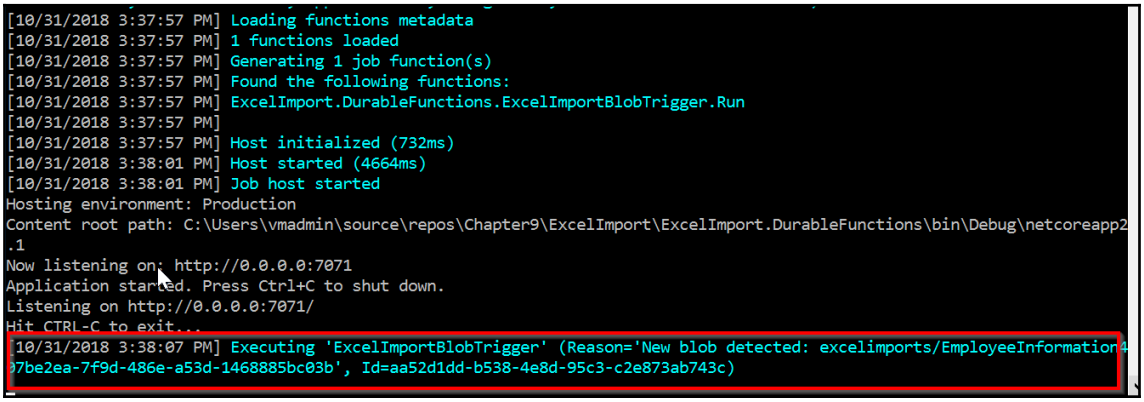
6. Configure `ExcelImport.DurableFunctions` as the default project, as shown in the following screenshot:



7. Create a breakpoint in `ExcelImportBlobTrigger` and run the application by pressing the *F5* key. If everything is configured properly, you should see the console as follows:



8. Let's now upload a new file by running the `ExcelImport.Client` application. Immediately after the file is uploaded, the Blob trigger will be fired, as shown in the following screenshot. Your breakpoints should also be hit along with this:



We are done with creating the Blob trigger that gets fired whenever a new blob is added to the blob container.

How to do it...

In this recipe, we have created a new function app based on the Azure Functions V2 runtime, which is based on the .NET Core framework and can run on all platforms that support .NET Core (such as Windows and Linux OSes). We have also created a Blob trigger and configured it to run when a new Blob is added by configuring the connection string setting. We have also created a `local.setting.json` configuration file to store the config values that are used in local development. After we created the Blob trigger, we ran the `ExcelImport.Client` application to upload a file to validate that the Blob trigger is getting executed.

There's more...

All the configurations will be taken from the `local.settings.json` file while you are running the functions in your local environment. However, when you deploy the functions to Azure, all the configurations items (such as connection string and app settings) will be referenced from the **Application Settings** of your function app. Make sure that you create all the configuration items in the function app after you deploy the functions.

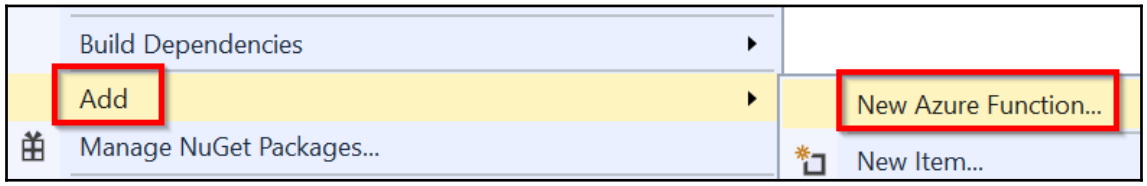
Creating the Durable Orchestrator and triggering it for each Excel import

This recipe is one of the most important and interesting ones. We will learn how to create the Durable Orchestrator responsible for managing all the activity functions that we create for the different individual tasks required to complete the `ExcelImport` project.

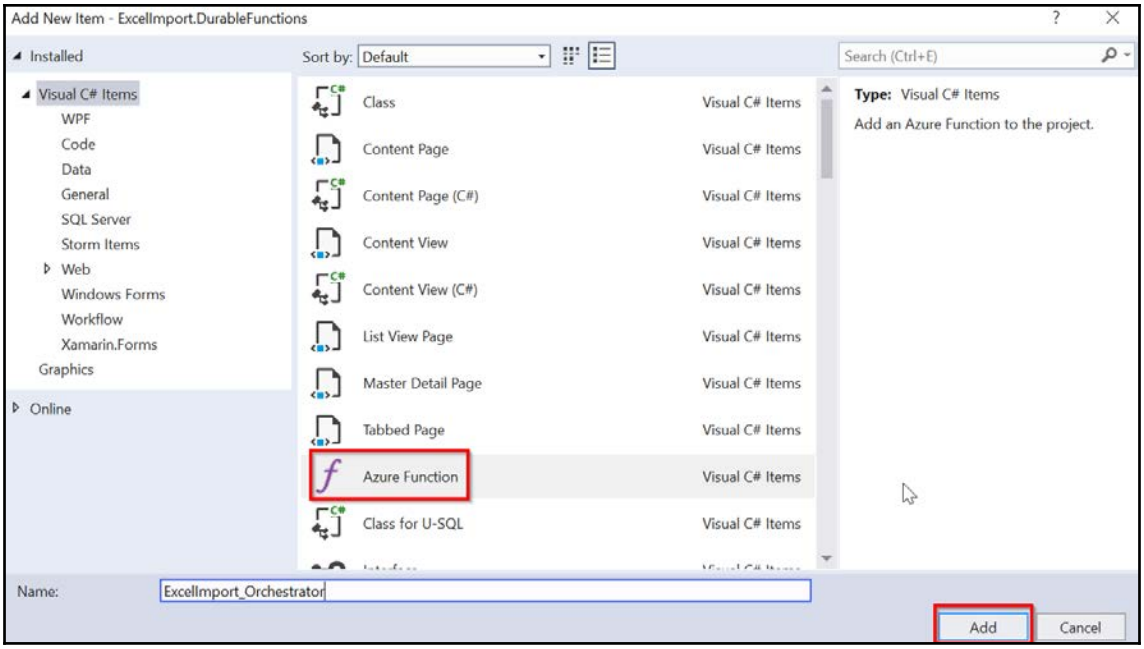
How to do it...

Perform the following steps:

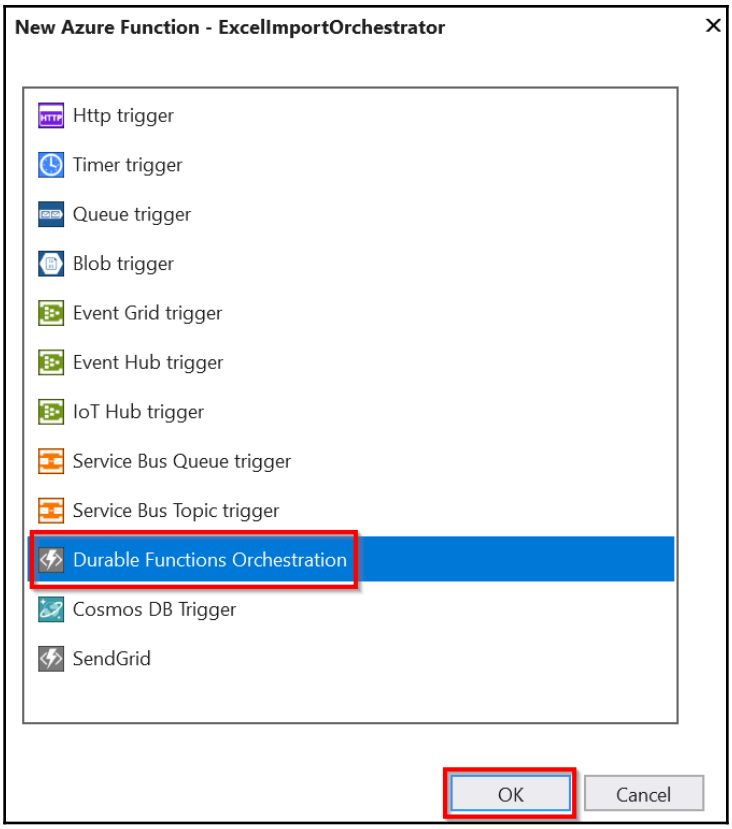
1. Create a new function by right-clicking on `ExcelImport.DurableFunctions`, click on **Add**, and then choose **New Azure Function**, as shown in the following screenshot:



2. In the **Add new Item** popup, choose **Azure Function**, provide the name `ExcelImport_Orchestrator`, and click on **Add**, as shown in the following screenshot:



3. In the **New Azure Function** popup, select the **Durable Function Orchestration** template and click on the **OK** button, which creates the following:
    - `HttpStart`: This is the Durable Function's starter function (an HTTP trigger), which works as a client that can invoke the Durable Orchestrator. However, in our project, we will not be using this HTTP Trigger; we will be using the logic inside it in our `ExcelImportBlobTrigger` Blob trigger to invoke the Durable Orchestrator.
    - `RunOrchestrator`: This is the actual durable Orchestrator that is capable of invoking and managing the activity functions.
    - `SayHello`: A simple activity function. We will create a few activity functions. Let's go ahead and remove this method:

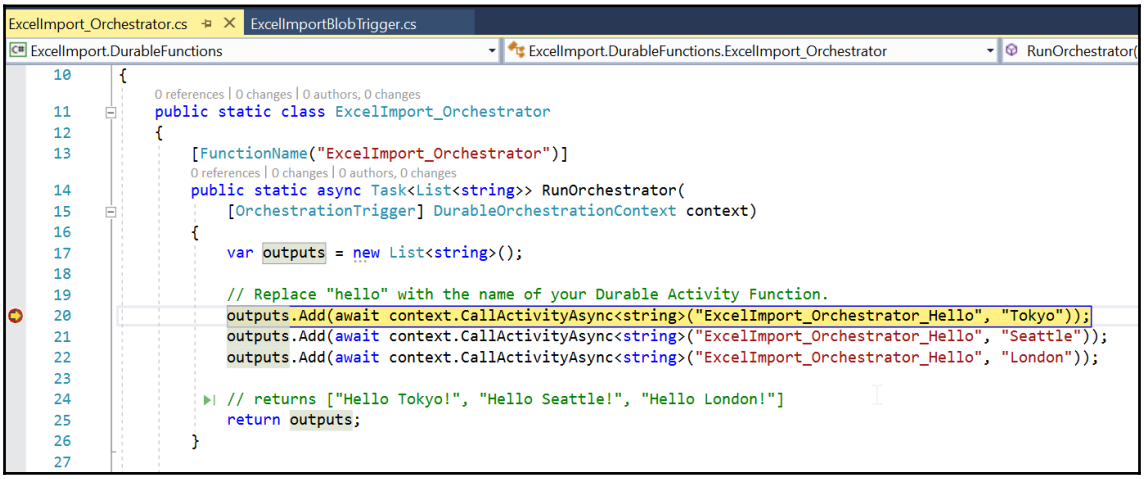


4. In the `ExcelImportBlobTrigger` Blob trigger, let's make the following code changes to invoke the Orchestrator:
    - Decorate the function to be async
    - Add the Orchestration client output bindings
    - Call `StartNewAsync` using the `DurableOrchestrationClient`
5. The code in the `ExcelImportBlobTrigger` function should look like the following after making these changes:

```
using System.IO;
 using Microsoft.Azure.WebJobs;
 using Microsoft.Azure.WebJobs.Host;
 using Microsoft.Extensions.Logging;
namespace ExcelImport.DurableFunctions
 {
 public static class ExcelImportBlobTrigger
 {
```

```
 [FunctionName("ExcelImportBlobTrigger")]
 public static async void Run(
 [BlobTrigger("Excelimports/{name}", Connection =
"StorageConnection")]Stream myBlob,
 string name,
 [OrchestrationClient]DurableOrchestrationClient starter,
 ILogger log)
 {
 string instanceId = await
starter.StartNewAsync("ExcelImport_Orchestrator", name);
log.LogInformation($"C# Blob trigger function Processed blob\n
Name:{name} \n Size: {myBlob.Length} Bytes");
}
 }
 }
```

6. Create a breakpoint in the `ExcelImport_Orchestrator` Orchestrator function and run the application by pressing *F5*.
7. Let's now upload a new file (while `ExcelImport.DurableFunctions` is running) by running the `ExcelImport.Client` function. (You can also directly upload the Excel file from the Azure portal.) After the file is uploaded, in just a few moments the breakpoint in the `ExcelImport_Orchestrator` function should be hit, as shown in the following screenshot:



We have learned how to invoke the Durable Orchestration function from the Blob trigger.

How it works...

We started the recipe by creating the Orchestration function and then we made changes to the `ExcelImportBlobTrigger` Blob trigger by adding the `OrchestratonClient` output bindings to invoke the Durable Orchestrator function.

When you create a new Orchestration function, it creates a few activity functions. In the next recipes, we will remove them and create new activity functions for our requirements.

There's more...

In this recipe, we have used `DurableOrchestrationClient`, which understands how to start and terminate Durable Orchestrations.

Here are a few of the important operations that are supported:

- Start an instance using the `StartNewAsync` method
- Terminate an instance using the `TerminateAsync` method
- Query the status of the currently running instance using the `GetStatusAsync` method
- It can also raise an event to the instance to update about any external event using the `RaiseEventAsync` method

You can learn more about these at `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-instance-management#sending-events-to-instances`.

Reading Excel data using activity functions

In this recipe, we will retrieve all data from specific Excel sheets by writing an activity function.

Let's now make some code changes to the Orchestration function by writing a new activity function that can read data from an Excel sheet that is uploaded to the blob container.

Getting ready

In this recipe, we will create the activity trigger named `ReadExcel_AT` function that reads the data from the blob stored in the storage account. This activity trigger performs the following jobs:

1. Connects to the blob using a function, `ReadBlob`, of a class named `StorageManager`.
2. Reads the data from the Excel using a component called **EPPlus**. You can read more about it at `https://github.com/JanKallman/EPPlus`.
3. Returns the data from the Excel file as a collection of employee objects.

Next, install the following NuGet packages in the `ExcelImport.DurableFunctions` project:

```
Install-Package WindowsAzure.Storage
Install-Package EPPlus
```

How to do it...

If you think of Durable Functions as a workflow, then the activity trigger function can be treated a workflow step that takes some kind of optional input, performs some functionality, and returns an optional output. It's one of the core concepts of Azure Durable Functions.

You can learn more about the Activity Trigger at `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-types-features-overview`.

Before we start creating the activity trigger function, let's first build the dependency functions.

Read data from Blob Storage

Perform the following steps:

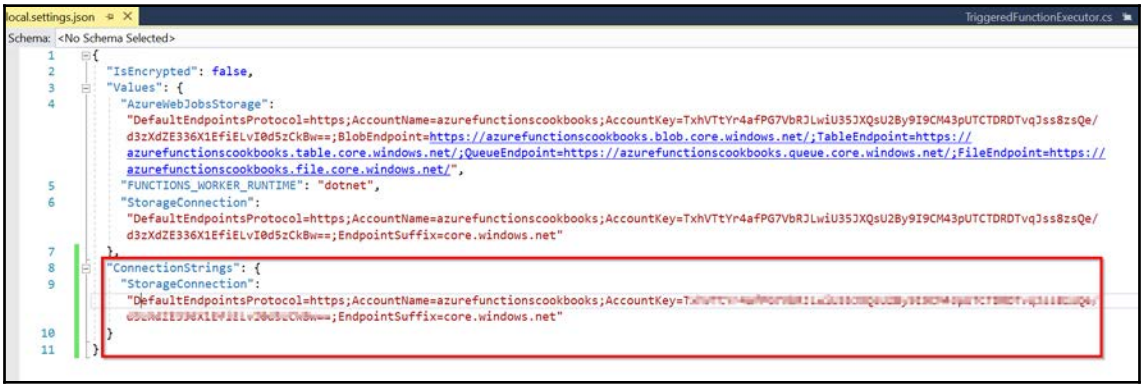
1. Create a class named `StorageManager` and paste in the following code. This code connects to the specified storage account, reads the data from the blobs, and returns a `Stream` object to the caller function:

```
class StorageManager
 {
 public async Task<Stream> ReadBlob(string BlobName)
 {
 var builder = new ConfigurationBuilder()
 .SetBasePath(Directory.GetCurrentDirectory())
 .AddJsonFile("local.settings.json", optional: true,
reloadOnChange: true);
 IConfigurationRoot configuration = builder.Build();
CloudStorageAccount cloudStorageAccount =
CloudStorageAccount.Parse(configuration.GetConnectionString("Storag
eConnection"));
CloudBlobClient cloudBlobClient =
cloudStorageAccount.CreateCloudBlobClient();
CloudBlobContainer ExcelBlobContainer =
cloudBlobClient.GetContainerReference("Excel");
 CloudBlockBlob cloudBlockBlob =
ExcelBlobContainer.GetBlockBlobReference(BlobName);
return await cloudBlockBlob.OpenReadAsync();
}
 }
```

2. Paste the following namespace references into the `StorageManager` class:

```
using Microsoft.Extensions.Configuration;
 using Microsoft.WindowsAzure.Storage;
 using Microsoft.WindowsAzure.Storage.Blob;
 using System.IO;
 using System.Threading.Tasks;
```

3. Finally, add a connection string (if it's not been done already) of the storage account to the `local.settings.json` file, as shown here:



Read Excel data from the stream

Perform the following steps:

1. Create a class named `EPPLusExcelManager` and paste the following code. This class has a method named `ReadExcelData`, which uses a library named `EPPlus` to read the data from the Excel file (`.xlsx` extension). It reads each row, creates an `Employee` object for each row, and then returns an employee collection. We will create the `Employee` class in a moment:

```
class EPPLusExcelManager
 {
 public List<Employee> ReadExcelData(Stream stream)
 {
 List<Employee> employees = new List<Employee>();
 //FileInfo existingFile = new
FileInfo("EmployeeInformation.xlsx");
 using (ExcelPackage package = new ExcelPackage(stream))
 {
 ExcelWorksheet ExcelWorksheet = package.Workbook.Worksheets[0];
 for (int EmployeeIndex = 2; EmployeeIndex <
ExcelWorksheet.Dimension.Rows + 1; EmployeeIndex++)
 {
 employees.Add(new Employee()
 {
 EmpId = Convert.ToString(ExcelWorksheet.Cells[EmployeeIndex,
1].Value),
 Name = Convert.ToString(ExcelWorksheet.Cells[EmployeeIndex,
```

```
2].Value),
 Email = Convert.ToString(ExcelWorksheet.Cells[EmployeeIndex,
3].Value),
 PhoneNumber = Convert.ToString(ExcelWorksheet.Cells[EmployeeIndex,
4].Value)
 });
 }
 }
 return employees;
 }
 }
```

2. Now, let's create another class named `Employee` and copy the following code:

```
public class Employee
 {
 public string EmpId { get; set; }
 public string Name { get; set; }
 public string Email { get; set; }
 public string PhoneNumber { get; set; }
 }
```

If you build the application now, you should not see any errors. We are done with developing the dependencies for our first activity trigger function. Let's now start building the actual activity trigger.

Create the activity function

Perform the following steps:

1. Create a new activity function named `ReadExcel_AT` that connects to the blob using the `StorageManager` class that we developed in the previous section, and then reads the data using the `EPPLusExcelManager` class. Copy the following code to the `ExcelImport_Orchestrator` class:

```
[FunctionName("ReadExcel_AT")]
 public static async Task<List<Employee>> ReadExcel_AT(
 [ActivityTrigger] string name,
 ILogger log)
 {
 log.LogInformation("Orchestration started");
StorageManager storageManager = new StorageManager();
 Stream stream = null; ;
 log.LogInformation("Reading the Blob Started");
 stream = await storageManager.ReadBlob(name);
 log.LogInformation("Reading the Blob has Completed");
```

```
 EPPLusExcelManager ePPLusExcelManager = new EPPLusExcelManager();
 log.LogInformation("Reading the Excel Data Started");
 List<Employee> employees =
ePPLusExcelManager.ReadExcelData(stream);
 log.LogInformation("Reading the Blob has Completed");
 return employees;
 }
```

2. Add `System.IO` to the namespace list if it's not there already and build the application.
3. Let's now invoke this activity function from the Orchestrator. Go to the `ExcelImport_Orchestrator` Orchestrator function and replace it with the following code. The Orchestration function invokes the activity function by passing the name of the Excel that is uploaded so that the activity function reads the data from the Excel file:

```
[FunctionName("ExcelImport_Orchestrator")]
 public static async Task<List<string>> RunOrchestrator(
 [OrchestrationTrigger] DurableOrchestrationContext context)
 {
 var outputs = new List<string>();
 string ExcelFileName = context.GetInput<string>();
 List<Employee> employees = await
context.CallActivityAsync<List<Employee>>("ReadExcel_AT",
ExcelFileName);
return outputs;
 }
```

4. Let's run the application and then upload an Excel file. If everything is configured properly, then you should see something shown as follows in the `ReadExcel_AT` activity trigger function, where you can see the number of employee records being read from the Excel sheet:

There's more...

The Orchestrator function receives the input using the `GetInput()` method of the `DurableOrchestratorContext` class. This input is passed by the Blob trigger using the function `StartNewAsync` method of the `DurableOrchestrationClient` class.

Auto-scaling Cosmos DB throughput

In the previous recipe, we read data from the Excel and put it into an employee collection. The next step is to insert the collection into a Cosmos DB collection. However, before inserting the data into the Cosmos DB collection, we need to understand that in real-world scenarios, the number of records that we would need to import would be huge and so you might face performance issues if the capacity of the Cosmos DB collection is not sufficient.

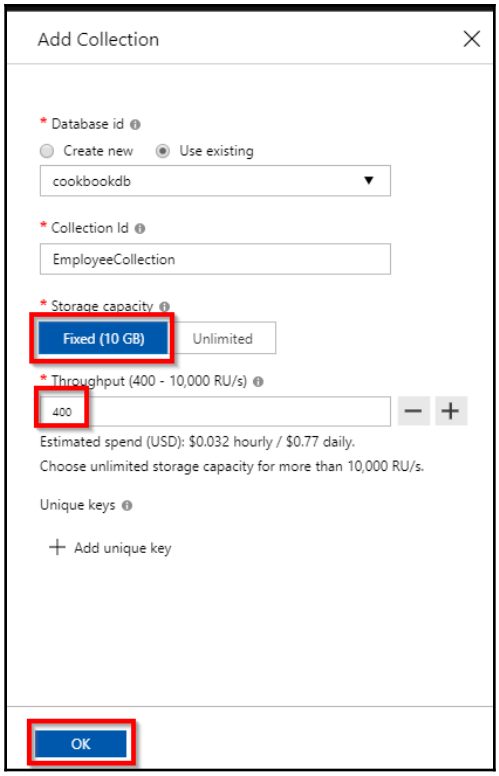
Cosmos DB collection throughput is measured by the number of **Request Units** (**RU**) allocated to the collection. You can read more about it at `https://docs.microsoft.com/en-us/azure/cosmos-db/request-units`.

Also, in order to lower costs, for every service, it is recommended to have the capacity at a lower level and increase it whenever needed. The Cosmos DB API allows us to control the number of RUs based on our needs. As we need to do a bulk import, we will increase the RUs before we start importing the data. Once the importing process is complete, we can decrease the RUs to the minimum level.

Getting ready

Perform the following prerequisites:

1. Create a Cosmos DB account by following the instructions mentioned in the article at `https://docs.microsoft.com/en-us/azure/cosmos-db/create-sql-api-dotnet`.
2. Create a Cosmos database and a collection with fixed storage and set the request units to `400` per second, as shown in the following screenshot:



For the sake of simplicity, I have taken **Fixed (10 GB)** as the **Storage capacity**. However, in production loads, depending on your data models, you might have to go with **Unlimited** storage capacity.

3. Run the following command in the NuGet package manager to install the dependencies of Cosmos DB:

```
Install-Package Microsoft.Azure.WebJobs.Extensions.CosmosDB
```

How to do it...

Perform the following steps:

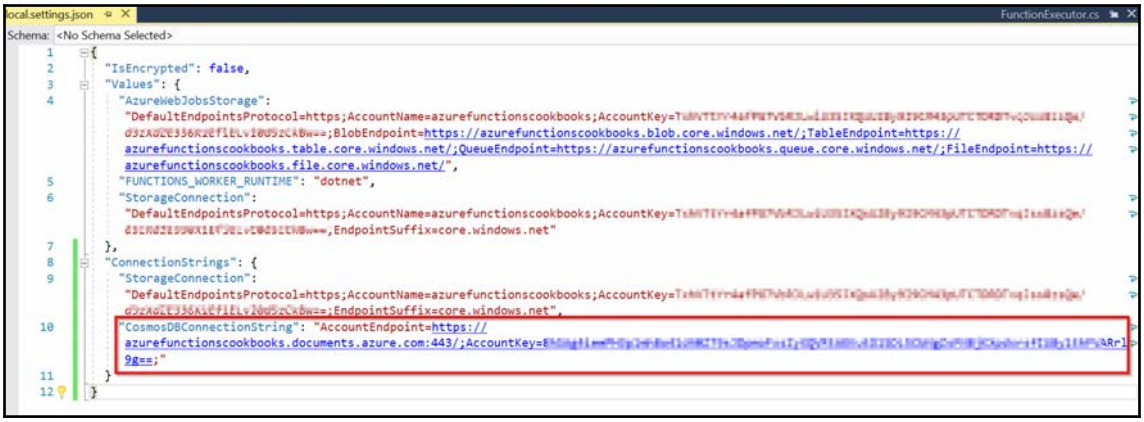
1. Create a new activity trigger named `ScaleRU_AT` in the `ExcelImport_Orchestrator.cs` file. The function should look something like this, and accepts the number of RUs to be scaled up to, along with the Cosmos DB binding using which we replaced the throughput:

```
[FunctionName("ScaleRU_AT")]
 public static async Task<string> ScaleRU_AT(
 [ActivityTrigger] int RequestUnits,
 [CosmosDB(ConnectionStringSetting =
"CosmosDBConnectionString")]DocumentClient client
 )
 {
 DocumentCollection EmployeeCollection = await
client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollect
ionUri("cookbookdb", "EmployeeCollection"));
 Offer offer = client.CreateOfferQuery().Where(o => o.ResourceLink
== EmployeeCollection.SelfLink).AsEnumerable().Single();
 Offer replaced = await client.ReplaceOfferAsync(new OfferV2(offer,
RequestUnits));
 return $"The RUs are scaled to 500 RUs!";
 }
```

2. Add the following namespaces to the `ExcelImport_Orchestrator.cs` file:

```
using System.Linq;
 using Microsoft.Azure.Documents;
 using Microsoft.Azure.Documents.Client;
```

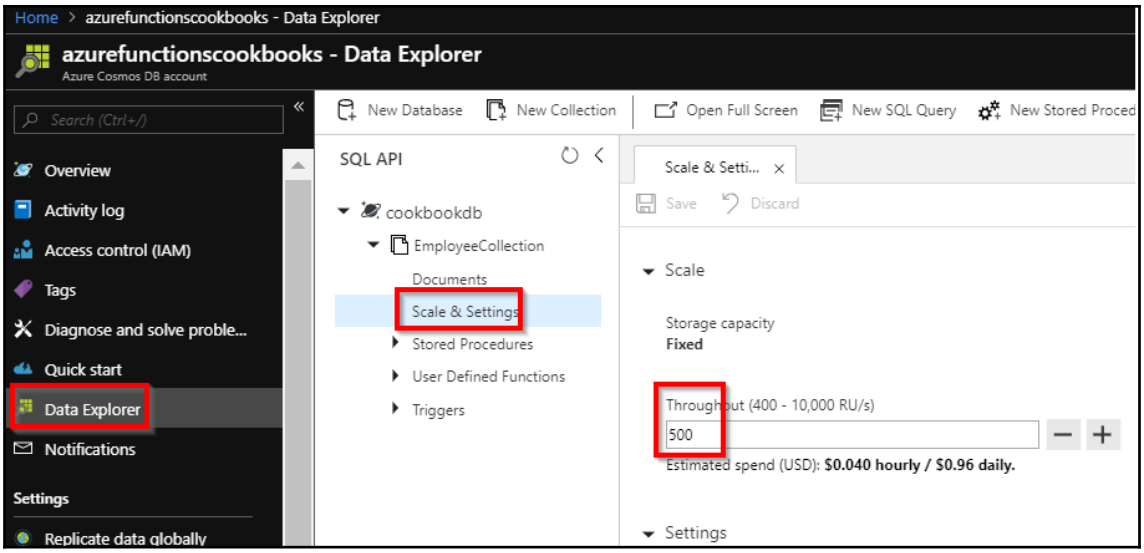
3. Create a new connection string for Cosmos DB, as shown in the following screenshot. You can copy the connection from the **Keys** blade of the Cosmos DB account:



4. Now, in the `ExcelImport_Orchestrator` function, add the following line to invoke `ScaleRU_AT`. In this example, I'm passing `500` as the RU value. Depending on your requirements, you may choose a different value:

```
await context.CallActivityAsync<string>("ScaleRU_AT", 500);
```

5. Now, upload an Excel file to trigger the Orchestration, which internally invokes the new activity trigger, `ScaleRU_AT`, and if everything went well, the new capacity of the Cosmos DB collection should be `500`. Let's navigate to Cosmos DB's **Data Explorer** tab and navigate to the **Scale & Settings** section, where you can view **500** as the new throughput of the collection, as shown here:



There's more...

The Cosmos DB collection's capacity is represented as a resource called **offer.** In this recipe, we have retrieved the existing offer and replaced it with a new offer. You can learn more about this at `https://docs.microsoft.com/en-us/rest/api/cosmos-db/offers`.

Bulk inserting data into Cosmos DB

Now that we have scaled up the collection, it's time to insert the data into the Cosmos DB collection. In this recipe, we will learn about one of the simplest ways of inserting data into Cosmos DB, to make the recipe simple and straightforward.

How to do it...

Perform the following steps:

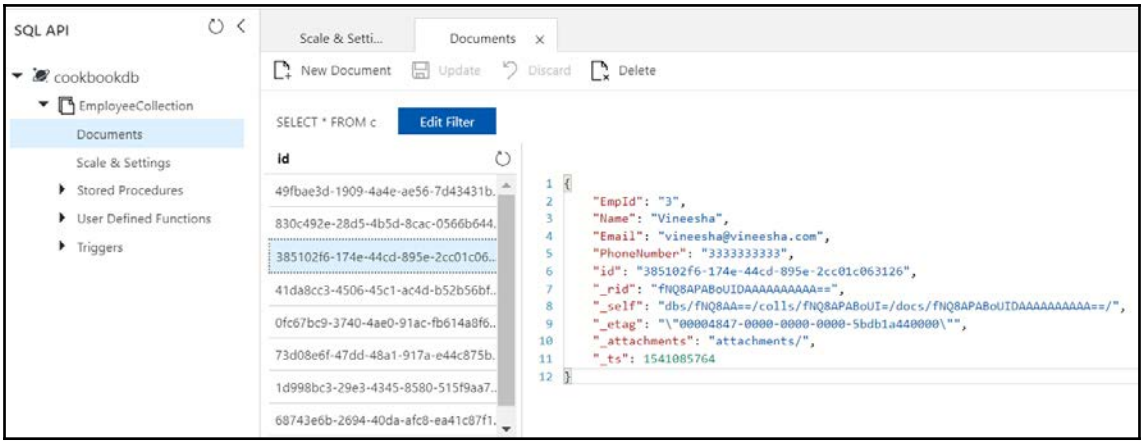
1. Create a new activity trigger named `ImportData_AT`, which takes employee collection as input and saves the data in the collection. Paste the following code into the new activity trigger:

```
[FunctionName("ImportData_AT")]
 public static async Task<string> ImportData_AT(
 [ActivityTrigger] List<Employee> employees,
 [CosmosDB(ConnectionStringSetting =
"CosmosDBConnectionString")]DocumentClient client,
 ILogger log)
 {
 foreach (Employee employee in employees)
 {
 await
client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("
cookbookdb", "EmployeeCollection"), employee);
 log.LogInformation($"Successfully inserted {employee.Name}.");
 }
 return $"Data has been imported to Cosmos DB Collection
Successfully!";
 }
```

2. Let's add the following line to the Orchestration function that invokes the `ImportData_AT` activity trigger:

```
await context.CallActivityAsync<string>("ImportData_AT",
employees);
```

3. Let's run the application and upload an Excel file to test the functionality. If everything went well, you should see all the records created in the Cosmos DB collection, as shown here:



There's more...

The Cosmos DB team has released a library called Cosmos DB bulk executor. You can learn more about this at `https://docs.microsoft.com/en-us/azure/cosmos-db/bulk-executor-overview`.

In this recipe, I have hardcoded the name of the collection and the database to make it simple. You will have to configure them in the app settings file.

9
Implementing Best Practices for Azure Functions

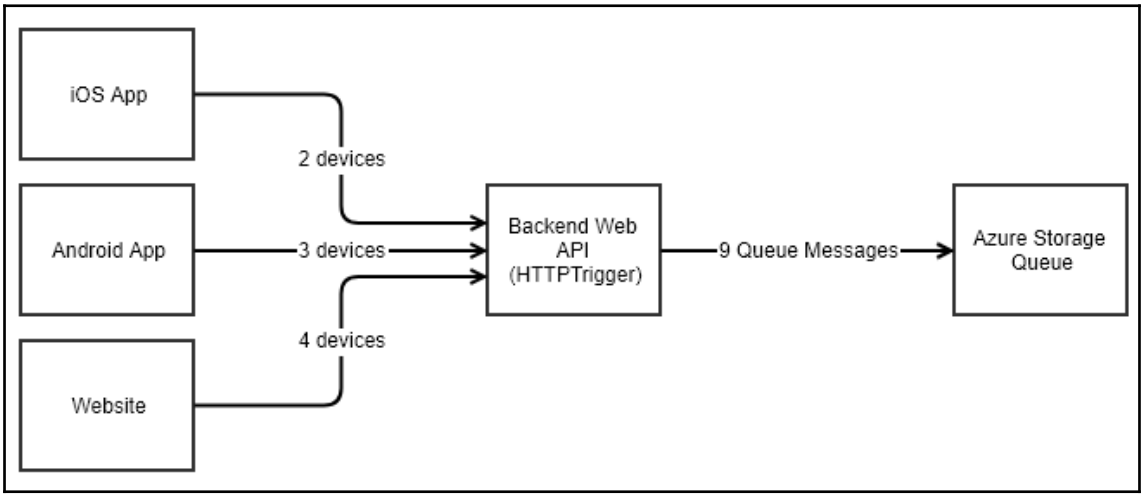
In this chapter, you will learn a few of the best practices that can be followed while working with Azure Functions, such as the following:

- Adding multiple messages to a queue using the `IAsyncCollector` function
- Implementing defensive applications using Azure Functions and queue triggers
- Handling massive ingress using Event Hubs for IoT and other similar scenarios
- Avoiding cold starts by warming the app at regular intervals
- Enabling authorization for function apps
- Controlling access to Azure Functions using function keys
- Securing Azure Functions using Azure Active Directory
- Configuring throttling of Azure Functions using API Management
- Securely accessing SQL Database from Azure Functions using Managed Service Identity
- Shared code across Azure Functions using class libraries
- Using strongly typed classes in Azure Functions

Adding multiple messages to a queue using the IAsyncCollector function

In the first chapter, you learned how to create a queue message for each request coming from the HTTP request. Now let's assume that each user is registering their devices using client applications (such as desktop apps, mobile apps, or any client websites) that can send multiple records in a single request. In these cases, the backend application should be smart enough to handle the load coming to it; there should be a mechanism to create multiple queue messages at once and asynchronously. You will learn how to create multiple queue messages using the `IAsyncCollector` interface.

Here is a diagram that depicts the data flow from different client applications to the backend web API:



In this recipe, we will simulate the requests using Postman, which will send the request to the **Backend Web API (HTTPTrigger)**, which can create all the queue messages in a single go.

Getting ready

These are the required steps:

1. Create a storage account using the Azure portal if you have not created one yet
2. Install Microsoft Storage Explorer from `http://storageexplorer.com/` if you have not installed it yet

How to do it...

Perform the following steps:

1. Create a new HTTP trigger named `BulkDeviceRegistrations` by setting the **Authorization Level** to **Anonymous**.
2. Replace the default code with the following code and click on the **Save** button to save the changes. The following code expects a JSON array as an input parameter with an attribute named `devices`. If found, it will iterate through the array items and then display them in the logs. Later, we will modify the program to bulk insert the array elements into the queue message:

```
#r "Newtonsoft.Json"
using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
public static async Task<IActionResult> Run(HttpRequest req,
ILogger log )
{
log.LogInformation("C# HTTP trigger function processed a
request.");
string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
dynamic data = JsonConvert.DeserializeObject(requestBody);
string Device = string.Empty;
for(int nIndex=0;nIndex<data.devices.Count;nIndex++)
{
Device = Convert.ToString(data.devices[nIndex]);
log.LogInformation("devices data" + Device);
}
return (ActionResult)new OkObjectResult("Program has been executed
Successfully.");
}
```

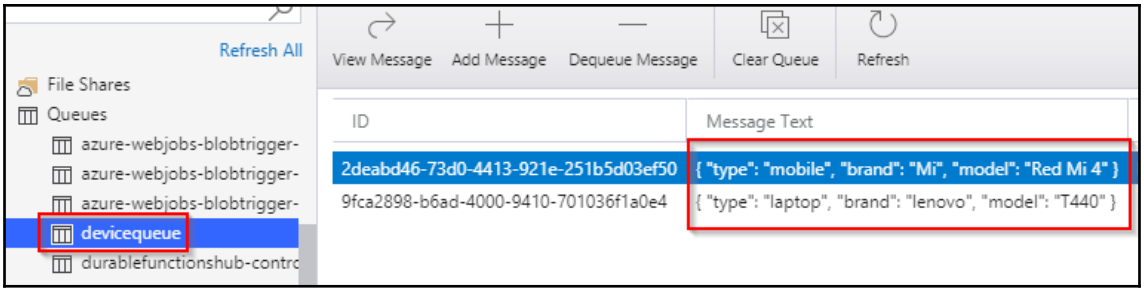
3. The next step is to create an Azure Queue storage output binding. Click on the **Save** button, navigate to the **Integrate** tab, and add a new **Azure Queue Storage** output binding. Then click on the **Select** button and provide the name of the queue and other parameters.
4. Click on the **Save** button and navigate to the code editor of the Azure Function. Add the additional code required for the output binding with the queue to save the messages, as shown in the following code. Make the highlighted changes in the code editor and click on the **Save** button to save the changes:

```
public static async Task<IActionResult> Run(HttpRequest req,
ILogger log,
IAsyncCollector<string> DeviceQueue )
{
....
....
for(int nIndex=0;nIndex<data.devices.Count;nIndex++)
{
 Device = Convert.ToString(data.devices[nIndex]);
 DeviceQueue.AddAsync(Device); }
....
....
```

5. Let's run the function from the **Test** tab of the portal with the following input request JSON:

```
{
"devices":
   [
     {
       "type": "laptop",
       "brand":"lenovo",
       "model":"T440"
     },
     {
        "type": "mobile",
        "brand":"Mi",
        "model":"Red Mi 4"
     }
   ]
}
```

6. Click on the **Run** button to test the functionality. Now open **Azure Storage Explorer** and navigate to the queue named `devicequeue`. As shown in the following screenshot, you should see two records:



How it works...

We created a new HTTP function that has a parameter of the `IAsyncCollector<string>` type, which can be used to store multiple messages in a queue service at once and asynchronously. This approach of storing multiple items asynchronously will reduce the load on the instances.

Finally, we tested the invocation of the HTTP trigger from the Azure portal and also saw the queue messages get added using Azure Storage Explorer.

There's more...

You can also use the `ICollector` interface in place of `IAsyncCollector` if you would like to store multiple messages synchronously.

Note that you might have to install Azure Storage Extensions for the adding of output bindings if you've not done so already. In Azure Functions V2 runtime, adding extensions to each of the services (storage, in this case) is mandatory.

Implementing defensive applications using Azure Functions and queue triggers

For many applications, even after performing multiple tests of different environments, there might still be unforeseen reasons that the application might fail. Developers and architects cannot predict all unexpected inputs throughout the lifespan of an application being used by business users or general end users. So, it's good practice to make sure that your application alerts you if there are any errors or unexpected issues with the applications.

In this recipe, you will learn how Azure Functions help us handle these kinds of issues with minimal code.

Getting ready

These are the required steps:

1. Create a storage account using the Azure portal if you have not created one yet
2. Install Azure Storage Explorer from `http://storageexplorer.com/` if you have not installed it yet

How to do it...

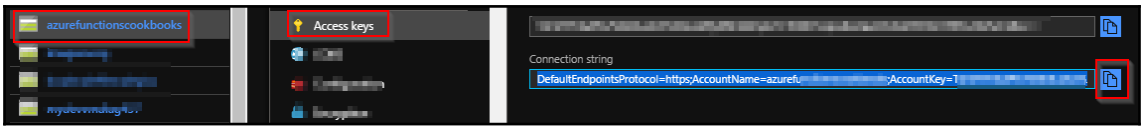
In this recipe, we will do the following:

- Develop a console application using C# that connects to the storage account and creates queue messages in the queue named `myqueuemessages`
- Create an Azure Function queue trigger named `ProcessData` that is fired whenever a new message is added to the queue named `myqueuemessages`

CreateQueueMessage – C# console application

Perform the following steps:

1. Create a new console application using the .NET Core C# language and create an app setting key named `StorageConnectionString` with your storage account connection string. You can get the connection string from the **Access Keys** blade of the storage account as shown:



2. Install the `Configuration` and `Queue Storage` NuGet packages using the following commands:

```
Install-Package Microsoft.Azure.Storage.Queue
Install-Package System.Configuration.ConfigurationManager
```

3. Add the following namespaces:

```
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Queue;
using System.Configuration;
```

4. Add the following function to your console application and call it from the `Main` method. The `CreateQueueMessages` function creates 100 messages with the index as the content of each message:

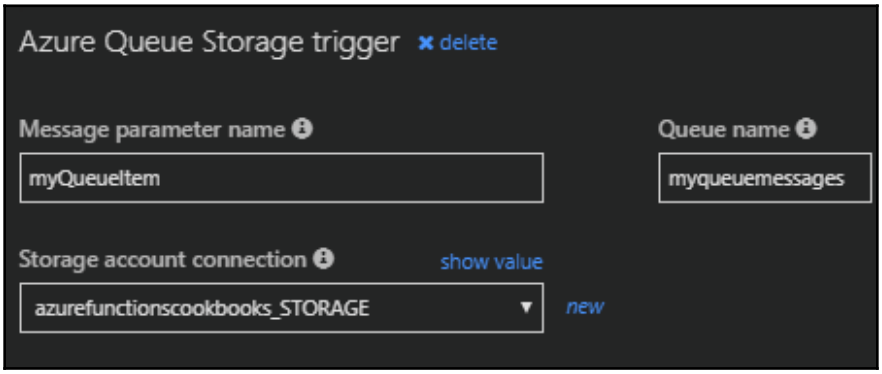
```
static void CreateQueueMessages()
{
   CloudStorageAccount storageAccount =
    CloudStorageAccount.Parse(ConfigurationManager.AppSettings
    ["StorageConnectionString"]);
   CloudQueueClient queueclient =
    storageAccount.CreateCloudQueueClient();

   CloudQueue queue = queueclient.GetQueueReference
    ("myqueuemessages");
   queue.CreateIfNotExists();

   CloudQueueMessage message = null;
   for(int nQueueMessageIndex = 0; nQueueMessageIndex <= 100;
    nQueueMessageIndex++)
   {
```

```
        message = new CloudQueueMessage(Convert.ToString
         (nQueueMessageIndex));
        queue.AddMessage(message);
        Console.WriteLine(nQueueMessageIndex);
    }
}
```

Developing the Azure Function – queue trigger

Perform the following steps:

1. Create a new Azure Function named `ProcessData` using the queue trigger and configure the `myqueuemessages` queue. This is how the **Integrate** tab should look after you create the function:



2. Replace the default code with the following code:

```
using System;
public static void Run(string myQueueItem,
 ILogger log)
{
    if(Convert.ToInt32(myQueueItem)>50)
    {
        throw new Exception(myQueueItem);
    }
    else
    {
        log.LogInformation($"C# Queue trigger function
         processed: {myQueueItem}");
    }
}
```

The preceding queue trigger logs a message with the content of the queue (it's just a numerical index) for the first 50 messages and then throws an exception for the all the messages whose content is greater than 50.

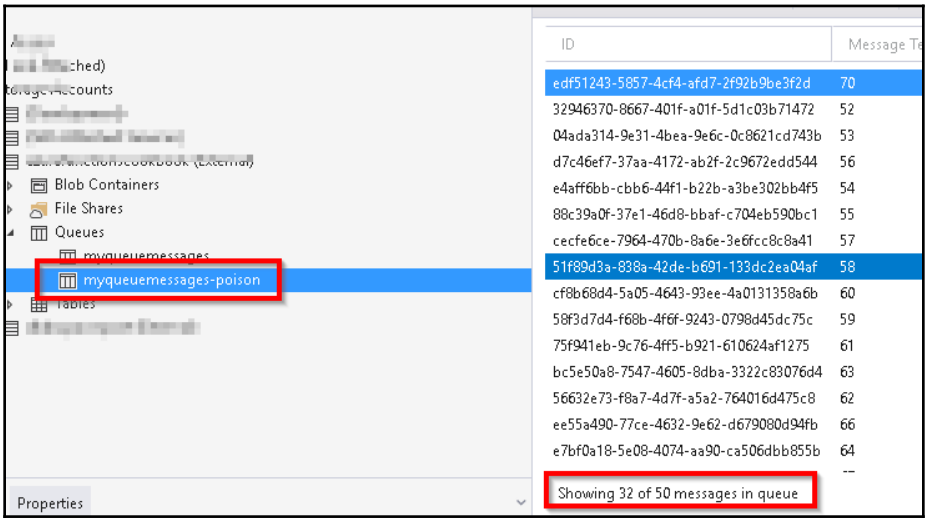
Running tests using the console application

Perform the following steps:

1. Let's execute the console application by pressing *Ctrl + F5*, navigate to **Azure Storage Explorer**, and view the queue contents.
2. In just a few moments, you should start viewing messages in the `myqueuemessages` queue. Currently, both the Azure portal and Storage Explorer display the first 32 messages. You need to use the C# Storage SDK to view all the messages in the queue.

Don't be surprised if you notice that your messages in `myqueuemessage` are vanishing. It's expected that as soon as a message is read successfully, the message gets deleted from the queue.

3. As shown here, you should also see a new queue named `myqueuemessages-poison` (`<OriginalQueuename>-Poison`) with the 50 other queue messages in it. The Azure Function runtime will automatically take care of creating a new queue and adding the messages that are not read properly by Azure Functions:



How it works...

We have created a console application that creates messages in the Azure Queue storage , and we have also developed a queue trigger that is capable of reading the messages in the queue. As part of simulating an unexpected error, we are throwing an error if the value in the queue message content is greater than 50.

Azure Functions will take care of creating a new queue with the name `<OriginalQueueName>-Poison` and will insert all the unprocessed messages in the new queue. Using this new poison queue, developers can review the content of the messages and take necessary actions to fix errors in the applications.

The Azure Function runtime will take care of deleting the queue message after Azure Function execution has completed successfully. If there are any problems in the execution of the Azure Function, it automatically creates a new poison queue and adds the processed messages to the new queue.

There's more...

Before pushing a queue message to the poison queue, the Azure Function runtime tries to pick the message and process five times. You can learn about how this process works by adding a new `dequecount` parameter of the `int` type to the `Run` method and logging its value.

Handling massive ingress using Event Hubs for IoT and other similar scenarios

In many scenarios, you might have to handle massive amounts of incoming data; the incoming data might be coming from sensors and telemetry data, and it could be as simple as the data sent from Fitbit devices. In these scenarios, we need to have a reliable solution that is capable of handling massive amounts of data. Azure Event Hubs is one such solution that Azure provides. In this recipe, you will learn how to integrate Event Hubs and Azure Functions.

Getting ready

Perform the following steps:

1. Create an Event Hubs namespace by navigating to **Internet of Things** and choosing **Event Hubs**
2. Once the Event Hubs namespace is created, navigate to the **Overview** tab and click on the **Event Hub** icon to create a new Event Hub
3. By default, a **Consumer Group** named `$Default` is created, which we will be using in this recipe

How to do it...

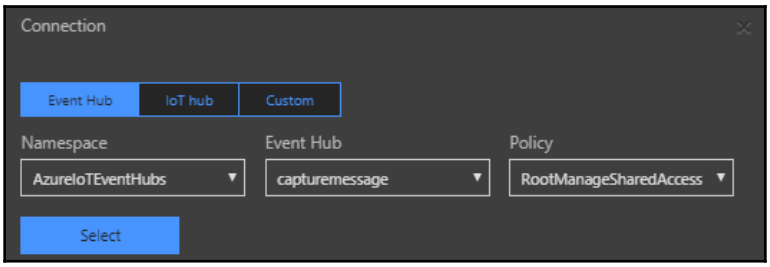
We will be doing the following in this recipe:

- Creating an Azure Function event hub trigger
- Developing a console application that simulates **Internet of Things** (**IoT**) data

Creating an Azure Function event hub trigger

Perform the following steps:

1. Create a new Azure Function by choosing **Event Hub Trigger** in the template list.
2. Once you have selected the template, you might have to install the extensions. Then, you will need to provide the name of the event hub, `capturemessage`. If you don't have any connections configured yet, you need to click on the **New** button.
3. Clicking on the **New** button will open a **Connection** popup, where you can choose your **Event Hub** and other details. Choose the required details and click on the **Select** button, as shown in the following screenshot:



4. The previous step will populate the **Event Hub Connection** drop-down. Finally, click on **Create** to create the function.

Developing a console application that simulates IoT data

Perform the following steps:

1. Create a new console application that will send events to the Event Hub. I have named it `EventHubApp`.
2. Run the following commands in the NuGet package manager to install the required libraries and interact with Azure Event Hubs:

```
Install-Package Microsoft.Azure.EventHubs
Install-Package Newtonsoft.Json
```

3. Add the following namespaces and a reference to `System.Configuration.dll`:

```
using Microsoft.Azure.EventHubs;
using System.Configuration;
```

4. Add the connection string to `App.config`, which is used to connect the event hub. This is the code for `App.config`. You can get the connection string by clicking on the **ConnectionStrings** link in the **Overview** tab of the event hub namespace:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<startup>
<supportedRuntime version="v4.0"
 sku=".NETFramework,Version=v4.6.1" />
</startup>
<appSettings>
<add key="EventHubConnection"
  value="Endpoint=sb://<event hub namespace
   here>.servicebus.windows.net/;Entitypath=<Event Hubname>;
  SharedAccessKeyName= RootManageSharedAccessKey;
   SharedAccessKey=<Key here>"/>
</appSettings>
</configuration>
```

5. Create a new C# class file and place the following code in the new class file:

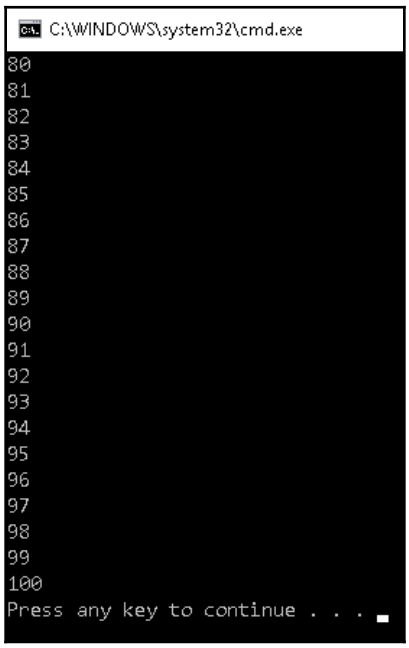
```
using System;
using System.Text;
using Microsoft.Azure.EventHubs;
using System.Configuration;
using System.Threading.Tasks;

namespace EventHubApp
{
   class EventHubHelper
   {
      static EventHubClient eventHubClient = null;
      public static async Task GenerateEventHubMessages()
      {

         EventHubsConnectionStringBuilder conBuilder = new
          EventHubsConnectionStringBuilder
          (ConfigurationManager.AppSettings
          ["EventHubConnection"].ToString());

         eventHubClient =
          EventHubClient.CreateFromConnectionString
          (conBuilder.ToString());
         string strMessage = string.Empty;
         for (int nEventIndex = 0; nEventIndex <= 100;
          nEventIndex++)
         {
            strMessage = Convert.ToString(nEventIndex);
            await eventHubClient.SendAsync(new EventData
             (Encoding.UTF8.GetBytes(strMessage)));
            Console.WriteLine(strMessage);
         }
         await eventHubClient.CloseAsync();
      }
   }
}
```

6. In your `main` function, place the following code, which invokes the method for sending the message:

```
namespace EventHubApp
{
   class Program
   {
      static void Main(string[] args)
      {
         EventHubHelper.GenerateEventHubMessages().Wait();
```

```
            }
        }
    }
```

7. Now execute the application by pressing *Ctrl + F5*. You should see something similar to what is shown here:



8. When the console is printing the numbers, you can navigate to the Azure Function to see that the event hub gets triggered automatically and logs the numbers that are being sent to the Event Hub.

Avoiding cold starts by warming the app at regular intervals

By now, you might be aware of the fact that you can create Azure functions in the following two hosting plans:

- App Service plan
- Consumption plan

You will get all the benefits of serverless architecture only when you create the function app using the consumption plan. However, one of the concerns that developers report about using the consumption plan is something called cold starting, which refers to spinning up an Azure function to serve the requests when there have been no requests for quite some time. You can learn more about this topic at `https://blogs.msdn.microsoft.com/appserviceteam/2018/02/07/understanding-serverless-cold-start/`.

In this recipe, we will learn a technique that could be used to always keep the instance live and warm so that all requests are served properly.

The App Service plan is a dedicated hosting plan where your instances are reserved for you and they can always be warm even if there are no requests for quite a bit of time.

Getting ready

In order to complete this recipe, we need to have a function app with the following:

- An HTTP trigger named **HttpALive**
- A timer trigger named **KeepFunctionAppWarm** that runs every 5 minutes and makes an HTTP request to the HttpALive HTTP trigger

If you have clearly understood what a cold start is, then it will be clear to you that there would be no concerns if your application had traffic regularly during the day. So, if we can ensure that the application has traffic all day, then the Azure Function instance will not be deprovisioned and so there wouldn't be any concerns about the Consumption plan.

How to do it...

In this recipe, we will create a timer trigger that simulates traffic to the HTTP trigger, causing the function app to be alive all the time and the serverless instances to be always in the provisioned state.

Creating an HTTP trigger

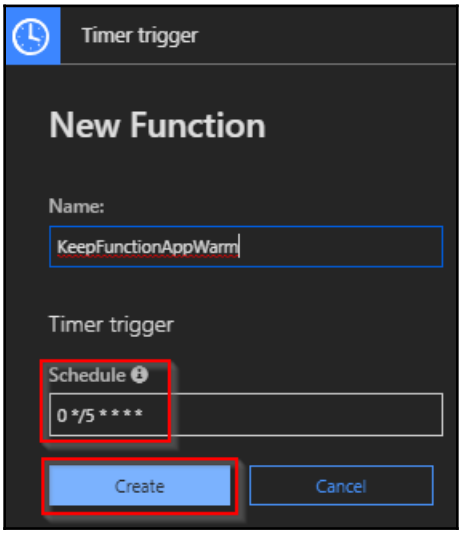
Create a new HTTP trigger and replace the following code that just prints a message when it is executed:

```
using System.Net;
using Microsoft.AspNetCore.Mvc;
public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
return (ActionResult)new OkObjectResult($"Hello User! Thanks for keeping me
Warm");
}
```

Creating a timer trigger

Perform the following steps:

1. Click on the **+** icon, search for `timer`, and click on the **Timer trigger** button.
2. In the **New Function** popup, provide the details. The **Schedule** here is a CRON expression that ensures that the timer trigger gets triggered automatically every 5 minutes:



3. Paste the following code in the code editor and save the changes. The following code simulates traffic by making HTTP requests programmatically. Be sure to replace `<<FunctionAppName>>` with the actual name of your function app:

```
using System;
public async static void Run(TimerInfo myTimer, ILogger log)
{
using (var httpClient = new HttpClient())
{
var response = await
httpClient.GetAsync("https://<FunctionAppName>>.azurewebsites.net/a
pi/HttpALive");
}
}
```

There's more...

If you play around with Azure Functions, you might notice that the cold start times for Azure Functions that have C# as the language are just a few seconds. However, if you are working with Azure Functions that have JavaScript (such as Node.js) as the language, then the cold start time would be greater as the runtime would need to download all the npm packages that are dependencies for your application. In those scenarios, a feature named **Run-From-Package** comes in very handy. We will learn how to implement this in the upcoming chapter.

See also

See the *Deploying Azure Functions using Run From Package* recipe of `Chapter 10`, *Configuring of Serverless Applications in the Production Environment*.

Enabling authorization for function apps

If your web API (HTTP trigger) is being used by multiple client applications and you would like to provide access only to the intended and authorized applications, then you need to implement authorization in order to restrict access to your Azure Function.

Getting ready

I assume that you already know how to create an HTTP trigger function. Download the Postman tool from `https://www.getpostman.com/`. The Postman tool is used for sending HTTP requests. You can also use any tool or application that can send HTTP requests and headers.

How to do it...

Perform the following steps:

1. Create a new HTTP trigger function (or open an existing HTTP function). Make sure that when creating the function, you select **Function** as the option in the **Authorization level** drop-down.

If you would like to go with an existing HTTP trigger function that we have created in one of our previous recipes, click on the **Integrate** tab, change the **Authorization level** to **Function** and click on the **Save** button to save the changes.

2. In the **Code Editor** tab, grab the function URL by clicking on the **Get Function URL** link available in the right-hand corner of the code editor in the `run.csx` file.
3. Navigate to Postman and paste the function URL:

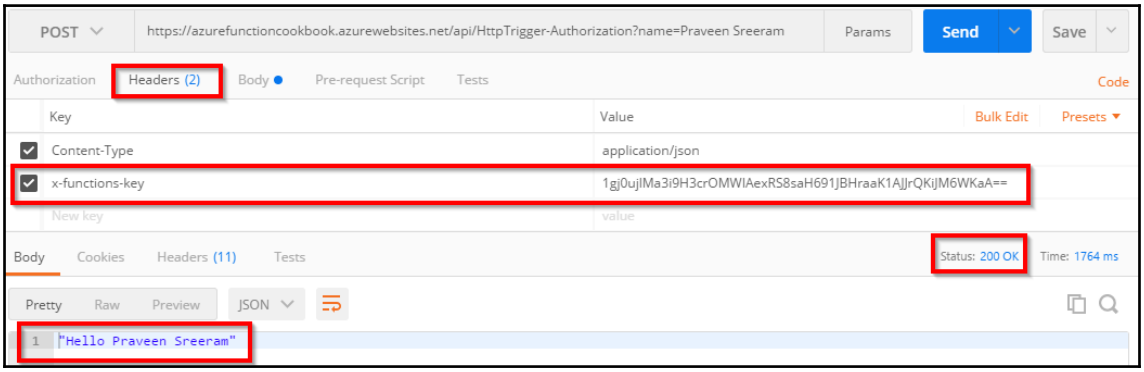
4. Observe that the URL has the following query strings:
    - `code`: This is the default query string that is expected by the function runtime and validates the access rights of the function. The validation functionality is automatically enabled without the need for writing the code by the developer. All of this is taken care of just by setting the **Authorization level** to **Function**.
    - `name`: This is a query string that is required by the HTTP trigger function.
5. Let's remove the `code` query string from the URL in Postman and try to make a request. You will get a **401 Unauthorized** error.

How it works...

When you make a request via Postman or any other tool or application that can send HTTP requests, the request will be received by the underlying Azure App Service web app (note that Azure Functions are built on top of App Services) that first checks the presence of the header name `code` either in the query string collection or in the **Request Body**. If it finds it, then it validates the value of the code query string with the function keys. If it's a valid one, then it authorizes the request and allows the runtime to process the request. Otherwise, it throws an error with a **401 Unauthorized** message.

There's more...

Note that the security key (in the form of the query string parameter named `code`) in the preceding example is used for demonstration purposes only. In production scenarios, instead of passing the key as a query string parameter (the `code` parameter), you need to add the `x-functions-key` as an HTTP header, as shown in the following screenshot:



Controlling access to Azure Functions using function keys

You have now learned how to enable the authorization of an individual HTTP trigger by setting the **Anonymous Level** field with the `value` function in the **Integrate** tab of the HTTP trigger function. It works well if you have only one Azure Function as a backend web API for one of your applications and you don't want to restrict access to the public.

However, in enterprise-level applications, you will end up developing multiple Azure Functions across multiple function apps. In those cases, you need to have fine-grained granular access to your Azure Function for your own applications or for some other third-party applications that integrate your APIs in their applications.

In this recipe, you will learn how to work with function keys within Azure Functions.

How to do it...

Azure supports the following keys, which can be used to control access to the Azure functions:

- **Function keys**: These can be used to grant authorization permissions to a given function. These keys are specific to the function with which they are associated.
- **Host keys**: We can use these to control the authorization of all the functions within an Azure function app.

Configuring the function key for each application

If you are developing an API using Azure Functions that can be used by multiple applications, then it's good practice to have a different function key for each client application that is going to use your functions.

Navigate to the **Manage** tab of Azure Functions to view and manage all the keys related to the function.

By default, a key with the name `default` is generated for us. If you would like to generate a new key, then click on the **Add new function key** button.

As per the preceding instruction, I have created the keys for the following applications:

- `WebApplication`: The key name `WebApplication` is configured to be used in the website that uses the Azure Function
- `MobileApplication`: The key name `MobileApplication` is configured to be used in the mobile app that uses the Azure Function

In a similar way, you can create different keys for any other app (such as an IoT application) depending on your requirements.

The idea behind having different keys for the same function is to have control over the access permissions to the usage of the functions by different applications. For example, if you would like to revoke the permissions only to one application but not for all applications, then you would just delete (or revoke) that key. In that way, you are not impacting other applications that are using the same function.

Here is the downside of the function keys; if you are developing an application where you need to have multiple functions and each function is being used by multiple applications, then you will end up having many keys. Managing these keys and documenting them would be a nightmare. In that case, you can go with host keys, which are discussed next.

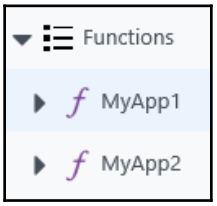
Configuring one host key for all the functions in a single function app

Having different keys for different functions is a good practice when you have a handful of functions used by a few applications. However, things might get worse if you have many functions and many client applications leveraging your APIs. Managing the function keys in these large enterprise applications with huge client bases would be painful. To make things simple, you can segregate all related functions into a single function app and configure the authorization for each function app instead of for each individual function. You can configure authorization for a function app using host keys.

Here are the two different types of host keys available:

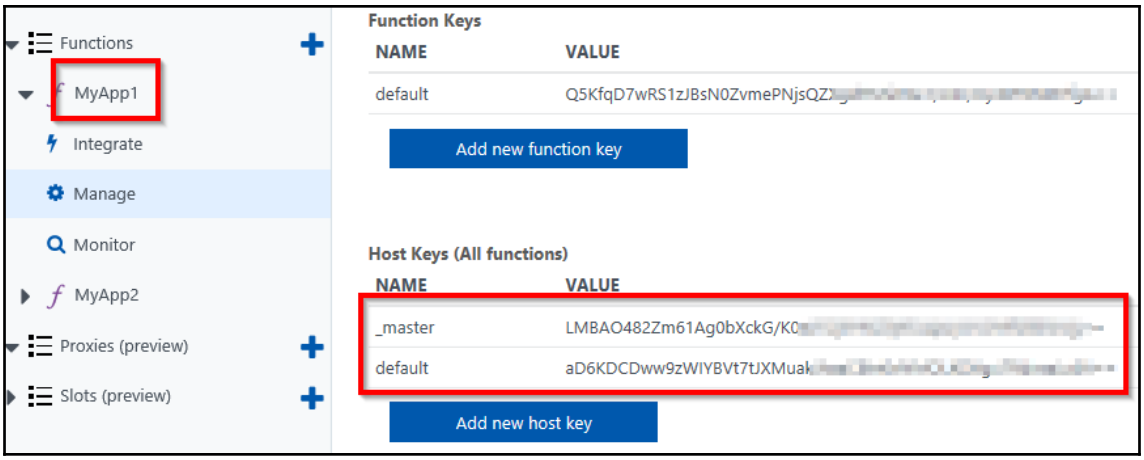
- Regular host keys
- Master key

Create two HTTP trigger apps, as shown in the following screenshot:

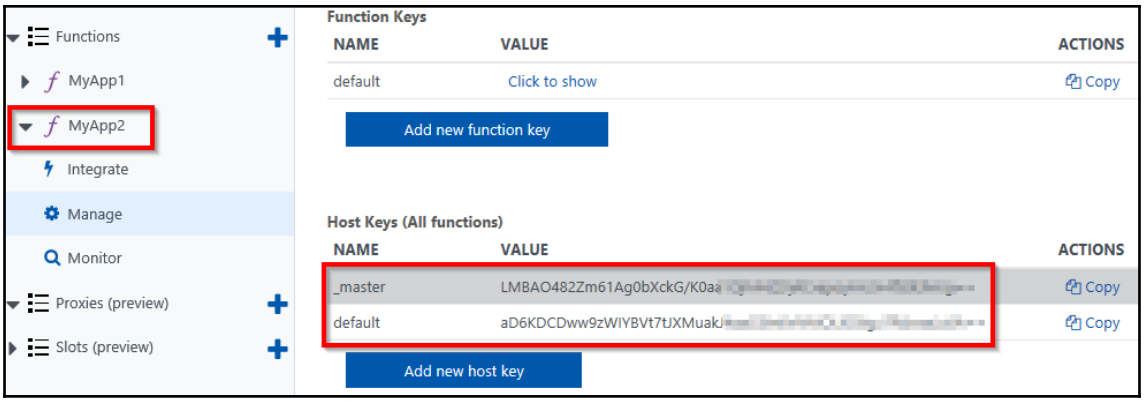


Navigate to the **Manage** tab of both the apps, as shown in the following screenshots. You will notice that both the master key and the host keys are the same in both the apps:

- This is the management tab of `MyApp1`:



- This is the management tab of `MyApp2`:



As with the case of function keys, you can also create multiple host keys if your function apps are being used by multiple applications. You can control the access of each of the function apps by different applications using different keys.
You can create multiple host keys by following the same steps that you followed when creating the regular function keys.

There's more...

If you think that the key has been compromised, then you can regenerate it at any time by clicking on the **Renew** button. Note that when you renew a key, all the applications that access the function would no longer work and would give a **401 Unauthorized** status code error.

You can delete or revoke the key if it is no longer used in any of the applications.

Here's a table with some more guidance on keys:

Key type	**When should I use it?**	**Is it revocable (can it be deleted)?**	**Renewable?**	**Comments**
Master key	When the **Authorization level** is **Admin**	No	Yes	You can use a master key for any function within the function app irrespective of the authorization level configured.
Host key	When the **Authorization level** is **Function**	Yes	Yes	You can use the host key for *all* the functions within the function app.
Function key	When the **Authorization level** is **Function**	Yes	Yes	You can use the function key *only* for a given function.

Microsoft doesn't recommend sharing the master key as it is also used by runtime APIs. Be extra cautious with master keys.

Securing Azure Functions using Azure Active Directory

In the previous recipe, we learned how to enable security based on client applications accessing Azure Functions. However, if your requirement is to authenticate the end users accessing the functions against the Azure **Active Directory** (**AD**), then Azure Functions provides an easy way to configure this called EasyAuth.

Thanks to Azure App Service, from which the EasyAuth feature is inherited, we can do what we need to do without writing a single line of code.

Getting ready

In this recipe, to make things simple, we will use the default AD that is created when you create an Azure account. However, in your real-time production scenarios, you would already have an existing AD that needs to be integrated. I would recommend going over this article: `https://docs.microsoft.com/en-us/azure/active-directory-b2c/active-directory-b2c-tutorials-web-app`.

How to do it...

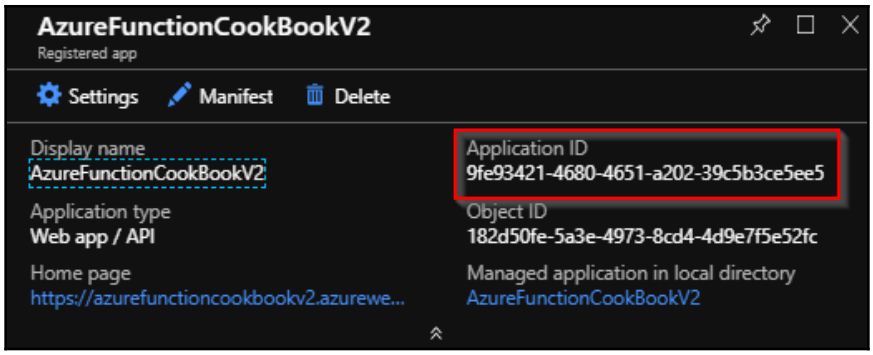
This recipe will involve the following:

- Configuring Azure AD to the function app
- Registering the client app in Azure AD
- Granting the client app access to the backend app
- Testing the authentication functionality using a JWT token

Configuring Azure AD to the function app

Perform the following steps:

1. Navigate to the **Platform features** section of Azure Functions.
2. In the **Authentication/Authorization** blade, perform the following steps to enable the AD authentication:
    1. Click on the **On** button to enable the authentication.
    2. Choose the **Login using Azure Active Directory** menu option.
    3. Click on the **Not Configured** button to start configuring the options.
3. The next step is to choose an existing or create a new registration for the client application that we want to provide access to. This can be done by pressing the **Express** button in the **Management Mode** field. Also, I opted to create a new one and provided **AzureFunctionCookbookV2** as the name for my app registration. Click **OK** to save the configurations, which will take you to the following screen.
4. Grab the **Application ID** as shown. We will be using it while testing in a few moments:

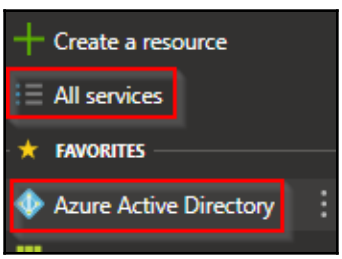


5. That's it. Without writing a single line of code, we are done with configuring an Azure AD instance that sits as a security layer and allows access only to authenticated users. In other words, we have enabled OAuth for our backend function app using Azure AD. Let's quickly test it by accessing any of the HTTP triggers that you have in the function app. I have used Postman to do this. As expected, you will get an error asking you to log in.
6. With the current configurations, none of the external client applications will be able to access our backend API. In order to provide access, we need to perform the following steps:
    1. Register all the client apps in Azure AD (for our example, we will do a registration for the Postman app).
    2. Grant access to the backend app.

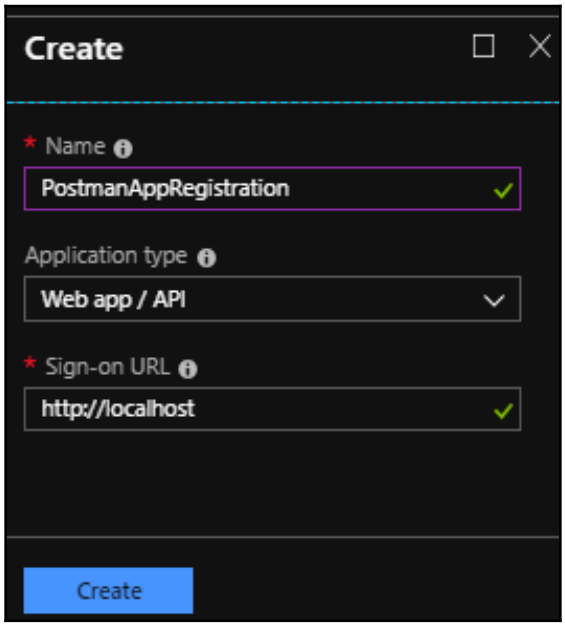
Registering the client app in Azure AD

Perform the following steps:

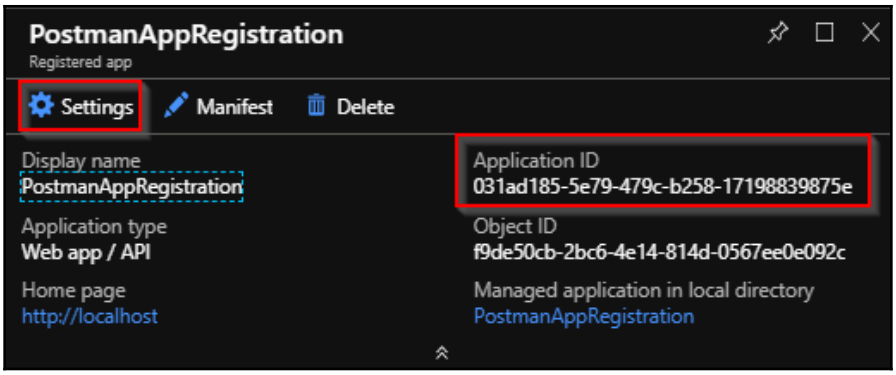
1. Navigate to Azure AD by clicking on the **Azure Active Directory** button, as shown. If you don't see it in the favorites list, you can search in the **All Services** blade, which is also highlighted in the following screenshot:



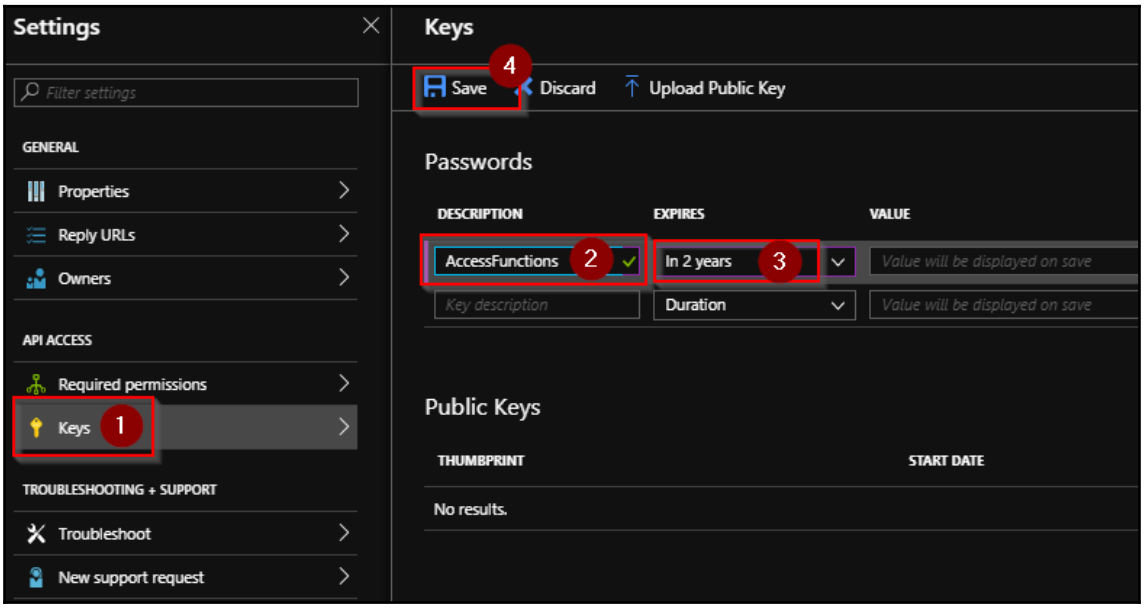
2. In the AD menu, click on **App Registrations** and then click on the **New application registration** button.
3. Fill in the fields as follows and click on the **Create** button to complete the registration for our Postman app. As our client app is Postman, the Sign-on URL doesn't hold any importance, so just `http://localhost` should be good for our example:



4. In just a moment, the app will be created and you will be taken to the following screen. Grab the **Application ID** and save it on your notepad. We will be using it in the upcoming steps. Click on the **Settings** button:



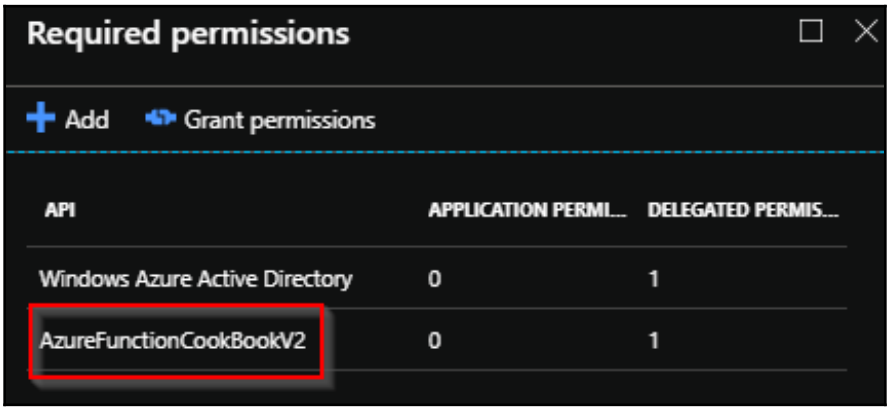
5. In the **Settings** blade, click on the **Keys** menu item to generate a key, which we will be passing from Postman. In order to generate the key, we first need to provide a **Description** and the **Duration** after which the key should expire. Provide the details as shown in the following screenshot and click on the **Save** button. The actual key is displayed to us in the **value** field only once immediately after you click on **Save** button, so be sure to copy it and store it in a secure place. We will be using this in a few moments:



Granting the client app access to the backend app

Once the client application is registered, we need to provide it the access to our backend app. In this section, we will learn how to configure it:

1. Click on the **Required Permissions** tab and click on the **Add** button, which shows the **Add API Access** blade, where you choose the required API (in our case, it is our backend Azure Functions API).
2. In the **Add API Access** blade, click on the **Select an API** button; initially, default APIs will be displayed. You need to search for your backend app with the name that you have provided (in my case, it was **AzureFunctionCookBookV2**). Select the backend app and click on the **Select** button.
3. The next step is to provide the actual permissions. Click on the **Select Permissions** tab and check the **Access <Backend App name>**, then click on **Select** and then on the **Done** button.
4. Ensure that you get the following screen. You can also click on the **Grant Permission** button to apply the changes:



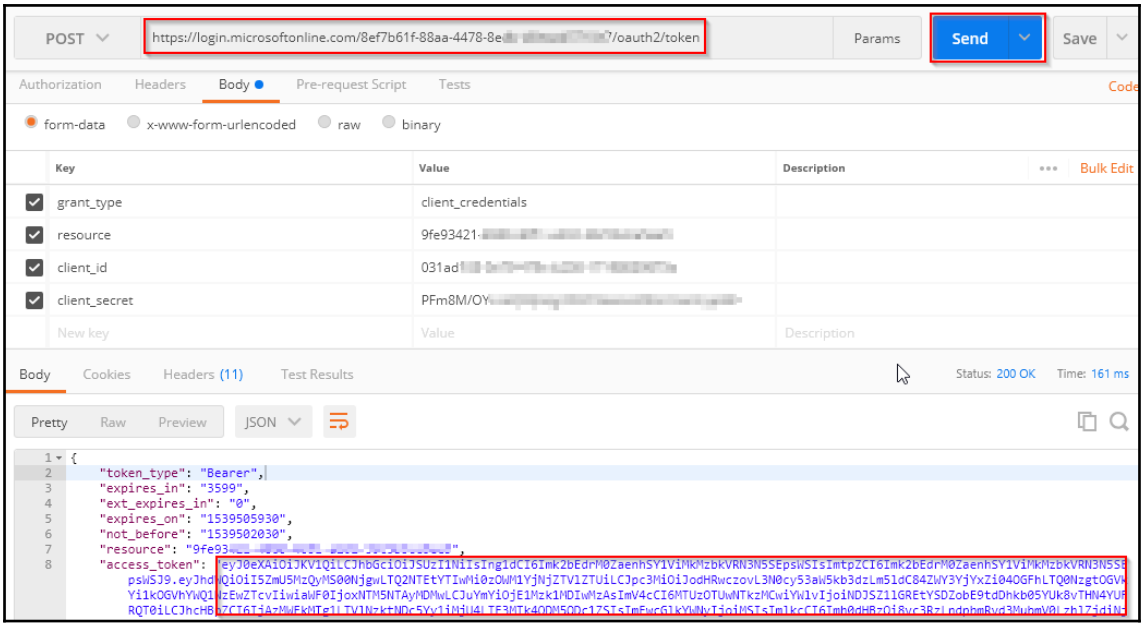
Testing the authentication functionality using a JWT token

You should have the following ready to test the functionality using Postman:

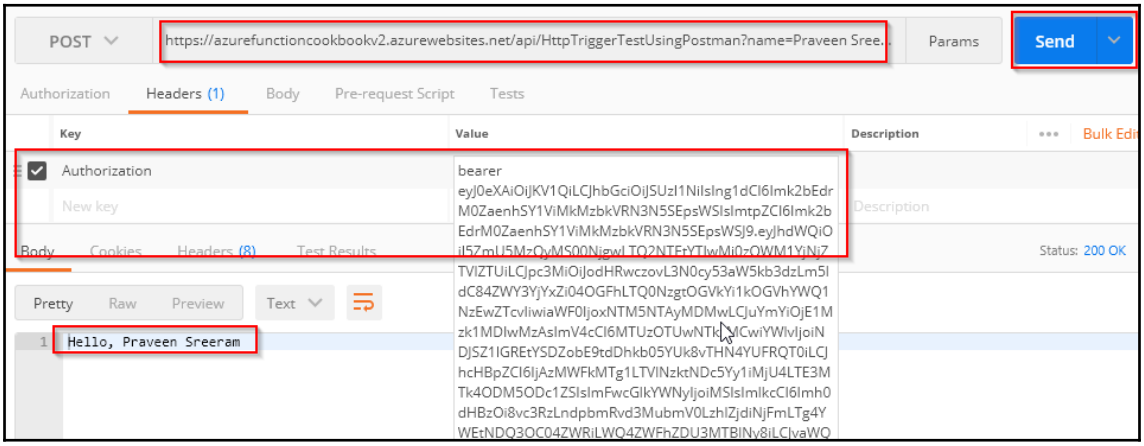
1. OAuth 2.0 token endpoint, you can get this in the **Endpoints** tab of Azure AD and grab the URL:
    - **Grant type**: A hardcoded `client_credentials` value.

- **Client ID of the client application**: You noted it in the fourth step of the *Registering the client app in Azure AD* section.
- **Key that you generated for your client application**: You noted it in the fifth step of the *Registering the client app in Azure AD* section.
- **Resource**: Resource to which we need to access. It's the client ID of the backend application; you noted it in the fourth step of the *Configuring Azure AD to the function app* section.

2. Once you have all that information, you need to pass all the parameters and make a call to an Azure AD tenant, which returns the bearer token as follows:



3. The next and final step is to make a call to the actual backend (the Azure Function HTTP trigger) by passing the bearer JWT token (`access_token`) that we copied from the preceding screen:



4. As shown in this screenshot, add an **Authorization** header and paste the JWT token. Don't forget to provide the text **bearer** word.

Configuring throttling of Azure Functions using API Management

We have already learned in previous chapters that we can use Azure Functions HTTP triggers as backend web API. If you want to restrict the number of requests by your client applications to, let's say, 10 requests per second, then you might have to develop a lot of logic. Thanks to **Azure API Management**, you don't need to write any custom logic if you integrate Azure Functions with API Management.

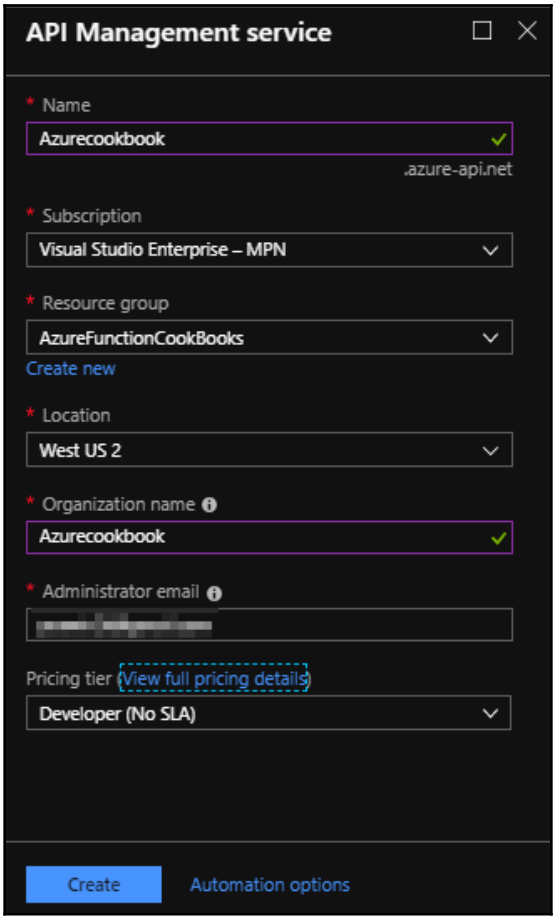
In this recipe, we will learn how to restrict clients to API access only once per minute for a given IP address. The following are the high-level steps that we will follow:

1. Creating an Azure API Management service
2. Integrating Azure Functions with API Management
3. Configuring request throttling using inbound policies
4. Testing the rate limit inbound policy configuration

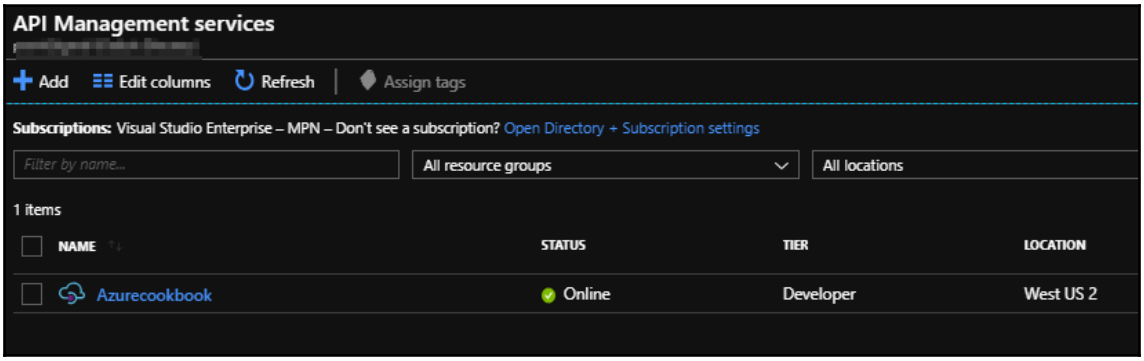
Getting ready

To get started, we need to create an Azure API Management service by performing the following steps:

1. Search for **API Management** and provide all the following details. In the following example, I have chosen the **Developer** pricing tier. But for your production applications, you need to choose non-developer tiers (Basic/Standard/Premium) as the **Developer (No SLA)** tier doesn't provide any SLAs. Once you have reviewed all the details, click on the **Create** button:



2. At the time of writing, it takes around 30 minutes to create an API Management instance. Once it has been created, you can view it in the **API Management services** blade:



How to do it...

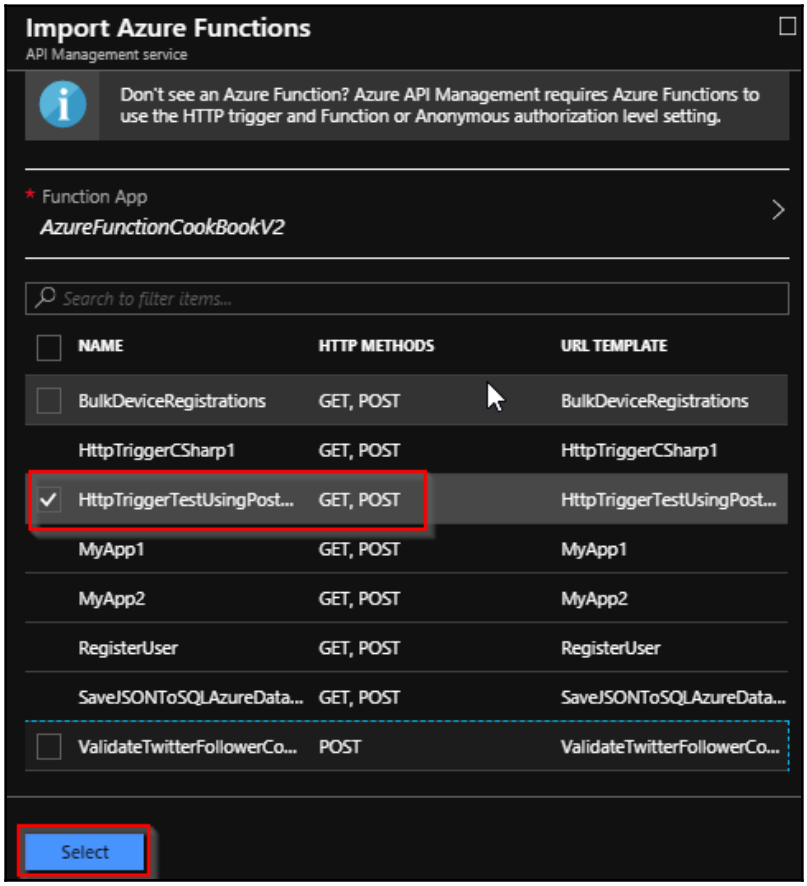
In order to leverage the API Management capabilities, we need to integrate the service endpoints (in our case, the HTTP triggers that we have created) with the API Management service. This section talks about the steps required for integration. Let's start integrating both.

Integrating Azure Functions with API Management

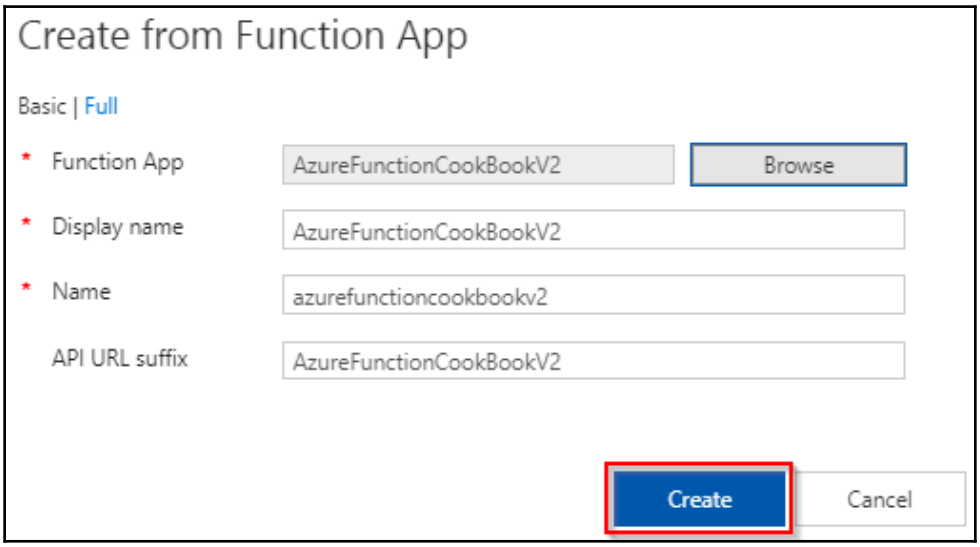
Perform the following steps:

1. Navigate to the **APIs** blade of the API Management Instance that you have created and click on the **Function App** tile.
2. You will see a **Create from Function App** popup where you can click on the **Browse** button, which will open a side bar with the title **Import Azure Functions**, where you can configure the function apps. Click on the **Configure Required Setting** button to view all the function apps that have HTTP triggers in them. Once you have select the function app, click on the **Select** button.

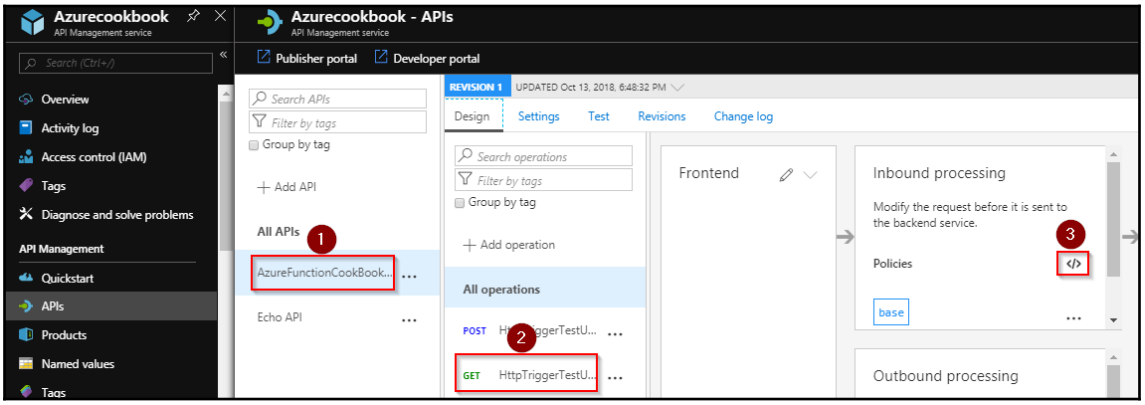
3. The next step is to choose the HTTP trigger that you would like to integrate with Azure API Management. After clicking on the **Select** button, as mentioned in the previous step, all the HTTP triggers associated with the selected function app will appear, as shown in the following screenshot. I have chosen only one HTTP trigger to make things simple:



4. After performing all the preceding steps, the **Create from Function App** popup will appear, looking something like the following. Once you have reviewed the details, click on the **Create** button:



5. If everything goes fine, you should get a screenshot as follows. Now we are done with integrating Azure Functions with API Management:



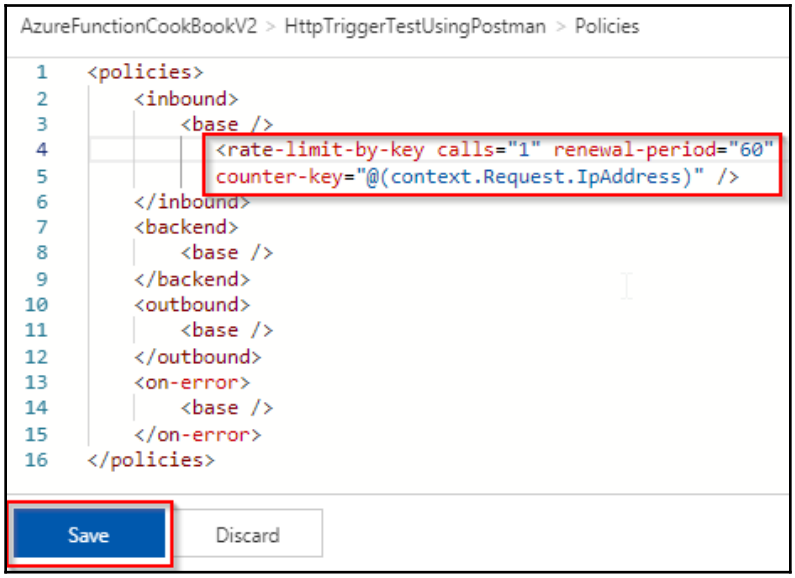
Configuring request throttling using inbound policies

Perform the following steps:

1. As shown in the preceding screenshot, choose the required operation (**GET**) and click on the inbound policy editor link (labeled 3), which will open the policy editor.

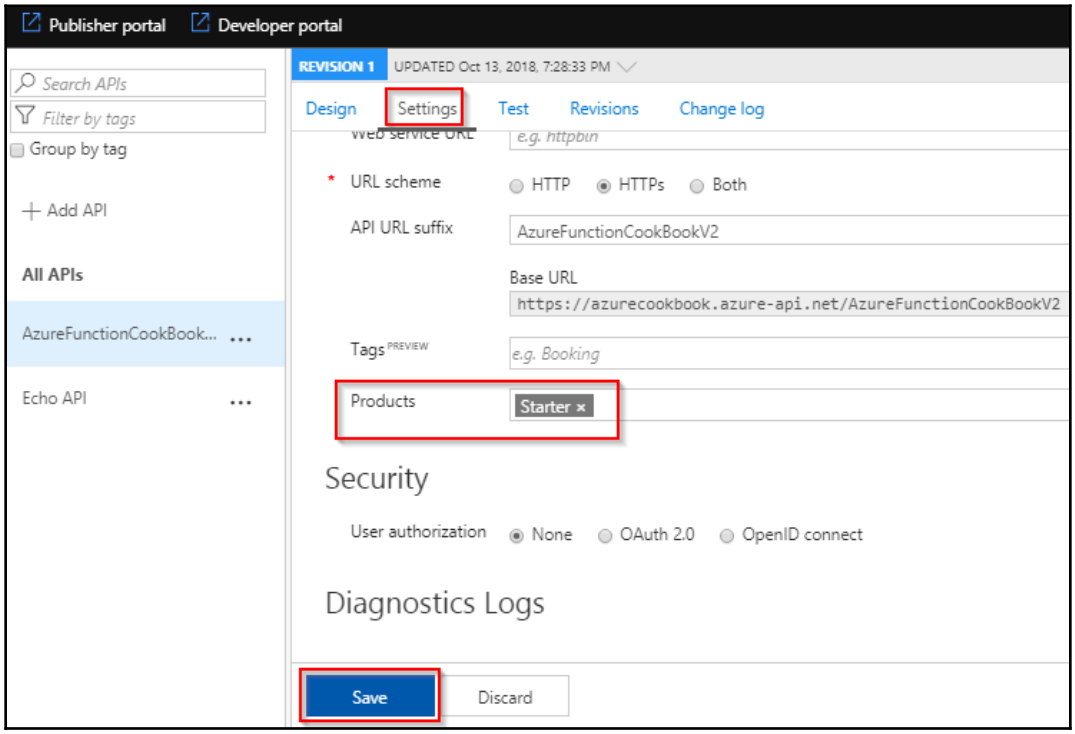
API Management allows us to control the behavior of the backend APIs (in our case, HTTP triggers) using API Management policies. You can control both the inbound and outbound request responses. You can read more about it at `https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-policies`.

2. As we need to restrict the request rate within API Management before sending the request to the backend function app, we need to configure the rate limit in the inbound policy. Create a new policy as shown with a value of 1 for the **calls** attribute and a value of `60` (in seconds) for the `renewal-period` attribute, and finally set the `counter-key` to the IP address of the client application:



With this inbound policy, we are instructing API Management to restrict one request per minute for a given IP address.

3. One final step before we test the throttling is publishing the API by navigating to the **Settings** tab in the preceding step and associating the API with a published product (in my case, I have a default starter product that is already published). As shown in the following screenshot, choose the required product and click on the **Save** button:



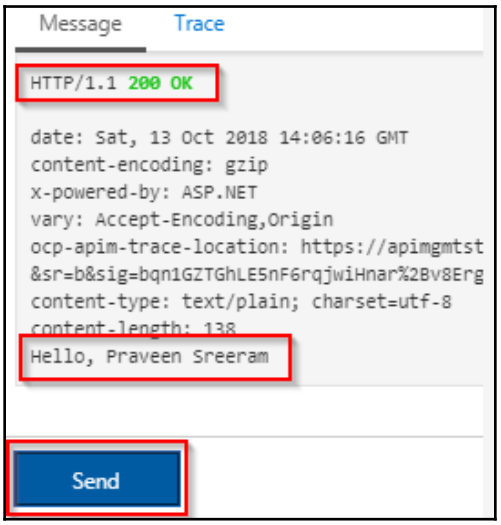
Products in API Management are a group of APIs to which the developers of different client applications can subscribe. For more information about API Management products, refer to `https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-add-products`.

Testing the rate limit inbound policy configuration

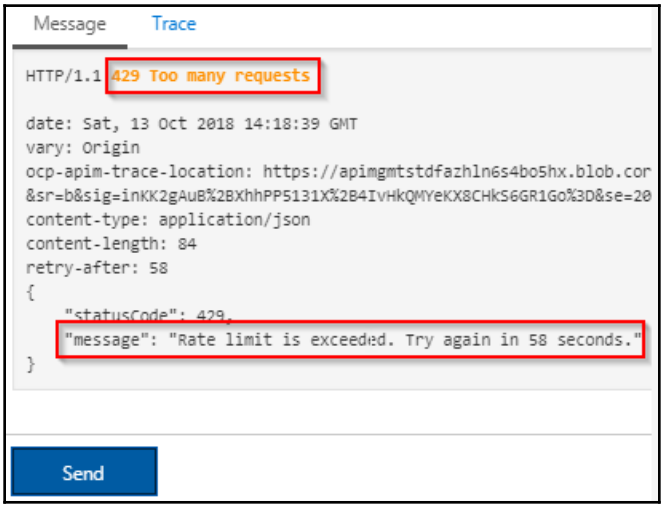
Perform the following steps:

1. Navigate to the **Test** tab and add any required parameters or headers that are expected by the HTTP trigger. In my case, my HTTP trigger requires a parameter named `name`.

2. Now, click on the **Send** button that appears when you complete the preceding step to make your first request. You should see something like the following after getting a response from the backend:

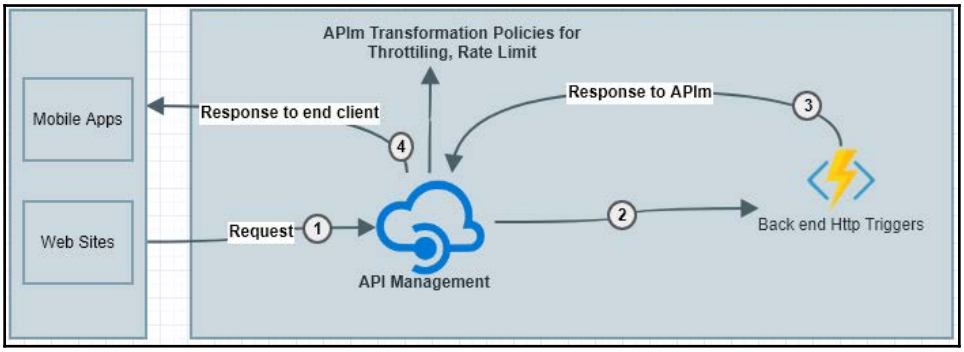


3. Now immediately click the **Send** button again. As shown here, you should see an error, as our inbound policy rule is to allow only one request per minute for a given IP address:



How it works...

In this recipe, we have created an Azure API Management instance and integrated an Azure Function App to leverage the API Management features. Once they were integrated, we created an inbound policy that restricts clients to just one call per minute from a given IP address. Here is a high-level diagram that depicts the whole process:



Securely accessing SQL Database from Azure Functions using Managed Service Identity

In one of our recipes, *Azure SQL Database interactions using Azure Functions*, from `Chapter 3`, *Seamless Integration of Azure Functions with Azure Services*, we learned how to access a SQL Database and its objects from Azure Functions by providing the connection string (username and password).

Let's say that, for some reason, you change the password to an account, meaning that any applications using that account wouldn't be able to gain access. As a developer, wouldn't it be good if there was a facility where you didn't need to worry about the credentials and, instead, the framework took care of authentication? In this recipe, we will learn how to access a SQL Database from an Azure Function without providing a user ID or password by using a feature called Managed Service Identity.

At the time of writing this recipe, the code related to retrieving the access token was available only with Azure Functions V1 (.NET framework) but not with V2 (.NET Core). By the time you are reading this book, it might be available in the latest version of the .NET Core framework, and so this recipe should work with Azure Functions v2 runtime as well.

Getting ready

This recipe requires us to create the Azure Functions (with the V1 runtime) and the SQL Database in the same resource group. If you haven't created these, create them and come back to this recipe to continue. Here are the steps that we will be performing in this recipe:

1. Creating a function app using Visual Studio 2017 with V1 runtime
2. Creating a Logical SQL Server and a SQL database
3. Enabling Managed Service Identity from the portal
4. Retrieving Managed Service Identity information using the Azure CLI
5. Allowing SQL Server access to the new Managed Service Identity
6. Executing the HTTP trigger and testing

How to do it...

We will perform this recipe using the following steps:

1. Create a function app using Visual Studio 2017 with the V1 runtime
2. Create a Logical SQL Server and a SQL Database
3. Enabling the Managed Service Identity

Creating a function app using Visual Studio 2017 with V1 runtime

Perform the following steps:

1. Create a new function app by choosing the **Azure Functions v1** runtime.
2. Once the HTTP trigger is created, replace the function with the following code:

```
public static class HttpTriggerWithMSI
    {
        [FunctionName("HttpTriggerWithMSI")]
        public static async Task<HttpResponseMessage>
Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route
= null)]HttpRequestMessage req, TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a
request.");

            string firstname = string.Empty, lastname =
string.Empty, email = string.Empty, devicelist = string.Empty;

            dynamic data = await req.Content.ReadAsAsync<object>();
            firstname = data?.firstname;
            lastname = data?.lastname;
            email = data?.email;
            devicelist = data?.devicelist;

            SqlConnection con = null;
            try
            {
                string query = "INSERT INTO EmployeeInfo
(firstname,lastname, email, devicelist) " + "VALUES
(@firstname,@lastname, @email, @devicelist) ";

                con = new
SqlConnection("Server=tcp:dbserver.database.windows.net,1433;Initia
l Catalog=database;Persist Security
Info=False;MultipleActiveResultSets=False;Encrypt=True;TrustServerC
ertificate=False;Connection Timeout=30;");
                SqlCommand cmd = new SqlCommand(query, con);

                con.AccessToken = (new
AzureServiceTokenProvider()).GetAccessTokenAsync("https://database.
windows.net/").Result;

                cmd.Parameters.Add("@firstname", SqlDbType.VarChar,
                 50).Value = firstname;
```

```
                cmd.Parameters.Add("@lastname", SqlDbType.VarChar,
50)
                 .Value = lastname;
                cmd.Parameters.Add("@email", SqlDbType.VarChar, 50)
                 .Value = email;
                cmd.Parameters.Add("@devicelist",
SqlDbType.VarChar)
                 .Value = devicelist;
                con.Open();
                cmd.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                throw ex;
            }
            finally
            {
                if (con != null)
                {
                    con.Close();
                }
            }
            return req.CreateResponse(HttpStatusCode.OK, "Hello,
Successfully inserted the data");
        }
```

The preceding code is the code that I copied from `Chapter 3`, *Seamless Integration of Azure Functions with Azure Services*, with the following changes, but the connection string doesn't have user ID and password details.

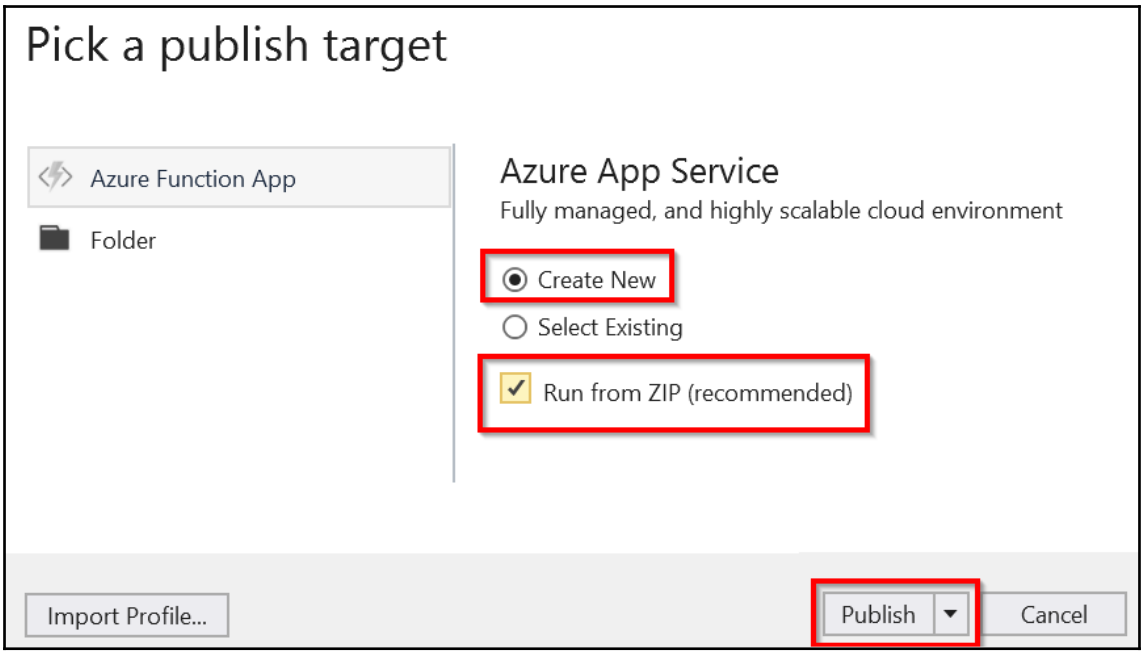
3. Add a new line of code to retrieve the access token:

```
con.AccessToken = (new
AzureServiceTokenProvider()).GetAccessTokenAsync("https://database.
windows.net/").Result;
```

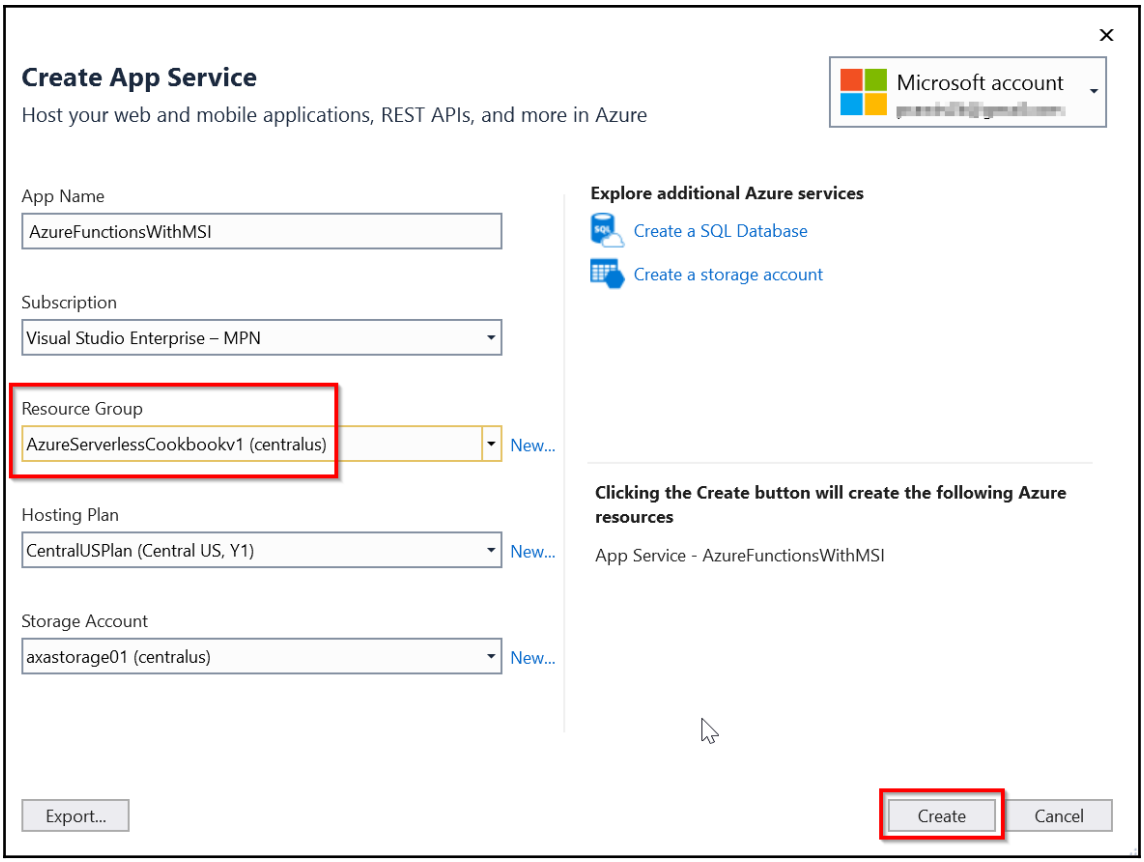
4. Add the following NuGet packages to the function app:

```
Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory
Install-Package Microsoft.Azure.Services.AppAuthentication
```

5. Once you have ensured that there are no build errors, publish the function app to Azure. This step ensures that we create the function app with V1 runtime. Click on the **Publish** button, which opens the popup as shown:



6. Next, provide the **Resource Group** and other details, as shown in the following screenshot, and click on the **Create** button:



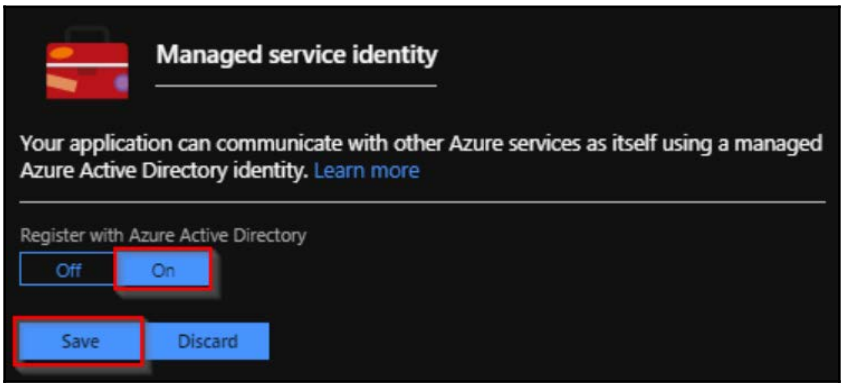
Creating a Logical SQL Server and a SQL Database

Create a SQL Server and a SQL database in the same resource group where you have created the Azure function app. In my case, my resource group name is `AzureServerlessCookbookv1`.

Enabling the managed service identity

Perform the following steps:

1. Navigate to the **Platform features** of the function app and click on **Managed service identity**
2. In the **Managed service identity** tab, click on **On** and **Save**, as shown:



Retrieving Managed Service Identity information

Perform the following steps:

1. Authenticate your Azure Account's identity using Azure CLI by running the `az login` command in the Command Prompt, as shown in the following screenshot:

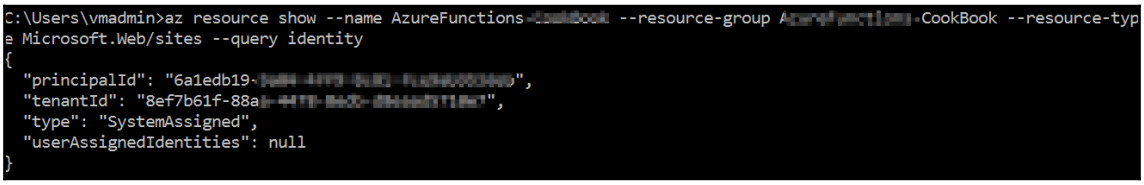
```
C:\Users\vmadmin>az login
Note, we have launched a browser for you to login. For old experience with device code, use "az login --use-device-code"
```

2. You will be prompted to provide your Azure account credentials to log into the Azure portal. Once you have provided your credentials, it will show you the available subscriptions in the command console.

3. Now we need to retrieve the service principle details by running the following command:

```
az resource show --name <<Function App Name>> --resource-group
<<Resource Group>> --resource-type Microsoft.Web/sites --query
identity
```

4. If you have configured the managed identity properly, you will see something similar to the following as the output to the preceding command:



5. Make a note of `principalId`, which is retrieved in the preceding step. We will be using it in the next section.

Allowing SQL Server access to the new Managed Identity Service

In this section, we will create an admin user that has access to the SQL Server that we created earlier:

1. Run the following command in the Command Prompt by passing the `principalId` that you noted in the previous section:

```
az sql server ad-admin create --resource-group AzureServerlessCookbookv1 --
server-name azuresqlmsidbserver --display-name sqladminuser --object-id
<Principe Id>
```

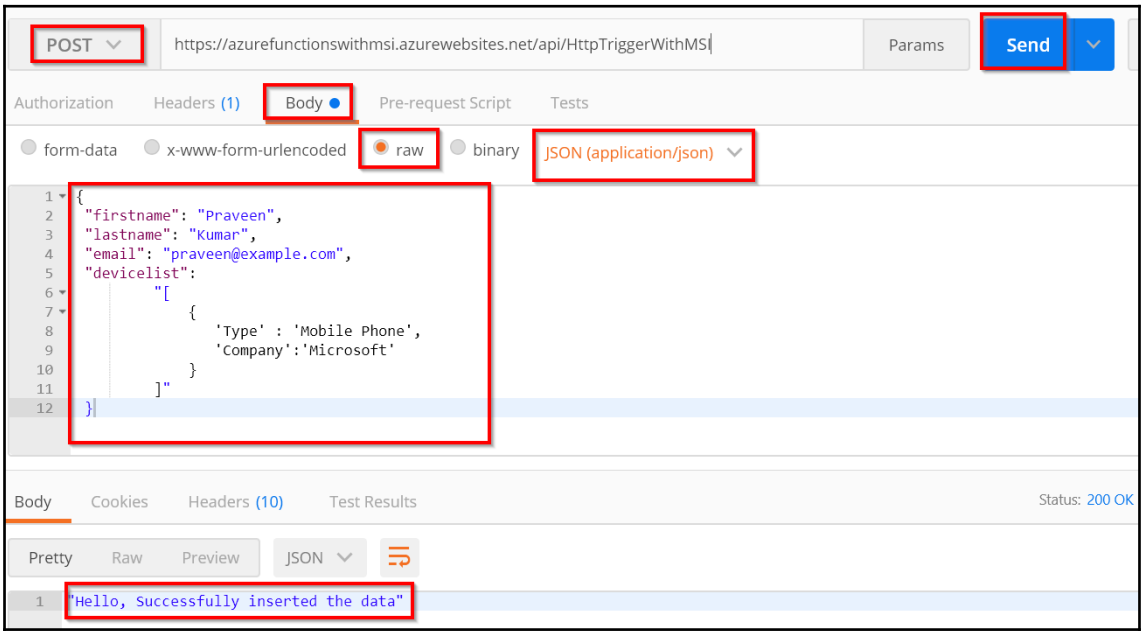
2. Running the preceding command creates a new admin user in the **master** database of the SQL Server.
3. Create a table named `EmployeeInfo` using the following script:

```
CREATE TABLE [dbo].[EmployeeInfo](  [PKEmployeeId] [bigint] IDENTITY(1,1)
NOT NULL,  [firstname] [varchar](50) NOT NULL,  [lastname] [varchar](50)
NULL,  [email] [varchar](50) NOT NULL,  [devicelist] [varchar](max) NULL,
CONSTRAINT [PK_EmployeeInfo] PRIMARY KEY CLUSTERED  (  [PKEmployeeId] ASC
)  )
```

Executing the HTTP trigger and testing

Perform the following steps:

1. Open **Postman** and submit a request as shown:



2. Let's review the SQL Database to see whether the record is inserted:

There's more...

Ensure that you have all the following namespaces in the class:

```
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Data.SqlClient;
using System.Data;
using System;
using Microsoft.Azure.Services.AppAuthentication;
```

See also

See the *Azure SQL Database interactions using Azure Functions* recipe of `Chapter 3`, *Seamless Integration of Azure Functions with Azure Services*.

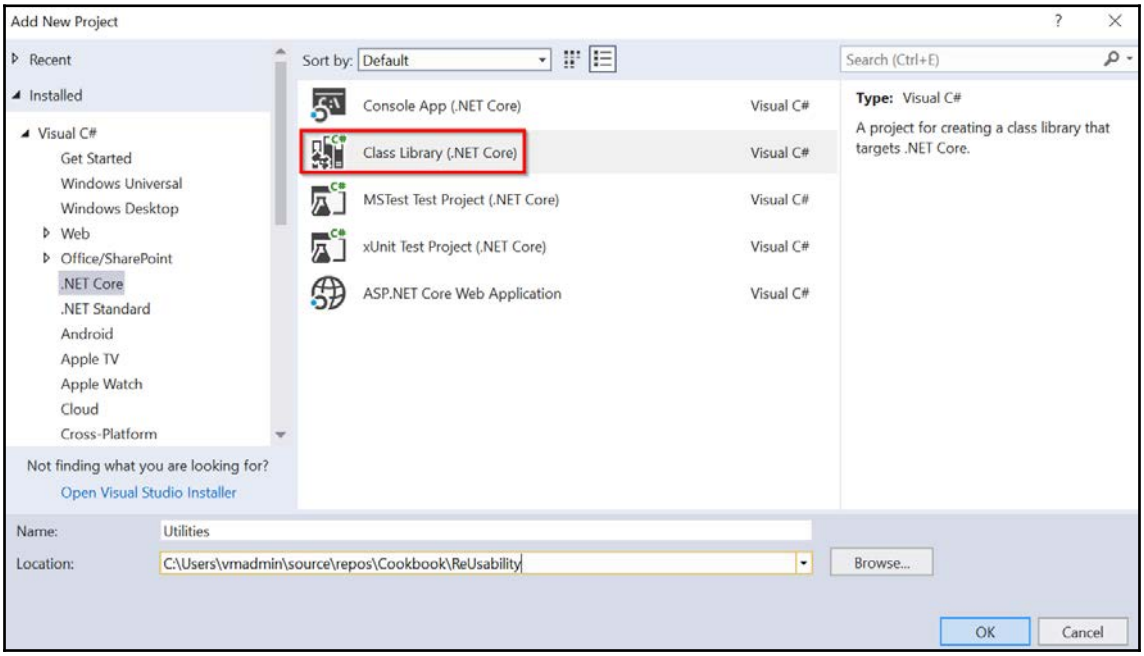
Shared code across Azure Functions using class libraries

You have learned how to reuse a `Helper` method within an Azure function app. However, you cannot reuse it across other function apps or any other type of application, such as a web app or a WPF application. In this recipe, we will develop and create a new `.dll` file and you will learn how to use the classes and their methods in Azure Functions.

How to do it...

Perform the following steps:

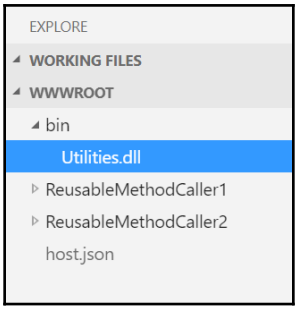
1. Create a new **Class Library** application using Visual Studio. I have used Visual Studio 2017:



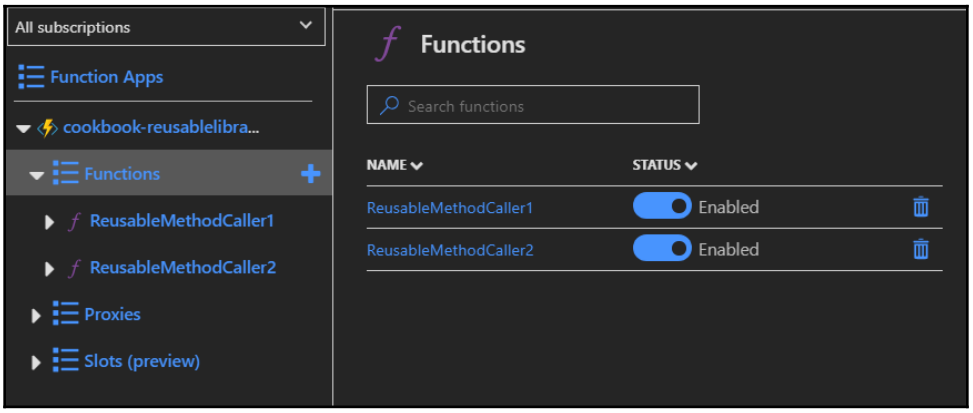
2. Create a new class named `Helper` and paste the following code in the new class file:

```
namespace Utilities
{
    public class Helper
    {
        public static string GetReusableFunctionOutput()
        {
            return "This is an output from a Resuable Library
across functions";
        }
    }
}
```

3. Change **Build Configuration** to **Release** and build the application to create the `.dll` file, which will be used in our Azure Functions.
4. Navigate to the **App Service Editor** of the function app by clicking on the **App Service Editor** button, which is available under **Platform Features**.
5. Now create a new `bin` folder by right-clicking in the empty area below the files located in **WWWROOT.**
6. After clicking on the **New Folder** item in the obtained screen, a new textbox will appear, wherein you will need to provide the name as `bin`.
7. Next, right-click on the `bin` folder and select the `Upload Files` option to upload the `.dll` file that we created in Visual Studio
8. This is how it looks after we upload the `.dll` file to the `bin` folder:



9. Navigate to the Azure Function where you would like to use the `shared` method. To demonstrate, I have created two Azure Functions (one HTTP trigger and one timer trigger):



10. Let's navigate to the `ReusableMethodCaller1` function and make the following changes:
    - Add a new `#r` directive, as follows, to the `run.csx` method of the `ReusableMethodCaller1` Azure Function. Note that `.dll` is required in this case:

      ```
      #r "../bin/Utilities.dll"
      ```

    - Add a new namespace, as follows:

      ```
      using Utilities;
      ```

11. We are now ready to use the `GetReusableFunctionOutput` shared method in our Azure Function. Now replace the code of the HTTP trigger with the following:

    ```
    log.LogInformation(Helper.GetReusableFunctionOutput());
    ```

12. When you run the application, you should see the following message in the logs:

```
2018-11-20T03:24:36.696 [Information] Compilation succeeded.
2018-11-20T03:24:42.914 [Information] Executing 'Functions.ReusableMethodCaller1' (Reason='This function was
programmatically called via the host APIs.', Id=556b0d73-07ed-4d34-bac3-4ad8252adbf9)
2018-11-20T03:24:42.936 [Information] C# HTTP trigger function processed a request.
2018-11-20T03:24:42.936 [Information] This is an output from a Resuable Library across functions
2018-11-20T03:24:42.937 [Information] Executed 'Functions.ReusableMethodCaller1' (Succeeded, Id=556b0d73-07ed-
4d34-bac3-4ad8252adbf9)
```

13. Repeat the same steps of adding the reference and the namespace of the utilities library even in the second Azure Function, `ReusableMethodCaller2`. If you have made the changes successfully, you should see something like what follows:

```
2018-11-20T03:29:27  Welcome, you are now connected to log-streaming service.
2018-11-20T03:29:53.251 [Information] Script for function 'ReusableMethodCaller2' changed. Reloading.
2018-11-20T03:29:53.385 [Information] Compilation succeeded.
2018-11-20T03:29:59.994 [Information] Executing 'Functions.ReusableMethodCaller2' (Reason='Timer fired at 2018-11-
20T03:29:59.9938806+00:00', Id=357d4c51-1920-47f0-91d7-87aad0611955)
2018-11-20T03:30:00.075 [Information] C# Timer trigger function executed at: 11/20/2018 3:30:00 AM
2018-11-20T03:30:00.075 [Information] This is an output from a Resuable Library across functions
2018-11-20T03:30:00.075 [Information] Executed 'Functions.ReusableMethodCaller2' (Succeeded, Id=357d4c51-1920-47f0-91d7-87aad0611955)
```

How it works...

We have created a `.dll` file that contains the reusable code that can be used in any of the Azure Functions that require the functionality made available by the `.dll` file.

Once the `.dll` file was ready, we created a `bin` folder in the function app and added the `.dll` file to the `bin` folder.

Note that we have added the `bin` folder to the **WWWROOT** so that it is available to all the Azure Functions available in the function app.

There's more...

If you would like to use the shared code only in one function, then you would need to add the `bin` folder along with the `.ddl` file in the required `Azure Function` folder.

Another major advantage of using class libraries it that it improves performance, as they are already compiled and ready for execution.

Using strongly typed classes in Azure Functions

In our initial chapters, we developed an HTTP trigger named `RegisterUser` that acts as a web API and can be consumed by any application that's capable of making HTTP requests. However, there might be some other requirements, where you might have different applications that create messages in a queue with the details required for creating a user. For the sake of simplicity, we will be using Azure Storage Explorer to create a queue message.

In this recipe, we will look at how to get the details of the user from the queue using strongly typed objects.

Getting ready

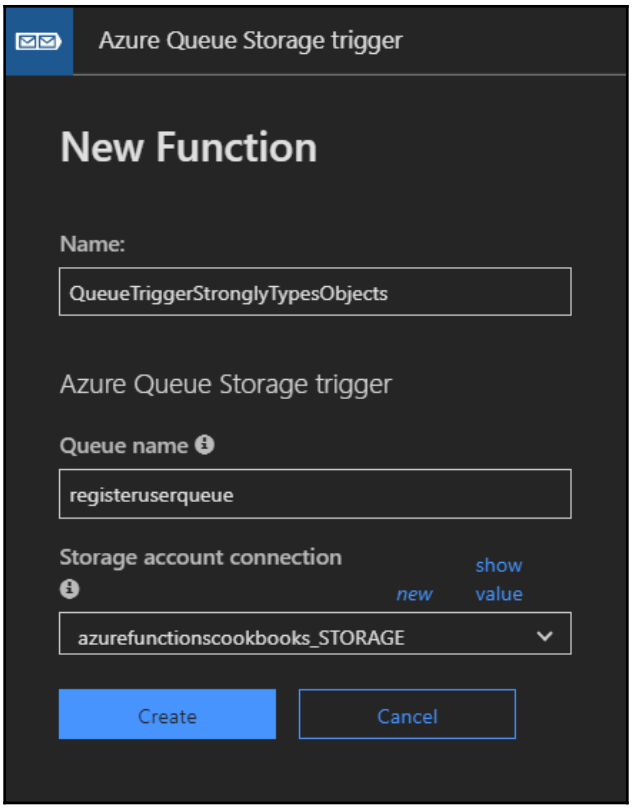
Before moving further, perform the following steps:

1. Create a storage account (I have created `azurefunctionscookbook`) in your Azure subscription
2. Install Microsoft Azure Storage Explorer if you haven't installed it already
3. Once Storage Explorer has been created, connect to your Azure storage account

How to do it...

Perform the following steps:

1. Using the Azure Storage Explorer, create a queue named `registeruserqueue` in the storage account named `azurefunctionscookbook`. We assume that all the other applications would be creating messages in the `registeruserqueue` queue.
2. Navigate to **Azure Functions** and create a new Azure Function using **Azure Queue Storage trigger**, then choose the queue that we have created.
3. You might be prompted to install storage extensions if not installed already. Once you install the extensions, provide the details of the queue and click on the **Create** button, as shown in the following screenshot:

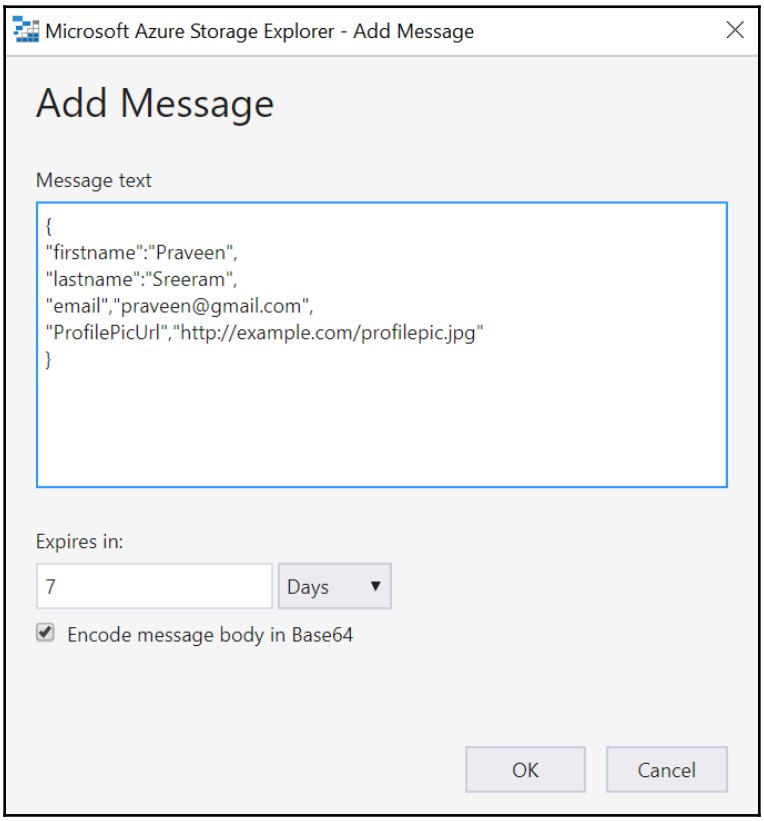


4. Once the function is created, replace the default code with the following code. Whenever a queue message is created, the JSON message will be deserialized automatically and populated in an object named `myQueueItem`. In the following code, we are just printing the values of the objects in the **Logs** window:

```
using System;
public static void Run(User myQueueItem, ILogger log)
{
    log.LogInformation($"A Message has been created for a new
User");
    log.LogInformation($"First name: {myQueueItem.firstname}" );
    log.LogInformation($"Last name: {myQueueItem.lastname}" );
    log.LogInformation($"email: {myQueueItem.email}" );
    log.LogInformation($"Profile Url: {myQueueItem.ProfilePicUrl}"
);
}
public class User
{
```

```
        public string firstname { get;set;}
        public string lastname { get;set;}
        public string email { get;set;}
        public string ProfilePicUrl { get;set;}
    }
```

5. Navigate to **Azure Storage Explorer** and create a new message in `registeruserqueue`, as shown in the following screenshot:



6. Click on **OK** to create the queue message and navigate back to the Azure Function and look at the logs, as shown in the following screenshot:

```
2018-11-20T00:37:51.745 [Information] Executing 'Functions.QueueTriggerStronglyTypesObjects' (Reason='New queue message detected on
'registeruserqueue'.', Id=e043cc8e-dd49-42e7-8dd7-dd34db6021a6)
2018-11-20T00:37:51.747 [Information] A Message has been created for a new User
2018-11-20T00:37:51.747 [Information] First name: Praveen
2018-11-20T00:37:51.747 [Information] Last name: Sreeram
2018-11-20T00:37:51.747 [Information] email: praveen@gmail.com
2018-11-20T00:37:51.747 [Information] Profile Url: http://example.com/profilepic.jpg
2018-11-20T00:37:51.748 [Information] Executed 'Functions.QueueTriggerStronglyTypesObjects' (Succeeded, Id=e043cc8e-dd49-42e7-8dd7-
dd34db6021a6)
```

How it works...

We have developed a new queue function that gets triggered when a new message gets added to the queue. We have created a new queue message with all the details required to create the user. You can further reuse the Azure Function code to pass the user object (in this case, `myQueueItem`) to the database layer class, which is capable of inserting the data into a database or any other persistent medium.

There's more...

In this recipe, the type of the queue message parameter that was accepted by the `Run` method was `User`. The Azure Functions runtime will take care of deserializing the JSON message available in the queue to the custom type; `user`, in our case.

10
Configuring of Serverless Applications in the Production Environment

In this chapter, we will learn the following recipes:

- Deploying Azure Functions using Run From Package
- Deploying Azure Function using ARM templates
- Configuring custom domain to Azure Functions
- Techniques to access Application Settings
- Creating and generating open API specifications using Swagger
- Breaking down large APIs into small subsets of APIs using proxies
- Moving configuration items from one environment to another using resources

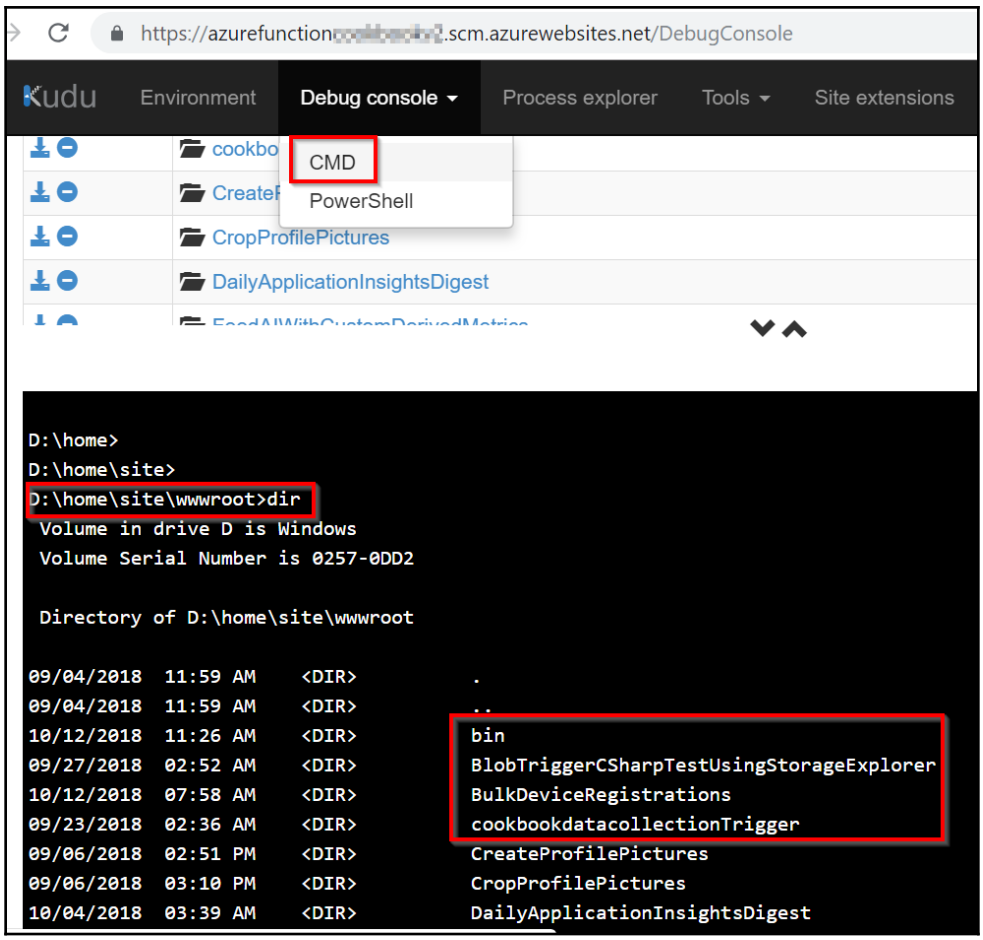
Introduction

We have been discussing all the different features of Azure Functions that help developers quickly build backend applications. This chapter's focus is on the configurations that one needs to make in a non-development environment (such as Staging, UAT, and production).

Deploying Azure Functions using the Run From Package

We have been learning about different techniques for developing Azure Functions and deploying it to the cloud.

As you might already be aware, each function app can have multiple functions hosted in it. All the code related to those functions will be located in the `D:\home\site\wwwroot` folder:



`D:\home\site\wwwroot` is the location where the runtime would look for the binaries and all the configuration files that are required for executing the application.

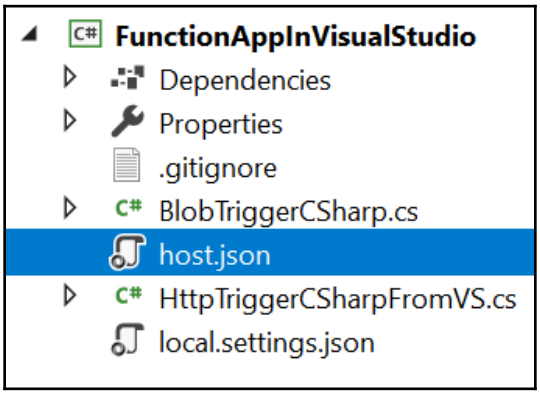
In this recipe, we will learn another new technique, called **Run From Package** (earlier called **Run From Zip**) to deploy the Azure Function as a package.

Using Run From Package, we can change the default location to an external storage account.

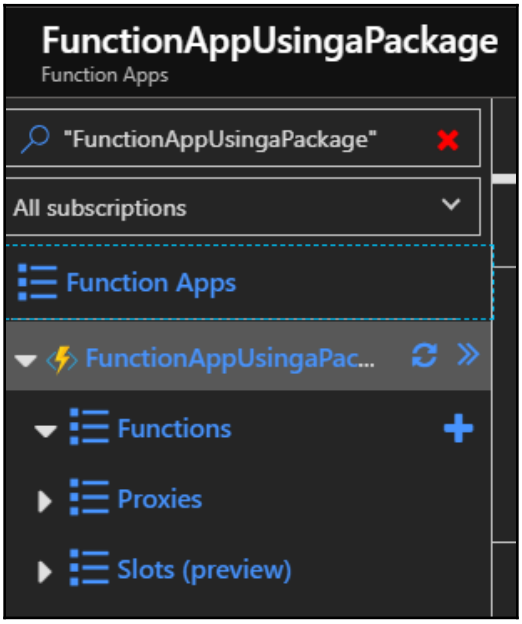
Getting ready

We need the following to complete this recipe:

1. Visual Studio 2017 installed in your local developer machine; create one or more Azure Functions using Visual Studio. For this example, I have created one HTTP trigger and one timer trigger:



2. Create an empty function app using the Azure Management portal:

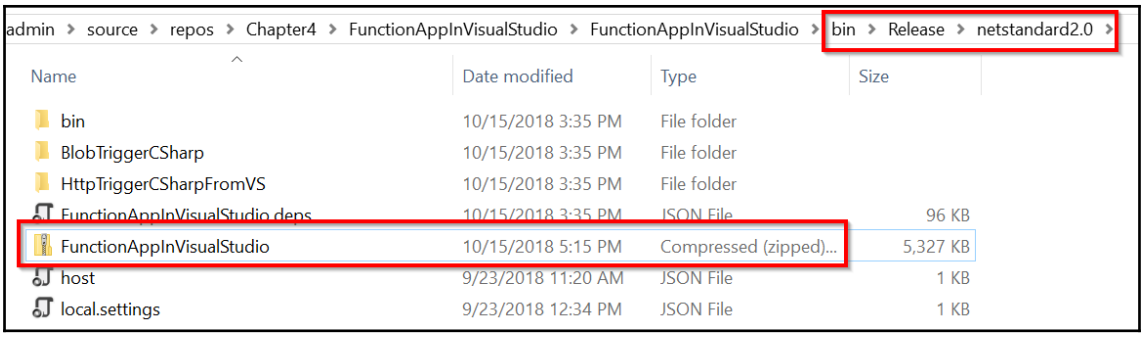


3. A storage account—we will upload the package file to the storage account.

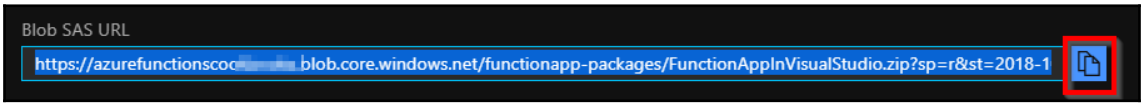
How to do it...

Perform the following steps:

1. Create a package file for the application. I'm using the same application that we created in `Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools*.
2. Navigate to the location where you see the `bin` folder along with other files related to your functions. Create a `.zip` file out of the files, which is highlighted in the following screenshot:



3. Create a Blob container (with private access) and upload the package file either from the portal or by using Azure Storage Explorer.
4. The next step is to generate a **shared access signature** (**SAS**) token for the Blob Container so that Azure Function runtime has the required permissions to access the files located in the container. You can learn more about SAS at `https://docs.microsoft.com/en-us/azure/storage/common/storage-dotnet-shared-access-signature-part-1`:



5. Navigate to the **Application settings** of the function app that we created, and create a new app setting with the `WEBSITE_RUN_FROM_PACKAGE` key and the value to be the **Blob SAS URL** that you created in the previous step, as shown here. Click on **Save** to save the changes:

6. That's it. After the preceding configuration, you can test the function:

How it works...

When the Azure Function runtime finds an app setting with the name `WEBSITE_RUN_FROM_PACKAGE`, it understands that it should look up the packages in the storage account. So, on the fly, the runtime downloads the files and uses them to launch the application.

There's more...

You can learn more about this technique and its advantages at `https://github.com/Azure/app-service-announcements/issues/84`.

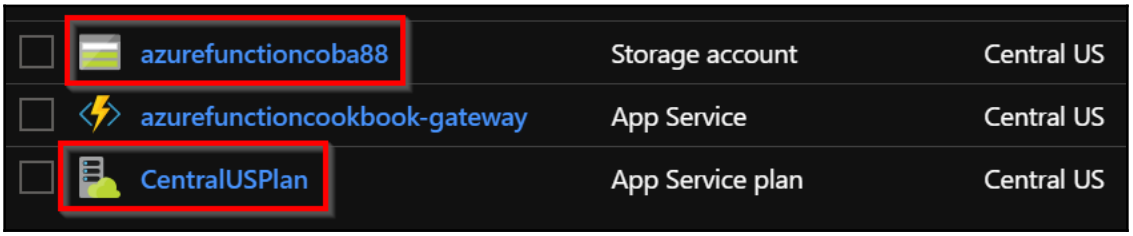
Deploying Azure Function using ARM templates

So far, we have been manually provisioning Azure Functions using the Azure Management portal.

In this recipe, we will learn how to automate the process of provisioning the Azure Function using **Azure Resource Manager** (**ARM**) templates.

Getting ready

Before, we start authoring the ARM templates, we need to understand the other Azure services on which the Azure Function depends. The following services are automatically created when you create a function app:



As shown in the preceding screenshot, the function app (in this case `azurefunctioncookbook-gateway`) is dependent on an **App Service Plan** and a Storage Account:

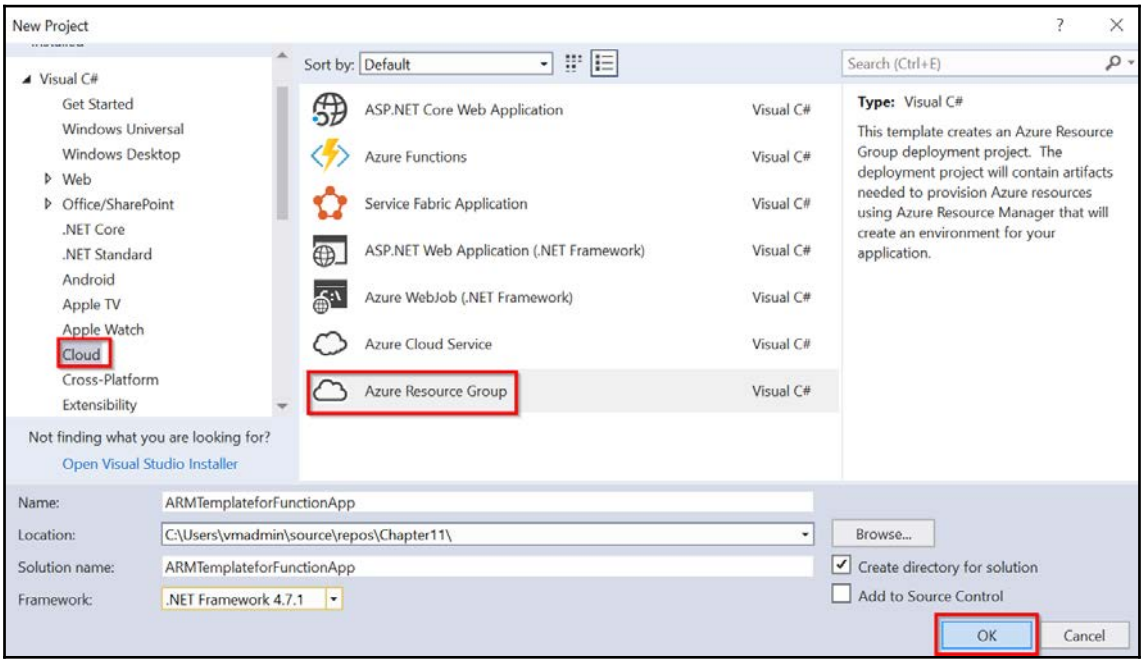
- **App Service plan**: This could be either a regular **App Service plan** or a **Consumption plan**.
- **Storage account**: Azure Function runtime uses the Storage account to log diagnostic information that we can use for troubleshooting.
- **Application Insights**: An optional Application Insights account. If we are not using Application Insights, we need to create an application setting with the name `AzureWebJobsDashboard` in the application settings of the function that uses Azure Storage Table Service to log diagnostic information.

Along with these services, we would obviously need to have a resource group. In this recipe, we will assume that the resource group already exists.

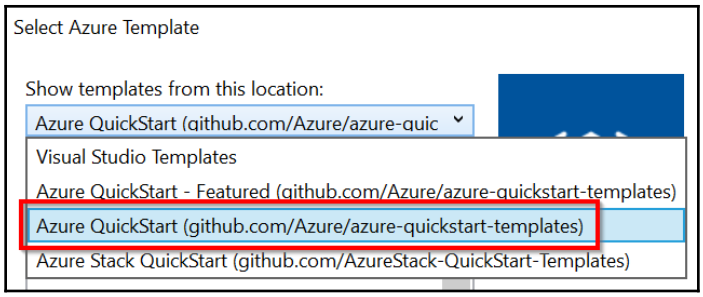
How to do it...

By now, you know that while authoring Azure Functions, we need to ensure that we also accommodate an App Service plan and a storage account. Let's start authoring the ARM template using Visual Studio:

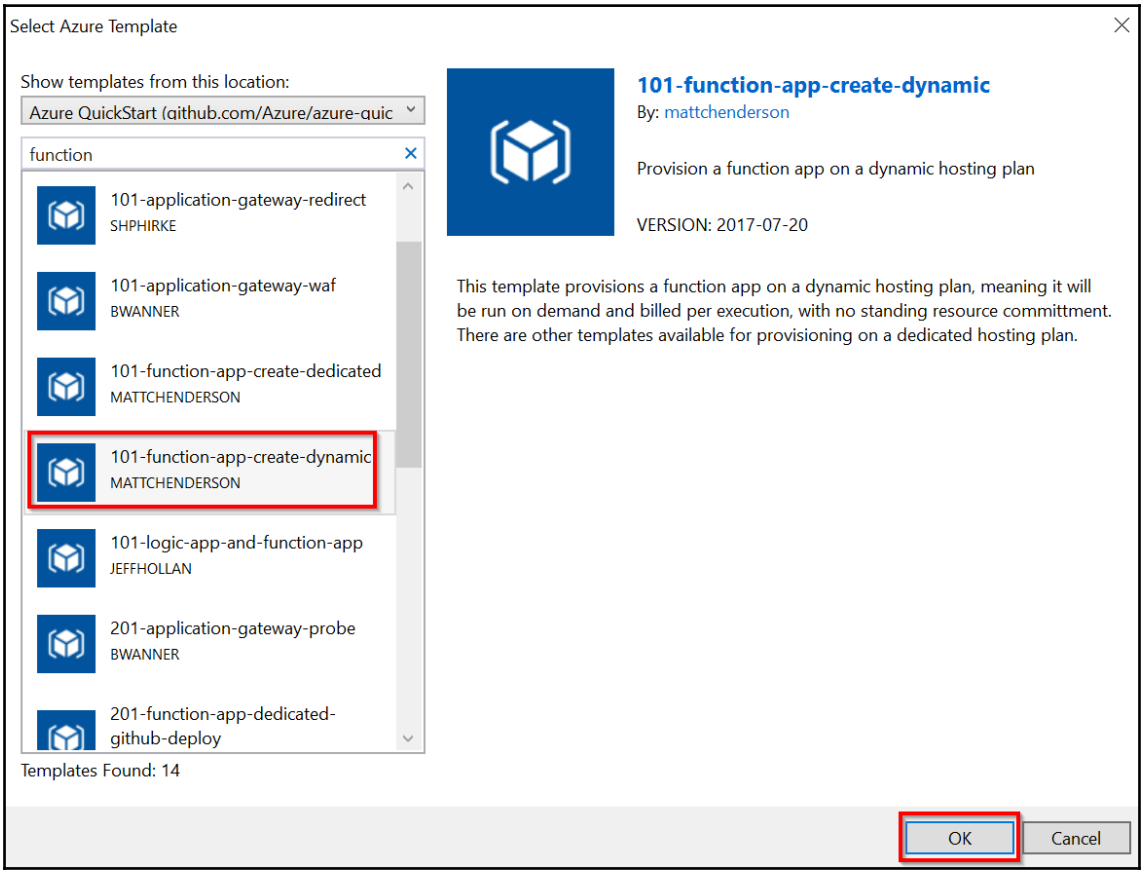
1. Create a new project by choosing **Visual C#** | **Cloud** and then choose **Azure Resource Group**:



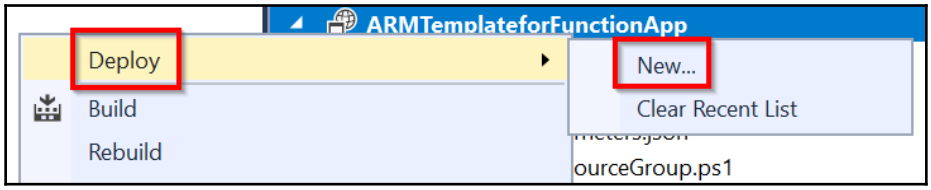
2. Clicking on the **OK** button in the previous step will open up the **Select Azure Template** where you choose the **Azure QuickStart (github.com/Azure/azure-quickstart-templates)** templates:



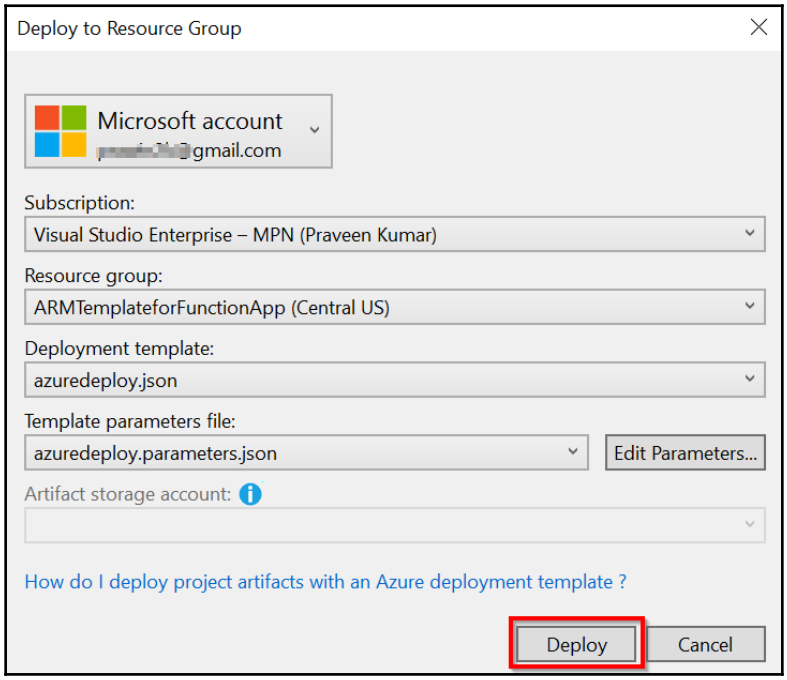
3. Search for the word `function` and click on the **101-function-app-create-dynamic** template to create the Azure function app with the **Consumption plan**:



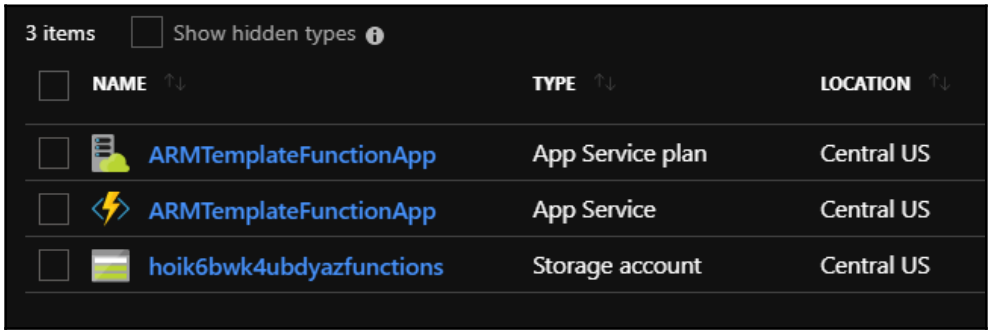
4. The required JSON template will be created in Visual Studio. You can learn more about the JSON content at `https://docs.microsoft.com/en-us/azure/azure-functions/functions-infrastructure-as-code`.
5. Deploy the ARM for provisioning the function app and its dependent resources. You can deploy it by right-clicking on the project name (in my case `ARMTemplateforFunctionApp`), clicking on **Deploy,** and then clicking on the **New** button:



6. Choose the **Subscription**, **Resource group**, and other parameters for provisioning the function app. Choose all the mandatory fields and click on the **Deploy** button:



7. That's it! In a few minutes, the deployment will start and each of the resources mentioned in the ARM JSON templates will be provisioned:



There's more...

Here are some of the advantages of provisioning Azure Resources using ARM templates:

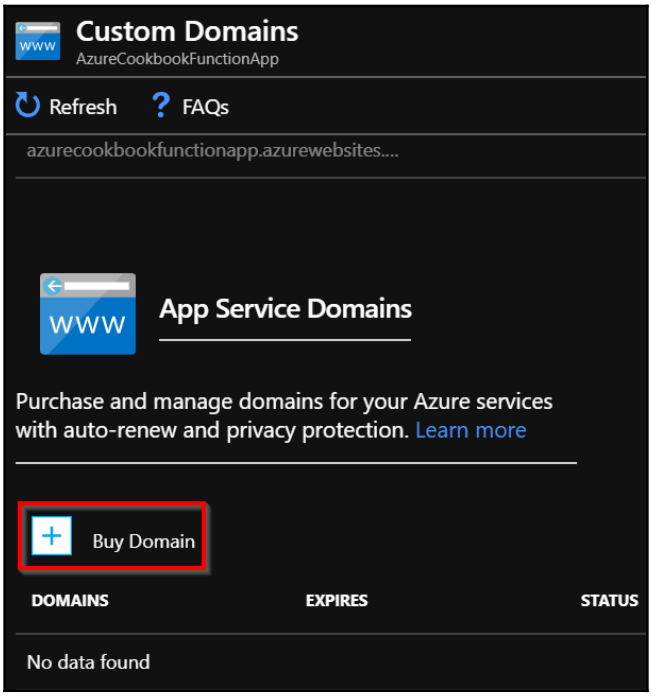
- By having the configurations in the JSON files, it's helpful for developers to push the files into some kind of version-control system, such as Git or TFS, so that we can maintain the versions of the files to track all the changes.
- It's also possible to create the services in different environments in no time.
- With the ARM templates, we can also push them in CI/CD pipelines to automate the provisioning for additional environments.

Configuring custom domain to Azure Functions

By now, looking at the default URL in the `functionappname.azurewebsites.net` format of the Azure function app, you might be wondering whether it's possible to have a separate domain instead of the default domain, as your customers might have their own domains. Yes, it's possible to configure a custom domain for the function apps. In this recipe, we will learn how to configure it.

Getting ready

Create a domain with any of the domain registrars. You can also purchase a domain right from the portal using the **Buy Domain** button, which is available in the **Custom Domains** blade:



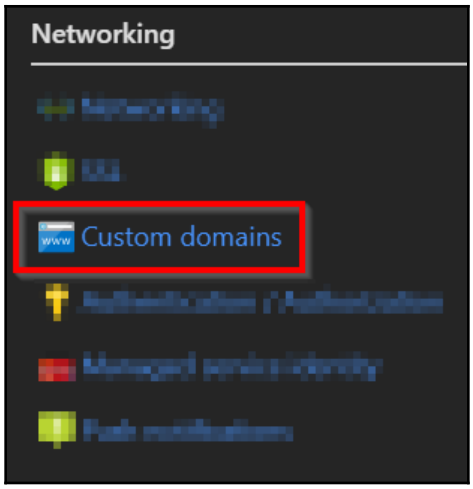
Once your domain is ready, create the following DNS records using the domain registrar:

- A record
- CName record

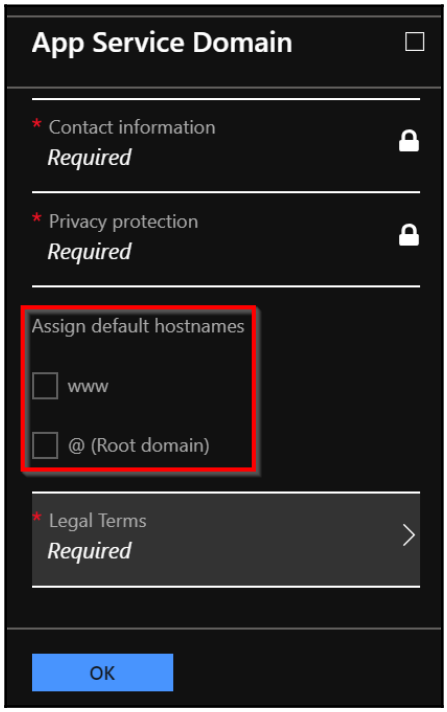
How to do it...

Perform the following steps:

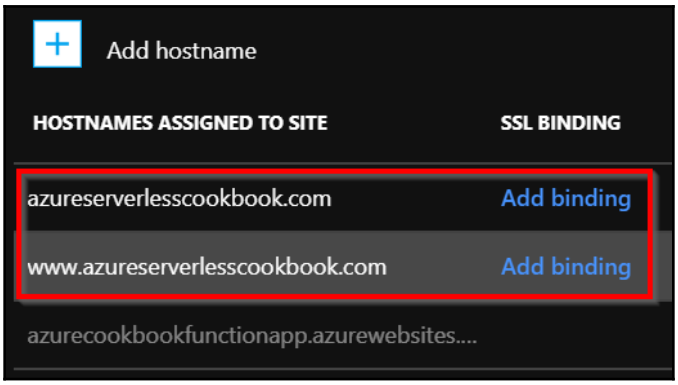
1. Navigate to the **Custom domains** blade of the Azure function app for which you would like to configure a domain:



2. If you have created a custom domain from the Azure portal, it will prompt you to choose the hostnames, as shown in the **App Service Domain** blade:



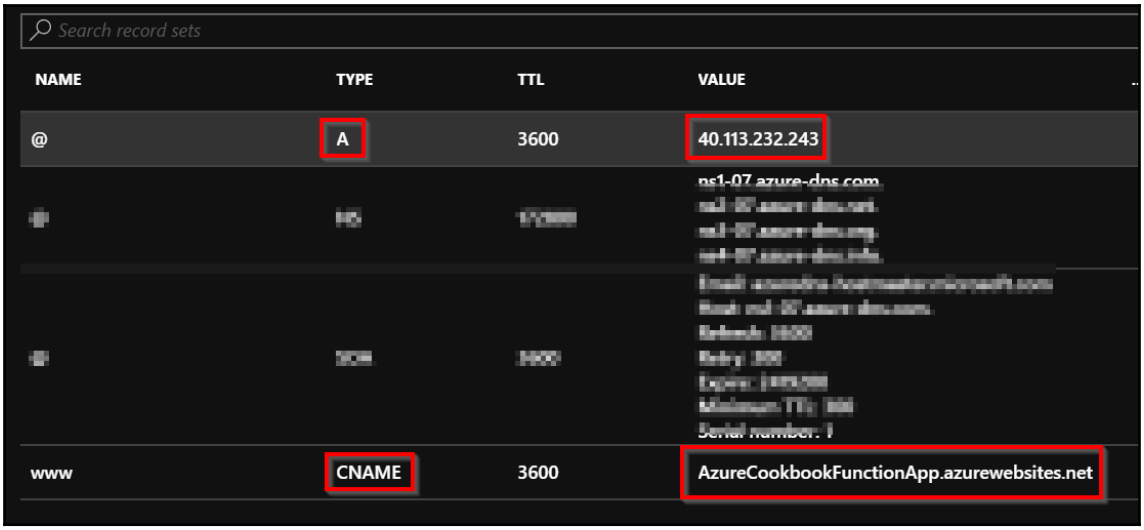
3. If you have chosen both of them, then that's it. All the work in integrating the function app and the custom domain is pretty much done for you by the Azure Management portal. You can view the integration of the hostnames here:



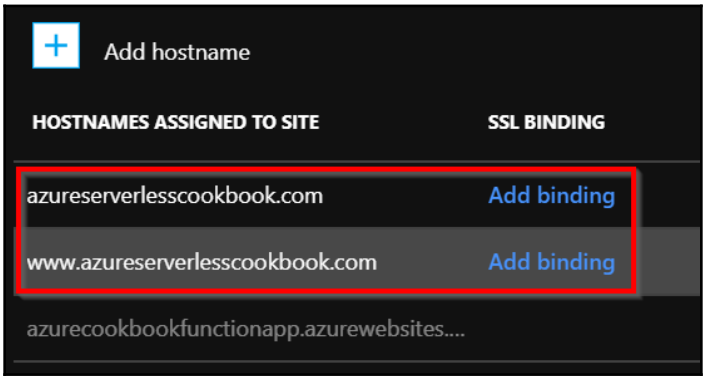
Configuring function app with an existing domain

If you already have a custom domain and you would like to integrate it with the function app, you need to create the following two records in the DNS:

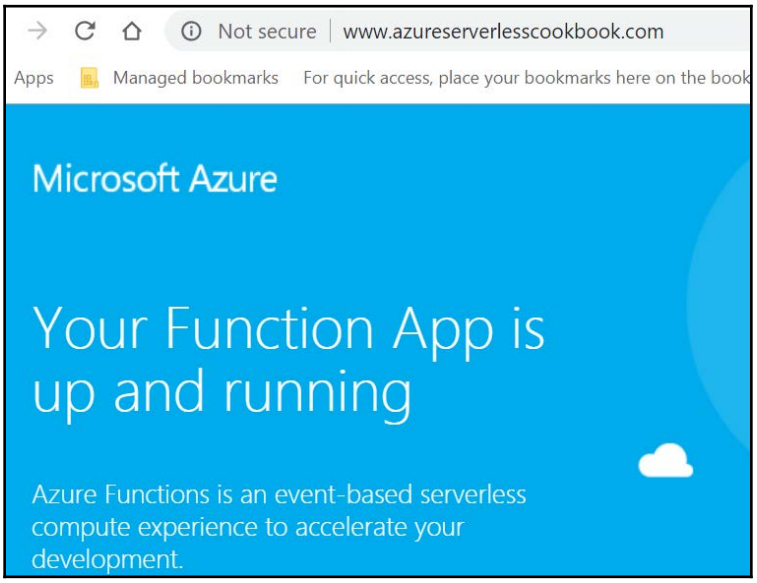
1. Create a A record and CNAME record in the domain registrar. You can get the IP address from the **Custom Domains** blade:



2. Navigate to the **Custom Domains** blade of the function app and create the following hostnames:



That's it. You have integrated a custom domain with the Azure function app. You can now browse your function app using the new domain instead of the default one that Azure provides you:



Techniques to access Application Settings

In every application, you will have at least a few configuration items that you might not want to hardcode. Instead, you would want them to change in the future, after the application goes live, without touching the code.

In general, I would classify the configuration items into two categories:

- Some of the configuration items might be different across environments, for example, the connection strings of the database and SMTP server
- Some of them might be the same across environments, such as some constant numbers that are used in some calculations in the code

Whatever might be the use of the configuration value, you need to have a place to store them that is accessible to your application.

In this recipe, we will learn how and where to store these configuration items and different techniques to access them from your application code.

Getting ready

Create an Azure Function with the V2 Function runtime if not created already. I will use the function app that we created in `Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools*.

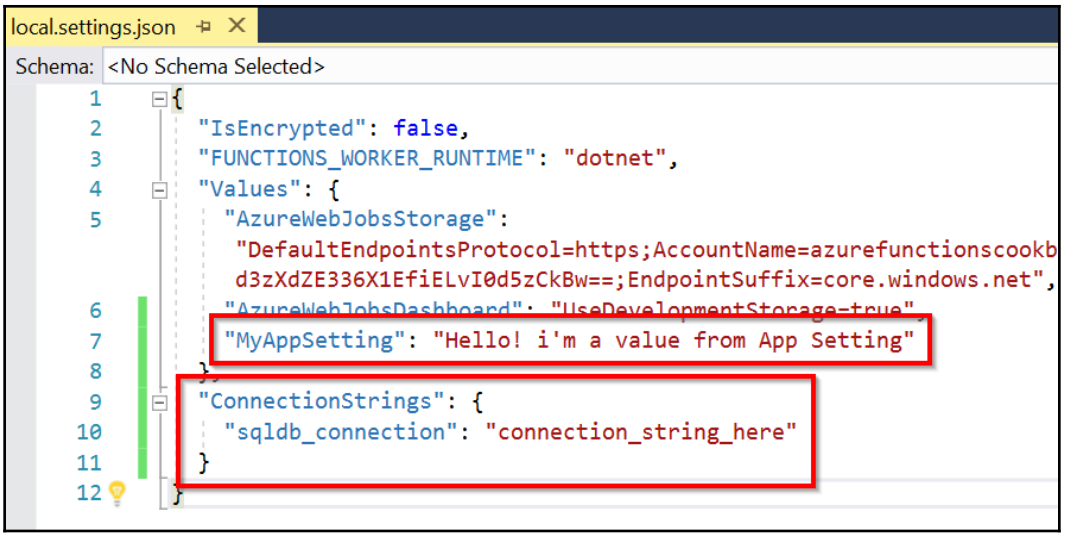
How to do it...

In this recipe, we will look at a few ways of accessing the configuration values.

Accessing Application Settings and connection strings in the Azure Function code

Perform the following steps:

1. Create a configuration items with the `MyAppSetting` key and a `ConnectionStrings` with the `sqldb_connection` key in the `local.settings.json` file. `local.settings.json` should look something like the following screenshot:

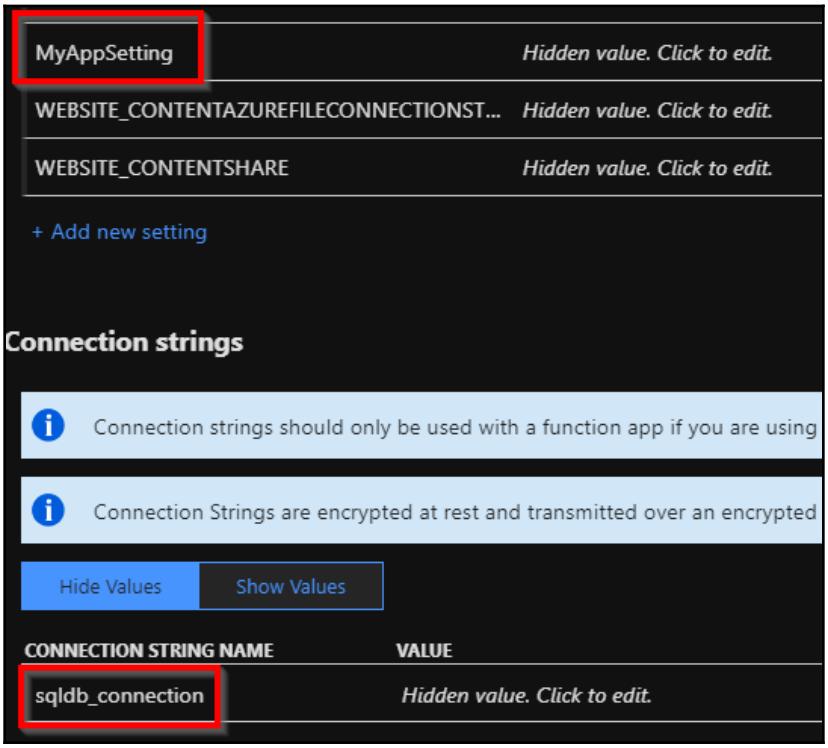


2. Replace the existing code with the following code. We have added a few lines that read the configuration values and the connection strings:

```
public class HttpTriggerCSharpFromVS
 {
 [FunctionName("HttpTriggerCSharpFromVS")]
 public static IActionResult
Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route
= null)]HttpRequest req,ILogger logger)
 {
 var configuration = new ConfigurationBuilder()
 .AddEnvironmentVariables()
 .AddJsonFile("appsettings.json", true)
 .Build();
var ValueFromGetConnectionStringOrSetting =
configuration.GetConnectionStringOrSetting("MyAppSetting");
 logger.LogInformation("GetConnectionStringOrSetting" +
ValueFromGetConnectionStringOrSetting);
var ValueFromConfigurationIndex= configuration["MyAppSetting"];
 logger.LogInformation("ValueFromConfigurationIndex" +
ValueFromConfigurationIndex);
var ValueFromConnectionString =
configuration.GetConnectionStringOrSetting("ConnectionStrings:sqldb
_connection");
 logger.LogInformation("ConnectionStrings:sqldb_connection" +
ValueFromConnectionString);
string name = req.Query["name"];
 return name != null ? (ActionResult)new OkObjectResult($"Hello,
```

```
{name}")
 : new BadRequestObjectResult("Please pass a name on the query
string or in the request body");
 }
 }
```

3. Publish the project to Azure by right-clicking on the project and then clicking on **Publish** in the menu.
4. Add the configuration key and the connection string in the **Application Settings** blade:



5. Run the function by clicking on the **Run** button, which logs the output in the **Output** window:

```
2018-10-20T13:17:33  Welcome, you are now connected to log-streaming service.
2018-10-20T13:17:54.660 [Information] Executing 'HttpTriggerCSharpFromVS' (Reason='This function was
programmatically called via the host APIs.', Id=95e43756-962e-4210-bf16-0303be4117f9)
2018-10-20T13:17:54.847 [Information] GetConnectionStringOrSettingHello! i'm a value from App Setting
2018-10-20T13:17:54.847 [Information] ValueFromConfigurationIndexHello! i'm a value from App Setting
2018-10-20T13:17:54.847 [Information] ConnectionStrings:sqldb_connectionconnection_string_here
2018-10-20T13:17:54.858 [Information] Executed 'HttpTriggerCSharpFromVS' (Succeeded, Id=95e43756-962e-4210-bf16-
0303be4117f9)
```

Application setting – binding expressions

In the previous section, we learned how to access the configuration settings from the code. Sometimes, you might want to configure some of the declarative items too. You could achieve that using binding expression. You will understand what I mean in a moment when we look at the code:

1. Open the Visual Studio and make changes to the `Run` method to add a new parameter for configuring the `QueueTrigger`:

```
public static IActionResult Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post
    ILogger logger,
    [QueueTrigger("hardcodedqueuename")] string queue)
{
```

2. The `hardcodedqueuename` parameter is the name of the queue to which messages will be created. It's obvious that hardcoding the name of the queue is not a good practice. In order to make it configurable, you need to make use of application setting binding expression:

```
public static IActionResult Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]HttpRequest req,
    ILogger logger,
    [QueueTrigger("%queuename%")] string queue)
```

3. The application setting key must be enclosed in `%...%` and a key with the name `queuename` should be created in the **Application Settings**.

Creating and generating open API specifications using Swagger

For a backend web API developer, one of their responsibilities is to provide a proper documentation to the frontend application developers so that they can consume the APIs without any problems. In order to consume any API, the following are the two minimum things that one needs to understand:

- The input parameters and their data types
- The output parameters and their data types

So, it's the responsibility of the backend developers to provide proper documentation for the APIs and it's not easy to provide proper documentation as there are many tools and standards/specifications that are available for providing proper documentation for the REST APIs. One such standard is known as the open API specification (it's popularly known as Swagger).

Azure Functions provide us with the required tooling support for generating the open API definitions for our HTTP triggers. In this recipe, we will learn how to generate them.

Getting ready

Create a function and create one or more HTTP triggers. In order to make it simple, I have created a function app and one HTTP trigger, which accepts `Get` methods only.

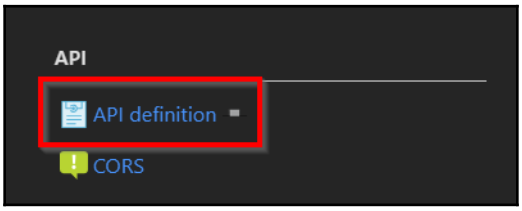
Note that at the time of writing, the API definition feature is supported only by the Azure Function runtime V1.0 and it's still in preview. It doesn't work with V2.o yet.

Ensure that the Azure function app is configured to point to runtime version 1, as shown in the **Application Settings**:

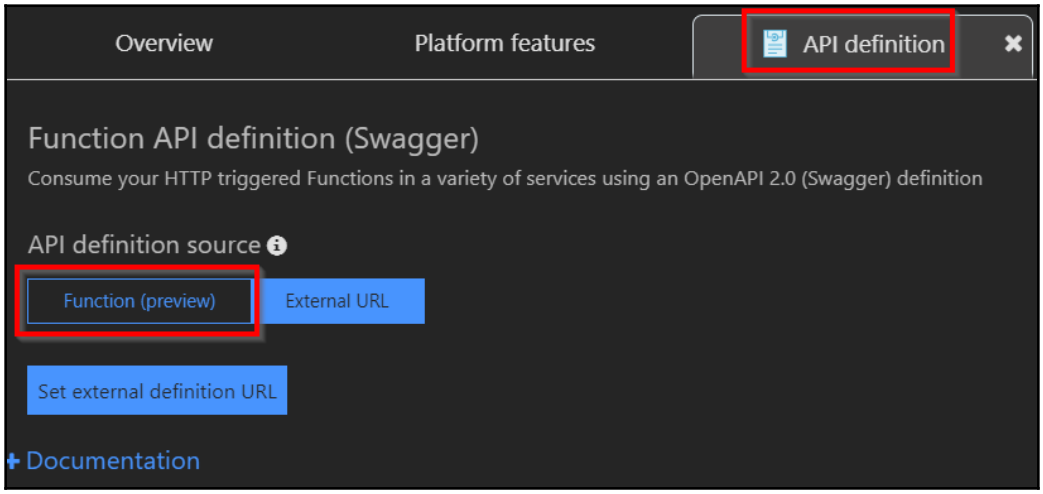
How to do it...

Perform the following steps:

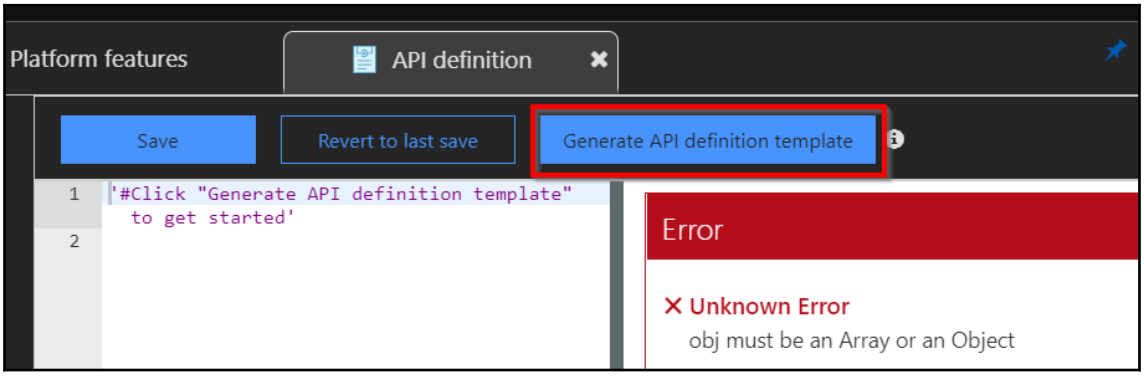
1. Navigate to the **Platform features** and click on the **API definition** tab:



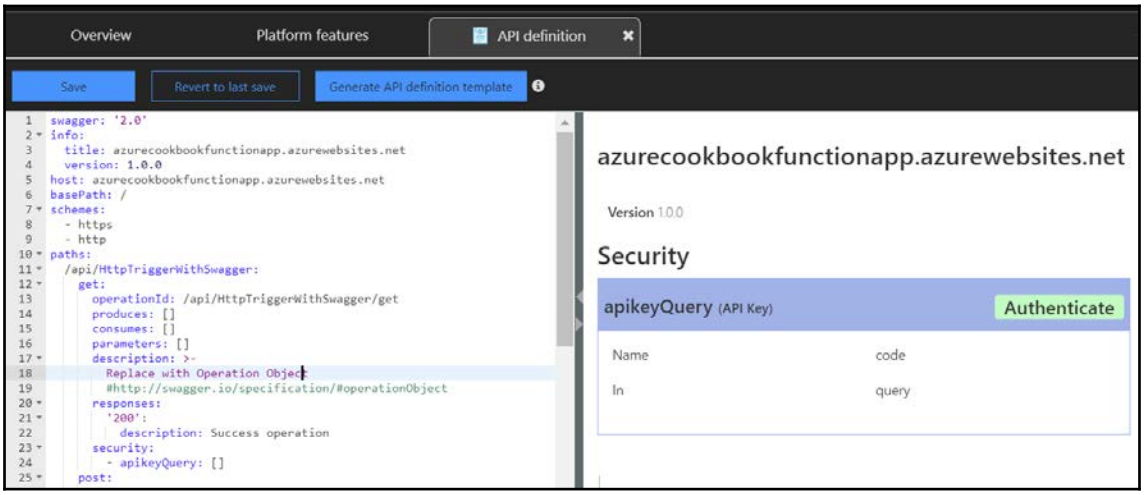
2. In the **API definition** tab, click on the **Function (preview)** option to enable the source of the API definition:



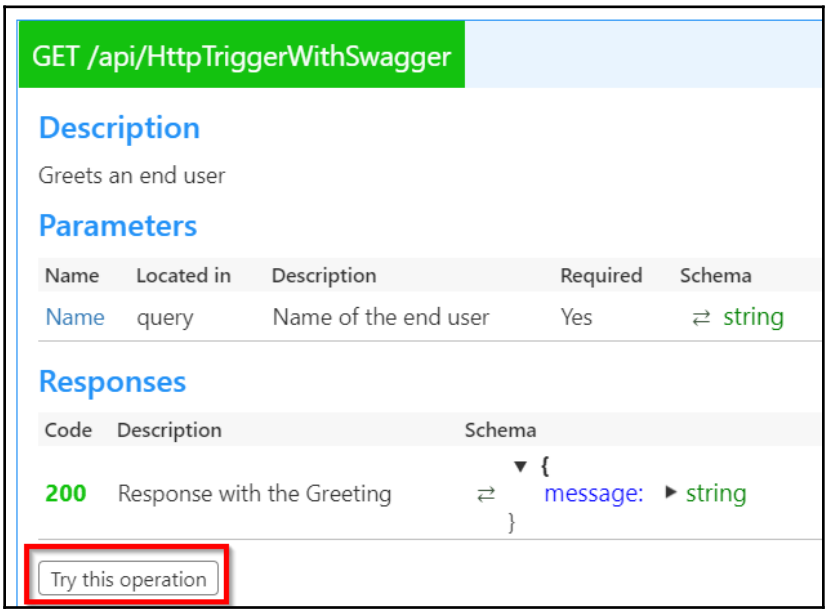
3. As soon as you click on the **Function (preview)** button, the feature will be enabled. However, you might see an error in the new tab that is opened. Don't worry, as the feature is still in preview, Microsoft might fix it before it goes **generally available** (**GA**). Then, click on **Generate API definition template**:



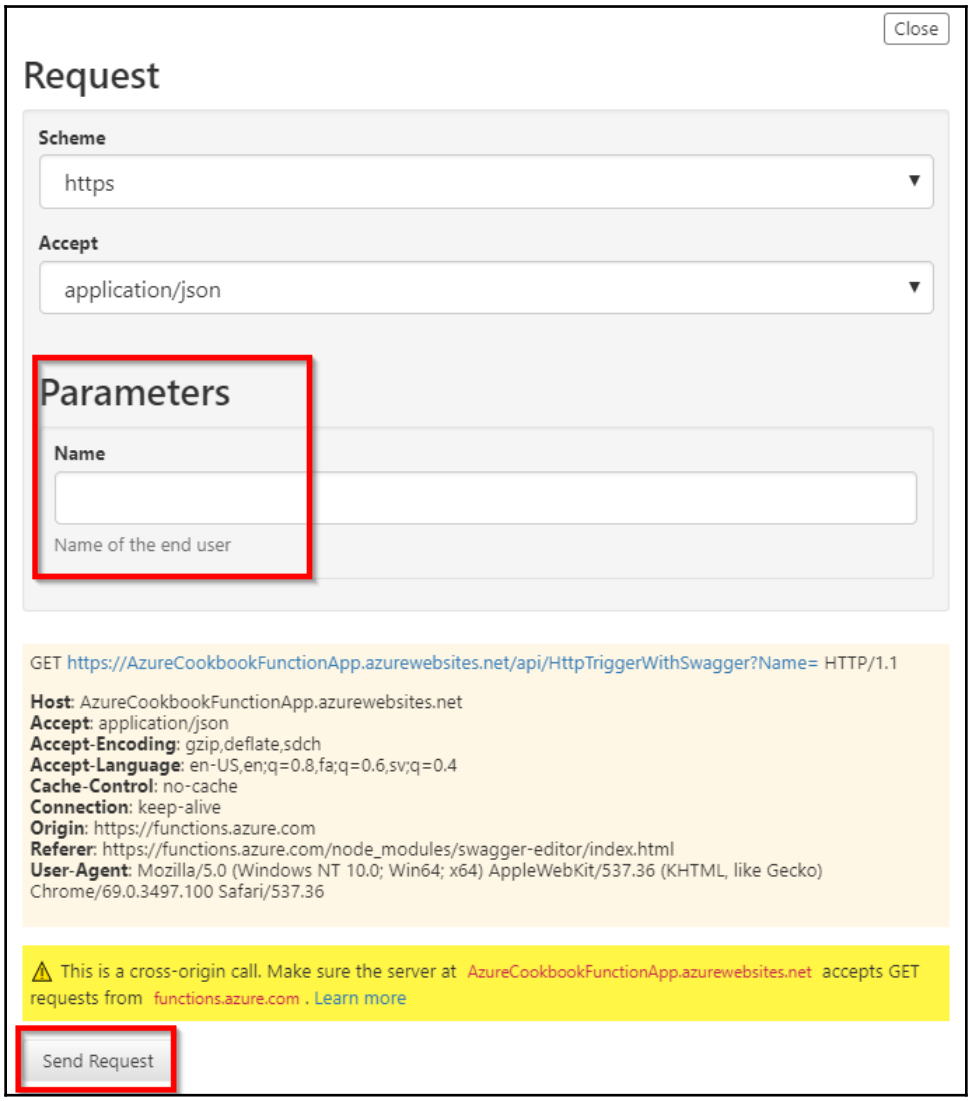
4. This will just create a template of the open API definition. It's the cloud developer's responsibility to fill in the template, based on the APIs that they have developed. It should look something like this. We will change the template in a moment:



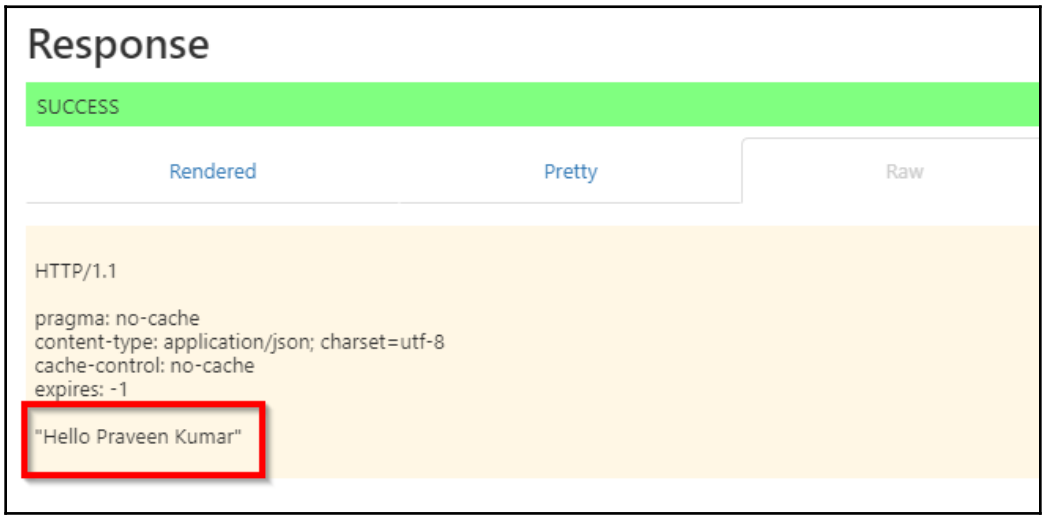
5. In the preceding screenshot, the code tab contains all the default templates required for generating the Swagger definition based on the open API specification. The right section shows how the Swagger UI looks. The Swagger UI is something that will be shared with the other client application development teams who consume the backend APIs.
6. Let's replace the default template by adding the required parameters of the API operations, and click on the **Save** button to save the changes. To make it simple, I just made a few changes that describe the API and its operations. It should be straightforward to understand.
7. The Swagger UI will look as follows with proper messages along with request and response formats:



8. It also allows us to run some tests. Click on the **Try this operation** button that is shown in the preceding screenshot. It opens up a window where you can provide input:



9. I have provided `Praveen Kumar` as the value for the name, clicked on the **Send Request** button, and got the following output:

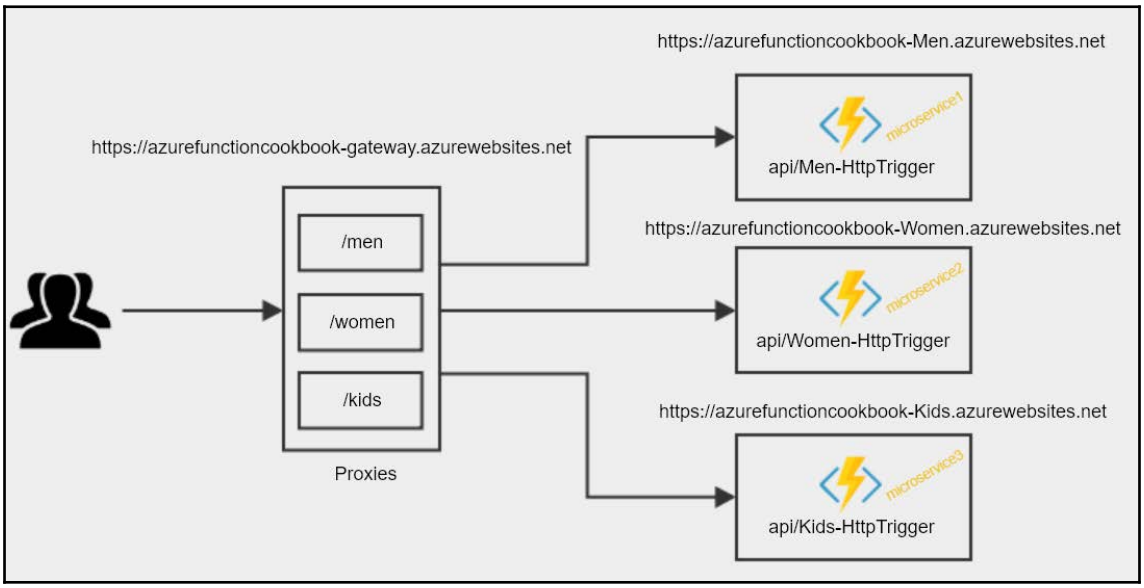


Breaking down large APIs into small subsets of APIs using proxies

In recent times, one of the buzzwords in the industry is microservices, where you develop your web components as microservices that can be managed (scaling, deployment, and so on) individually without impacting the other related components. Though the subject of microservices is itself a huge one, we will try to achieve building a very few microservices that could be managed individually as independent function apps. But, we will expose them to the external world as a single API with different operations with the help of Azure Function Proxies.

Getting ready

In this recipe, we will be implementing the following architecture:



Let's assume that we are working for an e-commerce portal where we just have three modules (men, women, and kids) and our goal is to build the backend APIs in a microservice architecture where each microservice is independent of the others.

In this recipe, we will achieve this by creating the following function apps:

- A gateway component (function app) that is responsible for controlling the traffic to the right microservice based on the route (**/men**, **/women**, and **/kids**). In this function app, we would be creating Azure Function Proxies that will redirect the traffic using route configurations.
- Three new function apps where each of them is treated as a separate microservice.

How to do it...

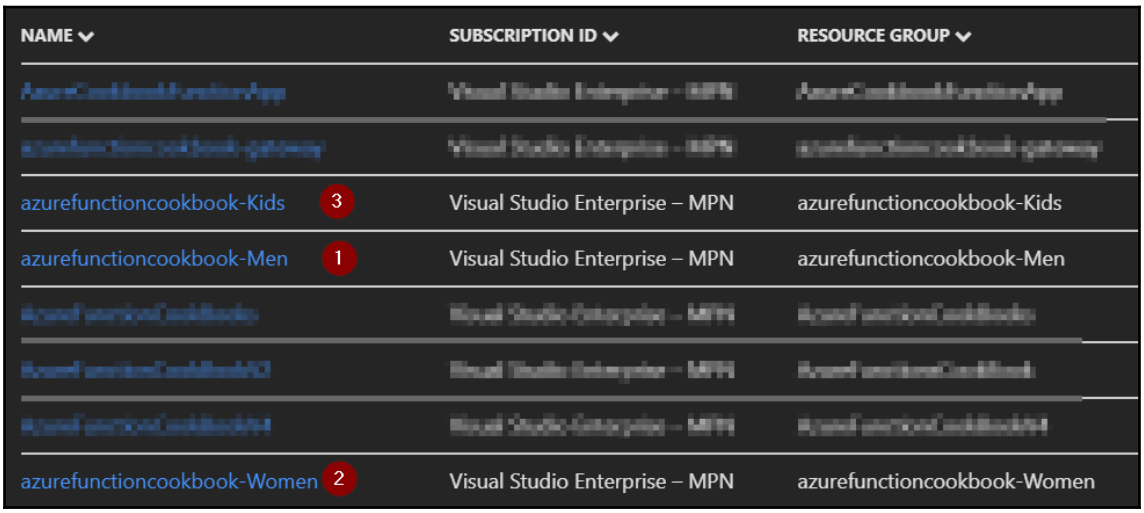
In this recipe, we will be performing the following steps:

1. Create all three microservices with one HTTP trigger in each of them
2. Create, proxy and configure the respective microservice
3. Test the proxy URL

Creating microservices

Perform the following steps:

1. Create three function apps for each of the microservices that we have planned:



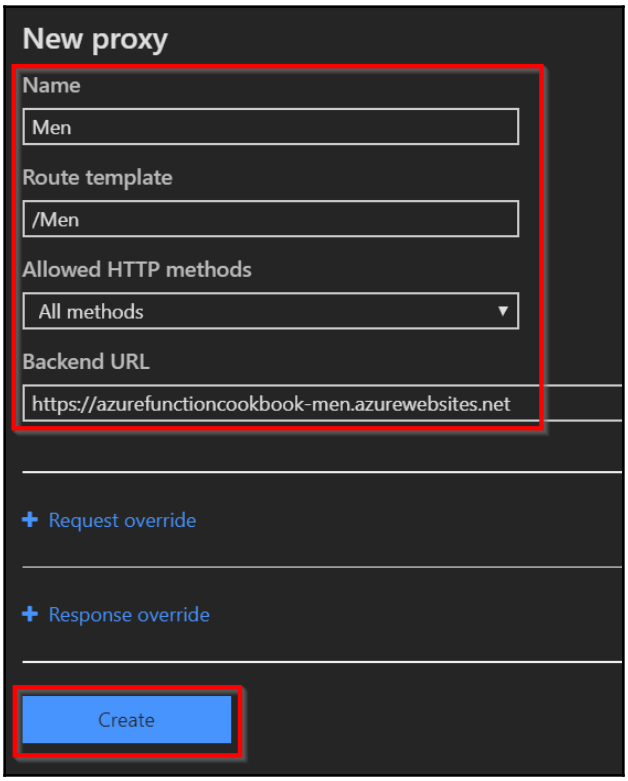
2. Create the following anonymous HTTP triggers in each of the function apps that display a message something shown as follows:

Http Trigger name	**Output message**
`Men-HttpTrigger`	`Hello <<Name>> - Welcome to the Men Microservice`
`Women-HttpTrigger`	`Hello <<Name>> - Welcome to the Women Microservice`
`Kids-HttpTrigger`	`Hello <<Name>> - Welcome to the Kids Microservice`

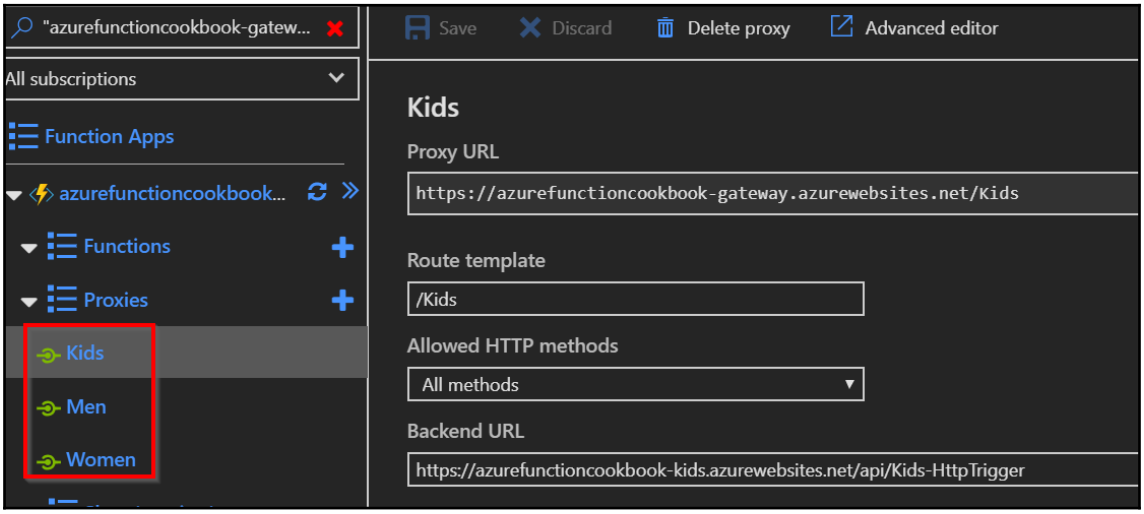
Creating the gateway proxies

Perform the following steps:

1. Navigate to the gateway function app and create a new proxy:



2. You will be taken to the details blade:



3. Create the proxies for women and kids. Here are the details of all three proxies. Note that the backend URLs (of the function apps) might be different in your case:

Proxy name	**Route template**	**Backend URL (the URLs of the HTTP triggers created in the previous step)**
Men	/Men	`https://azurefunctioncookbook-men.azurewebsites.net/api/Men-HttpTrigger`
Women	/Women	`https://azurefunctioncookbook-women.azurewebsites.netapi/Women-HttpTrigger`
Kids	/Kids	`https://azurefunctioncookbook-kids.azurewebsites.netapi/Kids-HttpTrigger`

4. Once you create the three proxies, the list will look something like this:

5. In the preceding screenshot, you can view three different domains. However, if you would like to share these with the client applications, you don't need to share these URLs. All you need to do is share the URL of the proxies that you can view in the proxy tab. Here are the proxy URLs of the three proxies we have created:
    - `https://azurefunctioncookbook-gateway.azurewebsites.net/Men`
    - `https://azurefunctioncookbook-gateway.azurewebsites.net/Women`
    - `https://azurefunctioncookbook-gateway.azurewebsites.net/Kids`

Testing the proxy URLs

As you already know, our HTTP triggers accept a required `name` parameter, we need to pass the `name` query string to the proxy URL. Let's access the following URLs in the browser:

- **Men**:

- **Women**:

- **Kids**:

Observe the URLs of the three screenshots. You will notice that they look like they are being served from one single application with different routes. However, they are three different microservices that could be managed individually.

There's more...

All the microservices that we have created in this recipe are anonymous, which means they are publicly accessible to everyone. In order to make them secure, you need to follow either of the approaches recommended in `Chapter 9`, *Implementing Best Practices for Azure Functions*.

The Azure Function proxies also allow you to intercept the original request and, if required, you can add new parameters and pass them to the backend API. Similarly, you can add additional parameters and pass the response back to the client application. You can learn more about Azure Function proxies in the official documentation at `https://docs.microsoft.com/en-us/azure/azure-functions/functions-proxies`.

See also

- The *Controlling access to Azure Functions using function keys* recipe of `Chapter 9`, *Implementing Best Practices for Azure Functions*
- The *Securing Azure Functions using Azure AD* recipe of `Chapter 9`, *Implementing Best Practices for Azure Functions*
- The *Configuring throttling of the Azure Functions using API Management* recipe under `Chapter 9`, *Implementing Best Practices for Azure Functions*

Moving configuration items from one environment to another using resources

Every application that you develop will have many configuration items (such as application settings as connection strings) that will be stored in `Web.Config` files for all your .NET-based web applications.

In the traditional on-premises world, the `Web.Config` file would be located in the server and the file would be accessible to all people who have access to the server. Although it is possible to encrypt all the configuration items of `Web.Config`, it had its limitations, and they're not easy to decrypt every time you want to view or update them.

In the Azure PaaS world, with Azure App Services, you still can have the `Web.Config` files and they work as they used to in the traditional on-premises world. However, Azure App Service provides us with additional feature in terms of application settings, where you can configure these settings (either manually or via ARM templates), and these settings are stored in an encrypted format. But you can view them as normal text in the portal if you have access.

Depending on the application type, the number of application settings might grow to a large size, and if you want to create new environments, then creating these application settings would take quite a bit of time. In this recipe, we will learn a tip of exporting and importing these application settings from a lower environment (say, Dev) to a higher environment (say, Prod).

Getting ready

Perform the following steps:

1. Create a function app (say, `MyApp-Dev`) if not created already
2. Create some application settings:

APP SETTING NAME	VALUE
AppSetting0	*Hidden value. Click to edit.*
AppSetting1	*Hidden value. Click to edit.*
AppSetting2	*Hidden value. Click to edit.*
AppSetting3	*Hidden value. Click to edit.*
AppSetting4	*Hidden value. Click to edit.*
AppSetting5	*Hidden value. Click to edit.*
AppSetting6	*Hidden value. Click to edit.*
AppSetting7	*Hidden value. Click to edit.*
AppSetting8	*Hidden value. Click to edit.*
AppSetting9	*Hidden value. Click to edit.*

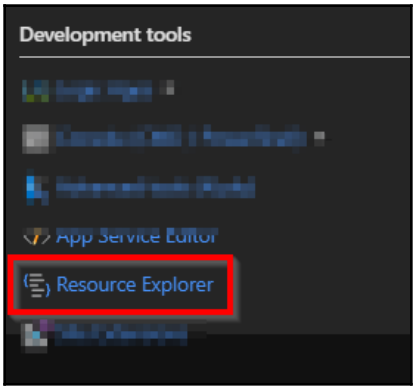
3. Create another function app (say, `MyApp-Prod`)

In this recipe, we will learn how easy it is to copy the application settings from one function to another. This technique will be handy when there are many app settings.

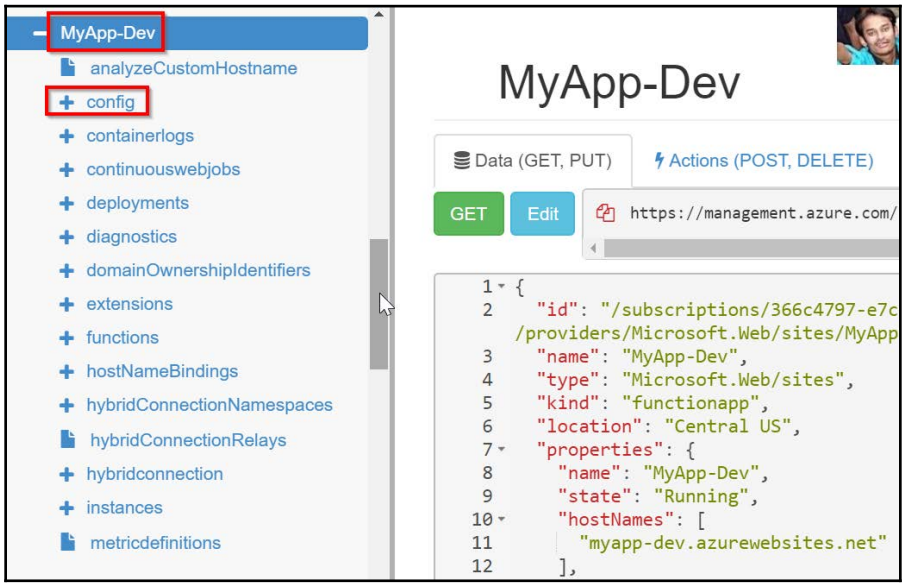
How to do it...

Perform the following steps:

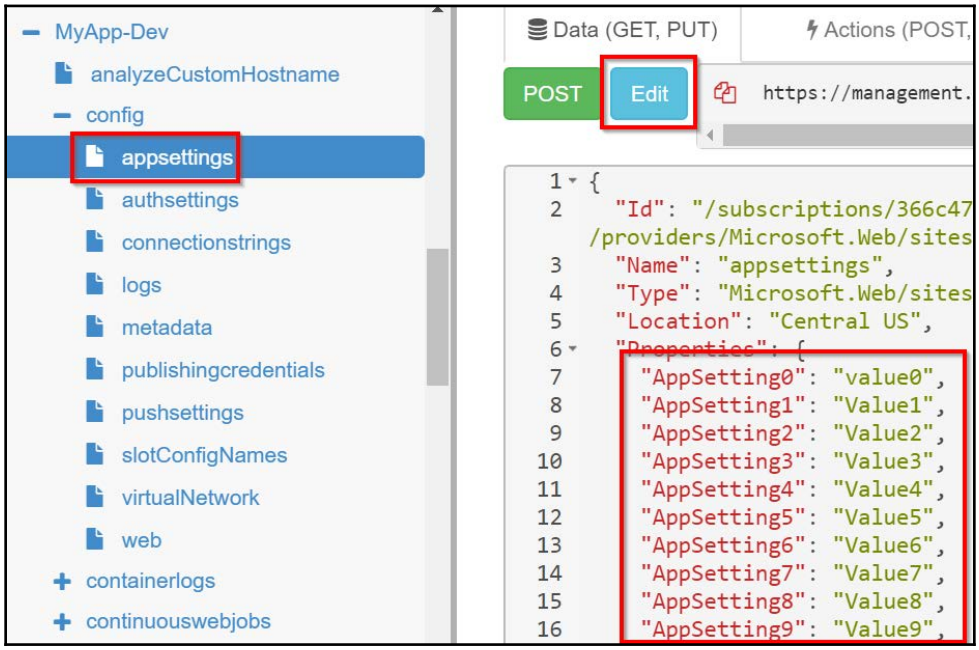
1. Navigate to the **Platform features** tab of the `MyApp-Dev` Function app and click on the **Resource Explorer**:



2. The **Resource Explorer** will be opened, and from there you can traverse all the internal elements of a given service:

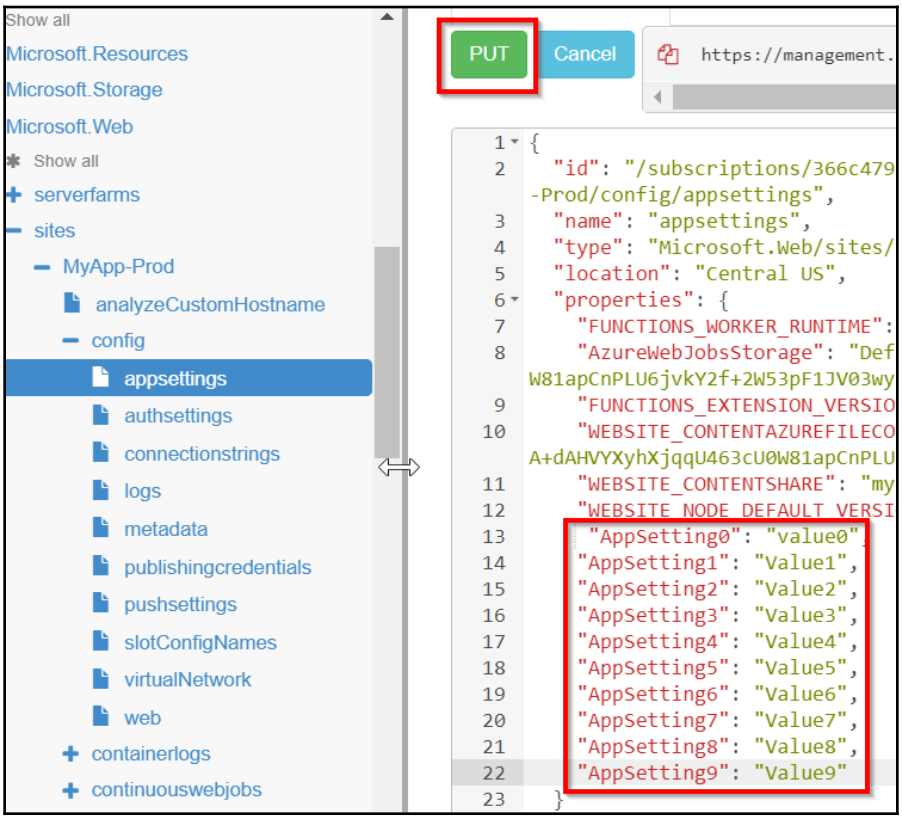


3. Click on the **config** element, as shown in the preceding screenshot, which opens all the items related to configurations:



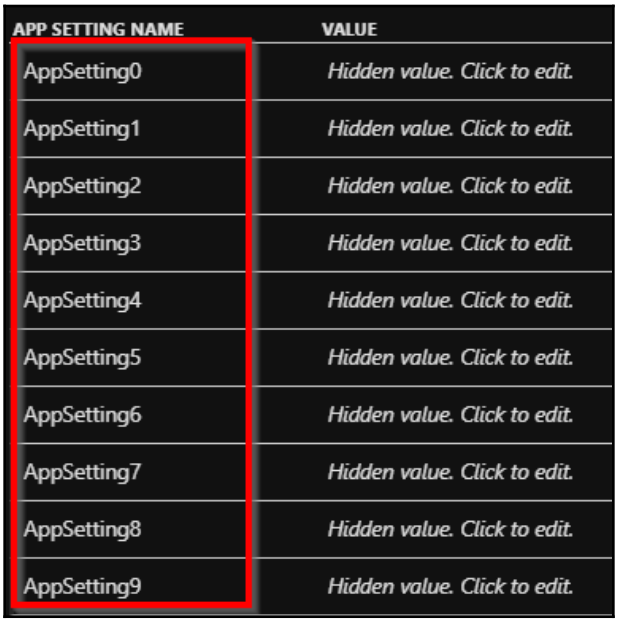
4. **Resource Explorer** will display all the application settings in the right-hand window. Now, you can either edit them by clicking on the **Edit** button, which is highlighted in the preceding screenshot, or you can copy all the application settings from `AppSetting0` to `AppSetting 9`.

5. Navigate to the `MyApp-Prod` function app (which won't have the application settings highlighted in the previous screenshot), click on the **Resource Explorer**, and then click on the **config** | **appsettings** elements to open the existing application settings. It should look something like this:



6. Click on the **Edit** button and paste the content that you copied earlier. Once you've reviewed the settings, click on **PUT**, which is shown in the preceding screenshot.

7. Navigate to the application setting blade of the `MyApp-Prod` function app:



You should see all the application settings that we have created in the **Resource Explorer** in a single shot.

11
Implementing and Deploying Continuous Integration Using Azure DevOps

In this chapter, you will learn the following:

- Continuous integration – creating a build definition
- Continuous integration – queuing a build and triggering it manually
- Configuring and triggering an automated build
- Continuous integration – executing unit test cases in the pipeline
- Creating a release definition
- Triggering the release automatically

Introduction

As a software professional, you might have already been aware of different software development methodologies that people practice. Irrespective of the methodology being followed, there will be multiple environments, such as dev, staging, and production, where the application life cycle needs to be followed with these critical stages related to development:

1. Develop based on the requirements
2. Build the application and fix any errors

3. Deploy/release the package to an environment (dev/stage/prod)
4. Test against the requirements
5. Promote the release to the next environment (from dev to stage and stage to prod)

Note that for the sake of simplicity, the initial stages, such as requirement gathering, planning, design, and architecture, are excluded just to emphasize the stages that are relevant to this chapter.

For each change that you make to the software, we need to build and deploy the application to multiple environments, and it might be the case that different teams are responsible for releasing the builds to different environments. As different environments and teams are involved, considering the amount of time that is spent in running the builds, deploying them in different environments would be more dependent on the processes that different teams follow.

In order to streamline and automate a few of the steps mentioned earlier, in this chapter, we will discuss some of the popular techniques that the industry follows in order to deliver software quickly, with minimal infrastructure.

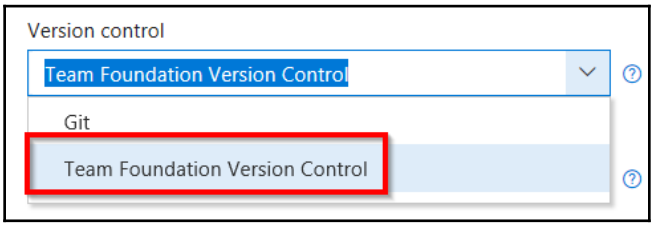
In previous chapters, most of the recipes provided us with a solution for an individual business problem. However, in this chapter, the entire chapter as a single entity will try to provide you with a solution for the Continuous Integration and Continuous Delivery of your business-critical application.

The Azure DevOps team continuously adds new features to Azure DevOps at `https://dev.azure.com` (formerly known as VSTS at `https://www.visualstudio.com`) and updates the user interface as well. Don't be surprised if screenshots that are provided in this chapter don't match those of your screens at `https://dev.azure.com` while you are reading this.

Prerequisites

Create the following if you have don't have them already:

1. Create an Azure DevOps organization of your choice at `https://dev.azure.com` and create a new project in that account. While creating the project, you can either choose **Git** or **Team Foundation Version Control** as your version control repository. I have used **Team Foundation Version Control** for my project:



2. Configure the Visual Studio project that you developed in `Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools for Azure Functions*, to Azure DevOps. You can go through the `https://www.visualstudio.com/en-us/docs/setup-admin/team-services/set-up-vs` link to follow the step-by-step process of creating a new account and project using Azure DevOps.

> *I will be making some small changes to the response messages embedded within the code to shown different outputs. Ensure that you modify the unit tests accordingly. Otherwise, the build will fail.*

Continuous integration – creating a build definition

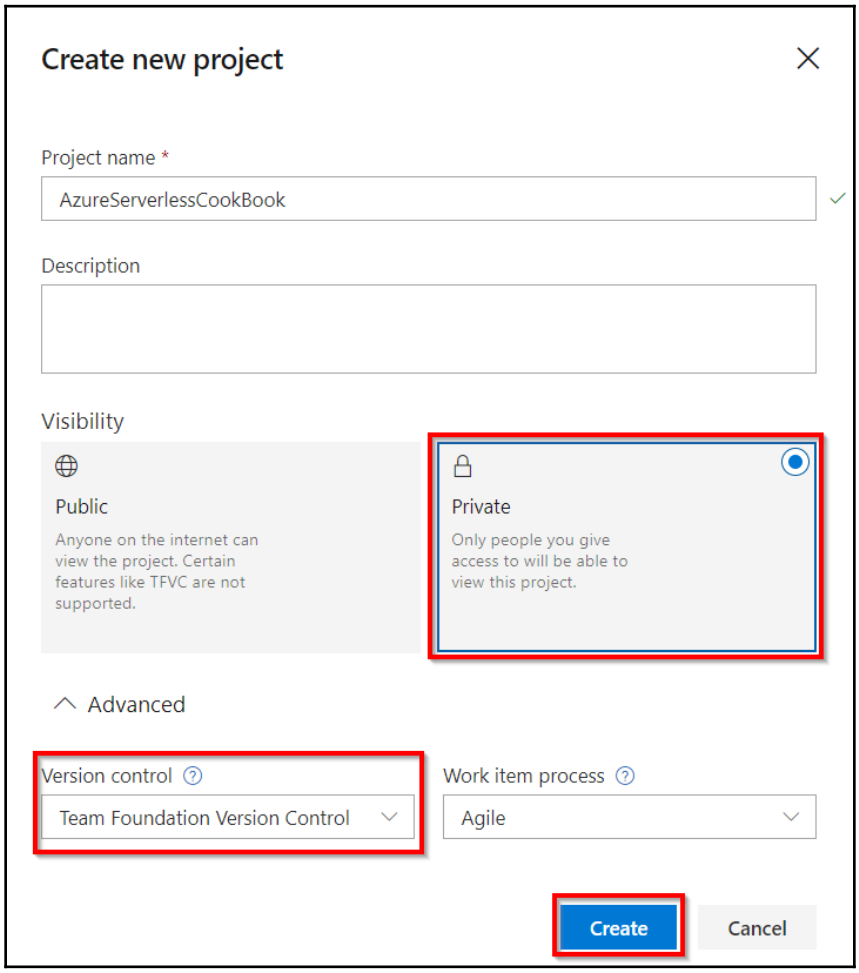
A build definition is a set of tasks that are required to configure an automated build of your software. In this recipe, we will perform the following:

1. Create the build definition template
2. Provide all the inputs required for each of the steps to create the build definition

Getting ready

Perform the following prerequisites:

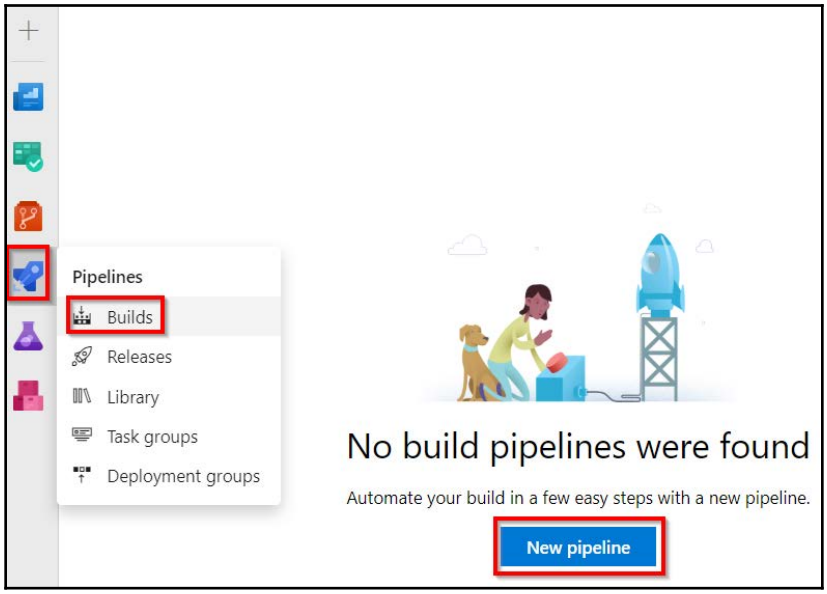
1. Create an Azure DevOps account.
2. Create a project by choosing **Team Foundation Version Control**, as shown in the following screenshot:



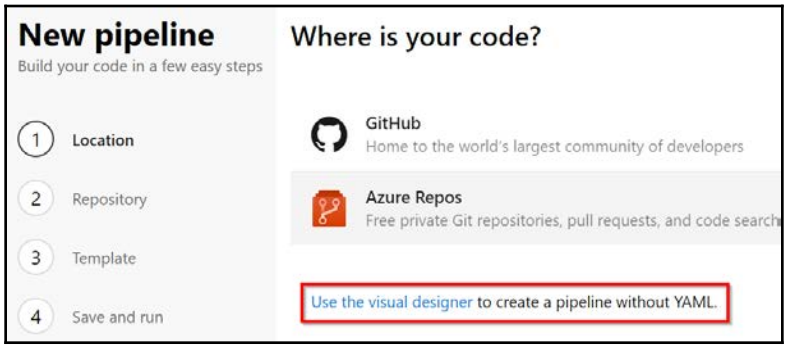
How to do it...

Perform the following steps:

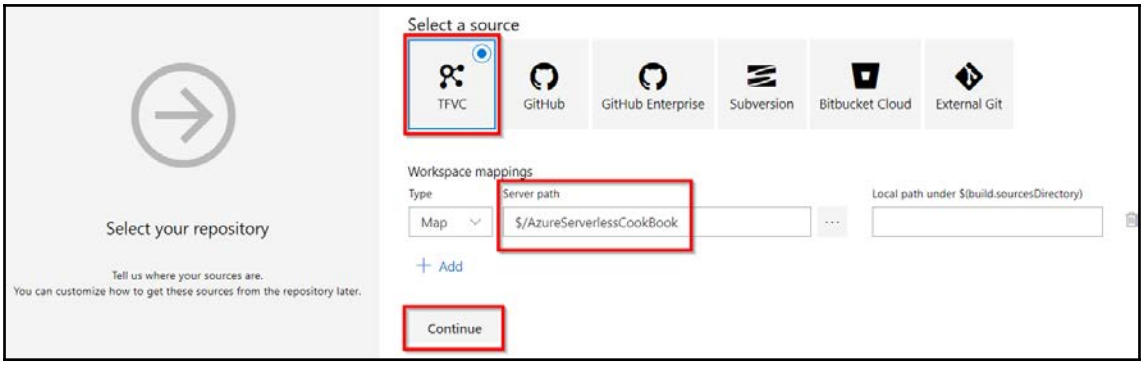
1. Navigate to the **Pipelines** tab in your Azure DevOps account, click on **Builds,** and choose **New pipeline** to start the process of creating a new build definition, as shown in the following screenshot:



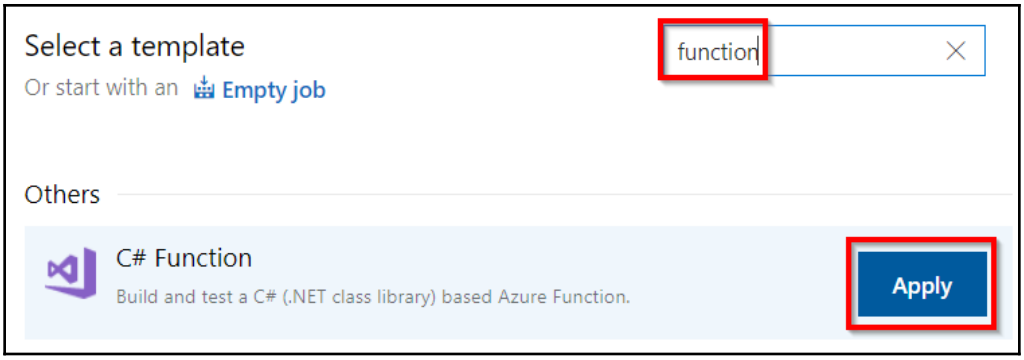
2. In the next step, click on the **Use a visual designer** link, as shown in the following screenshot:



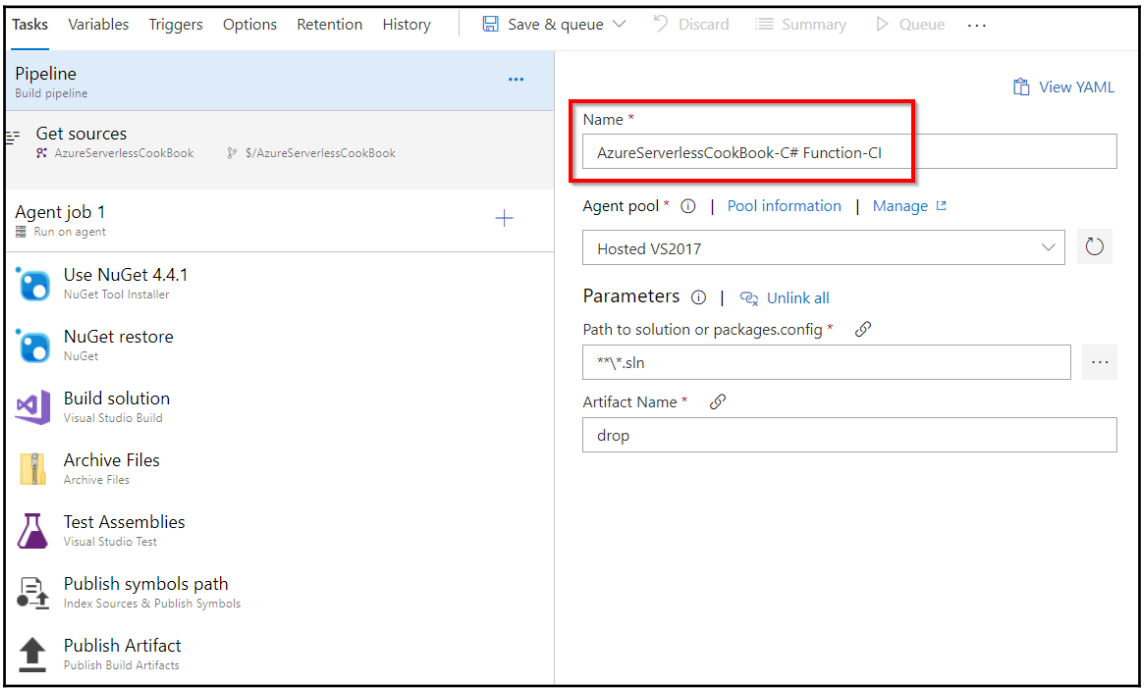
3. You will be taken to the **Select your repository** screen where you can choose your repository. For this example, I have mine sourced in **TFVC**. As shown next, select **TFVC** and click on **Continue**. Make sure that you have chosen your project, which in my case is **AzureServerlessCookBook**:



4. You will be taken to the **Select a template** step, where you can choose the template required for your application. For this recipe, we will choose **C# Function**, as shown in the following screenshot, by clicking on the **Apply** button:

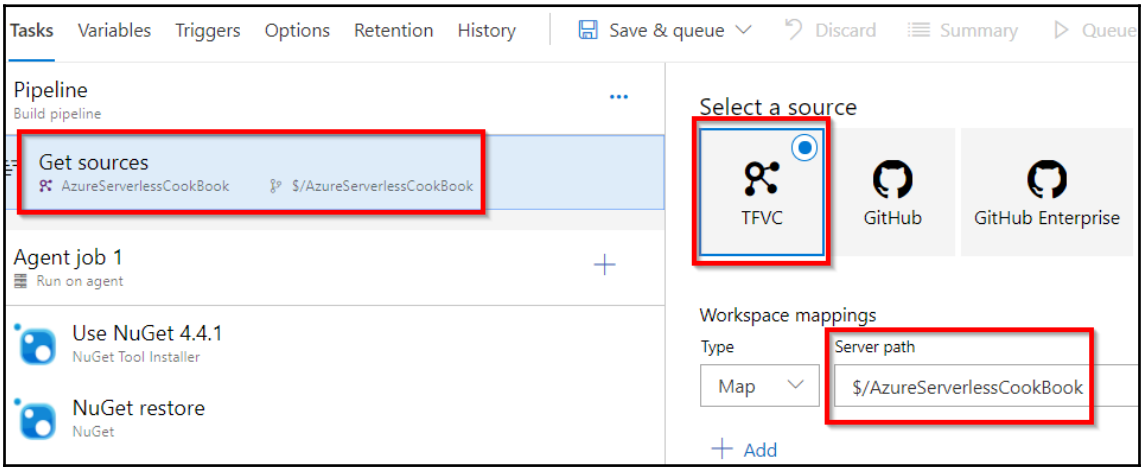


5. The *create build* step is a set of steps used to define the build template, where each step has certain attributes that we need to review and provide inputs for each of those fields based on our requirements. Let's start by providing a meaningful name in the *pipeline* step and also ensure that you choose **Hosted VS2017** in the **Agent Pool** drop-down, as shown in the following screenshot:



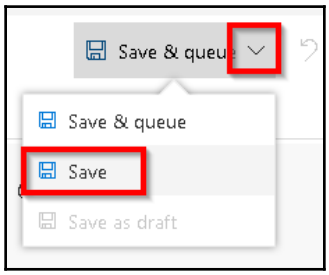
An agent is software hosted on the cloud that is capable of running a build. As our project is based on VS2017, we have chosen **Hosted VS2017**.

6. In the **Get sources** step, ensure that the following are selected:
    1. Select the version control system that you would like to have.
    2. Choose the repository that you want to build. In my example, I have chosen **AzureServerlessCookBook**:



7. Leave the default options for all the following steps:
    - **Use NuGet and NuGet restore**: This step is required for downloading and installing all the required packages for the application.
    - **Build solution**: This step uses MS build and has all the predefined commands to create the build.
    - **Test assemblies**: This would be useful if we had any automated tests. We will make some changes to this step in the *Continuous integration – executing unit test cases in the pipeline* recipe later in this chapter.
    - **Archive files**: This step lets us archive the folders in the required format.
    - **Publish symbols path**: These symbols are useful if you want to debug your app hosted in the Agent VM.
    - **Publish artifact**: The step has configuration related to the artifacts and the path for storing the artifact (build package).

8. Once you review all the values in all the fields, click on **Save**, as shown in the following screenshot, and click on **Save** again in the **Save build definition** popup:



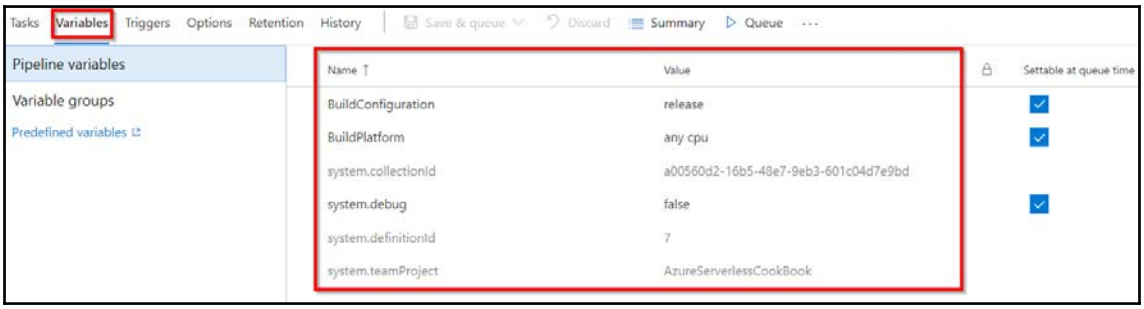
How it works...

A build definition is just a blueprint of the tasks that are required for building a software application. In this recipe, we have used a default template to create the build definition. We can choose a blank template and create the definition by choosing the tasks available in Azure DevOps as well.

When you run the build definition (either manually or automatically, which will be discussed in the subsequent recipes), each of the tasks will be executed in the order in which you have configured them. You can also rearrange the steps by dragging and dropping them in the pipeline section.

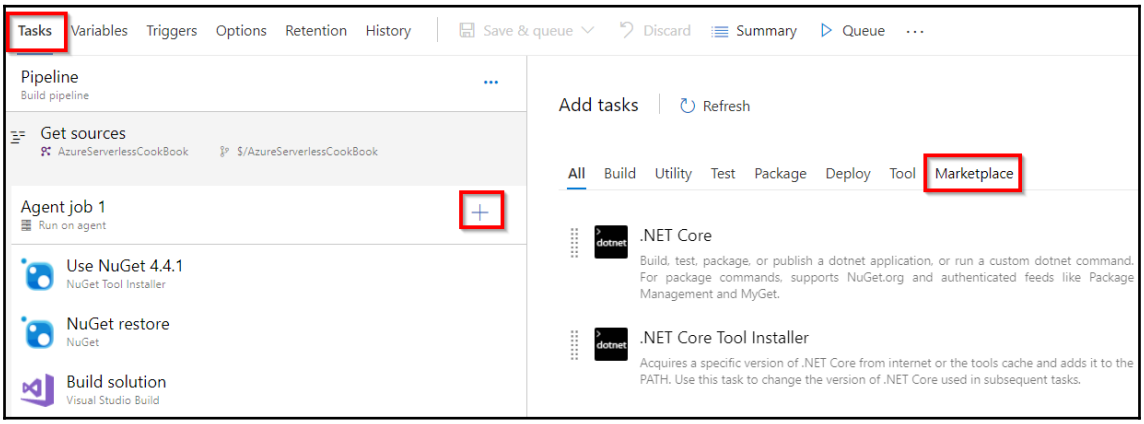
The build process starts with getting the source code from the chosen repository and downloading the required NuGet packages, and then starts the process of building the package. Once that process is complete, it creates a package and stores it in a folder configured for the `build.artifactstagingdirectory` directory (refer to the **Path to publish** field of the **Publish artifact** task).

You can learn about all different type of variables in the **Variables** tab shown here:



There's more...

- Azure DevOps provides many tasks. You can choose a new task for the template by clicking on the **Add Task (+)** button, as shown in the following screenshot:



- If you don't find a task that suits your requirements, you can search for a suitable one in the marketplace by clicking on the **Marketplace** button shown in the preceding screenshot.
- C# function has the correct set of tasks required to set up the build definition for Azure Functions as well.

Continuous integration – queuing a build and triggering it manually

In the previous recipe, you learned how to create and configure the build definition. In this recipe, you will learn how to trigger the build manually and understand the process of building the application.

Getting ready

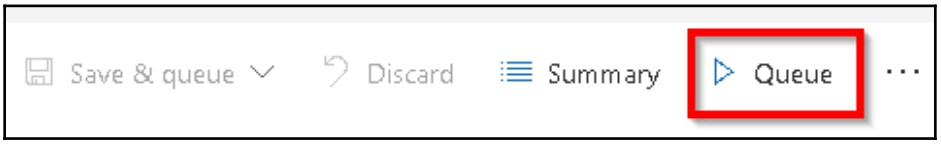
Before we begin, make sure of the following:

- You have configured the build definition as mentioned in the previous recipe
- All your source code is checked in to the Azure DevOps team project

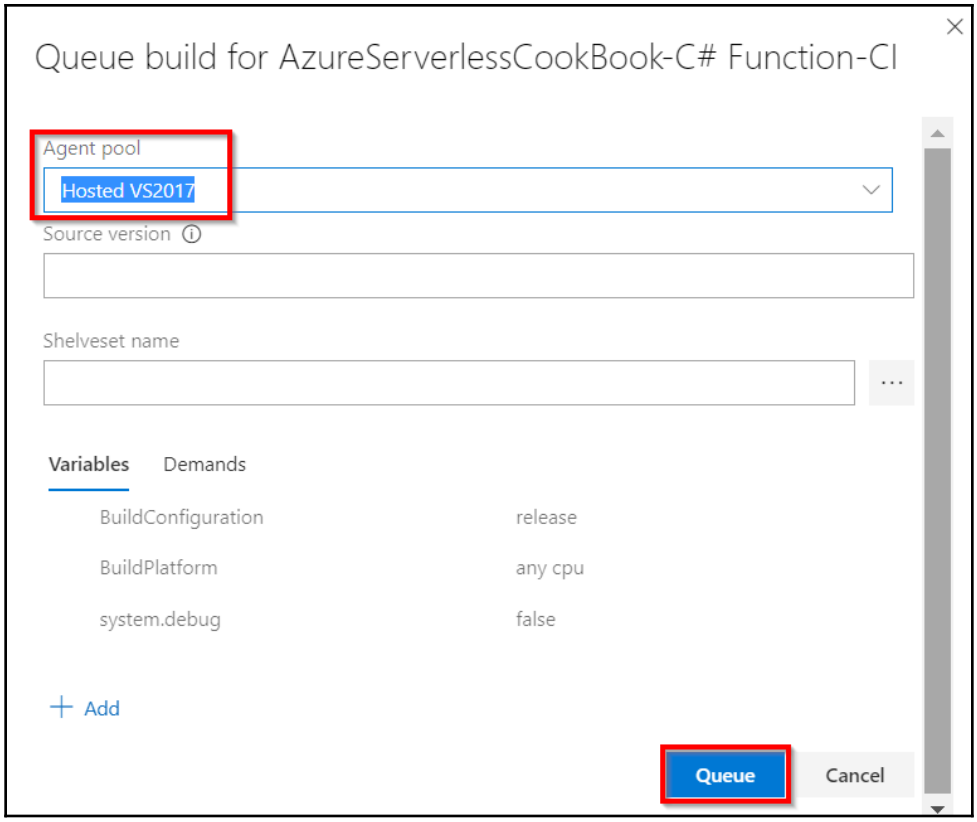
How to do it...

Perform the following steps:

1. Navigate to the build definition named `AzureServerlessCookBook-C#Function-CI` and click on the **Queue** button available on the right-hand side, as shown in the following screenshot:

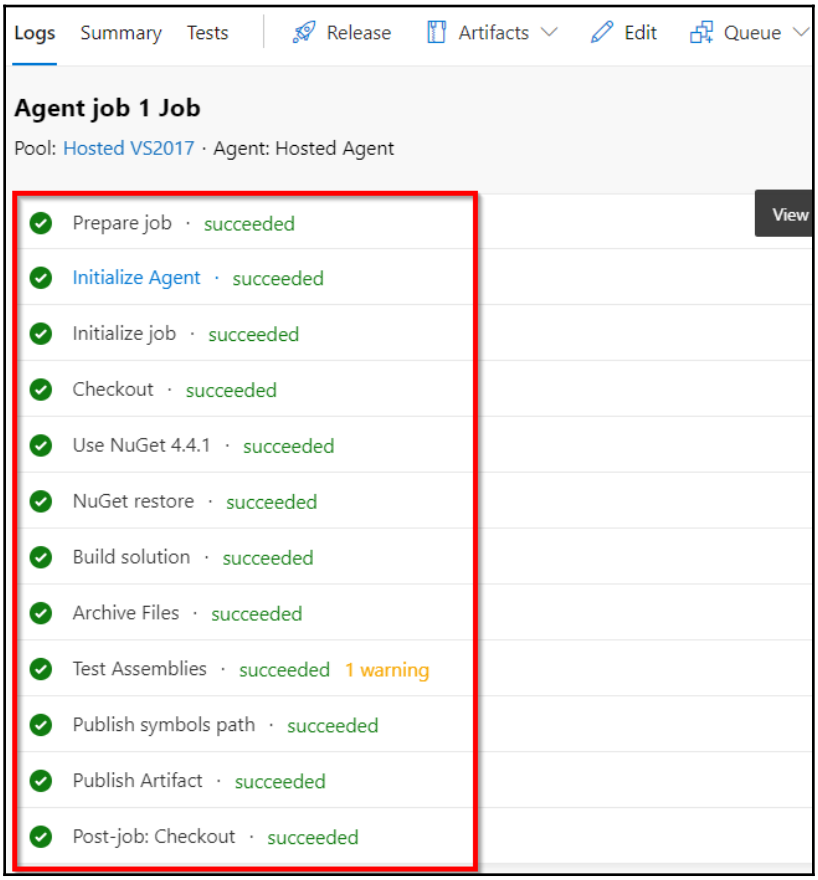


2. In the **Queue build for AzureServerlessCookBook-C# Function-CI** popup, make sure that the **Hosted VS2017** option is chosen in the **Agent pool** drop-down if you are using Visual Studio 2017 and click on the **Queue** button, as shown in the following screenshot:

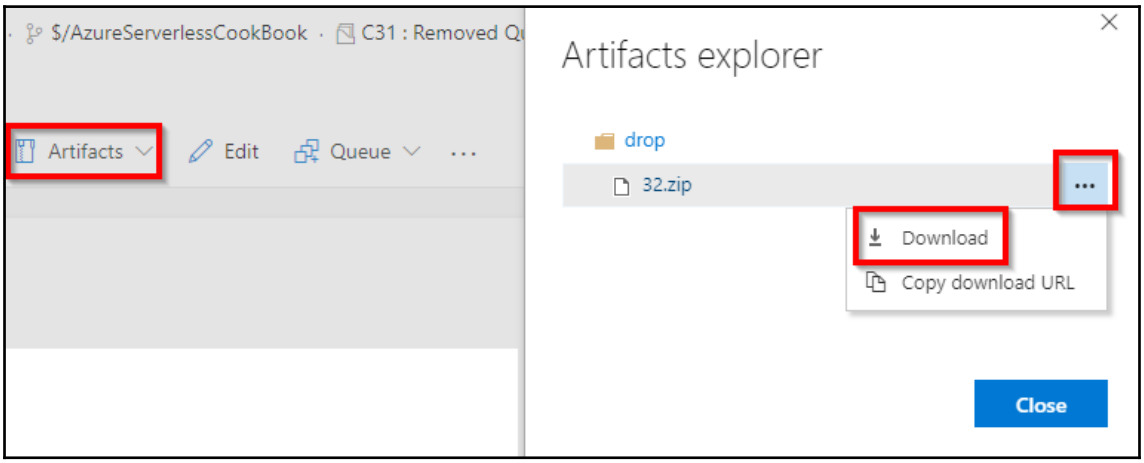


3. In just a few moments, the build will be queued and the message will be displayed, as shown in the following screenshot:

4. Clicking on the build ID (in our case, **20181025.1**) will start the process, and it waits for a few seconds for an available agent to start it.
5. After a few moments, the build process will start, and in just a few minutes, the build will be completed and you can review the steps of the build in the logs, as shown here. Ignore the warning that is shown for **Test Assemblies** for now. We will fix this in the recipe *Continuous integration – executing unit test cases in the pipeline* later in this chapter:



6. If you would like to view the output of the build, click on the **Artifacts** button highlighted in the following screenshot. You can download the files by clicking on the **Download** button as shown here:



Configuring and triggering an automated build

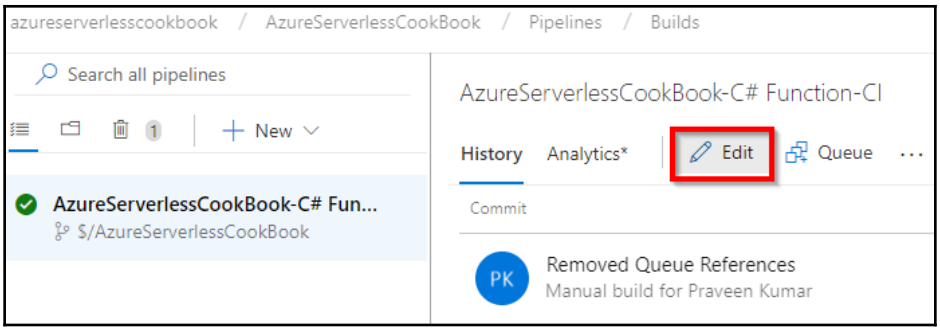
For most applications, it might not make sense to perform manual builds in Azure DevOps. It would make sense if we can configure **Continuous Integration (CI)** by automating the process of triggering the build for each check-in/commit done by the developers.

In this recipe, you will learn how to configure continuous integration in Azure DevOps for your team project and also trigger the automated build by making a change to the code of the HTTP trigger Azure Function that we created in `Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools for Azure Functions*.

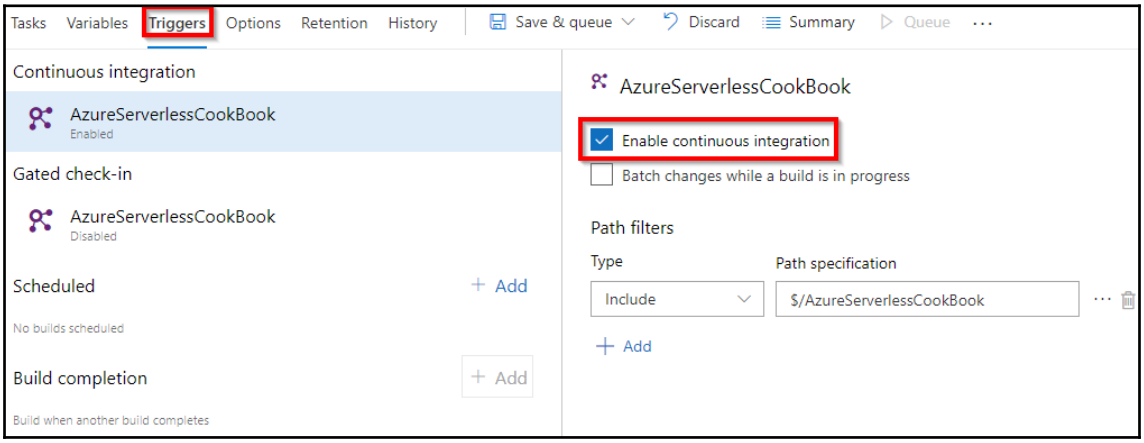
How to do it...

Perform the following steps:

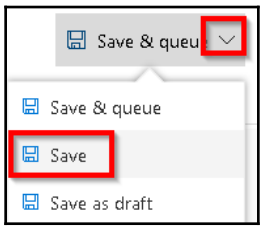
1. Navigate to the build definition `AzureServerlessCookBook-C# Function-CI` by clicking on the **Edit** button as shown in the following screenshot:



2. Once you are in the build definition, click on the **Triggers** menu, shown as follows:



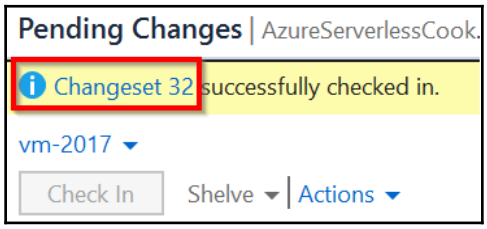
3. Now, click on the **Enable continuous integration** checkbox to enable the automated build trigger.
4. Save the changes by clicking on the arrow mark available beside the **Save & queue** button and click on the **Save** button available in the drop-down menu, which is shown in the following screenshot:



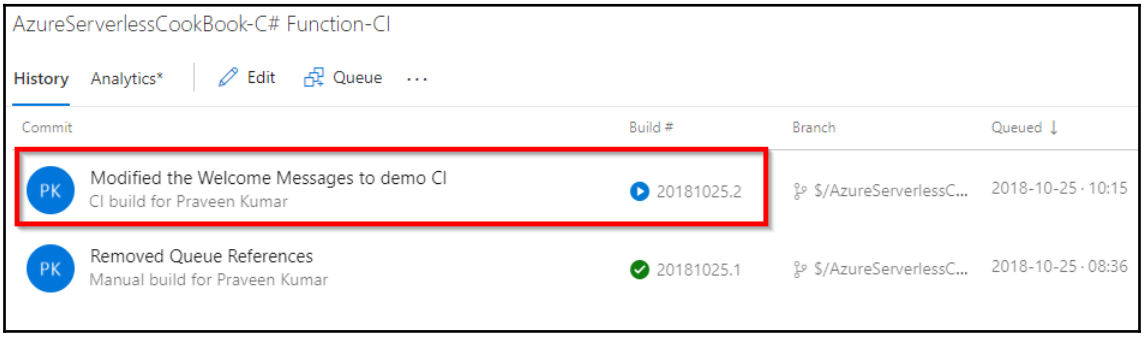
5. Let's navigate to the Azure Function project in Visual Studio. Make a small change to the last line of the `Run` function source code that is shown next. I just replaced the word `hello` with `Automated Build Trigger test by`:

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

6. Let's check in the code and commit the changes to the source control. As shown here, you will get a new change set ID generated. In this case, it is **Changeset 32**:



7. Now, immediately navigate back to the Azure DevOps build definition to see that a new build got triggered automatically and is in progress, as shown in the following screenshot:



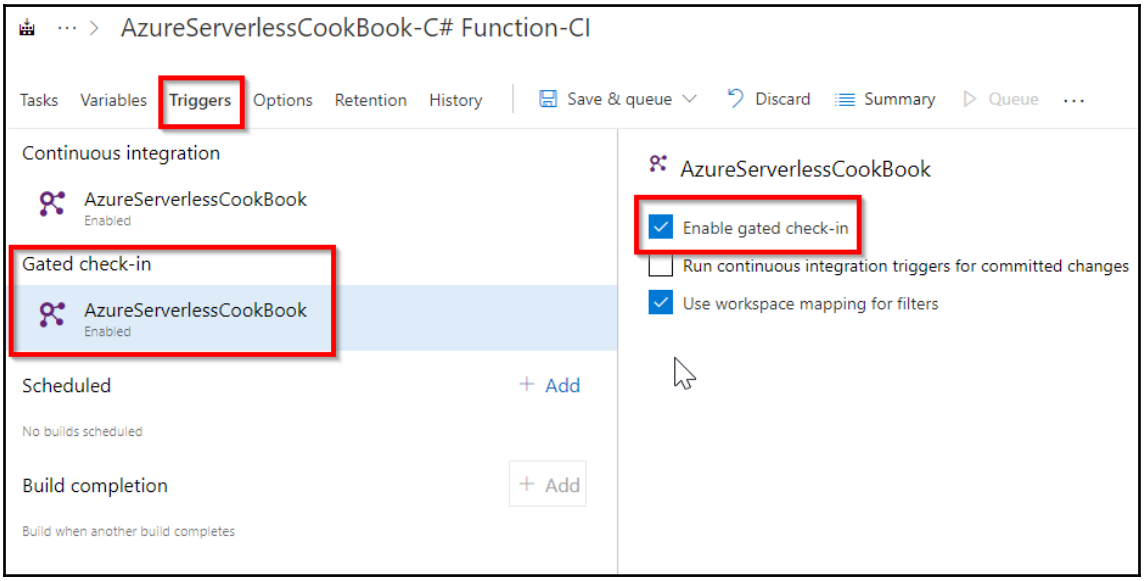
How it works...

These are the steps followed in this recipe:

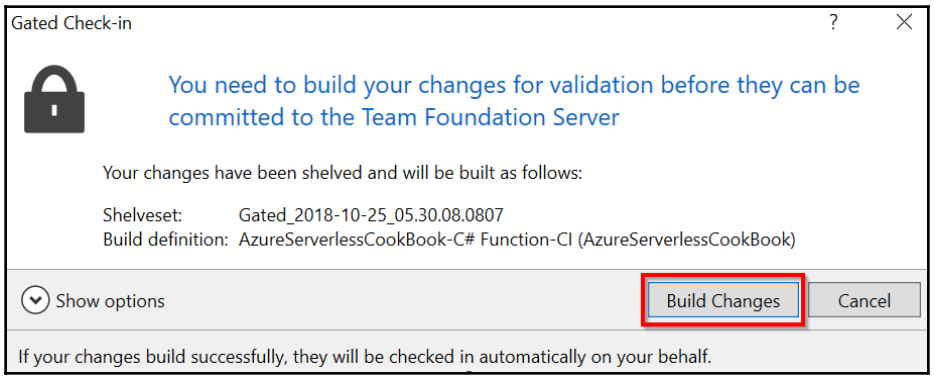
1. We enabled the automatic build trigger for the build definition
2. We made a change to the codebase and checked it into TFVC
3. Automatically, a new build got triggered in Azure DevOps—builds immediately after the code is committed to the TFVC

There's more...

If you would like to restrict the developers to checking in code only after a successful build, then you need to enable gated-check-in. In order to enable this, edit the build definition and then navigate to the **Triggers** tab and **Enable gated check-in**, as shown in the following screenshot:



Now, go back to Visual Studio and make some changes to the code. If you try to check in the code without building the application from within Visual Studio, then you will get an alert, as shown here:



Click on **Build Changes** in the preceding step to start the build in Visual Studio. As soon as the build in Visual Studio is complete, the code will be checked into Azure DevOps and then a new build will be triggered automatically, as shown here:

Continuous integration – executing unit test cases in the pipeline

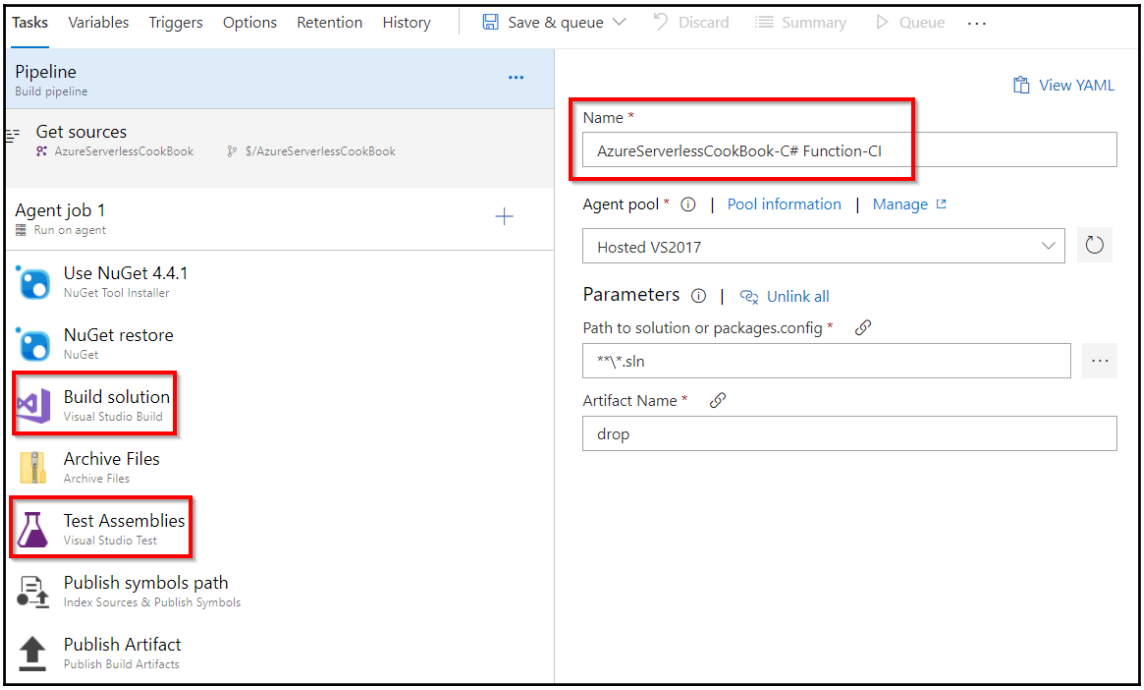
One of the most important steps in any software development methodology is to write automated unit tests that validate the correctness of our code. It is also important that we run these unit tests every time the developer releases new code, to provide test code coverage.

In this recipe, we will learn how to incorporate the process of building the unit tests that we developed in the *Developing Unit Tests for Azure Functions HTTP Triggers* recipe of `Chapter 6`, *Exploring Testing Tools for the Validation of Azure Functions.*

How to do it..

Perform the following steps:

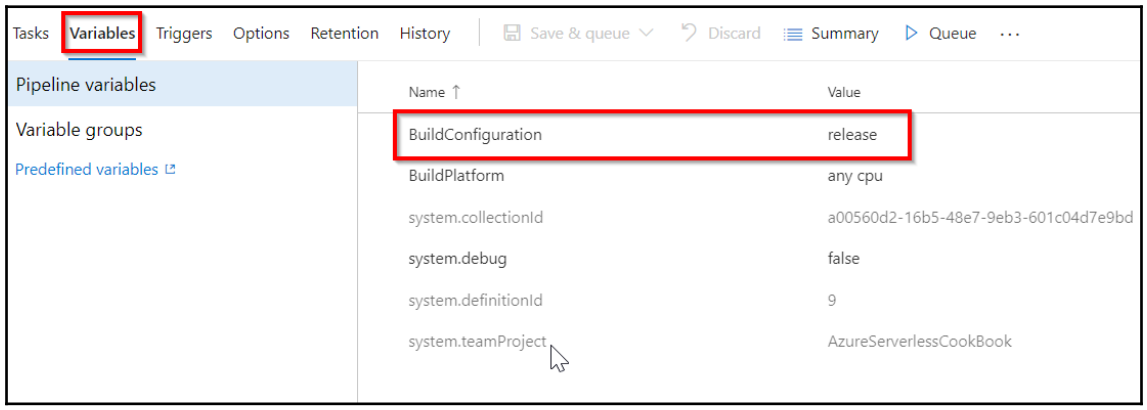
1. In the *Continuous integration - creating a build definition* recipe of this chapter, we utilized a build template that had both the **Build solution** and build **Test Assemblies** steps, as shown in the following screenshot:



2. Click on **Test Assemblies** as shown in the previous screenshot. Replace the default configuration with the one provided here, in the **Test files** field:

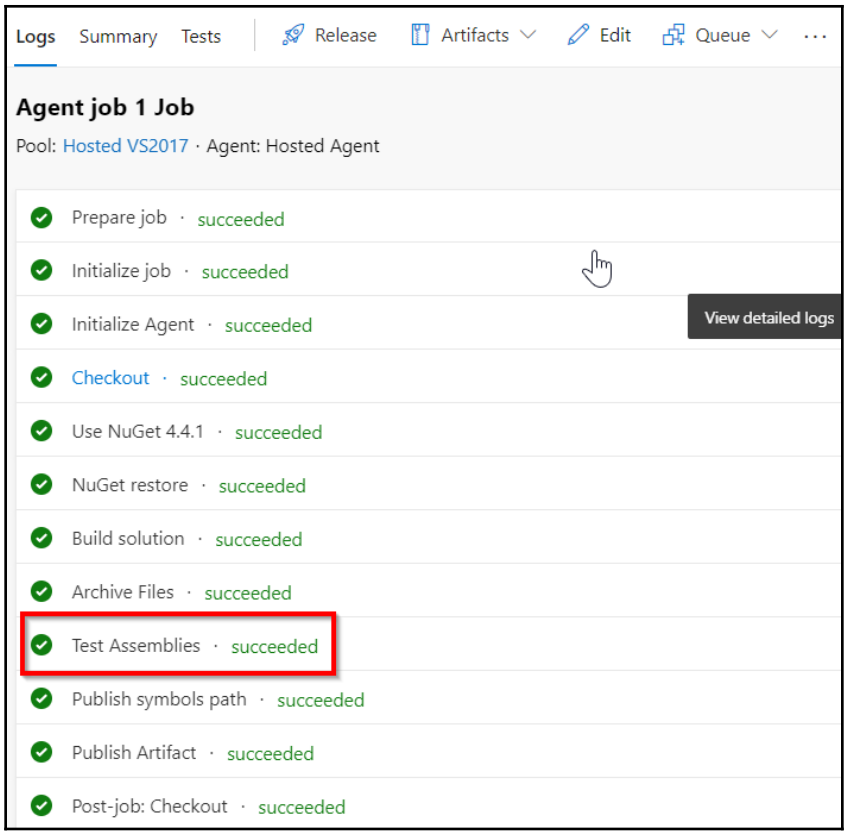
```
**\$(BuildConfiguration)\**\*test*.dll
!**\obj\**
!**\*TestAdapter.dll
```

3. The previous configuration settings let the test runner do the following:
    - Search for any `.dll` file with the word **Test** in its name that is located in the `release` folder any where in the artifacts. You might be wondering where `release` has come into picture. It's the value of the `$(BuildConfiguration)` variable in the **Variables** section shown in the following screenshot:

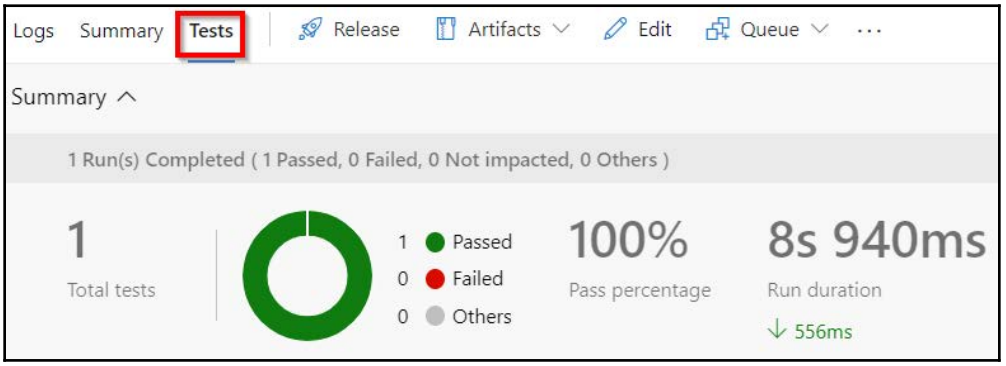


    - Ignore all the `.dll` files in the `obj` folder as our goal is to work only on the `.dll` files located inside the `release` folder.

4. That's it. Let's now queue the build by clicking on the **Queue** button after saving the changes. After a few minutes, the build pipeline will get passed without any warnings, as shown in the following screenshot:



5. Here is the summary of the test cases. You can see the chart that shows the percentage of the test cases that have passed and failed:



There's more...

If you have followed all the naming conventions as per the instructions, then you won't face any issues with this recipe. However, if you have used a different name for the unit project and if you haven't used the word `test` somewhere in the name of project (which is the same name as the generated `.dll` file), then feel free to change the format in the following setting:

```
**\$(BuildConfiguration)\**\*whateverwordyouhaveinthenameofthedllfile*.dll
```

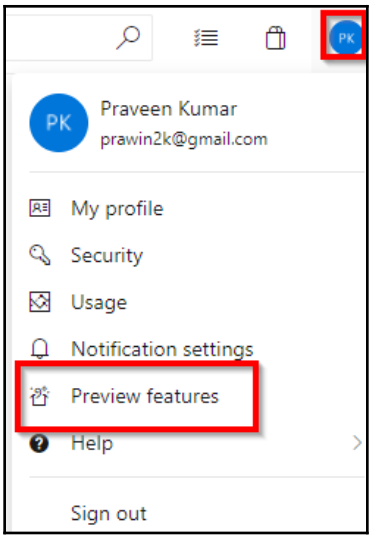
In the recipe, you used `*`, `**`, and `!`, which are called file matching patterns. You can learn more about the file matching patterns at `https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/file-matching-patterns?view=vsts`.

Creating a release definition

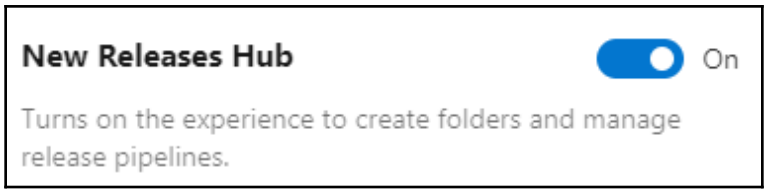
Now that we know how to create a build definition and trigger an automated build in Azure DevOps—pipelines, our next step is to release or deploy the package to an environment where the project stakeholders can review it and provide feedback. In order to do that, first we need to create a release definition in the same way that we created the build definitions.

Getting ready

I have used the new release definition editor to visualize the deployment pipelines. The release definition editor is still in preview. By the time you read this, if it is still in preview, then you can enable it by clicking on the profile image and then clicking on the **Preview features** link, as shown in the following screenshot:



You can then enable new release definition editor, as shown in the following screenshot:

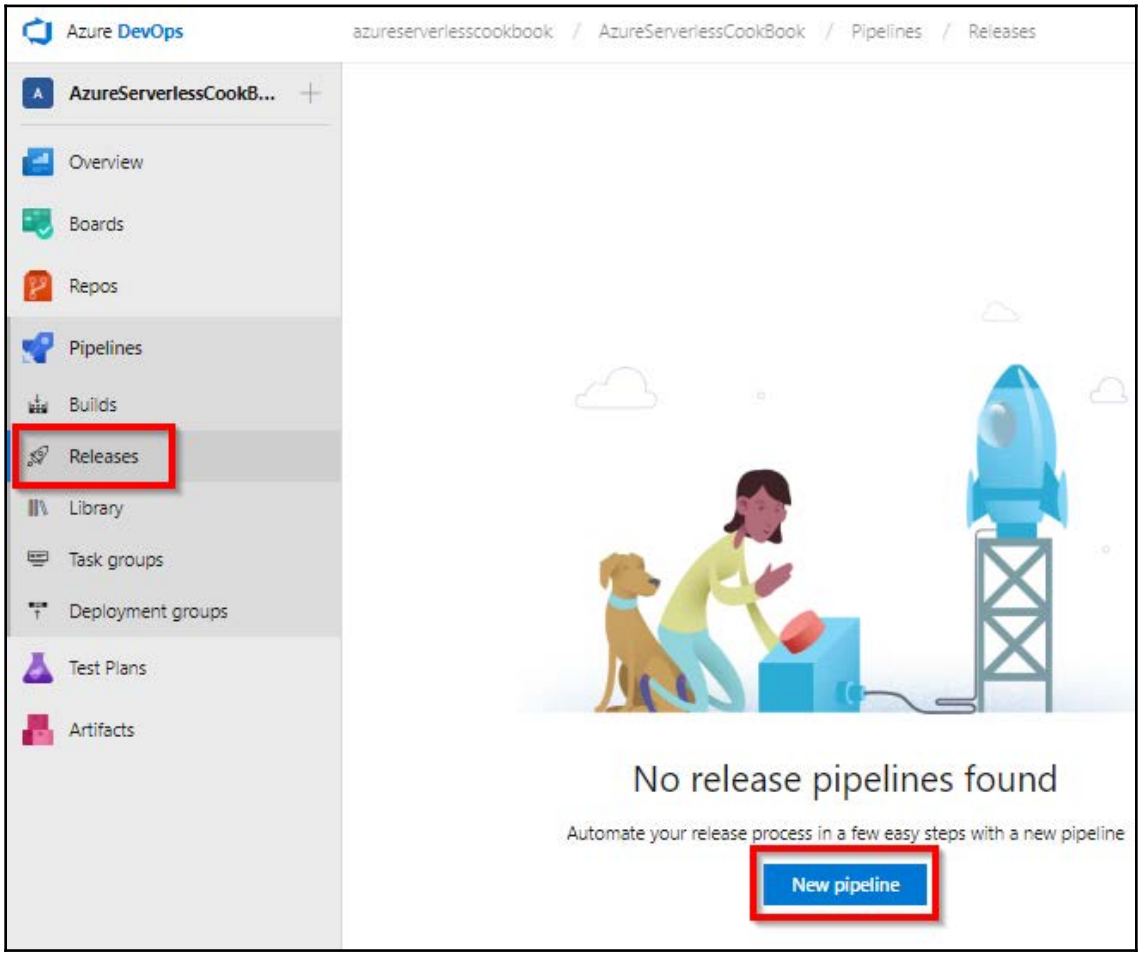


Let's get started with creating a new release definition.

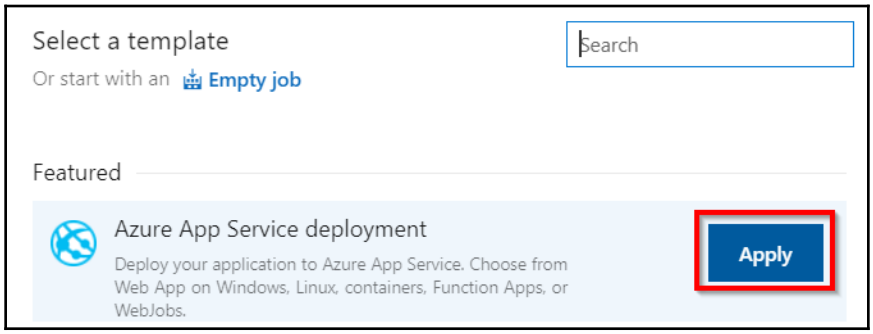
How to do it...

Perform the following steps:

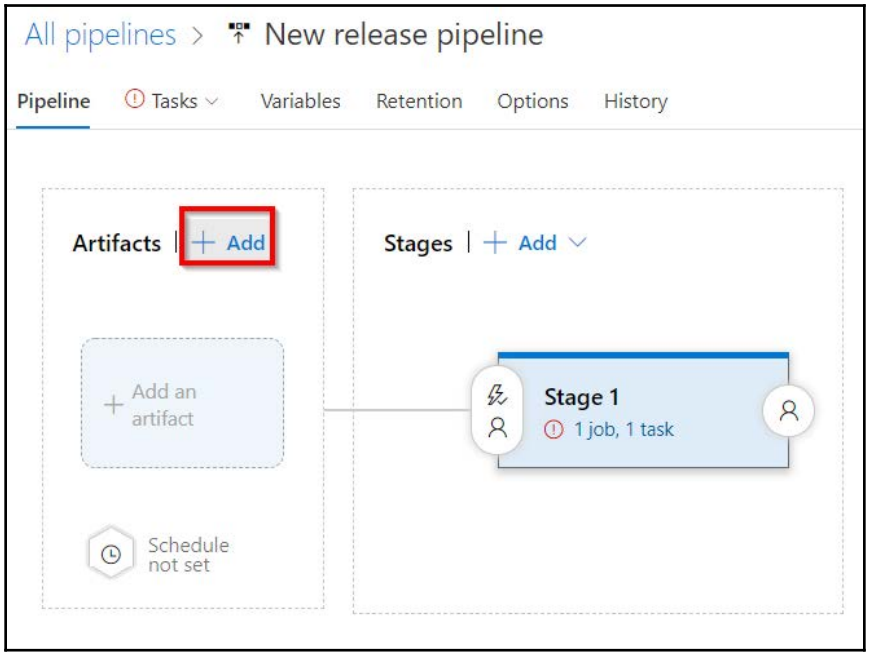
1. Navigate to the **Releases** tab, as shown in the following screenshot, and click on the **New Pipeline** link:



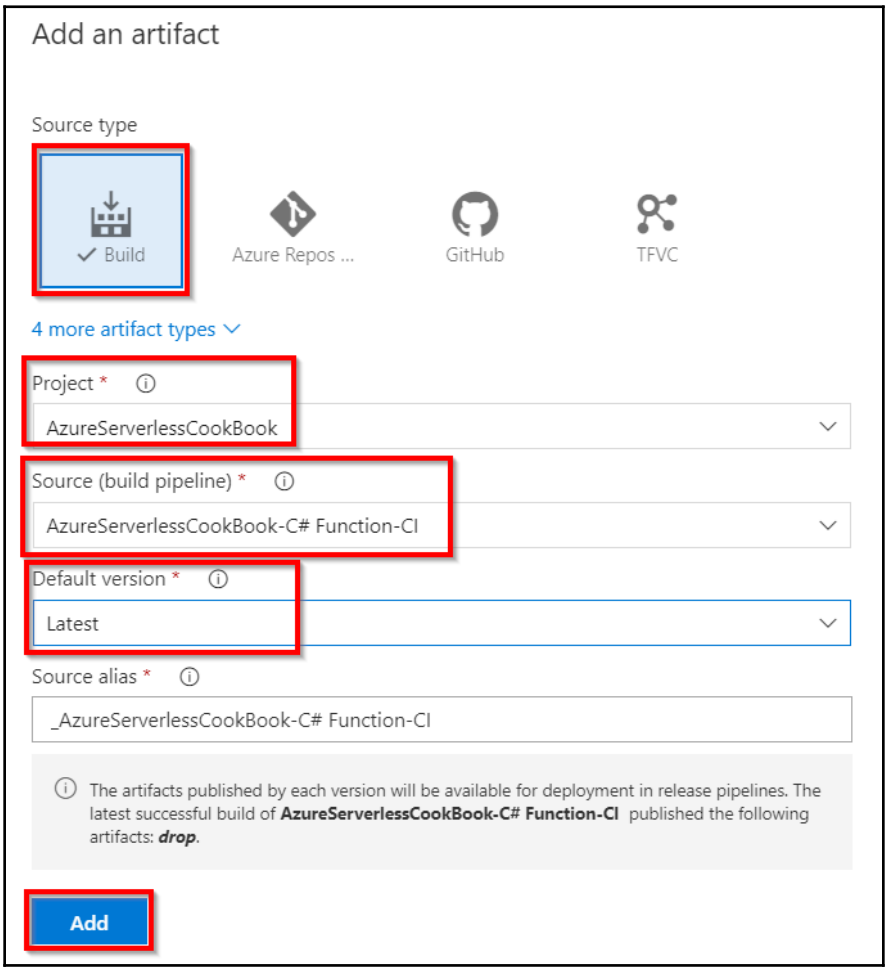
2. The next step is to choose a **Release definition** template. In the **Select a template** popup, select **Azure App Service deployment** and click on the **Apply** button, as shown in the following screenshot. Immediately after clicking on the **Apply** button, a new environment (stage) popup will be displayed. For now, just close the **Environment** popup:



3. Click on the **Add** button available in the **Artifacts** box to add a new artifact, as shown in the following screenshot:

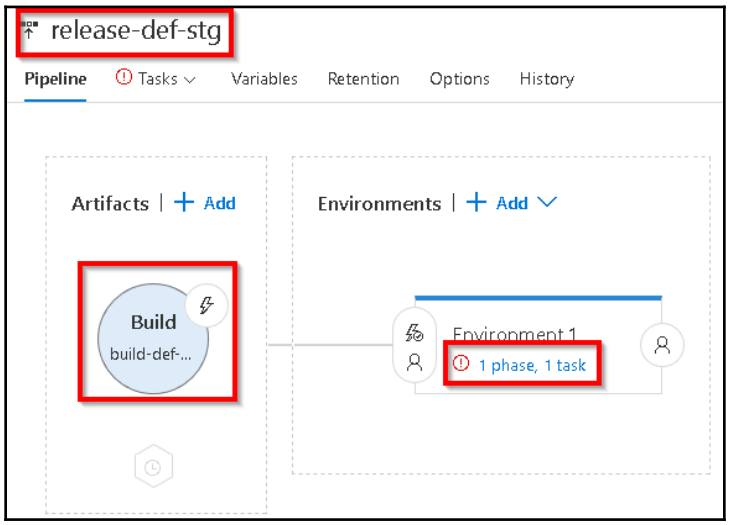


4. In the **Add an artifact** popup, make sure that you choose the following:
    1. **Source type**: **Build**
    2. **Project**: The team project your source code is linked to
    3. **Source (build definition)**: The build definition name where your builds are created
    4. **Default Version**: **Latest**

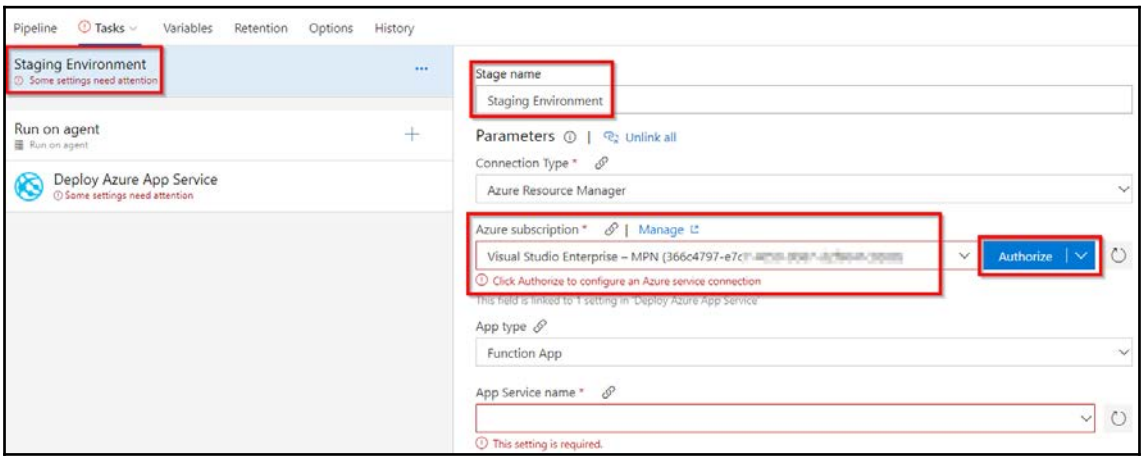


5. After reviewing all the values on the page, click on the **Add** button to add the artifact.

6. Once the artifact is added, the next step is to configure the stages, where the package needs to be published. Click on the **1 phase, 1 task** link, shown in the following screenshot. Also, change the name of the release definition to `release-def-stg`:

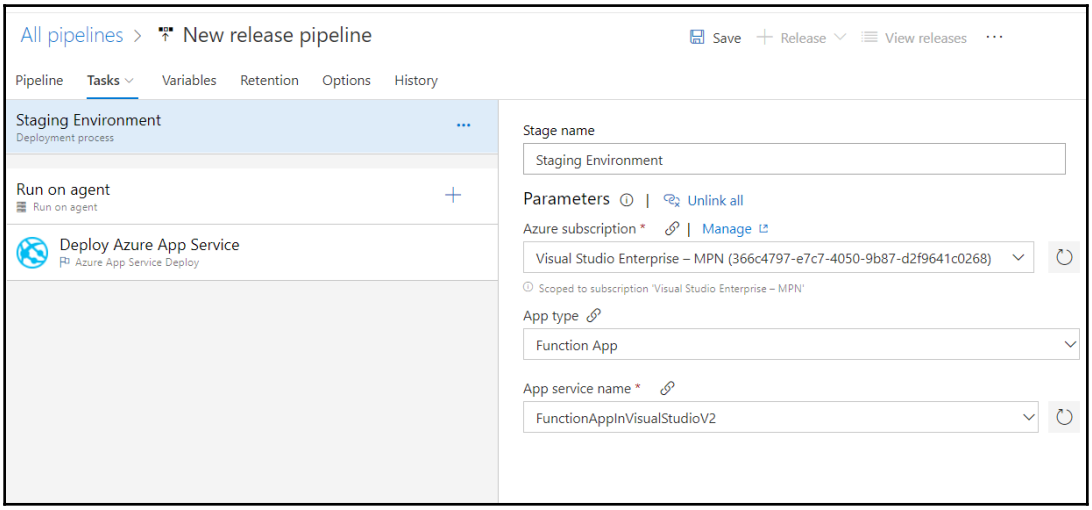


7. You will be taken to the **Tasks** tab, shown next. Provide a meaningful name to the **Stage name** field. I have provided the name `Staging Environment` for this example. Next, choose the Azure subscription in which you would like to deploy the Azure Function. You will need to click on the **Authorize** button to provide the permissions, as shown here:

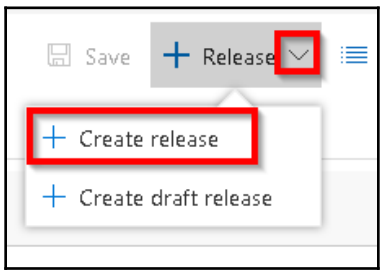


8. After authorizing the subscription, you need to ensure that you have chosen the **App type** to be **Function App**, and then choose the function app name in the **App Service name** to which you would like to deploy the package, as shown here:

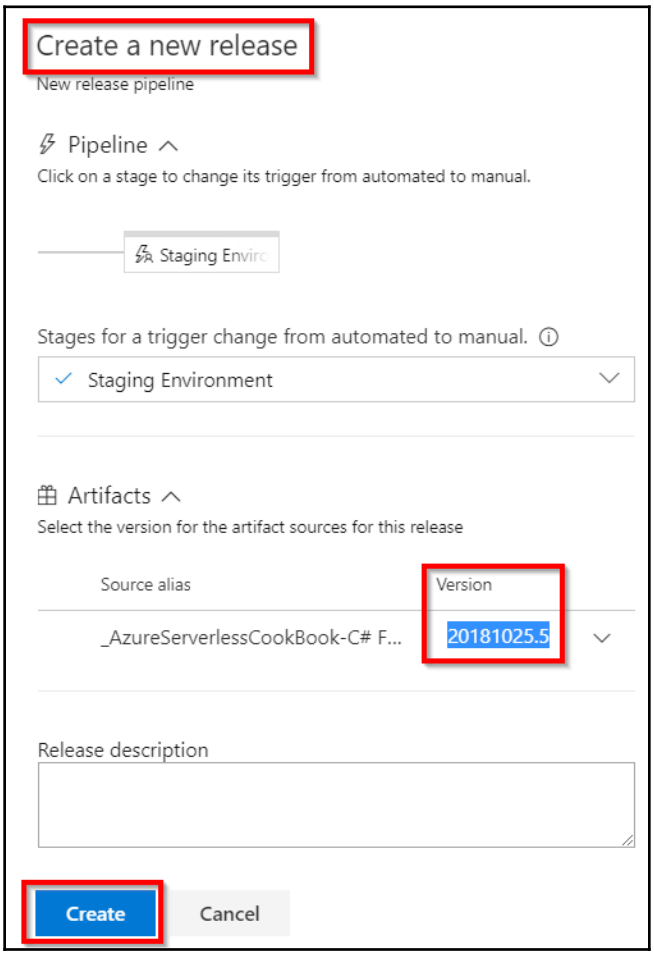
If you don't see your subscription or app service, refresh the item by clicking on the icon that is available after the Authorize button of the above screen screenshot.



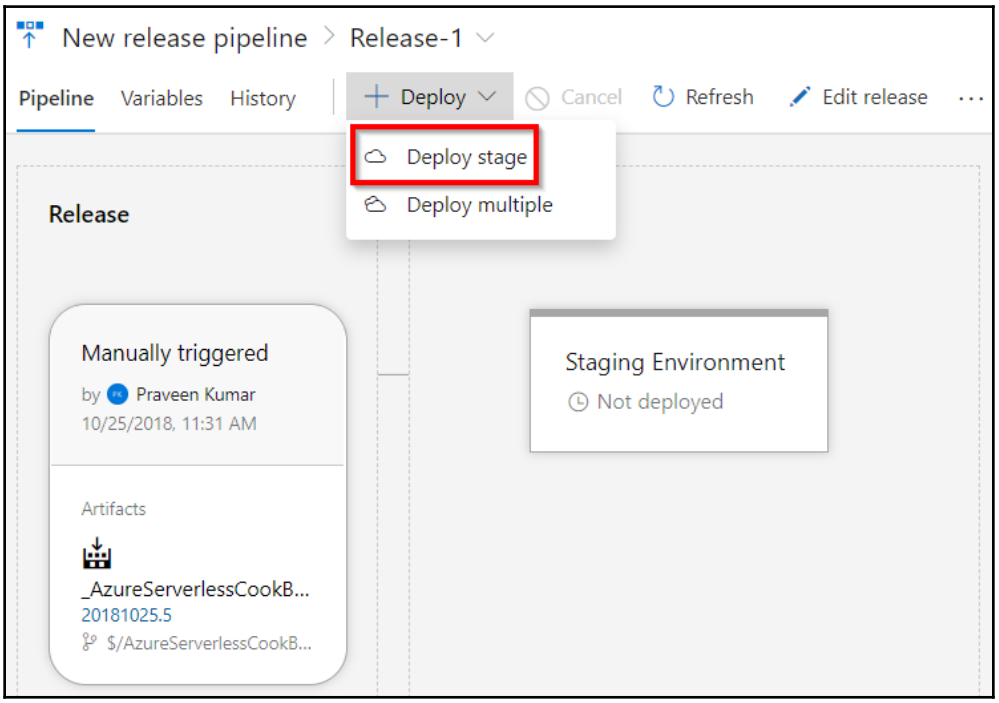
9. Click on the **Save** button to save the changes. Now, let's use this release definition and try to create new release by clicking on **Create release**, as shown in the following screenshot:



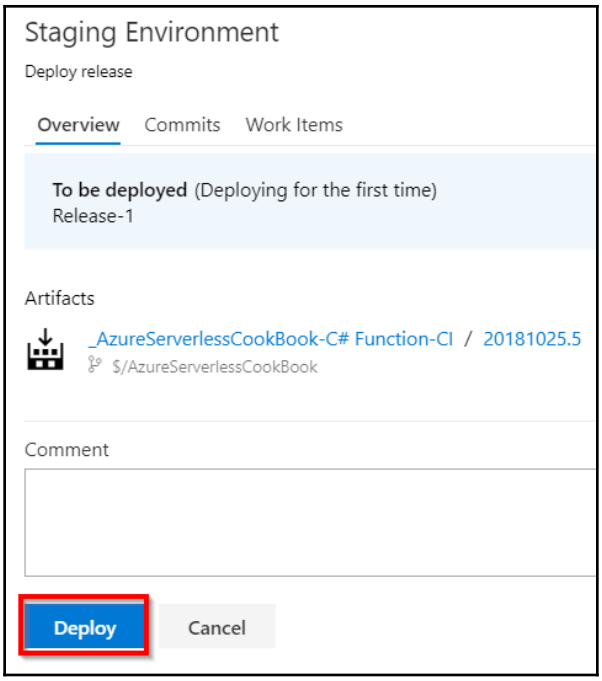
10. Next, you will be taken to the **Create a new release** popup where you can configure the build definition that needs to be used. As we have only one, we can see only one build definition. You also need to choose the right version of the build, as shown here. Once you review it, click on the **Create** button to queue the release:



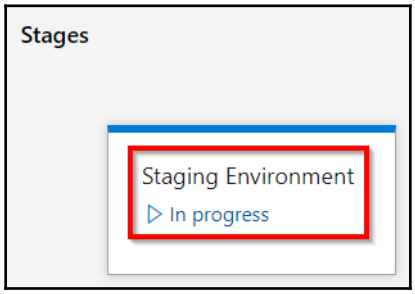
11. Clicking on the **Create** button in the preceding step will take you to the following step. Click on the **Deploy stage** button as shown here to initiate the process of deploying the release:



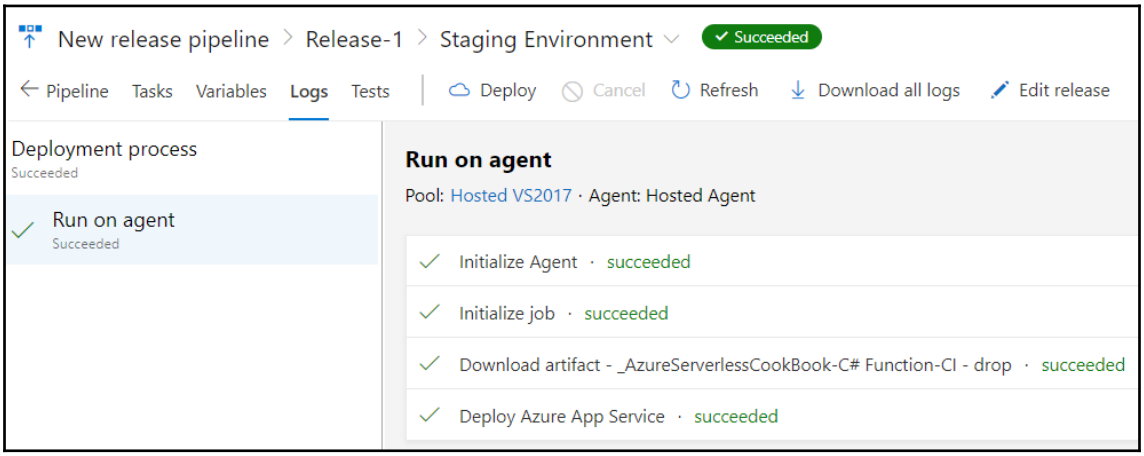
12. You will now be prompted to review the associated artifacts. Once you review them, click on the **Deploy** button as shown here:



13. Immediately, the process will start and it will show **In Progress** to indicate the progress of the release, as shown here:



14. Click on the **In Progress** links shown in the previous screenshot to review the progress. As shown next, the release process succeeded:



How it works...

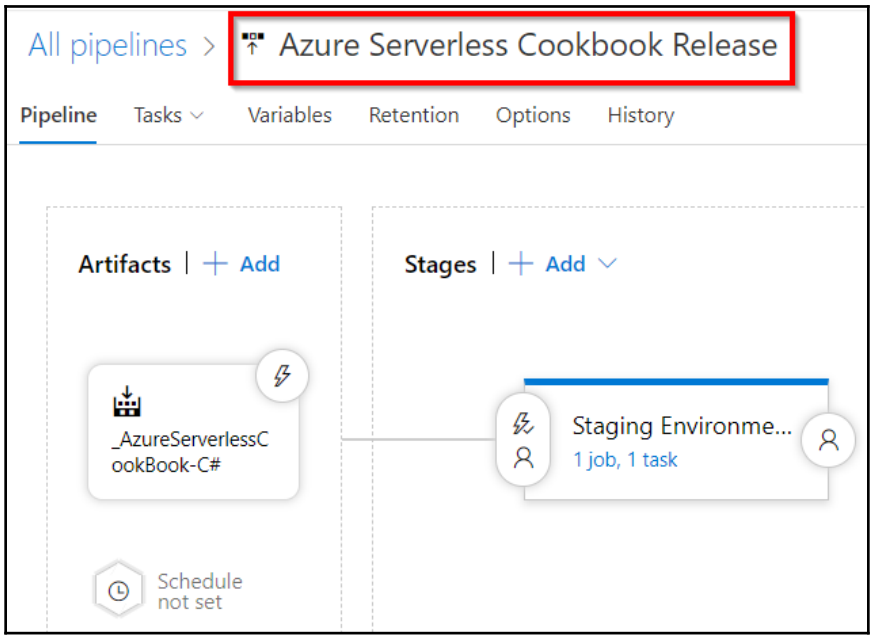
In the **Pipeline** tab, we have created artifacts and an environment named staging, and linked them together.

We have also configured the environment to have the Azure App Service related to the Azure Functions that we created in `Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools for Azure Functions*.

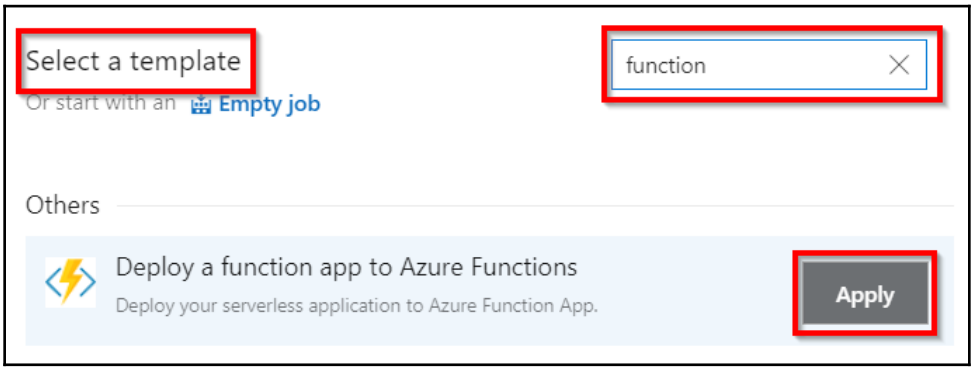
There's more...

If you are configuring continuous deployment for the first time, you might see a button with the text **Authorize** in the Azure **App Service deployment** step. Clicking on the **Authorize** button will open a pop-up window where you will be prompted to provide your Azure Management portal's credentials.

You can rename the release pipeline by clicking on the name at the top, as shown here:



Currently, there is a template specific to Azure Functions, shown next. However, it looks like it's not working. By the time you read this book, you should probably give a try:



See also

The *Deploying an Azure Function app to Azure Cloud using Visual Studio* recipe of `Chapter 4`, *Understanding the Integrated Developer Experience of Visual Studio Tools for Azure Functions.*

Triggering the release automatically

In this recipe, you will learn how to configure continuous deployment for an environment. In your project, you can configure dev/staging or any other pre-production environment, and configure continuous deployment to streamline the deployment process.

In general, it is not recommended that you configure continuous deployment for a production environment. However, this might depend on various factors and requirements. Be cautious and think about various scenarios before you configure continuous deployment for your production environment.

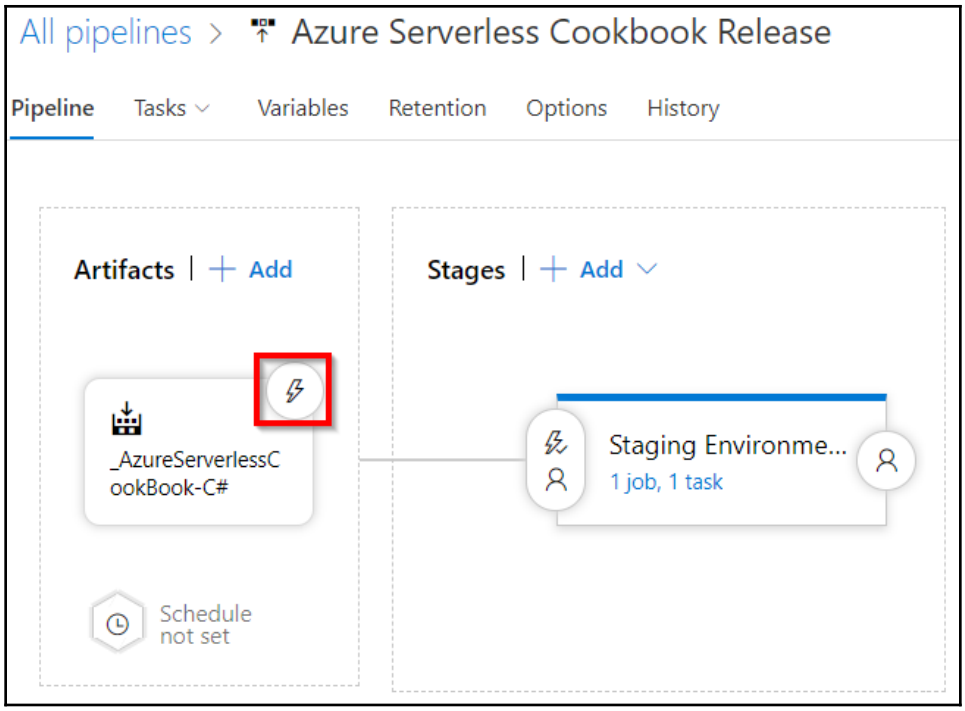
Getting ready

Download and install the Postman tool if you have not installed it yet.

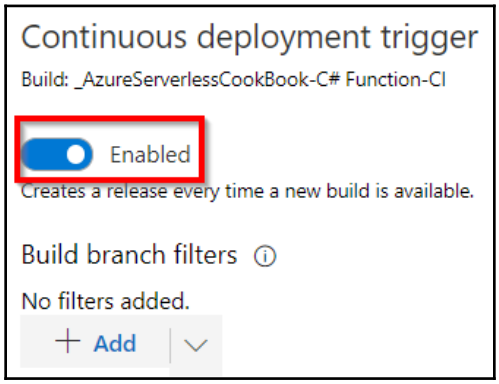
How to do it...

Perform the following steps:

1. By default, the releases are configured to be pushed manually. Let's configure Continuous Deployment by navigating back to the **Pipeline** tab and clicking on the **Continuous deployment trigger**, as shown in the following screenshot:



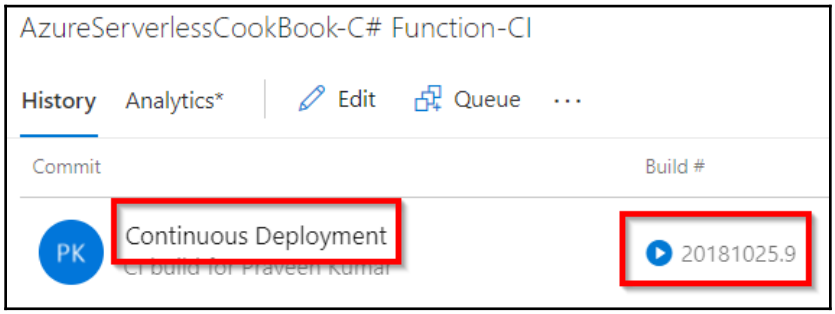
2. As shown in the following screenshot, enable the **Continuous deployment trigger** and click on **Save** to save the changes:



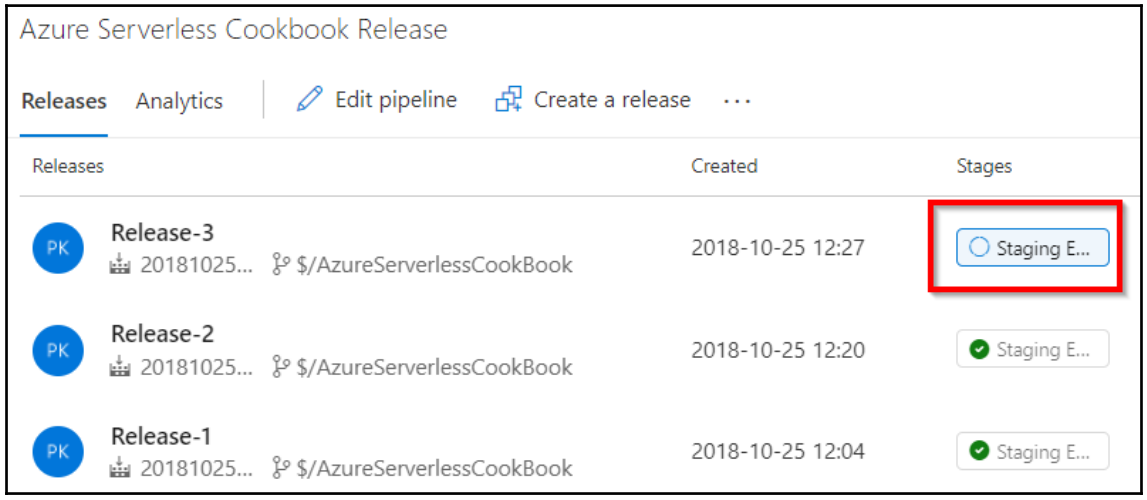
3. Navigate to Visual Studio and make some code changes, as highlighted here:

```
return name != null ? (ActionResult)new OkObjectResult($"Automated
Build Trigger and Release test by, { name}")
          : new BadRequestObjectResult("Please pass a name on the
query string or in the request body");
```

4. Now, check in the code with a comment, **Continuous Deployment**, to commit the changes to Azure DevOps. As soon as you check in the code, navigate to the **Builds** tab to see a new build get triggered, as shown in the following screenshot:



5. Navigate to the **Releases** tab after the build is complete to see that a new release got triggered automatically, as shown in the following screenshot:



6. Once the release process is complete, you can review the change by making a request to the HTTP Trigger using the Postman tool:

How it works...

In the **Pipeline** tab, we have enabled the **Continuous deployment trigger**.

Every time a build associated with `AzureServerlessCookBook-C# Function-CI` is triggered, the `Azure Serverless Cookbook Release` release will be automatically triggered to deploy the latest build to the designated environment. We have also seen the automatic release in action by making a code change in Visual Studio.

There's more...

You can also create multiple environments and configure the definitions to release the required builds to those environments.

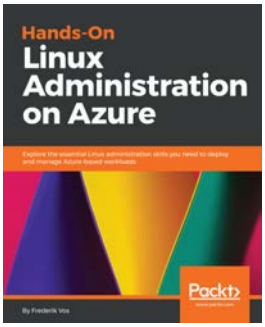
Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Architecting Microsoft Azure Solutions – Exam Guide 70-535
Sjoukje Zaal

ISBN: 978-1-78899-173-5

- Use Azure Virtual Machines to design effective VM deployments
- Implement architecture styles, like serverless computing and microservices
- Secure your data using different security features and design effective security strategies
- Design Azure storage solutions using various storage features
- Create identity management solutions for your applications and resources
- Architect state-of-the-art solutions using Artificial Intelligence, IoT, and Azure Media Services
- Use different automation solutions that are incorporated in the Azure platform



Hands-On Linux Administration on Azure
Frederik Vos

ISBN: 978-1-78913-096-6

- Understand why Azure is the ideal solution for your open source workloads
- Master essential Linux skills and learn to find your way around the Linux environment
- Deploy Linux in an Azure environment
- Use configuration management to manage Linux in Azure
- Manage containers in an Azure environment
- Enhance Linux security and use Azure's identity management systems
- Automate deployment with Azure Resource Manager (ARM) and Powershell
- Employ Ansible to manage Linux instances in an Azure cloud environment

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

T

U

V

W