Azure Serverless Computing Cookbook

Third Edition

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

Praveen Kumar Sreeram

Azure Serverless Computing Cookbook

Third Edition

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

Praveen Kumar Sreeram

Azure Serverless Computing Cookbook, Third Edition

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Author: Praveen Kumar Sreeram

Technical Reviewers: Stefano Demiliani, Greg Leonardo, and Kasam Shaikh

Managing Editor: Mamta Yadav

Acquisitions Editors: Rahul Hande and Suresh Jain

Production Editor: Deepak Chavan

Editorial Board: Vishal Bodwani, Ben Renow-Clarke, Joanne Lovell, Arijit Sarkar, and Dominic Shakeshaft

First Edition: August 2017

Second Edition: November 2018

Third Edition: May 2020

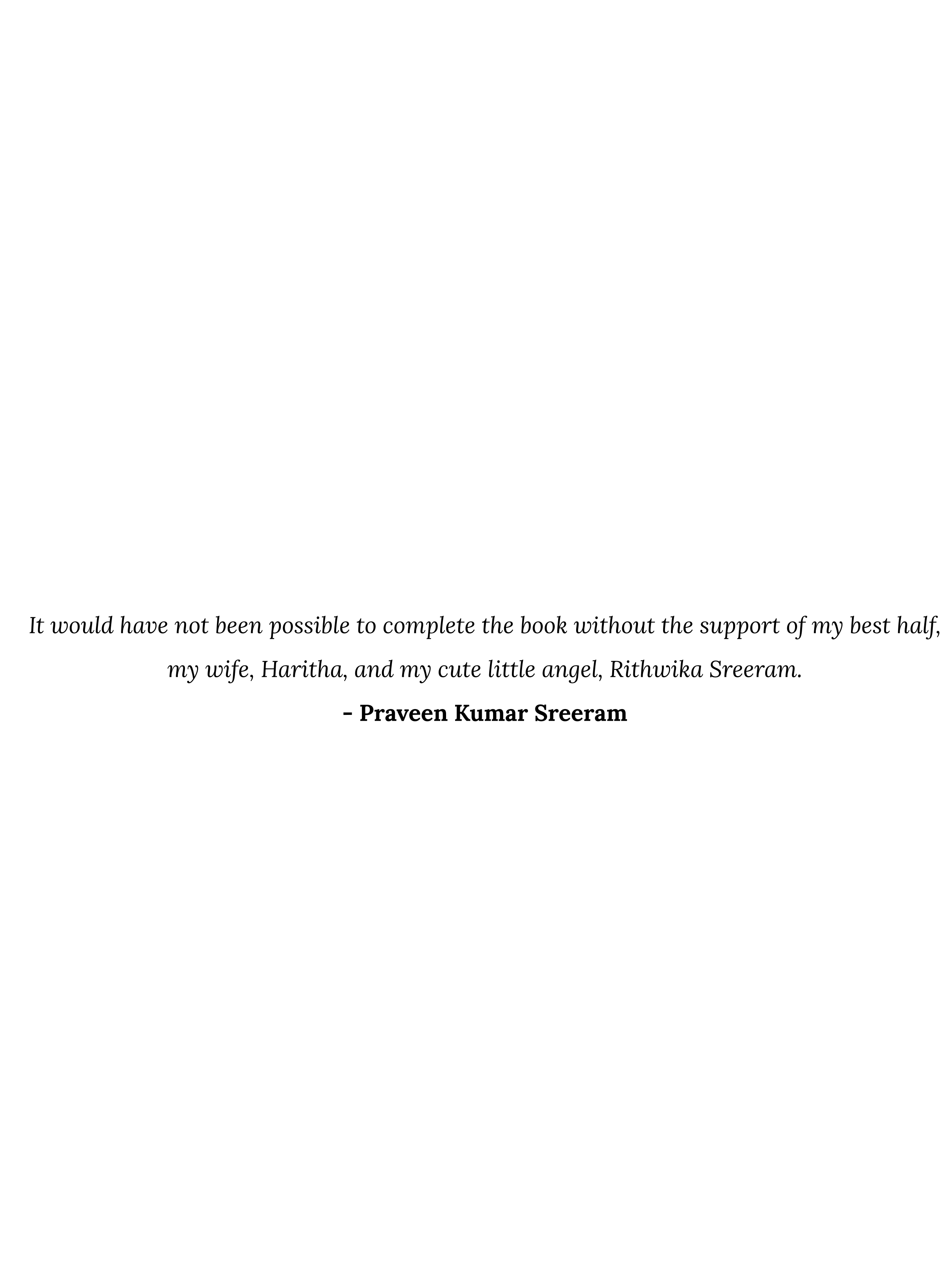
Production Reference: 1280520

ISBN: 978-1-80020-660-1

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK



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.

- Praveen Kumar Sreeram

Table of Contents

Preface i

Chapter 1: Accelerating cloud app development using Azure Functions 1

Introduction ... 2
Building a back-end web API using HTTP triggers ... 3
- Getting ready ... 3
- How to do it... ... 4
- How it works... ... 9
- See also ... 9

Persisting employee details using Azure Table Storage output bindings ... 9
- Getting ready ... 10
- How to do it... ... 10
- How it works... ... 13

Saving profile picture paths to queues using queue output bindings ... 15
- Getting ready ... 15
- How to do it... ... 15
- How it works... ... 17

Storing images in Azure Blob Storage ... 17
- Getting ready ... 17
- How to do it... ... 18
- How it works... ... 20
- There's more... ... 20

Resizing an image using an ImageResizer trigger 20
Getting ready 21
How to do it... 21
How it works... 25

Chapter 2: Working with notifications using the SendGrid and Twilio services 27

Introduction 28
Sending an email notification using SendGrid service 29
Getting ready 29
Creating a SendGrid account API key from the Azure portal 29
Generating credentials and the API key from the SendGrid portal 31
Configuring the SendGrid API key with the Azure Function app 33
How to do it... 33
Creating a storage queue binding to the HTTP trigger 33
Creating a queue trigger to process the message of the HTTP trigger 35
Creating a SendGrid output binding to the queue trigger 36
How it works... 38
Sending an email notification dynamically to the end user 39
Getting ready 39
How to do it... 39
Accepting the new email parameter in the RegisterUser function 39
Retrieving the UserProfile information in the SendNotifications trigger 40
How it works... 42
There's more... 42
Implementing email logging in Azure Blob Storage 43
How to do it... 43
How it works... 45

Modifying the email content to include an attachment ... 45
Getting ready ... 46
How to do it... ... 46
Customizing the log file name using the IBinder interface ... 46
Adding an attachment to the email ... 47
Sending an SMS notification to the end user using the Twilio service ... 48
Getting ready ... 49
How to do it... ... 51
How it works... ... 53

Chapter 3: Seamless integration of Azure Functions with Azure Services 55

Introduction ... 56
Using Cognitive Services to locate faces in images ... 56
Getting ready ... 56
How to do it... ... 58
There's more... ... 64
Monitoring and sending notifications using Logic Apps ... 65
Getting ready ... 66
How to do it... ... 66
How it works... ... 74
Integrating Logic Apps with serverless functions ... 74
How to do it... ... 75
There's more... ... 79
Auditing Cosmos DB data using change feed triggers ... 79
Getting ready ... 80
How to do it... ... 82
There's more... ... 86

Integrating Azure Functions with Data Factory pipelines 87
Getting ready... 88
How to do it... 96

Chapter 4: Developing Azure Functions using Visual Studio 111

Introduction 112
Creating a function application using Visual Studio 2019 112
Getting ready 113
How to do it... 113
How it works... 115
There's more... 115
Debugging Azure Function hosted in Azure using Visual Studio 115
Getting ready 116
How to do it... 116
How it works... 120
There's more... 120
Connecting to the Azure Storage from Visual Studio 120
Getting ready 121
How to do it... 121
How it works... 124
There's more... 124
Deploying the Azure Function application using Visual Studio 125
How to do it... 125
There's more... 128
Debugging Azure Function hosted in Azure using Visual Studio 128
Getting ready 129
How to do it... 129

Deploying Azure Functions in a container 133
Getting ready 133
Creating an ACR 134
How to do it... 135
Creating a Docker image for the function application 136
Pushing the Docker image to the ACR 137
Creating a new function application with Docker 139
How it works... 143

Chapter 5: Exploring testing tools for Azure functions 145

Introduction 146
Testing Azure functions 146
Getting ready 146
How to do it... 146
Testing HTTP triggers using Postman 147
Testing a blob trigger using Storage Explorer 149
Testing a queue trigger using the Azure portal 152
There's more... 155
Testing an Azure function in a staging environment using deployment slots 155
How to do it... 156
There's more... 161
Creating and testing Azure functions locally using Azure CLI tools 163
Getting ready 163
How to do it... 163

Validating Azure function responsiveness using Application Insights 166
Getting ready 167
How to do it... 168
How it works... 177
There's more... 177
Developing unit tests for Azure functions with HTTP triggers 177
Getting ready 177
How to do it... 178

Chapter 6: Troubleshooting and monitoring Azure Functions 183

Introduction 184
Troubleshooting Azure Functions 184
How to do it... 184
Viewing real-time application logs 185
Diagnosing the function app 186
Integrating Azure Functions with Application Insights 188
Getting ready 188
How to do it... 188
How it works... 190
There's more... 191
Monitoring Azure Functions 191
How to do it... 191
How it works... 193
Pushing custom metrics details to Application Insights Analytics 194
Getting ready 195
How to do it... 195
Creating a timer trigger function using Visual Studio 196
Configuring access keys 200

Integrating and testing an Application Insights query 202
Configuring the custom-derived metric report 203
How it works... 205
Sending application telemetry details via email 205
Getting ready 206
How to do it... 206
How it works... 212
Integrating Application Insights with Power BI using Azure Functions 212
Getting ready 214
How to do it... 214
Configuring Power BI with a dashboard, a dataset, and the push URI 214
Creating an Azure Application Insights real-time Power BI—C# function 219
How it works... 222
There's more... 223

Chapter 7: Developing reliable serverless applications using durable functions 225

Introduction 226
Configuring durable functions in the Azure portal 227
Getting ready 227
How to do it... 228
Creating a serverless workflow using durable functions 231
Getting ready 231
How to do it... 231
Creating the orchestrator function 231
Creating an activity function 233
How it works... 234
There's more... 234

Testing and troubleshooting durable functions 234
Getting ready 235
How to do it... 235
Implementing reliable applications using durable functions 237
Getting ready 237
How to do it... 238
Creating the orchestrator function 238
Creating a GetAllCustomers activity function 239
Creating a CreateBARCodeImagesPerCustomer activity function 240
How it works... 242
There's more... 243

Chapter 8: Bulk import of data using Azure Durable Functions and Cosmos DB 245

Introduction 246
Business problem 246
The durable serverless way of implementing CSV imports 247
Uploading employee data to blob storage 247
Getting ready 248
How to do it... 248
There's more... 251
Creating a blob trigger 252
How to do it... 252
There's more... 255
Creating the durable orchestrator and triggering it for each CSV import ... 255
How to do it... 255
How it works... 259
There's more... 259

Reading CSV data using activity functions ... 260
Getting ready ... 260
How to do it... ... 260
Reading data from blob storage ... 260
Reading CSV data from the stream ... 262
Creating the activity function ... 263
There's more... ... 265
Autoscaling Cosmos DB throughput ... 266
Getting ready ... 267
How to do it... ... 268
There's more... ... 269
Bulk inserting data into Cosmos DB ... 269
How to do it... ... 270
There's more... ... 271
Chapter 9: Configuring security for Azure Functions 273
Introduction ... 274
Enabling authorization for function apps ... 274
Getting ready ... 274
How to do it... ... 275
How it works... ... 276
There's more... ... 276
Controlling access to Azure Functions using function keys ... 276
How to do it... ... 277
There's more... ... 280
Securing Azure Functions using Azure Active Directory ... 281
Getting ready ... 281
How to do it... ... 281

Throttling Azure Functions using API Management 292
Getting ready 292
How to do it... 294
How it works... 299
Securely accessing an SQL database from Azure Functions using Managed Identity 300
How to do it... 300
Configuring additional security using IP whitelisting 309
Getting ready... 309
How to do it... 310
There's more 312

Chapter 10: Implementing best practices for Azure Functions 315

Introduction 316
Adding multiple messages to a queue using the IAsyncCollector function 316
Getting ready 317
How to do it... 317
There's more... 320
Implementing defensive applications using Azure functions and queue triggers 320
Getting ready 321
How to do it... 321
Running tests using the CreateQueueMessage console application 324
There's more... 325
Avoiding cold starts by warming the app at regular intervals 325
Getting ready 326
How to do it... 326

Sharing code across Azure functions using class libraries 328
How to do it... 328
There's more... 331
Migrating C# console application to Azure functions using PowerShell 332
Getting ready 333
How to do it... 333
Implementing feature flags in Azure functions using App Configuration ... 339
Getting ready 340
How to do it... 340

Chapter 11: Configuring serverless applications in the production environment 353

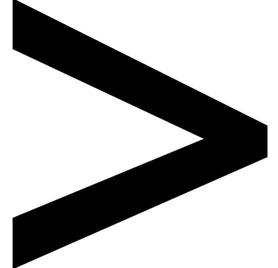
Introduction 354
Deploying Azure functions using the Run From Package feature 354
Getting ready 355
How to do it... 356
How it works... 358
Deploying Azure functions using ARM templates 358
Getting ready 358
How to do it... 359
There's more... 362
Configuring a custom domain for Azure functions 362
Getting ready 362
How to do it... 363
Techniques to access application settings 368
Getting ready 368
How to do it... 368

Breaking down large APIs into smaller subsets using proxies 372
Getting ready 373
How to do it... 374
There's more... 378
Moving configuration items from one environment to another 378
Getting ready 379
How to do it... 380

Chapter 12: Implementing and deploying continuous integration using Azure DevOps 385

Introduction 386
Prerequisites 387
Continuous integration—creating a build definition 388
Getting ready 388
How to do it... 389
How it works... 395
There's more... 396
Continuous integration—queuing a build and triggering it manually 396
Getting ready 396
How to do it... 397
Continuous integration—configuring and triggering an automated build 399
How to do it... 400
How it works... 403
Continuous integration—executing unit test cases in the pipeline 403
How to do it... 403
There's more... 406

Creating a release definition 406
Getting ready 406
How to do it... 406
How it works... 414
There's more... 414
Triggering a release automatically 415
Getting ready 415
How to do it... 415
How it works... 417
There's more... 417
Integrating Azure Key Vault to configure application secrets 418
How to do it... 418
How it works... 427
Index 429



Preface

About

This section briefly introduces the author, the reviewers, the coverage of this cookbook, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the recipes.

About Azure Serverless Computing Cookbook, Third Edition

This third edition of *Azure Serverless Computing Cookbook* guides you through the development of a basic back-end web API that performs simple operations, helping you understand how to persist data in Azure Storage services. You'll cover the integration of Azure Functions with other cloud services, such as notifications (SendGrid and Twilio), Cognitive Services (computer vision), and Logic Apps, to build simple workflow-based applications.

With the help of this book, you'll be able to leverage Visual Studio tools to develop, build, test, and deploy Azure functions quickly. It also covers a variety of tools and methods for testing the functionality of Azure functions locally in the developer's workstation and in the cloud environment. Once you're familiar with the core features, you'll explore advanced concepts such as durable functions, starting with a "hello world" example, and learn about the scalable bulk upload use case, which uses durable function patterns, function chaining, and fan-out/fan-in.

By the end of this Azure book, you'll have gained the knowledge and practical experience needed to be able to create and deploy Azure applications on serverless architectures efficiently.

About the author

Praveen Kumar Sreeram is an author, Microsoft Certified Trainer, and certified Azure Solutions Architect. He has over 15 years of experience in the field of development, analysis, design, and the delivery of applications of various technologies. His projects range from custom web development using ASP.NET and MVC to building mobile apps using the cross-platform Xamarin technology for domains such as insurance, telecom, and wireless expense management. He has been given the Most Valuable Professional award twice by one of the leading social community websites, CSharpCorner, for his contributions to the Microsoft Azure community through his articles. Praveen is highly focused on learning about technology, and blogs about his learning regularly. You can also follow him on Twitter at **@PrawinSreeram**. Currently, his focus is on analyzing business problems and providing technical solutions for various projects related to Microsoft Azure and .NET Core.

First of all, my thanks go to the Packt Publishing team, including Mamta Yadav, Arijit Sarkar, and Rahul Hande.

I would also like to express my deepest gratitude to A. Janardhan Setty, A. Vara Lakshmi and their family for all the support.

Thanks to Vijay Raavi, Ashis Nayak, and Manikanta Arrepu for their support in technical aspects.

About the reviewers

Stefano Demiliani is a Microsoft MVP in Business Applications, a Microsoft Certified DevOps Engineer and Azure Architect, and a long-time expert on Microsoft technologies. He works as a CTO for EID NAVLAB and his main activities are architecting solutions with Azure and Dynamics 365 ERP. He's the author of many IT books for Packt and a speaker on international conferences about Azure and Dynamics 365. You can reach him on Twitter or on LinkedIn.

Greg Leonardo is a veteran, father, developer, architect, teacher, speaker, and early adopter. He is currently a cloud architect helping organizations with cloud adoption and innovation. He has been working in the IT industry since his time in the military. He has worked in many facets of IT throughout his career. He is the president of TampaDev, a community meetup that runs #TampaCC, Azure User Group, Azure Medics, and various technology events throughout Tampa.

Kasam Shaikh, a Microsoft Certified Cloud Advocate, is a seasoned professional with a "can-do" attitude, having 13 years of demonstrated industry experience working as a cloud architect with one of the leading IT companies in Mumbai, India. He is the author of two best-selling books on Microsoft Azure and AI. He is also recognized as an MVP by a leading online community, C# Corner, and is also a global AzureAI speaker. He presents his tech-dose at KasamShaikh.com. He is the founder of DearAzure | Azure INDIA (AZ-INDIA) community—the fastest growing online community for learning Microsoft Azure.

Learning objectives

By the end of this book, you will be able to:

- Implement the continuous integration and continuous deployment (CI/CD) of Azure functions.
- Develop different event-based handlers in a serverless architecture.
- Integrate Azure functions with different Azure services to develop enterprise-level applications.
- Accelerate your cloud application development using Azure function triggers and bindings.
- Automate mundane tasks at various levels, from development to deployment and maintenance.
- Develop stateful serverless applications and self-healing jobs using durable functions.

Audience

If you are a cloud developer or architect who wants to build cloud-native systems and deploy serverless applications with Azure functions, this book is for you. Prior experience with Microsoft Azure core services will help you to make the most of this book.

Approach

This cookbook covers every aspect of serverless computing with Azure with a perfect blend of theory, hands-on coding, and helpful recipes. It contains several examples that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Hardware and software requirements

For an optimal learning experience, we recommend the following configuration:

- Visual Studio 2019
- Storage Explorer
- Azure Functions Core Tools (formerly Azure CLI Tools)
- Processor: Intel Core i5 or equivalent
- Memory: 4 GB RAM (8 GB preferred)
- Storage: 35 GB available space

Conventions

Code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"In this **`BlobTriggerCSharp`** class, the **`Run`** method has the **`WebJobs`** attribute with a connection string (in this case, it is **`AzureWebJobsStorage`**)."

Here's a sample block of code:

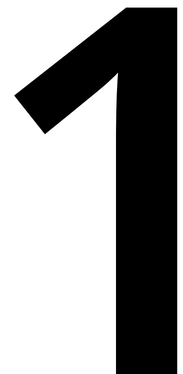
```
Install-Package Microsoft.Azure.Services.AppAuthentication
```

On many occasions, we have used angled brackets, **<>**. You need to replace these with the actual parameter, and not use these brackets within the commands.

Download resources

The code bundle for this book is also hosted on GitHub at https://github.com/PacktPublishing/Azure-Serverless-Computing-Cookbook-Third-Edition. You can find the YAML and other files used in this book, which are referred to at relevant instances.

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



Accelerating cloud app development using Azure Functions

In this chapter, we'll cover the following recipes:

- Building a back-end web API using HTTP triggers
- Persisting employee details using Azure Table storage output bindings
- Saving profile picture paths to queues using queue output bindings
- Storing images in Azure Blob Storage
- Resizing an image using an `ImageResizer` trigger

Introduction

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

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

Figure 1.1 highlights the key processes that you will learn about in this chapter:

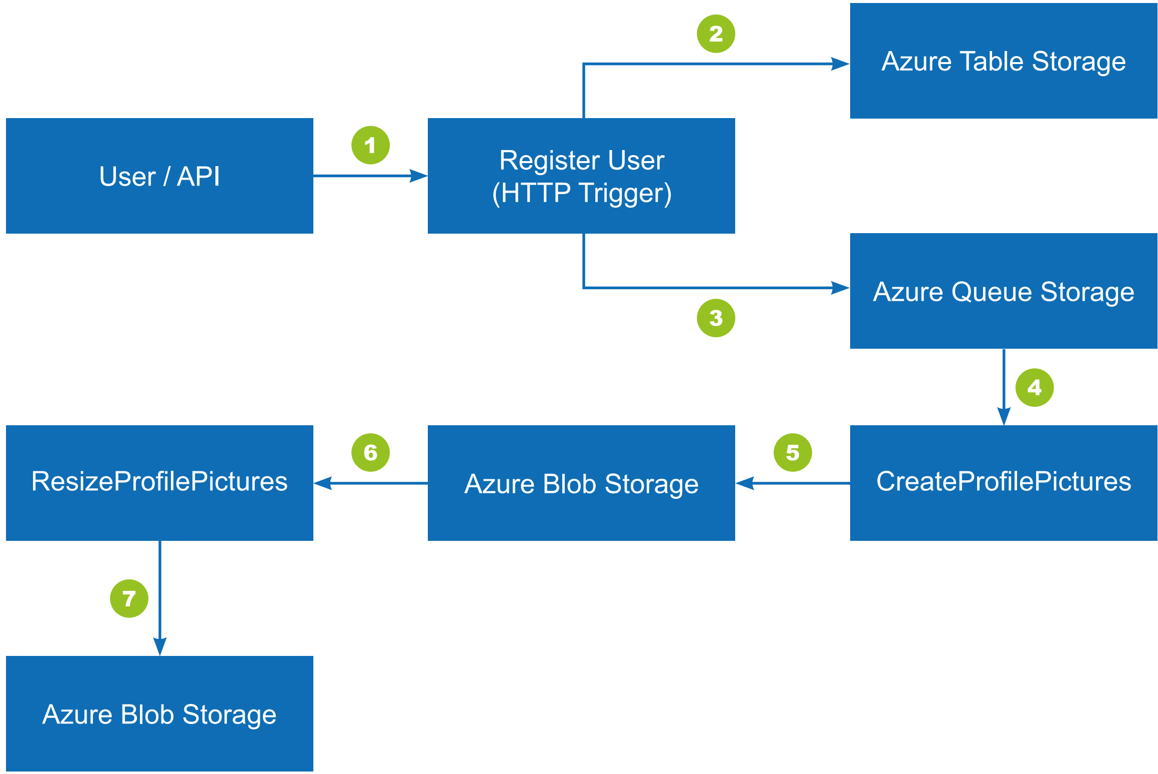


Figure 1.1: The key processes

Let's go through a step-by-step explanation of the figure to get a better understanding:

1. Client call to the **API**.
2. Persist employee details using **Azure Table Storage**.
3. Save profile picture links to queues.
4. Invoke a queue trigger as soon as a message is created.
5. Create the blobs in **Azure Blob Storage**.
6. Invoke the blob trigger as soon as a blob is created.
7. Resize the image and store it in **Azure Blob Storage**.

We'll leverage Azure Functions' built-in templates using HTTP triggers, with the goal of resizing and storing images in Azure Blob Storage.

Building a back-end web API using HTTP triggers

In this recipe, we'll use Azure's serverless architecture to build a web API using HTTP triggers. These HTTP triggers could be consumed by any front-end 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 back-end web API that responds to HTTP requests:

- Refer to https://azure.microsoft.com/free/ to see how to create a free Azure account.
- Visit https://docs.microsoft.com/azure/azure-functions/functions-create-function-app-portal to learn about the step-by-step process of creating a function application, and https://docs.microsoft.com/azure/azure-functions/functions-create-first-azure-function to learn how to create a function. While creating a function, a storage account is also created to store all the files.
- Learn more about Azure Functions at https://azure.microsoft.com/services/functions/.

> **Note**
>
> Remember the name of the storage account, as it will be used later in other chapters.

- Once the function application is created, please familiarize yourself with the basic concepts of triggers and bindings, which are at the core of how Azure Functions works. I highly recommend referring to https://docs.microsoft.com/azure/azure-functions/functions-triggers-bindings before proceeding.

> **Note**
>
> We'll be using C# as the programming language throughout the book. Most of the functions are developed using the Azure Functions V3 runtime. However, as of the time of writing, a few recipes were not supported in the V3 runtime. Hopefully, soon after the publication of this book, Microsoft will have made those features available for the V3 runtime as well.

How to do it...

Perform the following steps to build a web API using HTTP triggers:

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:

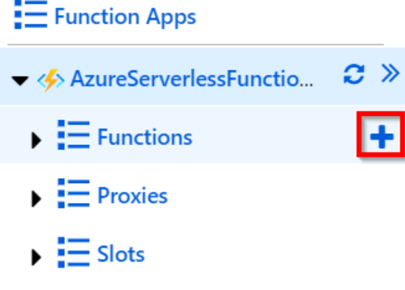


Figure 1.2: Adding a new function

3. You'll see the **Azure Functions for .NET - getting started** page, which prompts you to choose the type of tools based on your preference. For the initial few chapters, we'll use the **In-portal** option, which can quickly create Azure Functions right from the portal without making use of any tools. However, in the coming chapters, we'll make use of Visual Studio and Azure Functions Core Tools to create these functions:

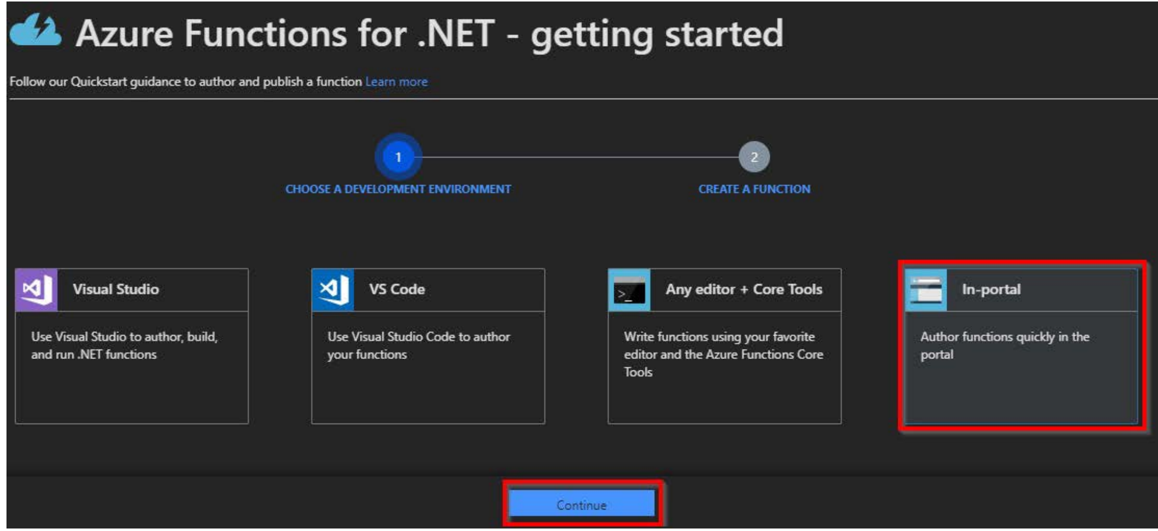


Figure 1.3: Choosing the development environment

4. In the next step, select **More templates...** and click on **Finish and view templates**, as shown in *Figure 1.4*:

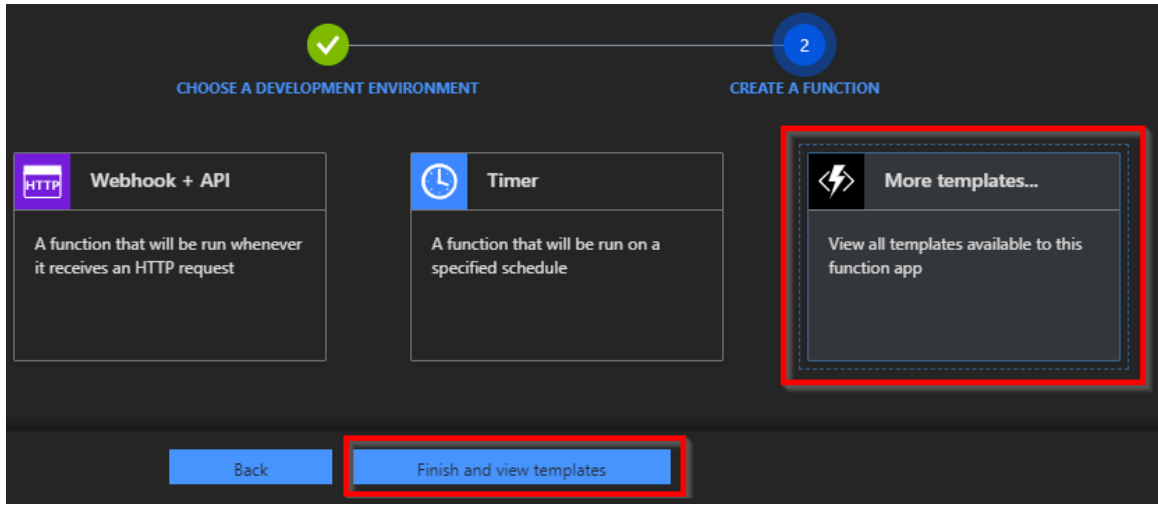


Figure 1.4: Choosing More templates... and clicking Finish and view templates

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

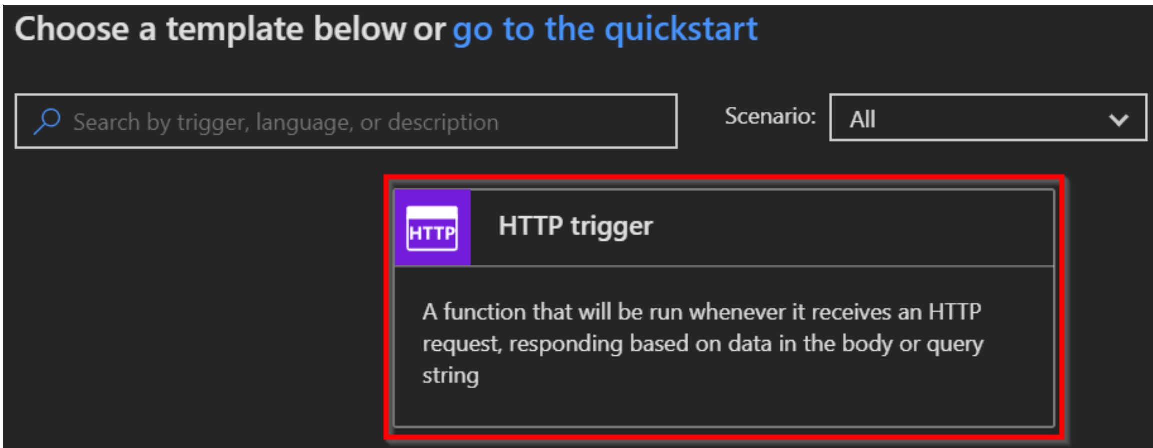


Figure 1.5: The HTTP trigger template

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 menu, choose the **Anonymous** option. You'll learn more about all the authorization levels in *Chapter 9*, *Configuring security for Azure Functions*:

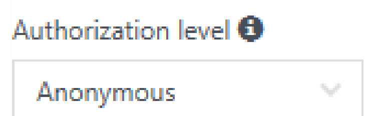


Figure 1.6: Selecting the authorization level

8. Click on the **Create** button to create the HTTP trigger function.
9. Along with the function, all the required code and configuration files will be created automatically and the **`run.csx`** file with editable code will get opened. Remove the default code and replace it with the following code. In the following example, we'll add two parameters (**`firstname`** and **`lastname`**), which will be displayed in the output as a result of triggering the HTTP trigger:

```
#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)
#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 testing the `RegisterUser` function using the test console. Click on the **Test** tab to open the test console:

Figure 1.7: Testing the HTTP trigger

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

Figure 1.8: Testing the HTTP trigger with input data

13. Make sure that you select **POST** in the **HTTP method** drop-down box.
14. After reviewing the input parameters, click on the **Run** button available at the bottom of the test console:

Figure 1.9: HTTP trigger execution and output

15. If the input request workload is passed correctly with all the required parameters, you'll see **Status: 200 OK**, and the output in the output window will be as shown in *Figure* 1.9.
16. Let's discuss how it works next.

How it works...

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

> **Note**
>
> For the sake of simplicity, validation of the input parameters is not executed in this exercise. Be sure to validate all input parameters in applications running in a production environment.

See also

- The *Enabling authorization for function apps* recipe in *Chapter 9*, *Configuring security for Azure Functions*.

In the next recipe, you'll learn about persisting employee details.

Persisting employee details using Azure Table Storage output bindings

In the previous recipe, you created an HTTP trigger and accepted input parameters. Now, let's learn how to store input data in a persistent medium. Azure Functions supports many ways to store data. For this example, we'll store data in Azure Table storage, a NoSQL key-value persistent medium for storing semi-structured data. Learn more about it at https://azure.microsoft.com/services/storage/tables/.

The primary key of an 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 row.

Getting ready

This recipe showcases the ease of integrating an HTTP trigger and the Azure Table storage service using output bindings. The Azure HTTP trigger function receives data from multiple sources and stores user profile data in a storage table named **`tblUserProfile`**. We'll follow the prerequisites listed here:

- For this recipe, we'll make use of the HTTP trigger that was created in the previous recipe.
- We'll also be using Azure Storage Explorer, a tool that helps us to work with data stored in an Azure storage account. Download it from http://storageexplorer.com/.
- Learn more about how to connect to a storage account using Azure Storage Explorer at https://docs.microsoft.com/azure/vs-azure-tools-storage-manage-with-storage-explorer.
- Learn more about output bindings at https://docs.microsoft.com/azure/azure-functions/functions-triggers-bindings.

Let's get started.

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**, and then click on the **Select** button:

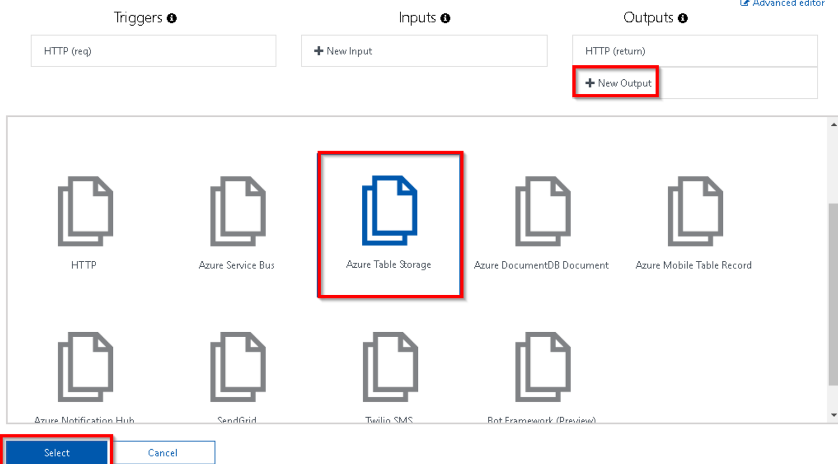


Figure 1.10: New output bindings

3. If you are prompted to install the bindings, click on **Install**; this will take a few minutes. Once the bindings are installed, choose the following settings for the **Azure Table Storage output** bindings:

   Table parameter name: This is the name of the parameter that will be used 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 the **Storage account connection** string is not displayed, click on **new** (as shown in *Figure 1.11*) to create a new one or to choose an existing storage account.

   The Azure Table storage output bindings should be as shown in *Figure 1.11*:

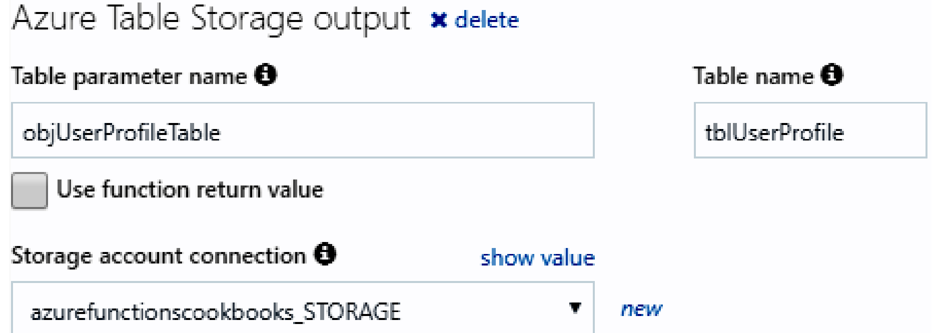
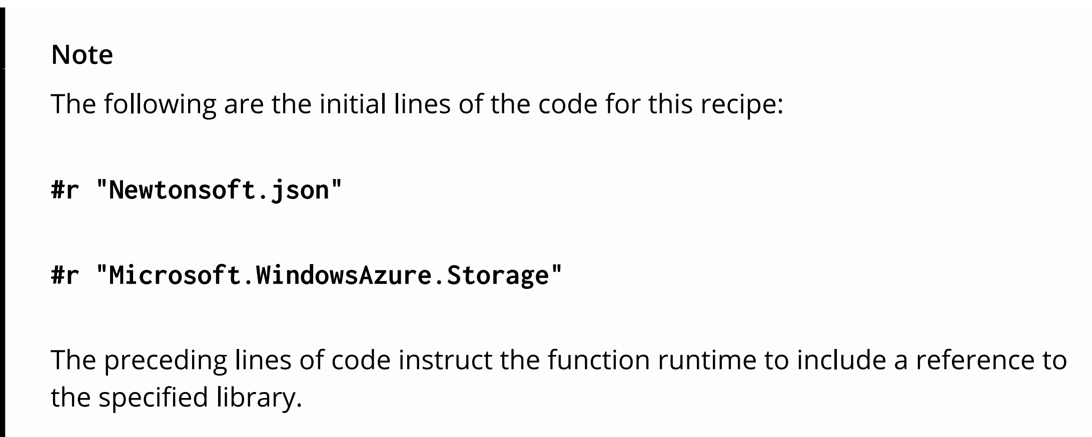


Figure 1.11: Azure Table Storage output bindings settings

4. Click on **Save** to save the changes.
5. Navigate to the code editor by clicking on the function name.



Note

The following are the initial lines of the code for this recipe:

```
#r "Newtonsoft.json"

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

The preceding lines of code instruct the function runtime to include a reference to the specified library.

Paste the following code into the editor. The code will accept the input passed by the end user and save it in Table storage; click **Save**:

```
#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. Execute the function by clicking on the **Run** button of the **Test** tab by passing the `firstname` and `lastname` parameters to the **Request body**.
7. If there are no errors, you'll get a **Status: 200 OK** message as the output. Navigate to **Azure Storage Explorer** and view the Table storage to see whether a table named `tblUserProfile` was created successfully:

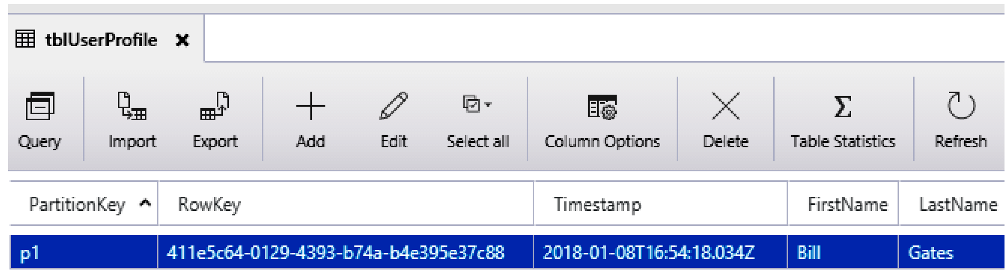


Figure 1.12: Viewing data in Storage Explorer

How it works...

Azure Functions allows us to easily integrate with other Azure services just by adding an output binding to a trigger. In this example, we have integrated an HTTP trigger with the Azure Table storage binding. We also configured an Azure storage account by providing a storage connection string and the Azure Table storage 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 to handle the Table storage, named `objUserProfileTable`, of the `CloudTable` type, to the `Run` method. We can perform all the operations on Azure Table storage using `objUserProfileTable`.

> **Note**
>
> The input parameters are not validated in the code sample. However, in a production environment, it's important to validate them before storing them in any kind of persistent medium.

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

> **Note**
>
> Learn more about handling operations on the Azure Table storage service at https://docs.microsoft.com/azure/cosmos-db/tutorial-develop-table-dotnet.

Understanding storage connections

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

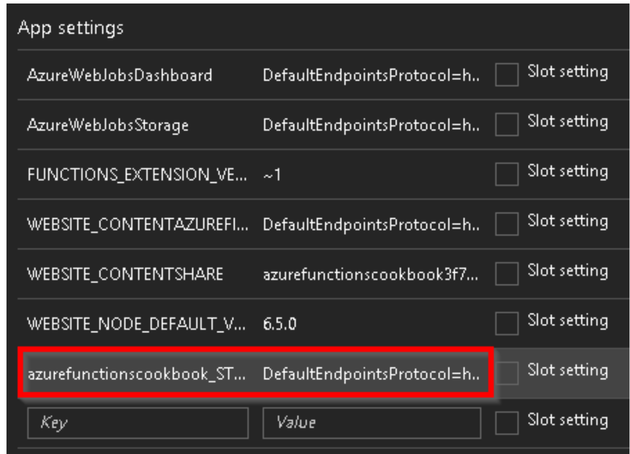


Figure 1.13: Application settings in the configuration blade

Navigate to **App settings** by clicking on the **Configuration** menu available in the **General Settings** section of the **Platform features** tab:

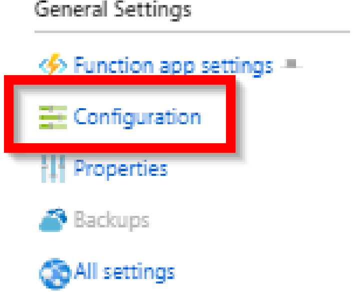


Figure 1.14: Configuration blade

You learned how to save data quickly using Azure Table storage bindings. In the next recipe, you'll learn how to save profile picture paths to queues.

Saving profile picture paths to queues using queue output bindings

The previous recipe highlighted 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 (and the next), you'll learn about the process of extracting the URL of an image and saving it in the blob container of an Azure storage account.

While the `ProfilePicUrl` input parameter can be used to download the picture from the internet, in the previous recipe, *Persisting employee details using Azure Table storage output bindings*, this was not feasible due to the time required to process the large size of the image, which might hinder the performance of the overall application. For this reason, it is faster to grab the URL of the profile picture and store it in a queue, which can be processed later before storing it in the blob.

Getting ready

We'll be updating the code of the `RegisterUser` function that was 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 queue name to `userprofileimagesqueue`.

   Storage account connection: It is important to select the right storage account in the **Storage account connection** field.
4. Click on **Save** to 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 shown in the following code:

```
public static async Task<IActionResult> Run( HttpRequest req,
CloudTable objUserProfileTable, IAsyncCollector<string> 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, you 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 **ProfilePicUrl** to 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 clicking on the **Run** button in the **Test** tab of the Azure Functions code editor window. Replace **"URL here"** with the URL of an image that's accessible over the internet; you'll need to make sure that the image URL provided is valid:

```
{
"firstname": "Bill",
"lastname": "Gates",
"ProfilePicUrl":"URL here"
}
```

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

9. *Figure* 1.15 represents the queue message that was created:

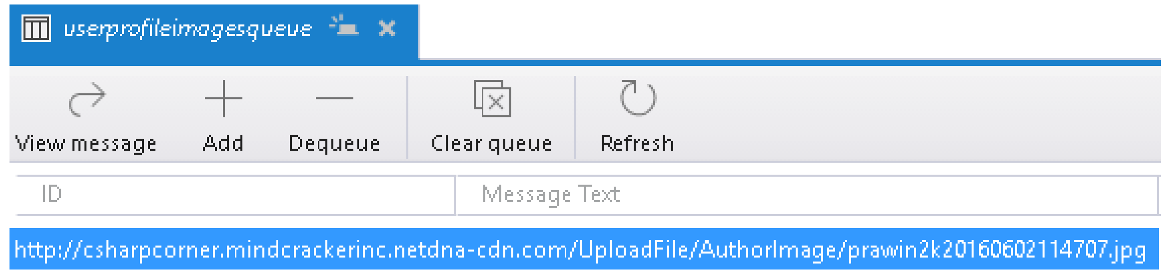


Figure 1.15: Viewing the output in Storage Explorer

How it works...

In this recipe, we added a queue message output binding and made the following changes to our existing code:

- We added a new parameter named **`out string objUserProfileQueueItem`**, which binds the URL of the profile picture as queue message content.
- We used the **`AddAsync`** method of **`IAsyncCollector`** in the **`Run`** method that saves the profile URL to the queue as a queue message.

In this recipe, you learned how to receive a URL of an image and save it in the blob container of an Azure storage account. In the next recipe, we'll store an image in Azure Blob Storage.

Storing images in Azure Blob Storage

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

Getting ready

While the previous recipe focused on creating queue output bindings, this recipe will explain how to grab an image's URL from a queue, create a corresponding byte array, and then write it to a blob.

Note that this recipe is a continuation of the previous recipes. Be sure to implement them first.

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 the function: Provide a meaningful name, such as `CreateProfilePictures`.

   Queue name: Name 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 based on 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 this to `outputBlob`.

   Path: Set this to `userprofileimagecontainer/{rand-guid}`.

   Storage account connection: Choose the storage account for saving the blobs and click on the **Save** button:

Figure 1.16: Azure Blob storage output binding settings

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 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, ILogger 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 profile picture paths to queues using queue output bindings* recipe.
10. Navigate to the **Azure Storage Explorer** window and look at the **`userprofileimagecontainer`** blob container. You should find a new blob:

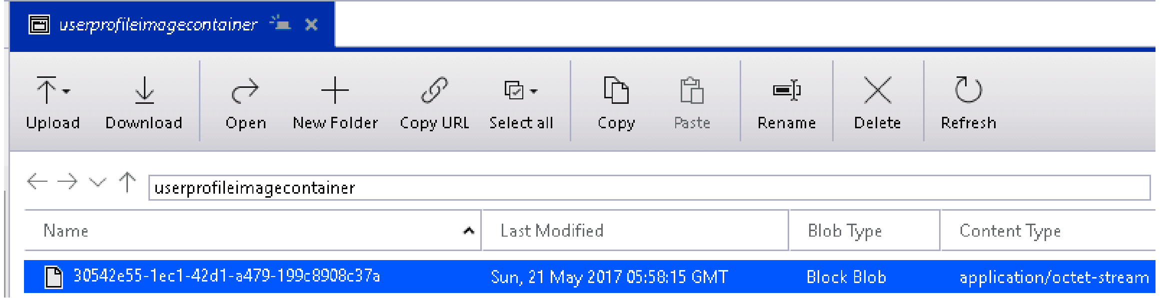


Figure 1.17: Azure Storage Explorer

The image shown in *Figure 1.17* can be viewed through any image viewing tool (such as MS Paint or Internet Explorer).

How it works...

We have created a queue trigger that gets executed when a new message arrives in the queue. Once it finds a new queue message, it reads the message, which is the 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 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.

> **Note**
>
> 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 the container automatically if it doesn't exist.
>
> Queue messages can only be used to store messages up to 64 KB in size. To store messages greater than 64 KB, developers must use Azure Service Bus.

In this recipe, you learned how to invoke an Azure function when a new queue item is added to the Azure Storage Queue service. In the next recipe, you'll learn how to resize an image.

Resizing an image using an ImageResizer trigger

With the recent revolution in high-end smartphone cameras, it has become easy to capture high-quality pictures that tend to have larger sizes. While a good quality picture is beneficial to the consumer, for an application developer or administrator, it proves to be a pain to manage the storage of a popular website, since most platforms recommend that users upload high-quality profile pictures. Given the dilemma, it makes sense to make use of libraries that help us reduce the size of high-quality images while maintaining aspect ratio and quality.

This recipe will focus on implementing the functionality of resizing images without losing quality using one of the NuGet packages called **`SixLabors.ImageSharp`**.

Getting ready

In this recipe, you'll learn how to use a library named `SixLabors` to resize an image to the required dimensions. For the sake of simplicity, we'll resize the image to the following dimensions:

- Medium with 200*200 pixels.
- Small with 100*100 pixels.

How to do it...

1. Create a new Azure function by choosing **Azure Blob Storage Trigger** from the templates.
2. Provide the following details after choosing the template:

   Name the function: Provide a meaningful name, such as `ResizeProfilePictures`.

   Path: Set this to `userprofileimagecontainer/{name}`.

   Storage account connection: Choose the storage account for saving the blobs and click on the **Save** button.
3. Review all the details and click on **Create** to create the new function.
4. Once the function is created, navigate to the **Integrate** tab, click on **New Output**, and choose **Azure Blob Storage**.
5. In the **Azure Blob Storage output** section, provide the following:

   Blob parameter name: Set this to `imageSmall`.

   Path: Set this to `userprofilesmallimagecontainer/{name}`.

   Storage account connection: Choose the storage account for saving the blobs and click on the **Save** button.
6. In the previous step, we added an output binding for creating a small image. In this step, let's create a medium image. Click on **New Output** and choose **Azure Blob Storage**. In the **Azure Blob Storage output** section, provide the following:

   Blob parameter name: Set this to `imageMedium`.

   Path: Set this to `userprofilemediumimagecontainer/{name}`.

   Storage account connection: Choose the storage account for saving the blobs and click on the **Save** button.

7. Now, we need to add the NuGet package references to the **Function App**. In order to add the packages, a file named **`function.proj`** needs to be created, as shown in *Figure 1.18*:

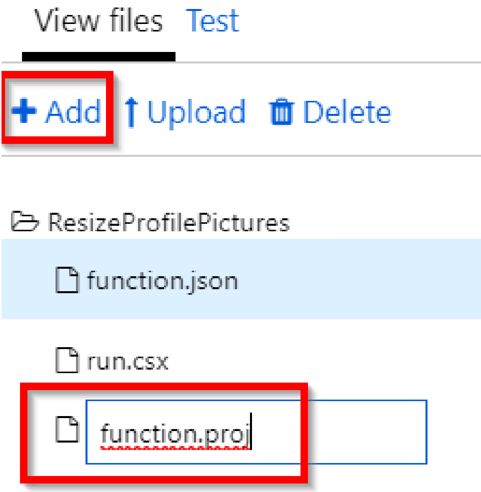


Figure 1.18: Adding a new file

8. Open the **`function.proj`** file, paste the following content to download the libraries related to **`SixLabors.ImageSharp`**, and then click on the **Save** button:

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>netstandard2.0</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="SixLabors.ImageSharp" Version="1.0.0-
beta0007" />
    </ItemGroup>
</Project>
```

9. Once the package reference code has been added in the previous step, you'll be able to view a **Logs** window similar to *Figure 1.19*. Note that the compiler may throw a warning in this step, which can be ignored:

```
2020-02-14T01:50:38.540 [Information] Restoring packages.
2020-02-14T01:50:38.566 [Information] Starting packages restore
2020-02-14T01:50:42.729 [Information] Restoring packages for D:\local\Temp\4e659916-ef5f-46af-94be-
22e622a8480c\function.proj...
2020-02-14T01:50:47.050 [Information] Installing System.Buffers 4.4.0.
2020-02-14T01:50:47.056 [Information] Installing SixLabors.Core 1.0.0-beta0008.
2020-02-14T01:50:47.056 [Information] Installing Microsoft.Net.Compilers.Toolset 3.1.0.
2020-02-14T01:50:47.057 [Information] Installing SixLabors.ImageSharp 1.0.0-beta0007.
2020-02-14T01:51:17.105 [Information] Generating MSBuild file D:\local\Temp\4e659916-ef5f-46af-94be-
22e622a8480c\obj\function.proj.nuget.g.props.
2020-02-14T01:51:17.183 [Information] Generating MSBuild file D:\local\Temp\4e659916-ef5f-46af-94be-
22e622a8480c\obj\function.proj.nuget.g.targets.
2020-02-14T01:51:17.189 [Information] Restore completed in 35.05 sec for D:\local\Temp\4e659916-ef5f-46af-94be-
22e622a8480c\function.proj.
2020-02-14T01:51:17.918 [Information] Packages restored.
```

Figure 1.19: A Logs window

10. Now, let's navigate to the code editor and paste the following code:

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats;
using SixLabors.ImageSharp.PixelFormats;
using SixLabors.ImageSharp.Processing;
public static void Run(Stream myBlob, string name,Stream imageSmall,Stream 
imageMedium, ILogger log)
{
           try
            {
                IImageFormat format;

                using (Image<Rgba32> input = Image.Load<Rgba32>(myBlob, 
out format))
                {
                    ResizeImageAndSave(input, imageSmall, ImageSize.Small, 
format);
                }

                myBlob.Position = 0;
                using (Image<Rgba32> input = Image.Load<Rgba32>(myBlob, 
out format))
                {
                    ResizeImageAndSave(input, imageMedium, ImageSize.
Medium, format);
                }
```

```
            }
            catch (Exception e)
            {
                log.LogError(e, $"unable to process the blob");
            }

    }
        public static void ResizeImageAndSave(Image<Rgba32> input, Stream 
output, ImageSize size, IImageFormat format)
        {
            var dimensions = imageDimensionsTable[size];

            input.Mutate(x => x.Resize(width: dimensions.Item1, height: 
dimensions.Item2));
            input.Save(output, format);
        }

        public enum ImageSize { ExtraSmall, Small, Medium }

        private static Dictionary<ImageSize, (int, int)> 
imageDimensionsTable = new Dictionary<ImageSize, (int, int)>()
        {
            { ImageSize.Small,      (100, 100) },
            { ImageSize.Medium,     (200, 200) }
        };
```

11. Now, navigate to the **RegisterUser** function and run it again. If everything is configured properly, the new containers should be created, as shown in *Figure 1.20*:

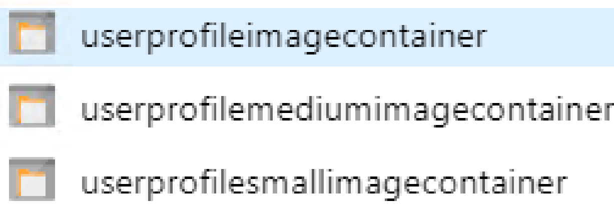


Figure 1.20: Azure Storage Explorer

12. Review the new images created in the new containers with the proper sizes, as shown in *Figure 1.21*:

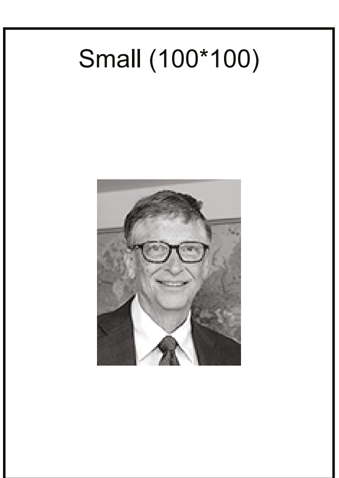


Figure 1.21: Displaying the output

How it works...

Figure 1.22 shows how the execution of the functions is triggered like a chain:

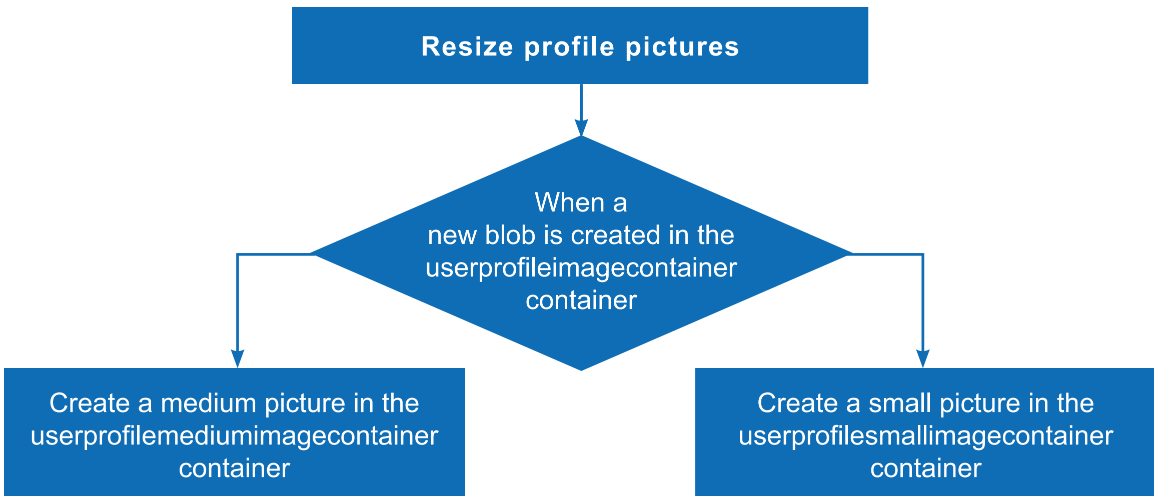
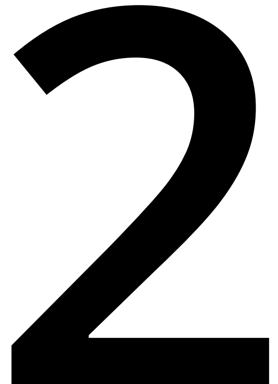


Figure 1.22: Illustration of the execution of the functions

We have created a new blob trigger function sample named **`ResizeProfilePictures`**, which will be triggered immediately after the original blob (image) is uploaded. Whenever a new blob is created in the **`userprofileimagecontainer`** blob, the function will create two resized versions in each of the containers—**`userprofilesmallimagecontainer`** and **`userprofilemediumimagecontainer`**—automatically.



Working with notifications using the SendGrid and Twilio services

In this chapter, we will look at the following:

- Sending an email notification using 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

One of the key features required for the smooth running of business applications is to have a reliable communication system between the business and its customers. The communication channel usually operates two-way, by either sending a message to the administrators managing the application or by sending alerts to 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 text messages. In this chapter, we will learn how to leverage both of these communication services to send messages between business administrators and end users.

Figure 2.1 is the architecture that we will be using for utilizing **SendGrid and Twilio Output Bindings** with HTTP and queue triggers:

1. Client applications (web/mobile) make Http Requests, which trigger the **Http Trigger**.
2. The **Http Trigger** creates a message to the **Queue**.
3. A **Queue Trigger** is invoked as soon as a message arrives at the queue.
4. **Send Grid Output Bindings** is executed.
5. An **Email** is sent to the end user.
6. **Twilio Output Bindings** is executed.
7. An **SMS** is sent to the end user:

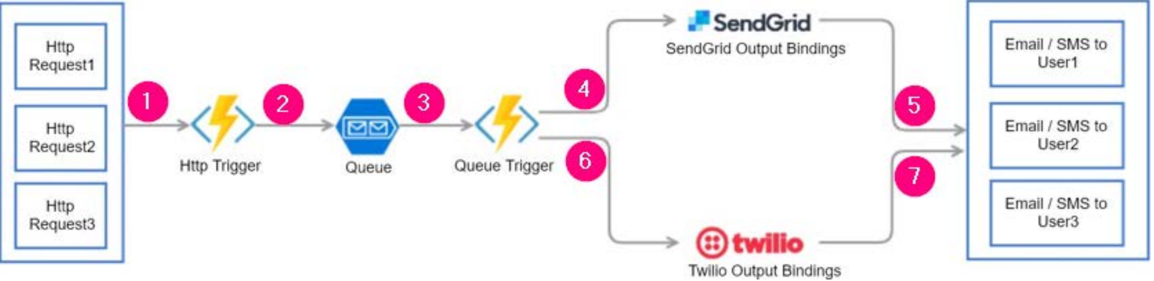


Figure 2.1: Architecture of SendGrid and Twilio output bindings

Sending an email notification using SendGrid service

In this recipe, we will learn how to create a SendGrid output binding and send an email notification, containing static content, to the website administrator. Since our use case involves just one administrator, we will be hard-coding the email address of the administrator in the **To address** field of the **SendGrid output (message) binding**.

Getting ready

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

1. We will create a SendGrid account API key from the Azure portal.
2. We will generate an API key from the SendGrid portal.
3. We will configure the SendGrid API key with the Azure Function app.

Creating a SendGrid account API key from the Azure portal

In this section, we'll be creating a **Send** service and also generate the API by performing the following steps:

1. Navigate to the Azure portal and create a **SendGrid Email Delivery** account by searching for it in the marketplace, as shown in *Figure 2.2*:

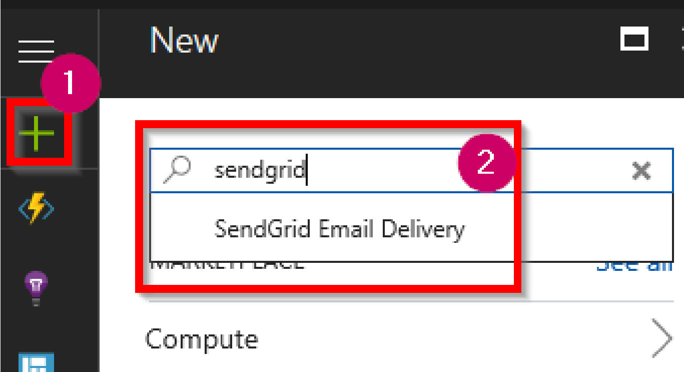


Figure 2.2: Searching for SendGrid Email Delivery in the marketplace

2. In the **SendGrid Email Delivery** blade, click on the **Create** button to navigate to **Create SendGrid Account**. Select **Free** in the **Pricing Tier** options, provide all the other details, and then click on the **Review + Create** button to review this information. Finally, click on the **Create** button, as shown in *Figure 2.3*:

Create SendGrid Account
SendGrid
Basics Tags Review + Create
Configure your SendGrid Account to deliver customer communication that drives engagement and growth using the cloud. Learn more
Project details
Subscription * Visual Studio Enterprise – MPN
Resource group * AzureServerlessFunctionCookbook
Create new
Location * (US) Central US
Account details
Name * azurecookbook
Password * ••••••••••••
Confirm password * ••••••••••••
Pricing Tier Free
25,000 email/month
Change plan
Contact details
Review + Create Previous Next: Tags >

Figure 2.3: Creating a SendGrid email delivery account

> **Note**
>
> At the time of writing, the SendGrid free account allows you to send 25,000 free emails per month. If you would like to send more emails, then you can review and change the pricing plans based on your needs.

3. Make a note of the password entered in the previous step. Once the account is created successfully, navigate to **SendGrid Account**. You can use the search box available at the top.

SendGrid is not a native Azure service. So, we need to navigate to the SendGrid website to generate the API key. Let's learn how to do that next.

Generating credentials and the API key from the SendGrid portal

Let's generate the API key by performing the following steps:

1. In order to utilize the SendGrid account in the Azure Functions runtime, we need to provide the SendGrid credentials as input for Azure Functions. You can generate those details 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 *Figure* 2.4:

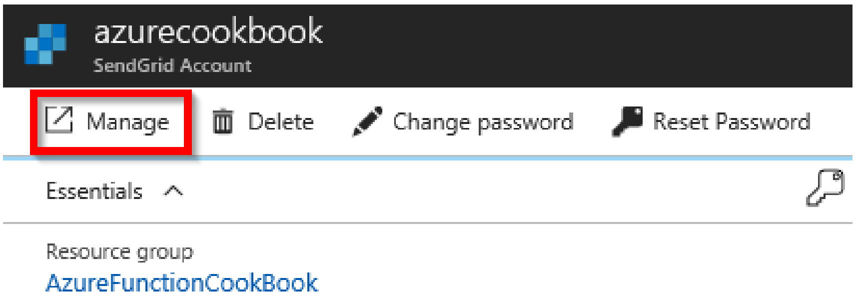


Figure 2.4: Acquiring SendGrid credentials in the Manage blade

2. In the SendGrid portal, click on the **Account Details** menu under **Settings** and copy the username, as shown in *Figure* 2.5:

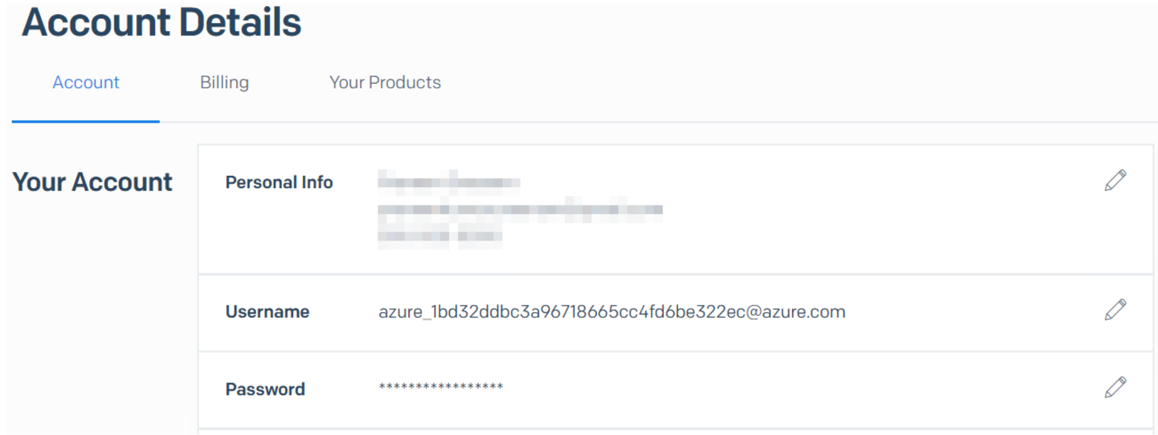


Figure 2.5: Copying the SendGrid credentials

3. In the SendGrid portal, the next step is to generate the API keys. Now, click on **API Keys** under the **Settings** section of the left-hand side menu, as shown in *Figure 2.6*:

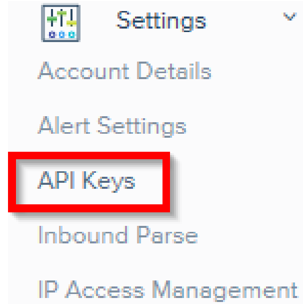


Figure 2.6: Generating API keys

4. On the **API Keys** page, click on **Create API Key**, as shown in *Figure 2.7*:

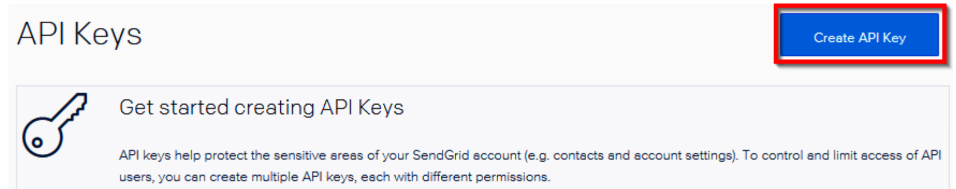


Figure 2.7: Creating API keys

5. In the **Create API Key** pop-up window, provide a name and choose **API Key Permissions**, and then click on the **Create & View** button.
6. 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 *Figure 2.8*:

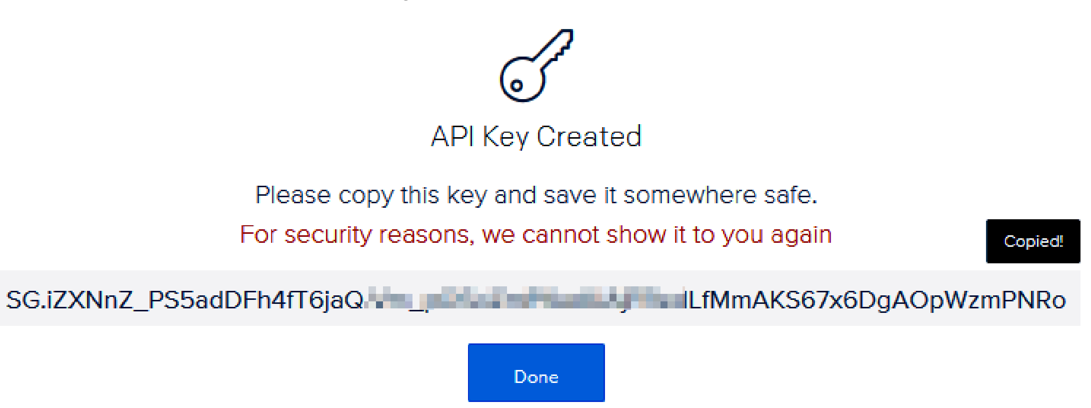


Figure 2.8: Copying the API key

Having copied the API key, we'll now configure it.

Configuring the SendGrid API key with the Azure Function app

Let's now configure the SendGrid API key by performing the following steps:

1. Create a new **App settings** configuration in the Azure Function app by navigating to the **Configuration** blade, under the **Platform features** section of the function app, as shown in *Figure 2.9*:

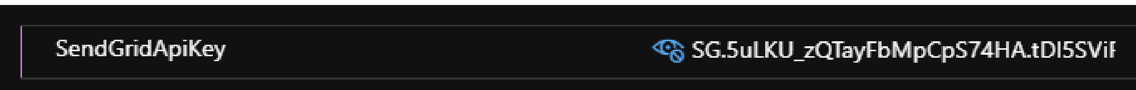


Figure 2.9: Creating a new app setting configuration

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 tasks:

1. We will create a storage queue binding to the HTTP trigger.
2. We will create a queue trigger to process the message of the HTTP trigger.
3. We will create a SendGrid output binding to the queue trigger.

Creating a storage queue binding to the HTTP trigger

Let's create the queue bindings now. This will allow us to create a message to be added to the queue.

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 the **Select** button to add the binding and provide the values shown in *Figure 2.10*, and then click on the **Save** button. Please make a note of the **Queue name** (in this case, **`notificationqueue`**), which will be used in a moment:

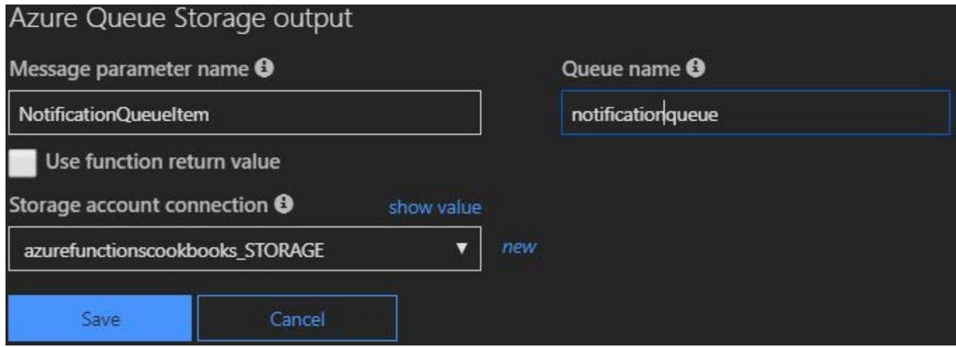


Figure 2.10: Adding a new output binding

3. Navigate to the **`Run`** method of the **`RegisterUser`** function and make the following highlighted changes. You added another queue output binding and added an empty message to trigger the queue trigger function. For now, you have not added a message to the queue. We will make changes to the **`NotificationQueueItem.AddAsync("");`** method in the *Sending an email notification dynamically to the end user* 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");
}
```

Let's now proceed to create the queue trigger.

Creating a queue trigger to process the message of the HTTP trigger

In this section, you'll learn how to create a queue trigger by performing the following steps:

1. Create an **Azure Queue Storage Trigger** by choosing the template shown in *Figure 2.11*:

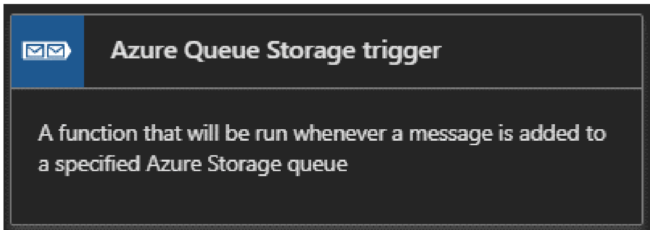


Figure 2.11: Creating an Azure Queue Storage Trigger

2. In the next step, provide the name of the queue trigger and provide the name of the queue that needs to be monitored for sending the notifications. Once you have provided all the details, click on the **Create** button to create the function:

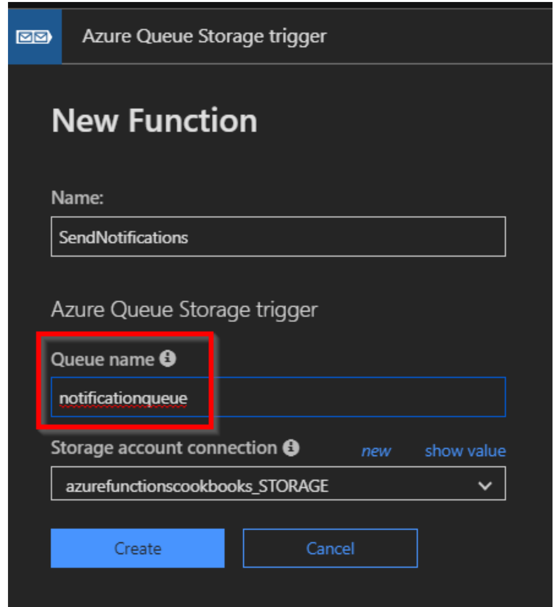


Figure 2.12: Creating a new function

3. After creating the queue trigger function, run the `RegisterUser` function to see whether the queue trigger is being 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 similar to *Figure 2.13*:

Logs Console Reconnect Copy logs Pause Clear Expand

```
2020-02-20T15:55:09.905 [Information] Executing 'Functions.SendNotifications' (Reason='This function was programmatically called via the host APIs.', Id=1177ce1e-91c6-4231-b8f8-2eb6fde5ff78)
2020-02-20T15:55:09.905 [Information] C# Queue trigger function processed: sample queue data
2020-02-20T15:55:09.906 [Information] Executed 'Functions.SendNotifications' (Succeeded, Id=1177ce1e-91c6-4231-b8f8-2eb6fde5ff78)
2020-02-20T15:56:45.092 [Information] Executing 'Functions.SendNotifications' (Reason='New queue message detected on 'notificationqueue'.', Id=83db935c-20e0-4313-bd9d-ea6a4e55ea36)
2020-02-20T15:56:45.092 [Information] Trigger Details: MessageId: 26d8dcf2-0a6a-4023-a257-9177610cba98, DequeueCount: 1, InsertionTime: 2/20/2020 3:56:43 PM +00:00
2020-02-20T15:56:45.093 [Information] C# Queue trigger function processed:
2020-02-20T15:56:45.093 [Information] Executed 'Functions.SendNotifications' (Succeeded, Id=83db935c-20e0-4313-bd9d-ea6a4e55ea36)
```

Figure 2.13: Invoking the queue trigger by running the RegisterUser function

Once we have ensured that the queue trigger is working as expected, we need to create the SendGrid bindings to send the email in the following section.

Creating a SendGrid output binding to the queue trigger

Perform the following steps to create the **SendGrid output** bindings to send the email:

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 (these are packages related to SendGrid). Click on the **Install** button to install the extensions if prompted, as shown in *Figure 2.14*. It might take a few minutes to install the extensions:

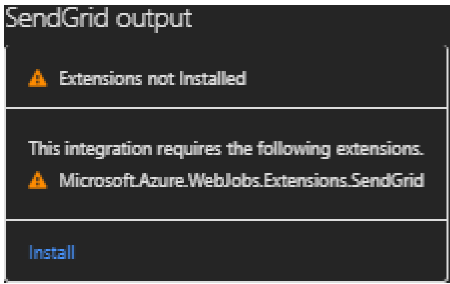


Figure 2.14: Notification to install extensions in the SendGrid bindings

> **Note**
>
> If there is no prompt notification, please delete the output binding and recreate it. You could also install the extensions manually by going through the instructions mentioned in https://docs.microsoft.com/azure/azure-functions/install-update-binding-extensions-manual.

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 the **Configuration** blade for storing the **SendGrid 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. This 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 body of the email.

  This is how the **SendGrid output** (message) binding should appear after providing all the fields:

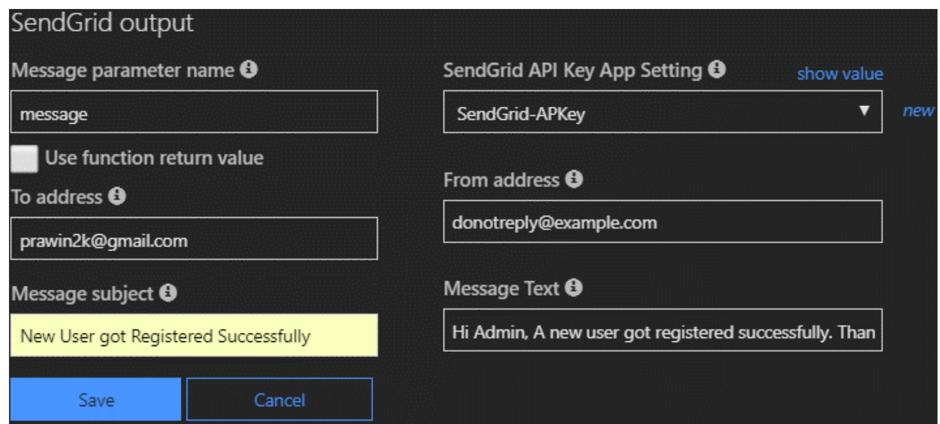


Figure 2.15: Adding details in the SendGrid output (message) binding

5. Once you review the values, click on **Save** to save the changes.
6. Navigate to the `Run` method of the `SendNotifications` function 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, *Sending an email notification dynamically to the end user*.

7. 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();
}
```

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

```
{
"firstname": "Bill",
"lastname": "Gates",
"ProfilePicUrl":"URL Here"
}
```

How it works...

The aim of this recipe is to send an email notification 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 hard-coding 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 the **App settings** to invoke the required APIs of the SendGrid library in order to send the emails.

Sending an email notification dynamically to the end user

In the previous recipe, we hard-coded 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 steps are configured properly:

- The SendGrid account is created and an API key is generated in the SendGrid portal.
- An App settings configuration is created in the configuration 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`**

Accepting the new email parameter in the RegisterUser function

Let's make changes to the **`RegisterUser`** function to accept the **`email`** parameter by performing the following steps:

1. Navigate to the **`RegisterUser`** function, in the **`run.csx`** file, and 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;
...
...
email = inputJson.email;
...
...
UserProfile objUserProfile = new UserProfile(firstname, lastname, string
profilePicUrl,email);
...
```

```
...
await NotificationQueueItem.AddAsync(JsonConvert.
SerializeObject(objUserProfile))
;
```

2. Update the following 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;}
}
```

Let's now move on to retrieve the user profile information.

Retrieving the UserProfile information in the SendNotifications trigger

In this section, we will perform the following steps to retrieve the user information:

1. Navigate to the **SendNotifications** function, in the **run.csx** file, and add the **NewtonSoft.Json** reference and also the namespace.
2. The queue trigger will receive the input in the form of a JSON string. We will use the **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 **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. After making all of the aforementioned highlighted changes to the **SendNotifications** function, click **Save**. In order to test this, you need to execute the **RegisterUser** function. Let's run a test by adding a new input field email to the test request payload of the **RegisterUser** function, 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:

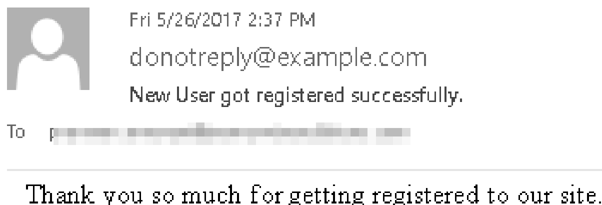


Figure 2.16: Email notification of successful registration

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 can be customized dynamically based on the requirements.

We can also clear the fields in the **SendGrid output** bindings, as shown in *Figure 2.17*:

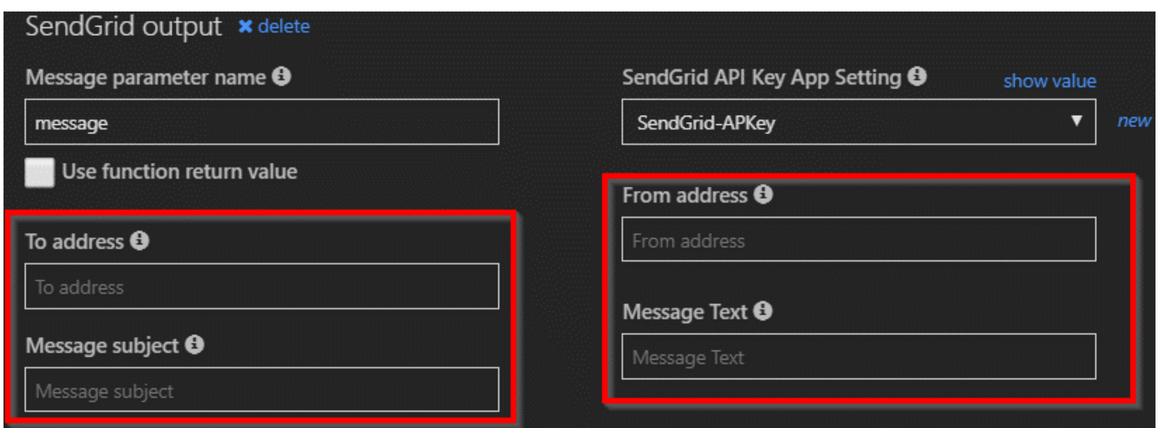


Figure 2.17: Clearing the fields in the SendGrid output bindings

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

There's more...

You can also add 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.");
```

Figure 2.18 shows the email, with my name in bold:

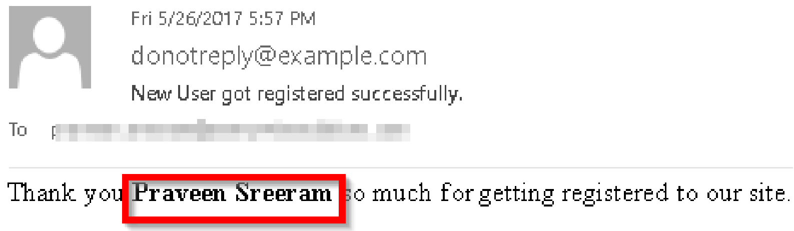


Figure 2.18: Customizing the email notification

In this recipe, you have learned how to send an email notification dynamically to the end user. Let's now move on to the next recipe.

Implementing email logging in Azure Blob Storage

Most of the business applications for automated emails are likely to involve sending emails containing various notifications and alerts to the end user. At times, it is not uncommon for users to not receive any emails, even though we, as developers, don't see any error in the application while sending such notification alerts.

There might be multiple reasons why such users might not have received the email. Each of the email service providers has different spam filters that can block the emails from the end user's inbox. As these emails may have important information to convey, 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 redundancy for the data stored in Table storage. You will also learn how to store 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**. If prompted, you will have to install **Storage Extensions**, so please install the extensions to continue forward.

2. Provide the requisite parameters in the **Azure Blob Storage output** section, as shown in *Figure 2.19*. Note the `.log` extension in the **Path** field:

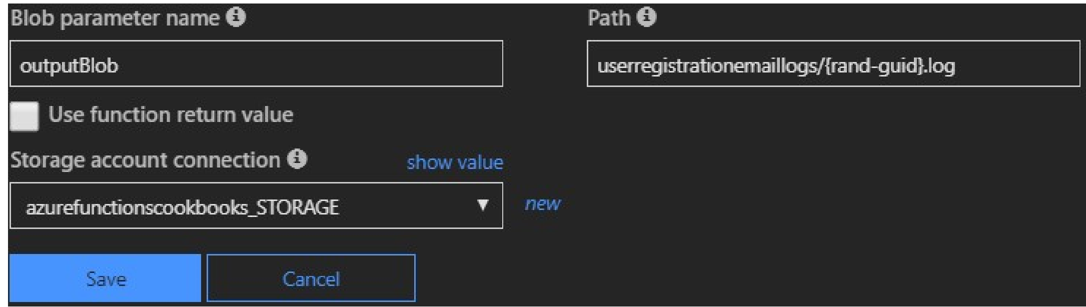


Figure 2.19: Adding details in the Azure Blob Storage output

3. Navigate to the code editor of the **run.csx** file of the **SendNotifications** function and make the following changes:

   Add a new parameter, **outputBlob**, of the **TextWriter** type to the **Run** method.

   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.

   Frame the email content by appending the required static text and the input parameters received in the 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. In the **RegisterUser** function, 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**:

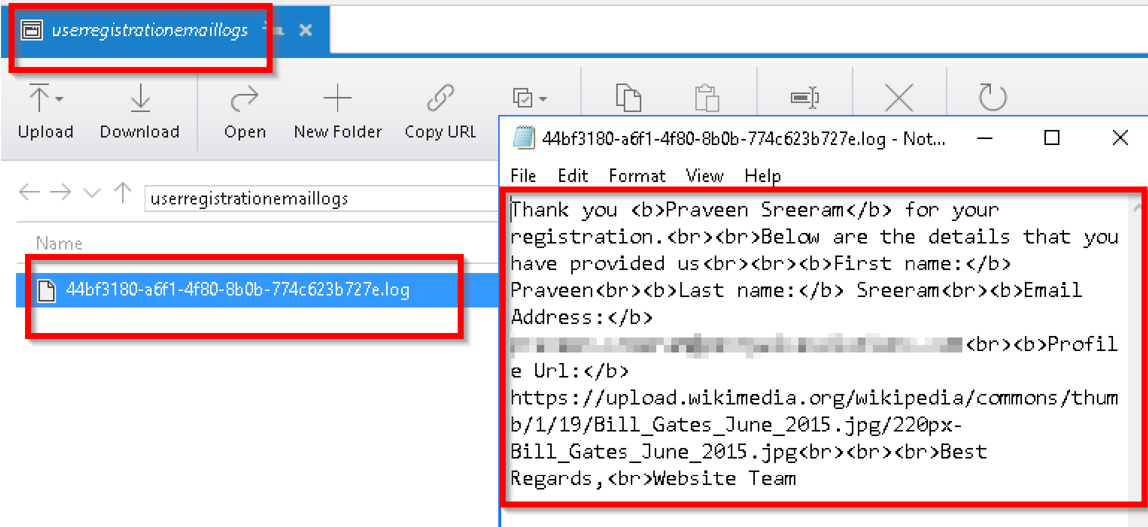


Figure 2.20: Displaying the log file created in userregistrationemaillogs

How it works...

We have created new Azure Blob Storage output bindings. As soon as a new request is received, the email content is created and written to a new **.log** file 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, which we will use as an email attachment for this instance. However, in real-world applications, you might not intend to send log files to the end user.

> **Note**
>
> 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 *Implementing email logging in Azure Blob Storage* recipe. If you are reading this first, make sure to go through the previous recipes of this chapter beforehand.

How to do it...

In this section, 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. The **IBinder** interface helps us in customizing the name of the file.
2. Send this file as an attachment to the email.

Customizing the log file name using the IBinder interface

Perform the following steps:

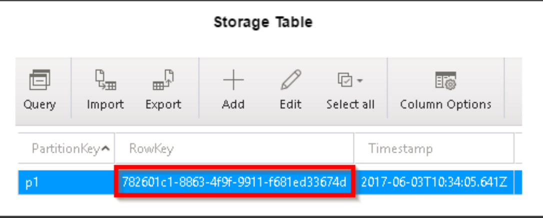
1. Navigate to the `run.csx` file of the `SendNotifications` function.
2. Remove the `TextWriter` object and replace it with the variable binder of the `IBinder` type. The following is the new signature of the `Run` method:

```
#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. Since you 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/
{ inputJson.RowKey}.log")))
{
emailLogBloboutput.WriteLine(emailContent);
}
```

4. In the **RegisterUser** function, 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 Azure Table storage, as shown in *Figure 2.21*:



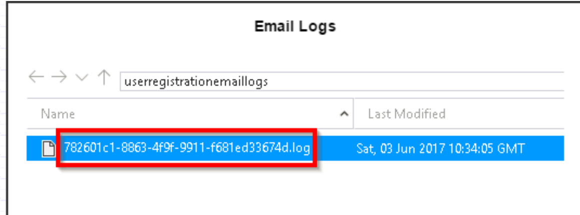


Figure 2.21: Email log file stored in Azure Table storage

Adding an attachment to the email

To add 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(emailContent)),
      "text/plain",
      "attachment",
      "Logs"
      );
```

2. Run a test using the same request payload that we used in the previous recipes.

3. *Figure 2.22* shows the email, along with the attachment:

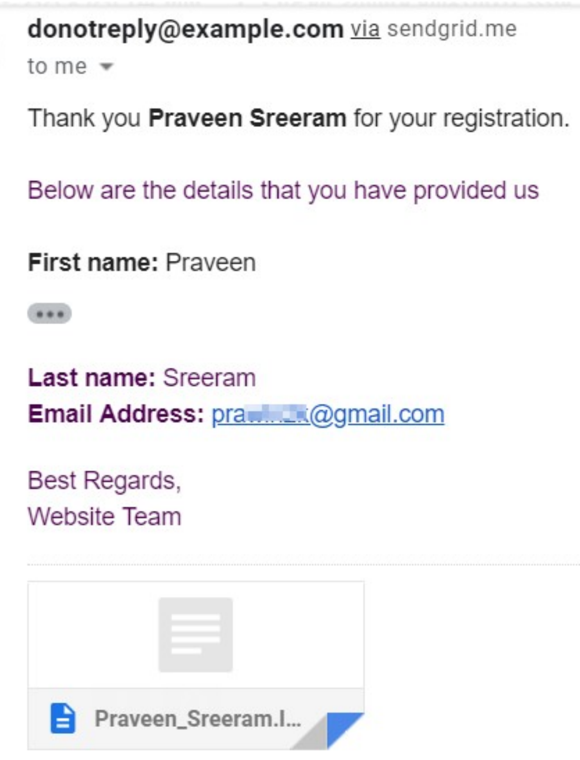


Figure 2.22: Displaying an email along with the attachment

> **Note**
>
> Learn more about the SendGrid API at https://sendgrid.com/docs/API_Reference/api_v3.html.

In this recipe, you have learned how to add an attachment to the email. Let's now move on to the next recipe.

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 text messages, using one of the leading cloud communication platforms, named Twilio.

> **Note**
>
> Twilio is a cloud communication platform-as-a-service platform. Twilio allows software developers to programmatically make and receive phone calls, send and receive text messages, and perform other communication functions using its web service APIs. Learn more about Twilio at https://www.twilio.com/.

Getting ready

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

1. Create a trial Twilio account at https://www.twilio.com/try-twilio.
2. Following the successful creation of the account, grab the **ACCOUNT SID** and **AUTH TOKEN** from the Twilio **Dashboard** and save it for future reference, as shown in *Figure 2.23*. You need to create two App settings in the **Configuration** blade of the function app for both of these settings:

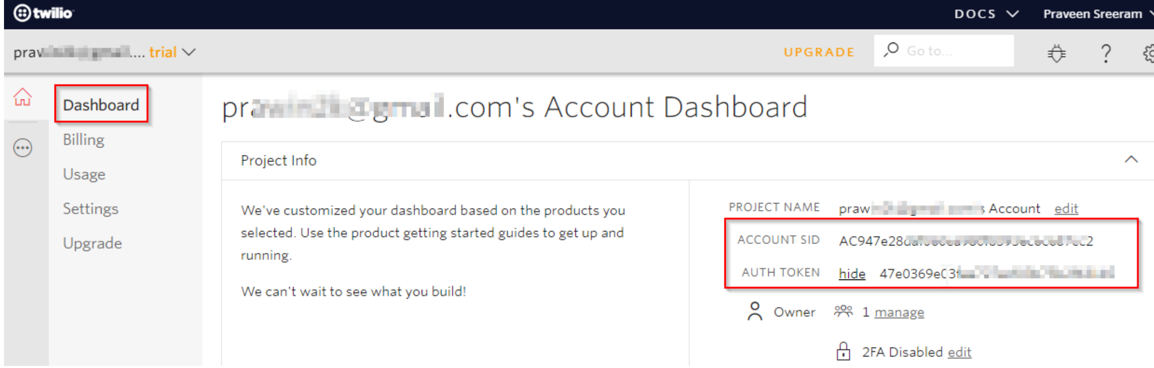


Figure 2.23: Twilio dashboard

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 to send 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.

On the **Get Started with Phone Numbers** page, click on **Get your first Twilio phone number**, as shown in *Figure 2.24*:

Get Started with Phone Numbers

Getting started with Twilio's phone numbers is easy! Search for local, toll-free, or mobile

Get your first Twilio phone number

Figure 2.24: Activating your number using Twilio

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

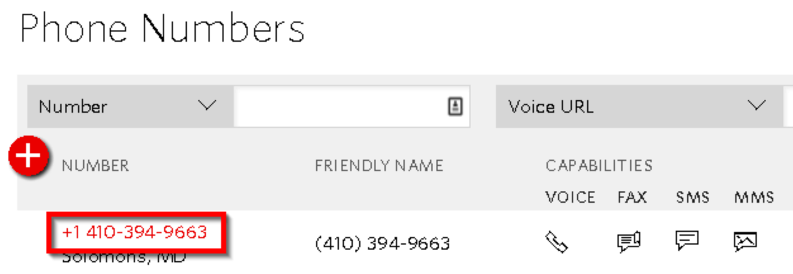


Figure 2.25: Displaying the activated number

5. The final step is to verify a number to which you would like to send an SMS. Click on the **+** icon, as shown in *Figure 2.26*, provide your number, and then click on the **Call Me** button:

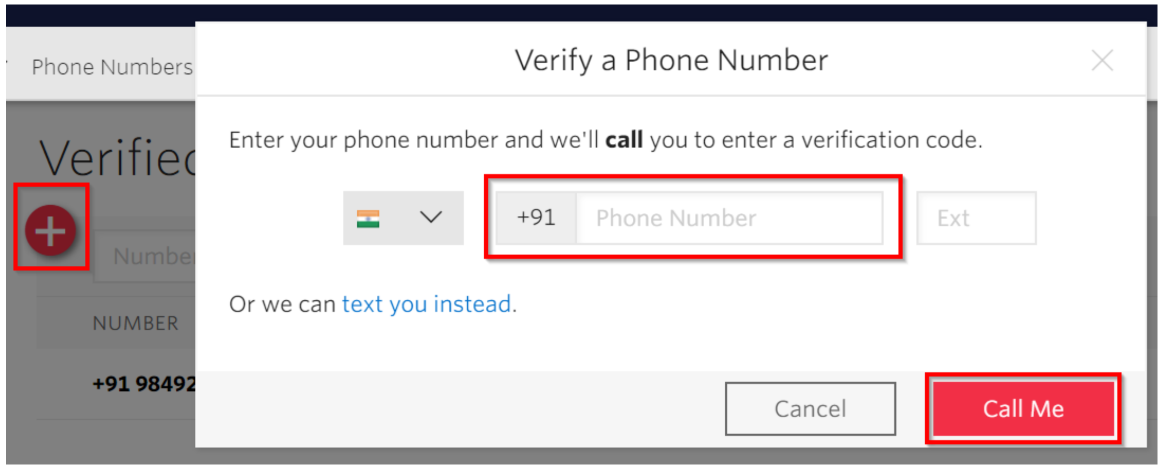


Figure 2.26: Verifying a phone number

6. You can have only one number in your trial account, which can be verified on Twilio's verified page: https://www.twilio.com/console/phone- numbers/verified. *Figure 2.27* shows the list of verified numbers:

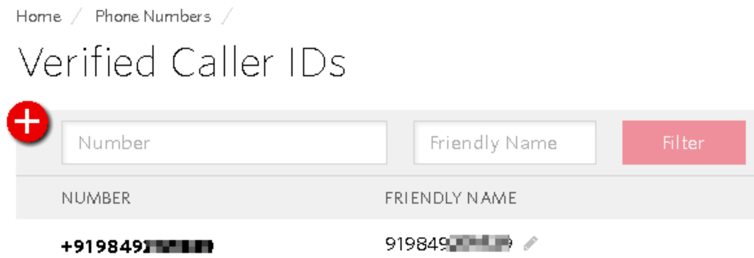


Figure 2.27: Verified caller IDs

How to do it…

Perform the following steps:

1. Navigate to the **Application settings** blade of the function app and add two keys for storing **`TwilioAccountSID`** and **`TwilioAuthToken`**, as shown in *Figure 2.28*:

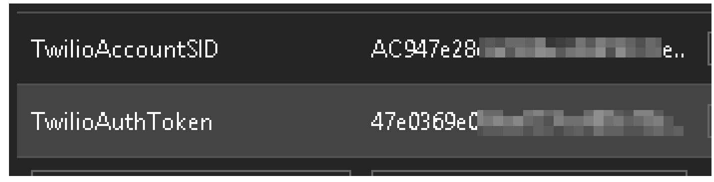


Figure 2.28: Adding two keys for storing TwilioAccountSID and TwilioAuthToken

2. Go to 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. To manually install the extensions, refer to the https://docs.microsoft.com/azure/azure-functions/install-update-binding-extensions-manual article. The **From number** is the one that is generated in the Twilio portal, which we discussed in the *Getting ready* section of this recipe:

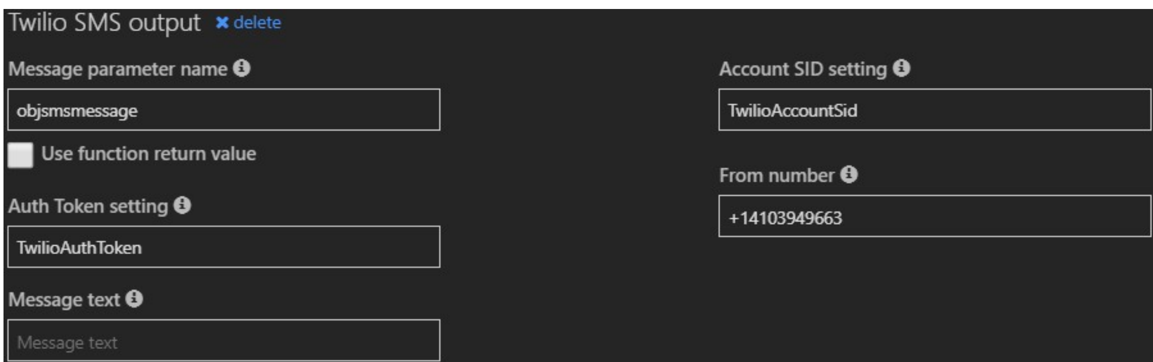


Figure 2.29: Twilio SMS output blade

4. Navigate to the code editor and add the following lines of code. In the following code, I have hard-coded 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(emailContent)),
      "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. *Figure* 2.30 shows the SMS that I have received:

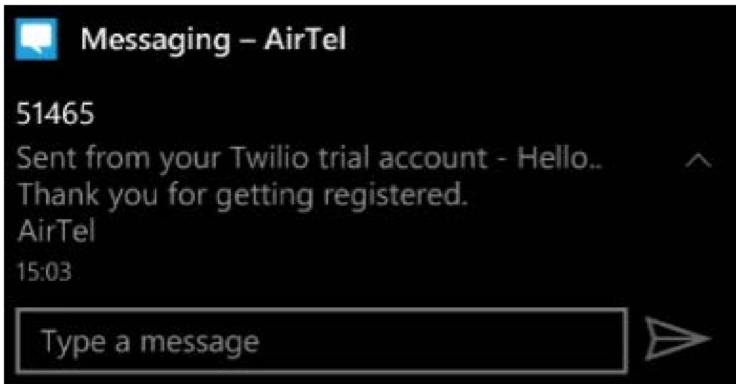


Figure 2.30: SMS received from the Twilio account

How it works...

We have created a new Twilio account and copied the account ID and app key to the App settings of the Azure Function app. The account ID and app key 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 hard-coded 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.

Watch the following video to view a working implementation: https://www.youtube.com/watch?v=ndxQXnoDIj8.

3

Seamless integration of Azure Functions with Azure Services

In this chapter, we'll cover the following recipes:

- Using Cognitive Services for face detection in images
- Monitoring and sending notifications using Logic Apps
- Integrating Logic Apps with serverless functions
- Auditing Cosmos DB data using change feed triggers
- Integrating Azure Functions with Data Factory pipelines

Introduction

One of the main 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, inventing and developing applications from scratch for each business requirement is practically impossible. We would first need to research the existing systems and see whether they fit business requirements. Often, it would not be easy to understand the APIs of the other systems and integrate them, especially when they have been developed by someone else.

Azure provides many connectors that can be leveraged to integrate business applications with other systems pretty easily.

In this chapter, we'll learn how to easily integrate the different services that are available within the Azure ecosystem.

Using Cognitive Services to locate faces in images

Microsoft offers Cognitive Services, which helps developers to leverage AI features in their applications.

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

Cognitive Services apply AI algorithms, so they might not always be accurate. The accuracy returned by Cognitive Services is always between 0 and 1, where 1 means 100% accurate. You can always use the accuracy value returned by Cognitive Services and implement your custom requirements based on the accuracy.

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 configured to access the storage account that is used to upload the blobs.

Creating a new Computer Vision API account

In this section, we'll create a new Computer Vision API account by performing the following steps:

1. Create a function app, if one has not been created already, by choosing **.NET Core** as the runtime stack.
2. Search for `Computer vision` and click on **Create**.

3. The next step is to provide all the details (name, resource group, and subscription) to create a Computer Vision API account. At the time of writing, the Computer Vision API has two pricing tiers. For this recipe, select the free one, **F0**, which allows 20 API calls per minute and is limited to 5,000 calls each month. For your production requirements, you should select the premium instance, **S1**.

Having created the Computer Vision API account, we'll now move on to configure the application settings.

Configuring application settings

In this section, we'll configure the application settings of Azure Functions by performing the following steps:

1. Once the Computer Vision API account has been generated, navigate to the **Keys and Endpoint** blade and copy **KEY 1** into the notepad:

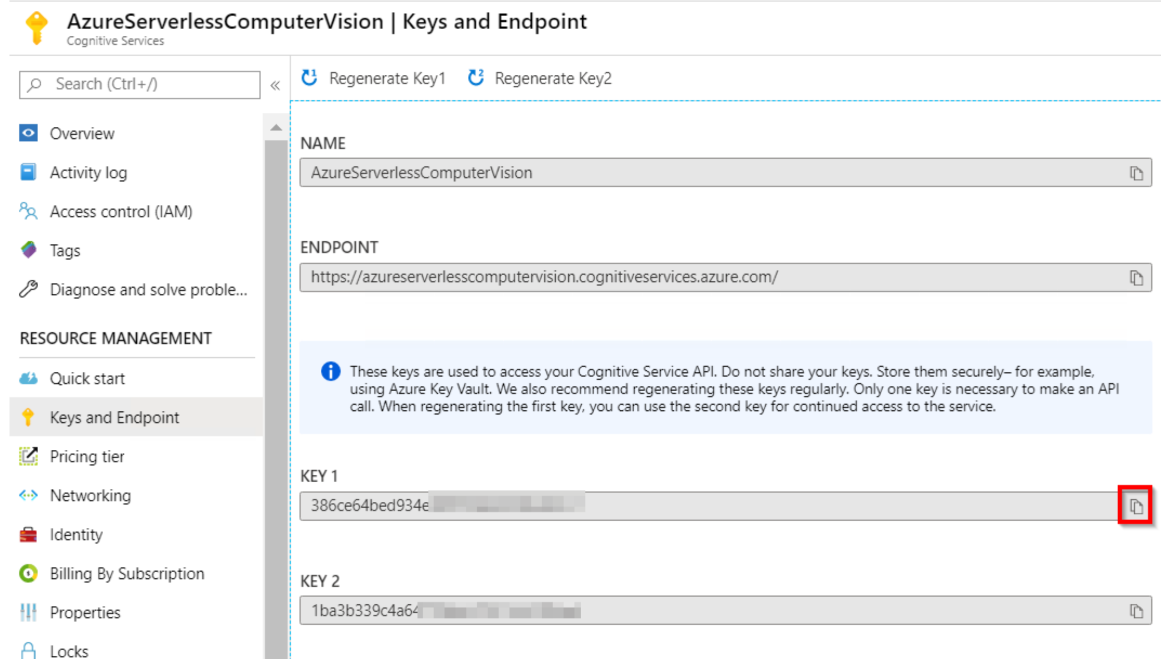


Figure 3.1: Computer Vision keys

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 its 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 this case, it is **East US**. In terms of passing the images to the Cognitive Services API, it is important to ensure that the endpoint of the API starts with the location name. It would be something like this: `https://eastus.api.cognitive.microsoft.com/vision/v1.0/analyze?visualFeatures=Faces&language=en`

Let's now move on to the next section to learn how to develop the Azure function.

How to do it...

In this section, you are going to learn how to leverage Cognitive Services in the blob trigger by performing the following steps:

1. Create a new function using one of the default templates named **Azure Blob Storage Trigger**.
2. Next, 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 *Figure 3.2*) at the end of this section:

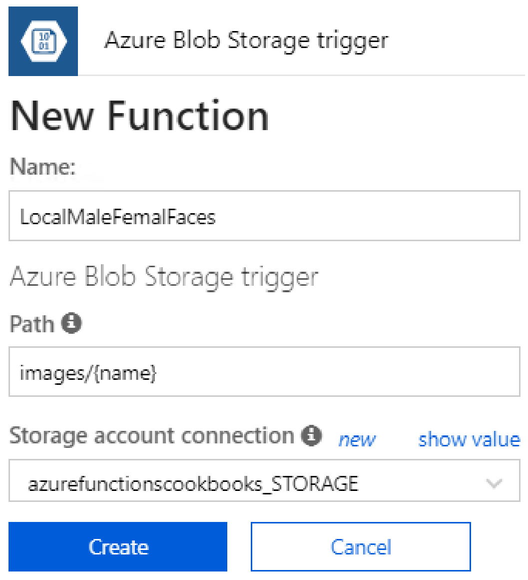


Figure 3.2: Creating an Azure Blob storage trigger

> **Note**
>
> While creating the function, the template creates one blob storage table output binding and allows you to provide a name for the **Table name** parameter. However, you can't assign the name of the parameter while creating the function. You will only be able to change it after it has been created. After reviewing all the details, click on the **Create** button to create the Azure function.

3. Once the function has been created, navigate to the **Integrate** tab, click on **New Output**, choose **Azure Table Storage**, and then click on the **Select** button. Provide the parameter values and then click on the **Save** button, as shown in *Figure* 3.3:

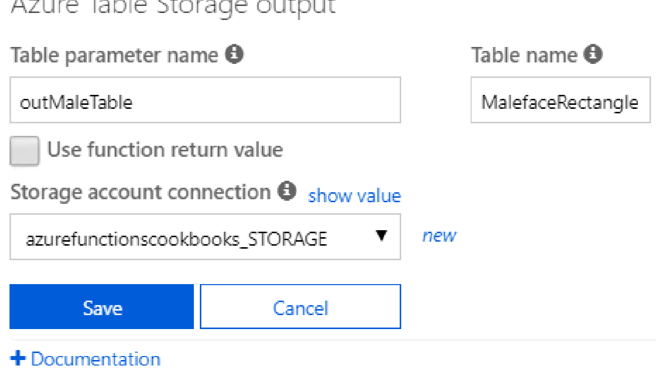


Figure 3.3: Azure Table storage output bindings

4. Let's now 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 then clicking on the **Select** button. This is how it looks after providing the input values:

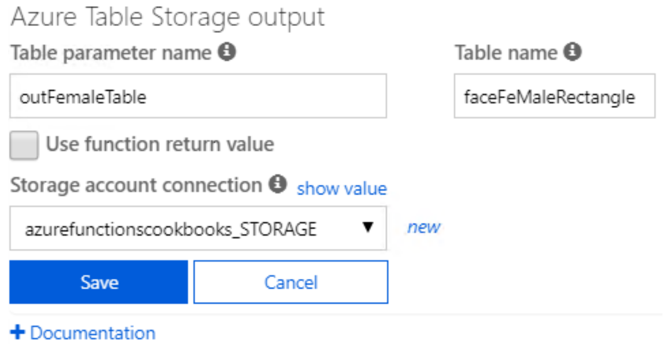


Figure 3.4: Azure Table storage output bindings

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 pertaining to women.
6. Navigate to the code editor of the `Run` method and copy the following code. The code will collect the image stream uploaded to the blob, which will then be passed as an input to Cognitive Services, which will then return some JSON with all the face information, including coordinates and gender details. Once this face information is received, you can store the face coordinates in 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://<location>.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. The code has a condition to check the gender and, based on the gender, it stores the information in the respective table storage.
8. Create a new blob container named **`images`** using Azure Storage Explorer, as shown in *Figure* 3.5:

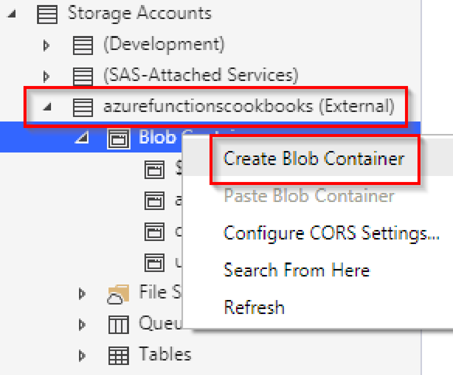


Figure 3.5: Azure Storage—Create Blob Container

9. Let's now upload a picture with male and female faces to the container named **`images`** using Azure Storage Explorer, as shown in *Figure* 3.6:

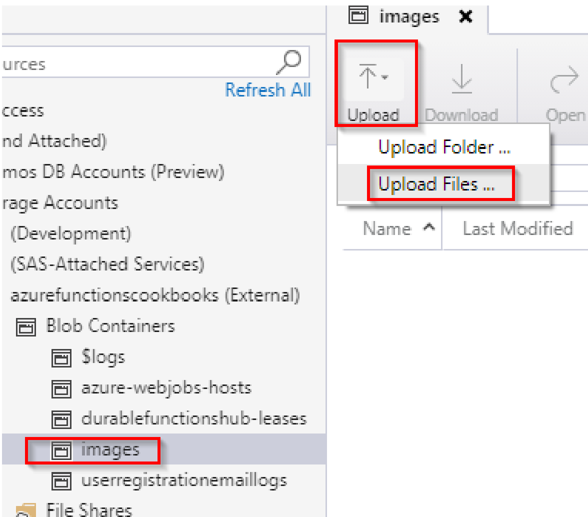


Figure 3.6: Azure Storage—Create Blob Container—Upload Files

10. The function will get 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
}
}
]
}
```

> **Note**
>
> A front-end developer with expertise in HTML5 and canvas-related technologies can even draw squares that locate the faces in images using the information provided by Cognitive Services.

11. The function has also created two different Azure Table storage tables, as shown in *Figure 3.7*:

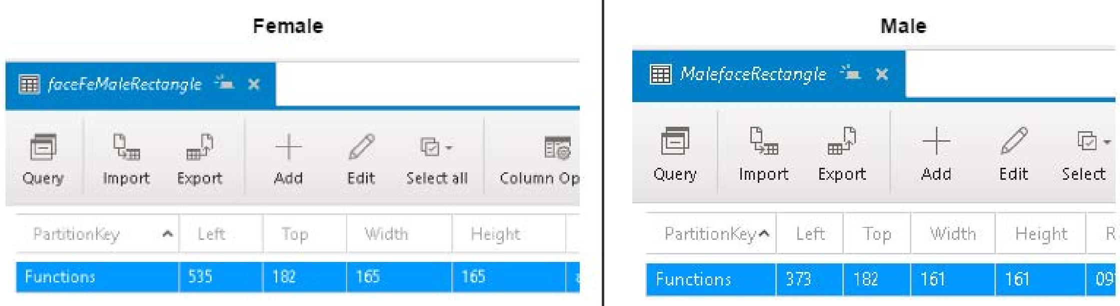


Figure 3.7: Azure Table storage—output values of the cognitive services

> **Note**
>
> 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 face locator templates invoke the API call by passing the `visualFeatures=Faces` parameter returns information relating to the following:

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

> **Note**
>
> Learn more about the Computer Vision API at https://docs.microsoft.com/azure/cognitive-services/computer-vision/home.

Use the **`Environment.GetEnvironmentVariable("KeyName")`** function to retrieve the information stored in the 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.

> **Note**
>
> It's considered a best practice to store all the keys and other sensitive information in the application settings.

In this recipe, you have learned how to integrate Cognitive Services with Azure Functions. Let's now move on to the next recipe to learn how to integrate Azure Functions with Logic Apps.

Monitoring and sending notifications using Logic Apps

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, including Facebook and Twitter. 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. Hence, he was looking for a solution that kept monitoring a particular hashtag and alerted him whenever a user with more than 50,000 followers tweets so that he can quickly have his team respond to that user.

> **Note**
>
> For the sake of simplicity, in this recipe, 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 exploring 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'll learn how to design a logic app that integrates with Twitter (for monitoring tweets) and Gmail (for sending emails).

Getting ready

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

- A valid Twitter account
- A valid Gmail account

When working with the recipe, you'll need to authorize Azure Logic Apps to access both Twitter and Gmail accounts.

The following are the logic app concepts that we'll be using in this recipe. They are the building blocks for developing logic apps:

- **Connectors**: Connectors are the wrappers around APIs that any system would provide to expose its features.
- **Actions**: Actions are steps in the Logic App workflow.
- **Trigger**: A trigger is the first step in any logic app, usually specifying the event that fires the trigger and starts running your logic app.

Learn more about them by referring to the following link: https://docs.microsoft.com/azure/connectors/apis-list

How to do it...

We'll go through the following steps:

1. Creating a new logic app.
2. Designing the logic app with Twitter and Gmail connectors.
3. Testing 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 portal**, search for `logic app`, and select **Logic App**.
2. In the **Create logic app** blade, once you have provided the **Name**, **Resource group**, **Subscription**, and **Location** information, click on the **Create** button to create the logic app:

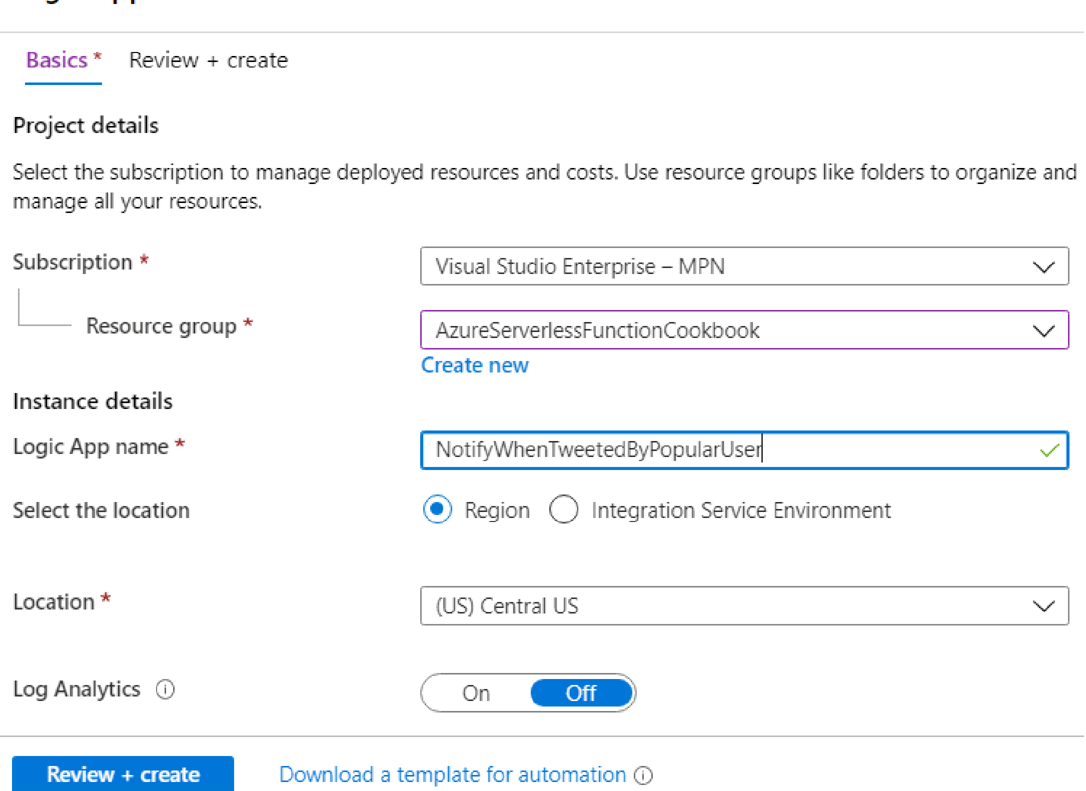


Figure 3.8: Creating a new logic app

In this section, we have created a logic app. Let's now move on to the next section.

Designing the logic app with Twitter and Gmail connectors

In this section, we will design the logic app by adding the required connectors and trigger, and by performing the following steps:

1. After the logic app has been created, navigate to **Logic app designer** and choose **Blank logic app**.
2. Next, you will be prompted to choose connectors. In the connectors list, search for `Twitter` and click on the Twitter connecter. This will show you the list of triggers associated with the Twitter connector, as shown in *Figure* 3.9. It will prompt you to connect to **Twitter** by asking for Twitter account credentials:

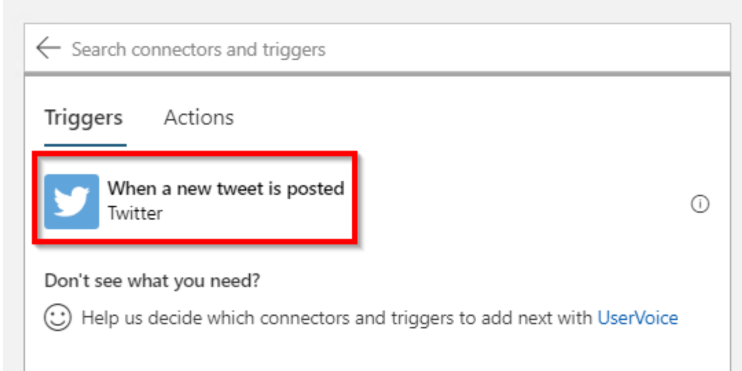


Figure 3.9: Logic app—selecting the trigger

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 appears after you provide the details:

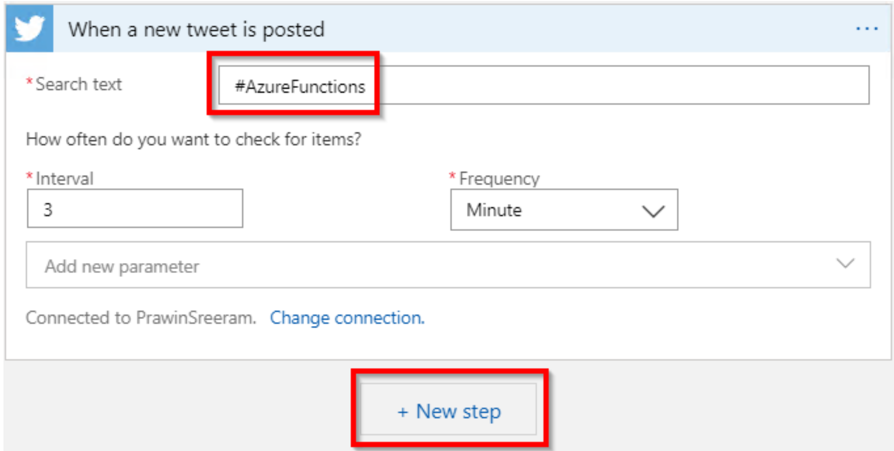


Figure 3.10: Logic app—providing the Twitter search text

4. Let's now add a new condition by clicking on **New step**, searching for **`control`**, and clicking on it, as shown in *Figure 3.11*:

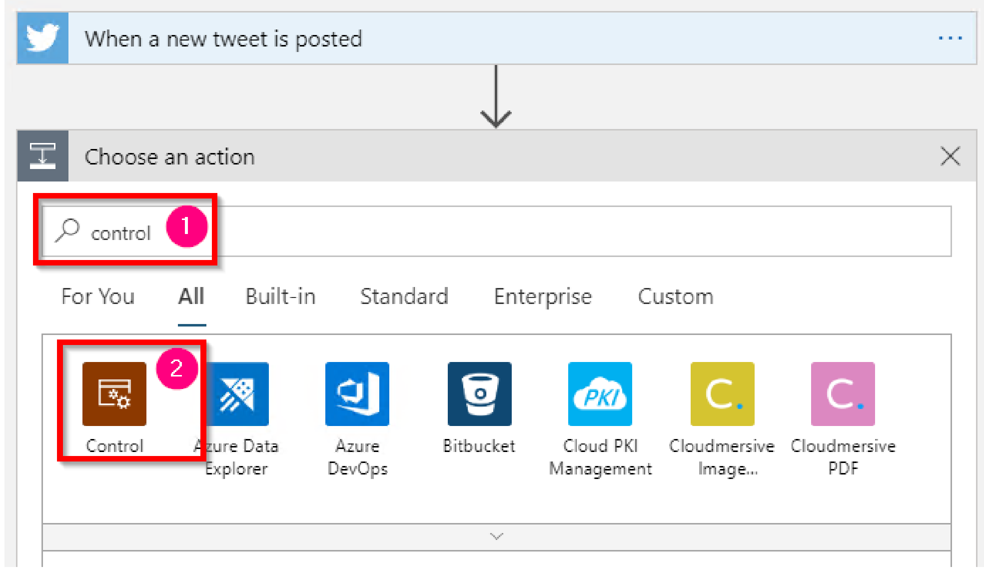


Figure 3.11: Logic app—searching for the control connector

5. It will now open **Actions**. Select **Condition**, as shown in *Figure 3.12*:

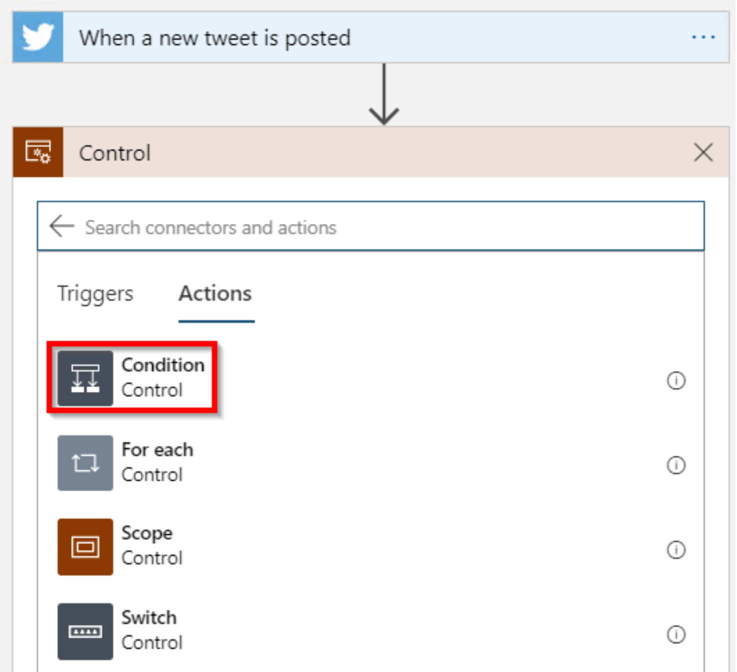


Figure 3.12: Logic app—selecting a condition

6. 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:

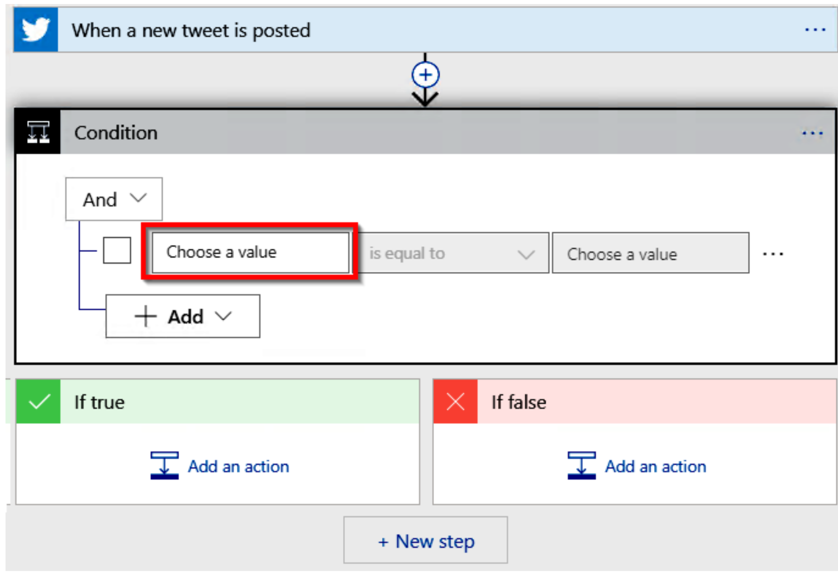


Figure 3.13: Logic app—selecting a condition—choosing a value

7. 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, you need to choose **Followers count**, as shown in *Figure 3.14*:

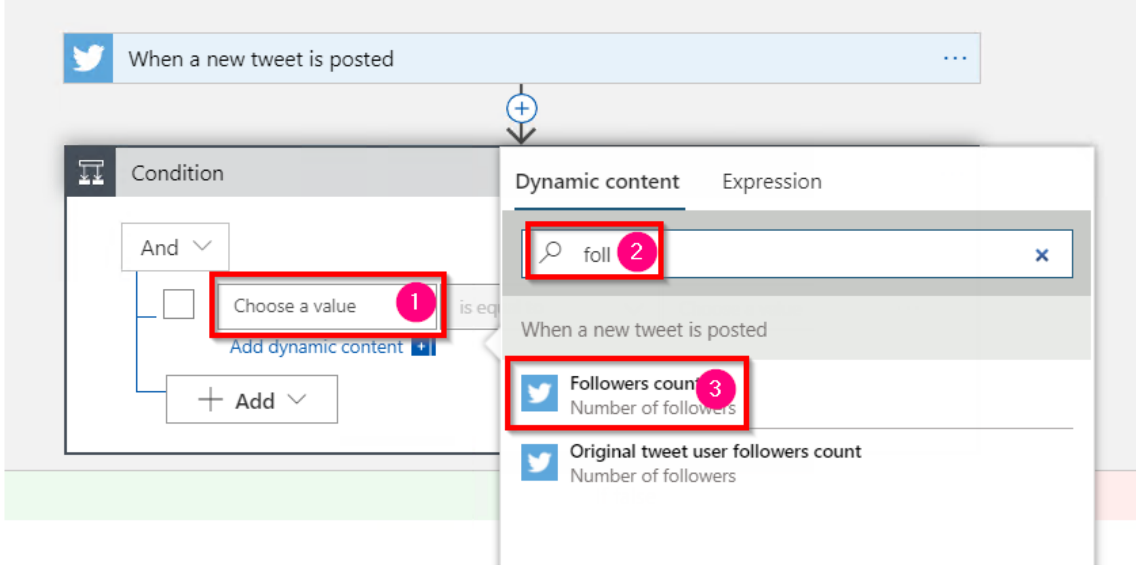


Figure 3.14: Logic app—selecting a condition—Followers count

8. Once you have chosen the **Followers count** parameter, you need to create a condition (followers count is greater than or equal to 200), as shown in *Figure 3.15*:

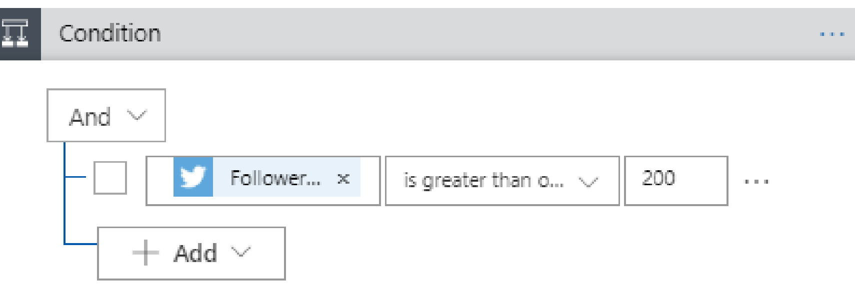


Figure 3.15: Logic app—selecting a condition—completed condition

9. In the **If true** section of the preceding **Condition**, click on the **Add an Action** button, search for the **Gmail** connector, and select **Gmail | Send email**, as shown in *Figure 3.16*:

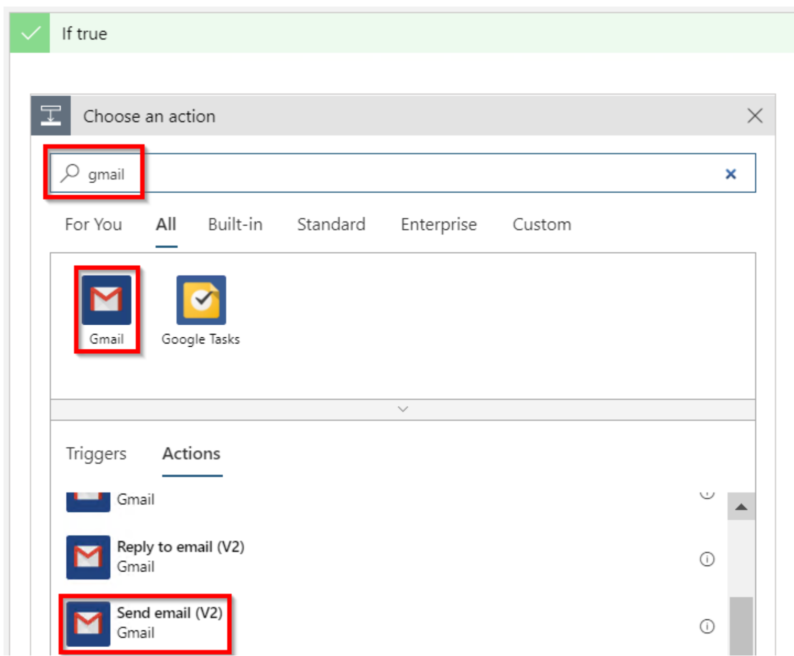


Figure 3.16: Logic app—If true action—sending an email using Gmail

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

11. Once you authorize, you can add parameters by clicking on the arrow, as shown in *Figure 3.17*, and selecting the **Subject** and **Body** checkboxes:

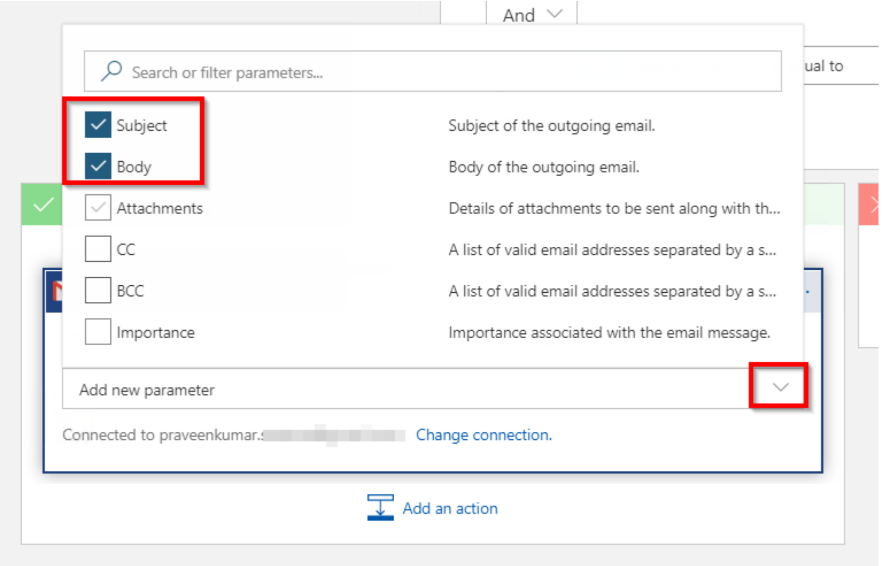


Figure 3.17: Logic app—If true action—configuring Gmail options

12. Once you have selected the fields, you can frame your email with **Add dynamic content** with the Twitter parameters, as shown in *Figure 3.18*:

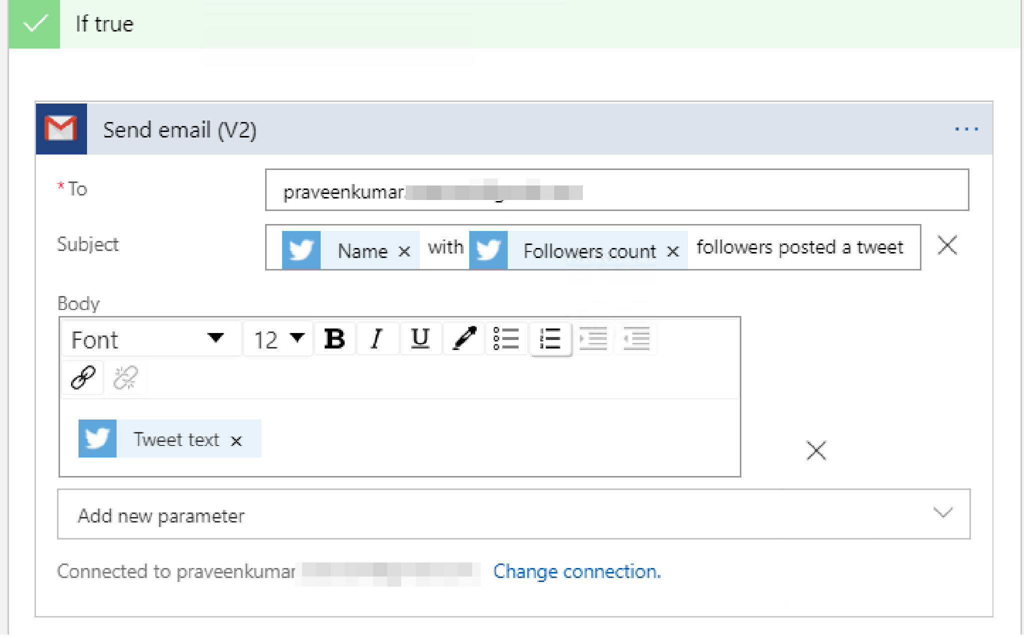


Figure 3.18: Logic app—If true action—configuring the subject and body

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

In this section, you have learned how to create a trigger that is triggered whenever a tweet is posted and you have also created a condition that sends emails using the Gmail connector if the followers count exceeds a specific value. Let's now move on to the next section to learn how to test the functionality.

Testing the logic app by tweeting the tweets with the specific hashtag

In this section, you will learn how to test the logic app by performing the following steps:

1. Post a tweet on Twitter with the hashtag **`#AzureFunctions`**, as shown in *Figure* 3.19:

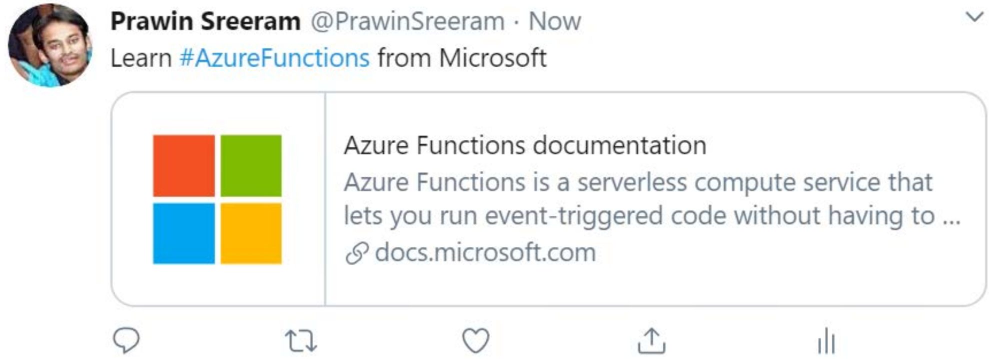


Figure 3.19: A tweet regarding the #AzureFunctions hashtag in Twitter

2. After three minutes, the logic app should have been triggered. Navigate to the **Overview** blade of the logic app and view **Runs history**:

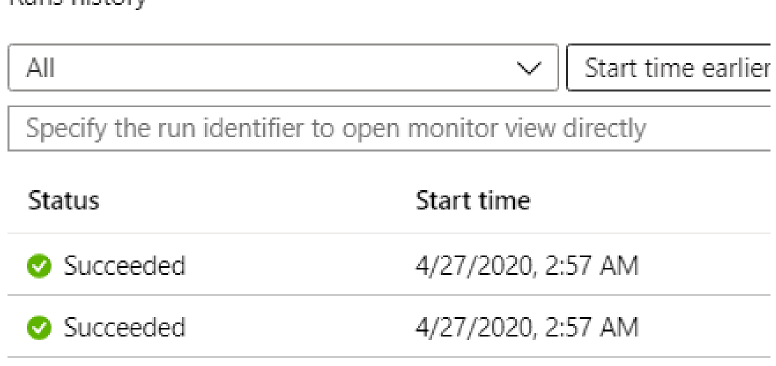


Figure 3.20: Logic app—Runs history

3. It triggered for me and I received the emails. One of them is shown in *Figure 3.21*:

Prawin Sreeram with 311 followers posted a tweet Inbox ×

praveenkumar
to me

Learn #AzureFunctions from Microsoft https://t.co/AXNcB2niUs

Figure 3.21: Logic app—email received following a tweet

How it works...

We have created a new logic app and have chosen the Twitter connector to monitor the tweets posted with the hashtag `#AzureFunctions` at three-minute intervals. 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.

In this recipe, we have designed a logic app that gets triggered whenever a tweet is posted on Twitter. You have also learned that Logic Apps allows you to add basic logic (in this recipe, a condition) without writing code. If you have some complex functionality, then it would not be possible to implement using Logic Apps. In such cases, Azure Functions would come in handy. Let's now move on to the next recipe to learn how to integrate Logic Apps with Azure Functions.

Integrating Logic Apps with serverless functions

In the previous recipe, you learned how to integrate different connectors using Logic Apps and developed a simple logic of checking whether the followers count is greater than `200`. As it was a simple logic, you were able to implement that in Logic Apps itself. If you need to implement a complex logic, then it wouldn't be possible. In that case, you can implement the complex logic in Azure Functions and invoke Azure Functions from Logic Apps.

In this recipe, you will see how to integrate Azure Functions with Logic Apps. For the sake of simplicity, we will not develop a complex logic. However, we will use the same logic (`followersCount` > `200`) in Azure Functions and invoke it from Logic Apps.

How to do it...

In this section, we'll integrate Azure Functions with Logic Apps by performing the following steps:

1. Create a new function by choosing the HTTP trigger with **Authorization Level** as **Anonymous**, and name it `ValidateTwitterFollowerCount`.
2. Replace the default code with the following, as shown in *Figure* 3.22. You can implement some complex logic here. The following Azure HTTP trigger accepts two parameters:

   The name of the owner of the tweet

   The follower count of the owner

   Just for the sake of simplicity, if the name of the user starts with the letter *p*, add `100` to the followers count. Otherwise, return the followers count. In your projects, you can implement complex business logic in the HTTP trigger:

Figure 3.22: Azure Function HTTP trigger

3. Navigate to the **NotifyWhenTweetedByPopularUser** logic app and click on **Logic App Designer** to start editing the logic app.
4. Now, hover the mouse on the arrow mark to reveal a **+** icon, as shown in *Figure* 3.23. Click on it and then click on **Add an action** button:

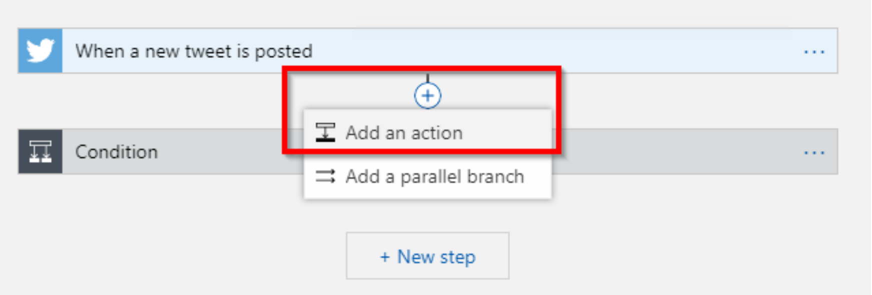


Figure 3.23: Logic app—Add an action

5. In the **Choose an action** section, search for **Functions**, click on **Azure Functions**, and then click on **Choose an Azure Function**, as shown in *Figure* 3.24:

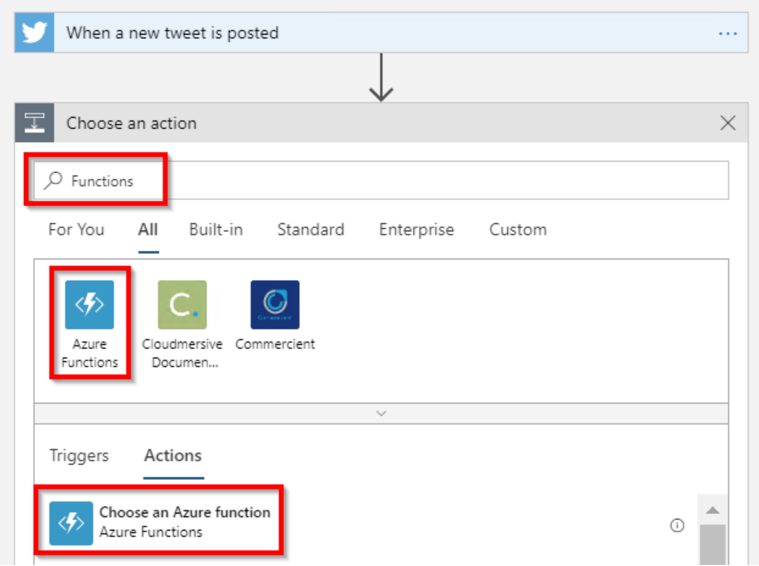


Figure 3.24: Logic app—searching for the Azure Functions connector

6. Clicking on the **Choose an Azure function** button in *Figure* 3.24 will reveal a list of all the available Azure function apps. You can search for the function app where you have developed the `ValidateTwitterFollowerCount` function, as shown in *Figure* 3.25:

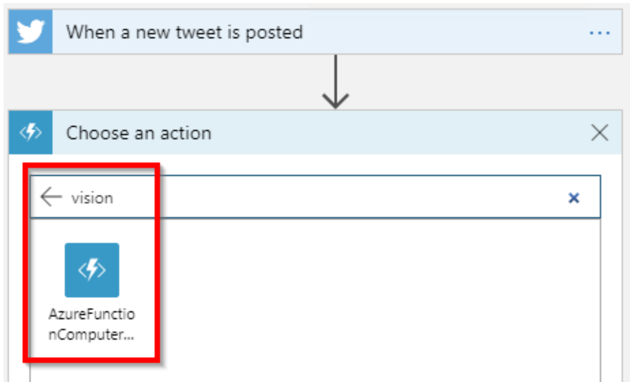


Figure 3.25: Logic app—choosing an Azure function app

7. When you choose the function app, it will show all the functions inside it. Click on the **`ValidateTwitterFollowerCount`** function, as shown in *Figure 3.26*:

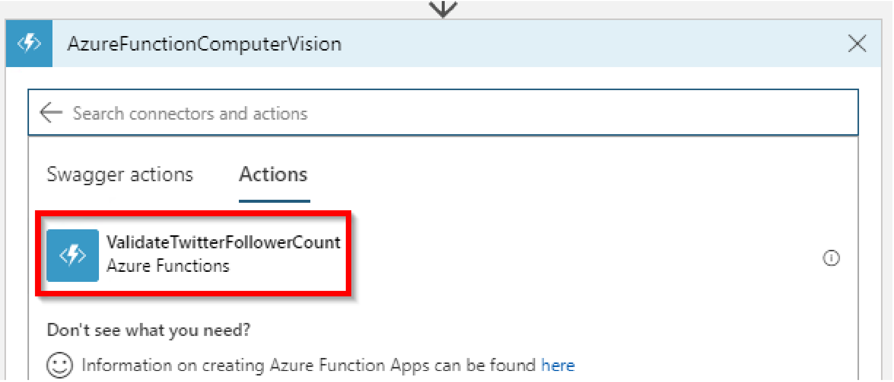


Figure 3.26: Logic app—choosing Azure Functions

8. Now, provide the input to the **`ValidateTwitterFollowerCount`** function, as shown in *Figure 3.27*:

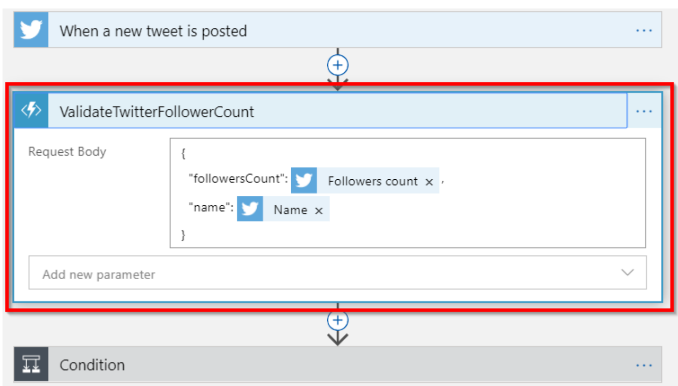


Figure 3.27: Logic app—passing inputs to Azure Functions

9. In order to receive the response, open the **Condition** step and add the **Body** variable, as shown in *Figure* 3.28. The **Body** variable contains the response body of the Azure function named `ValidateTwitterFollowerCount`. This means that, instead of directly comparing the Twitter follower count, we are comparing the response returned by the Azure Function HTTP trigger with the value `200`:

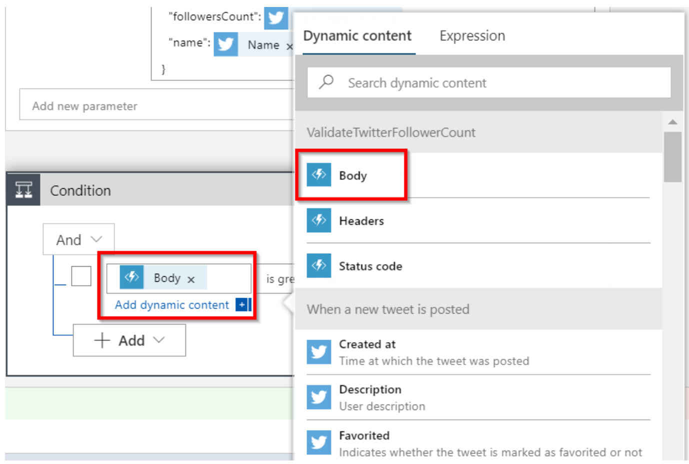


Figure 3.28: Logic app—receiving inputs from Azure Functions

10. That's it. Save the changes and go to Twitter and create a tweet with `#AzureFunctions`. The following is an example where the `followersCount` returned by Twitter for my tweet is `310`. However, the Azure function added `100` to the followers count and returned `410` in the response body:

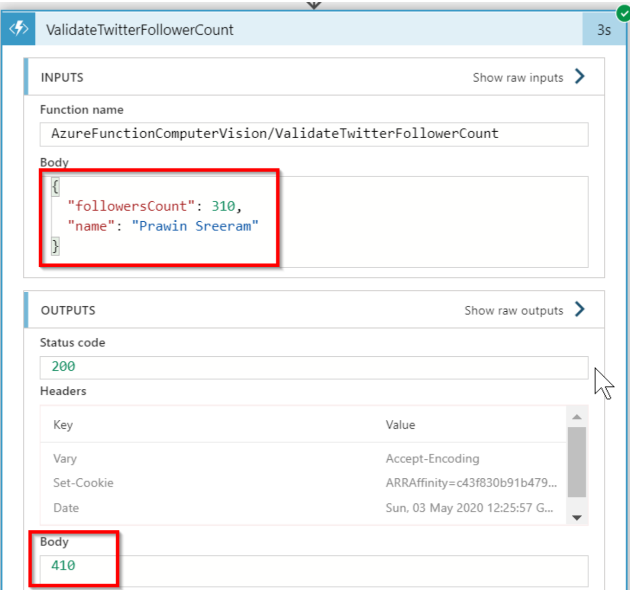


Figure 3.29: Logic app—execution history

In this section, you have learned how to integrate Logic Apps with Azure Functions.

There's more...

If you don't see the intended dynamic parameter, click on the **See more** button, as shown in *Figure 3.30*:

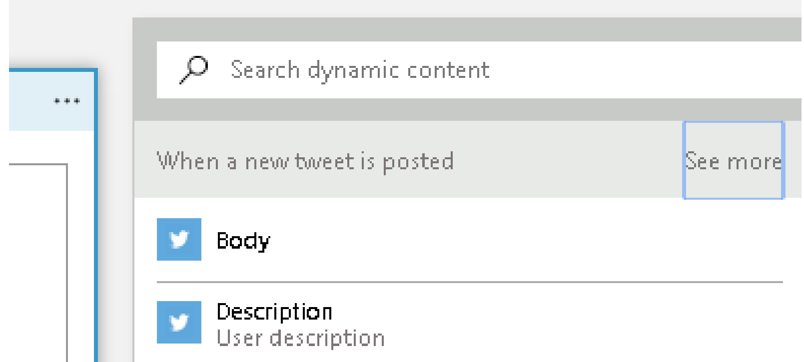


Figure 3.30: Logic app—dynamic content—see more

In this recipe, you have learned how to integrate Azure Functions with Logic Apps. In the next recipe, let's explore how to integrate Azure Functions with Cosmos DB.

Auditing Cosmos DB data using change feed triggers

You may 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, you will learn to integrate serverless Azure Functions with a serverless NoSQL database in Cosmos DB. You can read more about Cosmos DB at https://docs.microsoft.com/azure/cosmos-db/introduction.

It might often be necessary to keep the change logs of fields, attributes, items, and other aspects 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 in a separate audit table.

In this recipe, you'll learn how easy it is to audit the changes made to Cosmos DB containers by writing a simple function that gets triggered whenever there is a change to an item in a Cosmos DB container.

> **Note**
>
> In the world of relational databases, a container is the same as a database table and an item is the same as a record.

Getting ready

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

1. Create a Cosmos DB account.
2. Create a new Cosmos DB container where you can store data in the form of items.

Let's begin by creating a new Cosmos DB account.

Creating a new Cosmos DB account

Navigate to the Azure portal and create a new Cosmos DB account. You will 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 container

Create a new Cosmos DB container by performing the following steps:

1. Once the account has been created, create a new database and a container. You can create both of them in a single step right from the Azure portal.
2. Navigate to the **Overview** tab and click on the **Add Container** button to create a new container:

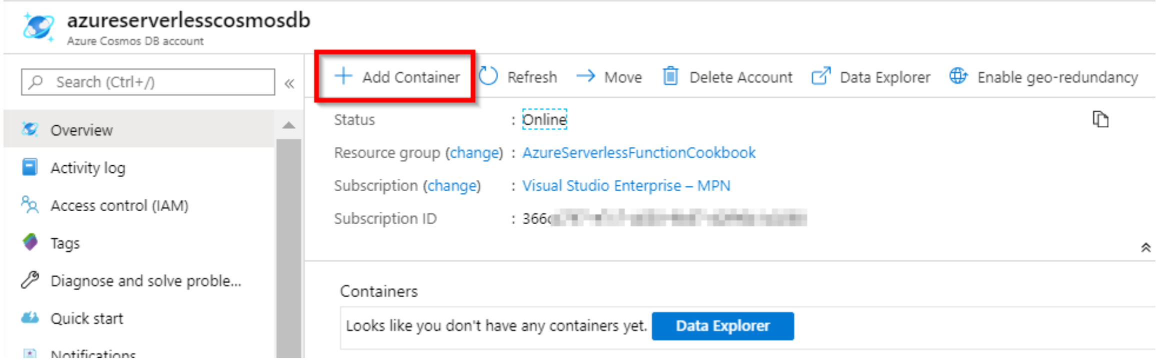


Figure 3.31: Cosmos DB account—Overview blade

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

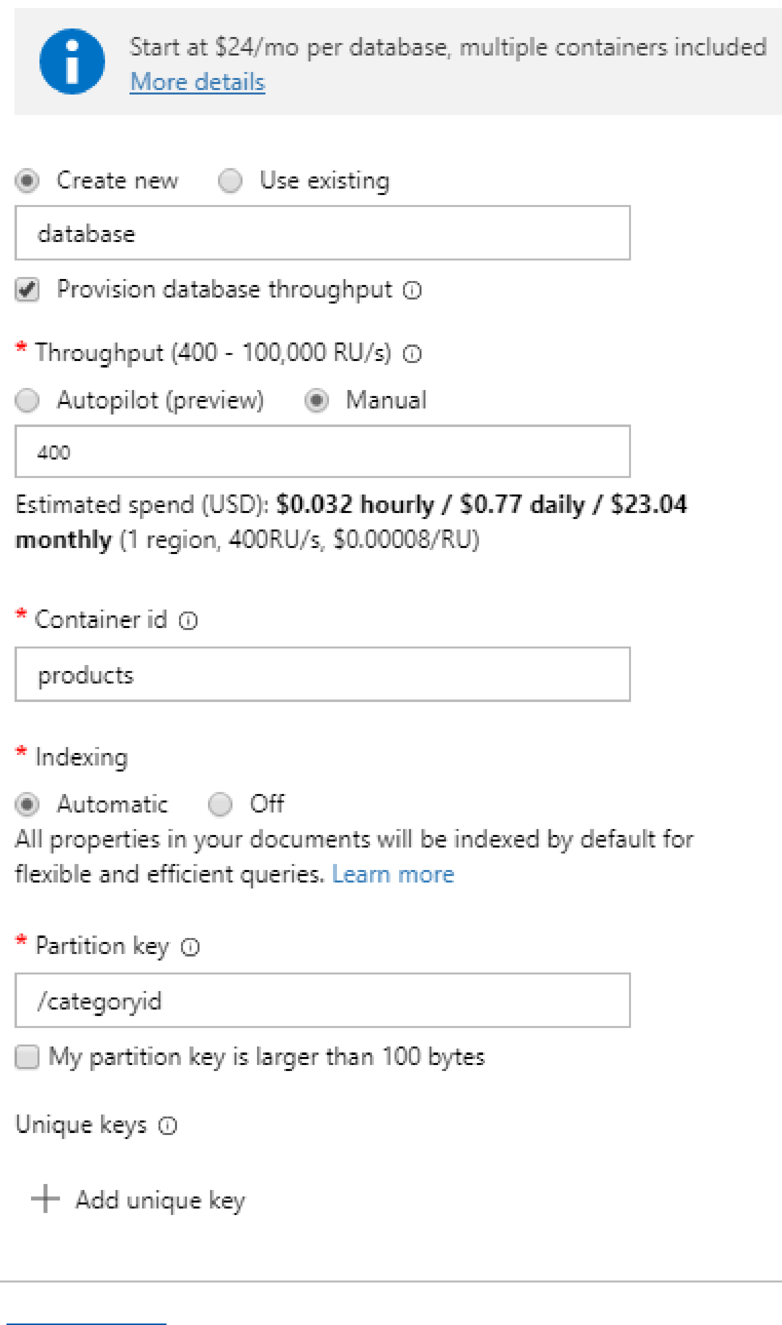


Figure 3.32: Cosmos DB—creating a container

The following are the details that you'll need to add to *Figure* 3.32:

Field Name	Value	Description
Database id	`database`	Database containing multiple Cosmos DB containers.
Container id	`products`	This is the name of the container where you will be storing the data.
Partition key	`/categoryid`	All the items of a given container will be segregated based on the partition. A separate partition is created for each unique value for the **`categoryid`** field in the product container that is created.
Throughput (400 – 10,000 RU/s)	`400`	This is the capacity of your Cosmos DB database. The performance of the reads and writes depends on the throughput that you configure when provisioning the container.

Figure 3.33: Cosmos DB—creating a container—fields

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

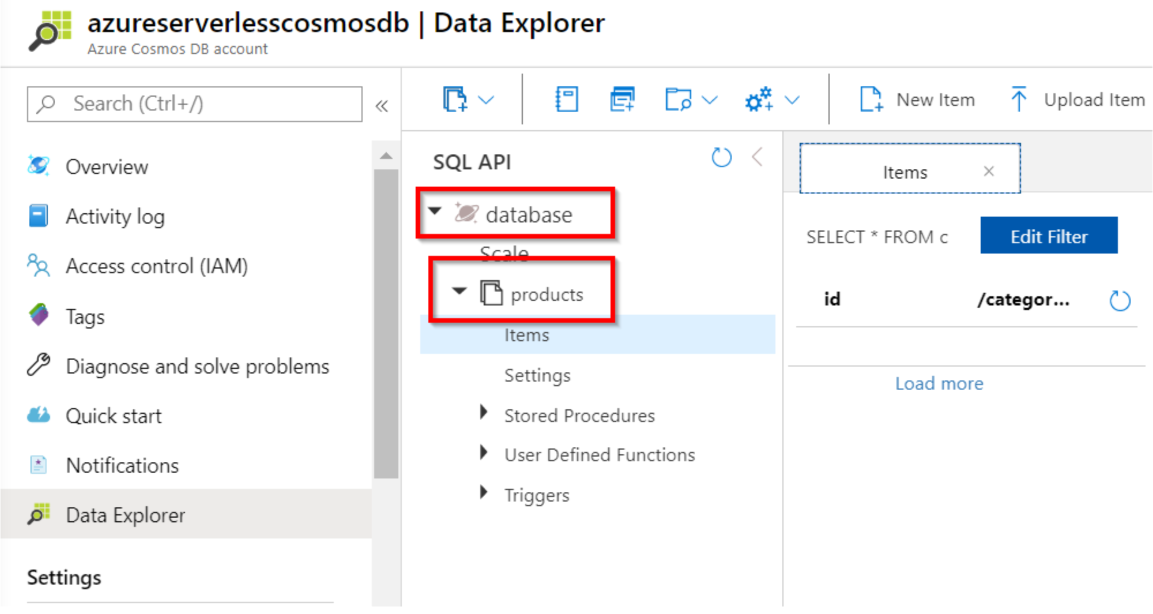


Figure 3.34: Data Explorer—Cosmos DB database and container

We have successfully created a Cosmos DB database and a container, as shown in *Figure* 3.34. Let's now go through how to integrate the container with a new Azure function and see how to trigger it whenever there is a change in the Cosmos DB container.

How to do it...

In this section, we'll integrate a Cosmos DB container with Azure Functions by performing 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, as shown in *Figure* 3.35:

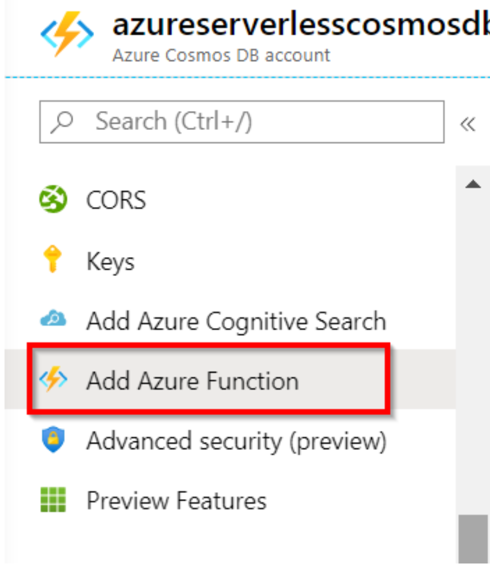


Figure 3.35: Cosmos DB—the Add Azure Function menu item

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), as shown in *Figure* 3.36:

1 Select container

Select the container to monitor for changes. Your Azure Function will receive batches of changed items to be processed.

products

2 Create Azure Function

Select an Azure Function app

cosmosdbazurefunctions

Name your Azure Function *

productsTrigger

Function language

C#

Save Discard

Figure 3.36: Cosmos DB—Azure function integration

3. Once you have reviewed the details, click on the **Save** button (shown in *Figure 3.36*) to create the new function, which will be triggered for every change that is made in the container. Let's quickly navigate to the Azure function app (in my case, it is **`cosmosdbazurefunctions`**) and see whether the new function with the name **`productsTrigger`** has been created. Here is what the function app looks like:

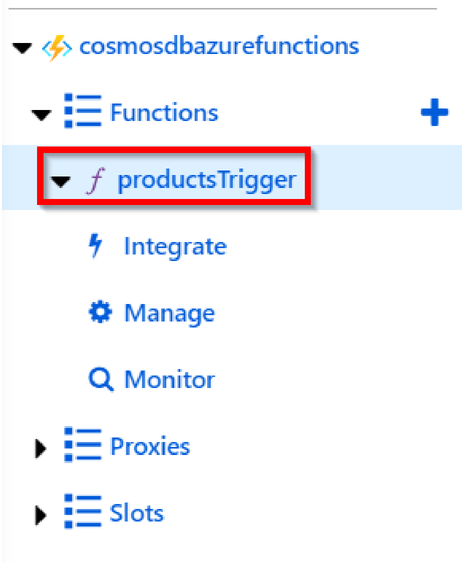


Figure 3.37: Azure Functions—Cosmos DB trigger

4. Replace the default code with the following code of the Azure Functions Cosmos DB trigger, which gets a list of all the items that were updated. The following code just prints the count of items that were updated and the ID of the first item 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("Items modified " + input.Count); log.
LogInformation("First Item Id " + input[0].Id);
}
}
```

5. Now, the integration of the Cosmos DB container and the Azure function is complete. Let's add a new item to the container and see how the trigger gets fired in action. Open a new browser window/tab (leaving the **productsTrigger** tab open in the browser), navigate to the container, and create a new item by clicking on the **New Item** button, as shown in *Figure* 3.38:

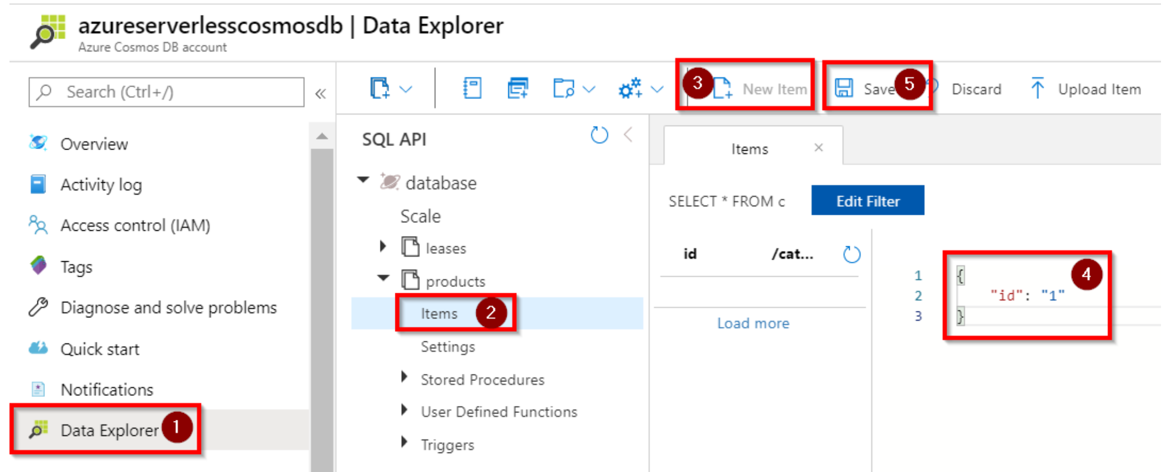


Figure 3.38: Data Explorer—creating a new item

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 item. It might look different for you, depending on your JSON structure:

Logs Console Reconnect Copy logs Pause Clear Expand

```
2020-05-03T03:16:27.132 [Information] Executing 'Functions.productsTrigger' (Reason='New changes on collection products at 2020-05-03T03:16:27.1319493Z', Id=25121da7-cee3-4c7f-acb0-b93d6a8c1b10)
2020-05-03T03:16:27.133 [Information] Items modified 1
2020-05-03T03:16:27.133 [Information] First Item Id 1
2020-05-03T03:16:27.134 [Information] Executed 'Functions.productsTrigger' (Succeeded, Id=25121da7-cee3-4c7f-acb0-b93d6a8c1b10)
```

Figure 3.39: Azure function—the Logs console

In this section, you have learned how to integrate Azure Functions with Cosmos DB. Let's now move on to the next section.

There's more...

While integrating Azure Functions to track Cosmos DB changes, it will automatically create a new container named **leases**, as shown in *Figure* 3.40. 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 container:

Containers

ID	Database	Throughput (RU/s)
products	database	400 (Shared)
leases	database	400 (Shared)

Figure 3.40: Cosmos DB—the Containers list

It's important to note that the Cosmos DB trigger wouldn't be triggered (at the time of writing) for any deletes in the container. It is only triggered for creates and updates to items in a container. If it is important for you to track deletes, and then you need to execute 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/azure/cosmos-db/change-feed

Don't forget to delete the Cosmos DB account and its associated containers if you think you will no longer use them, because the containers 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 an error saying that Cosmos DB extensions are not installed, then try creating a new Azure Cosmos DB trigger using the templates available, which should then prompt installation.

In this recipe, we first created a Cosmos DB account and created a database and a new container within it. Once the container 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 chose the required function app in which we wanted to create a Cosmos DB trigger. Once the integration was complete, we created a sample item in the Cosmos DB container and then verified that the function was triggered automatically for all the changes (all reads and writes but not deletes) that we will make on the container.

Let's now proceed to integrate Azure Functions with Azure Data Factory.

Integrating Azure Functions with Data Factory pipelines

In many enterprise applications, the need to work with data is definitely there, especially when there are a variety of heterogeneous data sources. In such cases, we need to identify tools that help us to extract the raw data, transform it, and then load the processed data into other persistent media to generate reports.

Azure assists organizations in carrying out the preceding scenarios by using a service called **Azure Data Factory** (**ADF**).

Azure Data Factory is another cloud-native serverless solution from Microsoft Azure. ADF can be used as an **Extract, Transform, and Load** (**ETL**) tool to process the data from various data sources, transform it, and load the processed data into a wide variety of data destinations. Before we start working with the recipe, I would recommend that you learn more about Azure Data Factory and its concepts at https://docs.microsoft.com/azure/data-factory/introduction.

When we have complex processing requirements, ADF will not let us write complex custom logic. Fortunately, ADF supports the plugging of Azure functions into ADF pipelines, where we can pass input data to Azure Functions and also receive the returned values from Azure Functions.

In this recipe, you'll learn how to integrate Azure Functions with ADF pipelines. The following is a high-level architecture diagram that depicts what we are going to do in this recipe:

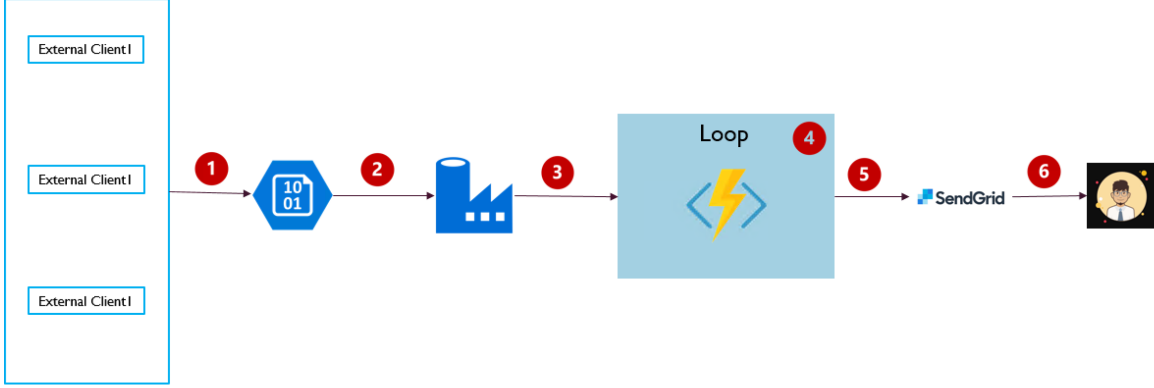


Figure 3.41: Integration of Azure Functions with an ADF pipeline

As shown in the preceding architecture diagram, we are going to implement the following steps:

1. Client applications upload the employee data in the form of CSV files to the storage account as blobs.
2. Trigger the ADF pipeline and read the employee data from the storage blob.
3. Call a **ForEach** activity in the Data Factory pipeline.
4. Iterate through every record and invoke Azure Function HTTP trigger to implement the logic of sending emails.
5. Invoke **SendGrid** output bindings to send the emails.
6. The end user receives the email.

Getting ready…

In this section, we'll create the prerequisites to start working on this recipe. The prerequisites for this recipe are the following.

1. Upload the CSV files to a storage container.
2. Create an Azure Function HTTP trigger with the authorization level set to **Function**.
3. Create a Data Factory instance.

Uploading the CSV files to a storage container

Please create a storage account and a container, and upload the CSV file that contains the employee information, as shown in *Figure* 3.42:

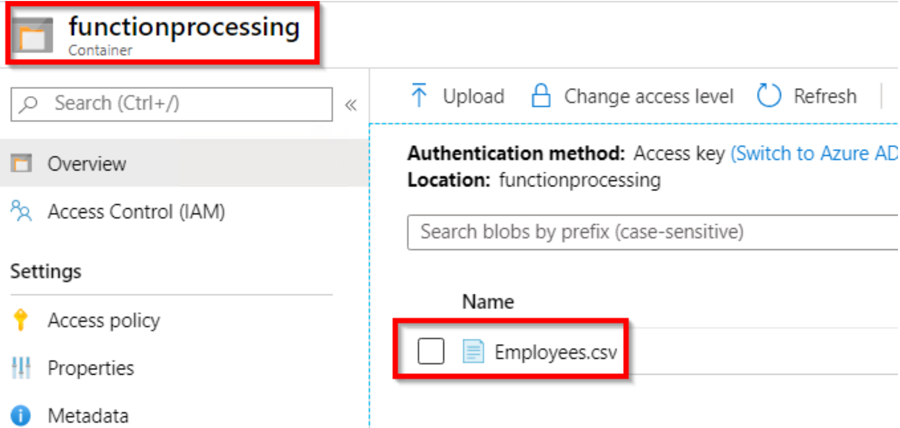


Figure 3.42: Storage container

The following is an example of the CSV file. Please make sure that there is a column named **Email**. We'll be using this field to pass data from the Data Factory pipeline to Azure Functions:

A	B	C	D	E
Emp Id	Name	Email	PhoneNumber	
1	Nischala	Nischala@gmail.com	1.11E+09	
2	Vivek	vivek@gmail.com	2.22E+09	
3	Khadir	Khadir@gmail.com	3.33E+09	
4	Bhargavi	Bhargavi@gmail.com	4.44E+09	
5	Praveen S	praveen@gmail.com	5.56E+09	
6	Meena	meena@gmail.com	6.67E+09	

Figure 3.43: Employee data in a CSV file

Having uploaded the **`Employees.csv`** file to a storage container, let's move on to the next section.

Creating an Azure Function HTTP trigger with the authorization level set to Function

In this section, we are going to create an HTTP trigger and also a linked service for the function app in the Data Factory service.

Create an HTTP function named **`SendMail`**. This receives an input name email and it also prints the values, as shown in *line* 18 in *Figure* 3.44:

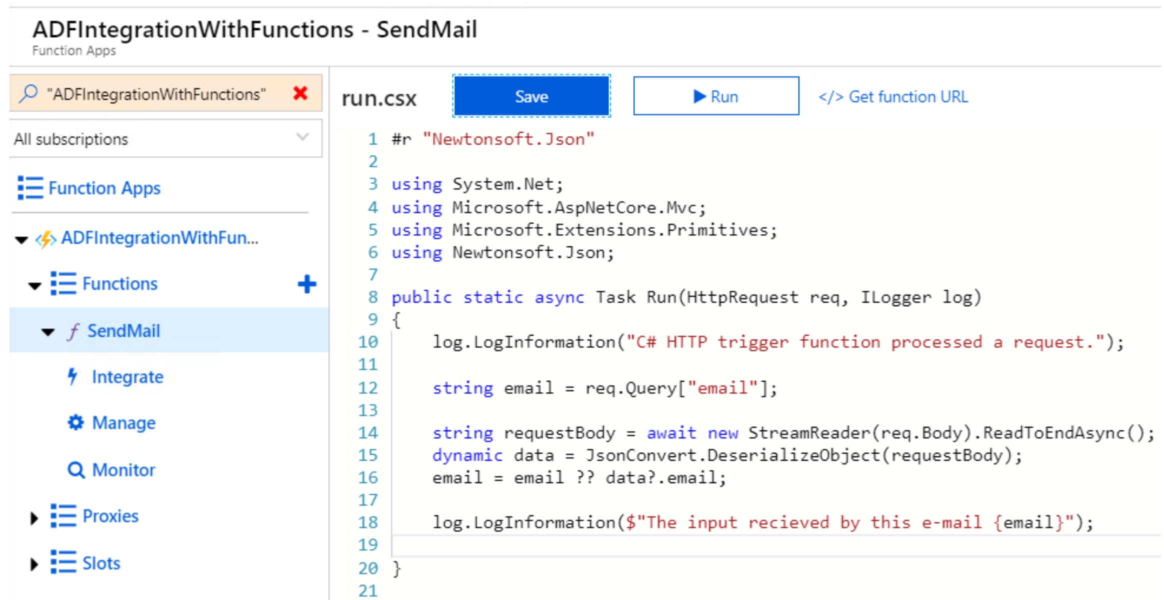


Figure 3.44: Creating an Azure function HTTP trigger

In this section, we have created an Azure function with the HTTP authorization set to **`Function`**. Let's now move on to the next section to create the Data Factory instance.

Creating a Data Factory instance

In this section, we'll create a Data Factory instance by performing the following steps.

1. Click on **Create a resource** and search for **`Data Factory`**, as shown in *Figure* 3.45. This will take you to the next step, where you must click on the **Create** button:

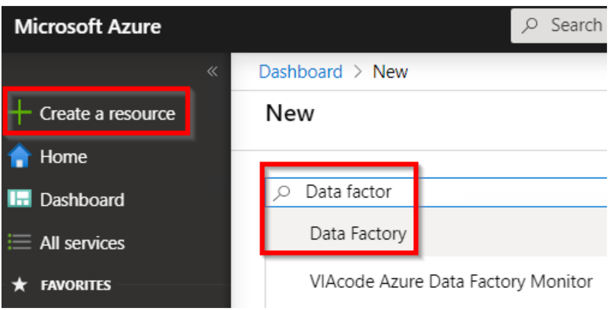


Figure 3.45: Searching for Data Factory

2. In the **New data factory** blade, provide the name and other details, as shown in *Figure* 3.46, and click on the **Create** button:

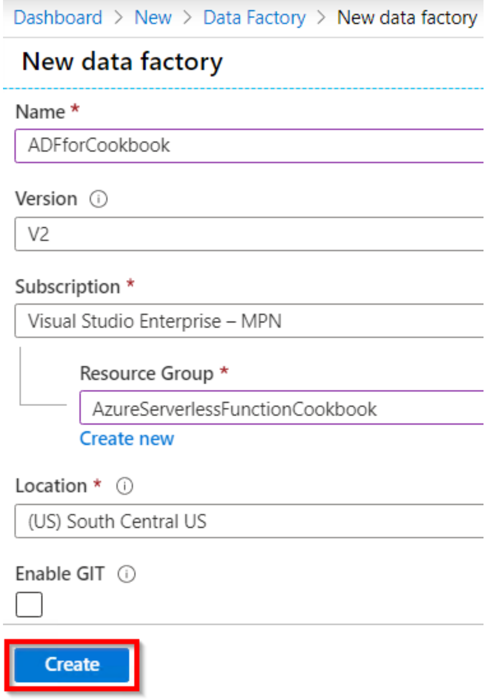


Figure 3.46: Creating a new Data Factory instance

3. Once the **Data Factory** service is created, click on the **Author & Monitor** button available in the **Overview** blade, as shown in *Figure* 3.47:

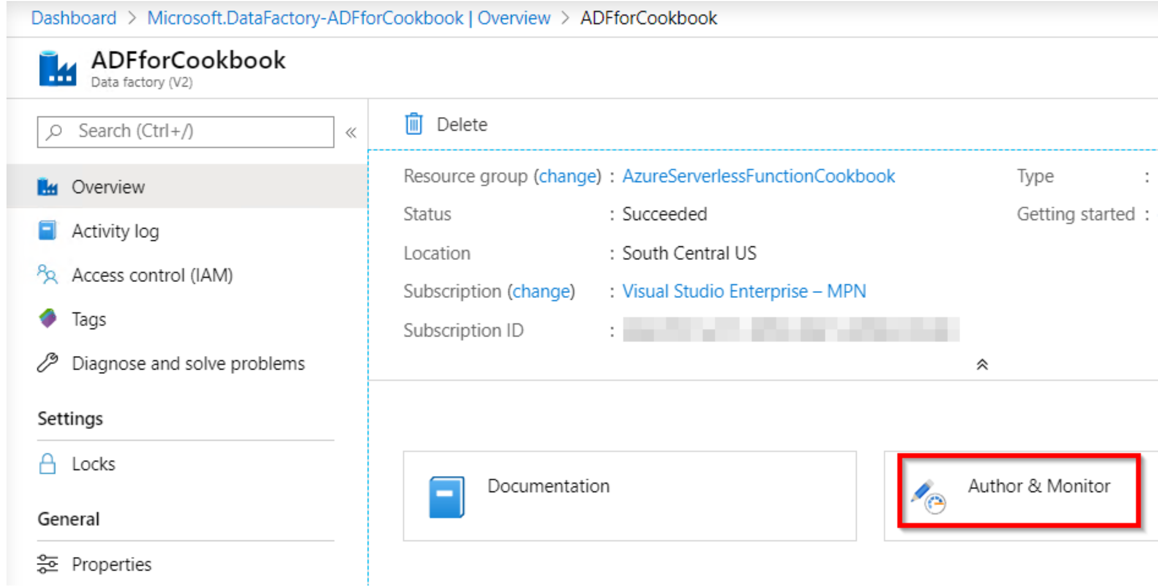


Figure 3.47: Author & Monitor

4. Now, it will open up a new browser tab and take you to the https://adf.azure.com page, where you can see the **Let's get started** section.

5. In the **Let's get started** view, click on the **Create pipeline** button, as shown in *Figure* 3.48, to create a new pipeline:

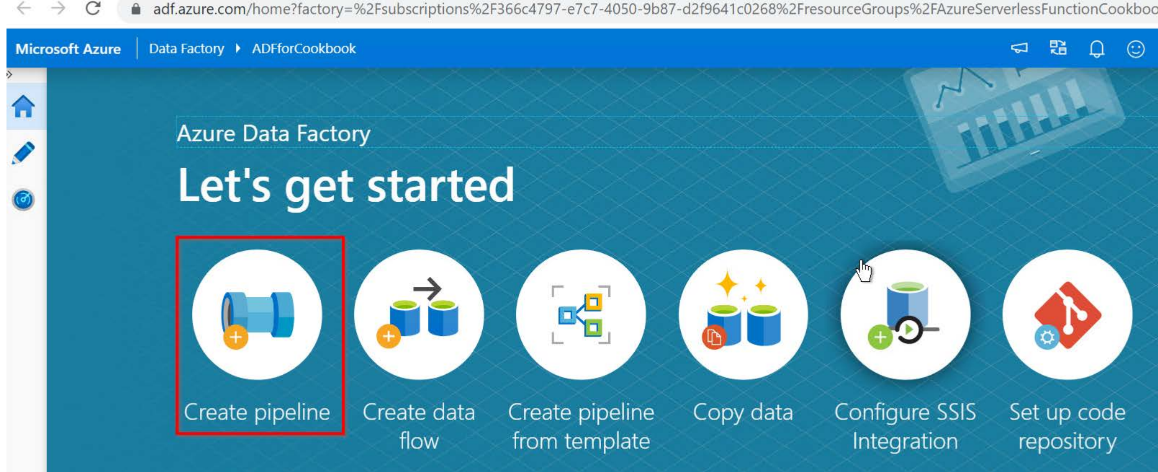


Figure 3.48: ADF—Let's Get Started

6. This will take you to the **Authoring** section, where you can author the pipeline, as shown in *Figure* 3.49:

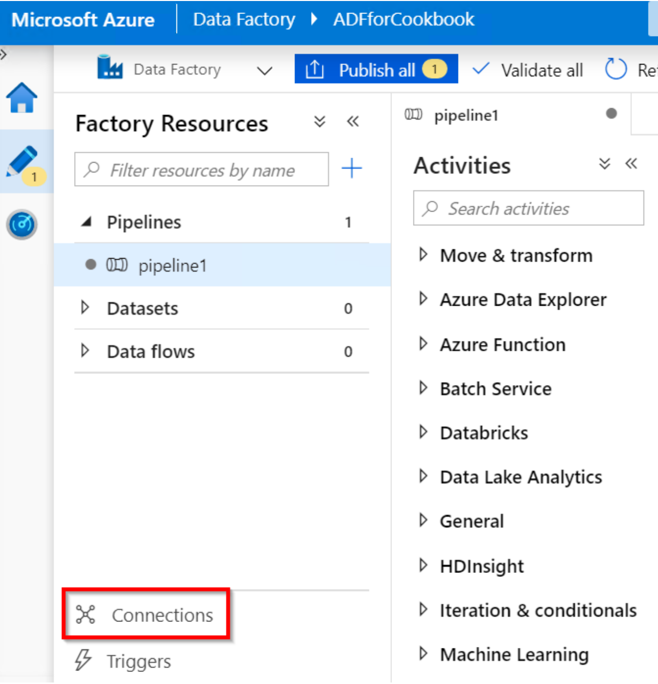


Figure 3.49: ADF—new pipeline

7. Before you start authoring the pipeline, you need to create connections to the storage account and Azure Functions. Click on the **Connections** button, as shown in *Figure* 3.49.

8. In the **Linked services** tab, click on the **New** button, search for **`blob`** in the **Data store** section, and select **Azure Blob Storage**, as shown in *Figure* 3.50:

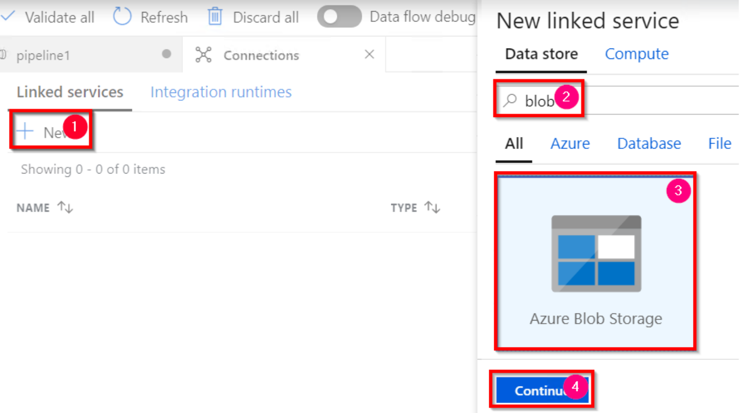


Figure 3.50: ADF—New linked service—choosing a linked service

9. In the **New linked service** pop-up window, provide the name of the linked service, choose **Azure subscription** and **Storage account name**, test the connection, and then click on the **Create** button to create the linked service for the storage account, as shown in *Figure 3.51*:

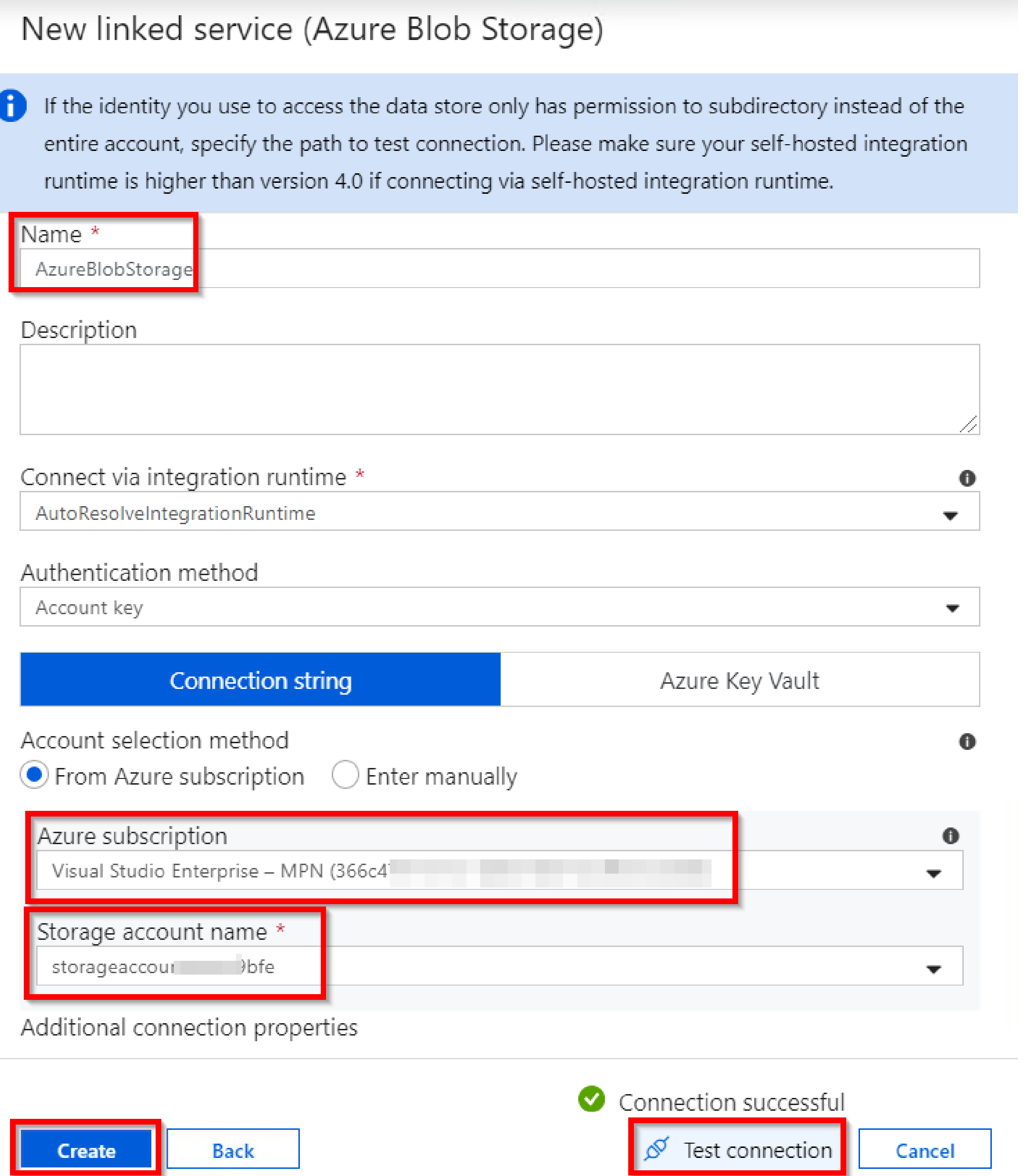


Figure 3.51: ADF—New linked service—providing connection details

10. Once you click on the **Create** button, this will create a linked service, as shown in *Figure 3.52*:

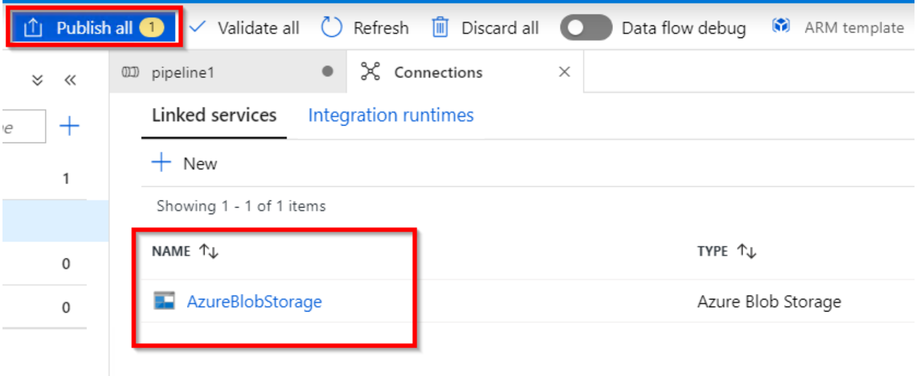


Figure 3.52: ADF—Linked services

11. After reviewing the linked service, click on **Publish all** to save the changes to the Data Factory instance.
12. Now create another linked service for Azure Functions by again clicking on **New button** in the **Connections** tab.
13. In the **New linked service** pop-up window, choose the **Compute** drop-down option, select **Azure Function**, and then click on the **Continue** button, as shown in *Figure 3.53*:

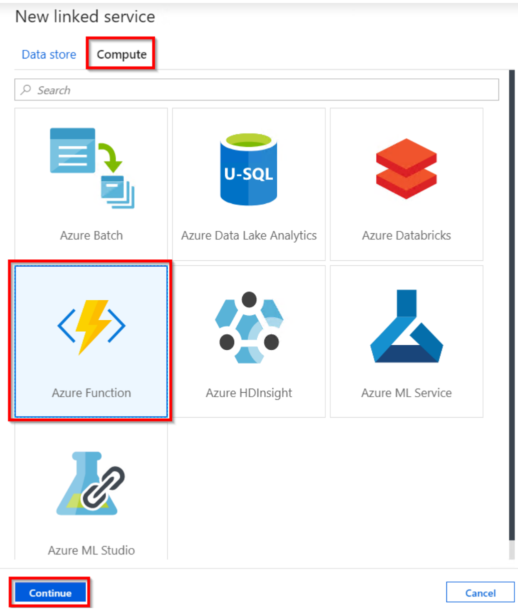


Figure 3.53: ADF—New linked service—choosing Azure Function

14. In the next step, provide a name to the linked service, choose the subscription function app, provide the **Function Key** value, and then click on **Create**, as shown in *Figure 3.54*:

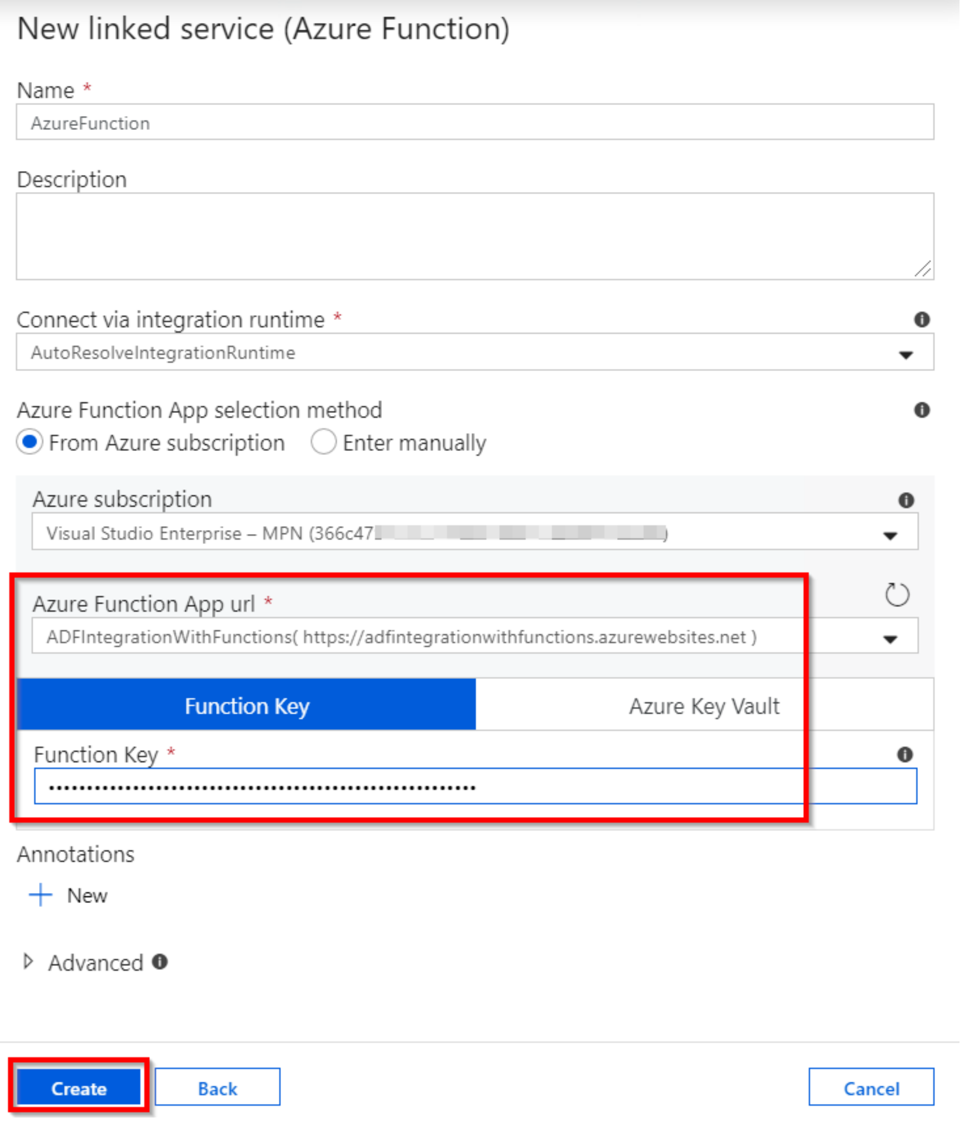


Figure 3.54: ADF—New linked service—Azure function app

> **Note**
>
> You can get the **Function Key** value from the **Manage** blade of the function app. Function keys are discussed in detail in the *Controlling access to Azure Functions using function keys* recipe of *Chapter 9, Configuring security for Azure Functions*.

15. Once the Azure function linked service is created, you should see something similar to *Figure* 3.55. Click on **Publish all** to save and publish the changes to the Data Factory service:

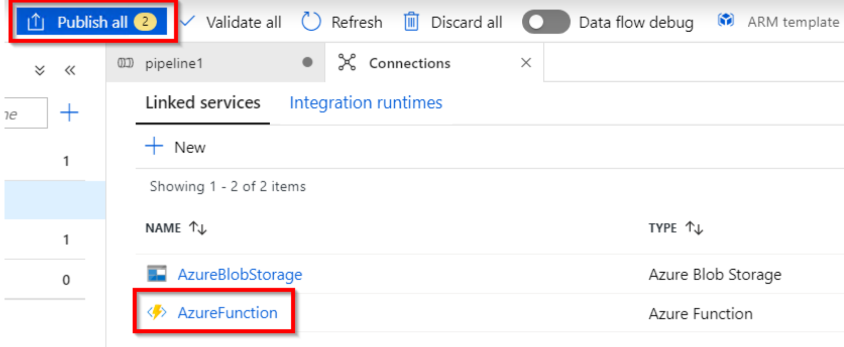


Figure 3.55: ADF—Linked services

In this section, we have created the following:

1. A Data Factory instance
2. A Data Factory pipeline
3. A linked service to the storage account
4. A linked service to Azure Functions

We will now move on to the next section to see how to build the Data Factory pipeline.

How to do it...

In this section, we are going to create the Data Factory pipeline by performing the following steps:

1. Create a **Lookup** activity that reads the data from the storage account.
2. Create a **ForEach** activity that takes input from the **Lookup** activity. Add an **Azure Function** activity inside the **ForEach** activity.
3. The **ForEach** activity iterates based on the number of input items that it receives from the **Lookup** activity and then invokes the Azure function to implement the logic of sending the emails.

Let's begin by **creating the Lookup activity by performing the following steps:**

1. Drag and drop the **Lookup** activity that is available in the **General** section and name the activity as `ReadEmployeeData`, as shown in *Figure* 3.56. Learn more about the activity by clicking on the **Learn more** button highlighted in *Figure* 3.56:

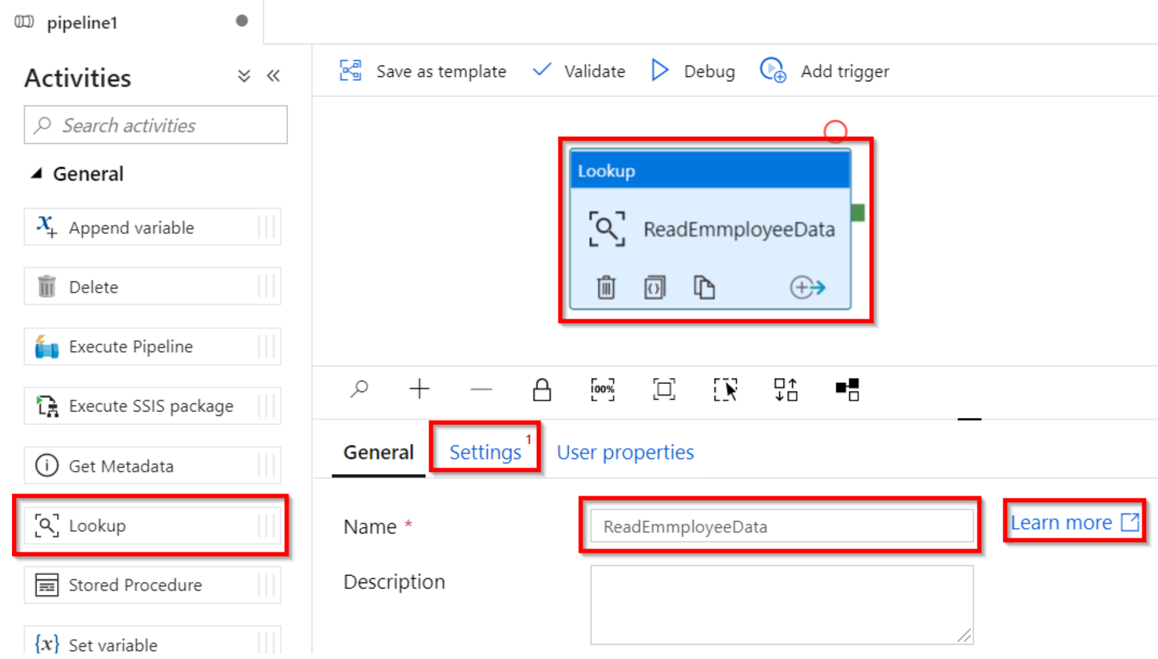


Figure 3.56: ADF—Lookup activity settings

2. Select the **Lookup** activity and click on the **Settings** button, which is available in *Figure* 3.56. By default, the **Lookup** activity reads only the first row. Your requirement is to read all the values available in the CSV file. So, uncheck the **First row only** checkbox, which is shown in *Figure* 3.57:

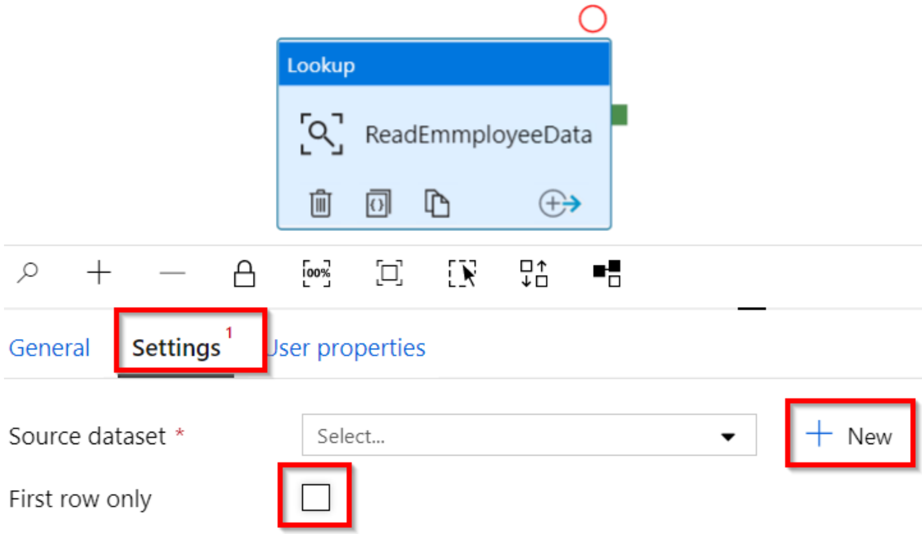


Figure 3.57: ADF—Lookup activity—new source dataset

3. The **Lookup** activity's responsibility is to read data from a blob. The **Lookup** activity requires a dataset to refer to the data stored in the blob. Let's create a dataset by clicking on the **New** button, as shown in *Figure* 3.57.

4. In the **New dataset** pop-up window, choose **Azure Blob Storage** and then click on the **Continue** button, as shown in *Figure* 3.58:

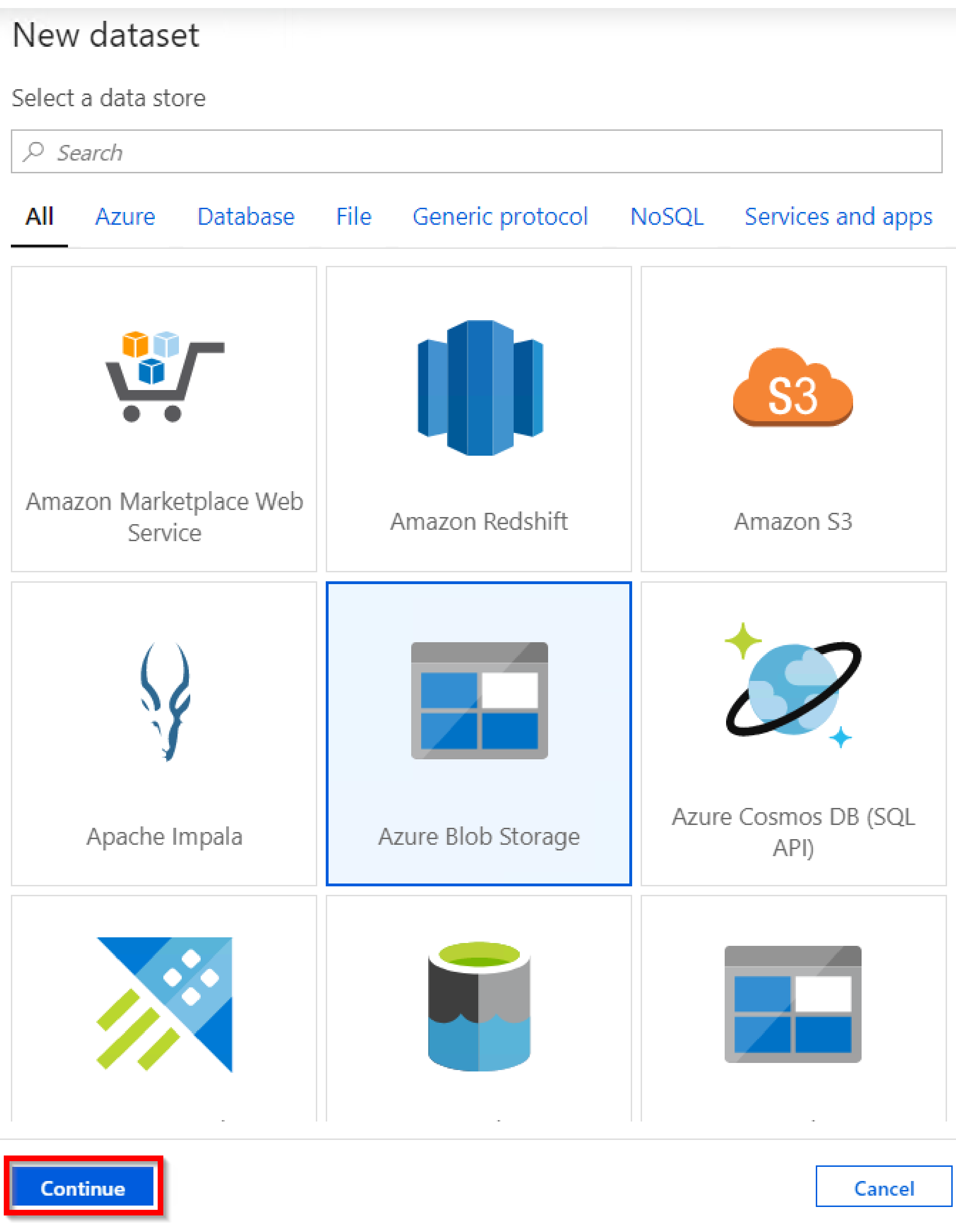


Figure 3.58: ADF—Lookup activity—new source dataset—choosing Azure Blob Storage

5. In the **Select format** pop-up window, click on the **Delimited Text** option, as shown in *Figure* 3.59, and click **Continue**:

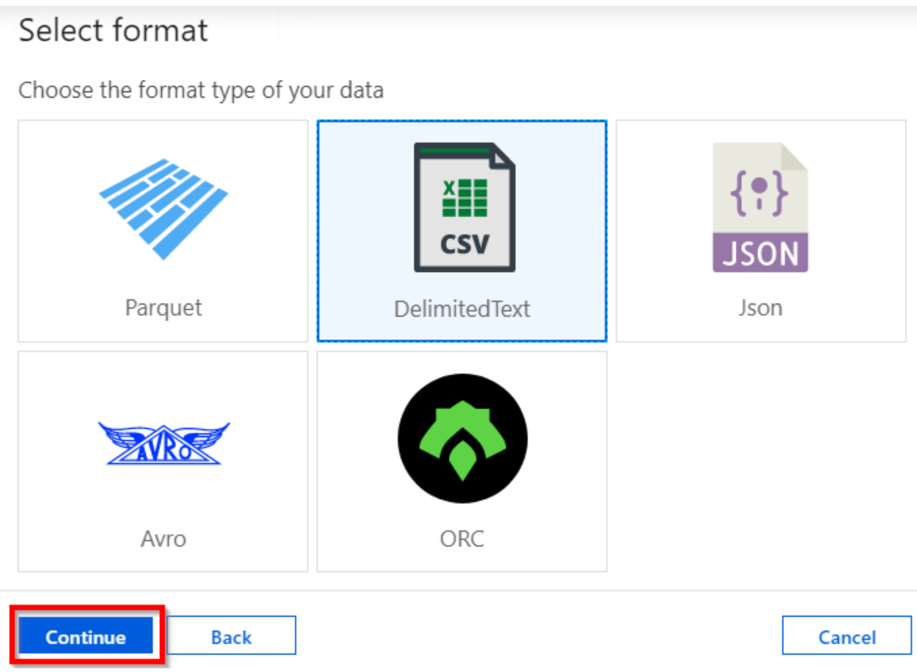


Figure 3.59: ADF—Lookup activity—new source dataset—choosing the blob format

6. In the **Set properties** pop-up window, choose `AzureBlobStorage` under **Linked service** (which we created in the *Getting ready* section of this recipe) and click on the **Browse** button, as shown in *Figure* 3.60:

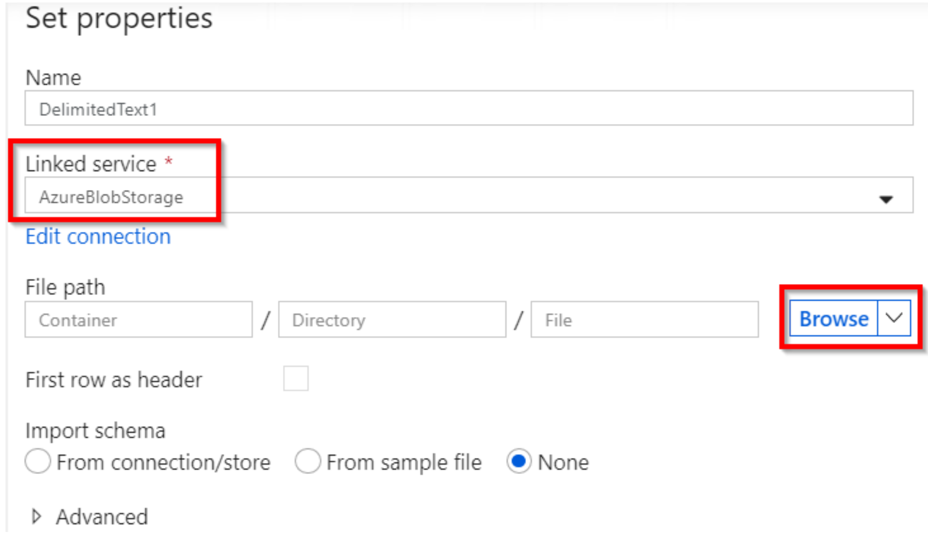


Figure 3.60: ADF—Lookup activity—new source dataset—Set properties

7. In the **Choose a file or folder** pop-up window, double-click on the blob container:

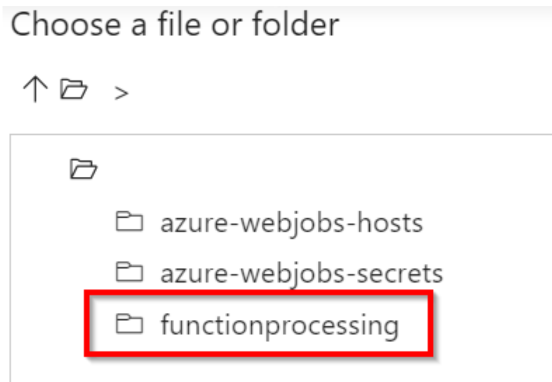


Figure 3.61: ADF—Lookup activity—new source dataset—selecting the blob container

8. This opens up all the blobs in which the CSV files reside, as shown in *Figure 3.62*. Once you have chosen the blob, click on the **OK** button:

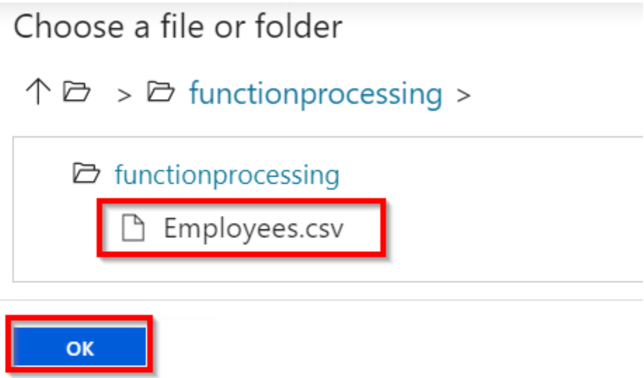


Figure 3.62: ADF—Lookup activity—new source dataset—selecting the blob

9. You'll be taken back to the **Set properties** pop-up window. Click on the **OK** button to create the dataset.

10. Once it is created, navigate to the dataset and mark the **First row as header** checkbox, as shown in *Figure 3.63*:

Figure 3.63: ADF—Lookup activity—new source dataset—First row as header checkbox

11. Now, the **Lookup** activity's **Setting** blade should look something like this:

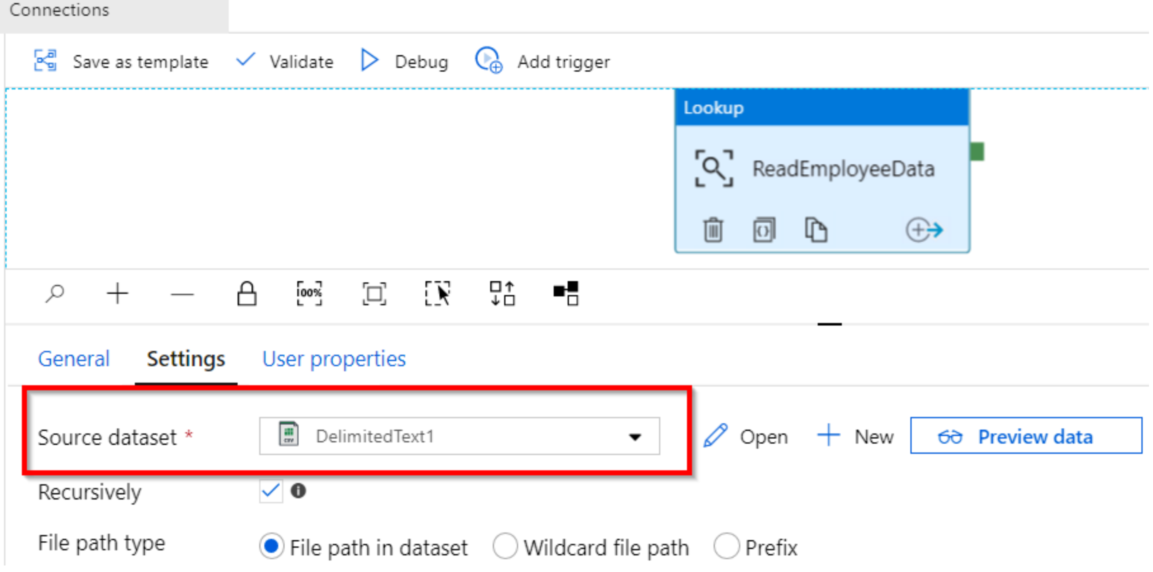


Figure 3.64: ADF—Lookup activity—selecting Source dataset

12. Drag and drop the **ForEach** activity to the pipeline and change its name to **`SendMailsForLoop`**, as shown in *Figure* 3.65:

pipeline1
Connections
Activities
Search activities
Move & transform
Azure Data Explorer
Azure Function
Batch Service
Databricks
Data Lake Analytics
General
HDInsight
Iteration & conditionals
Filter
ForEach
If Condition
Save as template
Validate
Debug
Add trigger
Lookup
ReadEmployeeData
ForEach
SendMailForLoop
Activities
No activities
General
Settings
Activities (0)
User properties
Name *
SendMailForLoop
Learn more
Description

Figure 3.65: ADF—creating a ForEach activity

13. Now, drag the green box that is available in the right-hand side of the **Lookup** activity and drop it on the **ForEach** activity, as shown in *Figure* 3.66, to connect them:

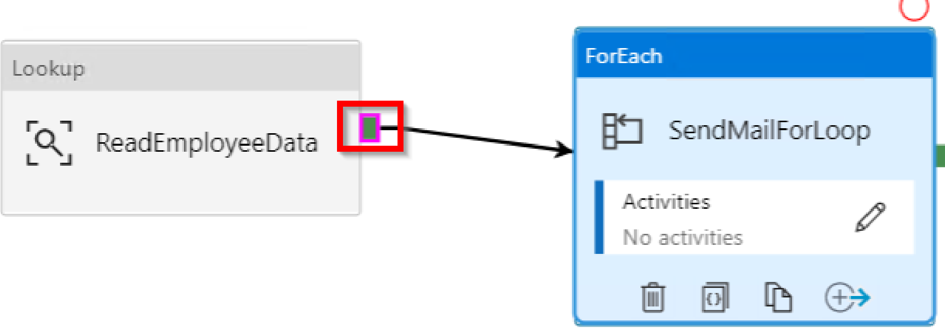


Figure 3.66: ADF—linking the Lookup and ForEach activities

14. Once the **Lookup** activity and the **ForEach** activity are connected, the **Lookup** activity can send the data to the **ForEach** activity as a parameter. In order to receive the data by the **ForEach** activity, go to the **Settings** section of the **ForEach** activity and click on the **Add dynamic content** option, available below the **Items** field as shown in *Figure* 3.67:

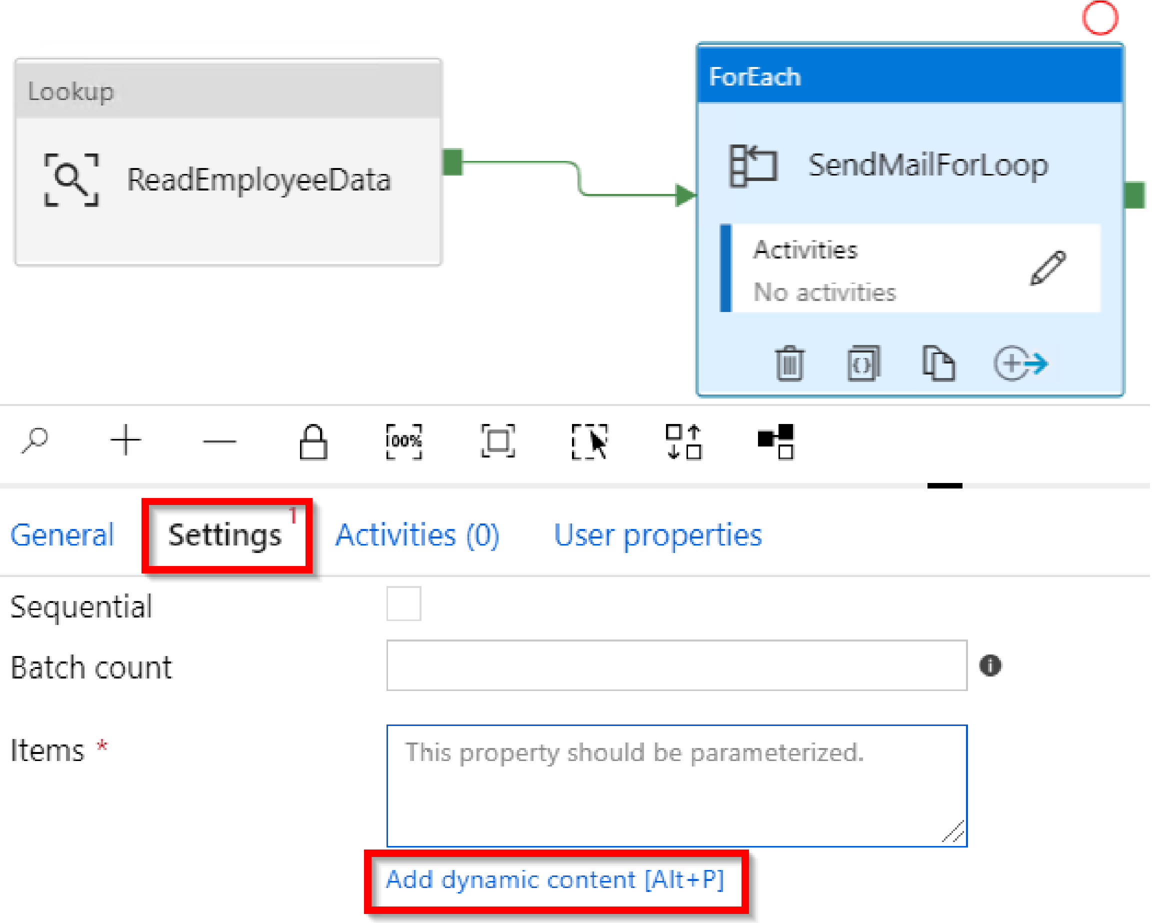


Figure 3.67: ADF—ForEach activity settings

15. In the **Add dynamic content** pop-up window, click the `ReadEmployeeData` activity output, which adds `@activity('ReadEmployeeData')` output to the text box. Now, append a value by typing `.value`, as shown in *Figure* 3.68, and click on the **Finish** button:

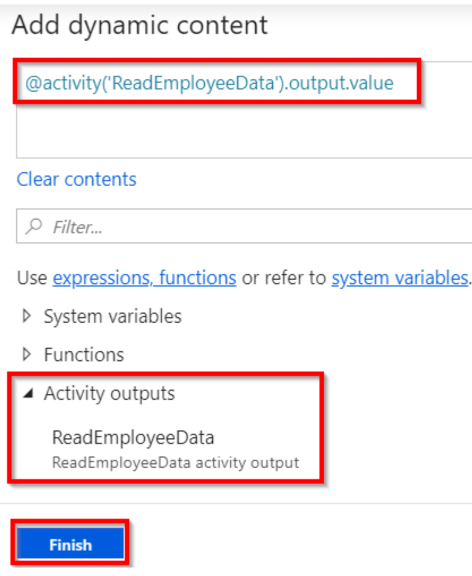


Figure 3.68: ADF—ForEach activity settings—choosing the output of the lookup activity

16. You should see something similar to what is shown in *Figure* 3.69 in the **Items** text box:

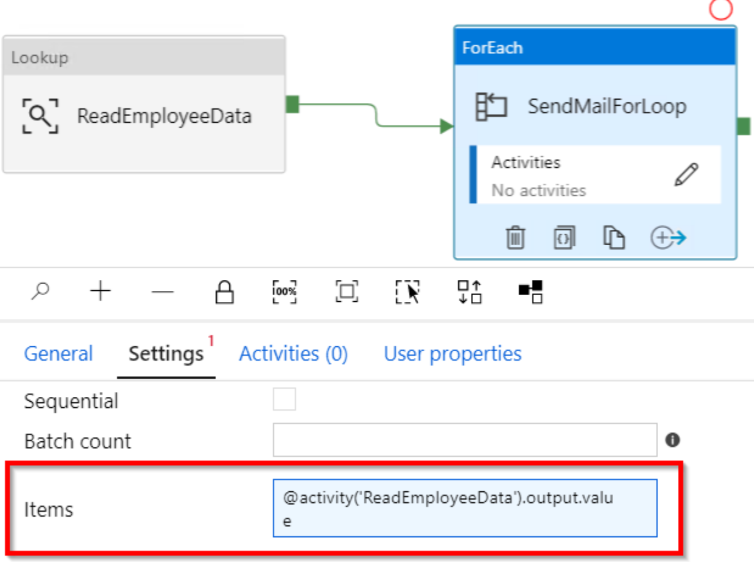


Figure 3.69: ADF—ForEach activity settings—configured input

17. Let's now click on the pen icon, which is available inside the **ForEach** activity as shown in *Figure* 3.70:

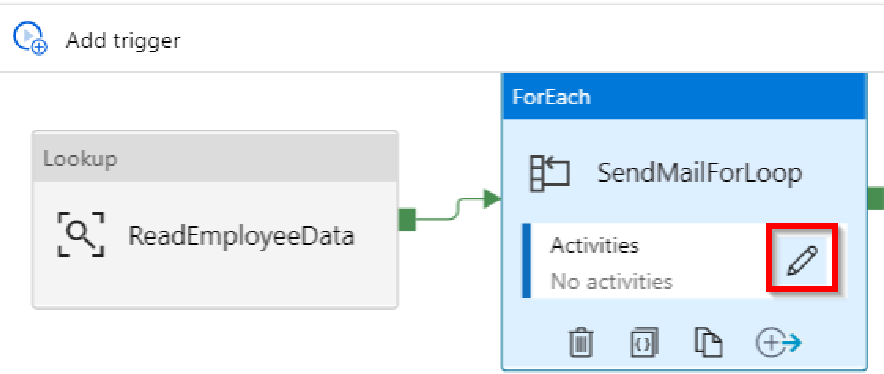


Figure 3.70: ADF—ForEach activity—editing

18. Drag and drop the **Azure Function** activity to the pipeline and change its name to **SendMail**, as shown in *Figure* 3.71, and click on the **Settings** button:

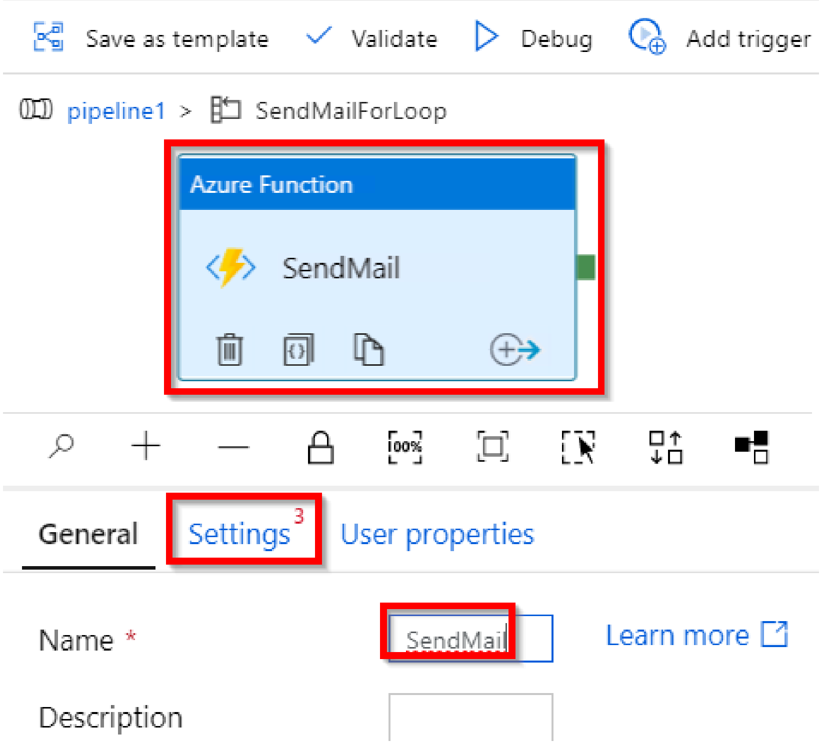


Figure 3.71: ADF—ForEach activity—adding a function activity

19. In the **Settings** tab, choose the `AzureFunction` linked service that is created in the *Getting ready* section of this recipe and also choose the **Function name** option, as shown in *Figure 3.72*:

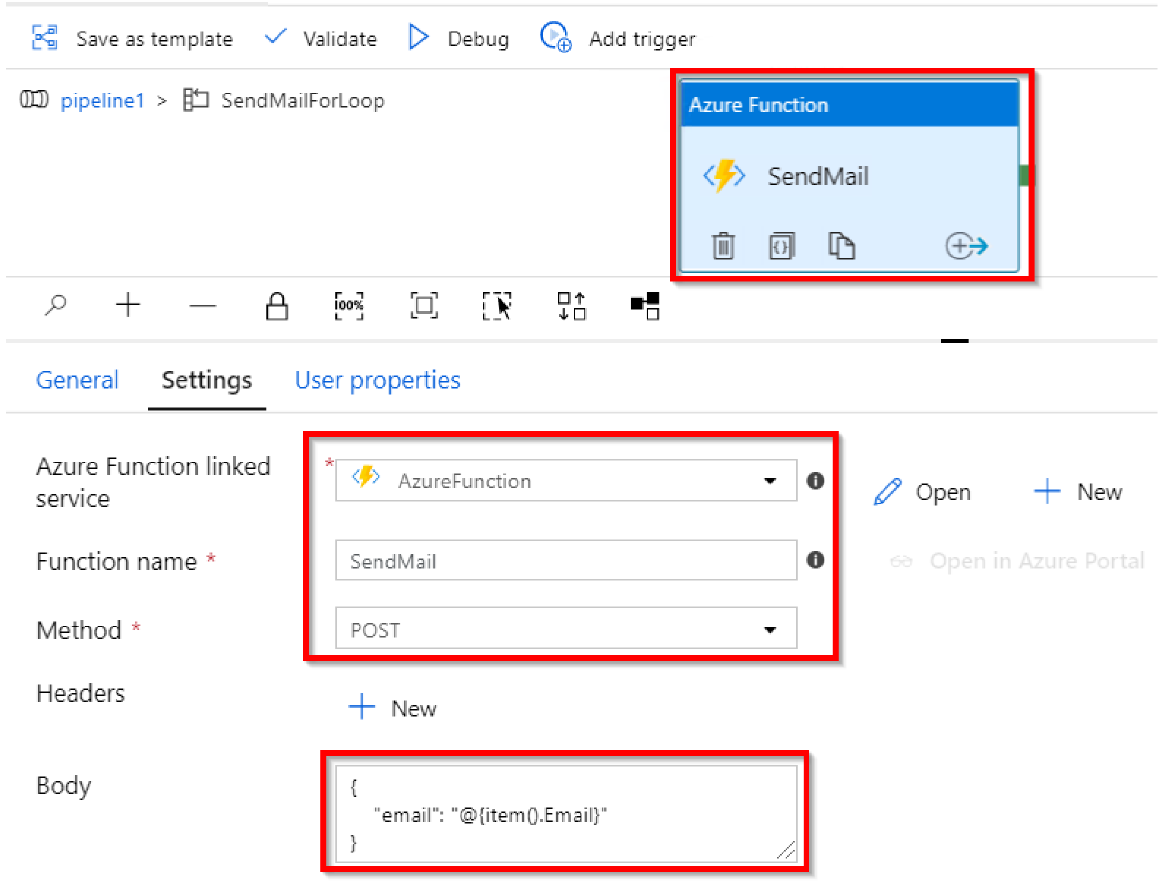


Figure 3.72: ADF—ForEach activity—passing inputs to the function activity

20. As shown in *Figure 3.72*, you need to provide the input to the Azure function named **SendMail**, which receives email as input. The expression provided in the **Body** field is called an ADF expression. Learn more about these at https://docs.microsoft.com/azure/data-factory/control-flow-expression-language-functions.

21. Now, click on the **Publish all** button to save the changes.

22. Once the changes are published, click on **Add trigger** and then the **Trigger now** button, as shown in *Figure* 3.73:

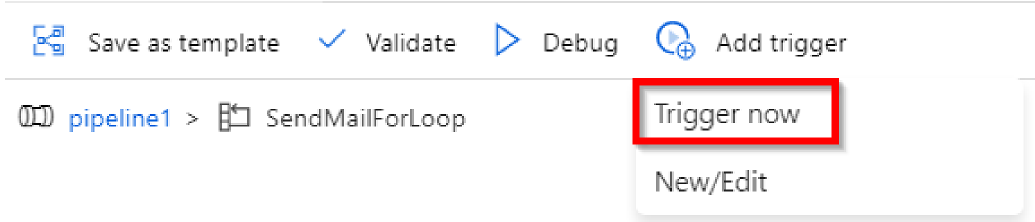


Figure 3.73: ADF—running the pipeline

23. A new pop-up window will appear, as shown in *Figure* 3.74. Click on **OK** to start running the pipeline:

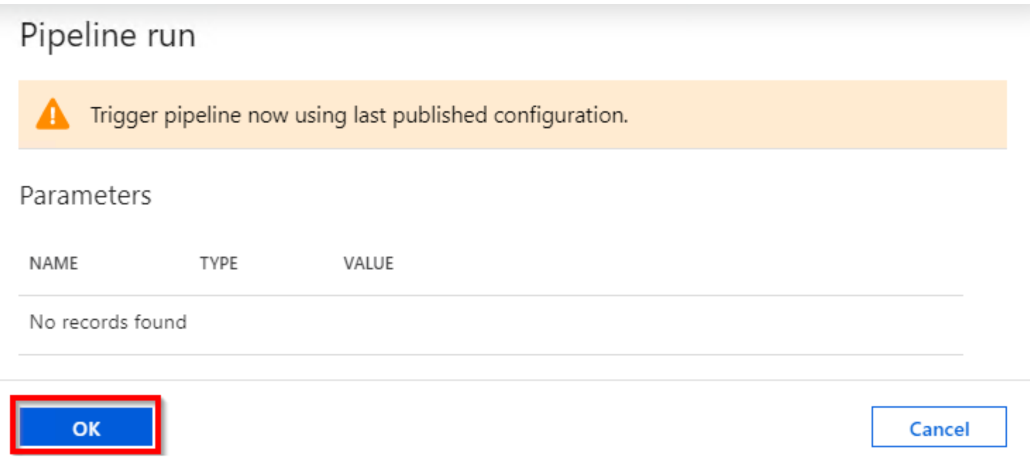


Figure 3.74: ADF—Pipeline run parameters

24. Click **OK** and immediately navigate to the Azure function and view the logs, as shown in *Figure 3.75*, to see the inputs received from the Data Factory instance:

```
Logs   Console                                        Reconnect  Copy logs  Pause  Clear  Collaps
2020-04-26T13:29:02.077 [Information] Executing 'Functions.SendMail' (Reason='This function was programmatically called via the host APIs.', Id=1965e7ef-fe41-4fec-b047-40440c0b7e23)
2020-04-26T13:29:02.077 [Information] C# HTTP trigger function processed a request.
2020-04-26T13:29:02.078 [Information] The input recieved by this e-mail Bhargavi@gmail.com
2020-04-26T13:29:02.078 [Information] Executed 'Functions.SendMail' (Succeeded, Id=1965e7ef-fe41-4fec-b047-40440c0b7e23)
2020-04-26T13:29:02.082 [Information] Executing 'Functions.SendMail' (Reason='This function was programmatically called via the host APIs.', Id=93aef44a-e990-4bf9-a8e2-6df90acddcfe)
2020-04-26T13:29:02.082 [Information] C# HTTP trigger function processed a request.
2020-04-26T13:29:02.082 [Information] The input recieved by this e-mail Khadir@gmail.com
2020-04-26T13:29:02.086 [Information] Executed 'Functions.SendMail' (Succeeded, Id=93aef44a-e990-4bf9-a8e2-6df90acddcfe)
2020-04-26T13:29:02.147 [Information] Executing 'Functions.SendMail' (Reason='This function was programmatically called via the host APIs.', Id=34b33677-62d9-42e6-95c4-a111daf28b88)
2020-04-26T13:29:02.147 [Information] C# HTTP trigger function processed a request.
2020-04-26T13:29:02.147 [Information] The input recieved by this e-mail meena@gmail.com
2020-04-26T13:29:02.147 [Information] Executed 'Functions.SendMail' (Succeeded, Id=34b33677-62d9-42e6-95c4-a111daf28b88)
2020-04-26T13:29:02.152 [Information] Executing 'Functions.SendMail' (Reason='This function was programmatically called via the host APIs.', Id=73e99649-8522-47ab-878b-f30684c6a783)
2020-04-26T13:29:02.152 [Information] C# HTTP trigger function processed a request.
2020-04-26T13:29:02.152 [Information] The input recieved by this e-mail praveen@gmail.com
2020-04-26T13:29:02.152 [Information] Executed 'Functions.SendMail' (Succeeded, Id=73e99649-8522-47ab-878b-f30684c6a783)
2020-04-26T13:29:02.156 [Information] Executing 'Functions.SendMail' (Reason='This function was programmatically called via the host APIs.', Id=3eb29784-030a-4950-b128-3d28e73de374)
2020-04-26T13:29:02.156 [Information] C# HTTP trigger function processed a request.
2020-04-26T13:29:02.156 [Information] The input recieved by this e-mail Nischala@gmail.com
2020-04-26T13:29:02.156 [Information] Executed 'Functions.SendMail' (Succeeded, Id=3eb29784-030a-4950-b128-3d28e73de374)
2020-04-26T13:29:02.164 [Information] Executing 'Functions.SendMail' (Reason='This function was programmatically called via the host APIs.', Id=09bdb96c-dc4d-4db2-8ed1-d10bf4ea965d)
2020-04-26T13:29:02.165 [Information] C# HTTP trigger function processed a request.
2020-04-26T13:29:02.165 [Information] The input recieved by this e-mail vivek@gmail.com
2020-04-26T13:29:02.165 [Information] Executed 'Functions.SendMail' (Succeeded, Id=09bdb96c-dc4d-4db2-8ed1-d10bf4ea965d)
```

Figure 3.75: ADF—Azure Functions—console logs

That's it! You have learned how to integrate Azure Functions as an activity inside the ADF pipeline.

The next step is to integrate the functionality of sending an email to the end user based on the input received. These steps have already been discussed in the *Sending an email notification dynamically to the end user* recipe in *Chapter 2*, *Working with notifications using the SendGrid and Twilio services*.

You can also monitor the progress of the pipeline execution by clicking on the **Monitor** tab, as shown in *Figure 3.76*:

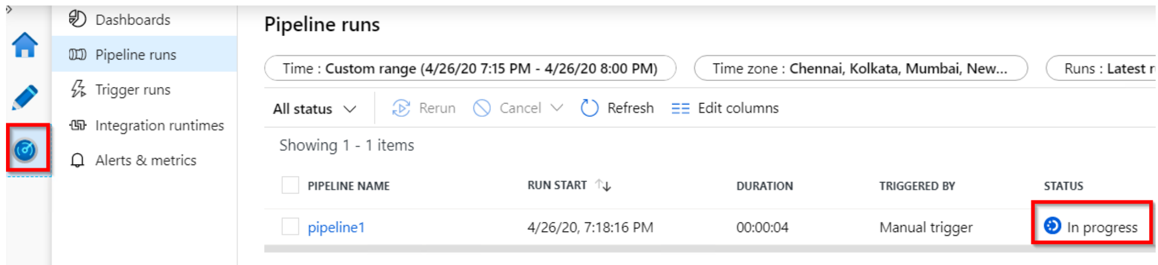


Figure 3.76: ADF—monitoring the pipeline

Click on the pipeline name to view detailed progress, as shown in *Figure 3.77*:

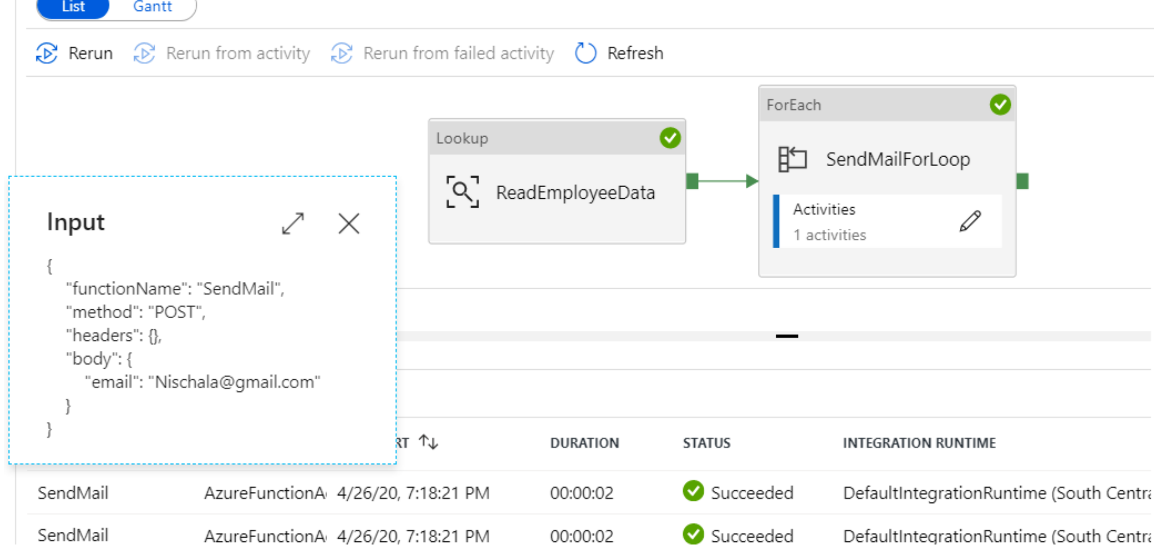


Figure 3.77: ADF—monitoring individual activities

In this recipe, you have learned how to integrate Azure Functions as an activity in an ADF pipeline.

In this chapter, you have learned how to integrate Azure Functions with various Azure services, including Cognitive Services, Logic Apps, and Data Factory.

4

Developing Azure Functions using Visual Studio

In this chapter, we'll cover the following:

- Creating a function application using Visual Studio 2019
- Debugging Azure Function hosted in Azure using Visual Studio
- Connecting to the Azure Storage from Visual Studio
- Deploying the Azure Function application using Visual Studio
- Debugging Azure Function hosted in Azure using Visual Studio
- Deploying Azure Functions in a container

Introduction

In previous chapters, you learned how to create Azure Functions right from the Azure Management portal. Here are a few of the features that we encountered:

- We can quickly create a function just by selecting one of the built-in templates provided by the Azure Functions runtime.
- Developers need not worry about writing plumbing code or understanding how to work with the frameworks.
- Configuration changes can be made right within the UI using the standard editor.

Despite the advantages provided by the Azure Management portal, moving over from a familiar **integrated development environment** (**IDE**) to something new can prove to be a daunting task for developers. To ease this transition, the Microsoft team has come up with a few tools that help developers to integrate Azure Functions into Visual Studio, with the aim of leveraging critical IDE features that are imperative for accelerating development efforts. Here are a few of the features:

- Developers benefit from IntelliSense support.
- The ability to debug code line by line.
- The values of variables can be quickly viewed while debugging the application.
- Integration with version control systems such as Azure DevOps (formerly known as **Visual Studio Team Services** (**VSTS**)).

You'll learn about some of the preceding features in this chapter, and see how to integrate code with Azure DevOps in *Chapter 12*, *Implementing and deploying continuous integration using Azure DevOps*.

Creating a function application using Visual Studio 2019

In this recipe, you will learn how to create an Azure function in Visual Studio 2019 with the latest available Azure Functions runtime. You'll also discover how to provide access to anonymous users.

Getting ready

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

- Download the latest version of Visual Studio 2019, which can be found here: https://visualstudio.microsoft.com/downloads/
- During the installation, choose **Azure development** in the **Workloads** section and then click on the **Install** button.

How to do it...

In this section, you'll create a function application and a HTTP function using Visual Studio by performing the following steps:

1. Open Visual Studio, choose **Create a new project**, select **Azure** in the platform dropdown, and then choose the **Azure Functions** template. Once you are ready, click on **Next**, as shown in *Figure 4.1*:

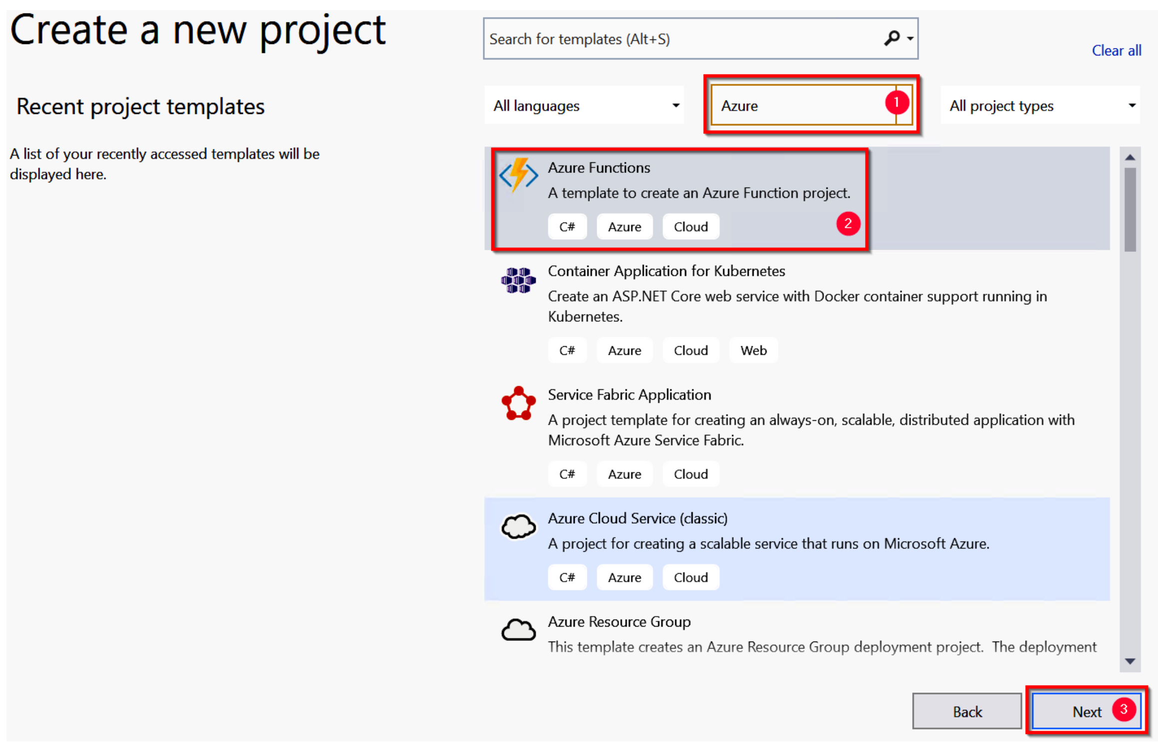


Figure 4.1: Create a new project

2. Now, you need to provide a name for the function application. Click on the **Create** button to go to the next step. As shown in *Figure 4.2*, choose **Azure Functions v3 (.NET Core)** from the drop-down menu, then select **Http trigger** with **Anonymous** in the **Authorization level** dropdown, and click on the **Create** button:

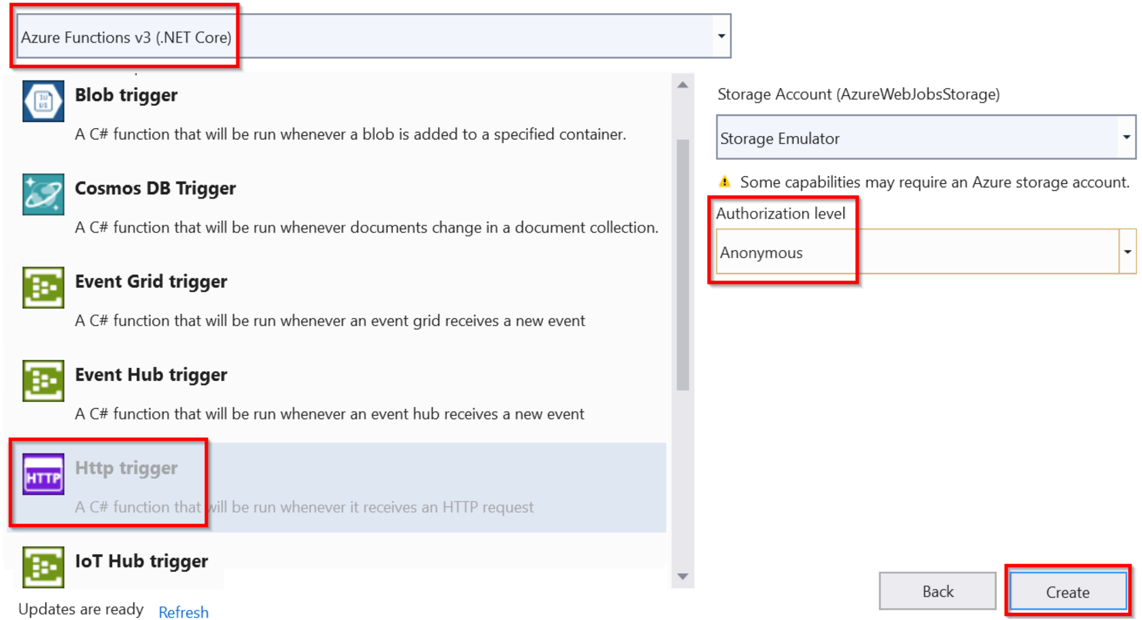


Figure 4.2: Create a new Azure Functions Application

3. You have successfully created the Azure Function application, 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 application, and also be sure to build the application to download the required NuGet packages.

4. After you create a new function, a new class will also be created, as shown in *Figure 4.3*:

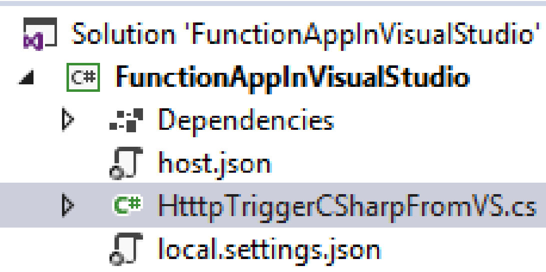


Figure 4.3: Azure Functions Solution Explorer

You have now successfully created a new HTTP triggered function application using Visual Studio 2019.

How it works...

As explained earlier, Visual Studio tools for Azure Functions allow developers to use their preferred IDE, which they may have been using for years. Using the tools of Azure Functions, we can use the same set of templates that the Azure Management portal provides in order to quickly create Azure Functions and integrate them with cloud services without writing any (or minimal) plumbing code.

The other advantage of using Visual Studio tools for Functions is that we don't need to have a live Azure subscription. We can debug and test Azure Functions right in our local development environment. The Azure **command-line interface** (**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 in their local environment is that *everything works fine on their local machine but not in the production environment*. With Azure Functions, developers need not worry about this dilemma as the Azure Functions runtime provided by the Azure CLI tools is exactly the same as the runtime available on the Azure cloud.

> **Note**
>
> We can always use and trigger an Azure service running on the cloud, even when we are developing Azure Functions locally.

Now that you understand how to create a function application using Visual Studio 2019, in the next recipe, you'll learn about debugging C# Azure Functions.

Debugging Azure Function hosted in Azure using Visual Studio

Once the basic setup of your Function creation is complete, the next step is to start working on developing the application as per your needs. Developers end up facing numerous technical issues that require tools to identify the root cause of the problem and fix it. These tools include debugging tools that help developers to step into each line of the code to view the values of the variables and objects and get a detailed view of the exceptions.

Getting ready

Download and install the Azure CLI (if these tools are not installed, Visual Studio will automatically download them when you run your functions from Visual Studio).

How to do it...

In this section, you'll learn how to configure and debug an Azure function in a local development environment within Visual Studio.

Perform the following steps:

1. In the previous recipe, you created the HTTP trigger 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 *Figure 4.4*:

```
HttpTriggerCSharpFromVS.cs
FunctionAppinVisualStudio    FunctionAppinVisualStudio.HttpTriggerCSharpFromVS    Run(HttpReques
10
11  namespace FunctionAppinVisualStudio
12  {
        0 references
13      public static class HttpTriggerCSharpFromVS
14      {
15          [FunctionName("HttpTriggerCSharpFromVS")]
            0 references
16          public static async Task<IActionResult> Run(
17              [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRequest req,
18              ILogger log)
19          {
20              log.LogInformation("C# HTTP trigger function processed a request.");
21
22              string name = req.Query["name"];
23
24              string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
25              dynamic data = JsonConvert.DeserializeObject(requestBody);
26              name = name ?? data?.name;
27
28              return name != null
29                  ? (ActionResult)new OkObjectResult($"Hello, {name}")
30                  : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
31          }
32      }
33  }
class System.String
Represents text as a sequence of UTF-16
```

Figure 4.4: The HTTP trigger function code

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

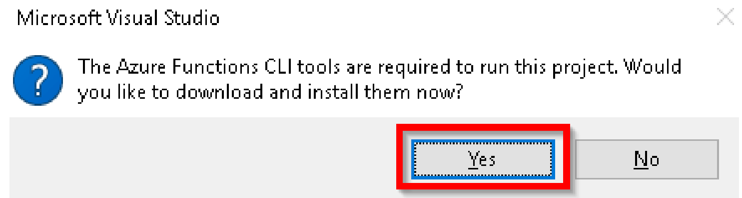


Figure 4.5: Azure Functions core tool installation

> **Note**
>
> The Azure Functions CLI has now been renamed Azure Functions Core Tools. Find out more about them at https://www.npmjs.com/package/azure-functions-core-tools.

4. Clicking on **Yes**, as shown in *Figure* 4.5, will start downloading the CLI tools. The download and installation of the CLI tools will take a few minutes.
5. Once the **Azure Functions CLI tools** have been installed successfully, a job host will be created and started. It will start monitoring requests on a specific port for all the functions of your function application. *Figure* 4.6 shows that the job host has started monitoring the requests to the function application:

```
C:\Users\vmadmin\AppData\Local\AzureFunctionsTools\Releases\3.3.1\cli_x64\func.exe
[2/18/2020 1:26:39 PM]   "MaxOutstandingRequests": -1,
[2/18/2020 1:26:39 PM]   "RoutePrefix": "api"
[2/18/2020 1:26:39 PM] }
[2/18/2020 1:26:39 PM] Starting JobHost
[2/18/2020 1:26:39 PM] Starting Host (HostId=vm2017-314325740, InstanceId=98859ded-ba3e-4489-a1b8-127953081ce3, Version=3.0.13107, ProcessId=15304, AppDomainId=1, InDebugMode=False, InDiagnosticMode=False, FunctionsExtensionVersion=(null))
[2/18/2020 1:26:39 PM] Loading functions metadata
[2/18/2020 1:26:39 PM] 1 functions loaded
[2/18/2020 1:26:40 PM] Generating 1 job function(s)
[2/18/2020 1:26:40 PM] Found the following functions:
[2/18/2020 1:26:40 PM] FunctionAppinVisualStudio.HttpTriggerCSharpFromVS.Run
[2/18/2020 1:26:40 PM]
[2/18/2020 1:26:40 PM] Initializing function HTTP routes
[2/18/2020 1:26:40 PM] Mapped function route 'api/HttpTriggerCSharpFromVS' [get,post] to 'HttpTriggerCSharpFromVS'
[2/18/2020 1:26:40 PM]
[2/18/2020 1:26:40 PM] Host initialized (615ms)
[2/18/2020 1:26:40 PM] Host started (627ms)
[2/18/2020 1:26:40 PM] Job host started
Hosting environment: Production
Content root path: C:\Users\vmadmin\source\repos\FunctionAppinVisualStudio\FunctionAppinVisualStudio\bin\Debug\netcoreapp3.0
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

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

[2/18/2020 1:26:47 PM] Host lock lease acquired by instance ID '00000000000000000000000073EC2A43'.
```

Figure 4.6: The Azure function job host

6. Let's try to access the function application by making a request to **`http://localhost:7071`** in any web browser:

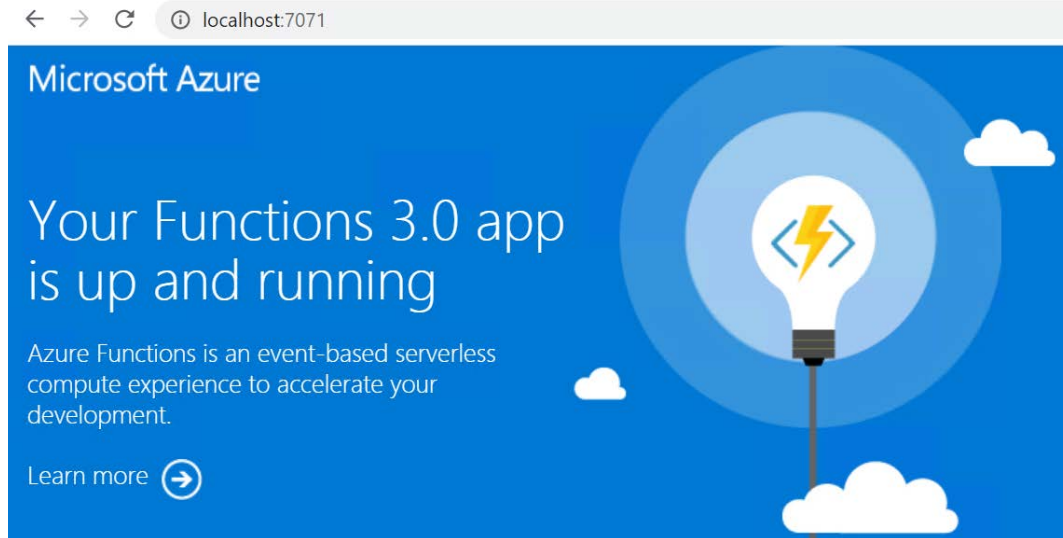


Figure 4.7: The Azure Functions 3.0 default web page

Now, type the complete URL of your HTTP trigger in the browser. The URL should look like this:

`http://localhost:7071/api/HttpTriggerCsharpFromVS?name=Praveen Sreeram.`

7. After entering the correct URL of the Azure function, as soon as you hit the *Enter* key in the address bar of the browser, the Visual Studio debugger will hit the debugging point (if you have one), as shown in *Figure 4.8*:

```
HttpTriggerCSharpFromVS.cs
FunctionAppinVisualStudio    FunctionAppinVisualStudio.HttpTriggerCSharpFrom    Run(HttpRequest req, ILogger log)
10
11  namespace FunctionAppinVisualStudio
12  {
        0 references
13      public static class HttpTriggerCSharpFromVS
14      {
15          [FunctionName("HttpTriggerCSharpFromVS")]
            0 references
16          public static async Task<IActionResult> Run(
17              [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRequest req,
18              ILogger log)
19          {
20              log.LogInformation("C# HTTP trigger function processed a request.");
21
22              string name = req.Query["name"];
23
24              string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
```

Figure 4.8: The Azure Function HTTP trigger—creating a debug point

8. You can also view the data of your variables, as shown in *Figure 4.9*:

```
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];
                       name    "Praveen Sreeram"
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;
```

Figure 4.9: The Azure Function HTTP trigger—viewing variable values

9. Once you complete debugging, you can press the F5 key to complete the execution process, after which, you'll see the output response in the browser, as shown in *Figure 4.10*:

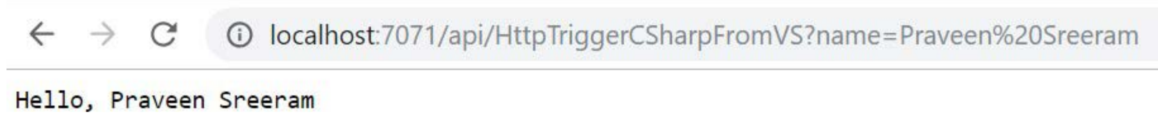


Figure 4.10: The HTTP trigger output

10. The function execution log will be seen in the job host console, as shown in *Figure 4.11*:

```
[2/18/2020 1:28:40 PM] Executing 'HttpTriggerCSharpFromVS' (Reason='This function was programmatically called via the ho
st APIs.', Id=f080505c-3d43-4f3a-b7fe-c2e7d3b3bbe7)
[2/18/2020 1:31:27 PM] C# HTTP trigger function processed a request.
[2/18/2020 1:32:18 PM] Executed 'HttpTriggerCSharpFromVS' (Succeeded, Id=f080505c-3d43-4f3a-b7fe-c2e7d3b3bbe7)
[2/18/2020 1:32:18 PM] Executed HTTP request: {
[2/18/2020 1:32:18 PM]   "requestId": "150dd18f-bb1a-4b82-a5d7-de6f3db5af6d",
[2/18/2020 1:32:18 PM]   "method": "GET",
[2/18/2020 1:32:18 PM]   "uri": "/api/HttpTriggerCsharpFromVS",
[2/18/2020 1:32:18 PM]   "identities": [
[2/18/2020 1:32:18 PM]     {
[2/18/2020 1:32:18 PM]       "type": "WebJobsAuthLevel",
[2/18/2020 1:32:18 PM]       "level": "Admin"
[2/18/2020 1:32:18 PM]     }
[2/18/2020 1:32:18 PM]   ],
[2/18/2020 1:32:18 PM]   "status": 200,
[2/18/2020 1:32:18 PM]   "duration": 218771
[2/18/2020 1:32:18 PM] }
```

Figure 4.11: The HTTP trigger execution log

You can add more Azure Functions to the function application, if required. In the next recipe, we'll 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 us with the following details:

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

There's more...

Using Visual Studio, we can directly create precompiled functions, which means that when we build our functions, Visual Studio creates a `.dll` file that can be referenced in other applications, just as we do for our 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.
- We can convert our traditional classes into Azure Functions easily, and refer to them in other applications seamlessly.

In this recipe, you have learned how to debug Azure Functions in the local development workstation. In the next recipe, you'll learn how to connect to a storage account available in Azure.

Connecting to the Azure Storage from Visual Studio

In both of the previous recipes, you learned how to create and execute Azure Functions in a local environment. You triggered the functions from a local browser. However, in this recipe, you'll 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 an Azure storage account, we can have our function triggered on our local machine. This helps developers to test their applications upfront, before deploying them to the production environment.

Getting ready

Perform the following steps:

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

How to do it...

In this section, you'll learn how to create a blob trigger that will trigger as soon as a blob is created in the storage account.

Perform the following steps:

1. Open the **`FunctionAppInVisualStudio`** Azure Function application in Visual Studio, and then add a new function by right-clicking on the **`FunctionAppInVisualStudio`** project. Click on **Add** | **New Azure Function**, which will open a pop-up window. Here, for the name field, enter **`BlobTriggerCSharp`** and then click on the **Add** button.
2. This will open another dialog box, where you can provide other parameters, as shown in *Figure 4.12*:

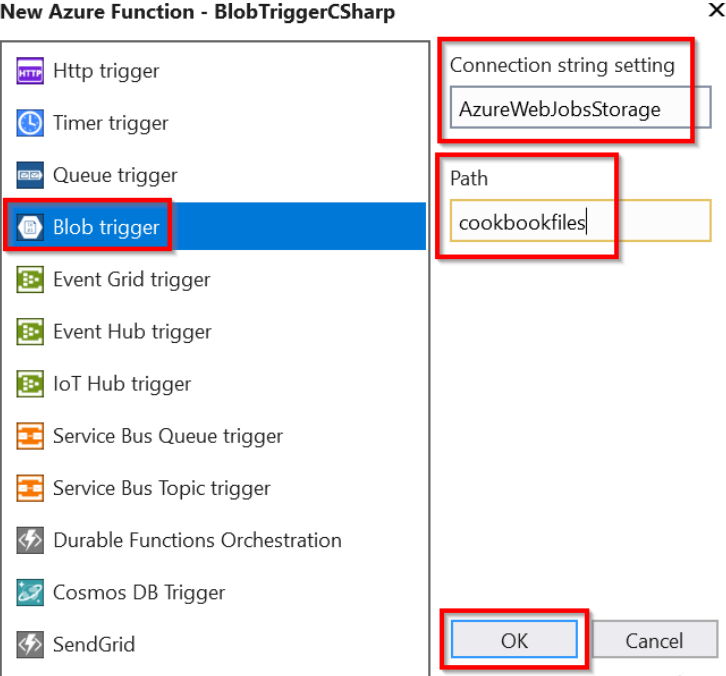


Figure 4.12: New Azure Function—HTTP trigger

3. In the **Connection string setting** field, provide **`AzureWebJobsStorage`** as the name of the connection string, and also provide the name of the blob container (in this case, it is **`cookbookfiles`**) in the **Path** input field, and then click on the **OK** button to create the new blob trigger function. A new blob trigger function will be created, as shown in *Figure 4.13*:

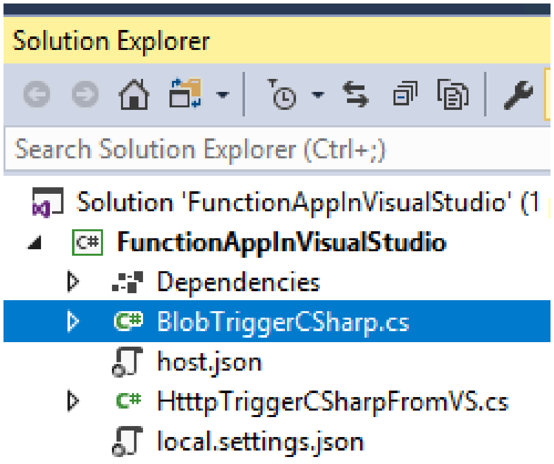


Figure 4.13: Azure Functions—Solution Explorer

4. As you learned in the *Building a back-end web API using HTTP triggers* recipe from *Chapter 1*, *Accelerating cloud app development using Azure Functions*, the Azure Management portal allows us to choose between a new or existing storage account. However, the preceding dialog box is not connected to our Azure subscription. So, let's 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 *Figure 4.14*:

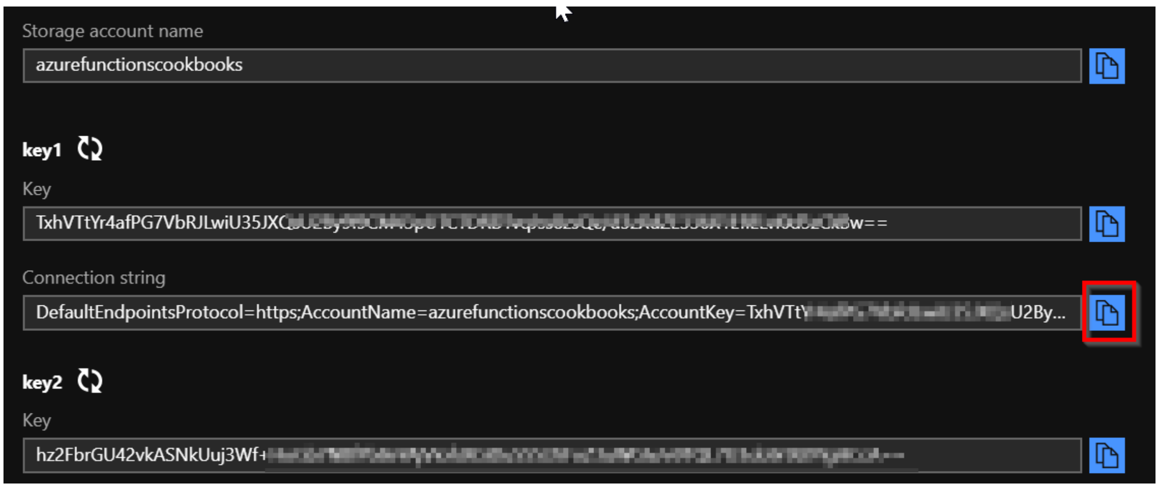


Figure 4.14: The storage account—the Access keys blade

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 application. Once you add the **Connection string** to the key named `AzureWebJobsStorage`, the `local.settings.json` file should look as shown in *Figure 4.15*:

```
{
    "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "DefaultEndpointsProtocol=https;AccountName=azurefunctioncookbooks
    "FUNCTIONS_WORKER_RUNTIME": "dotnet"
  }
}
```

Figure 4.15: Azure Functions—application settings

6. Open the `BlobTriggerCSharp.cs` file and create a breakpoint, as shown in *Figure 4.16*:

```
namespace FunctionAppinVisualStudio
{
    0 references
    public static class BlobTriggerCSharp
    {
        [FunctionName("BlobTriggerCSharp")]
        0 references
        public static void Run([BlobTrigger("cookbookfiles/{name}", Connection = "AzureWebJobsStorage")]Stream myBlob, string name, ILogger log)
        {
            log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");
        }
    }
}
```

Figure 4.16: The Azure Functions blob trigger—creating a breakpoint

7. Now, press the F5 key to start the job host, as shown in *Figure 4.17*:

```
Generating 2 job function(s)
Found the following functions:
FunctionAppinVisualStudio.BlobTriggerCSharp.Run
FunctionAppinVisualStudio.HttpTriggerCSharpFromVS.Run
```

Figure 4.17: The Azure Functions host log—generating functions

8. Let's add a new blob file using Azure Storage Explorer, as shown in *Figure 4.18*:

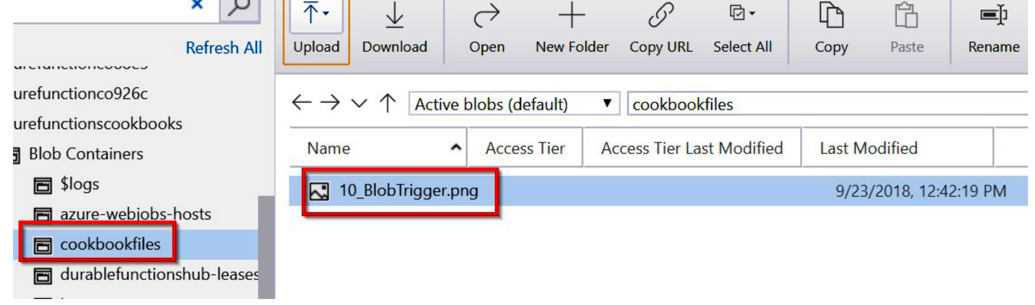


Figure 4.18: Storage Explorer

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 the local machine will detect that a new blob has been added and the debugger will hit the function, as shown in *Figure 4.19*:

```
7   namespace FunctionAppinVisualStudio
8   {
        0 references
9       public static class BlobTriggerCSharp
10      {
11          [FunctionName("BlobTriggerCSharp")]
            0 references
12          public static void Run([BlobTrigger("cookbookfiles/{name}", Connection = "AzureWebJobsStorage")]
13          {
14              log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.
15          }
16      }
17  }
18
```

Figure 4.19: Azure Functions blob trigger—breakpoint

That's it. You have learned how to trigger an Azure function in your local environment when an event occurs in Azure.

How it works...

In this **`BlobTriggerCSharp`** class, the **`Run`** method has the **`WebJobs`** attribute 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 account containers that you have specified. Whenever a new blob is added or updated, it automatically triggers the blob trigger in the current environment.

There's more...

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

> **Note**
>
> Learn more about WebJobs attributes at https://docs.microsoft.com/azure/app-service/webjobs-sdk-get-started.

In this recipe, you have learned how to create a blob trigger. In the next recipe, you'll learn how to deploy it to Azure.

Deploying the Azure Function application using Visual Studio

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

- The resource groups
- The App Service plan
- The Azure Function application

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

How to do it...

In this section, you'll learn how to deploy Azure Functions to Azure.

Perform the following steps:

1. Right-click on the project and then click on the **Publish** button to open the **Pick a publish target** dialog box.
2. In the **Pick a publish target** dialog box, choose the **Create New** option and click on the **Create Profile** button, as shown in *Figure 4.20*:

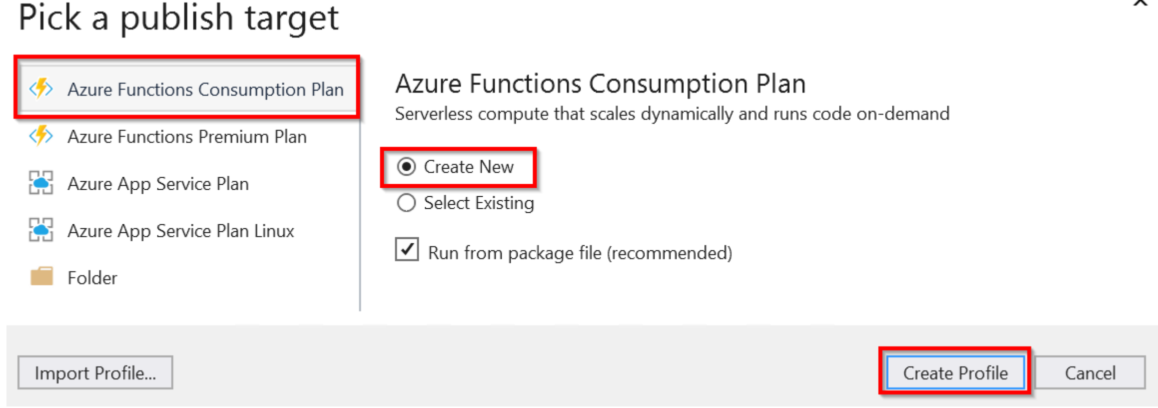


Figure 4.20: Visual Studio—Pick a publish target

3. In the **Create new 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 *Figure 4.21*:

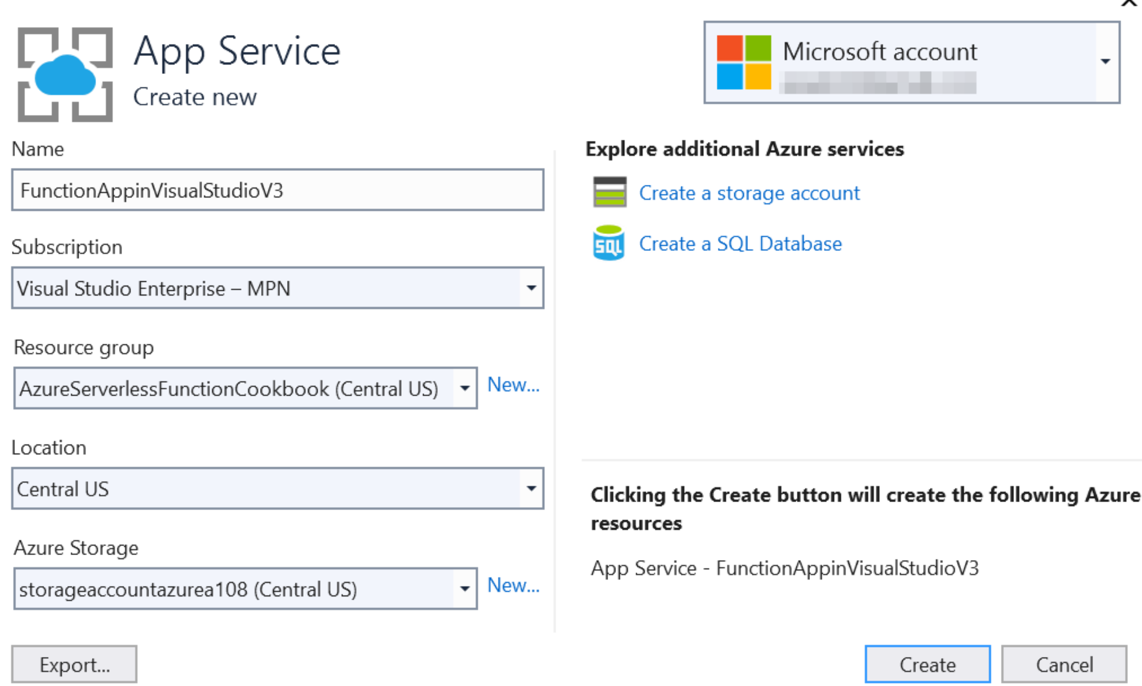


Figure 4.21: Visual Studio—creating a new App Service

4. After reviewing all the information, click on the **Create** button of the **Create new App Service** window. This should start deploying the services to Azure.

5. If everything goes well, you can view the newly created **Function App** in the Azure Management portal, as shown in *Figure 4.22*:

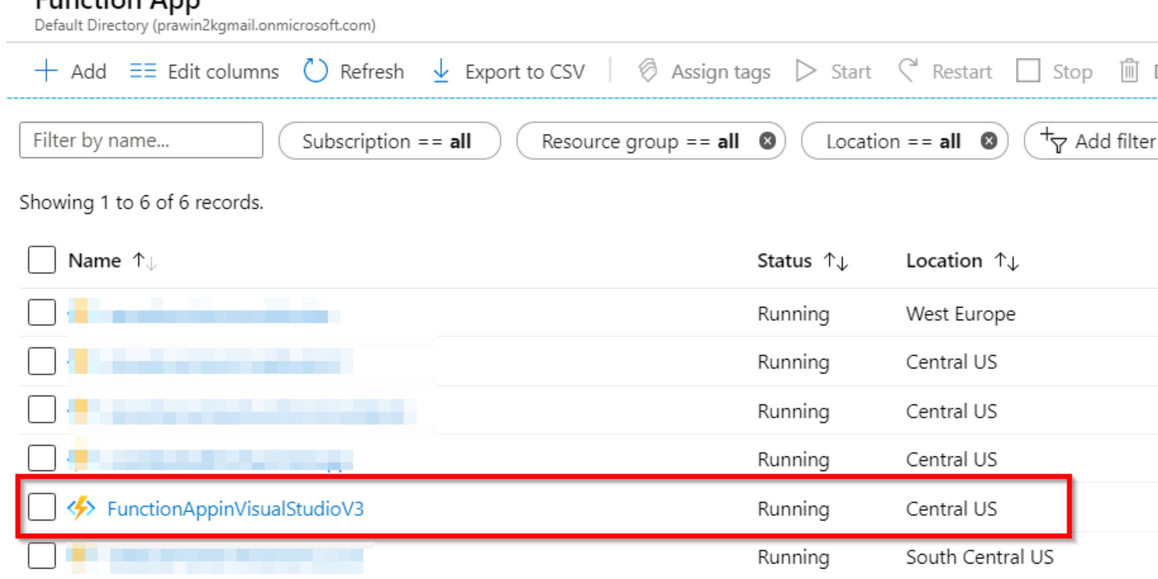


Figure 4.22: The function application listing

6. Hold on! Your job in Visual Studio is not yet done. You have just created the required services in Azure right from the Visual Studio IDE. Your next job is to publish the code from the local workstation to the Azure cloud. As soon as the deployment is complete, you'll be taken to the web deploy step, as shown in *Figure* 4.23. Click on the **Publish** button to start the process of publishing the code:

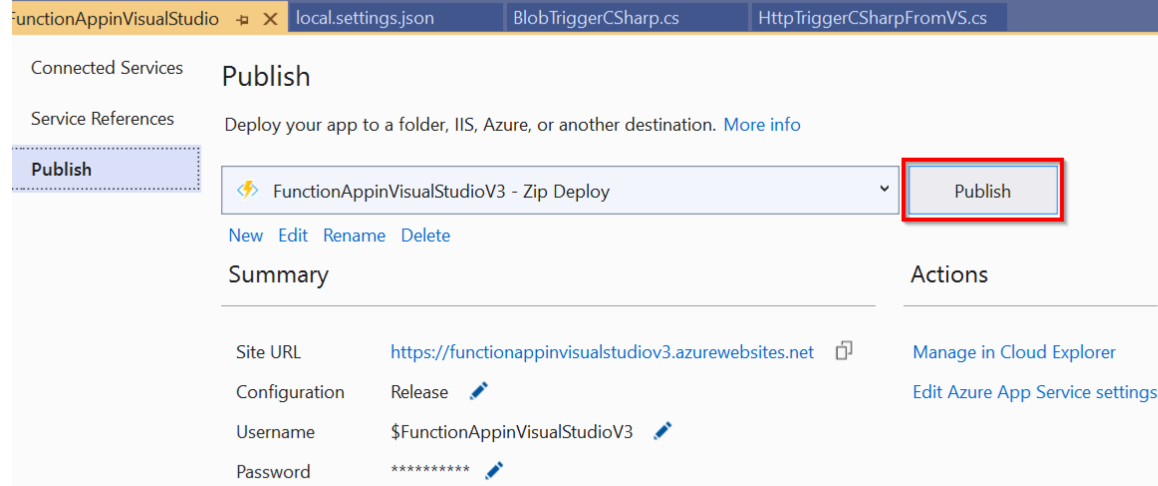


Figure 4.23: Visual Studio—Publish

7. That's it! You have completed the deployment of the Function application and its functions to Azure right from your preferred development IDE, Visual Studio. You can review function deployment in the Azure Management portal. Both Azure Functions were created successfully, as shown in *Figure 4.24*:

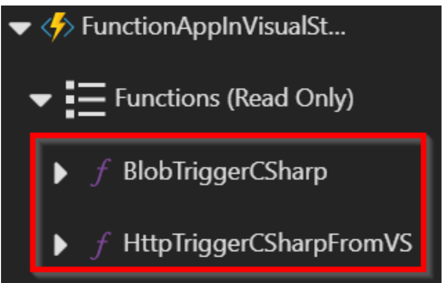


Figure 4.24: The function application—list

There's more...

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

In this recipe, we have deployed the Azure function to Azure. In the next recipe, you'll learn how to debug the Azure function from Visual Studio.

Debugging Azure Function hosted in Azure using Visual Studio

In one of the previous recipes, *Connecting to the Azure Storage from Visual Studio*, you learned how to connect a storage account from the local code. In this recipe, you'll learn how to debug the live code running in the Azure cloud environment. You'll perform the following steps in the **BlobTriggerCSharp** function of the **FunctionAppinVisualStudio** function application:

- Change the path of the container in the Azure Management portal to that of the new container.
- Open the function application in Visual Studio 2019.
- Attach the debugger from within Visual Studio 2019 to the required Azure function.
- Create a blob in the new storage container.
- Debug the application after the breakpoints are hit.

Getting ready

Create a container named **`cookbookfiles-live`** in the storage account. You'll be uploading a blob to this container.

How to do it...

In this recipe, you'll make the changes in Visual Studio that will let you debug the code hosted in Azure right from the local Visual Studio.

Perform the following steps:

1. Navigate to the **`BlobTriggerCSharp`** function in Visual Studio and change the path of the **`path`** variable to point to the new container, **`cookbookfiles-live`**:

```
[FunctionName("BlobTriggerCSharp")]
0 references
public static void Run([BlobTrigger("cookbookfiles-live/{name}",
{
```

Figure 4.25: The blob trigger function

2. Now, republish it by changing the configuration to **Debug | Any CPU**, as shown in *Figure 4.26*:

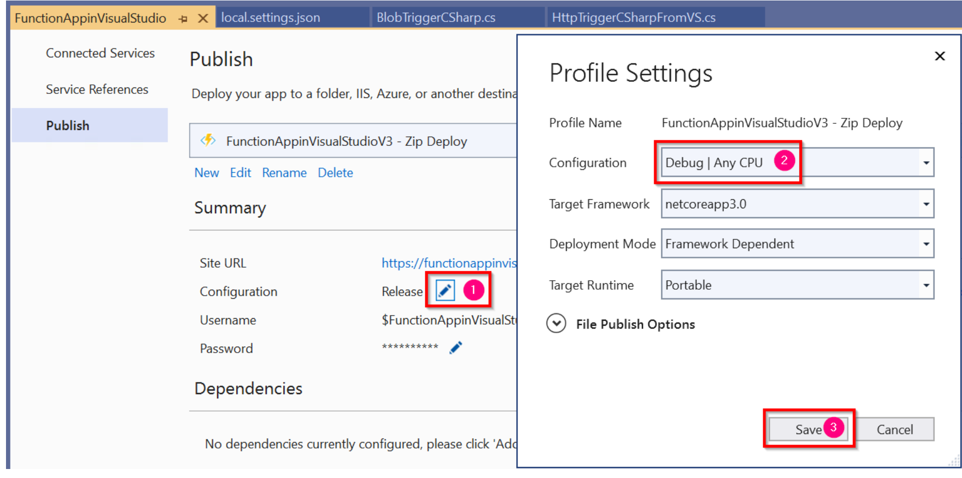


Figure 4.26: Visual Studio—Publish Profile Settings

> ### Note
>
> The preceding settings are to be used only in a non-production environment for testing. It's not recommended to deploy the package in **Debug** mode in your production environment. Once the testing is complete, you must republish the package in **Release** mode.

3. Once you publish it, the path of the container will look something like that shown in *Figure 4.27*:

FunctionAppinVisualStu...
- Functions (Read Only)
  - BlobTriggerCSharp
    - Integrate
    - Manage
    - Monitor
  - HttpTriggerCSharpFromVS
- Proxies (Read Only)
- Slots

```
1  {
2    "generatedBy": "Microsoft.NET.Sdk.Functions-3.0.1",
3    "configurationSource": "attributes",
4    "bindings": [
5      {
6        "type": "blobTrigger",
7        "connection": "AzureWebJobsStorage",
8        "path": "cookbookfiles-live/{name}",
9        "name": "myBlob"
10     }
11   ],
12   "disabled": false,
13   "scriptFile": "../bin/FunctionAppinVisualStudio.dll",
14   "entryPoint": "FunctionAppinVisualStudio.BlobTriggerCSharp.Run"
15 }
```

Figure 4.27: The blob trigger—Function.json

4. Open the function application in Visual Studio. Open **Cloud Explorer** in Visual Studio and navigate to your Azure function; in this case, it is `FunctionAppinVisualStudioV3`, as shown in *Figure 4.28*:

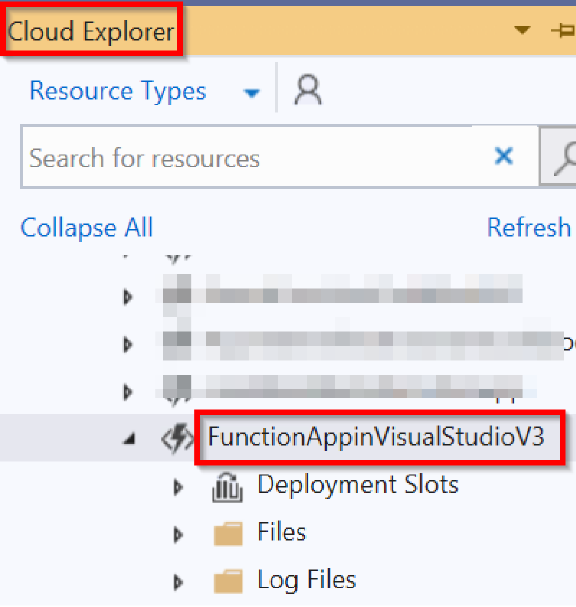


Figure 4.28: Visual Studio Cloud Explorer

5. Right-click on the function and click on **Attach Debugger**, as shown in *Figure* 4.29:

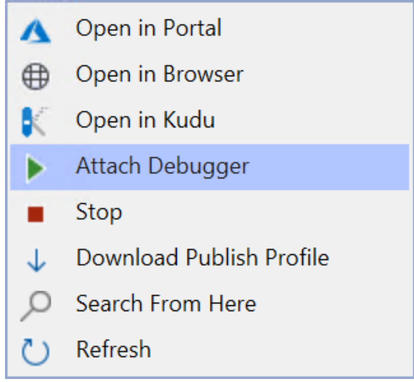


Figure 4.29: Clicking on Attach Debugger

6. Visual Studio will take some time to enable remote debugging, as shown in *Figure* 4.30:

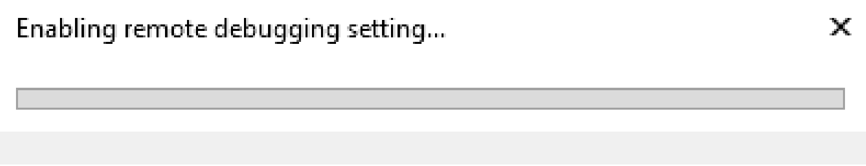


Figure 4.30: Visual Studio Cloud Explorer—enabling remote debugging

7. You can check whether the function application is working by opening it in the browser, as shown in *Figure* 4.31. This indicates that your function application is running as expected:

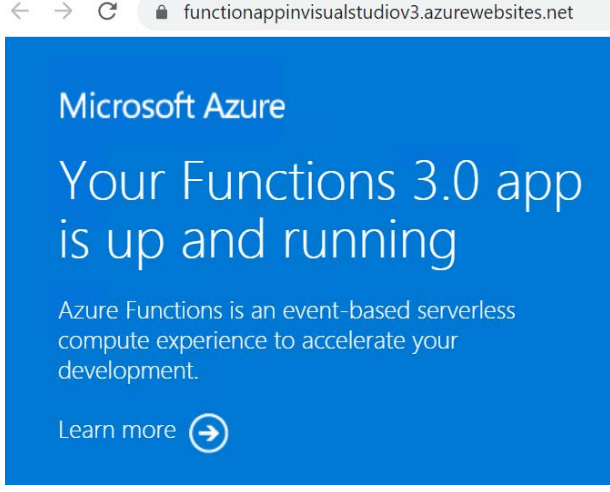


Figure 4.31: The function application default page

8. Navigate to **Storage Explorer** and upload a new file (in this case, I uploaded **`Employee.json`**) to the **`cookbookfiles-live`** container, as shown in *Figure* 4.32:

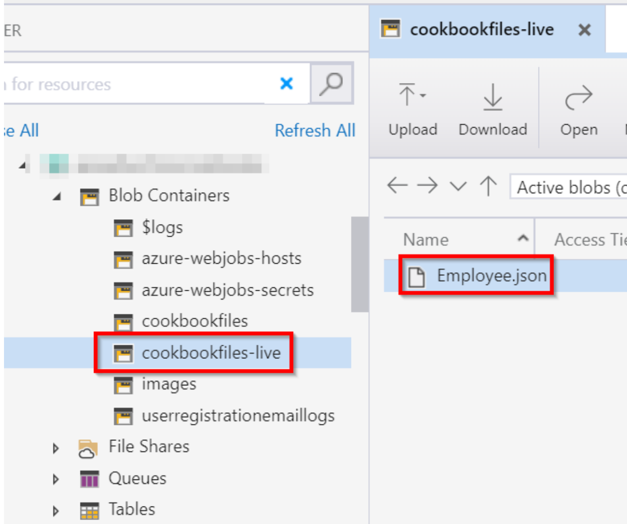


Figure 4.32: Uploading a new file

9. After a few moments, the debug breakpoint will be hit, as shown in *Figure* 4.33. You can also view the file name that has been uploaded:

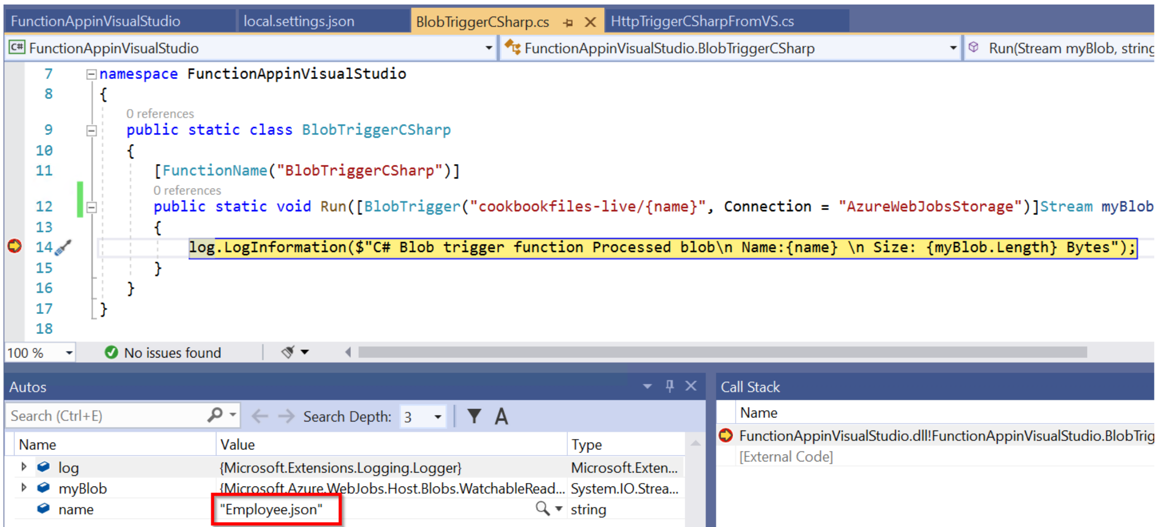


Figure 4.33: The blob trigger—the breakpoint hit

That's it. It's possible to debug the Azure Function application running in the cloud right from your IDE and you can also view the values of the variables.

In the next recipe, you'll learn how to deploy a function application as a Docker image.

Deploying Azure Functions in a container

You have now seen some of the major use cases for Azure Functions—in short, 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 to host server-side applications.

> **Note**
>
> You can take advantage of all the features of serverless (for example, autoscaling) only when you create your function application 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 will help. Yes, the combination of Azure Functions and Docker containers might not make sense, as we would lose all the serverless benefits (for example, autoscaling) of Azure Functions if we deployed to Docker.

However, there may be some customers whose existing workloads might be in a cloud (be it public or private), but now they want to leverage some of the Azure function triggers and related Azure services, and so they 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 the Azure CLI core tools from https://docs.microsoft.com/cli/azure/install-azure-cli?view=azure-cli-latest.
- Download Docker from https://hub.docker.com/editions/community/docker-ce-desktop-windows. Ensure that you install the version of Docker that is compatible with the `operating system (OS)` of your development environment.
- A basic knowledge of Docker and its commands is also required in order to build and run Docker images. You can go through the official Docker documentation https://docs.docker.com/ if you are not familiar with it.
- Create an Azure Container Registry (a registry to host Docker images in Azure) by performing the following steps. This can be used as a repository for all of the Docker images.

Creating an ACR

Azure Container Registry (**ACR**) is a service provided by Azure to host Docker images. It acts as a container repository. Let's create an ACR.

Perform the following steps:

1. Create a new ACR by searching for the container registry and providing the required details, as shown in *Figure* 4.34:

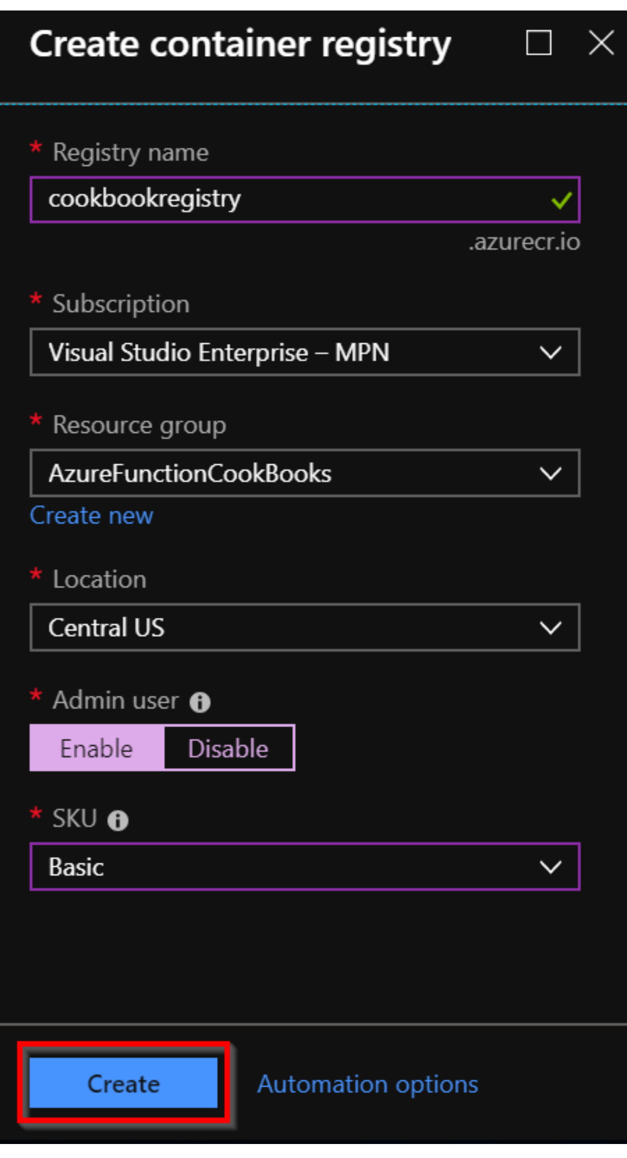


Figure 4.34: Azure Container Registry creation

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 *Figure* 4.35. You'll be using them later in this recipe:

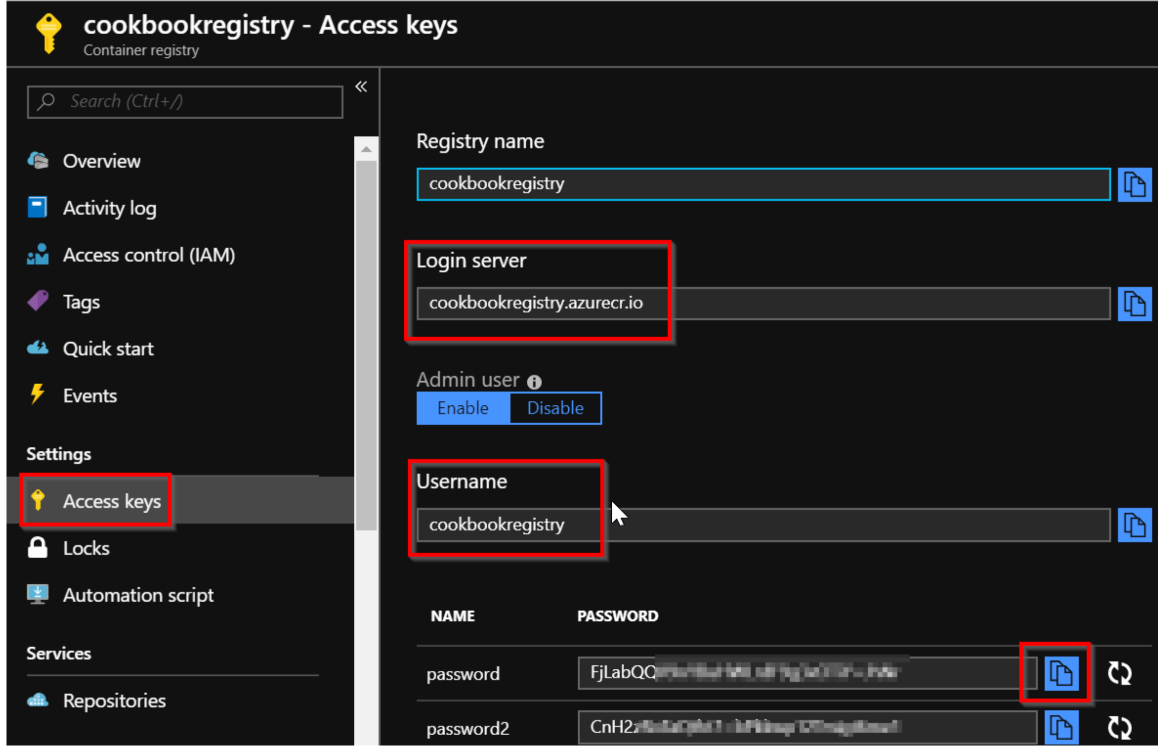


Figure 4.35: The Azure Container Registry—Access keys

Let's move on to the next section to learn how to deploy Azure Functions as a Docker image.

How to do it...

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

Let's make a small change to the **HttpTrigger** so that you understand that the code is running from Docker, as highlighted in *Figure* 4.36. 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");
}
```

Figure 4.36: Visual Studio—HTTP trigger

Let's now move on to learn how to create a Docker image for the function application.

Creating a Docker image for the function application

In this section, you'll learn how to create a Docker image and run it locally by performing the following steps:

1. The first step in creating a Docker image is to create a **Dockerfile** in your Visual Studio project. Create a **Dockerfile** (a text file with **.dockerfile** as the extension) with the following content:

```
FROM mcr.microsoft.com/azure-functions/dotnet:3.0
COPY ./ bin/Release/netcoreapp3.0 /home/site/wwwroot
```

2. Then, navigate to the command prompt (to the path of the project, as shown in *Figure* 4.37) and run the **docker build -t functionsindocker.** Docker command (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 *Figure* 4.37:

```
C:\Users\[redacted]\FunctionAppinVisualStudio\FunctionAppinVisualStudio>docker build -t functionsindocker .
Sending build context to Docker daemon  33.65MB
Step 1/2 : FROM mcr.microsoft.com/azure-functions/dotnet:3.0
 ---> 827c29622b16
Step 2/2 : COPY ./bin/Release/netcoreapp3.0 /home/site/wwwroot
 ---> 41f4ecd63638
Successfully built 41f4ecd63638
Successfully tagged functionsindocker:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.
```

Figure 4.37: Console—the Docker command

3. Once the image is successfully created, the next step is to run the Docker image on a specific port. Run the **`docker run -p 2305:80 functionsindocker`** command to execute it. You should see something like *Figure* 4.38:

```
C:\Users\vmadmin\source\repos\FunctionAppinVisualStudio\FunctionAppinVisualStudio>docker run -p 2305:80 functionsindocker
Hosting environment: Production
Content root path: /
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

Figure 4.38: Console—execution of the Docker build command

4. Verify that everything is working fine in the local environment by navigating to the localhost with the right port, as shown in *Figure* 4.39:

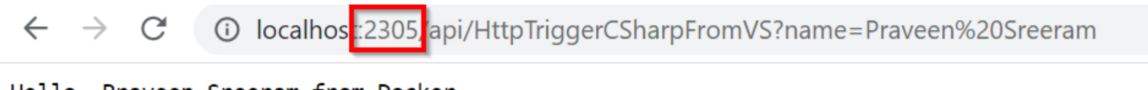


Figure 4.39: Output from the HTTP function hosted as a Docker container in the local environment

Let's move on to the next section to learn how to push the image to the ACR.

Pushing the Docker image to the ACR

In this section, you'll learn how to push the Docker image to the ACR by performing the following steps:

1. The first step is to ensure that you 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 your changes, run the **`docker images`** command, as shown in *Figure* 4.40:

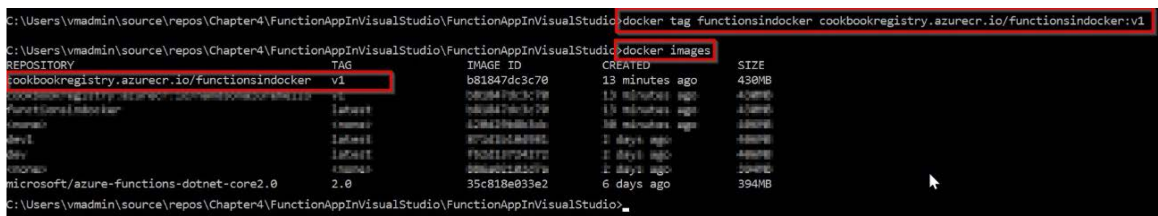


Figure 4.40: Console—execution of the Docker images command

2. In order to push the image to the ACR, you need to authenticate yourself to Azure. For this, you can use the Azure CLI commands. Log in to Azure using the **`az login`** command. Running this command will open a browser and authenticate your credentials, as shown in *Figure 4.41*:

```
C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>az login
Note, we have launched a browser for you to login. For old experience with device code, use "az login --use-device-code"
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "366c4797-e7c7-[illegible]",
    "isDefault": true,
    "name": "Visual Studio Enterprise \u2[illegible]",
    "state": "Enabled",
    "tenantId": "8ef7b61f-88aa-[illegible]",
    "user": {
      "name": "pr[illegible]gmail.com",
      "type": "user"
    }
  }
]

C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>
```

Figure 4.41: Console—logging in to Azure using az commands

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

```
C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>az acr login --name cookbookregistry
Login Succeeded

C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>
```

Figure 4.42: Console—logging in to the ACR using az commands

4. Once you have authenticated yourself, you can push the image to the ACR by running the **`docker push cookbookregistry.azurecr.io/functionsindocker:v1`** command, as shown in *Figure 4.43*:

```
C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>docker push cookbookregistry.azurecr.io/functionsindocker:v1
The push refers to repository [cookbookregistry.azurecr.io/functionsindocker]
3f58e334a394: Pushed
6f9d355b1699: Pushed
e954f34d5c20: Pushed
c30f8864b2d9: Pushed
60add06a0c0d: Pushed
8b15606a9e3e: Pushed
v1: digest: sha256:2beca04c3ffaf2ded6df71eff718f0841840101ea5f5deac9f4dcec1c6ab0d9c size: 1588

C:\Users\vmadmin\source\repos\Chapter4\FunctionAppInVisualStudio\FunctionAppInVisualStudio>
```

Figure 4.43: Console—execution of the Docker push command

5. Navigate to the ACR in the Azure Management portal and review whether your image was pushed to it properly in the **Repositories** blade, as shown in *Figure 4.44*:

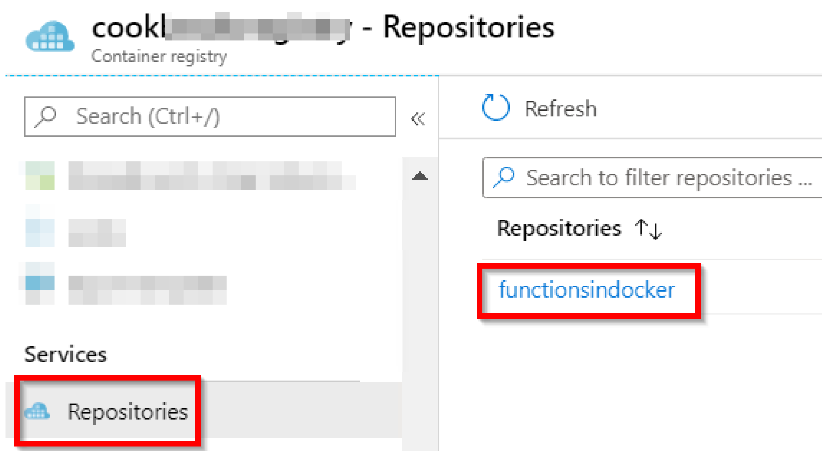


Figure 4.44: Azure ACR—Repositories view

You have successfully created an image and pushed it to the 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 application with Docker

In order to deploy the function application code as a Docker image, you need to set the **Publish Type** as **Docker Container** while creating the function application itself. Perform the following steps to create a new function application:

1. Navigate to the **New | Function App** blade and choose **Docker Container** as the option in the **Publish** field, and then provide the following information under the **Basics** tab:

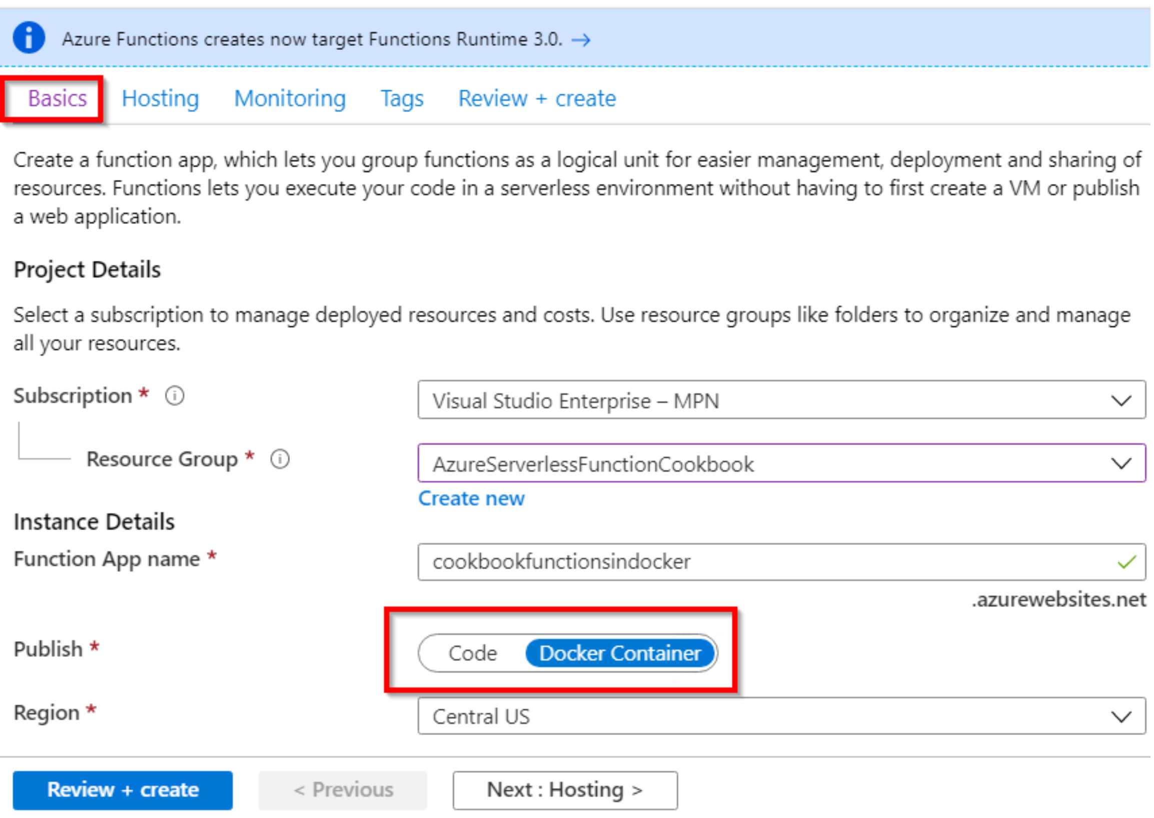


Figure 4.45: Function application creation—Basics

2. Now, in the **Hosting** tab, the **Linux** option is selected in the OS field. Choose **App service plan** in the **Plan type** field and then choose other fields as shown in *Figure 4.46*. Here, choose to create a new **App service plan** based on your requirements:

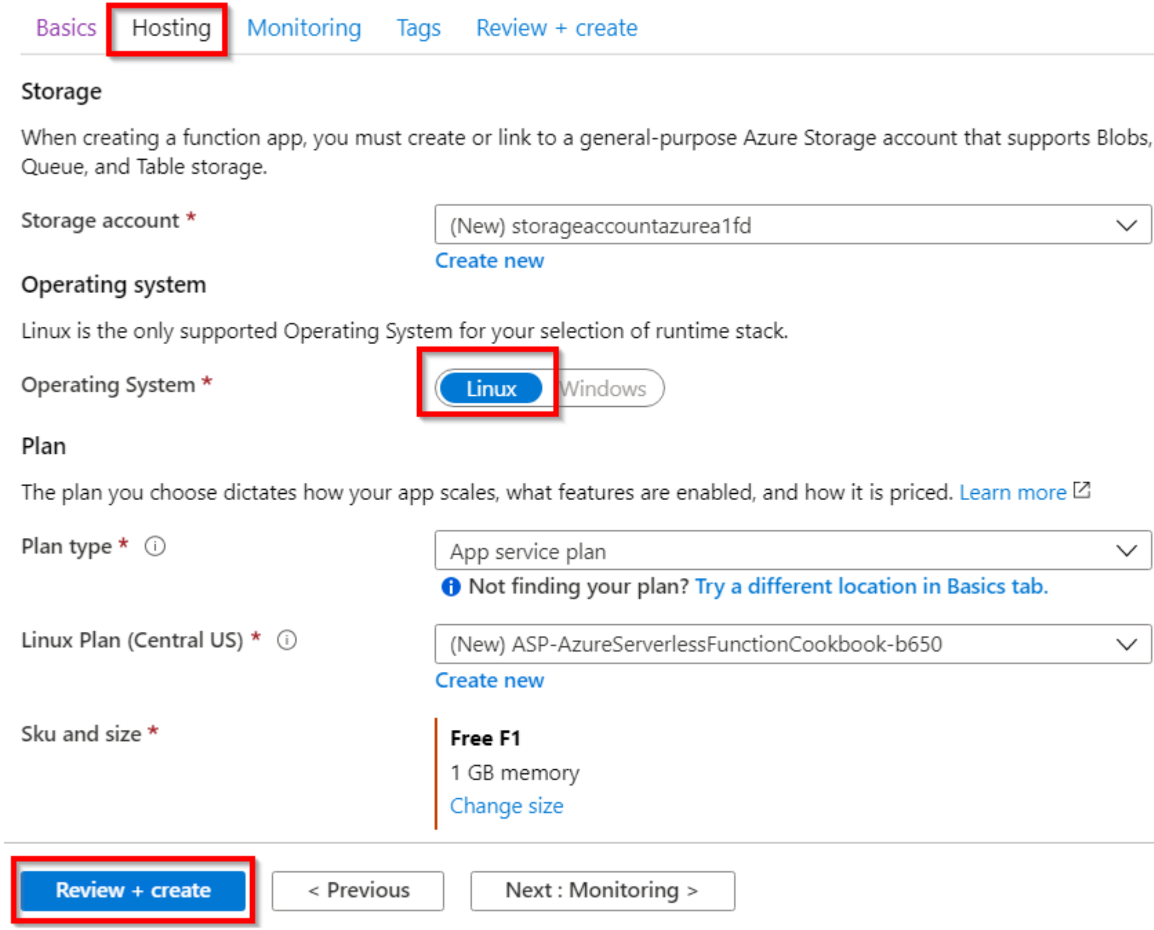


Figure 4.46: Function application creation—Hosting

3. Once you've reviewed all the details, click on the **Review + create** button to create the function application.

4. The next and most important step is to refer the Docker image that you have pushed to the ACR. This can be done by clicking on the **Configure container** button available in the **Platform** features tab and choosing **Azure Container Registry**, and then choosing the correct image, as shown in *Figure 4.47*:

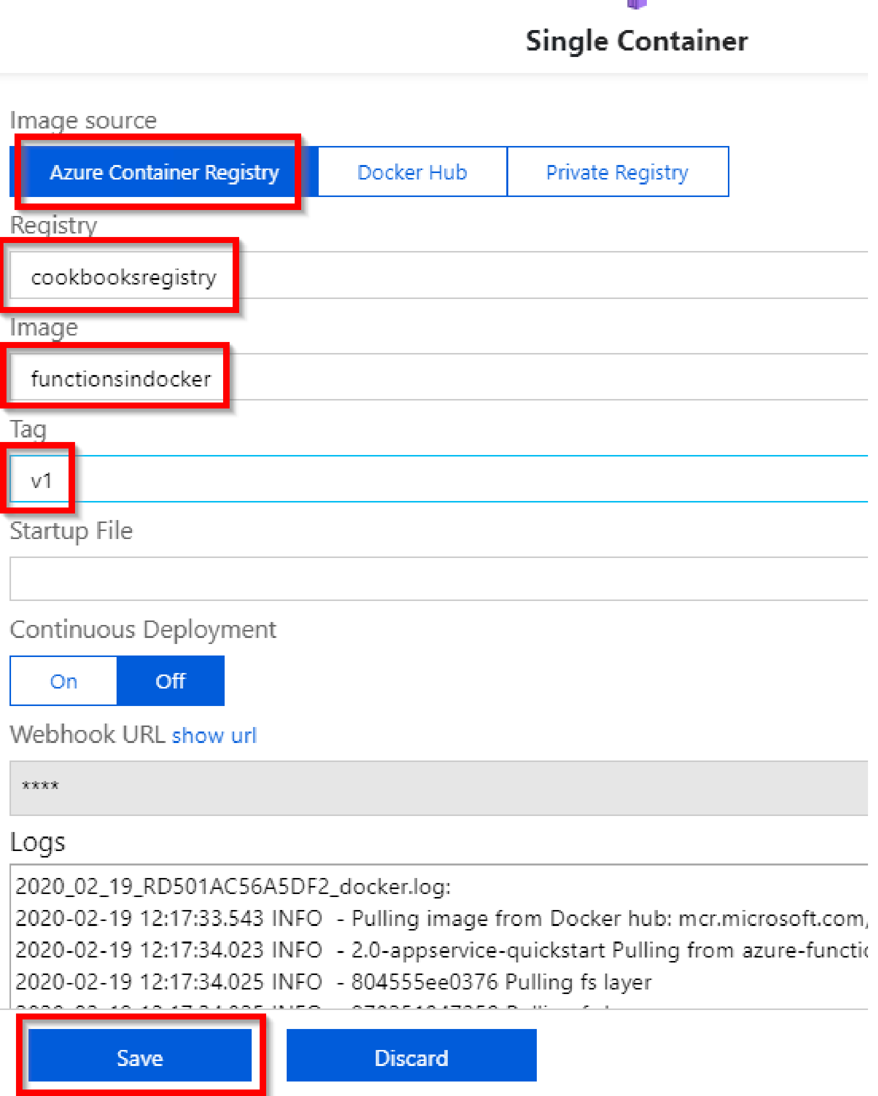


Figure 4.47: Function application creation—Docker image source

5. That's it. You have created a function application that helped you to deploy the Docker image by linking it to the image hosted in the Azure Container Registry. Let's quickly test `HttpTrigger` by navigating to the HTTP endpoint in the browser. The following is the output of the Azure function:

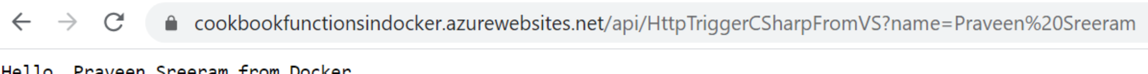


Figure 4.48: Output from the HTTP function hosted as a Docker container in the Azure environment

How it works...

In this recipe, you have done the following:

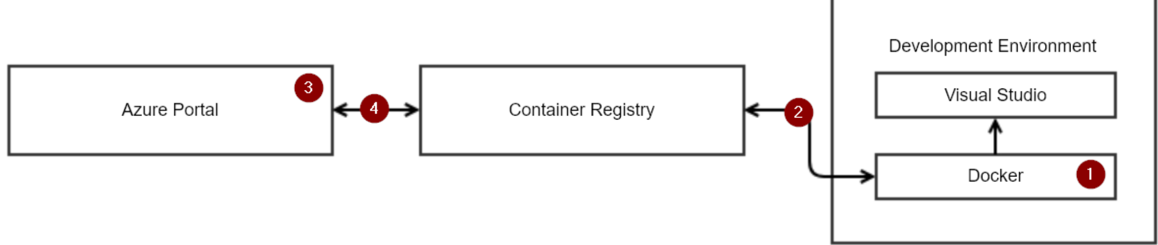


Figure 4.49: The Azure Function application as a Docker container—process diagram

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

1. Create a Docker image of the function application that you created in this chapter using Visual Studio.
2. Push the Docker image to the ACR.
3. From the Azure Management portal, while creating a new function application, choose the option to publish the executable package as a Docker image.
4. Attach the Docker image from the ACR (from step 2) to the Azure function (from step 3).

In this recipe, you have learned how to work with the Visual Studio IDE in developing Azure Functions and have also seen how to debug a local and remote version of the code from Visual Studio.

5

Exploring testing tools for Azure functions

In this chapter, we'll explore different ways of testing Azure functions in detail with the following recipes:

- Testing Azure functions
- Testing an Azure function in a staging environment using deployment slots
- Creating and testing Azure functions locally using Azure CLI tools
- Validating Azure function responsiveness using Application Insights
- Developing unit tests for Azure functions with HTTP triggers

Introduction

Up to this point, you have learned how to develop and apply Azure functions, in addition to validating the functionality of these functions. This chapter will explore some of the popular ways of testing different Azure functions. This includes running tests of HTTP trigger functions using Postman, as well as using Azure Storage Explorer to test Azure blob triggers, queue triggers, and other storage service–related triggers.

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

Testing Azure functions

The Azure Functions runtime allows us to create and integrate many Azure services. At the time of writing, there are more than 20 types of Azure function that you can create. This recipe will explain how to test the most common Azure functions; we'll look at the following:

- Testing HTTP triggers using Postman
- Testing a blob trigger using Azure Storage Explorer
- Testing a queue trigger using the Azure portal

Getting ready

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

- **Postman**: This is a tool that will allow you to make calls to APIs. You can download this from https://www.getpostman.com/.
- **Azure Storage Explorer**: 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 directly from Storage Explorer. You can download this from http://storageexplorer.com/.

How to do it...

In this section, we'll create three Azure functions using the default templates available in the Azure portal, and then test them with a variety of tools.

Testing HTTP triggers using Postman

When working with applications in a production environment, usually, developers would not have access to the Azure portal. Therefore, we need to rely on tools that will assist in testing the HTTP triggers. In this section, you'll learn how to test HTTP triggers using Postman.

Perform the following steps:

1. Create an HTTP trigger function that accepts the **`Firstname`** and **`Lastname`** parameters and sends a response back to the client. Once created, make sure that you set **Authorization Level** to **Anonymous**.
2. Replace the default code with the following. Note that, for the sake of simplicity, we have removed the validations. Real-time applications will require the validation of each 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 do the following:

   The first step is to choose the HTTP request method. Since the HTTP trigger function accepts most methods by default, choose the **GET** method, as shown in *Figure 5.1*:

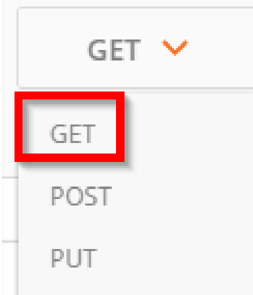


Figure 5.1: The Postman tool

   The next step is to provide the URL of the HTTP trigger. Remember to replace **`<HttpTriggerTestUsingPostman>`** with the actual HTTP trigger function name, as shown in *Figure 5.2*:

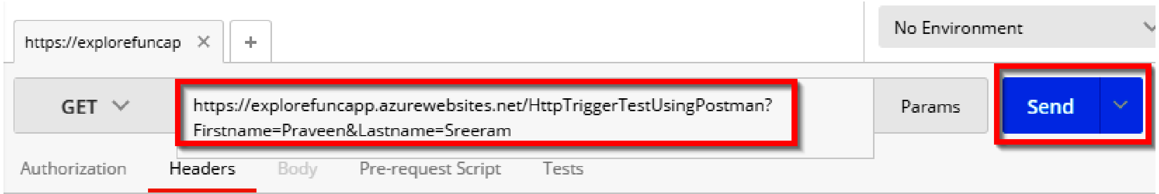


Figure 5.2: Providing the URL of the HTTP trigger

   Click on the **Send** button to make the request. If all the details expected by the API are provided, **Status: 200 OK** should be visible along with the response, as shown in *Figure 5.3*:

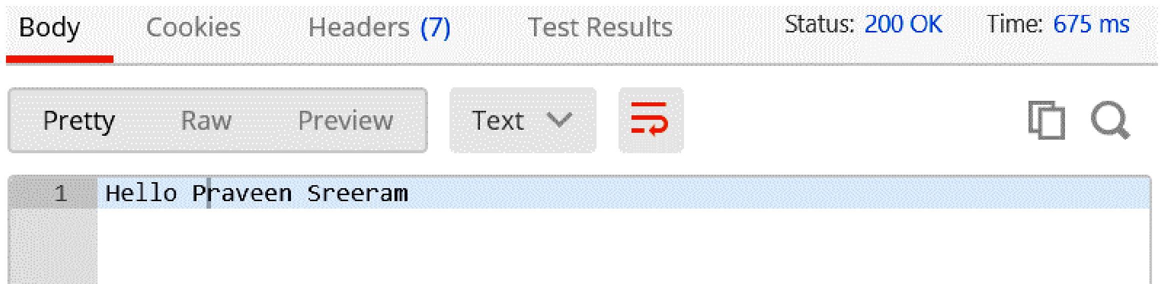


Figure 5.3: Output in the Postman tool

You have learned how to test an HTTP trigger. Let's now move on to the next section.

Testing a blob trigger using Storage Explorer

In this section, we'll test a blob trigger by performing the following steps:

1. Create a new blob trigger by choosing the **Azure Blob Storage trigger** template, as shown in *Figure* 5.4:

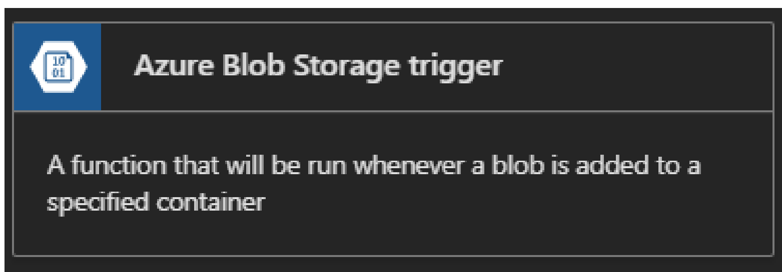


Figure 5.4: The Azure Blob Storage trigger template

2. Clicking on the template will prompt you to provide a storage account and a container for storing the blob. Enter the storage account name as **`BlobTriggerCSharpTestUsingStorageExplorer`**. In the **Azure Blob Storage trigger** template, set the **Path** value to **`samples-workitems/{name}`**, and select **`azurefunctionscookbooks_Storage`** from the **Storage account connection** drop-down list, as shown in *Figure* 5.5:

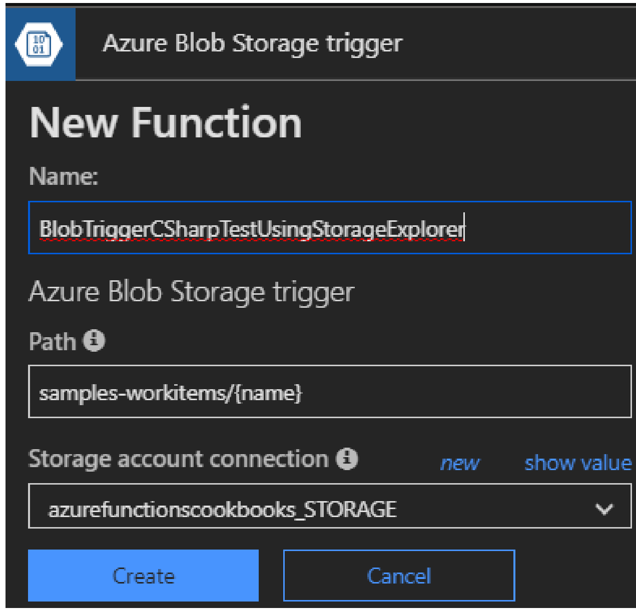


Figure 5.5: Azure Blob storage trigger creation

3. Let's now connect to the storage account that we'll be using in this recipe. Open **Microsoft Azure Storage Explorer** and click on the **Connect** symbol, as highlighted in *Figure* 5.6, to connect to **Azure Storage**:

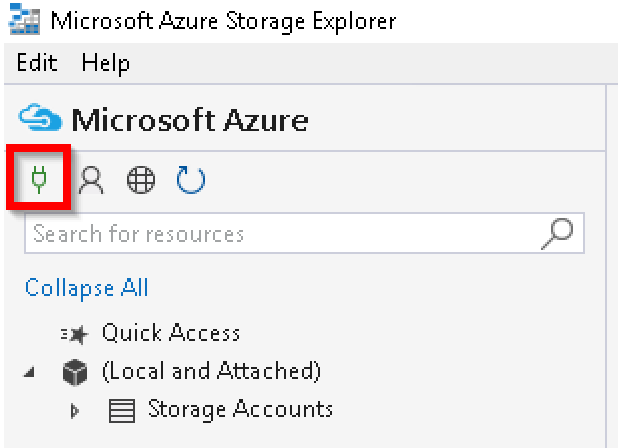


Figure 5.6: Azure Storage Explorer—connecting

4. You will be prompted to enter various details, including the storage connection string, **shared access signature** (**SAS**), and the 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** window, as shown in *Figure* 5.7:

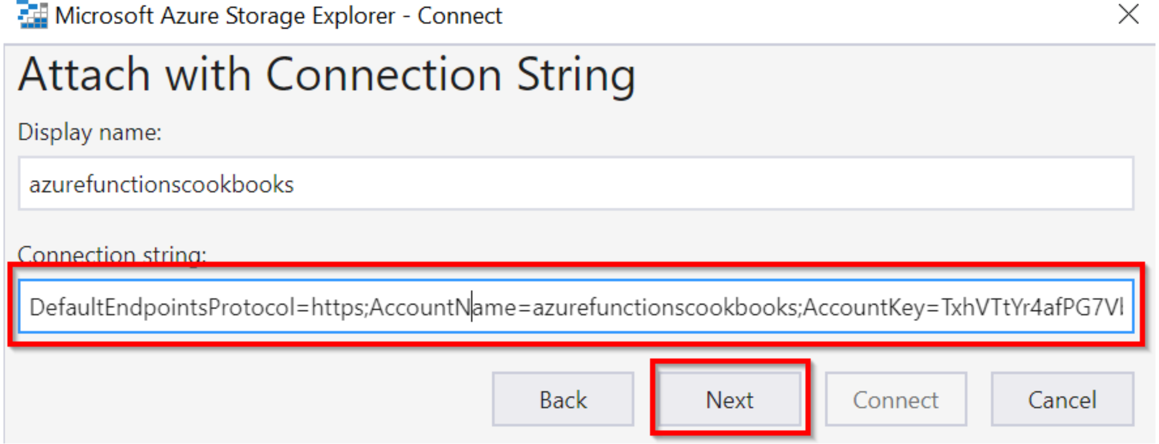


Figure 5.7: Azure Storage Explorer—Attach with Connection String

5. Clicking on the **Next** button, as shown in *Figure* 5.7, will redirect 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 *Figure* 5.8, you should now be connected to the Azure storage account, from which all Azure Storage services can be managed:

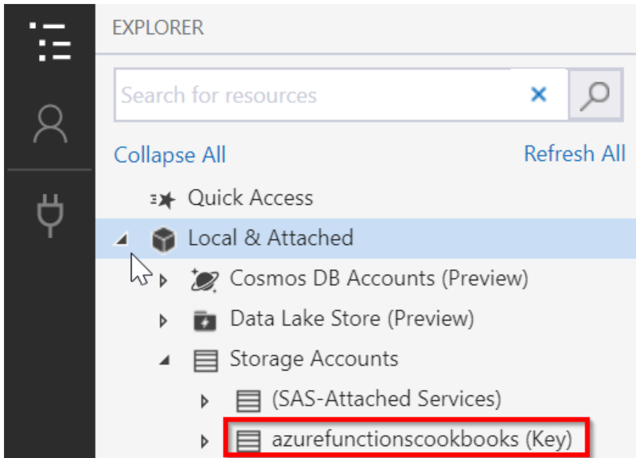


Figure 5.8: Azure Storage Explorer—connected storage account

7. Now, let's create a storage blob 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 *Figure* 5.9:

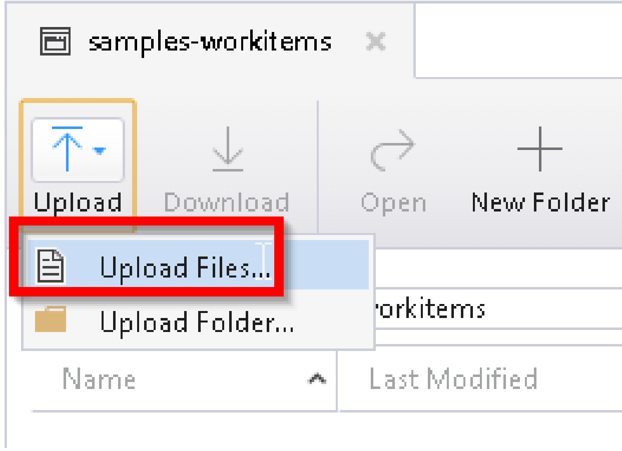


Figure 5.9: Azure Storage Explorer—Upload Files

8. In the **Upload Files...** window, choose a file 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 *Figure 5.10*. The log will show the Azure function being triggered successfully:

Logs Console Reconnect Copy logs Pause Clear Expand

```
2020-02-25T12:02:59  Welcome, you are now connected to log-streaming service. The default timeout is 2 hours. Change the timeout with the
App Setting SCM_LOGSTREAM_TIMEOUT (in seconds).
2020-02-25T12:03:00.073 [Information] Executing 'Functions.BlobTriggerCSharpusingStorageExplorer' (Reason='New blob detected: samples-
workitems/convertcsv (2).xlsx', Id=34c9aefb-1b54-4384-ba40-0891769c940f)
2020-02-25T12:03:00.083 [Information] C# Blob trigger function Processed blob
 Name:convertcsv (2).xlsx
 Size: 90815 Bytes
2020-02-25T12:03:00.083 [Information] Executed 'Functions.BlobTriggerCSharpusingStorageExplorer' (Succeeded, Id=34c9aefb-1b54-4384-ba40-
0891769c940f)
```

Figure 5.10: Azure Functions—blob trigger logs

Testing a queue trigger using the Azure portal

In this section, you'll learn how to test a queue trigger by performing the following steps:

1. Create a new **Azure Queue Storage trigger** template named `QueueTriggerTestusingPortal`, as shown in *Figure 5.11*. Make a note of the **Queue name**, `myqueue-items`, as you will need to create a queue service with the same name later using the Azure portal:

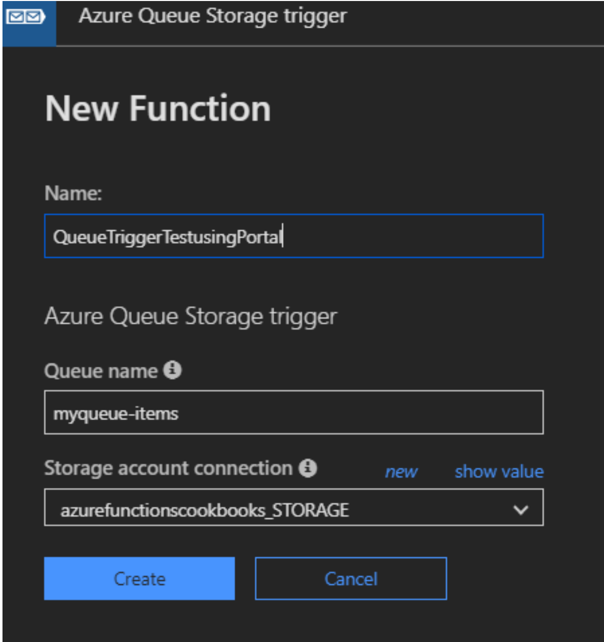


Figure 5.11: Azure Queue storage trigger creation

2. Navigate to the **Storage account** | **Overview** blade and click on **Queues**, as shown in *Figure* 5.12:

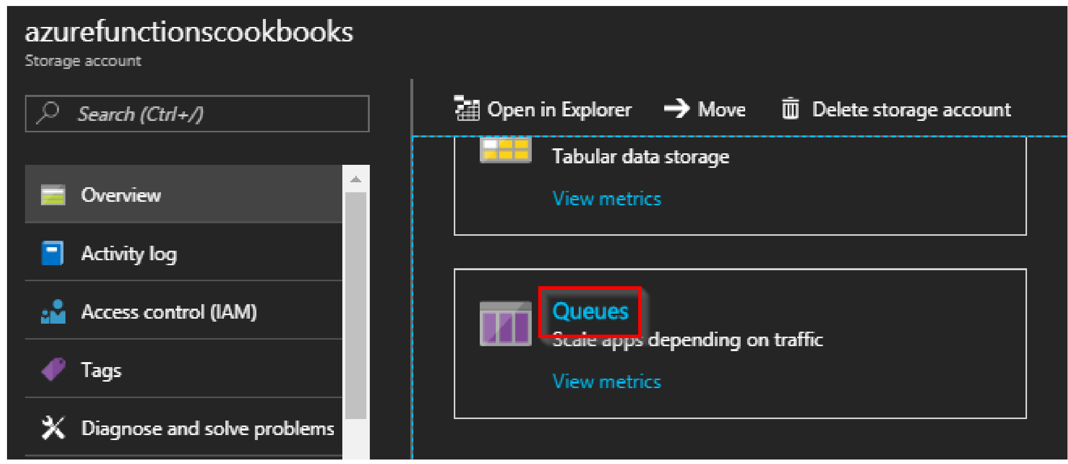


Figure 5.12: The Azure Storage Overview blade

3. In the **Queues** blade, click on **+Queue** to add a new queue:

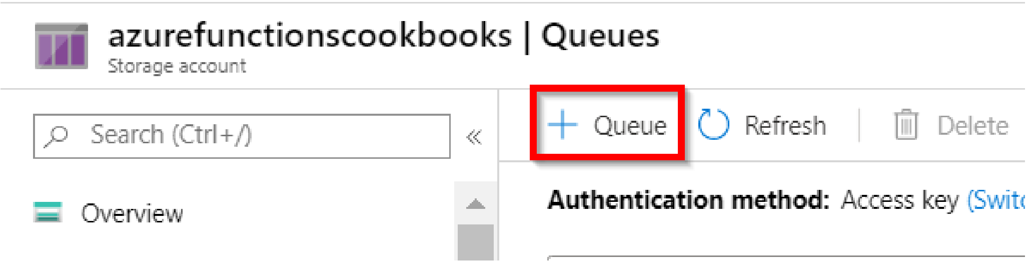


Figure 5.13: Azure Queue storage

4. Provide **`myqueue-items`** as the **Queue name** in the **Add queue** popup, as shown in *Figure* 5.14. This was the same name you used while creating the queue trigger. Click on **OK** to create the queue service:

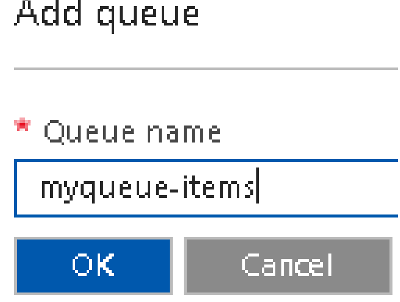


Figure 5.14: Azure Storage—adding a queue

5. Now, let's create a queue message. In the Azure portal, click on the `myqueue-items` queue service to navigate to the **Messages** blade. Click on the **Add message** button, as shown in *Figure* 5.15, and then provide some message text. Lastly, click on **OK** to create the queue message:

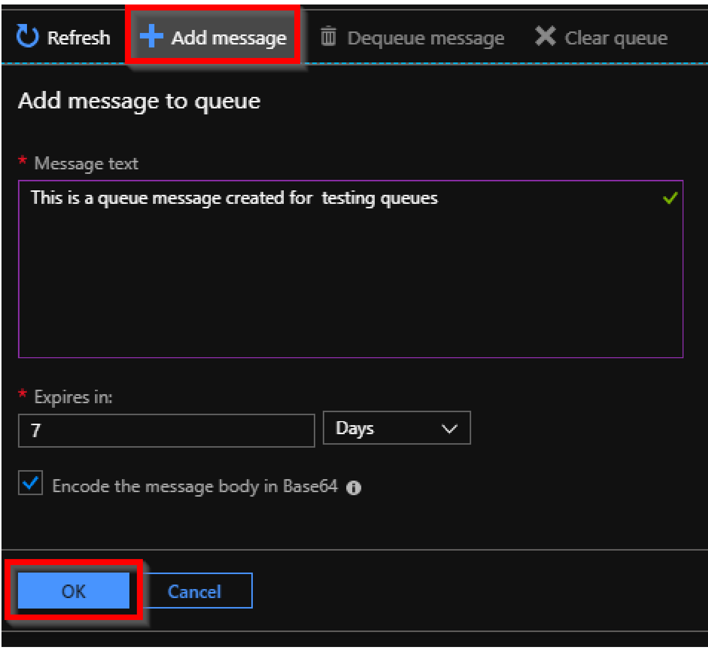


Figure 5.15: Azure Storage—adding a message to the queue

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 *Figure* 5.16:

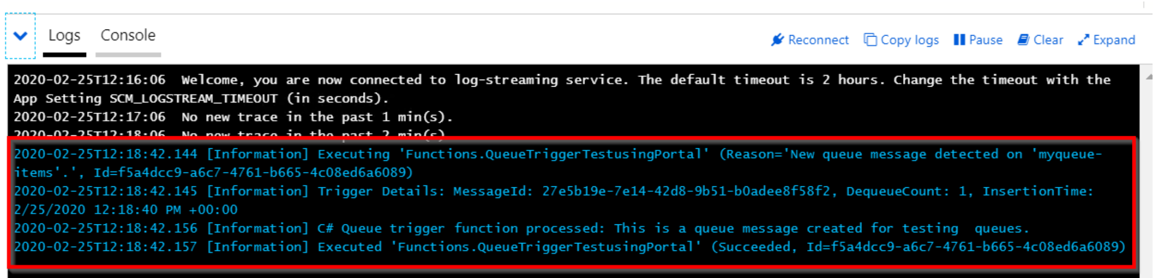


Figure 5.16: Azure queue trigger logs

There's more...

To allow API consumers to only use the `POST` method for your HTTP trigger, restrict it by choosing only **POST** in **Selected HTTP methods**, as shown in *Figure* 5.17. Navigate to this by clicking on the **Integrate** tab of the HTTP trigger:

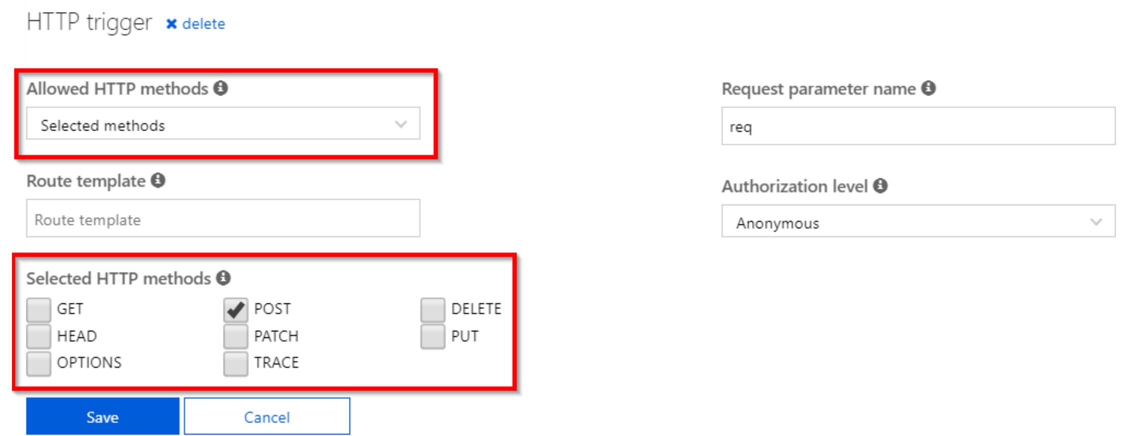


Figure 5.17: Azure HTTP trigger integration settings

This recipe explained how to test the most common Azure functions. In the next recipe, you'll learn how to test an Azure function in a staging environment.

Testing an Azure function in a staging environment using deployment slots

In general, every application requires pre-production environments, such as staging and beta, in order to review functionalities prior to publishing them for end users. Although pre-production environments are great and help multiple stakeholders validate the application's functionality against the business requirements, there are a number of pain points associated with managing and maintaining them. These include the following:

- We need to create and use a separate environment for our pre-production environments.
- Once the application's functionality is reviewed in pre-production and the IT Ops team gets the go-ahead, there will be some downtime in the production environment while deploying the code based on the new functionalities.

All the preceding limitations can be covered by Azure Functions using a feature called slots (known as deployment slots in the **App Service** service). A pre-production environment can be set up using slots. Here, developers can review all of the new functionalities and promote them (by swapping) to the production environment seamlessly based on requirements.

How to do it...

In this section, you'll create a slot and also learn how the swap works by performing 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 was 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 functions of the function app are live on your production environment at **`https://<<functionappname.azurewebsites.net>>`**.
5. Now, the customer has asked you to make some changes to both functions. Instead of making the changes directly to the functions of your production function app, you may need to create a slot.
6. Let's create a new slot with all the functions in your function app, named **`MyProductionApp`**.

7. Click on the **+** icon, available near the **Slots** section, as shown in *Figure 5.18*:

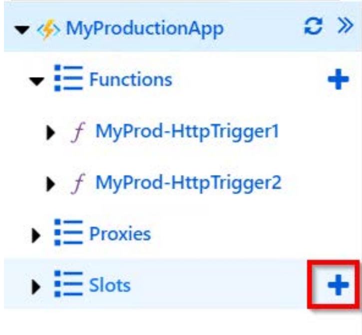


Figure 5.18: List of all functions in a function app

8. Enter a name for the new slot. Provide a meaningful name, something such as `staging`, as shown in *Figure 5.19*:

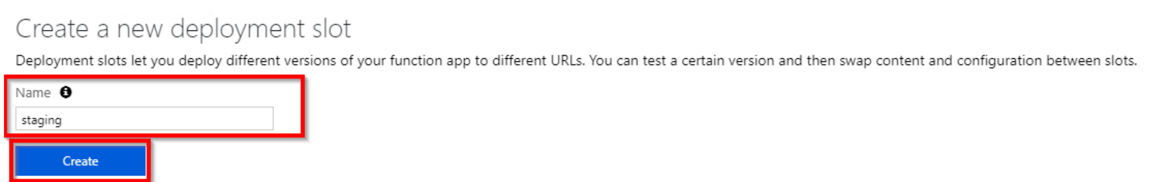


Figure 5.19: Creating a new deployment slot

9. Once you click on **Create**, a new slot will be created, as shown in *Figure 5.20*. If the functions are read-only, you can make them read-write in the function app settings of the **staging** slot:

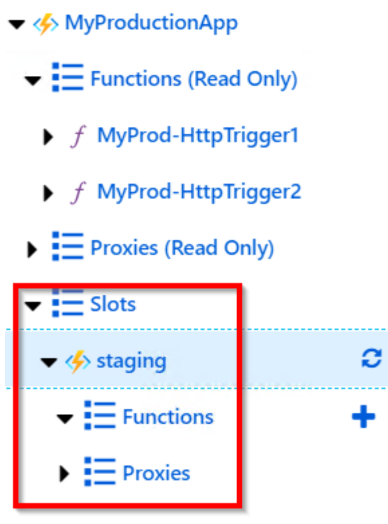


Figure 5.20: Slots view

10. To make the staging environment complete, copy all the Azure functions from the production environment (in this case, the **`MyProductionApp`** application) to the new staged slot named **`staging`**. Create two HTTP triggers and copy the code of both functions (**`MyProd-HttpTrigger1`** and **`MyProd-HttpTrigger2`**) from **`MyProductionApp`** to the new **`staging`** slot. Basically, copy all the functions to the new slot manually.
11. Change the word **`Production`** to **`Staging`** in the last line of both the functions in the **`staging`** slot, as shown in *Figure 5.21*. This is useful for testing the output of the swap operation:

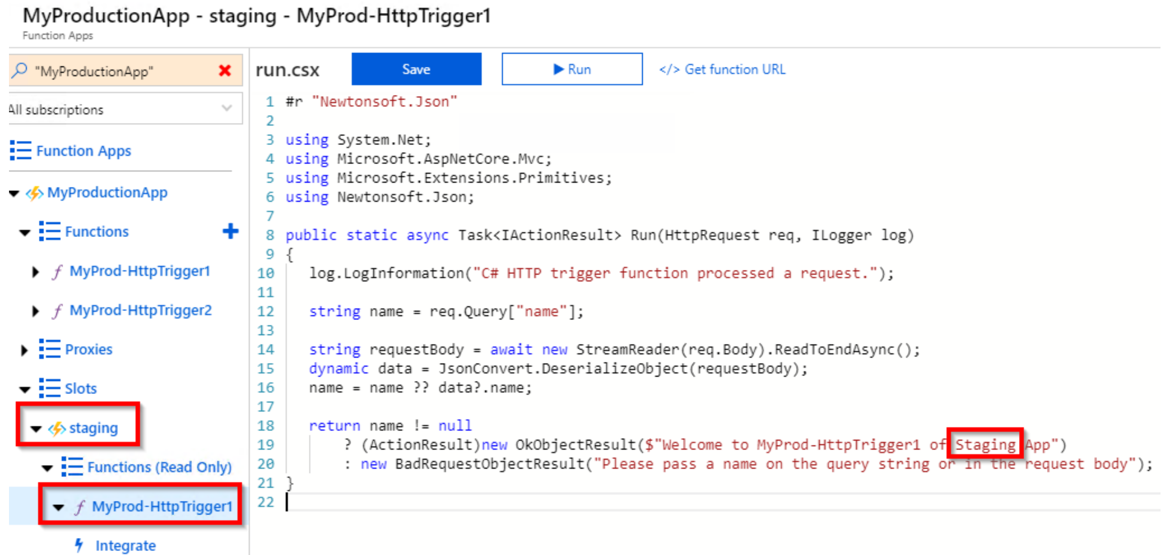


Figure 5.21: Staging slot—replacing the word "Production" with "Staging" for testing

> **Note**
>
> In all the slots that were created as a pre-production application, make sure that you use the same function names as those in your production environment.

12. Click on the **Swap** button, available in the **Overview** blade, as shown in *Figure 5.22*:

Figure 5.22: Swap operation

13. In the **Swap** blade, choose the following:

- **Source**: Choose the slot that you would like to move to production. In this case, we're swapping `staging` in general, but you can also swap across non-production slots.
- **Target**: Choose the production option, as shown in *Figure* 5.23:

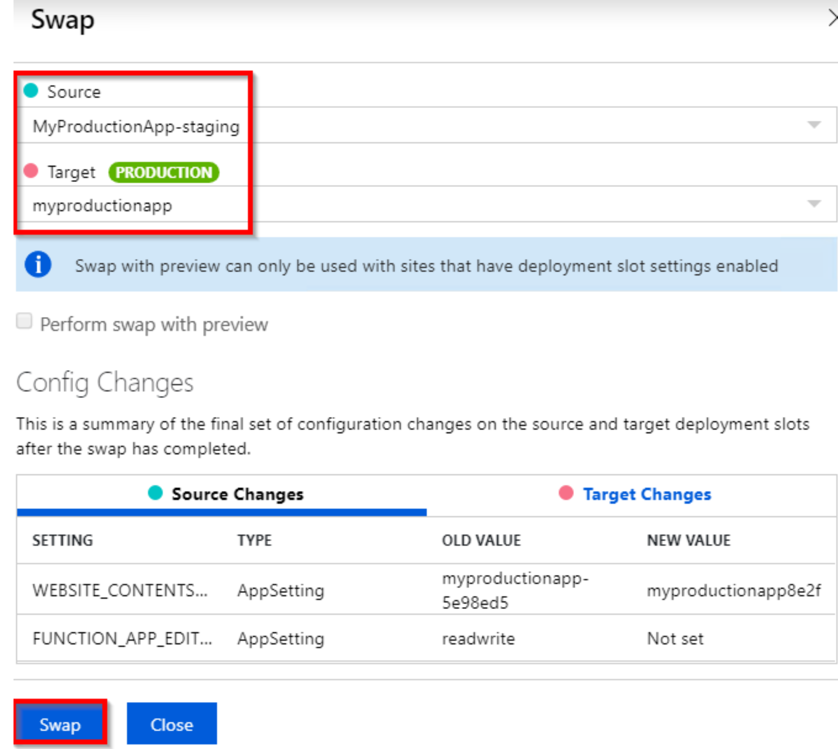


Figure 5.23: Swap operation overview with source and target slots

14. After reviewing the settings, click on the **Swap** button. It will take a few moments to swap the functions. A progress bar will appear, as shown in *Figure* 5.24:

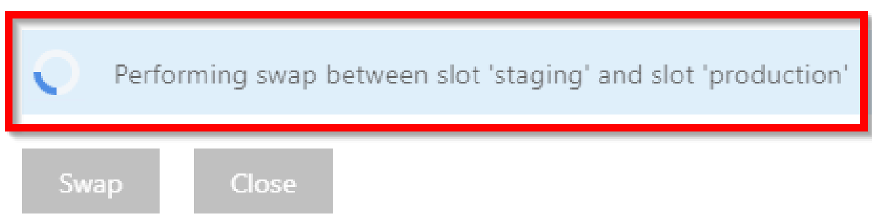


Figure 5.24: Performing the swap operation

15. After a minute or two, the **staging** and **production** slots have been swapped. Let's now review the **run.csx** script files of the production slot:

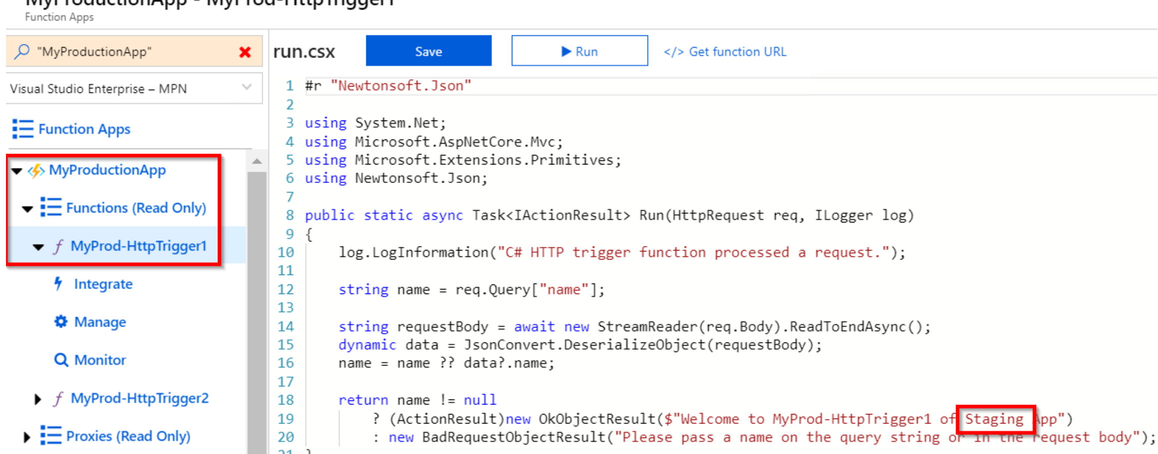


Figure 5.25: Changing the production slot's content

16. If there are no changes, click on the refresh button of the function app, as shown in *Figure 5.26*:

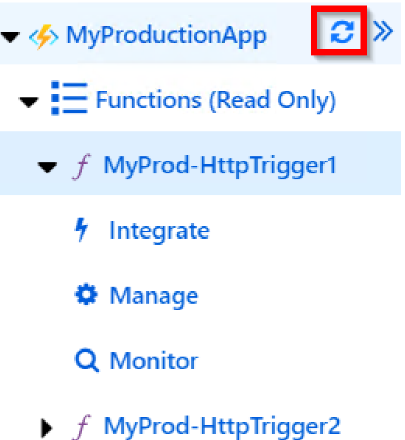


Figure 5.26: Azure function app—refresh

There's more...

Make sure that the application settings and database connection strings are marked as **Slot Setting** (slot-specific). Otherwise, the application settings and database connection strings will also be swapped, which could result in unexpected behavior. Mark any of these settings as such from the **Configuration** blade available in **Platform features**, as shown in *Figure 5.27*:

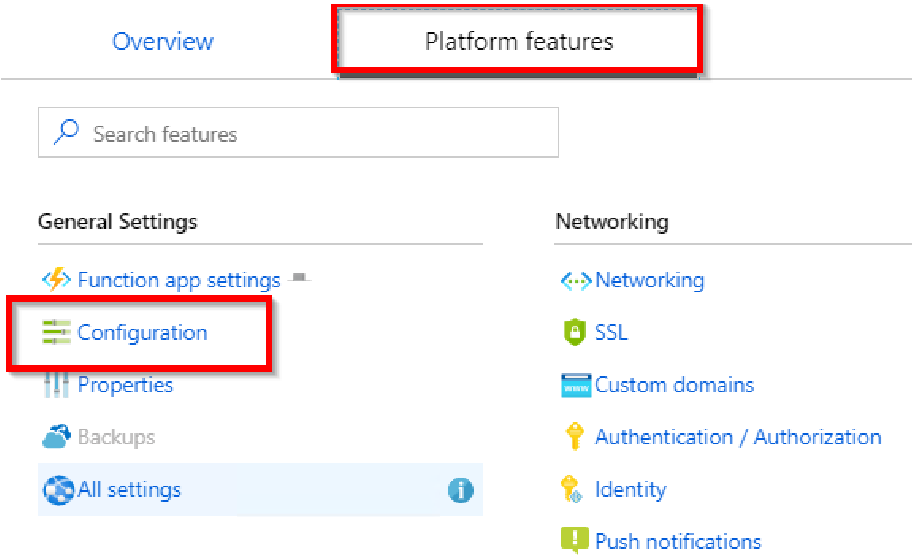


Figure 5.27: Platform features

Clicking on the **Configuration** blade will take you to a list of all settings. Click on the **Edit** button, which will open up the blade beneath, where you can mark any setting as a **Deployment slot setting**:

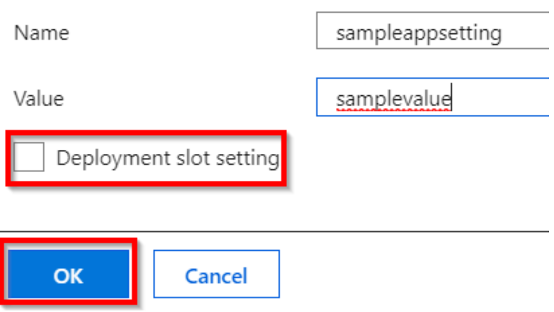


Figure 5.28: Add/Edit application setting

> **Note**
>
> All the functions in the recipe are HTTP triggers; note that we can have any kinds of triggers in a function app. The deployment slots are not limited to HTTP triggers. We can have multiple slots for each function app. The following are a few examples:
>
> Alpha
>
> Beta
>
> Staging

While creating a slot without enabling the feature of deployment slots, you'll see something similar to what is shown in *Figure* 5.29:

Create a new deployment slot

Deployment slots let you deploy different versions of your function app to different URLs. between slots.

Azure functions slots (preview) is currently disabled. To enable, visit function app settings.

Figure 5.29: Create a new deployment slot

We need to have all the Azure functions in each of the slots that should be swapped with the production function app:

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

In general, while using App Services, in order to create deployment slots, have the App Services plan set to either the Standard or Premium tier.

However, we can create a slot (only one) for the function app even if it is under the Consumption (or dynamic) plan.

In this recipe, we have learned how to create a pre-production environment using slots, which help developers to test new releases before taking them to a production environment.

Creating and testing Azure functions locally using Azure CLI tools

Most of the recipes so far have been created using either the browser or the Visual Studio **integrated development environment** (**IDE**).

Azure also provides tools for developers who prefer to work with the command line. These tools allow us to create Azure resources with simple commands right from the command line. In this recipe, we'll learn how to create a new function app, and we'll also understand how to create a function and deploy it to Azure directly from the command line.

Getting ready

Before proceeding further with the recipe, install Node.js and the Azure CLI. The download links for these tools are as follows:

- Download and install Node.js from https://nodejs.org/en/download/.
- Download and install Azure Functions Core Tools (also known as the Azure CLI) from https://docs.microsoft.com/azure/azure-functions/functions-run-local?tabs=windows%2Ccsharp%2Cbash.

How to do it...

Once the installation of the Azure CLI is complete, perform the following steps:

1. Create a new function app in the Azure CLI by running the following command:

   ```
   func init
   ```

   The following output should be displayed after executing the preceding command:

```
C:\Users\vmadmin\myazureclifunc>func init
Select a worker runtime:
dotnet
node
```

Figure 5.30: Command Prompt

In *Figure* 5.30, **`dotnet`** is selected by default. Pressing *Enter* will create the required files, as shown in *Figure* 5.31:

Figure 5.31: The Azure function app—project files in Windows Explorer

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

```
func new
```

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

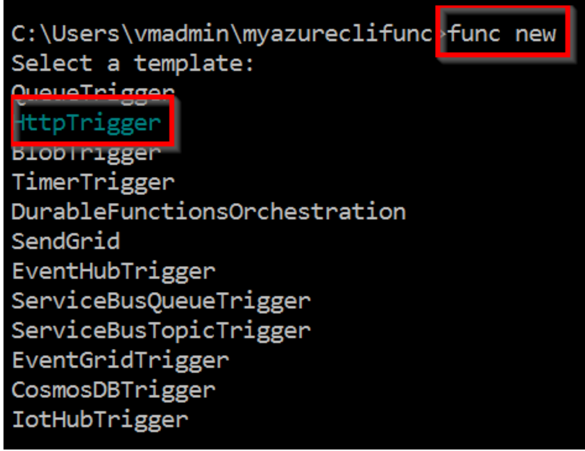


Figure 5.32: Creating a new function

3. As shown in *Figure* 5.32, we'll be prompted to choose the function template. For this recipe, we have chosen **`HttpTrigger`**. Choose **`HttpTrigger`** by using the down arrow key and then hit *Enter*. Choose the Azure function type based on your requirements. We can navigate between the options using the up/down arrow keys on our keyboard.

4. The next step is to provide a name for the Azure function that we are creating. Provide a meaningful name—here we're using **`HttpTrigger-CoreTools`**—and then press *Enter*, as shown in *Figure* 5.33:

```
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>_
```

Figure 5.33: Creating a new function

5. Use your preferred IDE to edit the Azure function code. In this recipe, we'll use Visual Studio Code to open the **`HttpTrigger`** function, as shown in *Figure* 5.34:

```
5   using Microsoft.AspNetCore.Mvc;
6   using Microsoft.Azure.WebJobs;
7   using Microsoft.Azure.WebJobs.Extensions.Http;
8   using Microsoft.AspNetCore.Http;
9   using Microsoft.Azure.WebJobs.Host;
10  using Microsoft.Extensions.Logging;
11  using Newtonsoft.Json;
12
13  namespace myazureclifunc
14  {
15      public static class HttpTrigger_CoreTools
16      {
17          [FunctionName("HttpTrigger_CoreTools")]
18          public static async Task<IActionResult> Run([HttpTrigger(AuthorizationLevel.Function, "get", '
19          {
20              log.LogInformation("C# HTTP trigger function processed a request.");
21
22              string name = req.Query["name"];
23
24              string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
25              dynamic data = JsonConvert.DeserializeObject(requestBody);
26              name = name ?? data?.name;
27
28              return name != null
29                  ? (ActionResult)new OkObjectResult($"Hello, {name}")
30                  : new BadRequestObjectResult("Please pass a name on the query string or in the request
31          }
32      }
33  }
```

Figure 5.34: Creating a new function

6. Let's test the Azure function right from our 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 *Figure* 5.35:

Figure 5.35: HTTP trigger output

In this recipe, we have learned how to add a new function app and a function using the Azure CLI.

Validating Azure function responsiveness using Application Insights

An application is only useful for a business if it is up and running. Applications might go down for multiple reasons. These reasons include the following:

- Any hardware failures, such as a server crash, hard disk errors, or any other hardware issue—even an entire datacenter might go down, although this would be very rare.
- 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 is vital to be notified when your site is not available or not responding to user requests. Azure provides a few tools to help by alerting you if your website is not responding or is down. One of these is **Application Insights**. Let's learn how to configure Application Insights to ping our Azure function app every 5 minutes and set it to send an alert if the function fails to respond.

> **Note**
>
> Application Insights is an **application lifecycle management** (**ALM**) tool that allows performance tracking, exception monitoring, and also the collection of application telemetry data.

Getting ready

In this section, we'll create an Application Insights instance and also learn how to create an availability test by performing the following steps:

1. Navigate to the Azure portal, search for **Application Insights**, click on the **Create** button, and then provide all the required details, as shown in *Figure* 5.36:

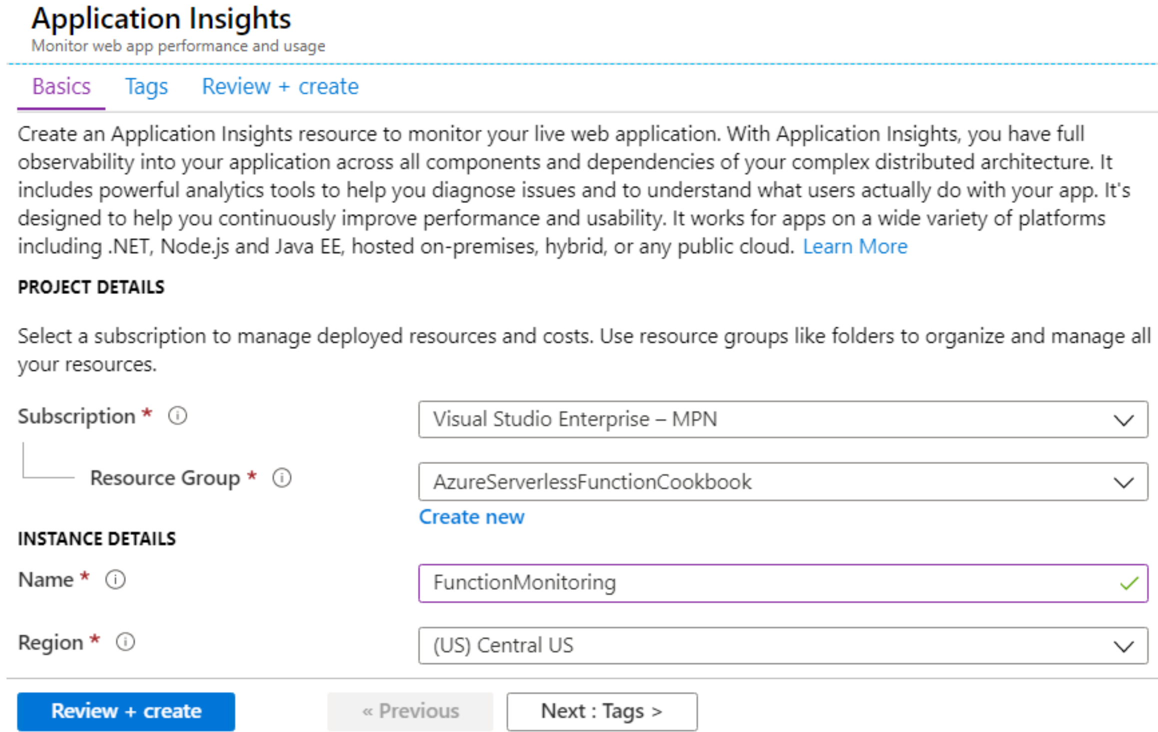


Figure 5.36: Creating a new Application Insights instance

2. Navigate to the function app's **Overview** blade and grab the function app URL, as shown in *Figure* 5.37:

Figure 5.37: Copying the function app URL

How to do it...

In this section, we'll learn how to do an automated ping test to the HTTP trigger using an availability test by performing the following steps:

1. Navigate to the **Availability** blade in Application Insights, as shown in *Figure* 5.38, and click on the **Add test** button:

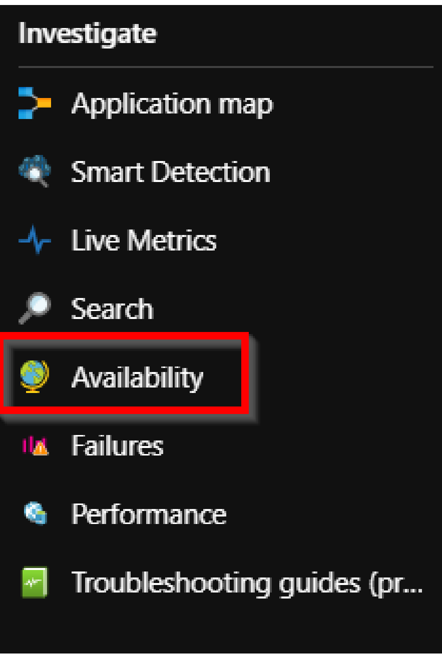


Figure 5.38: The Application Insights menu

2. In the **Create test** blade, we'll see the following four sections:

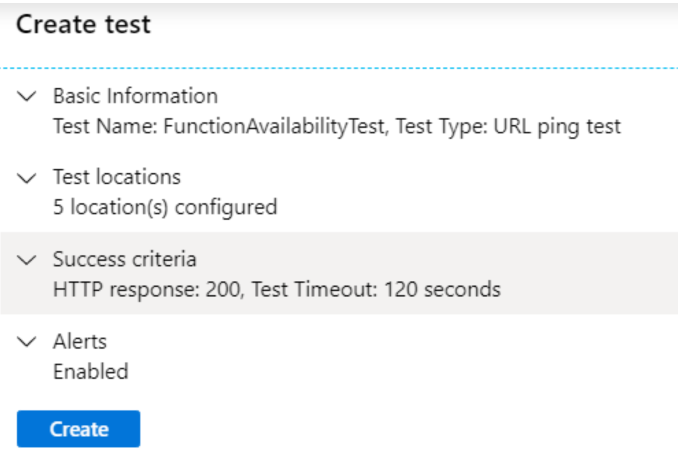


Figure 5.39: Creating an availability test

3. In the **Basic Information** section, please enter a meaningful name (in this case, we have used `FunctionAvailabilityTest`) based on the requirements and paste the function app URL, which was noted in *step* 2 of the *Getting ready* section, in the URL field of the **Basic Information** section of the **Create test** blade.

4. In the **Test Locations** section, choose the locations that we want Azure to perform ping tests for.

> **Note**
>
> A minimum of five locations and a maximum of 16 locations can be chosen.

5. After reviewing the details, click on the **Create** button to create the ping test, as shown in *Figure* 5.39.

6. The next step is to configure alerts. To do so, click on **...**, as shown in *Figure* 5.40:

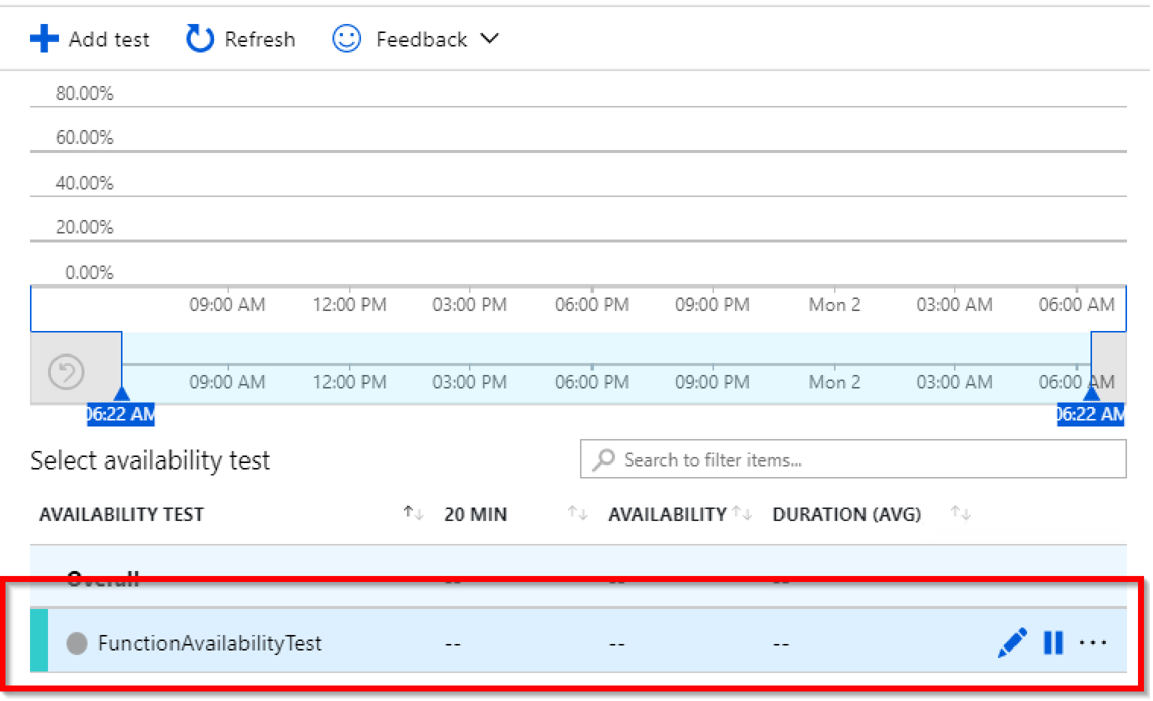


Figure 5.40: Editing the availability test

7. This opens up a context menu. In the context menu, select **Edit alert**, as shown in *Figure* 5.41:

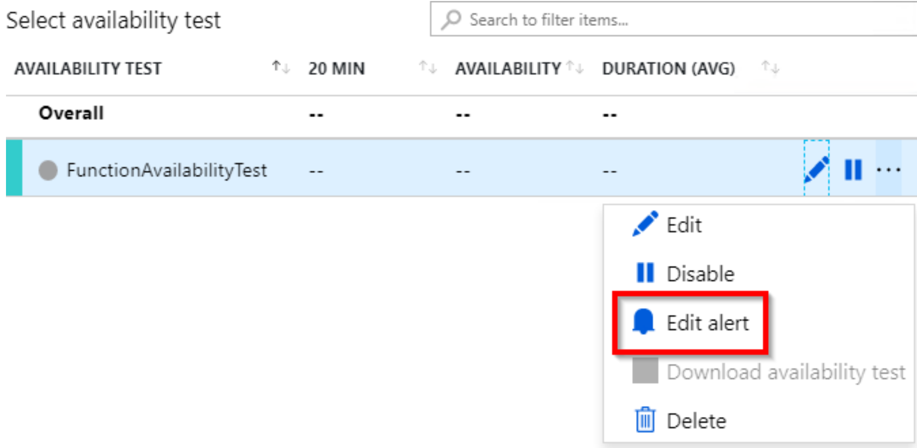


Figure 5.41: Editing the availability test—Edit alert

8. Clicking on the **Edit alert** button in the previous step will open up the **Rules management** blade, as shown in *Figure* 5.42. As described in the **CONDITION** section, an email will be sent to the listed recipients whenever the ping test fails in three or more selected locations:

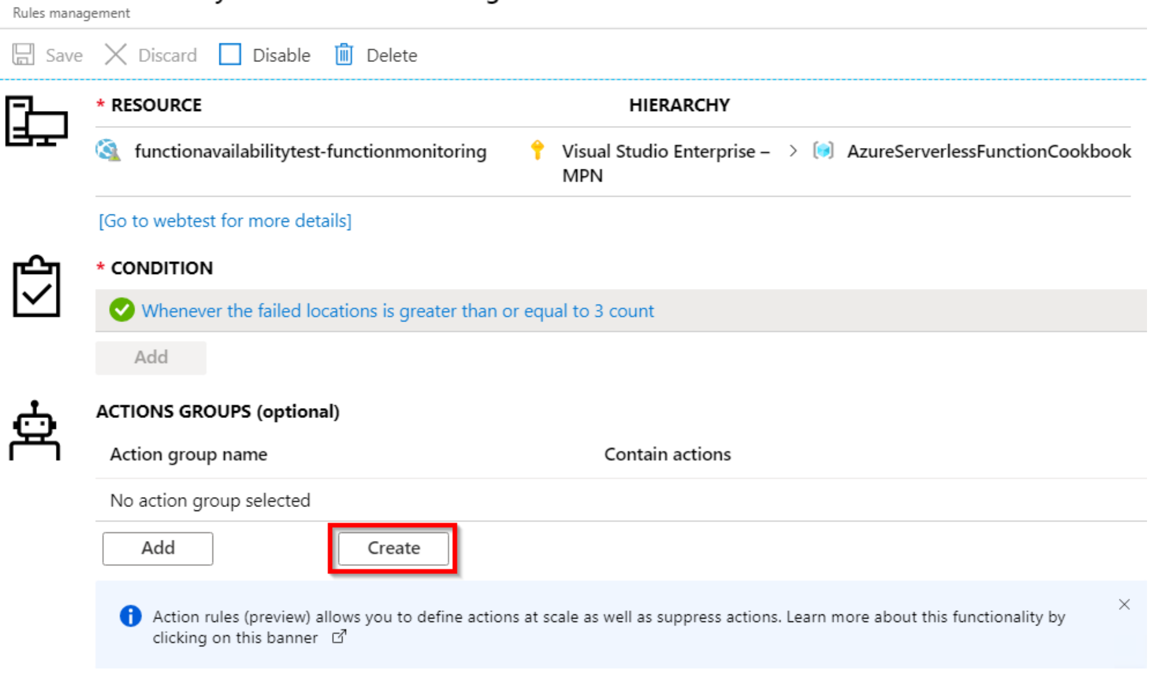


Figure 5.42: Availability tests—creating action groups

9. Our goal is to send a notification alert to the recipients whenever the condition (specified in the **CONDITION** section) meets the criteria. At the time of writing, Application Insights allows us to send the following types of notification:

- Email
- SMS
- Push notification
- Voice

10. Application Insights also allows us to invoke other apps/services in the event of failure. *Figure* 5.43 shows the different Azure services that are currently available to be invoked:

Automation Runbook

Azure Function

Email Azure Resource Manager Role

Email/SMS/Push/Voice

ITSM

LogicApp

Secure Webhook

Webhook

Figure 5.43: Availability tests—creating action groups—choosing a notification service

11. To make it simple, we'll choose to send an email alert whenever the ping test fails for three or more locations. In order to configure this, we need to create an action group. This can be achieved by clicking on the **Create** button of the **Rules management** blade. We will then be taken to the **Add action group** blade, as shown in *Figure* 5.44. Provide a name and choose the **Email/SMS/Push/Voice** item in the **Action Type** dropdown:

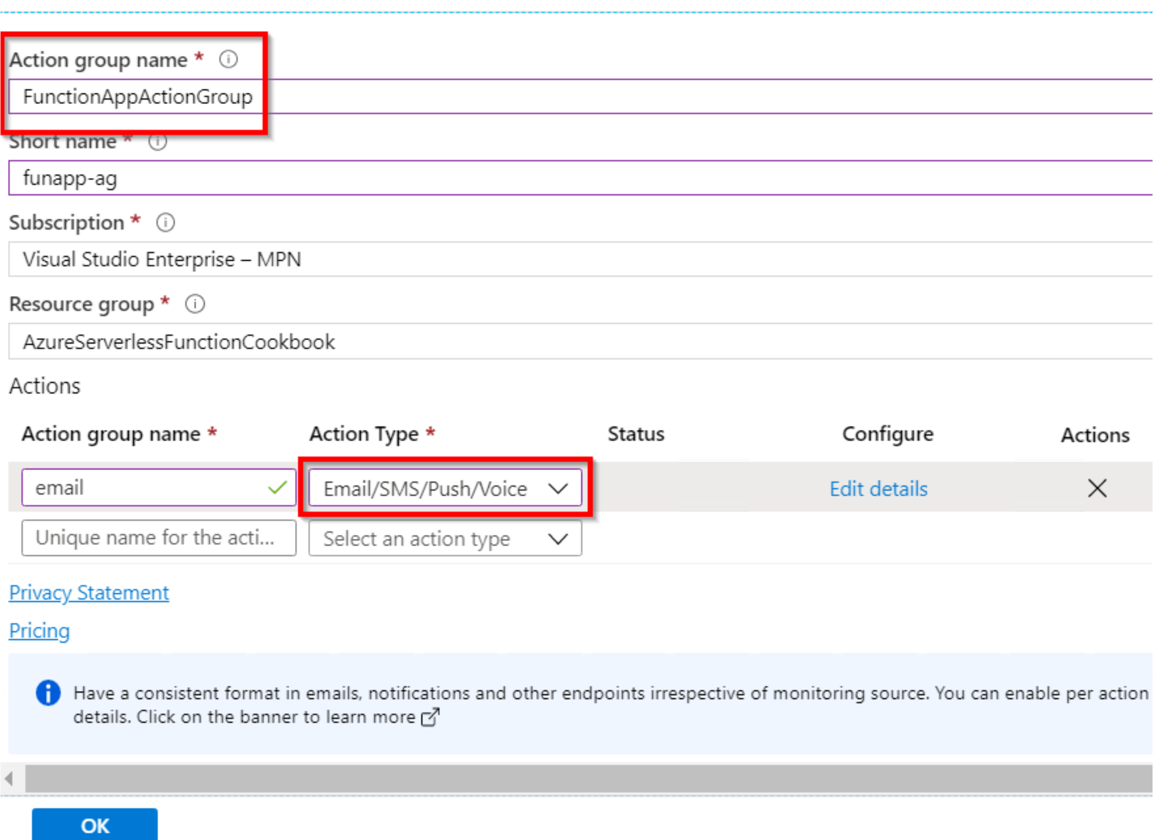


Figure 5.44: Availability tests—creating action groups

12. As soon as the action type is selected, we will be shown a new blade where we can check the **Email** option and provide a valid email address, as shown in *Figure* 5.45, and click **OK**:

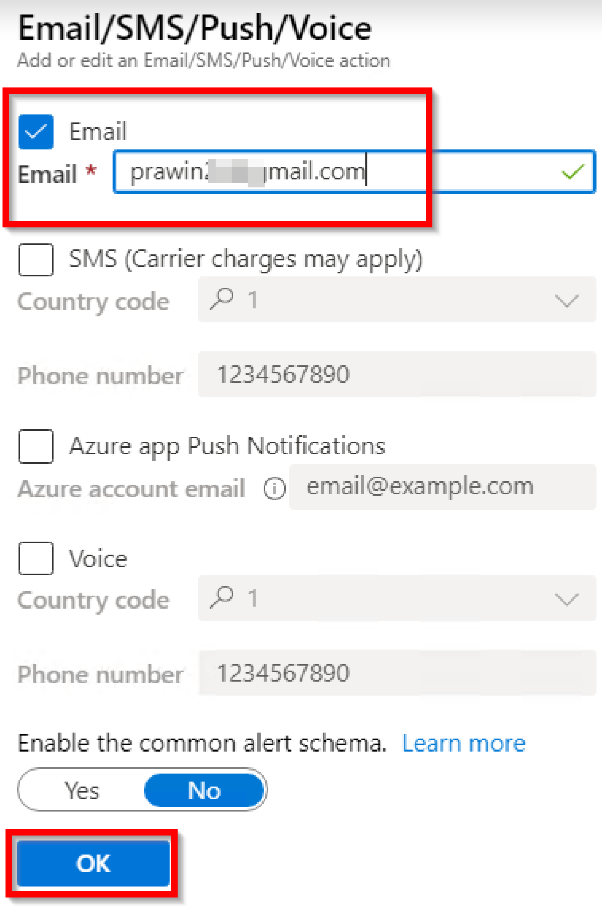


Figure 5.45: Availability tests—creating action groups—choosing email notification

13. That's it. We have now configured the action group:

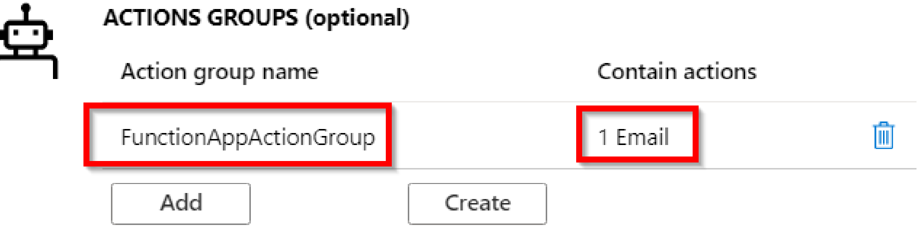


Figure 5.46: Availability tests—viewing action groups

14. Optionally, we can also choose the email content and severity, as shown in *Figure 5.47*:

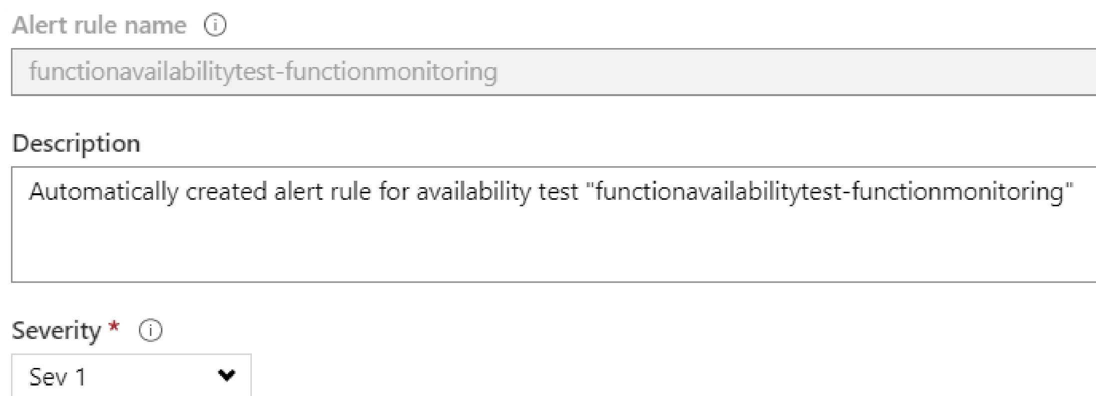


Figure 5.47: Availability tests—viewing action groups—entering email content

15. From now on, Application Insights will start doing the ping test for the locations that we selected in *step* 4. The availability of the function app can be verified as shown in *Figure 5.48*:

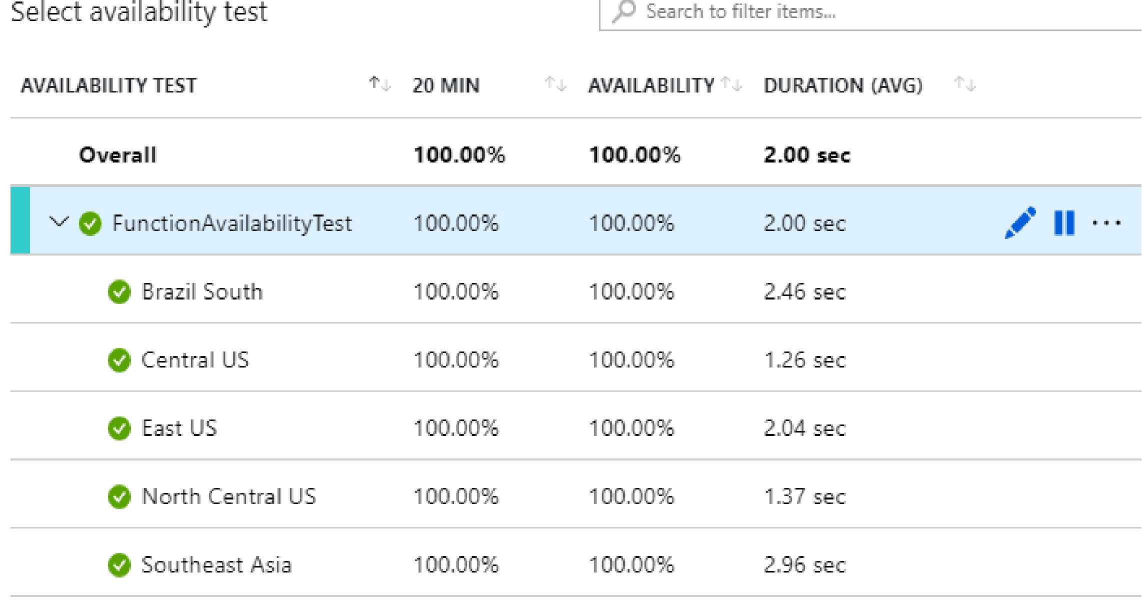


Figure 5.48: Availability tests—report

16. 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.

17. When the function app is stopped, Application Insights will try to access the function URL using the ping test. The response code will not be **Status:200 OK**, as the app was stopped, which means that the test failed. Furthermore, if it fails from three or more locations, the result will be as shown in *Figure* 5.49:

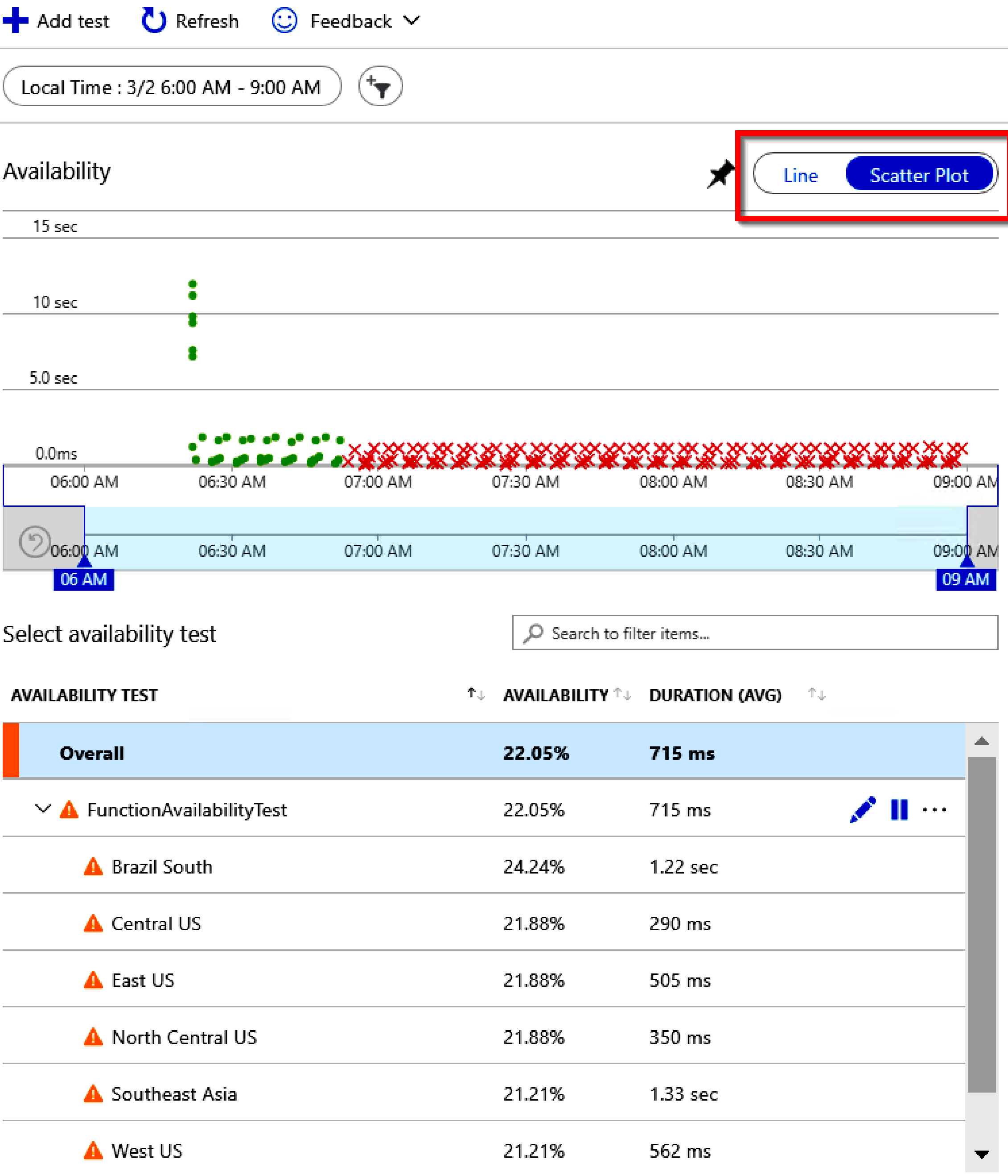


Figure 5.49: Availability tests—report—scatter plot

18. And finally, a notification should have been sent to the configured email, as shown in *Figure 5.50*:

Your Azure Monitor alert was triggered

Azure monitor alert rule functionavailabilitytest-functionmonitoring was triggered for functionavailabilitytest-functionmonitoring at March 2, 2020 7:00 UTC.

Alert rule description	Automatically created alert rule for availability test "functionavailabilitytest-functionmonitoring"
Rule ID	/subscriptions/3[illegible]3/resourcegroups/AzureServerlessFunctionCookbook/providers/microsoft.insights/metricalerts/functionavailabilitytest-functionmonitoring View Rule >
Resource ID	/subscriptions/3[illegible]/resourcegroups/AzureServerlessFunctionCookbook/providers/microsoft.insights/webtests/functionavailabilitytest-functionmonitoring View Resource >

Alert Activated Because:

See in the Azure portal >

Figure 5.50: Azure Monitor—availability test—email alert

How it works...

We have created an availability test, where our function app will be pinged once every 5 minutes from five different locations across the world. We can configure them in the **Test Locations** section of the **Create test** blade while creating the test. The default criterion of the ping is to check whether 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...

We can use a multi-step web test (using the **Test Type** option in the **Basic Information** section of the **Create test** blade) to test a page or functionality that requires navigation to multiple pages.

In this recipe, we have learned how to create an availability test that can be used to do a ping test to the function app and send alerts if the function app doesn't respond.

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 functions that we have developed here have been straightforward but, in your real-world applications, it may not be that simple as there will likely be many changes to the code that was initially created. It's good practice to write automated unit tests that help test the functionality of our Azure functions. Every time you run these automated unit tests, you can test all the various paths within the code.

In this recipe, we'll 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'll be using the Moq mocking framework and `xunit` to develop automated unit test cases for our Azure function. Having a basic working knowledge of Moq is a requirement for this recipe. Learn more about Moq at https://github.com/moq/moq4/wiki.

In order to make the unit test case straightforward, the lines of code that read the data from the **post** parameters to the **Run** method of **HTTPTriggerCSharpFromVS HTTPTrigger** have been highlighted in bold, as shown in the following code:

```
[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...

In order to complete this recipe, perform the following steps:

1. Create a new unit test project by right-clicking on the solution and then selecting **Add a new project** from the menu items. In the **Add a new project** window, choose **Test** in the project type list and choose **xUnit Test Project (.NET Core)** in the list of projects, as shown in *Figure* 5.51. Click **Next** and create the project:

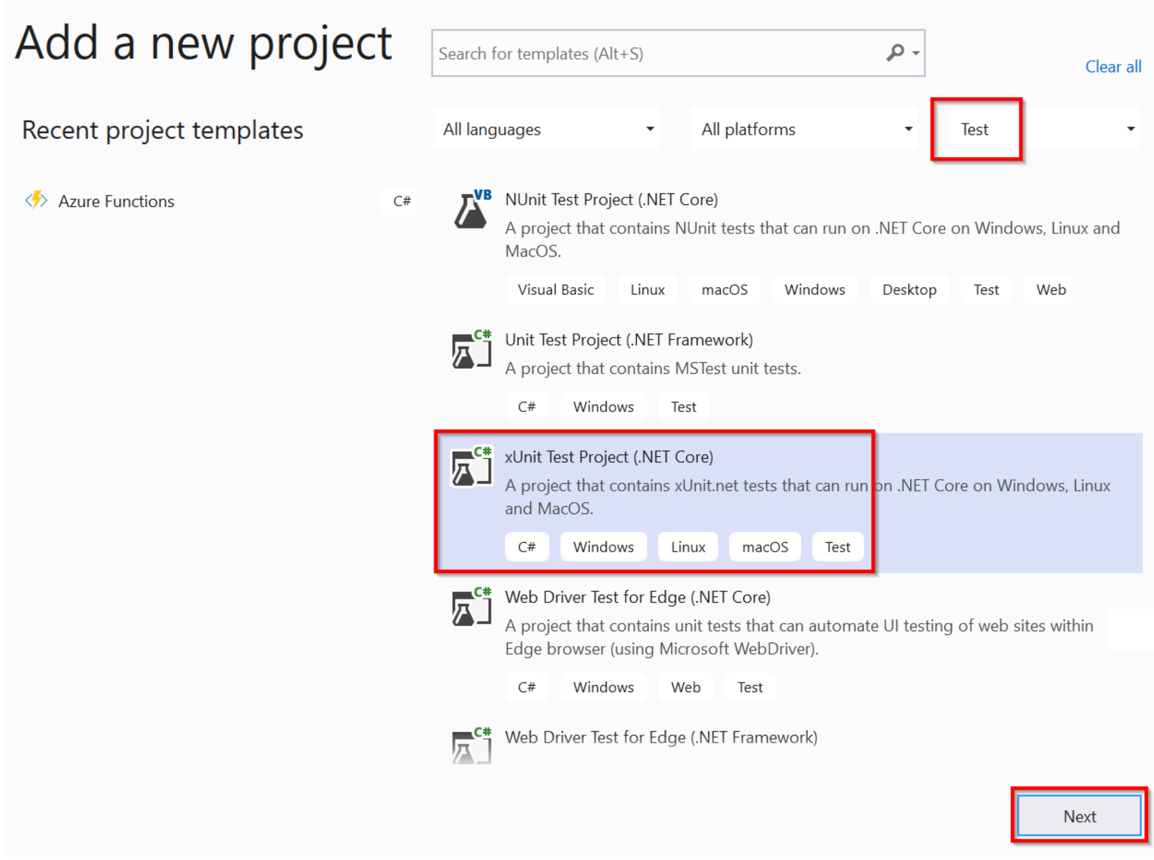


Figure 5.51: Visual Studio—adding a new xUnit project

2. Make sure that you choose **xUnit Test Project (.NET Core)** in the **Package Manager** console and install the Moq NuGet package using the following commands:

```
Install-Package Moq
```

In the unit test project, we also need the reference to the Azure function 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.

3. Add all the required namespaces to the unit test class 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);
        }
    }
}
```

4. Now, right-click on the unit test and click on **Run Test(s)**, as shown in *Figure* 5.52:

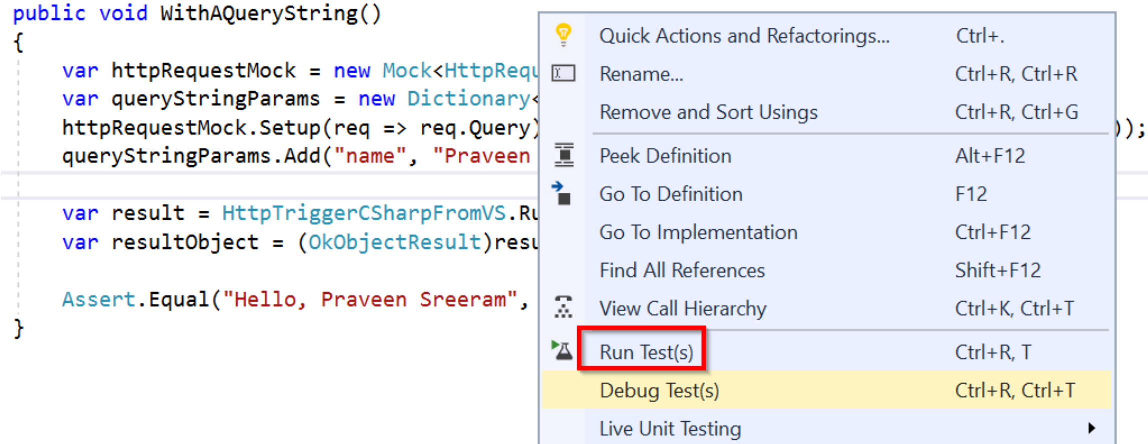


Figure 5.52: Visual Studio—running unit tests

5. If everything is set up correctly, the tests should pass, as in *Figure* 5.53:

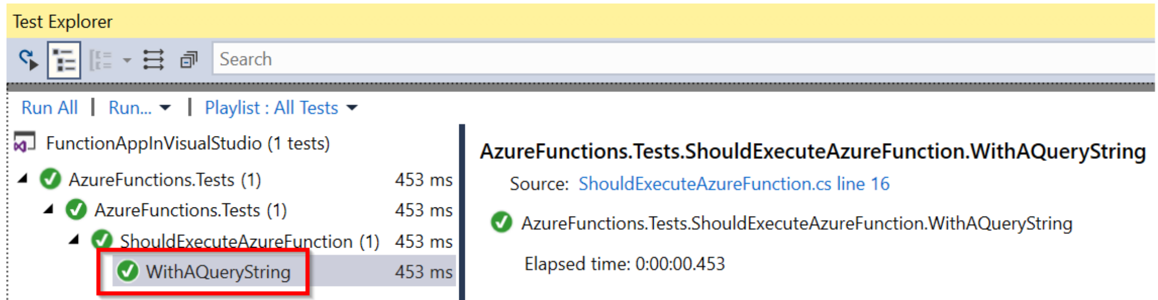


Figure 5.53: Visual Studio—viewing the unit test results

We have learned how to develop unit tests to validate Azure function code.

In this chapter, we have learned about various Azure services, tools, and features that can be leveraged for the testing and validation of Azure function apps and the monitoring of function app availability.

In the next chapter, we'll explore troubleshooting and monitoring function apps.

6

Troubleshooting and monitoring Azure Functions

In this chapter, you'll learn about the following:

- Troubleshooting Azure Functions
- Integrating Azure Functions with Application Insights
- Monitoring Azure Functions
- Pushing custom metrics details to Application Insights Analytics
- Sending application telemetry details via email
- Integrating Application Insights with Power BI using Azure Functions

Introduction

When it comes to application development, the development of a project and getting the application live is not the end of the story. It requires continuous monitoring of applications, analysis of its performance, and log reviews to predetermine issues that end users may face.

In this regard, Azure provides multiple tools to meet all of our monitoring requirements, right from the development stage through to the maintenance stage.

In this chapter, you'll learn how to utilize these tools and take the necessary action based on the information available. The following is an overview of what we'll cover in this chapter:

- Troubleshooting and fixing errors in Azure Functions
- Integrating Application Insights with Azure Functions to push the telemetry data
- Configuring email notifications to receive a summary of the errors, if any

Troubleshooting Azure Functions

In the world of software development, troubleshooting is a continuous process for identifying errors in applications. Troubleshooting is a very common practice that every developer should know how to apply in order to resolve errors and ensure that the application works as expected. Azure allows us to log information that will assist us with troubleshooting.

In this recipe, you'll learn how to view and interpret the application logs of our Function Apps using the Azure Functions log streaming feature.

How to do it...

Once we are done with development and have tested the apps thoroughly in our local environment, it's time to deploy them to Azure. There may be instances where we encounter issues after deploying an application to Azure, mainly due to incompatibility with the environment. For example, a developer might have missed out on creating app settings in the app. With a missing configuration key, the end product may not only produce faults, but it may also prove difficult to troubleshoot the error. In this recipe, you'll learn not only how to view real-time logs, but also how to use the **Diagnose and solve problems** feature.

Viewing real-time application logs

In this section, we are going to view the real-time application logs using the **Logs** feature provided by the Azure portal. We can achieve it by performing the following steps:

1. Navigate to **Platform features** of the function app and click on the **Log Streaming** button, where the **Application logs** can be seen, as shown in *Figure 6.1*:

```
Logs   Console                                   Reconnect  Copy logs  Pause  Clear  Expand
020-04-06T05:42:48.107 [Information] Executing 'Functions.HttpTriggerTestUsingPostman' (Reason='This function was
rogrammatically called via the host APIs.', Id=a5c61c18-386e-49b3-b1d3-1fceb054127a)
020-04-06T05:42:48.107 [Information] C# HTTP trigger function processed a request.
020-04-06T05:42:48.108 [Information] Executed 'Functions.HttpTriggerTestUsingPostman' (Succeeded, Id=a5c61c18-386e-
9b3-b1d3-1fceb054127a)
```

Figure 6.1: Azure Functions—Application Logs

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

2. Now, let's open any of the previously created Azure functions 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, as shown in *Figure 6.2*:

```
run.csx   Save   ▶ Run   </> Get function URL
1  #r "Newtonsoft.Json"
2
3  using System.Net;
4  using Microsoft.AspNetCore.Mvc;
5  using Microsoft.Extensions.Primitives;
6  using Newtonsoft.Json;
7
8  public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
9  {
10     log.LogInformation("C# HTTP trigger function processed a request.");
11
12     string name = req.Query["name"];
13
14     throw new Exception ("Object reference not set to an instance of an object.");
15
16     string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
17     dynamic data = JsonConvert.DeserializeObject(requestBody);
18     name = name ?? data?.name;
19
20     return name != null
21         ? (ActionResult)new OkObjectResult($"Hello, {name}")
22         : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
23 }
24
```

Figure 6.2: Azure Functions code editor—adding an exception

3. Subsequently, click on the **Save** button and then click on the **Run** button. Here, an exception is expected along with the message in the **Application logs** section, as shown in *Figure* 6.3:

Application logs | Web Server logs

```
08T03:39:01.645 [Information] Executing 'Functions.HttpTriggerUsingPostman' (Reason='This function was programmatically called via the host APIs.', Id=f0f13139-37c5-4453-9f22-
efb4)
08T03:39:01.776 [Information] C# HTTP trigger function processed a request.
08T03:39:01.777 [Error] Executed 'Functions.HttpTriggerUsingPostman' (Failed, Id=f0f13139-37c5-4453-9f22-87ae6b80efb4)Object reference not set to an instance of an object.
08T03:39:01.645 [Information] Executing 'Functions.HttpTriggerUsingPostman' (Reason='This function was programmatically called via the host APIs.', Id=f0f13139-37c5-4453-9f22-
efb4)
08T03:39:01.776 [Information] C# HTTP trigger function processed a request.
08T03:39:01.805 [Error] Executed 'Functions.HttpTriggerUsingPostman' (Failed, Id=f0f13139-37c5-4453-9f22-87ae6b80efb4)Object reference not set to an instance of an object.
```

Figure 6.3: Azure Functions—Application Logs

Once you have retrieved the application log, go ahead with diagnosing and solving the problems in the function app.

Diagnosing the function app

In the preceding section, you learned how to monitor application errors in real time, which will be helpful when it comes to quickly identifying and fixing any errors that you encounter. However, it is not always possible to monitor application logs and understand the errors encountered by the end users. Addressing this specific issue, Azure Functions provides another great tool, called **Diagnose and solve problems**:

We'll perform the following steps to diagnose the Azure Function app:

1. Navigate to **Platform features** and click on **Diagnose and solve problems**, available in the **Resource management** section, as shown in *Figure* 6.4:

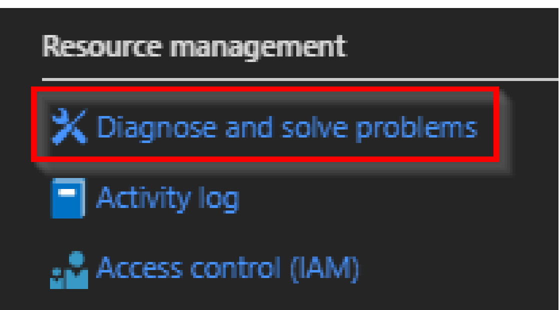


Figure 6.4: Azure Function App—Diagnose and solve problems

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

2. Soon after, we'll be taken to another pane to select the right category for the problems to be troubleshooted. Click on **5xx Errors** to view details regarding the exceptions that end users are facing, as shown in *Figure* 6.5:

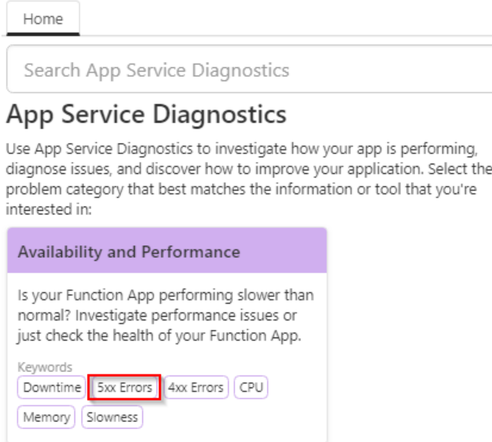


Figure 6.5: Azure App Service Diagnostics

3. From the list of tiles, click on the **Messaging Function Trigger Failure** tile and then click on the **Function Executions and Errors** link, as shown in *Figure* 6.6:

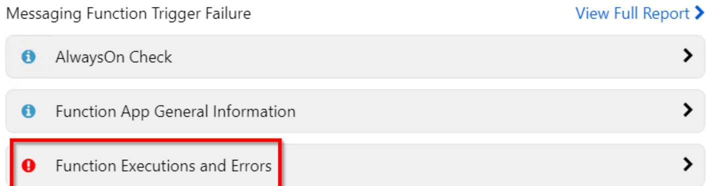


Figure 6.6: Function Executions and Errors

4. Click on **Function Executions and Errors** to view the detailed exceptions, as shown in *Figure* 6.7:

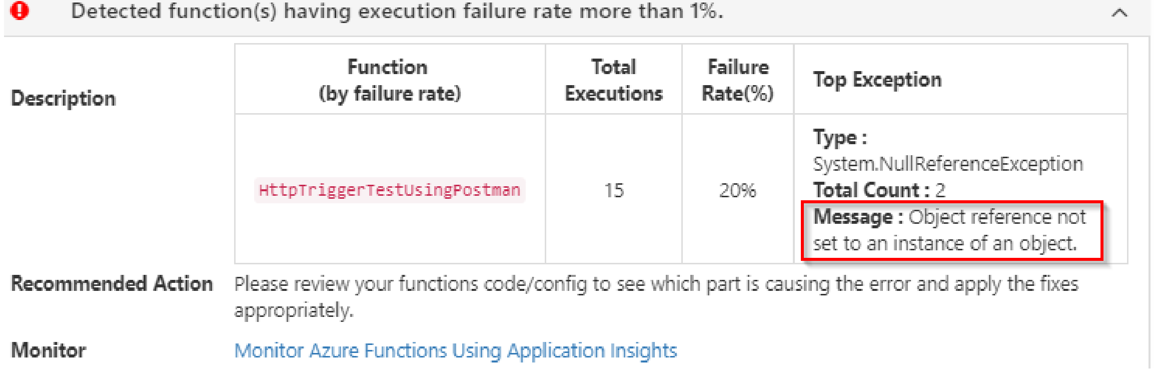


Figure 6.7: Viewing exceptions

In this recipe, you have learned how to use the diagnose and solve problems tool, which is available within the App Service context. Let's now move on to the next recipe to learn what Application Insights is and how to integrate it with Azure Functions.

Integrating Azure Functions with Application Insights

Application Insights is an **Application Lifecycle Management** (**ALM**) tool that assists with tracking performance, exception monitoring, and also collecting telemetry data of the applications. In order to leverage the features of Application Insights, we need to integrate Azure Functions with Application Insights. Once Application Insights is integrated into the application, it will start sending telemetry data to our Application Insights account, which is hosted on the cloud. This recipe will focus on integrating Azure Functions with Application Insights.

Getting ready

We created an Application Insights account in the *Validating Azure function responsiveness using Application Insights* recipe of *Chapter 5*, *Exploring testing tools for Azure functions*. Use an existing account or create an account using the following steps. If an Application Insights account was created in the previous recipe, ignore this step:

1. Navigate to **Azure Management Portal**, click on **Create a resource**, and then select **IT & Management Tools**.
2. Choose **Application Insights** and provide all the required details.

How to do it...

Once the Application Insights account has been created, perform the following steps:

1. Navigate to the **Overview** tab and copy the **Instrumentation Key**, as shown in *Figure 6.8*:

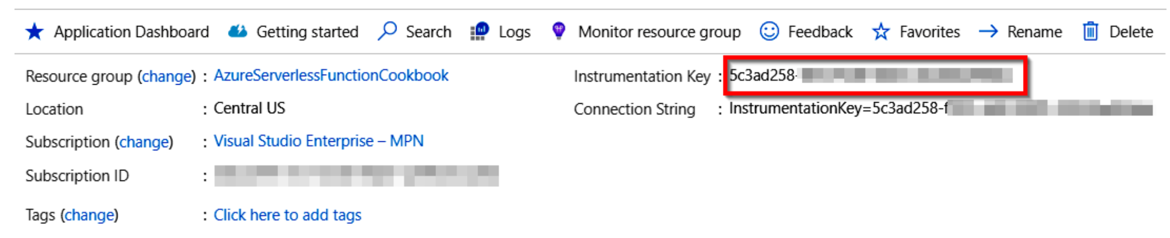


Figure 6.8: Application Insights—Overview

2. Navigate to the function apps for which you want to enable monitoring and go to the **Configuration** pane.

3. Add a new key (if it doesn't exist already) with the name **`APPINSIGHTS_INSTRUMENTATIONKEY`** and provide the instrumentation key that was copied from the Application Insights account, as shown in *Figure* 6.9, and then click on **Save** to save the changes:

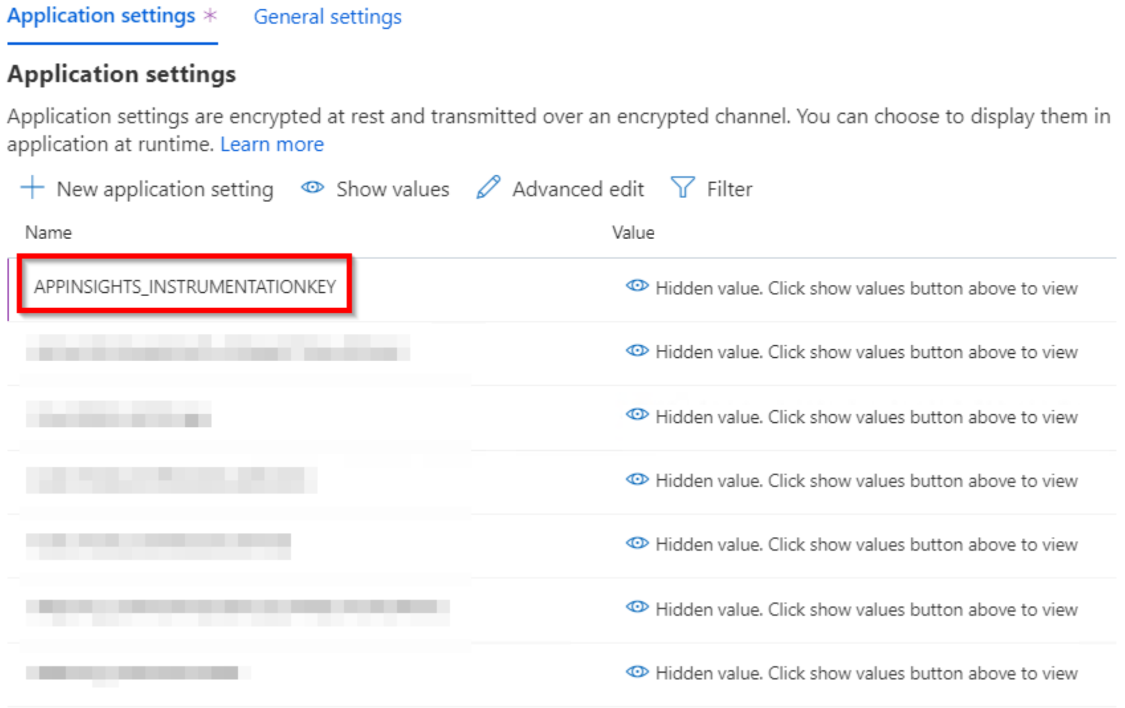


Figure 6.9: Azure Functions—Application settings

4. That's it. Let's now utilize all the features of Application Insights to monitor the performance of our 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 the web browser, as shown in *Figure 6.10*:

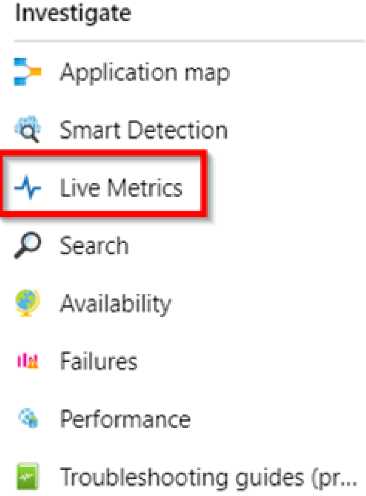


Figure 6.10: Application settings—Live Metrics menu item

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

5. After completing the tests, go to the browser tab that has Application Insights. The live traffic going to our function app should be displayed, as shown in *Figure 6.11*:

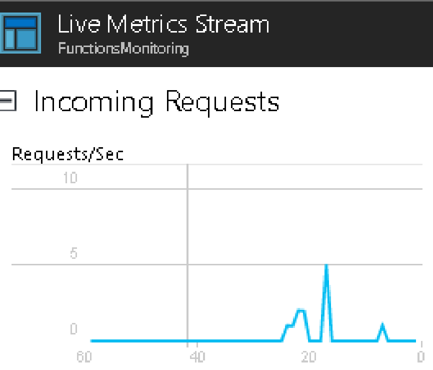


Figure 6.11: Application settings—Live Metrics Stream

How it works...

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

There's more...

Live Metrics Stream also allows us to view all the instances, along with some other data, such as the number of requests per second handled by each instance.

In this recipe, you have learned how to integrate Azure Functions with the Application Insights service. You have also seen the requests in the Live Metrics Stream to confirm whether integration has been implemented properly. Let's move on to the next recipe to learn more on how to monitor Azure Functions.

Monitoring Azure Functions

Monitoring Azure Functions is important if you want to know whether there are any errors that are raised by the application during testing.

In this recipe, you will learn how to view the logs that are written to Application Insights by Azure Functions' code. As a developer, this knowledge can help troubleshoot any exceptions that may occur during application development.

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

How to do it...

In this recipe, we'll learn how to review the application traces using Application Insight's **Logs**. Let's perform the following steps:

1. Navigate to the HTTP trigger that we created and replace the following code. I have 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 HTTP trigger 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. Any input can be used. The **Logs** console will show the following output:

Figure 6.12: Azure Functions—log information in the console

3. The logs in the preceding **Logs** console are only available when we are connected to the **Logs** console, which are not available offline. That's where **Application Insights** comes in handy. Navigate to the **Application Insights** instance that is integrated with the function app:

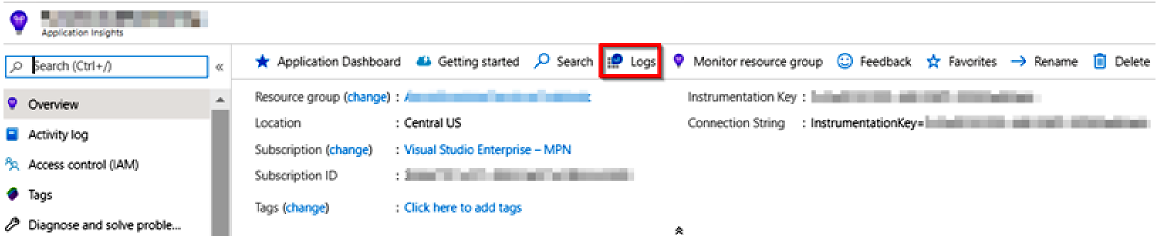


Figure 6.13: Application Insights—Overview pane

4. In the **Logs** query window, type the following **Kusto Query Language** (**KQL**) command:

```
traces
| where message contains "Azure Test"
| sort by timestamp desc
```

This will return all the traces sorted by date in descending order, as shown in *Figure 6.14*:

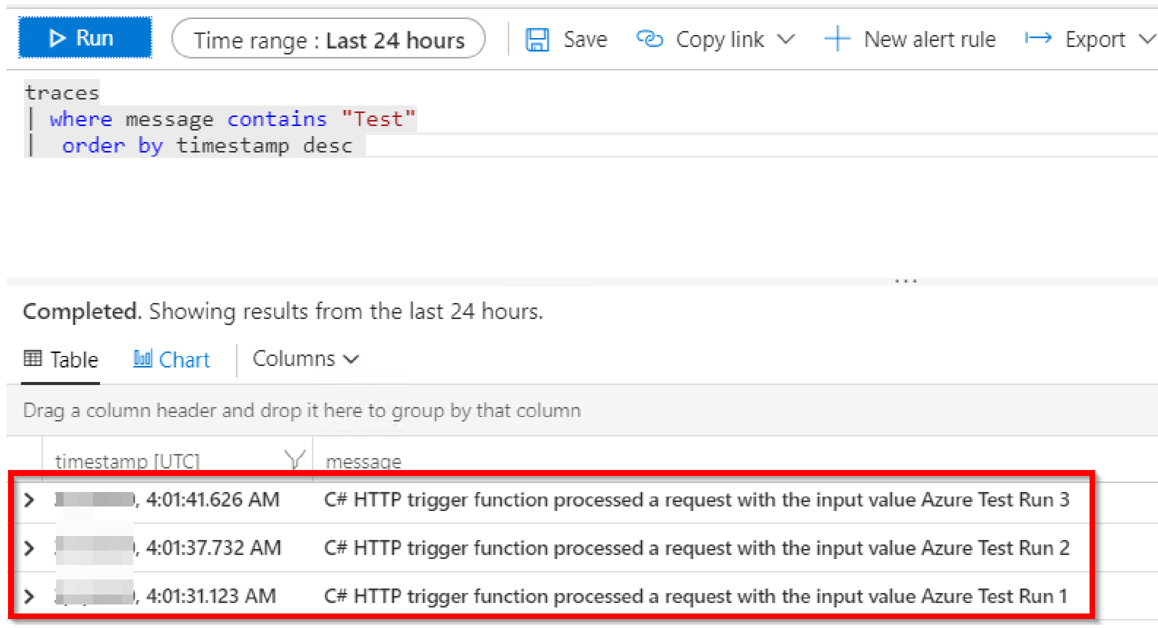


Figure 6.14: Application Insights—Logs

How it works...

In the HTTP trigger, add a `log` statement that displays the value of the `name` parameter that the user provides. In order to simulate a genuine end user, run the HTTP trigger a few times using different values. And after some time (around five minutes), click on the **Logs** button in the **Application Insights** button, which opens the analytics window. Here, we 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.

In this recipe, you have seen how to monitor the logs and write queries using Application Insights. Let's now move on to the next recipe to learn how to push custom metrics to Application Insights.

Pushing custom metrics details to Application Insights Analytics

At times, businesses may ask developers to provide analytics reports for a derived metric within Application Insights. So, what is a derived metric? Well, by default, Application Insights provides us with many insights into metrics, such as requests, errors, and exceptions.

We 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 to build a new report within Application Insights, we need to feed Application Insights data regarding the newly derived metric on a regular basis. Once the required data is fed regularly, Application Insights will take care of providing reports for our analysis.

We'll be using Azure Functions to feed Application Insights with a derived metric named **`requests per hour`**:

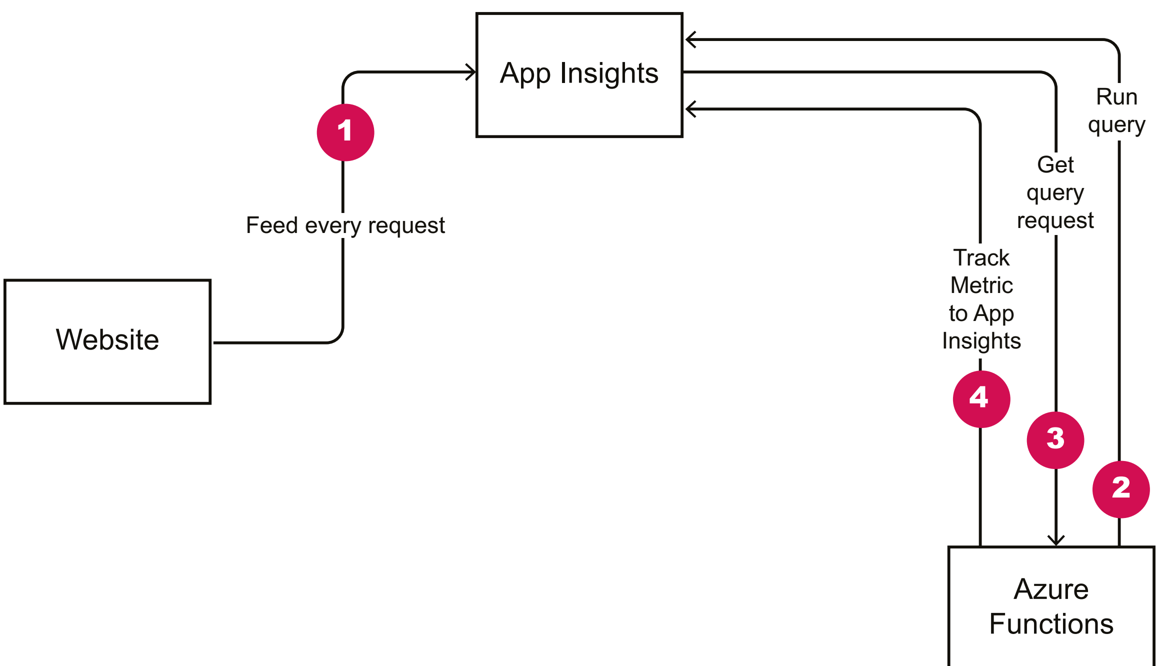


Figure 6.15: Feed Application—derived metrics to App Insights using Azure Functions

The following is a brief explanation of *Figure 6.15*:

1. The application (**Website**) feeds the request data to Application Insights for every request.
2. Azure Function timer triggers run continuously every five minutes and submit the query to **Application Insights**.
3. **Azure Functions** retrieves the results of the query in **Application Insights**.
4. This result is then used by **Application Insights** to calculate a custom metrics (requests for hour), which is pushed again to **Application Insights**.

For this example, we'll develop a query using the **Analytics Query Language** for the **`request per hour`** derived metric. We can make changes to the query to generate other derived metrics based on our requirements, such as identifying **`requests per hour`** for campaigns.

> **Note**
>
> In this recipe, we'll use KQL to query the data of Application Insights. KQL is a kind of SQL language that is used to make read-only requests to process data and return the results. Learn more about KQL at https://docs.microsoft.com/azure/application-insights/app-insights-analytics-reference.

Getting ready

We'll have to perform the following steps prior to starting with the recipe:

1. Create a new Application Insights account, if you do not already have one.
2. Make sure that you have a running application that integrates with Application Insights. Learn how to integrate an application with Application Insights at https://docs.microsoft.com/azure/application-insights/app-insights-asp-net.

How to do it...

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

Creating a timer trigger function using Visual Studio

In this section, we'll create an Azure Functions timer trigger that runs every minute by performing the following steps:

1. Create a new function by right-clicking on the function app project, as shown in *Figure 6.16*:

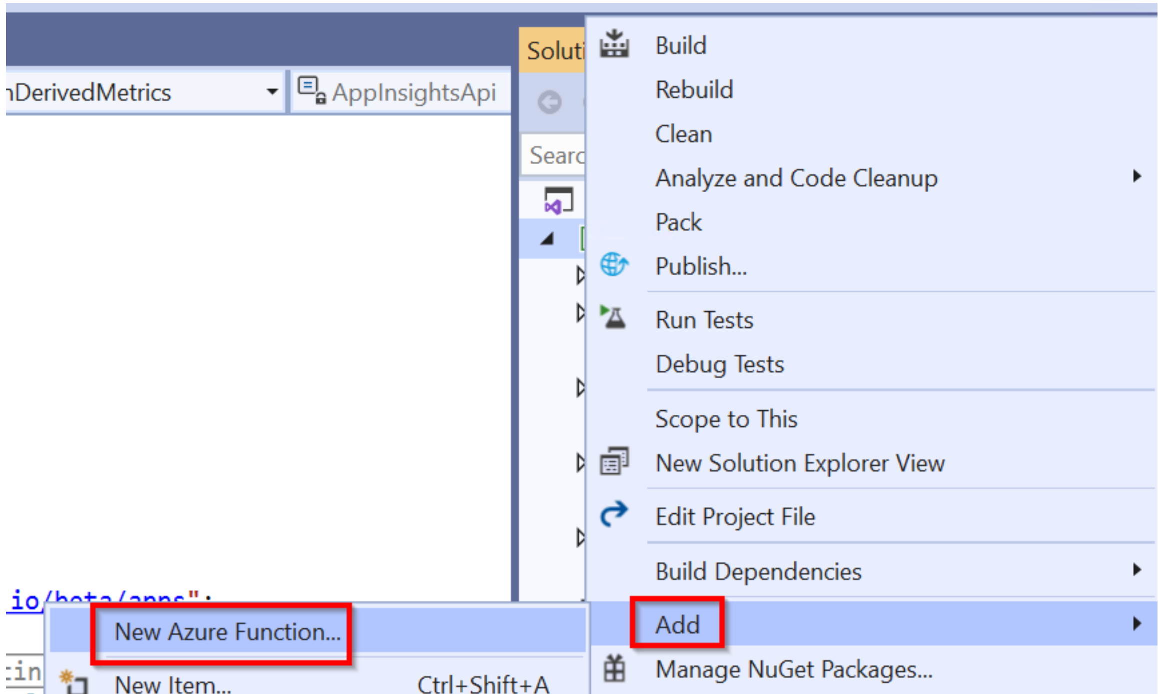


Figure 6.16: Visual Studio—adding a new Azure function

2. Now, in the **New Azure Function** window, choose **Timer Trigger** and provide the **0 */1 * * * *** CRON expression in the **Schedule** box. The CRON expression runs every minute. It can be changed later based on the frequency with which you would like to run the timer trigger. After reviewing all the details, click on the **OK** button to create the function.

3. Now, enter the following code into the new timer trigger function that you have created. The following code runs every minute (based on the CRON expression), runs a query (that you have configured) in Application Insights, and then creates a derived metric that can be used to create a custom report:

```
using System;
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;
```

```
using Newtonsoft.Json.Linq;
using System.Threading.Tasks;
using System.Net.Http;

namespace FunctionAppinVisualStudio
{
    public class FeedAIWithCustomDerivedMetrics
    {
        private const string AppInsightsApi = "https://api.
applicationinsights.io/beta/apps";


        private static readonly TelemetryClient TelemetryClient
= new TelemetryClient { InstrumentationKey = Environment.
GetEnvironmentVariable("AI_IKEY") };
        private static readonly string AiAppId = Environment.
GetEnvironmentVariable("AI_APP_ID");
        private static readonly string AiAppKey = Environment.
GetEnvironmentVariable("AI_APP_KEY");

        [FunctionName("FeedAIWithCustomDerivedMetrics")]
        public static async Task Run([TimerTrigger("0 */1 * * * *")]
TimerInfo myTimer, ILogger log)
        {
            log.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");

            await ScheduledAnalyticsRun(
                name: "Request per hour",
                query: @"requests | where timestamp > now(-1h)| summarize
count()",
                log: log);
        }
        public static async Task ScheduledAnalyticsRun(string name, string
query, ILogger log)
        {
            log.LogInformation($"Executing scheduled analytics run for
{name} at: {DateTime.Now}");


            string requestId = Guid.NewGuid().ToString();
            log.LogInformation($"[Verbose]: API request ID is
```

```
{requestId}");

            try
            {
                MetricTelemetry metric = new MetricTelemetry { Name = name
};
                metric.Context.Operation.Id = requestId;
                metric.Properties.Add("TestAppId", AiAppId);
                metric.Properties.Add("TestQuery", query);
                metric.Properties.Add("TestRequestId", requestId);
                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))
                        {
                            metric.Sum = result;
                            log.LogInformation($"[Verbose]: Metric result
is {metric.Sum}");
                        }
                        else
                        {
                            log.LogError($"[Error]: {resultJson.
ToString()}");
                            throw new FormatException("Query must result
in a single metric number. Try it on Analytics before scheduling.");
                        }
```

```
                    }
                }

                TelemetryClient.TrackMetric(metric);
                log.LogInformation($"Metric telemetry for {name} is
sent.");
            }
            catch (Exception ex)
            {

                var exceptionTelemetry = new ExceptionTelemetry(ex);
                exceptionTelemetry.Context.Operation.Id = requestId;
                exceptionTelemetry.Properties.Add("TestName", name);
                exceptionTelemetry.Properties.Add("TestAppId", AiAppId);
                exceptionTelemetry.Properties.Add("TestQuery", query);
                exceptionTelemetry.Properties.Add("TestRequestId",
requestId);
                TelemetryClient.TrackException(exceptionTelemetry);
                log.LogError($"[Error]: Client Request ID {requestId}:
{ex.Message}");


                throw;
            }
            finally
            {
                TelemetryClient.Flush();
            }
        }
    }
}
```

4. Install the Application Insights Nuget package in the Azure Function app project using the following Nuget command:

```
Install-package Microsoft.ApplicationInsights
```

Now that we have added the code, let's move on to the next section to configure the keys.

Configuring access keys

In order to have the Azure function access Application Insights programmatically, we need to create an API key. Let's configure the access keys by performing the following steps:

1. Navigate to Application Insights' **Overview** pane, and copy the **Instrumentation Key**. We'll 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'll be using this **Application ID** to create a new app setting with the name **`AI_APP_ID`** in the function app:

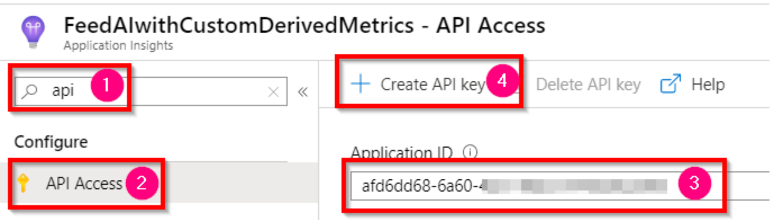


Figure 6.17: Application Insights—API Access pane

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 *Figure 6.18*. Provide a meaningful name, check the **Read telemetry** box, and click on **Generate key**:

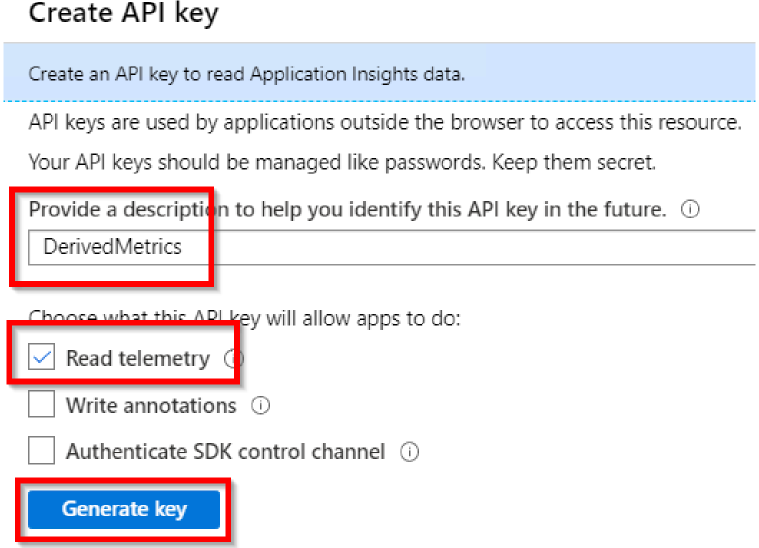
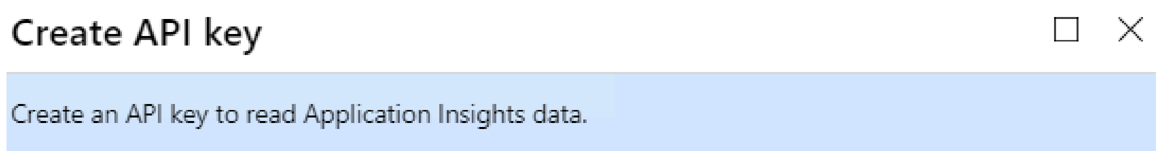


Figure 6.18: Application Insights—API Access pane—generating a new key

4. Soon after, the platform allows you to view and copy the key, as shown in *Figure 6.19*. We'll be using this to create a new app setting with the name **`AI_APP_KEY`** in our function app, so be sure to store it somewhere:



API keys are used by applications outside the browser to access this resource.

Your API keys should be managed like passwords. Keep them secret.

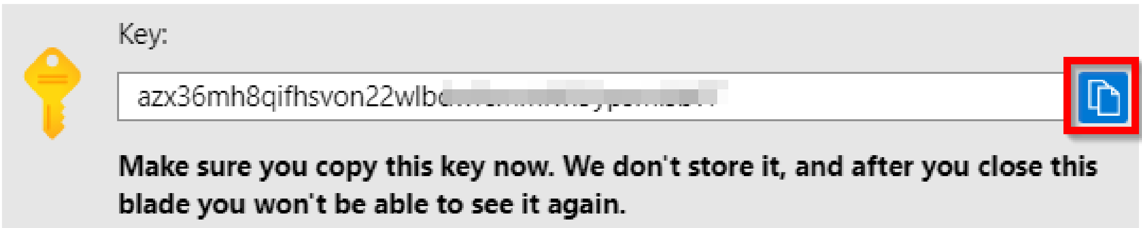


Figure 6.19: Application Insights – API Access pane—copying the new key

5. Create all three settings in the **`localsettings.json`** to perform some tests in the local environment:

```
local.settings.json
Schema: <No Schema Selected>
1  {
2      "IsEncrypted": false,
3    "Values": {
4      "AzureWebJobsStorage": "DefaultEndpointsProtocol=https;Account
        +GU                                        6jg==;En
5      "FUNCTIONS_WORKER_RUNTIME": "dotnet",
6      "AI_APP_ID": "afd6                         ",
7      "AI_APP_KEY": "azx36mh                     ",
8      "AI_IKEY": "dde9ad5a-3                     "
9    }
10 }
```

Figure 6.20: Azure Functions—configuration file

6. Create all three app setting keys in the **Configuration** pane of the function app, as shown in *Figure 6.21*. These three keys will be used in our Azure function named **`FeedAIwithCustomDerivedMetric`**:

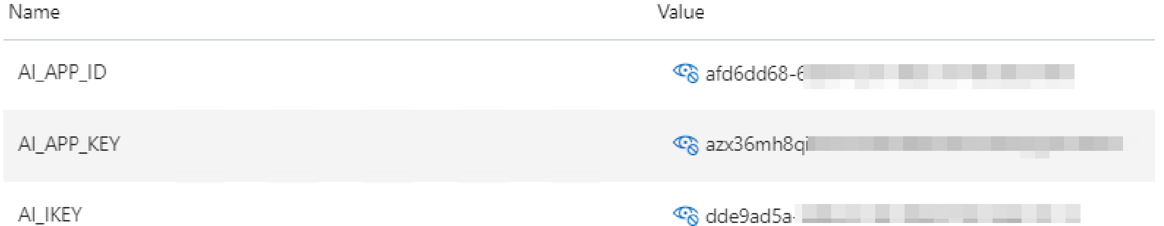


Figure 6.21: Azure Functions—configuration—App settings

We have developed the code and created all the required configuration settings. Let's now move on to the next section to test them.

Integrating and testing an Application Insights query

In this section, let's run and test an Application Insights query by performing the following steps:

1. First of all, let's test the **`requests per hour`** derived metric value. Navigate to the **Overview** pane of Application Insights and click on the **Logs** button.
2. In the **Logs** pane, write the following query in the **Query** tab, although custom queries can be written as per your requirements. Make sure that the query returns a scalar value:

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

3. Once the query is written, run it by clicking on the **Run** button to see the number of records, as shown in *Figure 6.22*:

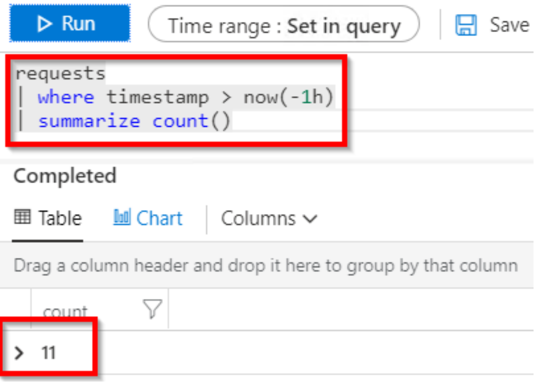


Figure 6.22: Application Insights—Logs—executing the query

4. Now, run the timer trigger for a few minutes in the local machine, which will push some data to Application Insights. We can then view the data, as shown in *Figure 6.22*.

In this section, we have learned how to develop and test an Application Insights query. Now, let's move on to the next section.

Configuring the custom-derived metric report

In this section, we'll create a custom report in Application Insights by configuring the derived metric that we have created in this recipe by performing the following steps:

1. Navigate to the Application Insights' **Overview** tab and click on the **Metrics** menu, as shown in *Figure 6.23*:

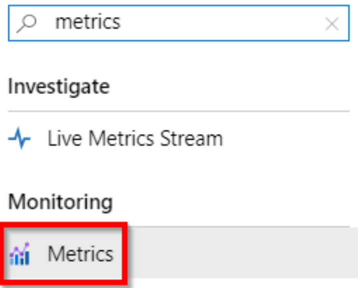


Figure 6.23: Application Insights—Metrics menu item

2. **Metrics Explorer** is where you can find analytics regarding different metrics. In **Metrics Explorer**, click on the **Add metric** button of any chart to configure the custom metric, Thereafter, you can configure the custom metric and all other details related to the chart. In the **METRIC NAMESPACE** drop-down menu, choose **azure.applicationinsights**, as shown in *Figure 6.24*, and then choose the **Request per hour** custom metric that you created:

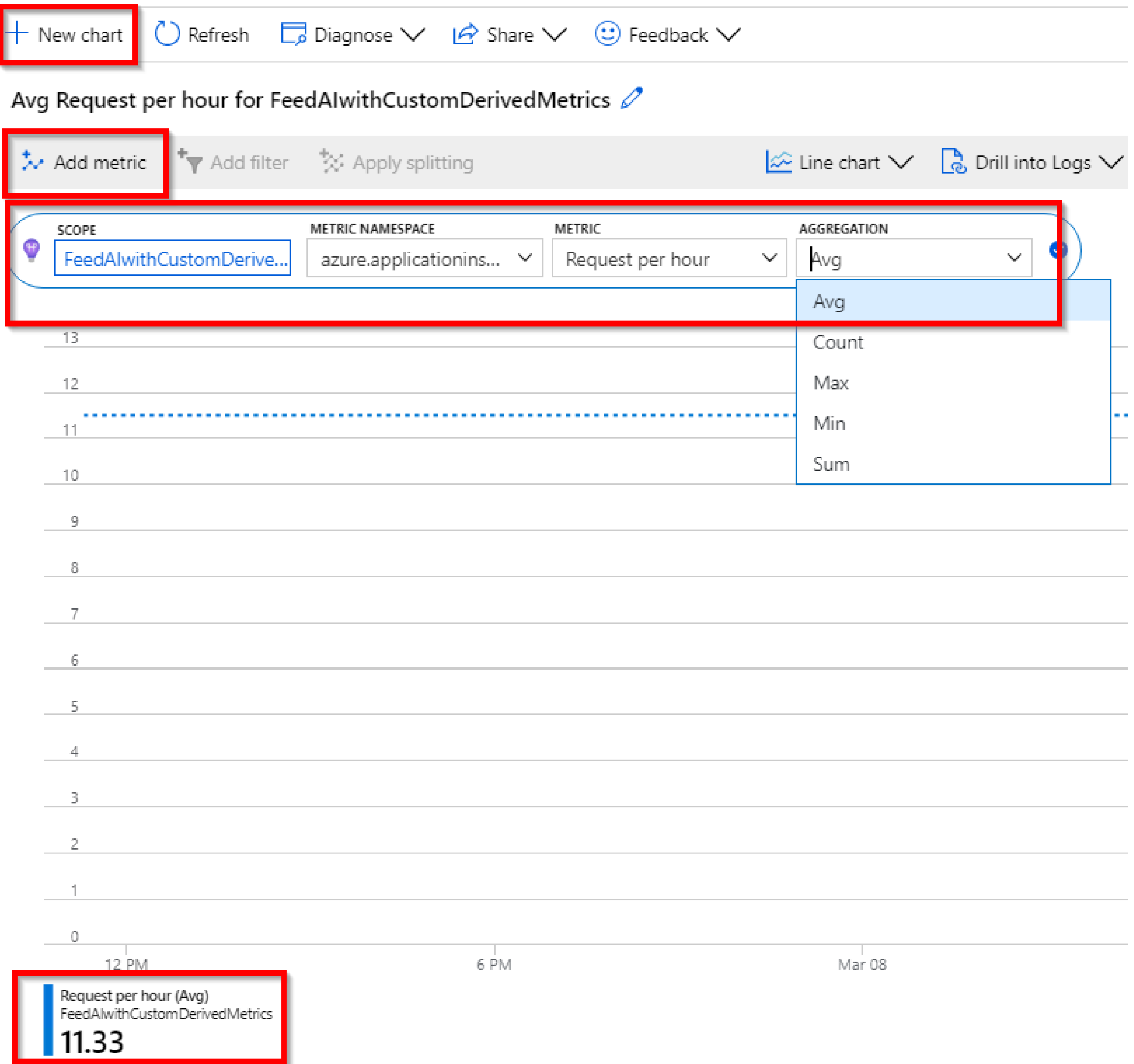


Figure 6.24: Application Insights—Metrics Explorer

How it works...

This is how the entire process works:

- We created the Azure function timer trigger using Visual Studio that runs every few minutes (one minute in this example. Feel free to change it to different values based on requirements).
- We configured the following keys in **Application settings** of the Azure Function app:

  The Application Insights' 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 fed 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 is repeated every minute.
- 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 our application, once live, will be able to receive a notification email with details regarding health, errors, response time, and so on, at least once a day.

In this recipe, we'll use the ability of Azure Functions' timer trigger to get all the required values from Application Insights and send the email using SendGrid. We'll look at how to do that in this recipe.

Getting ready

Perform the following steps in the first instance:

1. This recipe is dependent on the application settings created in the previous recipe, *Pushing custom metrics details to Application Insights Analytics*. Please ensure to add them before running the code of this recipe.
2. Create a new SendGrid account, if you do not already have one, and get the SendGrid API key.
3. Create a new Application Insights account, if one has not been created already.
4. Make sure that you have a running application that integrates with Application Insights.

> **Note**
>
> Learn how to integrate applications with Application Insights at https://docs.microsoft.com/azure/application-insights/app-insights-asp-net.

How to do it...

In this section, we'll create the application settings and also develop the query.

Configuring the application settings

To configure the application settings, perform the following steps:

1. Create the SendGrid API key in both the **appSettings.json** file of the local project as well as Azure App Settings:

   appSettings.json file:

```
"AI_APP_ID": "a
"AI_APP_KEY": "",
"AI_IKEY": "dde",
"SendGridAPIKey": "SG.5uLKU_"
```

Figure 6.25: Azure Functions—local configuration file

   App settings in the **Configuration** pane:

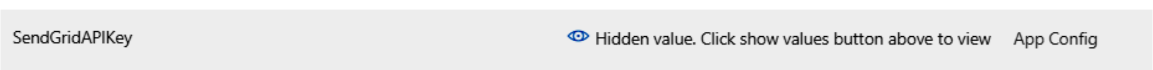


Figure 6.26: Azure Functions—configuration—App settings

Developing and validating the KQL query

In this section, we'll develop the KQL query and test it:

1. Develop the query that pulls the details regarding **Total Requests**, **Failed Requests**, and **Exceptions**. The query can be changed depending on the requirements. The following is a simple query used in this recipe:

```
requests
| where timestamp > ago(1d)
| summarize Row = 1, TotalRequests = sum(itemCount), FailedRequests =
sum(toint(success == 'False')),
RequestsDuration = iff(isnan(avg(duration)), '------',
tostring(toint(avg(duration) * 100) / 100.0))
| join (
exceptions
| where timestamp > ago(1d)
| summarize Row = 1, TotalExceptions = sum(itemCount)) on Row
| project TotalRequests, FailedRequests,TotalExceptions
```

2. Upon running the preceding query in the **Logs** section of Application Insights, the output in *Figure 6.27* will be visible:

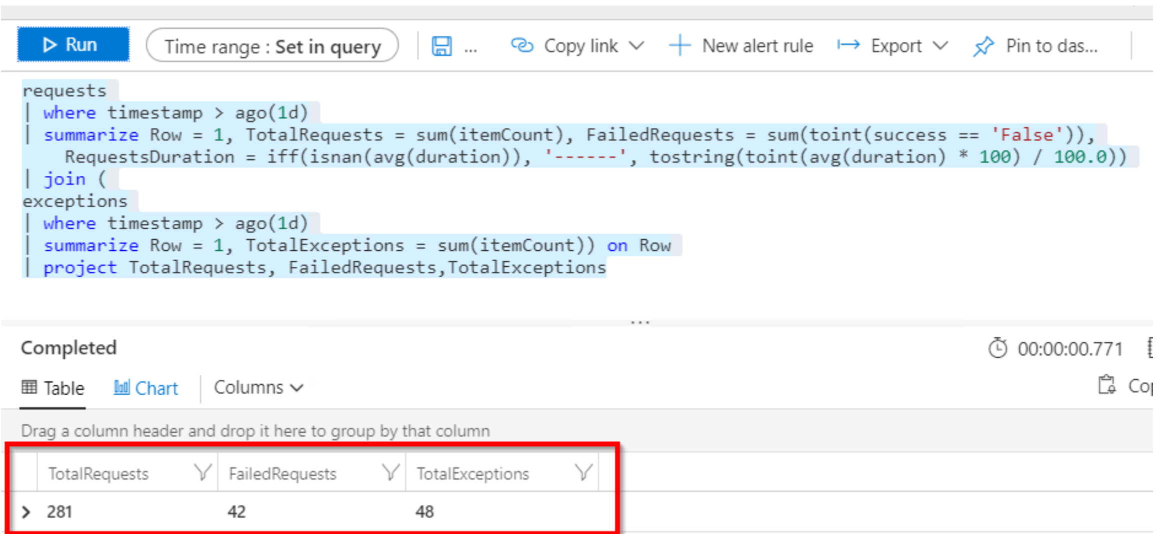


Figure 6.27: Application Insights—Logs—executing the query

We have developed the KQL query and tested it, so now let's move on to the next section.

Developing the code using the timer trigger of Azure Functions

In this section, we'll develop the timer trigger of Azure Functions, which calls the KQL query on a certain frequency (for example, every minute):

1. Create a new timer trigger function named **`ApplicationInsightsScheduledDigest`** in Visual Studio.
2. Add the following Nuget packages, if you don't have them already:

   `Microsoft.ApplicationInsights`

   `SendGrid`
3. Replace the default code with the following code. The code (as per the CRON expression) runs every minute (to make it simple, one minute is used. CRON expressions can be changed as per our requirements) to submit the query to Azure Application Insights to get the total requests, failed requests, and the exceptions. It also sends an email with all the data returned by the query to the end user:

```
using System;
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;
using System.Net.Http;
namespace FunctionAppinVisualStudio
{
    public static class ApplicationInsightsScheduledDigest
    {
        private const string AppInsightsApi = "https://api.
applicationinsights.io/v1/apps";
        private static readonly string AiAppId = Environment.
GetEnvironmentVariable("AI_APP_ID");
        private static readonly string AiAppKey = Environment.
GetEnvironmentVariable("AI_APP_KEY");
        private static readonly string SendGridAPIKey = Environment.
GetEnvironmentVariable("SendGridAPIKey");

        [FunctionName("ApplicationInsightsScheduledDigest")]
        public static async Task Run([TimerTrigger("0 */1 * * * *")]
TimerInfo myTimer, ILogger log)
        {
```

```
            log.LogInformation($"C# Timer trigger function executed at: 
{DateTime.Now}");

            string appName = "Azure Serverless Computing Cookbook";

            var today = DateTime.Today.ToShortDateString();

            DigestResult result = await ScheduledDigestRun(
                query: GetQueryString(),
                log: log
            );
            SendGridMessage message = new SendGridMessage();
            message.SetFrom(new EmailAddress("donotreply@example.com"));
            message.AddTo("prawin2k@gmail.com");
            message.SetSubject($"Your daily Application Insights digest 
report for {today}");
            var msgContent = GetHtmlContentValue(appName, today, result);
            message.AddContent("text/html", msgContent);
            var client = new SendGridClient(SendGridAPIKey);
            var response = await client.SendEmailAsync(message);
            log.LogInformation($"Generating daily report for {today} at 
{DateTime.Now}");
        }

        static string GetHtmlContentValue(string appName, string today, 
DigestResult result)
        {
            return $@"
                <html><body>
                <p style='text-align: center;'><strong>{appName} daily 
telemetry report {today}</strong></p>
                <p style='text-align: center;'>The following data shows 
insights based on telemetry from last 24 hours.</p>
                <table align='center' style='width: 95%; max-width: 
480px;'><tbody>
                <tr>
                <td style='min-width: 150px; text-align: 
left;'><strong>Total requests</strong></td>
                <td style='min-width: 100px; text-align: 
right;'><strong>{result.TotalRequests}</strong></td>
                </tr>
                <tr>
```

```
                <td style='min-width: 150px; text-align:
left;'><strong>Failed requests</strong></td>
                <td style='min-width: 100px; text-align:
right;'><strong>{result.FailedRequests}</strong></td>
                </tr>
                <td style='min-width: 150px; text-align:
left;'><strong>Total exceptions</strong></td>
                <td style='min-width: 100px; text-align:
right;'><strong>{result.TotalExceptions}</strong></td>
                </tr>
                </tbody></table>
                </body></html>
                ";
        }
        private static async Task<DigestResult> ScheduledDigestRun(string
query, ILogger log)
        {
            log.LogInformation($"Executing scheduled daily digest run at:
{DateTime.Now}");
            string requestId = Guid.NewGuid().ToString();
            log.LogInformation($"API request ID is {requestId}");
            try
            {
                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=P1W&query={query}";
                    using (var httpResponse = await httpClient.
GetAsync(apiPath))
                    {
                        httpResponse.EnsureSuccessStatusCode();
                        var resultJson = await httpResponse.Content.
ReadAsAsync<JToken>();
                        DigestResult result = new DigestResult
                        {
                            TotalRequests = resultJson.
```

```
SelectToken("tables[0].rows[0][0]")?.ToObject<long>().ToString("N0"),
                            FailedRequests = resultJson.
SelectToken("tables[0].rows[0][1]")?.ToObject<long>().ToString("N0"),
                            TotalExceptions = resultJson.
SelectToken("tables[0].rows[0][2]")?.ToObject<long>().ToString("N0")
                        };
                        return result;
                    }
                }
            }
            catch (Exception ex)
            {
                log.LogError($"[Error]: Client Request ID {requestId}:
{ex.Message}");
                throw;
            }
        }
        private static string GetQueryString()
        {
            return @"
            requests
            | where timestamp > ago(1d)
            | summarize Row = 1, TotalRequests = sum(itemCount),
FailedRequests = sum(toint(success == 'False')),
                RequestsDuration = iff(isnan(avg(duration)), '------',
tostring(toint(avg(duration) * 100) / 100.0))
            | join (
            exceptions
            | where timestamp > ago(1d)
            | summarize Row = 1, TotalExceptions = sum(itemCount)) on Row
            | project TotalRequests, FailedRequests,TotalExceptions
            ";
        }
    }
    struct DigestResult
    {
        public string TotalRequests;
        public string FailedRequests;
        public string TotalExceptions;
    }
}
```

4. *Figure 6.28* is a screenshot of the email received after the timer trigger has run:

Azure Serverless Computing Cookbook daily telemetry report 3/9/2020

The following data shows insights based on telemetry from last 24 hours.

Total requests	**281**
Failed requests	**42**
Total exceptions	**48**

Figure 6.28: Telemetry email

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.

Integrating Application Insights with Power BI using Azure Functions

Sometimes, we need to view real-time data regarding our application's availability or information relating to the 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 might need to develop a separate website and build, test, and host it somewhere.

In this recipe, you'll learn how easy it is to view real-time health information for the application by integrating Application Insights and Power BI. We'll 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'll be doing in the rest of the recipe:

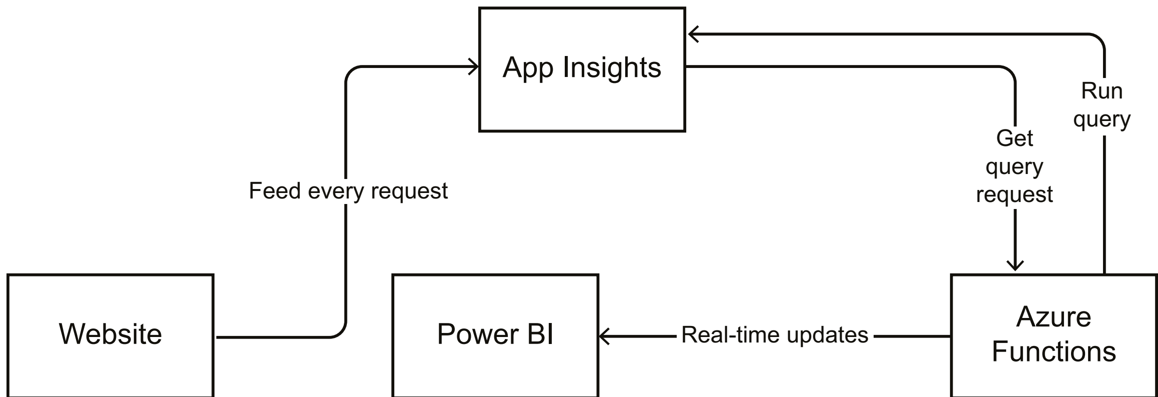


Figure 6.29: Flowchart for integrating real-time Application Insights monitoring with Power BI using Azure Functions

As depicted in the preceding flowchart, here is how the approach in this recipe works:

1. The website feed every request with the telemetry information to App Insights. In this recipe, we are not going to develop the website. However, in your projects, you will have your websites already integrated with App Insights, and this will push the telemetry.
2. An Azure Functions timer trigger will run the query in App Insights on a certain frequency to get query results from App Insights.
3. Once the results are received from Azure Functions, it will push the real-time updates to Power BI.

Getting ready

Perform the following initial steps in order to implement the functionality of this recipe:

1. Create a Power BI account at https://powerbi.microsoft.com/.
2. Create a new Application Insights account, if one has not been created already.
3. Make sure that you have a running application that integrates with Application Insights. Learn how to integrate applications with Application Insights at https://docs.microsoft.com/azure/application-insights/app-insights-asp-net.

> Note
>
> Use a work or school 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 and Yahoo.
>
> Make sure to follow the steps in the *Configuring access keys* section of the *Pushing custom metrics details to Application Insights Analytics* recipe to configure the following access keys: Application Insights instrumentation key, the application ID, and the API access key

How to do it...

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

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

In this recipe, we'll create a streaming dataset and add it to the dashboard by performing the following steps:

1. While using the Power BI portal for the first time, you might have to click on **Skip** on the welcome page, as shown in *Figure 6.30*:

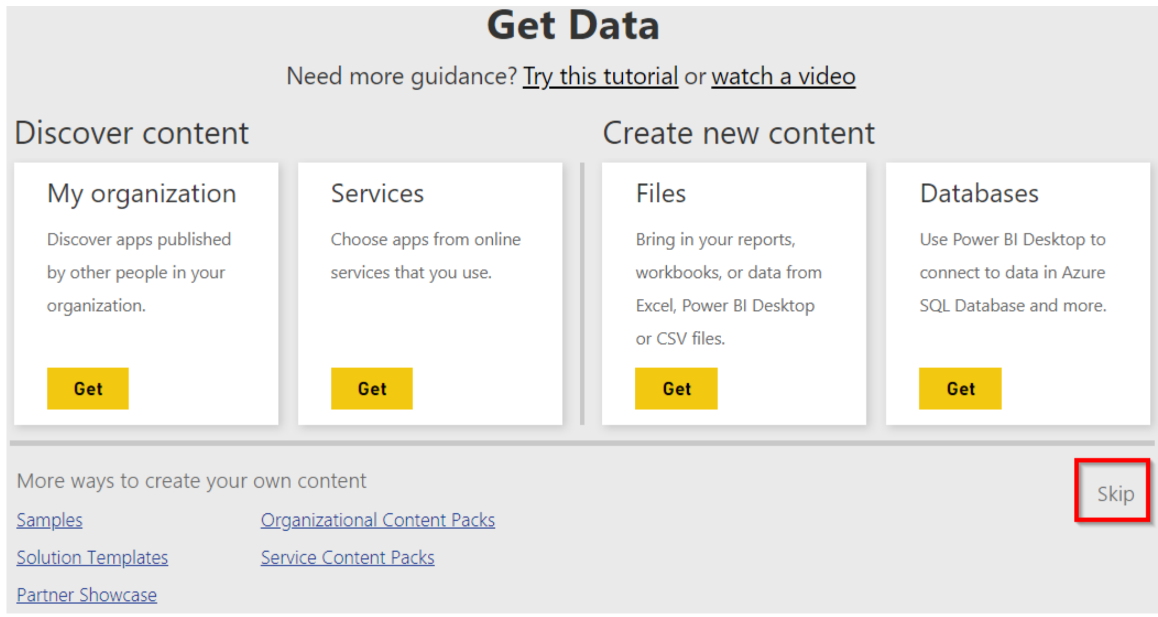


Figure 6.30: Power BI—Get Data—view

2. The next step is to create a streaming dataset by clicking on **Create** and then choosing **Streaming dataset**, as shown in *Figure 6.31*:

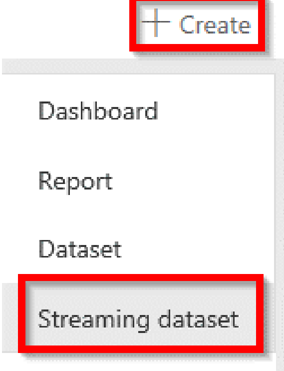


Figure 6.31: Power BI—creating a streaming dataset menu item

3. In the **New streaming dataset** step, select **API** and click on the **Next** button, as shown in *Figure 6.32*:

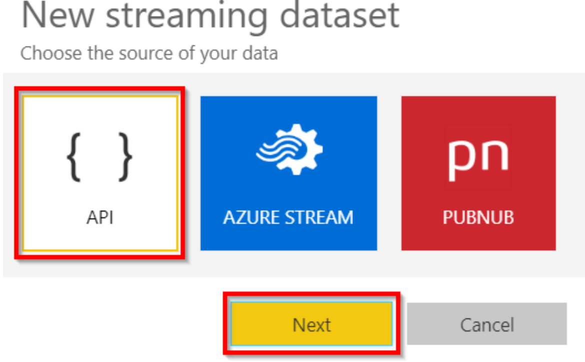


Figure 6.32: Power BI—streaming dataset—choosing the source of the data

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 to be pushed to the Power BI. For this recipe, I created a dataset with just one field, named **`RequestsPerSecond`**, of the **Number** type, and clicked on **Create**, as shown in *Figure 6.33*:

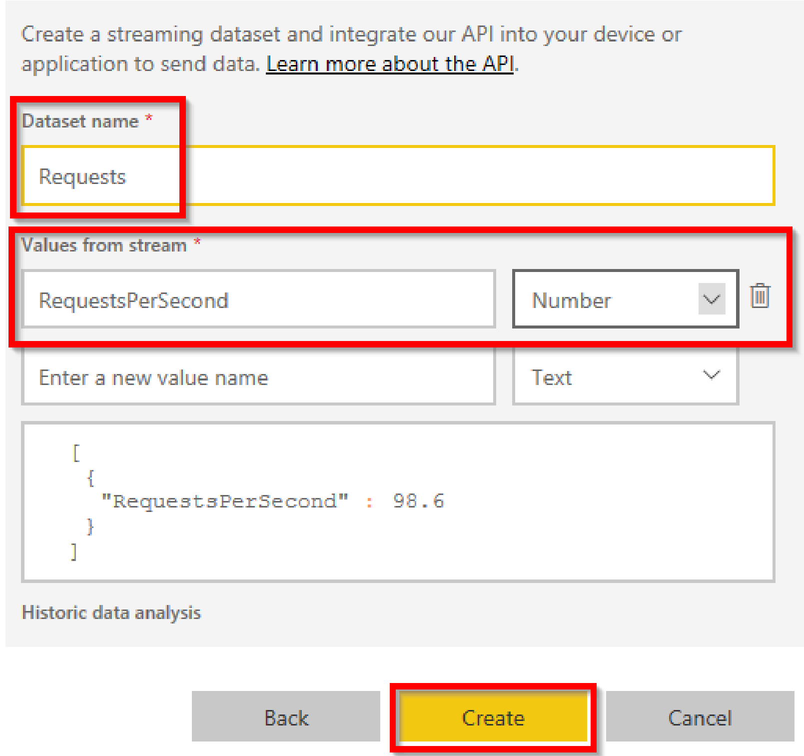


Figure 6.33: Power BI—creating a streaming dataset

5. Once the dataset is created, you'll be prompted with a push URL, as shown in *Figure* 6.34. Use this push URL in Azure Functions to push the `RequestsPerSecond` data every second (or however frequently you wish) with the actual value of requests per second. Then, click on **Done**:

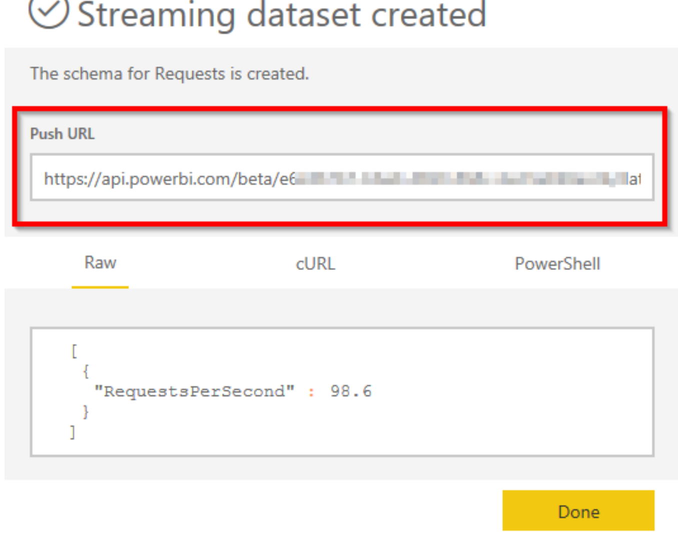


Figure 6.34: Power BI—Streaming dataset—Push URL

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 *Figure* 6.35:

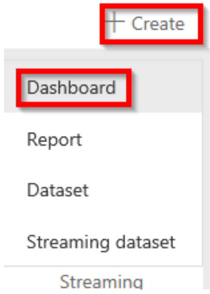


Figure 6.35: Power BI—Dashboard—creating a menu item

7. In the **Create dashboard** pop-up window, provide a meaningful name and click on **Create**, as shown in the following *Figure* 6.36, to create an empty dashboard:

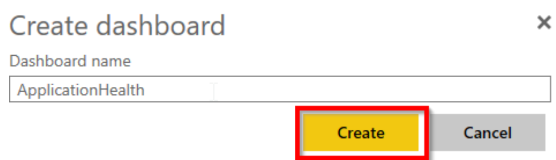


Figure 6.36: Power BI—Create dashboard

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

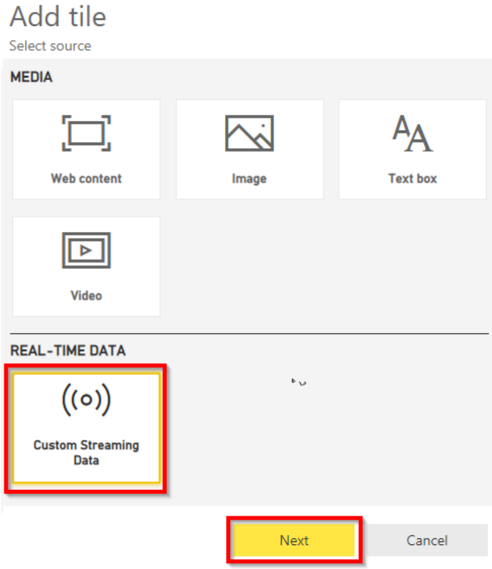


Figure 6.37: Power BI—dashboard—adding a tile

9. Select **Custom Streaming Data** and click on **Next**, as shown in *Figure* 6.38. In the following step, select the **Requests** dataset and click on the **Next** button:

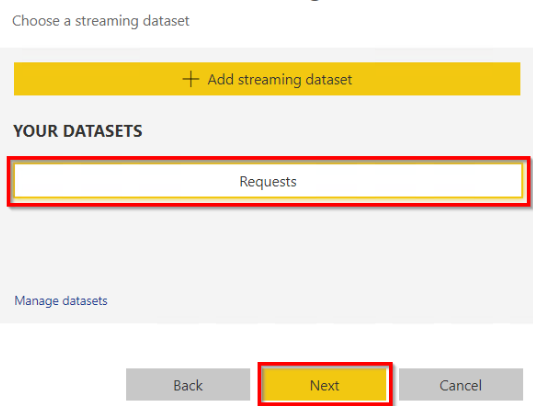


Figure 6.38: Power BI – dashboard—adding a tile—choosing the dataset

10. The next step is to choose **Visualization Type** (**Card** in this case) and select the fields from the data source, as shown in *Figure* 6.39:

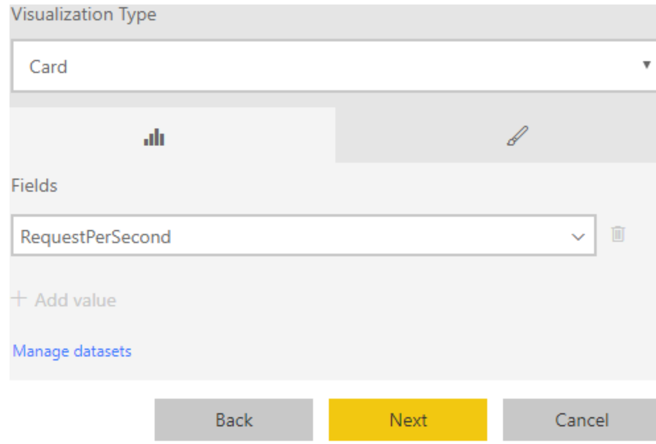


Figure 6.39: Power BI—dashboard—adding a tile—choosing the visualization type

11. The final step is to provide a name for the tile, namely, **`RequestsPerSecond`**. The name might not make sense in this case, but feel free to provide any name as per the project's requirements.

In this section, we have created the dashboard and dataset, and also added a tile to the dashboard.

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

In this section, we'll create an Azure Functions timer trigger to integrate Azure Application Insights with Power BI by performing the following steps:

1. Create a new Azure Functions timer trigger. Replace the default code with the following code. Make sure to configure the correct value for which the analytics query should pull the data. In my case, I have provided five minutes (**`5m`**) in the following code:

```
using System;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using System.Configuration;
using System.Text;
using Newtonsoft.Json.Linq;
using System.Threading.Tasks;
using System.Net.Http;
```

```
namespace FunctionAppinVisualStudio
{
    public static class ViewRealTimeRequestCount
    {
        private const string AppInsightsApi = "https://api.
applicationinsights.io/beta/apps";
        private const string RealTimePushURL = "https://pushurlhere";
        private static readonly string AiAppId = Environment.
GetEnvironmentVariable("AI_APP_ID");
        private static readonly string AiAppKey = Environment.
GetEnvironmentVariable("AI_APP_KEY");

        [FunctionName("ViewRealTimeRequestCount")]
        public static async Task Run([TimerTrigger("0 */5 * * * *")]
TimerInfo myTimer, ILogger log)
        {
            log.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");
            if (myTimer.IsPastDue)
            {
                log.LogWarning($"[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.LogInformation($"Executing real-time Power BI run at:{
DateTime.Now}");

        }
        private static async Task RealTimeFeedRun(string query, ILogger
log)
        {
            log.LogInformation($"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 = $"[{ \"requests\": "{ result}
"\}]";
                    string postData = "[{\"requests\":\"" + result +
"\"}]";
                    log.LogInformation($"[Verbose]: Sending data:
{postData}");

                    using (var response = await httpClient.
PostAsync(RealTimePushURL, new ByteArrayContent(Encoding.UTF8.
GetBytes(postData))))
                    {
                        log.LogInformation($"[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 preconfigured timer frequency.

2. *Figure* 6.40 represents a sequence of pictures showing the real-time data:

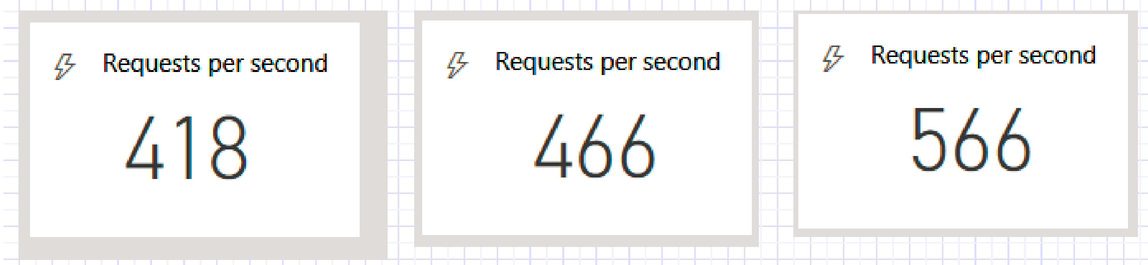


Figure 6.40: Sequence of pictures showing real-time data

How it works...

We have created the following items 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, you can view the real-time data in the Power BI tile of the dashboard.

There's more...

You should also be aware of the following:

- 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.
- In this recipe, we developed a timer trigger that runs every minute. It runs queries on Application Insights to view real-time data. Note that the Application Insights API has a rate limit. Using an Application Insights service for multiple applications may end up consuming all the capacity for that day. Take a look at https://dev.applicationinsights.io/documentation/Authorization/Rate-limits to understand more about API limits.

In this chapter, you have learned how to integrate Application Insights with Azure Functions, how to create custom data points, and how to create custom metrics in Application Insights. Finally, you have also learned how to integrate Application Insights with Power BI with the help of Azure Functions and how to create custom dashboards in Power BI.

Developing reliable serverless applications using durable functions

In this chapter, you'll learn about the following:

- Configuring durable functions in the Azure portal
- Creating a serverless workflow using durable functions
- Testing and troubleshooting durable functions
- Implementing reliable applications using durable functions

Introduction

When developing modern applications that need to be hosted in the cloud, we need to make sure that the applications are stateless. Statelessness is an essential factor in developing cloud-aware applications. For example, we should avoid persisting any data in a resource that is specific to any **virtual machine** (**VM**) instance provisioned to any Azure service (for example, App Service, the API, and so on). Otherwise, we won't be able to leverage some services, such as autoscaling functionality, as the provisioning of instances is dynamic. If we depend on any VM-specific resources, we'll end up facing problems with unexpected behaviors.

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

Although the overall intention of this book is to have each recipe of every chapter solve at least one business problem, the recipes in this chapter don't solve any real-time domain problems. Instead, this chapter as a whole provides some quick-start guidance to help you understand more about Durable Functions and its components, along with the approach to developing durable functions.

> **Note**
>
> For more information about Durable Functions and its related terminology, go through the official documentation, which is available at https://docs.microsoft.com/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

We'll continue the topic of durable functions in the next chapter, where you'll learn how easy it is to use them to develop a workflow-based application.

Durable Functions is a new way in Azure of handling statefulness in serverless architecture, along with other features, such as durability and reliability. Durable Functions is available as an extension to Azure Functions.

Configuring durable functions in the Azure portal

In this recipe, you'll learn about configuring durable functions. In order to develop durable functions, you need to create the following three functions:

- **Orchestrator client**: An Azure function that can manage orchestrator instances. It works as a client that will initiate the orchestrator objects.
- **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 their return values in local variables.
- **Activity functions**: These are the functions that will be called by the orchestrator function. Activity functions are where we develop the logic as per the requirements.
- Let's get started.

Getting ready

Download and install Postman from https://www.getpostman.com/ if you haven't already installed it. We'll be using Postman to test the durable functions.

Create a new function application if you haven't already created one. Ensure that the runtime version is **~3** in the **Application settings** part of the **Configuration** blade, as shown in *Figure 7.1*:

Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose t
controls below. Application Settings are exposed as environment variables for access by your applicatic

+ New application setting   Show values   Advanced edit   Filter

Name	Value
APPINSIGHTS_INSTRUMENTATIONKEY	Hidden value. Click show values button above to view
APPLICATIONINSIGHTS_CONNECTION_STRING	Hidden value. Click show values button above to view
AzureWebJobsStorage	Hidden value. Click show values button above to view
FUNCTIONS_EXTENSION_VERSION	~3
FUNCTIONS_WORKER_RUNTIME	Hidden value. Click show values button above to view

Figure 7.1: Configuration blade—Application settings

How to do it...

In this recipe, you'll learn about creating an orchestrator client by performing the following steps:

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

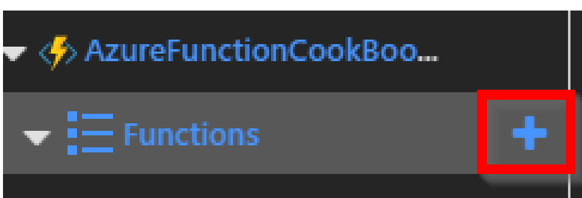


Figure 7.2: Azure Functions—listing

2. Create a new **Durable Functions HTTP starter** function by choosing **Durable Functions** in the **Scenario** drop-down menu, as shown in *Figure* 7.3:

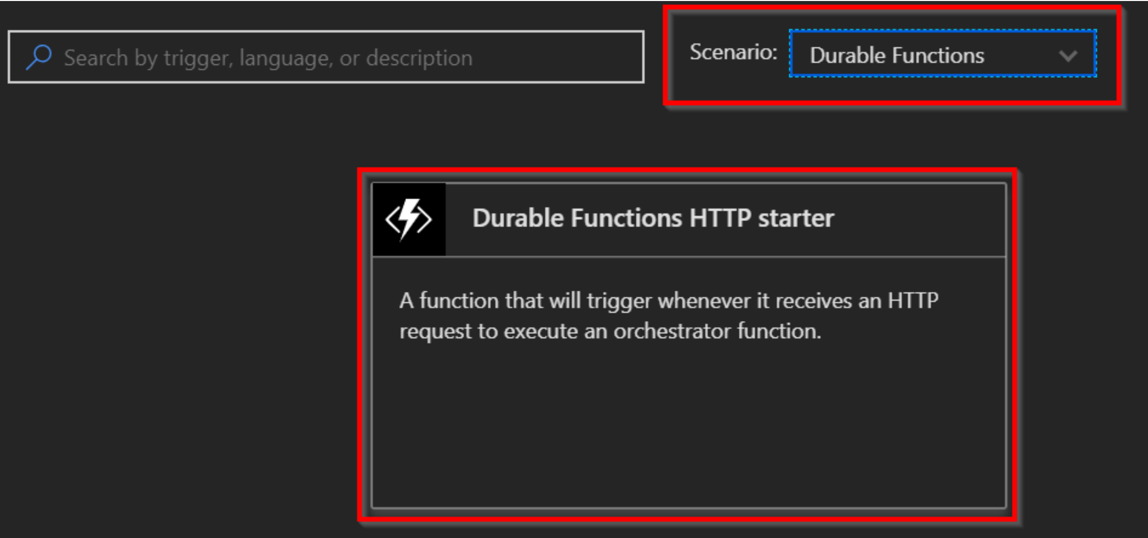


Figure 7.3: Azure Functions—templates

3. Click on **Durable Functions HTTP starter**, which will open a new tab, as shown in *Figure* 7.4. You now need to create a new HTTP function named **`HttpStart`**:

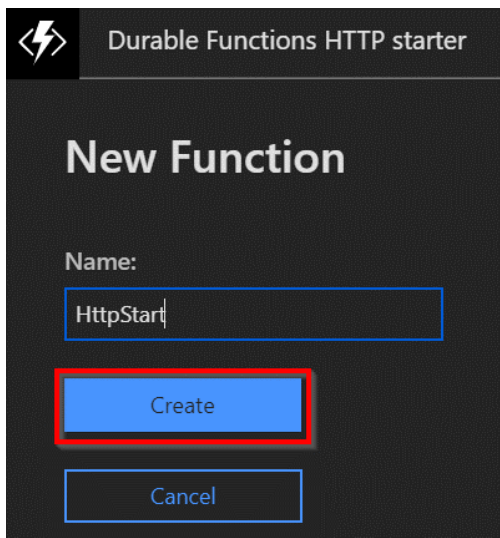


Figure 7.4: Durable Functions HTTP starter—creation

4. Soon after, you'll be taken to the code editor. The following function is an HTTP trigger, which accepts the name of the function 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);
}
```

5. Navigate to the **Integrate** tab and click on **Advanced editor**, as shown in *Figure* 7.5:

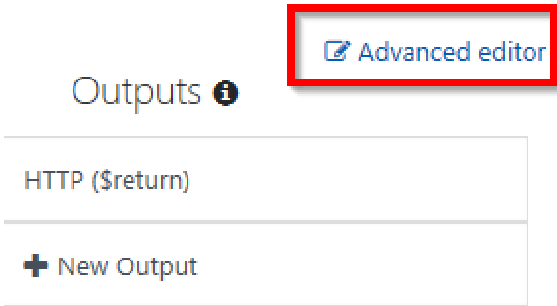


Figure 7.5: Durable Functions HTTP starter—the Integrate tab

6. In **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"
        }
    ]
}
```

> **Note**
>
> The `HttpStart` function works like a gateway for invoking all the functions in the function application. 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.

In this recipe, you have created the orchestrator client. Let's move on to create the orchestrator function itself.

Creating a serverless workflow using durable functions

The orchestrator function manages the workflow via code. The function can asynchronously call other Azure functions (named activity functions), which are the stages in the workflow.

In this recipe, you'll learn about orchestrator functions and activity functions.

Getting ready

Before moving forward, you can read more about orchestrator and activity trigger bindings at https://docs.microsoft.com/azure/azure-functions/durable-functions-bindings.

How to do it...

Here, you'll create the orchestrator function and the activity function.

Creating the orchestrator function

Complete the following steps:

1. Navigate to the Azure function templates and search for the **Durable Functions orchestrator** template, as shown in *Figure* 7.6:

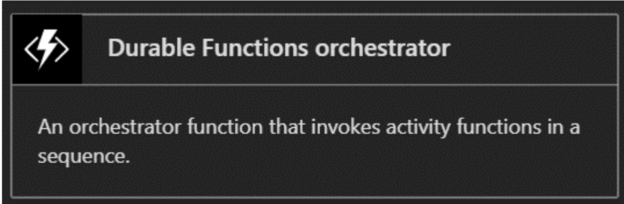


Figure 7.6: Durable Functions orchestrator—template

2. Once you click on the **Durable Functions orchestrator** tile, you'll be taken to the following tab, where you need to provide the name of the function. Once you have provided the name, click on the **Create** button to create the orchestrator function:

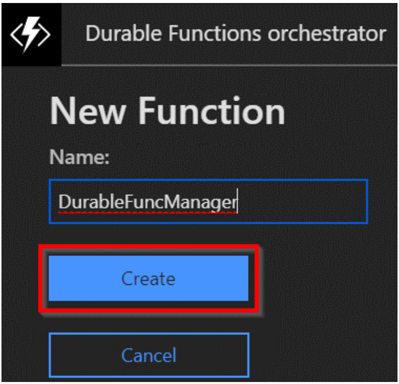


Figure 7.7: Durable Functions orchestrator—creation

3. In **`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 **`DurableOrchestrationContext`** 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"
    }
  ]
}
```

Now that you have created the orchestrator function, let's move on to the next section to create an activity function.

Creating an activity function

Activity functions contain the actual implementation logic. They act as the steps in the workflow that are managed by orchestrator functions. Let's create an activity function by performing the following steps:

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

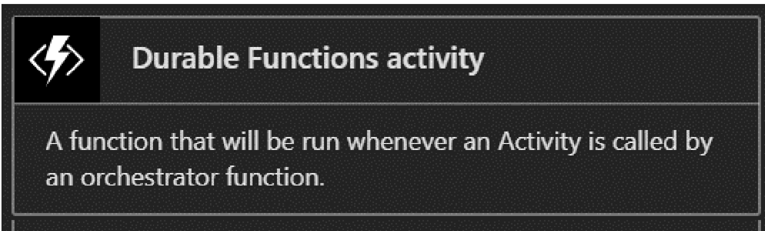


Figure 7.8: Durable Functions activity function—template

2. Replace the default code with the following code, which 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** section of the **Integrate** tab, replace the default code with the following code:

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

In this recipe, you have created an orchestration client, an orchestrator function, and an activity function. You'll learn how to test them in the next recipe.

How it works...

Let's take a look at the workings of the recipe:

- We first developed the orchestrator client (in our case, **`HttpStart`**) in the *Configuring durable functions in the Azure portal* recipe of this chapter, 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 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 a problem with the VMs, then the orchestrator can replace or resume service automatically.
- And lastly, we developed the activity function, which includes 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.

Let's move on to testing.

Testing and troubleshooting durable functions

In *Chapter 5*, *Exploring testing tools for Azure functions*, we discussed various ways of testing Azure functions. We can test durable functions with the same set of tools. However, the approach to testing is entirely different, as regular Azure functions implement one functionality and durable functions help us to achieve durable workflows.

In this recipe, you'll learn how to test and check the status of a durable function.

Getting ready

Download and install the following if you haven't done so already:

- The Postman tool, available from https://www.getpostman.com.
- Azure Storage Explorer, available from http://storageexplorer.com.
- Let's get started.

How to do it...

Perform the following steps:

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

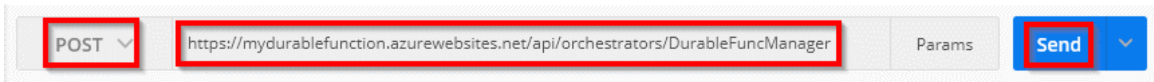


Figure 7.9: Making a POST request to the durable orchestrator using Postman

3. After clicking the **Send** button, you'll 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:

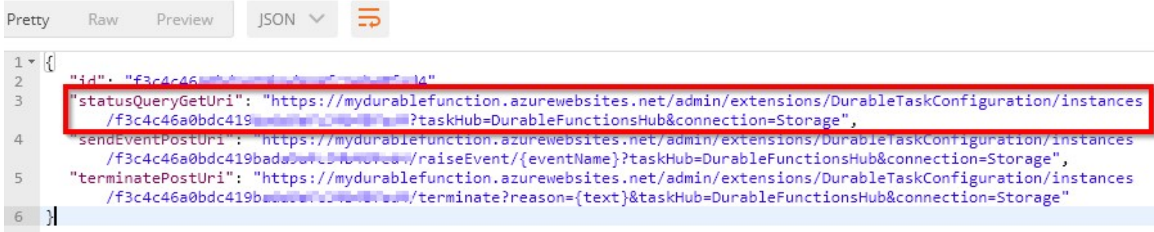


Figure 7.10: Viewing the response of the orchestrator function in Postman

4. Click on **`statusQueryGetUri`** in the preceding step to view the status of the function. Clicking on the link in *step* 3 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:

Body Cookies Headers (8) Test Results

Pretty Raw Preview JSON

```
{
    "instanceId": "53a8eca4e20b43e59718516397e94f0e",
    "runtimeStatus": "Completed",
    "input": null,
    "customStatus": null,
    "output": [
        "Hello Welcome Cookbook Readers!"
    ],
    "createdTime": "2018-10-21T16:46:42Z",
    "lastUpdatedTime": "2018-10-21T16:46:49Z"
}
```

Figure 7.11: Checking the status of the durable function in Postman

5. If everything goes well, you will see **`runtimeStatus`** as **`Completed`** in Postman, as shown in *Figure 7.11*. You'll also get eight records in Table storage, where the execution history is stored, as shown in *Figure 7.12*:

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

Figure 7.12: Checking the status of the durable function in Table storage

6. If something goes 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.

In this recipe, you have learned how to test a durable function. Let's move to the next recipe to learn how to develop reliable applications.

Implementing reliable applications using durable functions

One of the most commonly used ways to swiftly process data is to go with parallel processing. The main advantage of this approach is that we get the desired output pretty quickly, depending on the previously created sub-threads. This can be achieved in multiple ways using different technologies. However, a common challenge in these approaches is that if something goes wrong in the middle of a sub-thread, it's not easy to self-heal and resume from where things stopped.

In this recipe, we'll implement a simple way of executing a function in parallel with multiple instances using durable functions for the following scenario.

Assume that we have five customers (with IDs 1, 2, 3, 4, and 5, respectively) who need to generate 50,000 barcodes. It would take a lot of time to generate the barcodes owing to the involvement of image processing tasks. One simple way to quickly process this 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'll also simulate a simple use case to understand how durable functions auto-heal when the VM on which they are hosted goes down or is restarted.

Getting ready

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

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

How to do it...

In this recipe, we'll 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 real-world applications, we would have business logic for deciding the customers' eligibility, and, based on that logic, we 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

Create the orchestrator function by performing 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:

```
#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;
}
```

The preceding code invokes 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 barcodes that are generated. Finally, it waits until the activity functions for all customers are completed and then returns the sum of all the barcodes that are generated for all the customers.

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

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

In this section, we have created the orchestrator function, which calls and manages the activity functions. Let's move on to the next section.

Creating a GetAllCustomers activity function

In this section, we'll create an activity function called **GetAllCustomers** that returns all the customer IDs that should be processed. For simplicity, the customer IDs are hardcoded, but the customer IDs must be retrieved from a database in real time.

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** section of the **Integrate** tab, replace the default code with the following code:

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

We have developed the **GetAllCustomers** activity function, which retrieves all the customers for which the barcode images need to be generated. Let's move on to the next section.

Creating a CreateBARCodeImagesPerCustomer activity function

In this section, we will create another activity function called **CreateBARCodeImagesPerCustomer**, which will create the barcodes for a given customer. This activity function will be called multiple times depending on the number of customers. 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** section 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'll be stopping the function application to simulate a restart of the VM where the function will be running, and to see how the durable function resumes from where it was paused.

4. Make a **POST** request using Postman, as shown in *Figure* 7.13:

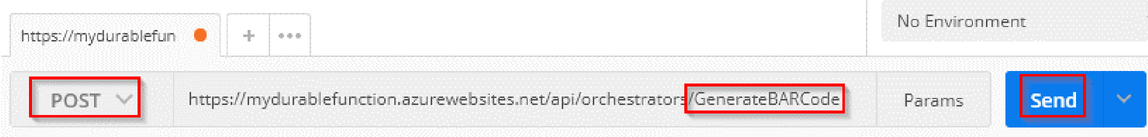


Figure 7.13: POST request to the durable function using Postman

5. Once you click on the **Send** button, you'll 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, navigate to the function application's **Overview** blade and stop the service by clicking on the **Stop** button:

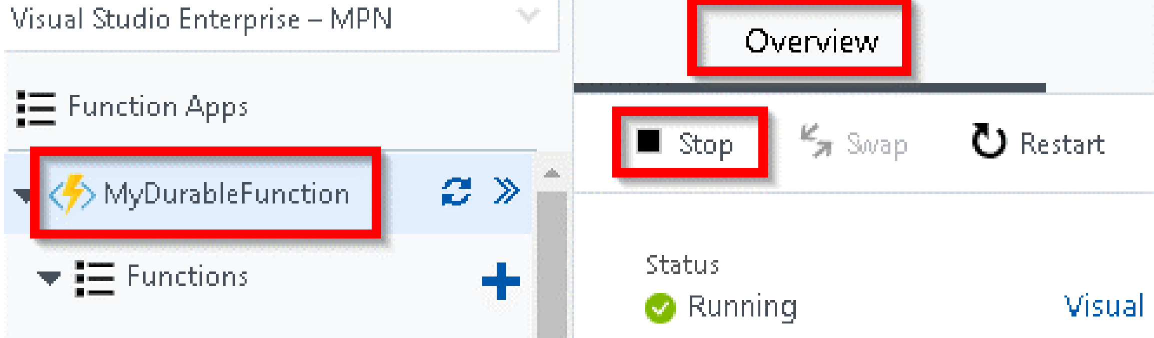


Figure 7.14: Azure function application—the Overview blade

7. The execution of the function will be stopped in the middle. Navigate to your 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		

Figure 7.15: Checking the status of the durable function in Table storage

8. After some time—in my case, after just 5 minutes—go back to the **Overview** blade and start the function application service. Notice that the durable function will resume from where it stopped. You didn't write any code for this; it's an out-of-the-box feature. The completed function is shown in *Figure 7.16*:

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

Figure 7.16: Checking the status of the durable function in Table storage

How it works...

Durable functions allow us to develop the reliable execution of our functions, which means that even if VMs crash or restart while a function is running, it automatically resumes its previous state. It does so with the help of something called checkpointing and replaying, where the history of the execution is stored in Table storage.

> **Note**
>
> You can learn more about the checkpointing and replaying feature at https://docs.microsoft.com/azure/azure-functions/durable-functions-checkpointing-and-replay.

There's more...

- If 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 resides in the storage account. The connection string of that storage account can be found in **Application settings**, and it was created while creating the function application:

Figure 7.17: Application settings—WEBSITE_CONTENTSHARE

You can find the storage account name in **Application settings**, as shown in *Figure* 7.17.

In this recipe, we have learned how to develop reliable applications using durable functions.

In this chapter, you have learned how to develop a reliable, workflow-based application using durable functions. You have created an orchestrator function that internally calls multiple activity functions that are responsible for implementing logic. The orchestrator function takes care of managing the activity functions.

8

Bulk import of data using Azure Durable Functions and Cosmos DB

In this chapter, we'll complete the following recipes:

- Uploading employee data to blob storage
- Creating a blob trigger
- Creating a durable orchestrator and triggering it for each CSV import
- Reading CSV data using activity functions
- Autoscaling Cosmos DB throughput
- Bulk inserting data into Cosmos DB

Introduction

In this chapter, we'll develop a mini-project by taking a very common use case that solves the business problem of sharing data across different applications using CSV. We'll use Durable Functions, which is an extension to Azure Functions that lets you write workflows by writing a minimal amount of code.

Here are the two core features of Durable Functions that we'll be using in the recipes of this chapter:

- **Orchestrator**: An orchestrator is a function that is responsible for managing all activity triggers. It can be treated as a workflow manager that has multiple steps. The 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.

> **Note**
>
> You can learn more about Durable Functions at https://docs.microsoft.com/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

Business problem

In general, every organization uses applications that are hosted on multiple platforms across different datacenters (either on the cloud or on-premises). Often, there will be a need to feed data from one application to another system. Usually, CSV spreadsheets (or, in some cases, JSON or XML files) are used to export data from one application and import it into another application.

You may think that exporting CSV files from one application to another would be a straightforward job, but if there are many applications that need to feed data to other applications, and on a weekly/monthly basis, then this process would become very tedious and there is a lot of scope for manual error. So, the solution is obviously to automate the process as far as possible.

In this chapter, we'll learn how to develop a durable solution based on serverless architecture using Durable Functions. *Chapter 7*, *Developing reliable serverless applications using durable functions*, already covers the basics of what durable functions are and how they work. In the aforementioned chapter, we implemented the solution from the portal. However, in this chapter, we'll implement a mini-project using Visual Studio 2019.

Before we start developing the project, let's try to understand the new serverless way of implementing the solution.

The durable serverless way of implementing CSV imports

The following diagram shows all the steps required to build the solution using serverless architecture:

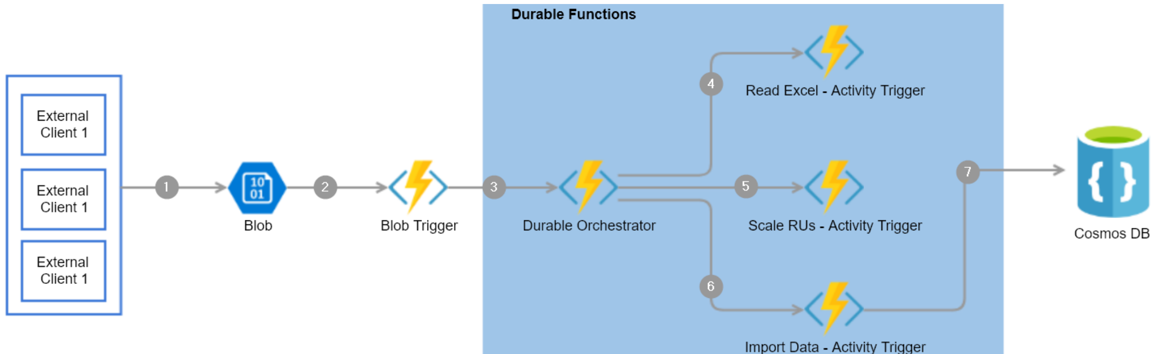


Figure 8.1: Durable Functions—architecture process flow

Here are the detailed steps pertaining to the preceding architecture diagram that will be implemented in this chapter:

1. External clients or applications upload a CSV file to blob storage.
2. A blob trigger gets triggered once the CSV file is uploaded successfully.
3. The durable orchestrator is started from the blob trigger.
4. The orchestrator invokes **Read CSV - Activity Trigger** to read the CSV 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 the Cosmos DB collection using Cosmos DB output bindings.

Let's now start building the client application that uploads the CSV file.

Uploading employee data to blob storage

In this recipe, we'll develop a console application for uploading the CSV sheet to blob storage.

Getting ready

Perform the following steps:

1. Install Visual Studio 2019.
2. Create a storage account and create a blob container with the name **`csvimports`**.
3. Create a CSV file with some employee data, as shown in *Figure* 8.2:

Emp Id	Name	Email	PhoneNumber
1	Nischala	Nischala@gmail.com	1111111111
2	Vivek	vivek@gmail.com	2222222222
3	Khadir	Khadir@gmail.com	3333333333
4	Bhargavi	Bhargavi@gmail.com	4444444444
5	Praveen Sreeram	praveen@gmail.com	5555555555
6	Meena	meena@gmail.com	6666666666

Figure 8.2: CSV file with employee data

How to do it…

In this section, we are going to create a .NET Core–based client application that uploads the **`csv`** file to the blob container by performing the following steps:

1. Create a new **Console App (.NET Core)** project named **`CSVImport.Client`** using Visual Studio, as shown in *Figure* 8.3:

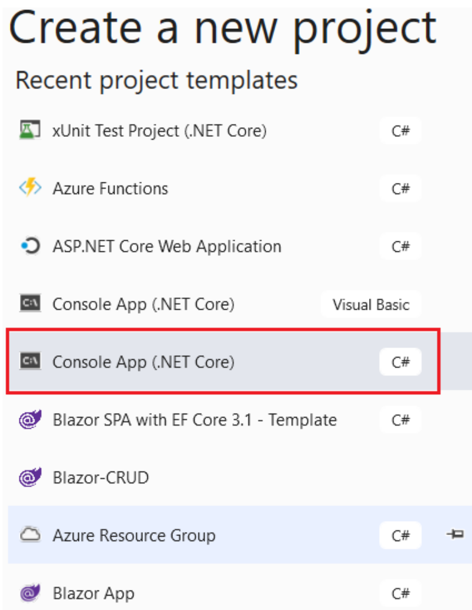


Figure 8.3: Creating a new Console App (.NET Core) project using Visual Studio

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.Azure.Storage;
using Microsoft.Azure.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 CSV file into the blob container that we have created. For the sake of simplicity, the following code uploads the CSV 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 into the **`Program.cs`** file of the **`CSVImport.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("StorageConnection"));
CloudBlobClient cloudBlobClient = cloudStorageAccount.
CreateCloudBlobClient();
CloudBlobContainer CSVBlobContainer = cloudBlobClient.
GetContainerReference("csvimports");
await CSVBlobContainer.CreateIfNotExistsAsync();
CloudBlockBlob cloudBlockBlob = CSVBlobContainer.
GetBlockBlobReference("employeeinformation" + Guid.NewGuid().ToString());
await cloudBlockBlob.UploadFromFileAsync(@"C:\ employeeinformation.csv");
}
```

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:

```
static void Main(string[] args)
{
        try
        {
        UploadBlob().Wait();
        }
        catch (Exception ex)
        {
        Console.WriteLine("An Error has occurred with the
        message" + ex.Message);
        }
    Console.WriteLine("Successfully uploaded.");
}
```

6. The next step is to create a configuration file named **`appsettings.json`** that contains the storage account's connection string, as shown in *Figure 8.4*:

```
appsettings.json
Schema: http://json.schemastore.org/appsettings
1 {
2   "ConnectionStrings": {
3     "StorageConnection":
      "DefaultEndpointsProtocol=https;AccountName=azurefunctions[illegible];AccountKey=TxhVTtYr4afPG7VbRJLwiU35JXQsU2B[illegible]
      d3zXdZE3[illegible];EndpointSuffix=core.windows.net"
4   }
5 }
6
```

Figure 8.4: Azure Functions—local configuration file

7. Go to the properties of the **`appsettings.json`** file and change **Copy to Output Directory** to **Copy if newer**, so that the properties can be read by the program as shown in *Figure 8.5*:

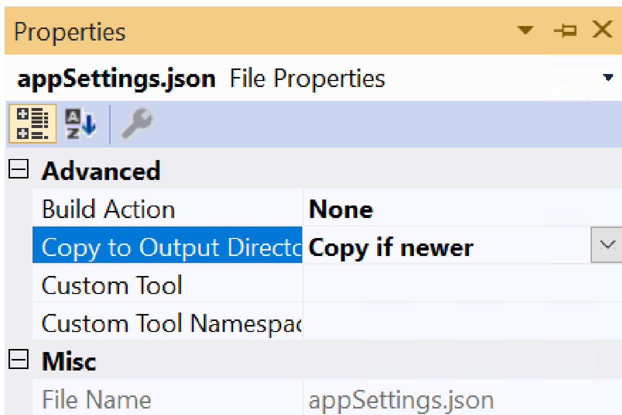


Figure 8.5: Azure Functions—appSettings.json properties—Copy if newer

8. Now, build the application and execute it. If you have configured everything correctly, then you should see something as shown in *Figure* 8.6:

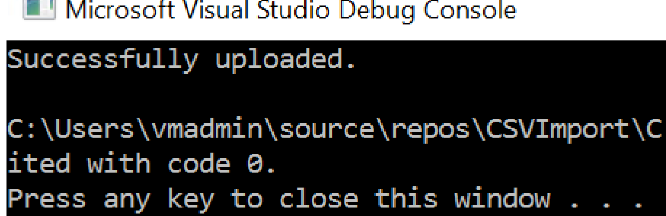


Figure 8.6: Azure Functions—console output

9. Let's now navigate to the storage account and go to the blob container named **`csvimports`**, where the uploaded CSV file should be visible, as shown in *Figure* 8.7:

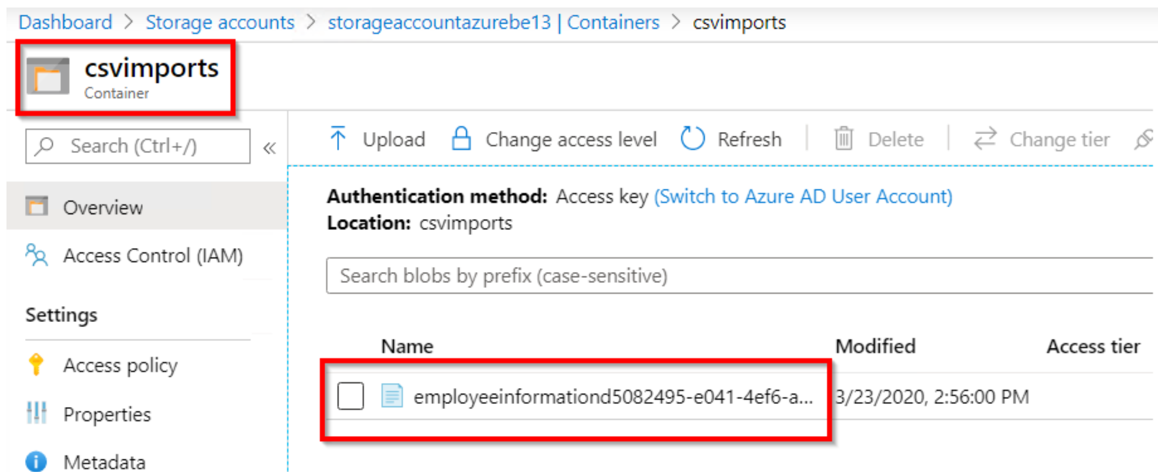


Figure 8.7: Storage container—uploaded blob

That's it. We have now developed an application that is responsible for uploading the blob.

There's more...

Make a note of the naming conventions that should be followed while creating the blob container. At the time of writing, the portal throws this error message if we do not adhere to 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.*

In this recipe, we have created a console application that uses storage assemblies to upload a blob (in our case, it is just a CSV file) to the designated blob container. Note that every time the application runs, a new file will be created in the blob container. In order to upload the CSV files with unique names, we are appending a GUID. Let's move on to the next recipe.

Creating a blob trigger

In this recipe, we'll create a function app with the Azure Functions V3 runtime and learn how to create a blob trigger using Visual Studio, and we'll also see how the blob trigger gets triggered when the CSV file is uploaded successfully to the blob container.

How to do it...

Perform the following steps:

1. Add a new project named `CSVImport.DurableFunctions` to the existing solution by choosing the **Azure Functions** template, as shown in *Figure* 8.8:

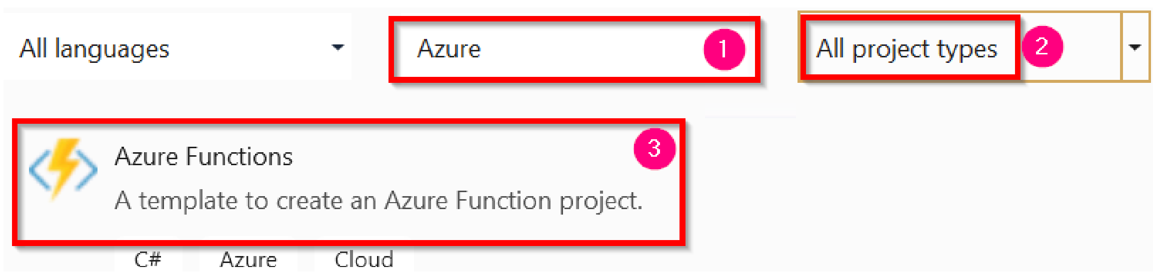


Figure 8.8: Visual Studio—creating a new Azure Functions project

2. The next step is to choose the Azure Functions runtime as well as the trigger. Choose **Azure Functions v3 (.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 CSV files are being uploaded:

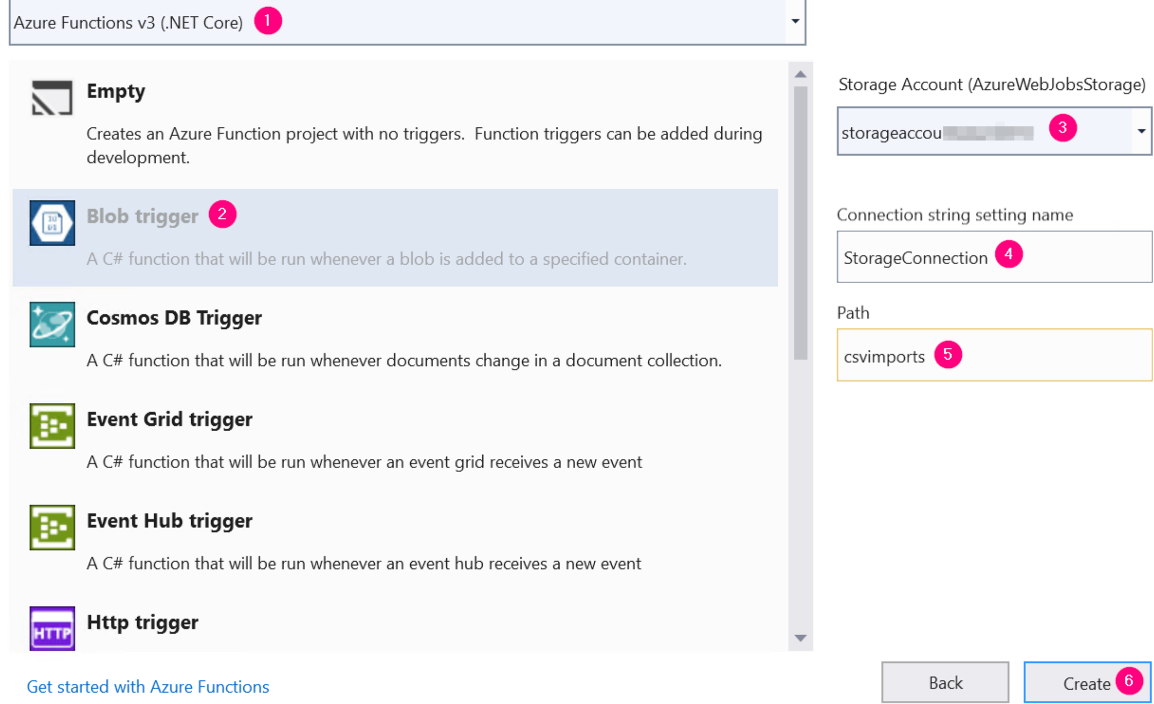


Figure 8.9: Visual Studio—creating a new function app

3. After creating the project, the structure should look something like *Figure 8.10*:

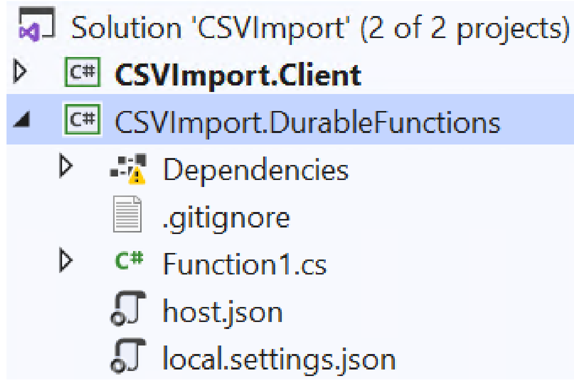


Figure 8.10: Visual Studio—function app—Solution Explorer

4. Let's add a connection string to the **DurableFunctions** project 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 *Figure 8.11*:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "DefaultEndpointsProtocol=https;AccountName=storage;AccountKey=veVBIwe023/1SqOexAg36m/+iBsXVV3TAOwNnRoJ+xZBQ4ZJJ/tgTGsRYCw==;BlobEndpoint=https://storageaccountazurebe13.blob.core.windows.net/;TableEndpoint=https://storageaccountazurebe13.table.core.windows.net/;QueueEndpoint=https://storageaccountazurebe13.queue.core.windows.net/;FileEndpoint=https://storageaccountazurebe13.file.core.windows.net/",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet",
    "StorageConnection": "DefaultEndpointsProtocol=https;AccountName=<<storagenamehere>>;AccountKey=<<Key Here>>;EndpointSuffix=core.windows.net"
  }
}
```

Figure 8.11: Azure Functions—configuration file

5. Now, open the **Function1.cs** file and rename it to **CSVImportBlobTrigger**, and also replace **Function1** (the name of the function) with **CSVImportBlobTrigger** (*line 11*), as shown in *Figure 8.12*:

```
namespace CSVImport.DurableFunctions
{
    public static class CSVImportBlobTrigger
    {
        [FunctionName("CSVImportBlobTrigger")]
        public static void Run([BlobTrigger("csvimports/{name}", Connection = "StorageConnection")]Stream
          myBlob, string name, ILogger log)
        {
            log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size:
              {myBlob.Length} Bytes");
        }
    }
}
```

Figure 8.12: Azure Functions—blob trigger code

6. Create a breakpoint in **CSVImportBlobTrigger** and run the application by pressing the F5 key. If everything is configured properly, the following should be visible on the console:

```
[3/23/2020 3:07:31 PM] Found the following functions:
[3/23/2020 3:07:31 PM] CSVImport.DurableFunctions.CSVImportBlobTrigger.Run
[3/23/2020 3:07:31 PM]
```

Figure 8.13: Azure Functions—console

7. Let's upload a new file by running the **CSVImport.Client** application. Immediately after the file is uploaded, the blob trigger will be fired. Your breakpoints should also be hit along with this.

We are done creating the blob trigger that gets fired whenever a new blob is added to the blob container.

We'll process the blob in the upcoming recipes of this chapter.

There's more...

All the configurations will be taken from the `local.settings.json` file while running the functions in our local environment. However, when deploying the functions to Azure, all the configuration items (such as the connection string and app settings) will be referenced from the application settings of your function app. Make sure to create all the configuration items in the function app after deploying the functions.

In this recipe, we have created a new function app based on the Azure Functions V3 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 `CSVImport.Client` application to upload a file to validate the fact that the blob trigger is getting executed.

Let' move on to the next recipe to learn how to create a durable orchestrator.

Creating the durable orchestrator and triggering it for each CSV import

This is one of the most important and interesting recipes. We'll 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 `CSVImport` project.

How to do it...

In this section, we are going to create an orchestrator and also learn how to invoke it by performing the following steps:

1. Create a new function by right-clicking on `CSVImport.DurableFunctions`, click on **Add**, and then choose **New Azure Function**, as shown in *Figure 8.14*:

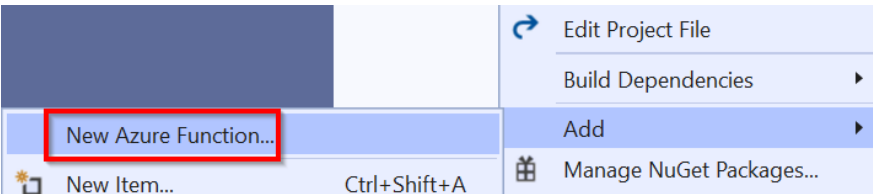


Figure 8.14: Visual Studio—adding a new function

2. In the **Add New Item** popup, choose **Azure Function**, provide the name **`CSVImport_Orchestrator`**, and click on **Add**, as shown in *Figure 8.15*:

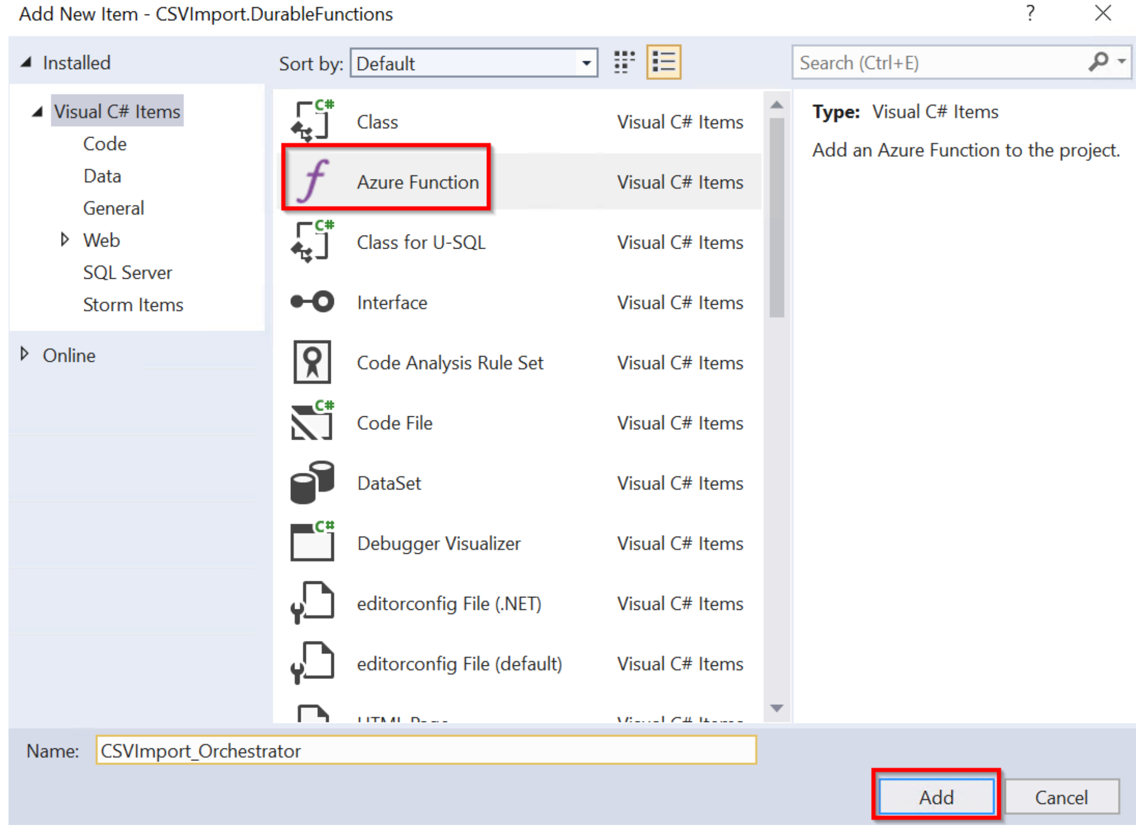


Figure 8.15: Visual Studio—adding a new function

3. In the **New Azure Function** popup, select the **Durable Functions 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'll not be using this HTTP trigger; we'll be using the logic inside it in our **`CSVImportBlobTrigger`** 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: Visual Studio creates this simple activity function. Let's go ahead and remove this default function. Once the default activity function is created, we'll create our activity function:

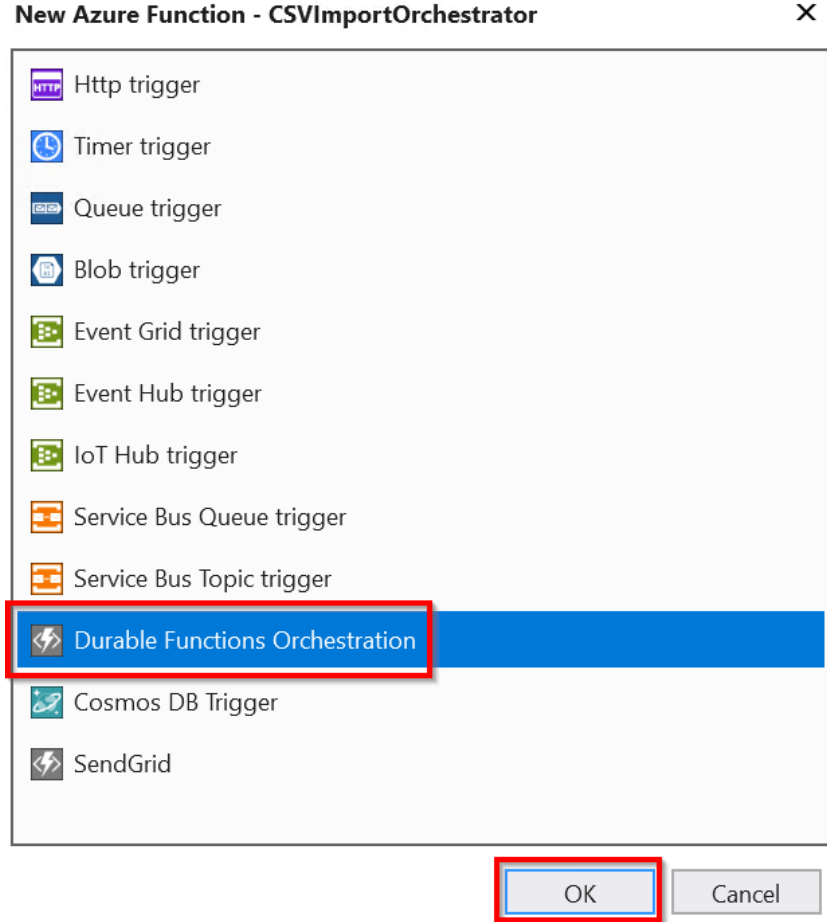


Figure 8.16: Visual Studio—adding a new Durable Functions orchestration trigger

4. In the **`CSVImportBlobTrigger`** 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 by using the attribute **`[DurableClient]`**.

   Call **`StartNewAsync`** using the **`IDurableOrchestrationClient`** reference.

5. The code in the **CSVImportBlobTrigger** function should appear as follows after making these changes:

```
using System.IO;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host
using Microsoft.Extensions.Logging;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;
namespace CSVImport.DurableFunctions
{
public static class CSVImportBlobTrigger
{
[FunctionName("CSVImportBlobTrigger")]
public static async void Run(
[BlobTrigger("csvimports/{name}", Connection = "StorageConnection")]
Stream myBlob, string name,
[DurableClient]IDurableOrchestrationClient starter, ILogger log)
{
string instanceId = await
starter.StartNewAsync("CSVImport_Orchestrator", name);
log.LogInformation($"C# blob trigger function Processed blob\n Name:{name}
\n Size: {myBlob.Length} Bytes");
}
}
}
```

6. Create a breakpoint in the **CSVImport_Orchestrator** function and run the application by pressing the F5 key on the keyboard.

7. Let's now upload a new file (while **CSVImport.DurableFunctions** is running) by running the **CSVImport.Client** function. (You can also upload the CSV file to the blob container directly from the Azure portal.) Once the file is uploaded, in just a few moments, the breakpoint in the **CSVImport_Orchestrator** function should be hit, as shown in *Figure 8.17*:

```
public static class CSVImport_Orchestrator
{
    [FunctionName("CSVImport_Orchestrator")]
    0 references
    public static async Task<List<string>> RunOrchestrator(
        [OrchestrationTrigger] IDurableOrchestrationContext context)
    {
        var outputs = new List<string>();

        // Replace "hello" with the name of your Durable Activity Function.
        outputs.Add(await context.CallActivityAsync<string>("CSVImport_Orchestrator_Hello", "Tokyo"));
        outputs.Add(await context.CallActivityAsync<string>("CSVImport_Orchestrator_Hello", "Seattle"));
        outputs.Add(await context.CallActivityAsync<string>("CSVImport_Orchestrator_Hello", "London"));

        // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
        return outputs;
    }
```

Figure 8.17: Durable Functions orchestration trigger breakpoint

We have learned how to invoke the durable orchestrator function from the blob trigger.

How it works...

We started the recipe by creating the orchestration function using Visual Studio, and then we made changes to the **`CSVImportBlobTrigger`** blob trigger by adding the **`OrchestratonClient`** output bindings to invoke the durable orchestrator function.

There's more...

In this recipe, we have used **`DurableClient`**, 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 provide an update regarding any external event using the **`RaiseEventAsync`** method.

> **Note**
>
> Learn more at https://docs.microsoft.com/azure/azure-functions/durable/durable-functions-instance-management?tabs=csharp#sending-events-to-instances.

In this recipe, we have learned how to create an orchestrator and how to invoke it. Let's now move on to the next recipe.

Reading CSV data using activity functions

In this recipe, we'll retrieve all the data from specific CSV 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 a CSV sheet located in the blob container. In this recipe, we'll create an activity trigger named **ReadCSV_AT** that reads the data from the blob stored in the storage account. This activity trigger performs the following jobs:

1. It connects to the blob using a function, **ReadBlob**, of a class named **StorageManager**.
2. It returns the data from the CSV file as a collection of employee objects.

Getting ready

Install the following NuGet package in the **CSVImport.DurableFunctions** project:

```
Install-Package Microsoft.Azure.Storage.blob
```

How to do it...

If you think of Durable Functions as a workflow, then the activity trigger function can be treated as a workflow step that takes some kind of optional input, performs some functionality, and returns an optional output. It is one of the core concepts of Azure Durable Functions.

Before we start creating the activity trigger function, let's first build the dependency functions.

Reading data from blob storage

Learn how to read data from blob storage by performing 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 static Task<string> 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["Values:StorageConnection"]);
            CloudBlobClient cloudBlobClient = cloudStorageAccount.
CreateCloudBlobClient(); CloudBlobContainer CSVBlobContainer =
cloudBlobClient.GetContainerReference("csvimports");
            CloudBlockBlob cloudBlockBlob = CSVBlobContainer.
GetBlockBlobReference(BlobName);
            string employeeContent;
            using (var memoryStream = new MemoryStream())
            {
                await cloudBlockBlob.DownloadToStreamAsync(memoryStream);
                employeeContent = System.Text.Encoding.UTF8.
GetString(memoryStream.ToArray());
            }
            return employeeContent;
        }}
```

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, ensure that the connection string of the storage account is added to the **local.settings.json** file, as shown in *Figure 8.18*:

Figure 8.18: Azure Functions—local configuration file

Build the application and ensure that there are no errors. In this section, we have learned how to read blob data. Let's now move on to the next section to parse the CSV data.

Reading CSV data from the stream

In this section, we'll learn how to read the CSV data by performing the following steps:

1. Create a class named **CSVManager** and paste the following code. This class has a method named **ReadEmployeeData**, which reads data from the CSV file content. It reads each row, creates an **Employee** object for each row, and then returns an employee collection. We'll create the **Employee** class in the next step:

```
class CSVManager
    {

        public static List<Employee> ReadEmployeeData(string
employeesListContent)
        {
            List<Employee> employees = new List<Employee>();
            var employeesList = employeesListContent.Split(Environment.
NewLine);

            for (int employeeIndex = 1; employeeIndex < employeesList.
Length; employeeIndex++)
            {
                var employee = employeesList[employeeIndex];
                if (employee != null & employee.Length > 1)
                {
                    var employeeColumns = employee.Split(",");
                    employees.Add(
                    new Employee()
                    {
                        EmpId = employeeColumns[0],
                        Name = employeeColumns[1],
                        Email = employeeColumns[2],
                        PhoneNumber = employeeColumns[3],
                    });
                }
            }
            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; }
}
```

3. Add the following namespaces:

```
using System;
using System.Collections.Generic;
```

Building the application now, you should not experience any errors. We are done with developing the dependencies for our first activity trigger function. Let's now start building the actual activity trigger.

Creating the activity function

In this section, you are going to learn how to develop an activity function by performing the following steps:

1. Create a new activity function named **ReadCSV_AT** that connects to the blob using the **StorageManager** class that we developed in the previous section, and then reads the data using the **CSVManager** class. Copy the following code to the **CSVImport_Orchestrator** class:

```
[FunctionName("ReadCSV_AT")]
public static async Task<List<Employee>> ReadCSV_AT([ActivityTrigger]
string name,
ILogger log)
        {
            log.LogInformation("ReadCSV_AT Started");
            log.LogInformation("Reading the blob Started");
            var EmployeeContent = await StorageManager.ReadBlob(name);
            log.LogInformation("Reading the blob has Completed");
            log.LogInformation("Reading the CSV Data Started");
            List<Employee> employees = CSVManager.
ReadEmployeeData(EmployeeContent);
            log.LogInformation("Reading the blob has Completed");
            log.LogInformation("ReadCSV_AT End");
            return employees;
        }
```

2. Let's now invoke the **ReadCSV_AT** activity function from the orchestrator. Go to the **CSVImport_Orchestrator** orchestration function and replace it with the following code. The orchestration function invokes the activity function by passing the name of the CSV that is uploaded so that the activity function reads the data from the CSV file:

```
[FunctionName("CSVImport_Orchestrator")]
        public static async Task<List<string>>
RunOrchestrator([OrchestrationTrigger] IDurableOrchestrationContext
context)
        {
            var outputs = new List<string>();
            string CSVFileName = context.GetInput<string>();
            {
                List<Employee> employees = await context.
CallActivityAsync<List<Employee>>("ReadCSV_AT", CSVFileName);
            }
            return outputs;
        }
```

3. Let's run the application and then upload a CSV file. If everything is configured properly, we should see something similar to the following in the **ReadCSV_AT** activity trigger function, where we can see the number of employee records being read from the CSV file as shown in *Figure 8.19*:

```
[FunctionName("ReadCSV_AT")]
0 references
public static async Task<List<Employee>> ReadCSV_AT([ActivityTrigger] string name,
er log)
{
    log.LogInformation("ReadExcel_AT Started");
    log.LogInformation("Reading the Blob Started");
    var EmployeeContent = await StorageManager.ReadBlob(name);
    log.LogInformation("Reading the Blob has Completed");
    log.LogInformation("Reading the Excel Data Started");
    List<Employee> employees = CSVManager.ReadEmployeeData(EmployeeContent);
    log.LogInformation("Reading the Blob has Completed");

    log.LogInformation("ReadExcel_AT End"); ≤1ms elapsed
    return employees;
}
```


Figure 8.19: Visual Studio—employee records being read from the CSV file

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 **StartNewAsync** method of the **DurableOrchestrationClient** class.

In this recipe, we have developed an activity function that reads data from a CSV file. Let's move on to the next recipe to learn how to automatically increase the throughput of Cosmos DB.

Autoscaling Cosmos DB throughput

In the previous recipe, we read data from a CSV file 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 a 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. Therefore, you may encounter performance issues if the capacity of the Cosmos DB collection is insufficient.

> **Note**
>
> Cosmos DB collection throughput is measured by the number of **Request Units** (**RUs**) allocated to the collection. Read more about it at https://docs.microsoft.com/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'll increase the RUs before we start importing the data. Once the importing process is complete, we can decrease the RUs to the minimum level.

Cosmos DB allows us to set throughput using two methods:

1. **Manual**: Using this method, we can set the throughput to the required number of RUs, either manually from the portal or programmatically. The number of RUs set will be fixed until it is changed by us.
2. **Autopilot**: This is a new feature and is currently in preview. It is not recommended to be used in production applications. In this method, we can set a predefined maximum number of RUs (for example, 20,000 RUs). In this method, Cosmos DB will decide how many RUs are to be used based on the load up to the maximum limit that's set (in our example, it's 20,000).

In this recipe, we'll learn how to increase the throughput (capacity) of Cosmos DB containers so that they can take the necessary load and update data without any performance issues.

Getting ready

Perform the following steps:

1. Create a Cosmos DB account (with the Core SQL API) by following the instructions mentioned in the article at https://docs.microsoft.com/azure/cosmos-db/create-sql-api-dotnet.
2. Create a Cosmos database and a collection and set the RUs to **400** per second, as shown in *Figure 8.20*:

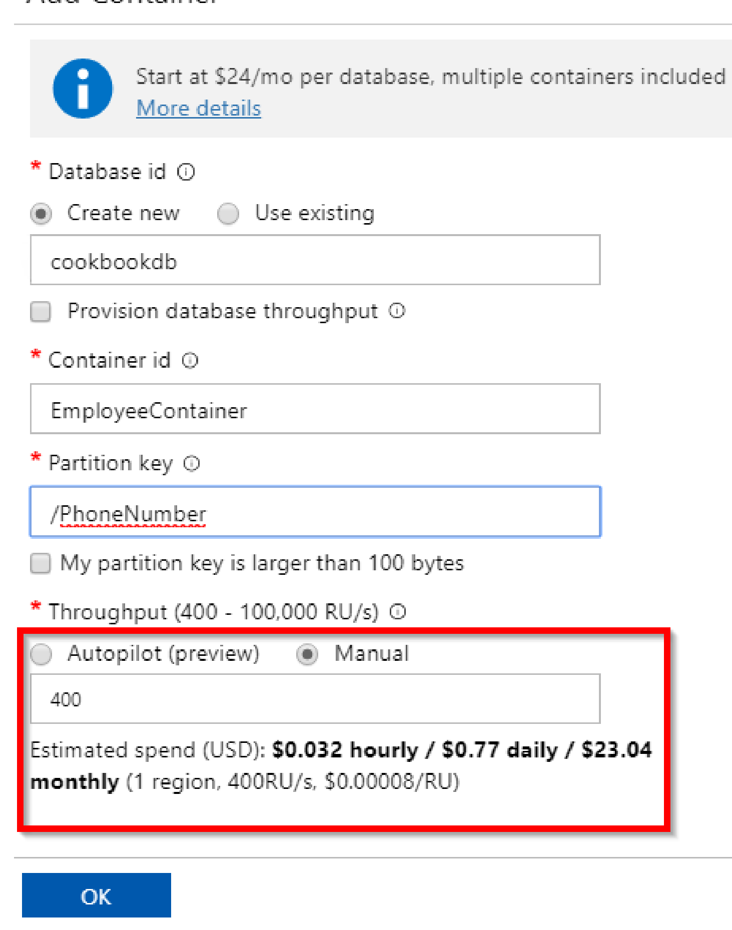


Figure 8.20: Cosmos DB—adding a container with 400 RUs

3. In the **CSVImport.DurableFunctions** project, 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 **`CSVImport_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 this function, we have replaced the original 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. CreateDocumentCollectionUri("cookbookdb",
"EmployeeContainer"));
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 **`CSVImport_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 *Figure* 8.21. Copy the connection from the **Keys** blade of the Cosmos DB account:

Figure 8.21: Azure Functions—local configuration file

4. Now, in the **CSVImport_Orchestrator** function, add the following line to invoke **ScaleRU_AT**. In this example, I'm passing **500** as the RU value. You can choose your value according to your project's requirements:

```
await context.CallActivityAsync<string>("ScaleRU_AT", 500);
```

5. Now, upload a CSV 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 now navigate to Cosmos DB's **Data Explorer** tab and navigate to the **Scale & Settings** section, where we can view **500** as the new throughput of the collection, as shown in *Figure 8.22*:

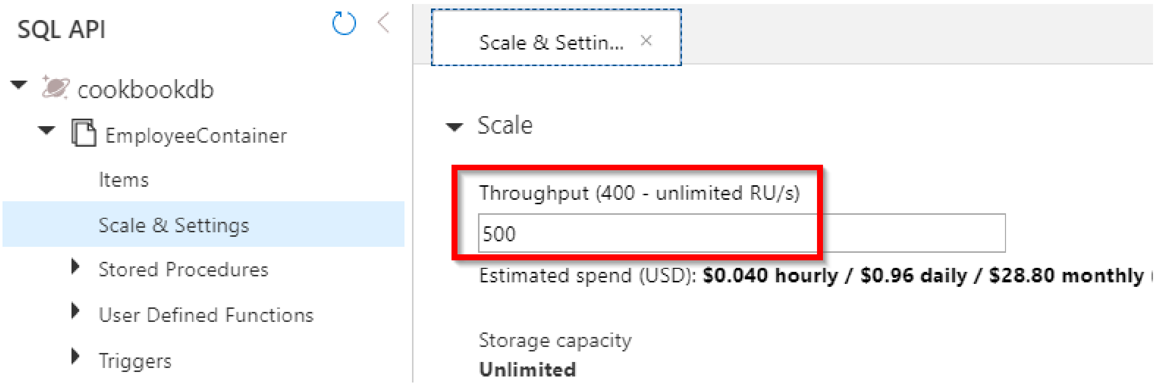


Figure 8.22: Cosmos DB—viewing the throughput in the Scale & Settings blade

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. Learn more about this at https://docs.microsoft.com/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, you will learn about one of the simplest ways of inserting data into Cosmos DB.

How to do it...

Perform the following steps:

1. Create a new activity trigger named **ImportData_AT**, which takes an employee collection as input and saves the data in the Cosmos DB container. Paste the following code into the new activity trigger that does the job:

```
[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", "EmployeeContainer"), 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 now run the application and upload the CSV file to test the functionality. If everything went well, we should see all the records created in the Cosmos DB collection, as shown in *Figure 8.23*:

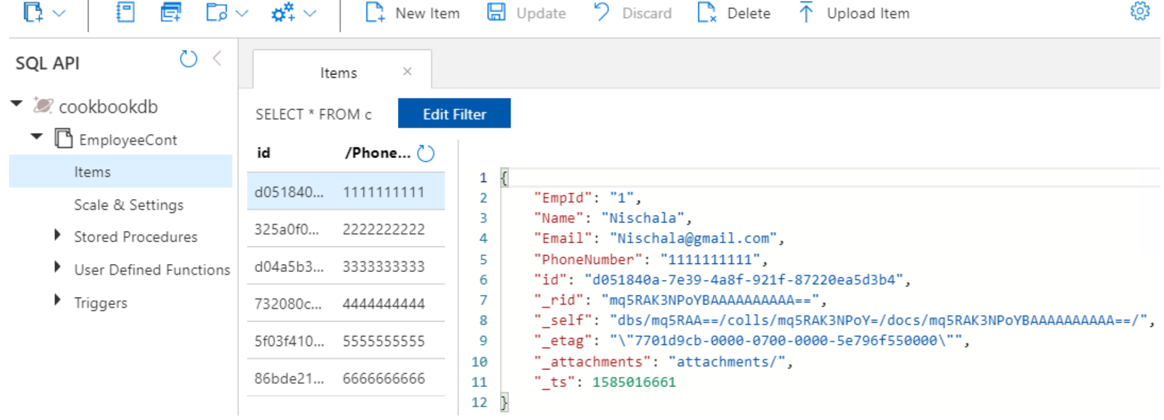


Figure 8.23: Cosmos DB—viewing documents

There's more...

The Cosmos DB team has released a library called Cosmos DB bulk executor, which can be used to perform bulk updates to a Cosmos DB container. Learn more about this at https://docs.microsoft.com/azure/cosmos-db/bulk-executor-overview.

In this recipe, we have hardcoded the names of our collection and database. We'll have to configure them in the app settings file.

In this chapter, you have learned how to develop a reliable application that can be used to upload CSV files using Durable Functions.

Configuring security for Azure Functions

In this chapter, we'll learn a few of the best practices that can be followed while working with Azure Functions, such as the following:

- Enabling authorization for function apps
- Controlling access to Azure Functions using function keys
- Securing Azure Functions using Azure Active Directory
- Throttling Azure Functions using API Management
- Securely accessing an SQL database from Azure Functions using Managed Identity
- Configuring additional security using IP whitelisting

Introduction

Even after the successful development of your application, and alongside continued maintenance and troubleshooting, there remains the concern of app security. Though covering all the security guidelines wouldn't be possible in just one chapter, we'll touch on a few of the techniques that every developer should follow while working with Azure Functions.

> **Note**
>
> The Azure Functions UI, shown in the screenshots in this chapter, is in preview at the time of writing. If this is still the case when reading this, click on the **Preview the new Azure Functions management experience** link as shown in *Figure 9.1* to navigate to the new UI.

Figure 9.1: Azure Functions—Preview the new Azure Functions management experience

Let's start learning about and implementing the security best practices for Azure Functions.

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.

In this recipe, you are going to learn how to enable authorization in Azure Functions and will gain clarity on the different types of authorization.

Getting ready

You should know by now how to create an HTTP trigger function. Download the Postman tool from 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...

In this section, we'll create and test the HTTP trigger's authorization functionality by performing the following steps:

1. Create a new HTTP trigger function (or open an existing HTTP function). When creating the function, select `Function` as the option in the **Authorization level** drop-down menu.

> **Note**
>
> If you would like to change the authorization level to an existing HTTP trigger function, 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:

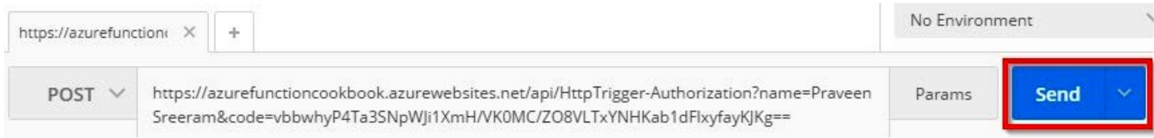


Figure 9.2: Postman—POST request to Azure Functions with the code query string parameter

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 **Authorization level** to `Function`.

   `name`: This is a query string that is required by the HTTP trigger function.

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 clients 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 Service) in this way:

- Azure Functions first checks for the presence of the header name `code`, either in the query string collection or in the request body.
- If the value of the `code` parameter is valid, then the request will be authorized and the runtime will process the request. Otherwise, a 401 unauthorized error message will be thrown.

There's more...

Note that the security key (in the form of the query string parameter named `code`) in this recipe is used for demonstration purposes only. In production scenarios, instead of passing the key as a query string parameter (the `code` parameter), add `x-functions-key` as an HTTP header, as shown in *Figure* 9.3:

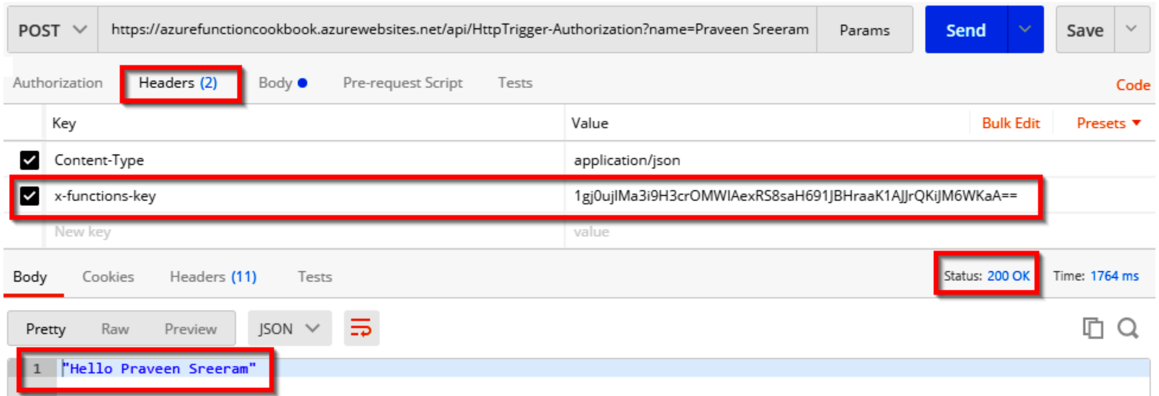


Figure 9.3: Postman—POST request to Azure Functions with the x-functions-key header

In this recipe, you have learned how to configure authorization for Azure Functions. Let's move on to the next recipe.

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 **Authorization level** field with the `Function` value in the **Integrate** tab of the HTTP trigger function. It works well when we use only one Azure function as a back-end web API for one of the applications and we don't want to provide access to the public.

However, in enterprise-level applications, we will end up developing multiple Azure functions across multiple function apps. In those cases, we need to have fine-grained granular access to Azure Functions for our own applications or for some other third-party applications that integrate our APIs in their applications.

This recipe will focus on understanding 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 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

When developing an API using Azure functions that can be used by multiple client applications, it's good practice to have a different function key for each client application that is going to use our functions.

Perform the following steps to configure the function key:

1. Navigate to the **Functions** tab, as shown in *Figure* 9.4:

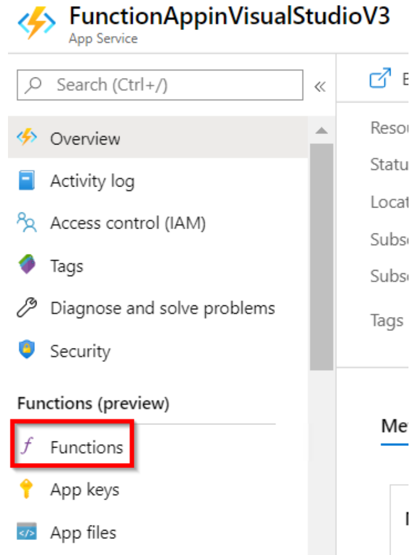


Figure 9.4: Azure Portal—link to Azure Functions

2. Now click on the Azure function (HTTP trigger) for which you would like to generate the keys:

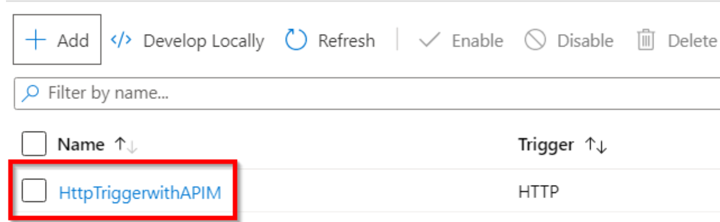


Figure 9.5: Navigate to the Azure function HTTP trigger

By default, a key with the name **`default`** will be generated for you. To generate a new key, click on the **New function key** button, as shown in *Figure* 9.6:

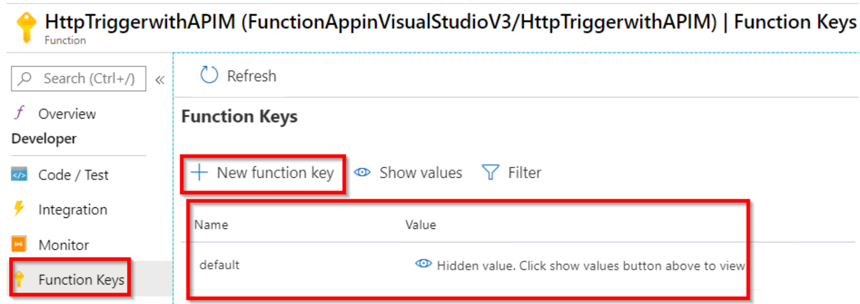


Figure 9.6: Azure function keys—creating a new function key

3. As per the preceding instructions, I have created keys for the following applications:

   `WebApplication`: The key name **`WebApplication`** is configured to be used for 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:

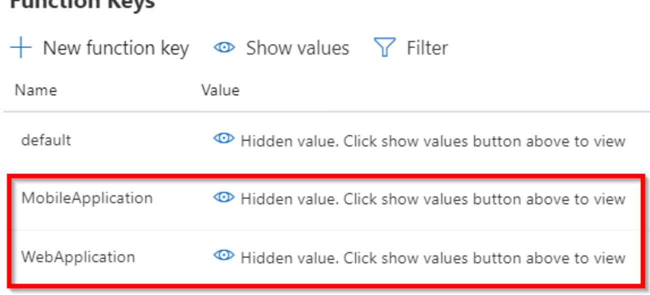


Figure 9.7: Azure function keys—list of function keys

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 for the different applications that are able to use the function. 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 situations like these, 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 we have many functions and client applications leveraging the same APIs. Managing the function keys in these large enterprise applications with huge client bases would be painful. To make things simple, segregate all related functions into a single function app and configure the authorization for each function app, instead of for each individual function.

Navigate to the **App keys** tab as shown in *Figure* 9.8:

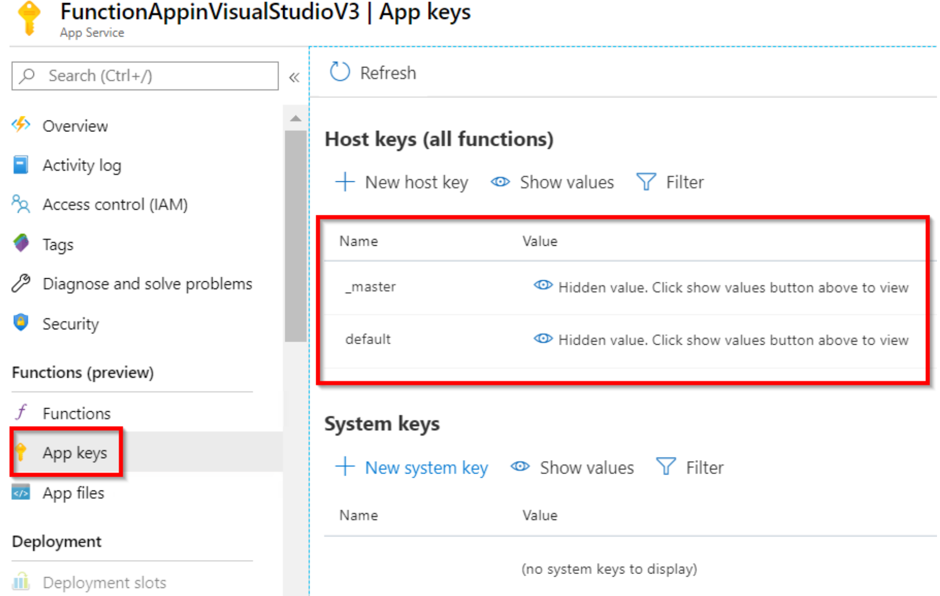


Figure 9.8: Azure Functions host keys

> **Note**
>
> As with the case of function keys, multiple host keys can be created if your function apps are used by multiple applications. In such cases, access to each of the function apps can be controlled by different applications using different keys.
>
> You can create multiple host keys by following the same steps we used for creating regular function keys.

There's more...

If a key has been compromised, then you can regenerate the key at any time by clicking on the **Renew** button. Note that when you renew a key, all the applications that access the function will no longer work and will return a 401 unauthorized error.

The key can be deleted or revoked if it is no longer used in any applications. Here's a table with some more guidance on key usage:

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	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	Use the host key for all the functions within the function app.
Function key	When the authorization level is Function	Yes	Yes	Use the function key only for a given function.

Figure 9.9: When to use Azure Functions app keys

> **Note**
>
> Microsoft doesn't recommend sharing the master key, as it is also used by runtime APIs. Be extra cautious with master keys.

In this recipe, you have learned how to enable security for HTTP triggers using function keys and admin keys. In the next recipe, we'll secure our Azure Functions using Azure Active Directory.

Securing Azure Functions using Azure Active Directory

One of the most important Azure Services related to security is **Azure Active Directory** (**Azure AD**). Azure AD is a cloud-based identity and access management service that helps developers to authenticate end users before accessing Azure Functions HTTP triggers. Azure Functions provides an easy way to integrate Azure AD with HTTP triggers called **EasyAuth**.

Thanks to Azure App Service, from which the **EasyAuth** feature is inherited, we can integrate Azure function HTTP triggers with Azure AD without writing a single line of code.

Getting ready

In this recipe, to make things simple, let's use the default Active Directory that is created when we create an Azure account. In real-time production scenarios, however, we'd ideally have an existing Active Directory that needs to be integrated. I would recommend going over this article for more information: docs.microsoft.com/azure/active-directory-b2c/tutorial-web-app-dotnet?tabs=applications.

How to do it...

This recipe will involve the following:

- Configuring Azure Active Directory for the function app
- Registering the client app in Azure Active Directory
- Granting the client app access to the back-end app
- Testing the authentication functionality using a JWT token

Configuring Azure Active Directory for the function app

In this section, we'll integrate the default Azure Active Directory with the function app. In order to integrate Azure Active Directory, please perform the following steps:

1. Navigate to the **Overview** section of Azure Functions and search for `Authentication` (or just `auth`) in the **Features** tab, as shown in *Figure* 9.10, and click on the **Configure** button:

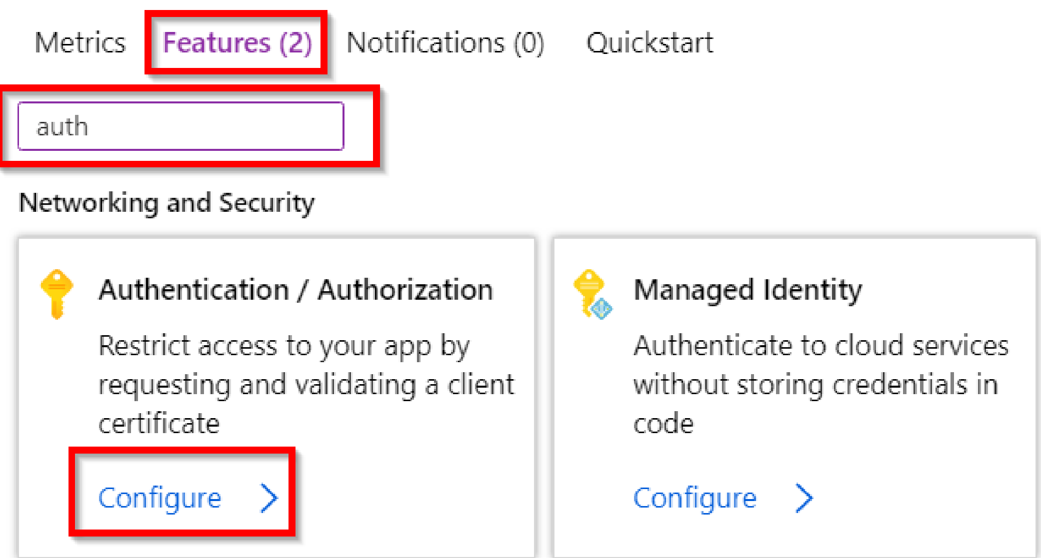


Figure 9.10: Function app overview page—searching for the authentication tile

2. In the **Authentication / Authorization** blade, perform the following steps to enable Active Directory authentication:

   Click on the **On** button to enable authentication.

   Choose the **Login using Azure Active Directory** option in the **Action to take when the request is not authorized** drop-down menu.

   Click on the **Not Configured** button of the **Azure Active Directory** field under the **Authentication Providers** section to start configuring the options, as shown in *Figure* 9.11:

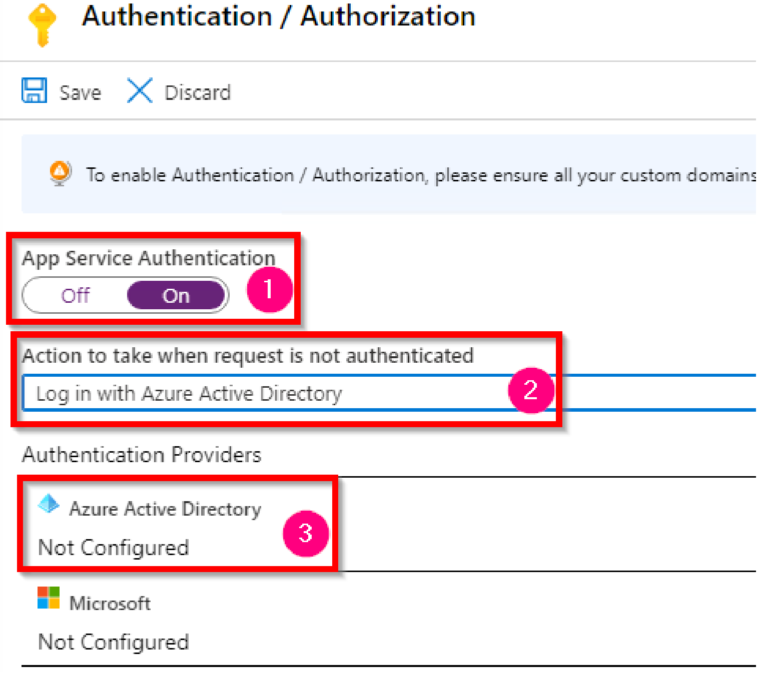


Figure 9.11: Azure function app—enabling authentication

3. The next step is to choose an existing registration or create a new registration for the client application that you want to provide access to. This can be done by pressing the **Express** button in the **Management mode** field, as shown in *Figure 9.12*:

Active Directory Authentication

These settings allow users to sign in with Azure Active Directory. Click here to learn more. Learn more

Management mode

Figure 9.12: Azure function app—choosing Express mode

4. Now, choose **Create New AD App** and provide **`AzureFunctionCookbookV3`** as the name in the **Create App** field. Click **OK** to save the configurations:

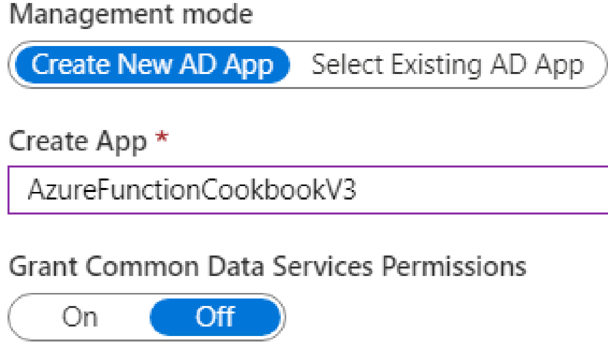


Figure 9.13: Azure function app—creating a new Azure Active Directory app

5. Now, an **App registrations** entry will be created for you with the name **`AzureFunctionCookbookV3`**. This can be viewed in the **App registrations** blade of the **Azure Active Directory** service:

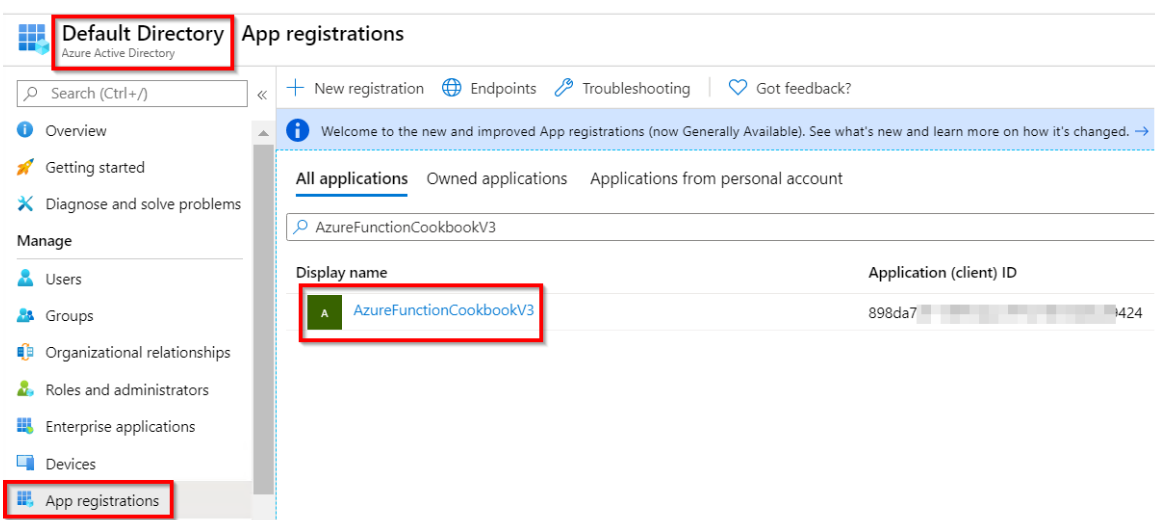


Figure 9.14: Azure Active Directory—App registrations

6. Grab the application ID, as shown in *Figure* 9.14, and store it in a Notepad file.

7. That's it. Without writing a single line of code, we are done with configuring an Azure Active Directory instance that sits as a security layer and allows access only to authenticated users. Let's quickly test it by trying to access any of the HTTP triggers present in the function app. As shown in *Figure* 9.15, try to access the HTTP trigger function using Postman. As expected, it will redirect you to log in. *Figure* 9.15 shows how it looks when you try to access the HTTP trigger:

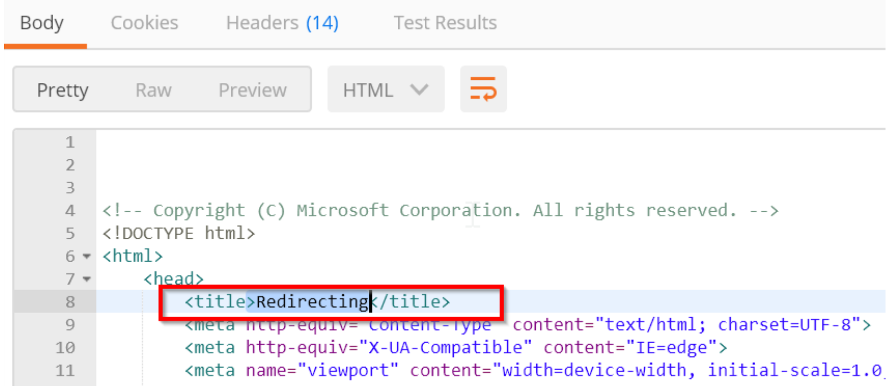


Figure 9.15: Postman—accessing the HTTP trigger

8. As you have integrated the function app with Azure Active Directory, it is not possible to access your back-end API (HTTP trigger). In order to provide access to the client applications that need to consume the HTTP trigger, you need to perform the following steps:

   Register the client apps in Azure Active Directory (for our example, we'll register the Postman app).

   Grant access to the client app created in *Step a* to access the back-end function app.

Registering the client app in Azure Active Directory

In order to provide access to a client app, you need to register the client app in Azure Active Directory and grant access to the HTTP trigger of the function app. In order to achieve this, perform the following steps:

1. Navigate to Azure Active Directory by clicking on the **Azure Active Directory** button, as shown in *Figure* 9.16. If this option is not available in the **FAVORITES** list, search in the **All services** blade, which is also highlighted in *Figure* 9.16:

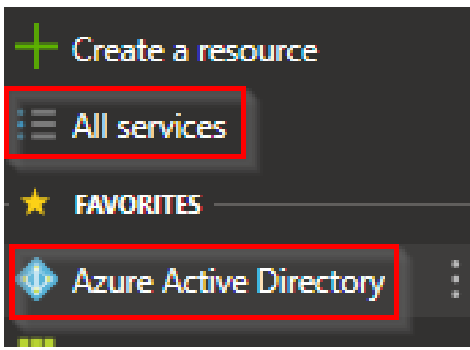


Figure 9.16: Azure portal menu—adding the Azure Active Directory link

2. In the Active Directory menu, click on **App registrations** and then click on the **New 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 using `http://localhost` should be good for our example:

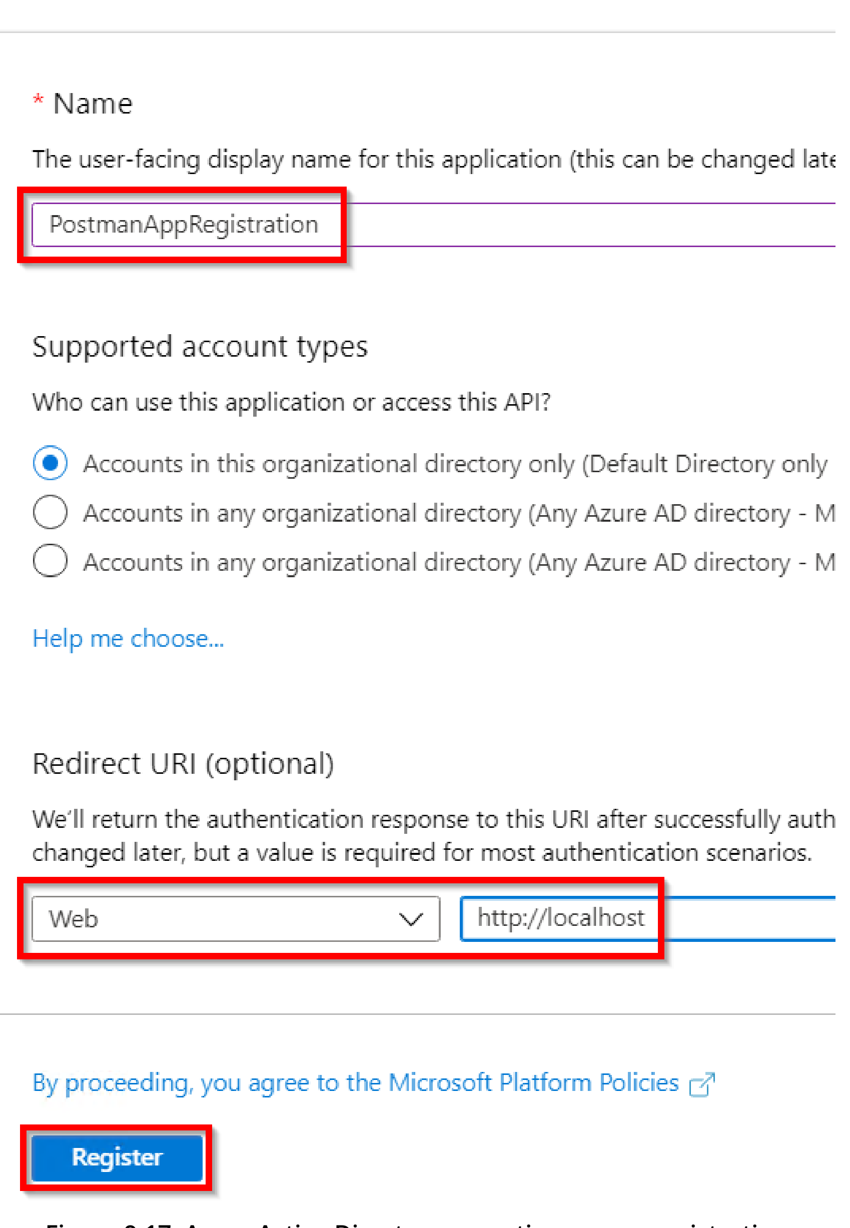


Figure 9.17: Azure Active Directory—creating an app registration

4. In just a moment, the app will be created, and you'll be taken to the screen shown in *Figure* 9.18. Grab the application ID and save it in a Notepad file. You'll be using it in the upcoming steps:

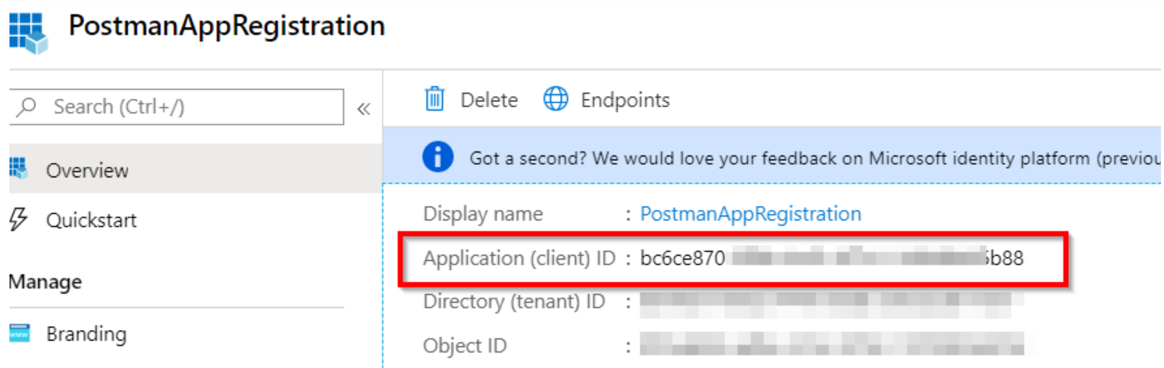


Figure 9.18: App Registration overview blade—copying the application ID

5. In the **Certificates & secrets** blade, click on the **New client secret** button item to generate a key, which we will be passing from Postman:

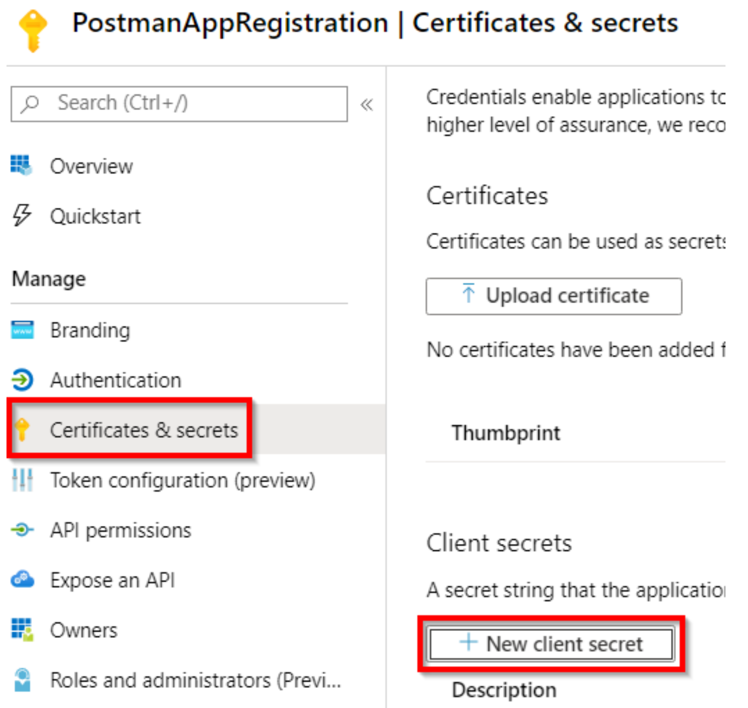


Figure 9.19: Azure app registration—Certificates & secrets

6. In the **Add a client secret** pop-up box, we first need to provide a description and the duration after which the key should expire.

7. Provide the details as shown in *Figure 9.20* and click on the **Add** button. The actual secret will be displayed to you in the value field only once immediately after clicking on the **Add** button, so be sure to copy it and store it in a secure place. You'll be using this in a few moments:

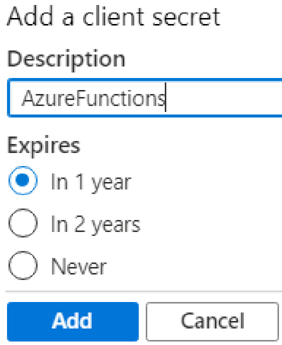


Figure 9.20: Azure app registration—creating a new secret

In this section, we created the app registration, along with a secret. Let's move on to the next section.

Granting the client app access to the back-end app

Once the client application is registered, you need to provide it with access to your back-end app. In this section, you'll learn how to configure it. Perform the following steps:

1. In **PostmanAppRegistration**, click on **API permissions**, as shown in *Figure 9.21*:

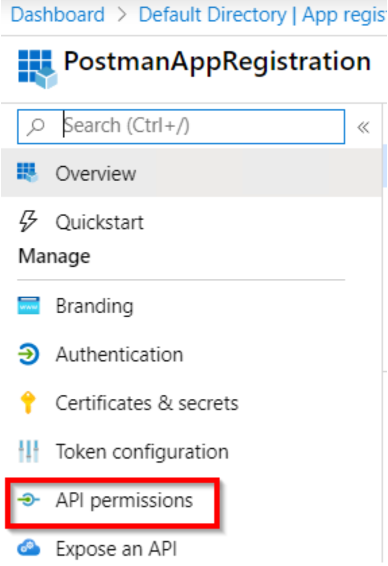


Figure 9.21: Azure app registration—API permissions

2. In the **API permissions** blade, click on the **Add a permission** button to navigate to the **Request API permissions** blade. Now, choose the **APIs my organization uses** tab and search for the app registration (in my case, it was `AzureFunctionCookbookV3`), as shown in *Figure* 9.22. Once the app registration is visible, click on it:

Request API permissions

Select an API

Microsoft APIs | APIs my organization uses | My APIs

Apps in your directory that expose APIs are shown below

AzureFunctionCookBookV3

Name	Application (client) ID
AzureFunctionCookbookV3	898…9424

Figure 9.22: Azure app registration—the Request API permissions blade—selecting the Azure function

3. In the next step, select **Delegated permissions**, click on the `user_impersonation` checkbox, and then click on the **Add permissions** button, as shown in *Figure* 9.23:

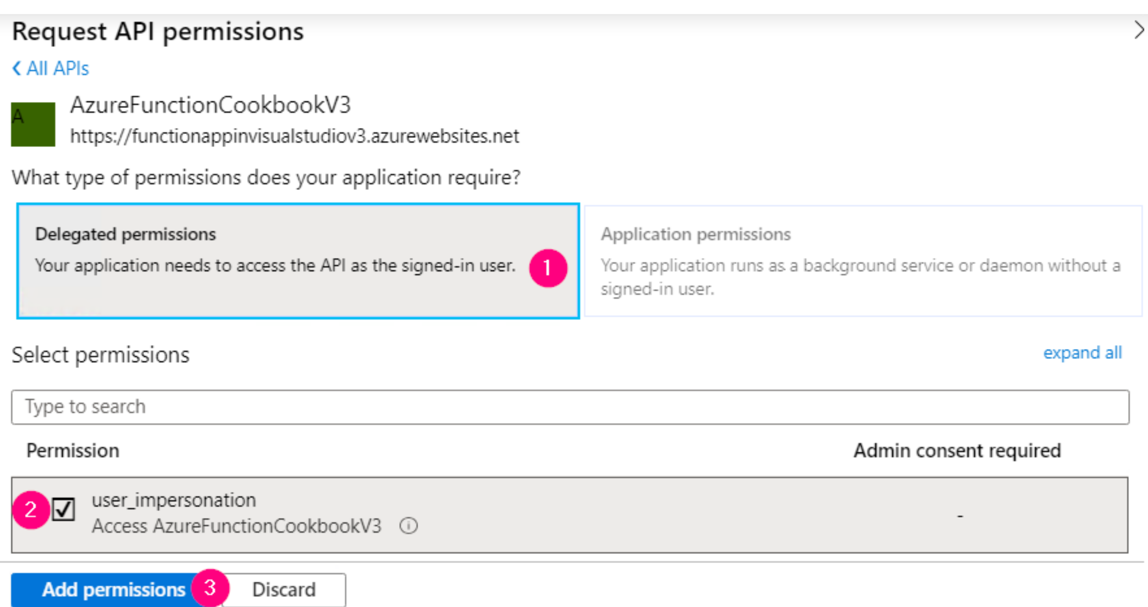


Figure 9.23: Azure app registration—the Request API permissions blade—adding permissions

4. Ensure that the following screen is visible. Clicking on the **Grant admin consent for Default Directory** button will apply the changes:

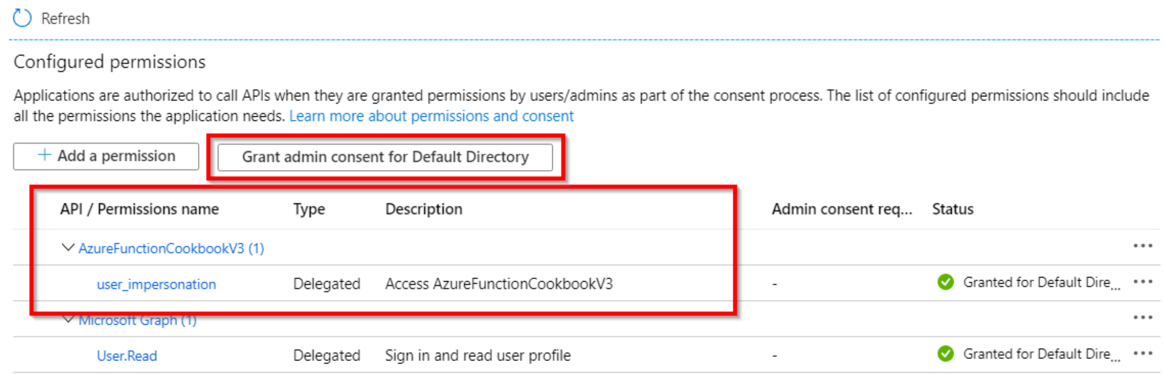


Figure 9.24: Azure app registration—Configured permissions

In this section, we granted the necessary permissions to the Azure function app. Let's move on to test the authentication functionality.

Testing the authentication functionality using a JWT token

In order to test the functionality, you need to use Postman. Carry out the following steps:

1. Get the following input details:
2. **OAuth 2.0 token endpoint**: Get this in the **Endpoints** tab of Azure Active Directory and copy the URL.
3. **Grant type**: A hardcoded `client_credentials` value.
4. **Client ID of the client application**: This was noted down in *Step* 4 of the *Registering the client app in Azure Active Directory* section.
5. **Secret that was generated for client application**: You copied it into Notepad in *Step* 6 of the *Registering the client app in Azure Active Directory* section.
6. **Scope**: The resource that you need to access. You need to pass the scope of the back-end application. You'll pass the default scope, which will be in `https://<functionappname>.azurewebsites.net/.default` format.
7. Once you have all the information at hand, pass all the parameters and make a call to the Azure Active Directory tenant, which will return the bearer token as shown in *Figure* 9.25. Copy the bearer token in a Notepad file:

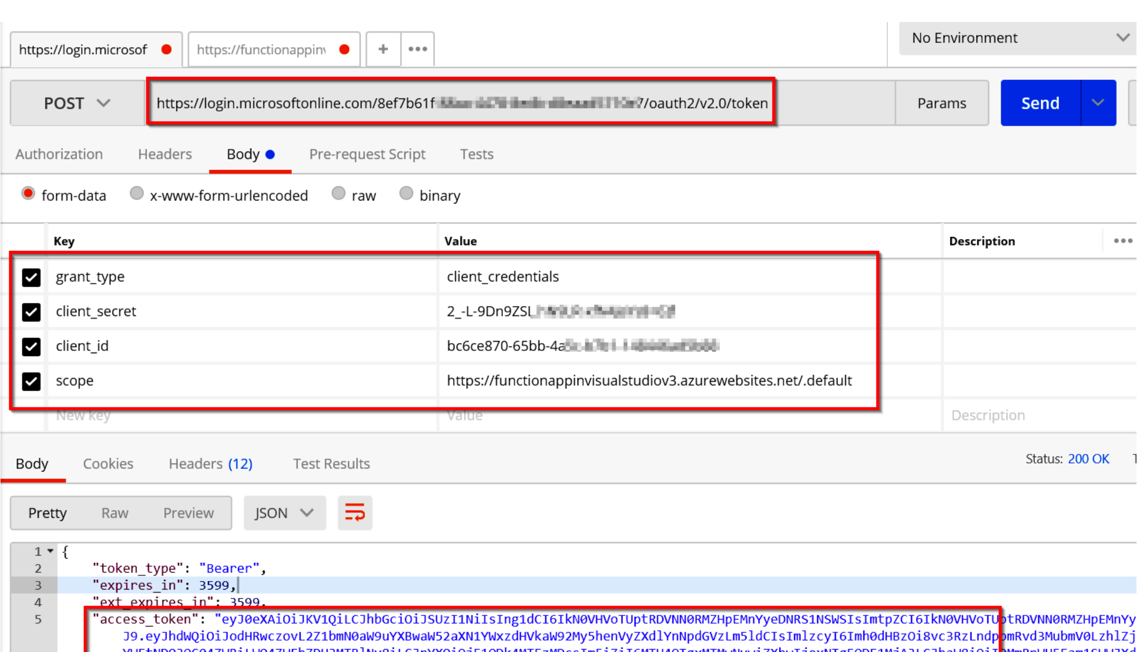


Figure 9.25: Azure Functions—requesting a bearer token

8. The next and final step is to make a call to the actual back end (the Azure function HTTP trigger) by passing the bearer JWT token (**access_token**) that you copied in the preceding step:

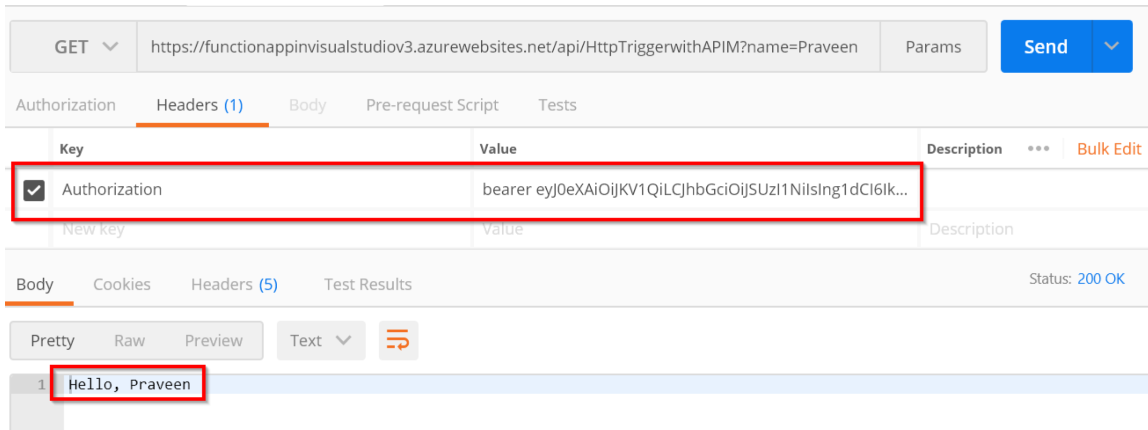


Figure 9.26: Azure function—invoking the HTTP trigger by passing a bearer token

As shown in *Figure 9.26*, add an **Authorization** header and paste the JWT token. Don't forget to provide the text `bearer` to the **Value** field.

In this recipe, you learned to configure authentication using Azure Active Directory without writing any code. In the next recipe, you'll learn how to integrate Azure API Management with Azure Functions to limit the number of requests from clients.

Throttling Azure Functions using API Management

You have already learned in previous chapters that we can use Azure Functions' HTTP triggers as a back-end web API. To restrict the number of requests by client applications to, let's say, 10 requests per second, we would usually have to develop a lot of logic. Thanks to Azure API Management, we don't need to write any custom logic if we integrate Azure Functions with API Management.

In this recipe, you'll learn how to restrict clients to only one API request per minute for a given IP address. The following are the high-level steps that we'll 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, you 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 production applications, you need to choose non-developer tiers (Basic/Standard/Premium), as the **Developer (No SLA)** tier doesn't provide any SLAs. After reviewing all the details, click on the **Create** button:

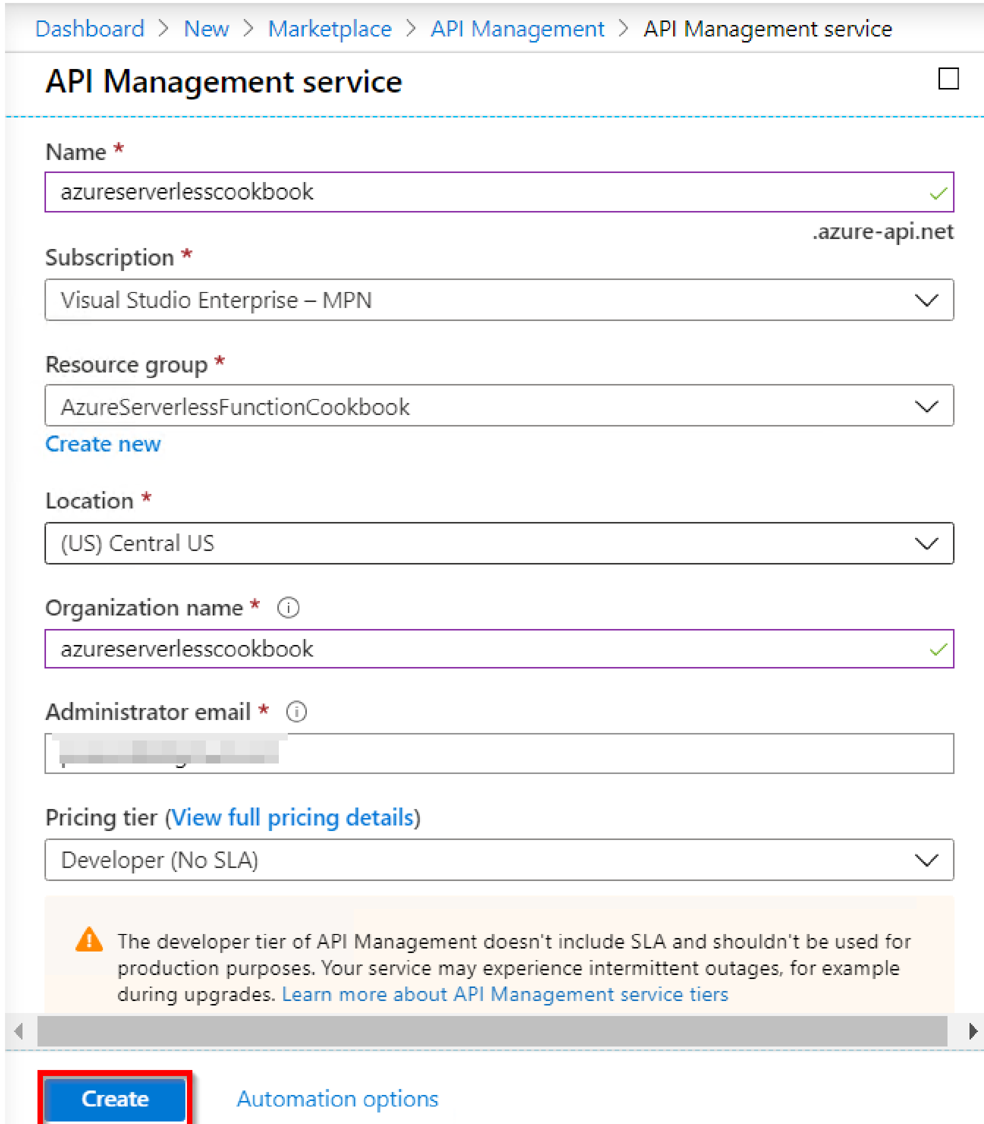


Figure 9.27: Creating an API Management service

2. At the time of writing, it takes around 30-40 minutes to create an API Management instance. Once it has been created, the instance can be viewed in the **API Management services** blade:

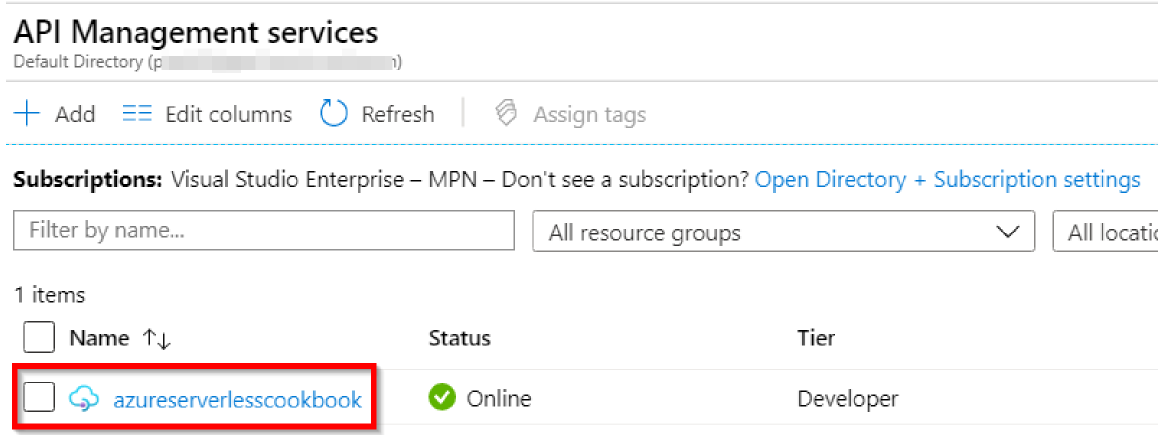


Figure 9.28: List of API Management services

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.

Integrating Azure Functions with API Management

In this section, you need to perform the following steps to integrate Azure Functions with the API Management service:

1. Navigate to the **APIs** blade of the API Management instance that you created, and click on the **Function App** tile.
2. You'll see a **Create from Function App** pop-up box where you can click on the **Browse** button, which will open a sidebar with the title **Import Azure Functions**, which is 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 chosen 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 *Figure* 9.29. I chose only one HTTP trigger to make things simple, and then clicked on the **Select** button, as shown in *Figure* 9.29:

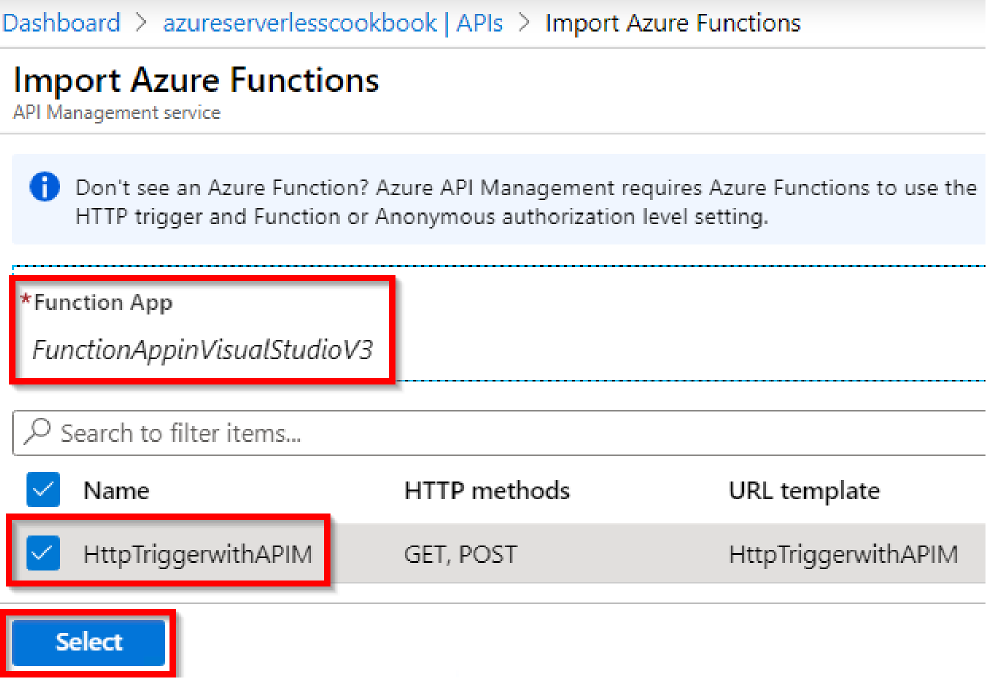


Figure 9.29: API Management services—importing Azure Functions

4. After performing all the preceding steps, the **Create from Function App** pop-up box will appear, as shown in *Figure* 9.30. Once you have reviewed the details, click on the **Create** button:

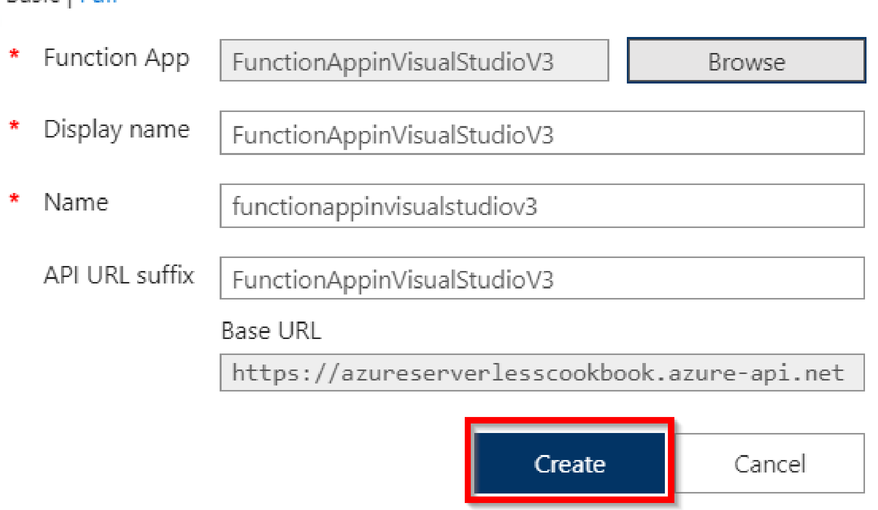


Figure 9.30: API Management services—creating APIs from Azure Functions

5. If everything goes fine, you should get something as shown in *Figure* 9.31. Now you are done with integrating Azure Functions with API Management:

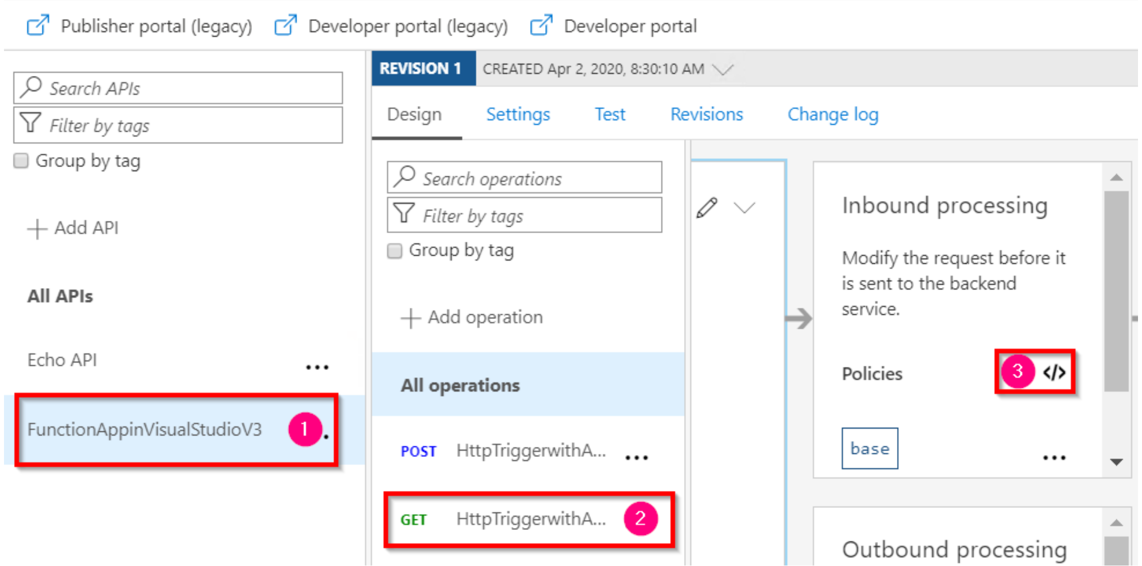


Figure 9.31: API Management services—configuring inbound policies

In this section, you learned how to import Azure Functions APIs into the API Management service. Let's move on to the next section.

Configuring request throttling using inbound policies

Perform the following steps to configure throttling using inbound policies:

1. As shown in *Figure* 9.31, choose the required operation (**GET**) and click on the inbound policy editor link (labeled **3** in *Figure* 9.31), which will open the policy editor.

> **Note**
>
> API Management allows us to control the behavior of the back-end APIs (in our case, HTTP triggers) using API Management policies. Both the inbound and outbound request responses can be controlled. Read more about it at docs.microsoft.com/azure/api-management/api-management-howto-policies.

2. As you need to restrict the request rate within API Management before sending the request to the back-end function app, you 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. Finally, set `counter-key` to the IP address of the client application:

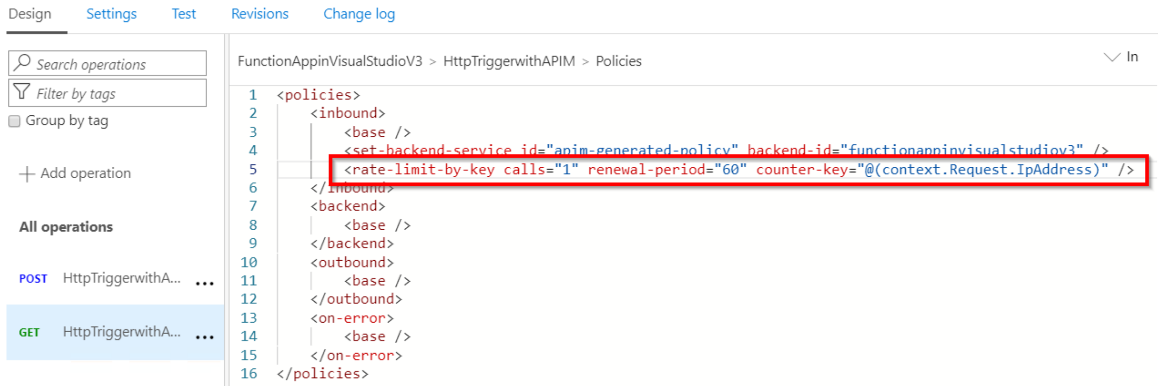


Figure 9.32: API Management services—configuring inbound policies—request throttling

> **Note**
>
> With this inbound policy, you are instructing API Management to restrict requests to one per minute for a given IP address.

3. Before you test the throttling, one final step is to publish the API by navigating to the **Settings** tab in the preceding step and associating the API with a published product (in your case, you have a default **Starter** product that is already published). As shown in *Figure* 9.33, choose the required product and click on the **Save** button:

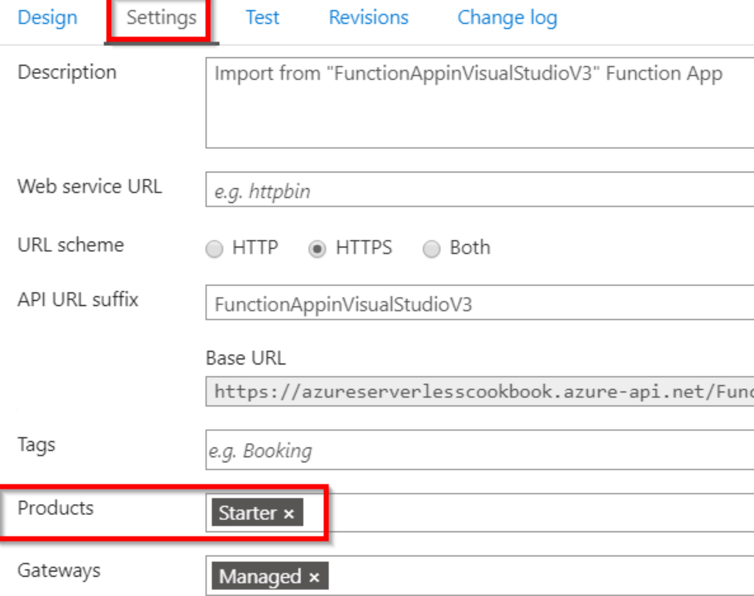


Figure 9.33: API Management services—configuring products

> **Note**
>
> 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 docs.microsoft.com/azure/api-management/api-management-howto-add-products.

Testing the rate limit inbound policy configuration

Test the rate limit by performing 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 after completing the preceding step to make the first request. You should see something similar to *Figure* 9.34 after getting a response from the back end:

HTTP response

Message Trace

```
HTTP/1.1 200 OK

content-encoding: gzip
content-length: 138
content-type: text/plain; charset=utf-
date: Thu, 02 Apr 2020 09:13:33 GMT
ocp-apim-trace-location: https://apims
28&sr=b&sig=wjkJGFZ1ObGLAq5Z%2FIESn%2F
vary: Accept-Encoding,Origin
Hello, Praveen Sreeram
```

Send Bypass CORS proxy

Figure 9.34: API Management services—testing the API in the console

3. Now, immediately click the **Send** button again. As shown in *Figure* 9.34, an error should be returned, as our inbound policy rule is to allow only one request per minute from a given IP address:

HTTP response

Message ! Trace

```
HTTP/1.1 429 Too Many Requests

content-length: 84
content-type: application/json
date: Thu, 02 Apr 2020 09:14:45 GMT
ocp-apim-trace-location: https://apimstbog31rv6ar9uwbf9qr.blob.core
28&sr=b&sig=n%2BSVQf8FvD2bYPkUjw55ptEuOAkLUEXlivFERUIzB%2F0%3D&se=2
retry-after: 58
vary: Origin
{
    "statusCode": 429,
    "message": "Rate limit is exceeded. Try again in 58 seconds."
}
```

Send Bypass CORS proxy

Figure 9.35: API Management services—testing rate limiting rules in the console

How it works...

In this recipe, we have created and configured 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:

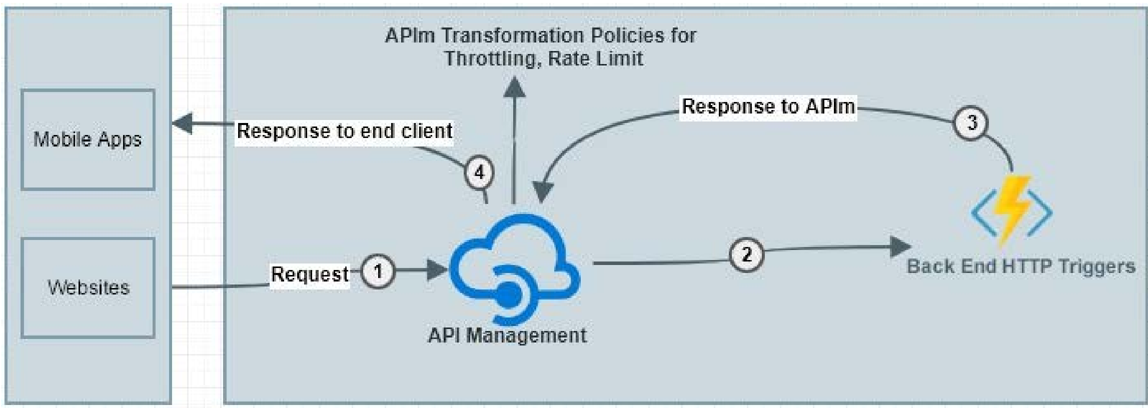


Figure 9.36: API Management integration with Azure Functions

The following is the overall process that we have configured in this recipe:

1. The **API Management** service receives the **Request**.
2. The **API Management** gateway forwards the request to the HTTP triggers. The request is forwarded only if the inbound policy is adhered to. Otherwise, an error is returned immediately.
3. The HTTP triggers respond to **API Management** with a response.
4. Finally, the response is sent to the end user by the **API Management** service.

Let's move on to the next recipe.

Securely accessing an SQL database from Azure Functions using Managed Identity

Let's say an employee has changed the password of the account as per their firm's security policy (to rotate the password every month). The applications using that account now wouldn't be able to gain access. For developers, wouldn't it be good if there was a facility where we don't need to worry about the credentials and, instead, the framework took care of authentication? In this recipe, you will learn how to access an SQL database from an Azure function (using Visual Studio) without providing a user ID or password by using a feature called Managed Service Identity.

How to do it...

In this recipe, we are going to perform the following steps:

1. Creating a function app using Visual Studio (if not done already)
2. Creating an SQL database
3. Enabling Managed Service Identity from the portal
4. Allowing SQL Server access to the new Managed Service Identity
5. Executing the HTTP trigger and testing

We'll use Visual Studio to develop an Azure HTTP trigger that connects to Azure SQL Database without providing any credentials (that is, the username and password).

Creating a function app using Visual Studio

In this section, we'll develop an Azure HTTP trigger using Visual Studio that connects to the database.

Perform the following steps:

1. Create a new function app by choosing the Azure Functions v3 runtime.
2. Create a new HTTP trigger with the name **`HttpTriggerWithMSI`** using **Anonymous Authorization level**.
3. Install the NuGet package with **`Install-Package System.Data.SqlClient`** using the package manager console.
4. Now, replace the function with the following code for the HTTP trigger:

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


            string firstname = string.Empty,
              lastname = string.Empty, email = string.Empty, devicelist =
string.Empty;

            string requestBody = await new StreamReader(req.Body).
ReadToEndAsync();
            dynamic data = JsonConvert.DeserializeObject(requestBody);
            firstname = firstname ?? data?.firstname;
            lastname = lastname ?? data?.lastname;
            email = email ?? data?.email;
            devicelist = 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;Initial Catalog=database;Persist SecurityInfo=False;
MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=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 new OkObjectResult("Hello, Successfully inserted the
data");

        }
}
```

> **Note**
>
> The connection string in the preceding code doesn't have any user ID or password details; it just has the server name and the database name.

5. To retrieve the access token, run the following code:

```
con.AccessToken = (new AzureServiceTokenProvider()).
GetAccessTokenAsync("https://database. windows.net/").Result;
```

6. Add the following NuGet packages to the function app:

```
Install-Package Microsoft.Azure.Services.AppAuthentication
```

7. 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;
```

8. After ensuring that there are no build errors, publish the function app by right-clicking on the project. Then, click on the **Publish** button, which will open the **Pick a publish target** window, as shown in *Figure* 9.37. Choose **Azure Functions Consumption Plan**, click on **Select Existing**, and then click the **Create Profile** button:

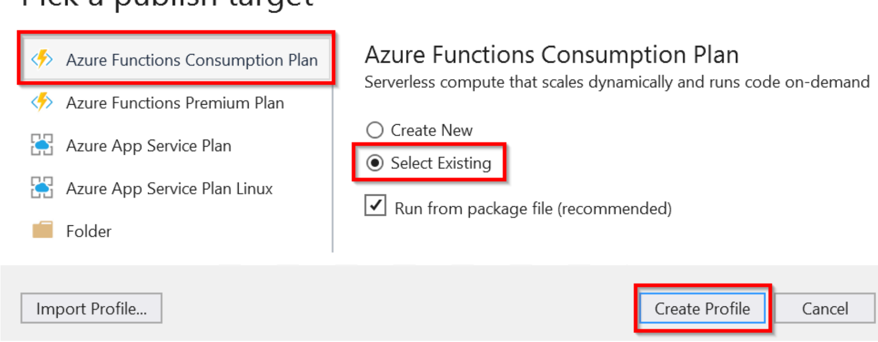


Figure 9.37: Visual Studio—picking a publish target

9. Next, provide values for **Resource Group** and **Function App**, click on the **OK** button, and then click on the **Publish** button to publish the HTTP trigger to the Azure function app.

In this section, we created the function app. Let's move to the next section to create the database.

Creating an SQL database

Create an SQL database by performing the following steps:

1. Click on **Create a resource** and search for **`SQL database`**, as shown in *Figure* 9.38:

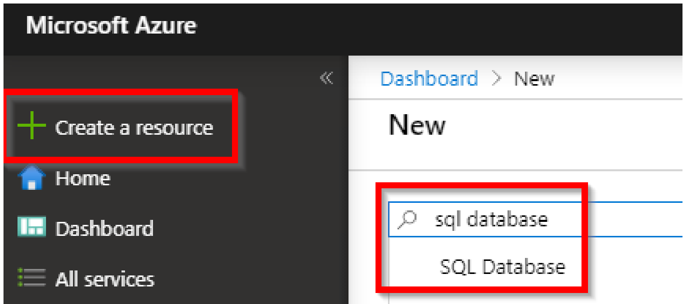


Figure 9.38: Search for SQL database in the Azure portal

2. In the **Create SQL Database** blade, provide all the details to create the SQL database, as shown in *Figure* 9.39:

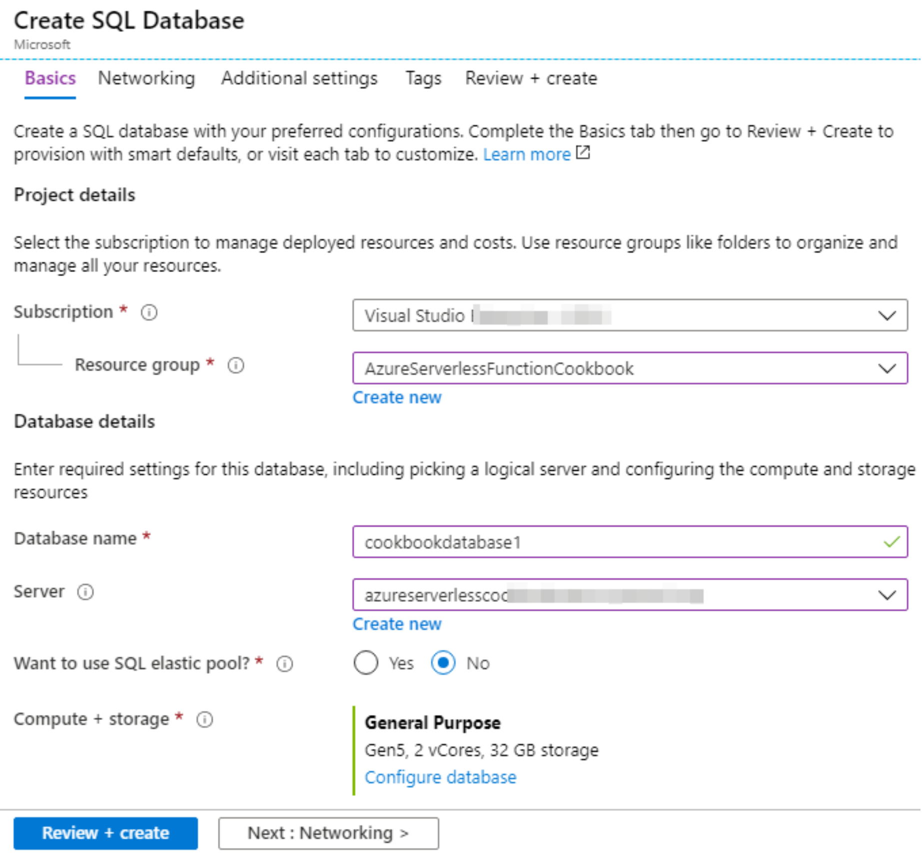


Figure 9.39: Creating an SQL database

In this section, you learned how to create an SQL database in an existing SQL server. Let's move on to the next section.

Enabling Managed Identity

Managed Identity is a feature of Azure Active Directory that will let the program authenticate the service automatically without providing any credentials. In this section, we'll enable Managed Identity for our Azure functions. Perform the following steps to do this:

1. Navigate to the **Identity** tab. Under the **System assigned** tab, click the **On** button of the **Status** toggle button and click on **Save**, as shown in *Figure* 9.40:

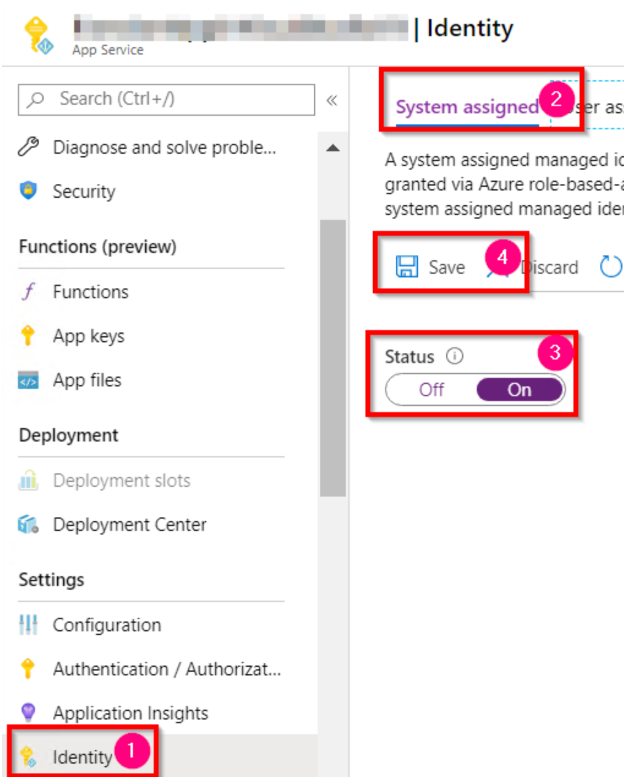


Figure 9.40: Azure Functions—enabling a system assigned managed identity

2. After clicking on the **Save** button, a pop-up box will be displayed, as shown in *Figure 9.41*. Click on **Yes**:

Enable system assigned managed identity

'FunctionAppforCookbookwithMSI' will be registered with Azure Active Directory. Once it is registered, 'FunctionAppforCookbookwithMSI' can be granted permissions to access resources protected by Azure AD. Do you want to enable the system assigned managed identity for 'FunctionAppforCookbookwithMSI'?

Yes No

Figure 9.41: Azure Functions—enabling a system assigned managed identity—confirmation

3. Once the details are saved, the object ID will be displayed, as shown in *Figure 9.42*. Grab the object ID and keep it in a Notepad file. We will use it in the following *Allowing SQL Server access to the new Managed Identity service* section of this recipe:

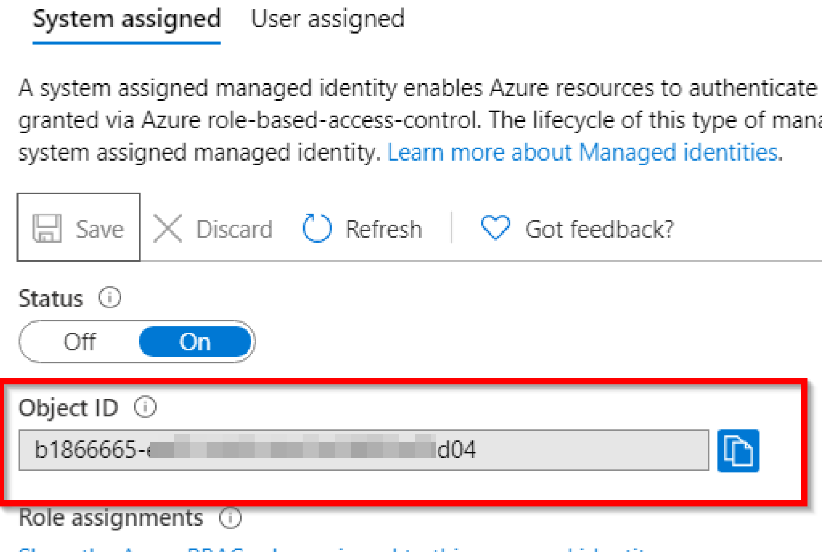


Figure 9.42: Azure Functions—system assigned managed identity—copying the object ID

In this section, we enabled the system assigned managed identity. Let's move on to the next section.

Allowing SQL Server access to the new Managed Identity service

In this section, we'll create an admin user that has access to the SQL server that our function app will connect to. Perform the following steps:

1. Authenticate your Azure account's identity using the Azure CLI by running the **az login** command in Command Prompt, as shown in *Figure 9.43*:

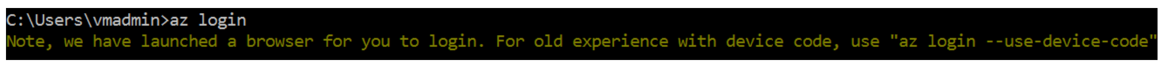


Figure 9.43: Command Prompt—using the az login command

2. You'll be prompted to provide your Azure account credentials to log in to the Azure portal. Once you have provided your credentials, it will show you the available subscriptions in the command console.
3. Run the following command in Command Prompt by passing the object ID that you noted in *Step 3* of the previous section:

```
az sql server ad-admin create --resource-group <<Resource Group name>>
--server-name <<SQL Server name>> --display-name sqladminuser --object-id
<object id>
```

The following is the output of the previous command:

```
C:\Windows\system32>az sql server ad-admin create --resource-group AzureServerlessFunctionCookbook --server-name
azureserverlesscookbookserver --display-name sqladminuser --object-id b18[illegible]2d04
{
  "id": "/subscriptions/[illegible]/resourceGroups/AzureServerlessFunctionCookbook/provi
ders/Microsoft.Sql/servers/azureserverlesscookbookserver/administratorOperationResults/ActiveDirectory",
  "kind": null,
  "location": "Central US",
  "login": "sqladminuser",
  "name": "ActiveDirectory",
  "resourceGroup": "AzureServerlessFunctionCookbook",
  "sid": "[illegible]",
  "tenantId": "[illegible]",
  "type": "Microsoft.Sql/servers/administrators"
}
```

Figure 9.44: Command Prompt—running the az commands

4. 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,
CONSTRAINT [PK_EmployeeInfo] PRIMARY KEY CLUSTERED (     [PKEmployeeId]
ASC
)     )
```

In this section, we enabled access to SQL Server using our object ID. Let's move on to the next section.

Executing the HTTP trigger and testing

In order to test whether the code can connect to the database without credentials (your username and password), perform the following steps:

1. Open Postman and submit a request as shown:

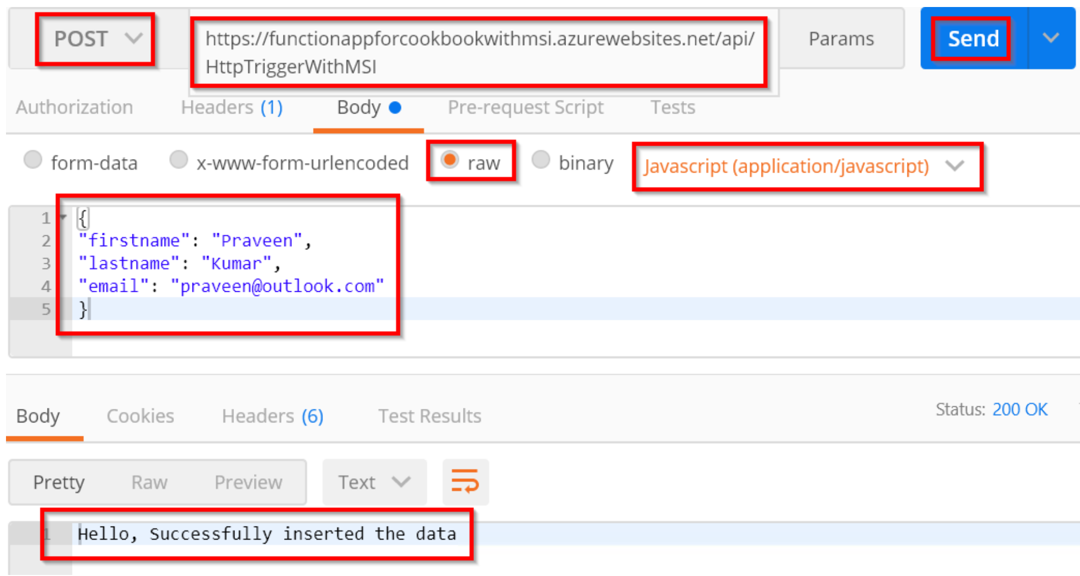


Figure 9.45: Postman—submitting a POST request to Azure Functions

2. Let's review the SQL database to see whether the record was inserted:

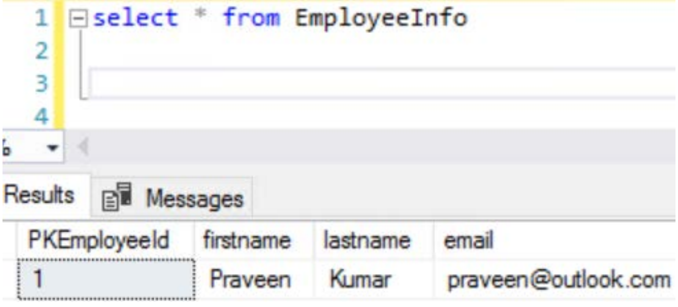


Figure 9.46: Postman—SSMS—Viewing the inserted data

In this recipe, we have learned how to access Azure SQL Database from an Azure function app without providing a password by leveraging the Managed Identity feature. Let's move on to the next recipe.

Configuring additional security using IP whitelisting

In this recipe, you'll learn a technique to secure and restrict access to your Azure functions only to those clients whose IP addresses are whitelisted.

Let's say you want to restrict the function app's access to the internal organization alone, as it will be used only by the users' apps hosted internally within the organization's network. To do this, you need to whitelist one or more IP addresses (or IP address ranges) to allow access to the Azure function app.

In the recipe, we are going to create access restriction rules. Rules are nothing but instructions on whether to allow or block access based on IP addresses, IP address ranges, and even virtual networks.

Getting ready…

Please create the following services if they are not created already:

- A function app
- An HTTP trigger function

How to do it...

In this section, you'll learn how to implement the whitelisting of IP addresses for a given function app:

1. Navigate to the Azure function app, click on the **Networking** blade, and then click the **Configure Access Restrictions** button, as shown in *Figure* 9.47:

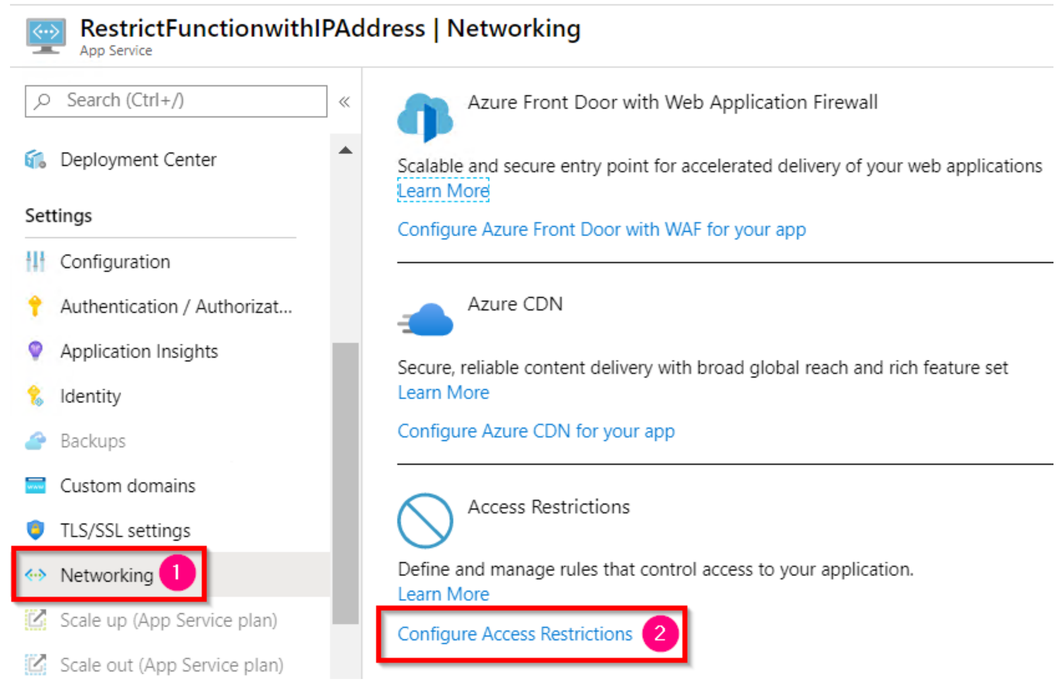


Figure 9.47: Azure Functions—Networking blade—Configure Access Restrictions

2. In the **Access Restrictions** blade, we can see a preconfigured rule that allows anyone to access the function app by default:

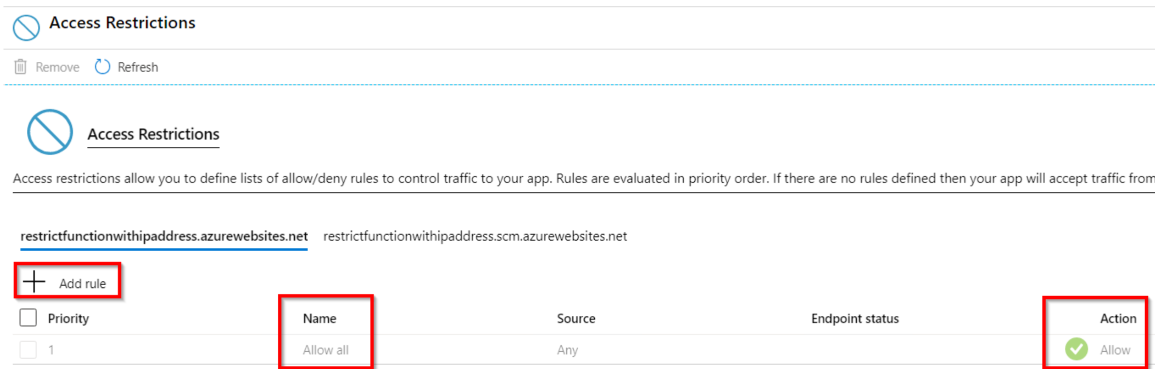


Figure 9.48: Azure Functions—Access Restrictions

3. Now create a new rule by clicking on the **Add rule** button to whitelist an IP address, as shown in *Figure* 9.49:

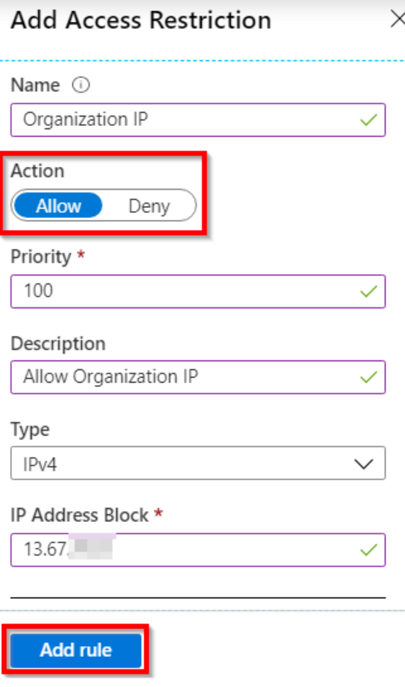


Figure 9.49: Creating a new Access Restriction rule

4. Provide a name, toggle the **Action** button to **Allow** to allow access (likewise, **Deny** is used to deny access), and then provide an IP address in the **IP Address Block** field. After reviewing all the required details, click the **Add rule** button, as highlighted in *Figure* 9.49.

5. As soon as a rule is created, it will be added to the list of rules, as shown in *Figure* 9.50:

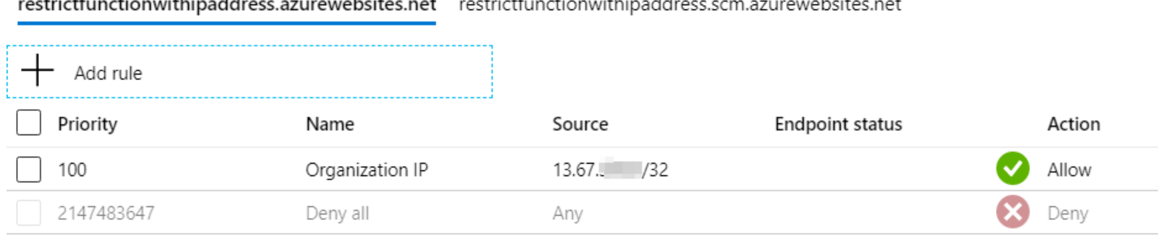


Figure 9.50: Azure Functions—list of access restrictions

6. Notice that the default **Allow all** rule has become a **Deny all** rule. This **Deny all** rule will restrict access to all other IP addresses except the IP that you have whitelisted using the **Allow** rule.

7. Now try to access the HTTP trigger that you created from the whitelisted IP. As shown in *Figure* 9.51, you will be able to access it:

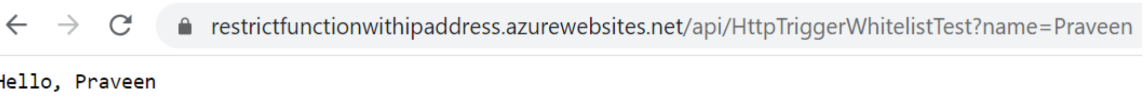


Figure 9.51: Azure Functions—testing the HTTP trigger using the browser

8. Now try it with another server that is not whitelisted. You should get an error, as shown in *Figure* 9.52:

Figure 9.52: 403 forbidden error when accessing the HTTP trigger from a blocked IP

There's more

The following is some additional information regarding access restrictions:

- If you don't have any other server to test the functionality, then set the rule to **Deny**, instead of **Allow**, as shown in *Figure* 9.53:

Figure 9.53: Azure Functions—Edit Access Restriction

- If you need an Azure App Service (hosted on Azure) to consume an HTTP trigger with access restrictions enabled, then whitelist all the outbound IP addresses of that App Service. To get the outbound IP addresses of the App Service, refer to *Figure 9.54*:

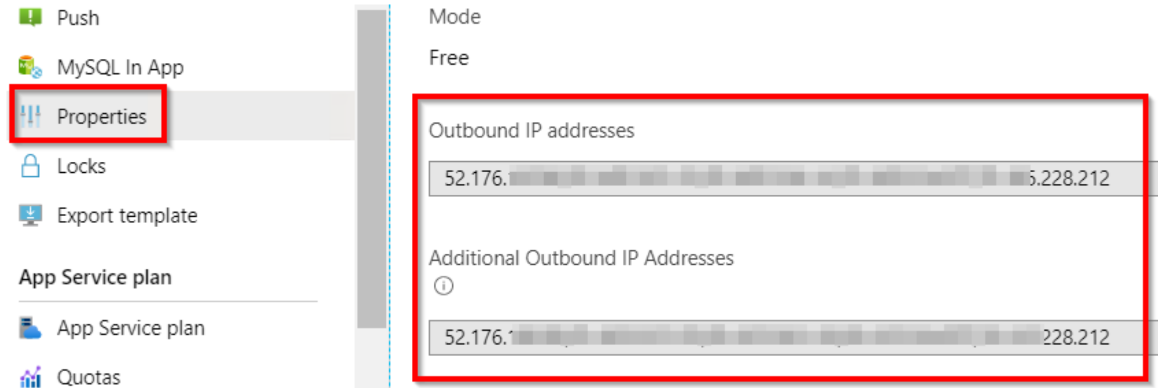


Figure 9.54: Azure App Service—Outbound IP addresses

In this recipe, you have learned how to restrict access to Azure functions and only whitelist certain IPs (such as your organization's IP addresses).

In this chapter, you have learned various ways of securing Azure functions, including:

- Securing individual HTTP triggers using authorization levels.
- Securing an entire function app using IP restrictions.
- Securing a function app based on users by using Azure Active Directory authentication.
- Allowing a function to securely access databases using managed identities.

Depending on the real-time scenarios in your projects, you can use any of the preceding techniques to improve the security of your applications.

10

Implementing best practices for Azure Functions

In this chapter, we'll learn some 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
- Avoiding cold starts by warming the app at regular intervals
- Sharing code across Azure functions using class libraries
- Migrating C# console application to Azure functions using PowerShell
- Implementing feature flags in Azure functions using the App Configuration service

Introduction

This chapter covers some of the most important and common best practices that are followed in cloud-native applications. Along with the best practices, you will also understand how to overcome some of the limitations of Azure functions. Furthermore, you will learn how to migrate jobs from on-premises to serverless environments.

Adding multiple messages to a queue using the IAsyncCollector function

In the *Saving profile picture paths to queues using queue output bindings* recipe of *Chapter 1*, *Accelerating cloud app development using Azure Functions*, 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 back-end application should be smart enough to handle the oncoming load; 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.

Let's look at a diagram that depicts the data flow from different client applications to the **Back-End Web API**.

At a given point of time, as shown in *Figure 10.1*:

- **iOS App** sends two messages.
- **Android App** sends three messages.
- **Website** sends four messages.

Each client app is sending multiple messages to the **HTTP trigger**, which could send all nine messages to **Azure Queue Storage** in a single call asynchronously:

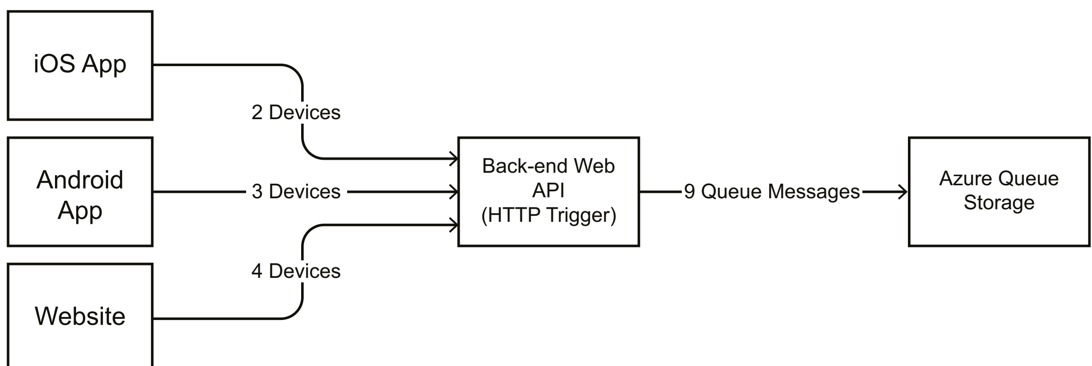


Figure 10.1: IAsyncCollector collector usage—process flow

In this recipe, we'll simulate the requests using Postman, which will send the requests to the **Back-End Web API (HTTP Trigger)**, which can create all the queue messages at once.

Getting ready

Before starting the recipe, please have the following ready to move further:

- Create a storage account using the Azure portal if you have not created one yet.
- Install Microsoft Storage Explorer from http://storageexplorer.com/ if you have not installed it yet.

How to do it...

In this section, we'll perform the following steps to create multiple messages to the queue asynchronously using the `IAsyncCollector` interface:

1. Create a new HTTP trigger named `BulkDeviceRegistrations` by setting **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 in this recipe, we'll 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, click on the **New Output** button, choose the **Azure Queue Storage** output binding, and click on the **Select** button as shown in *Figure 10.2*:

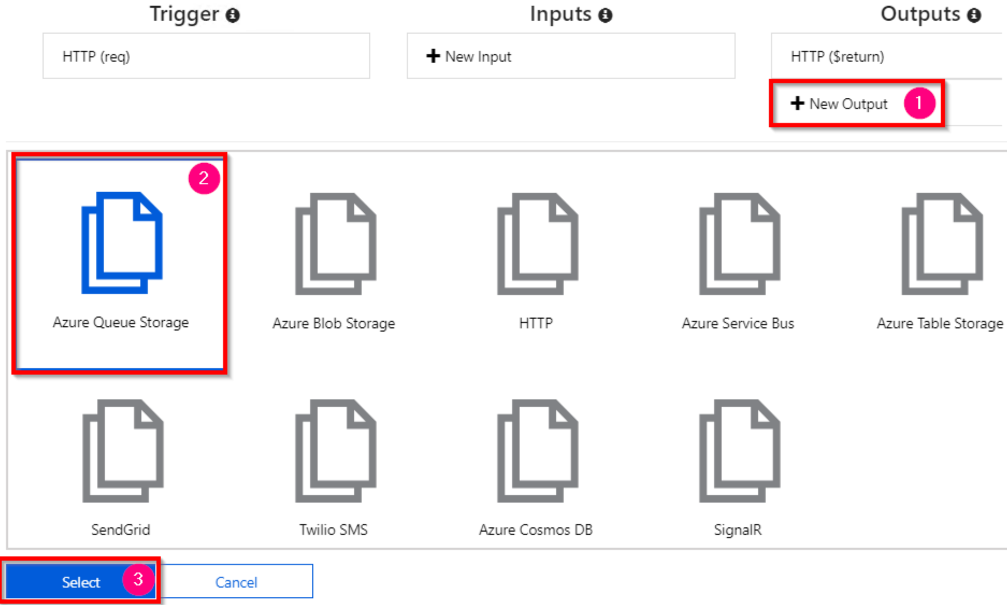


Figure 10.2: Creating a new Azure Queue Storage output binding

4. In the **Azure Queue Storage output** step, provide the values for **Message parameter name** and **Queue name**, and then choose the storage account in the **Storage account connection** dropdown, as shown in *Figure 10.3*. Click on the **Save** button to save the changes:

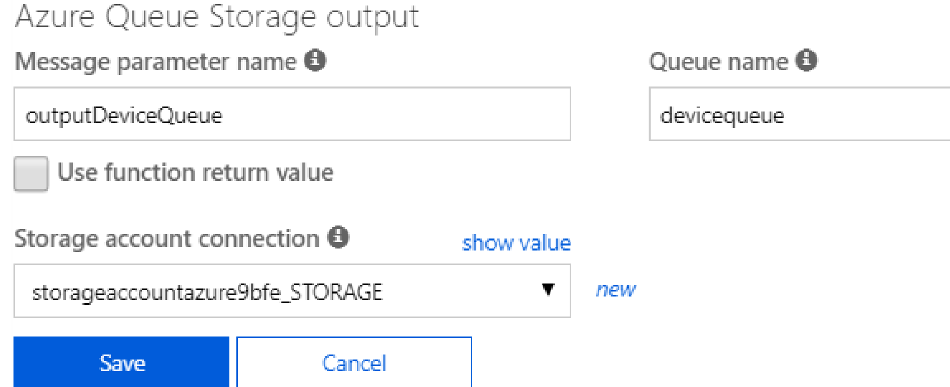


Figure 10.3: Azure Queue Storage output binding configuration

5. 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> outputDeviceQueue )
{
....
....
for(int nIndex=0;nIndex<data.devices.Count;nIndex++)
{
Device = Convert.ToString(data.devices[nIndex]); outputDeviceQueue.
AddAsync(Device);
              }
....
....
```

6. 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"
      }
    ]
}
```

7. Click on the **Run** button to test the functionality. Now open **Azure Storage Explorer** and navigate to the queue named **devicequeue**. As shown in *Figure* 10.4, we should see two records:

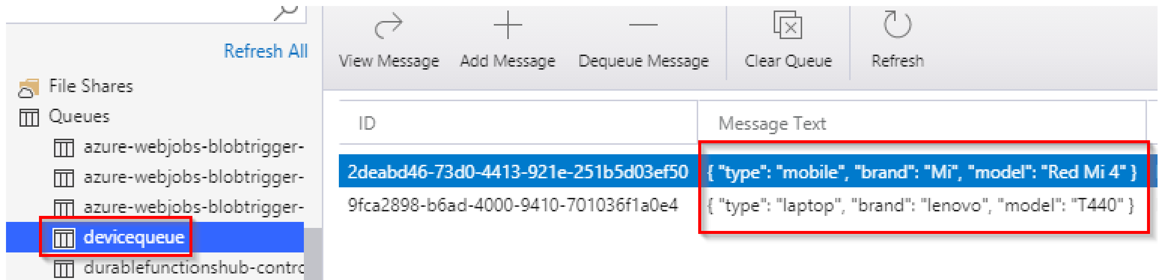


Figure 10.4: Device queue output

In this section, we have learned how to add messages to Azure Queue storage. Let's move on to the next section.

There's more...

You can also use the **ICollector** interface in place of **IAsyncCollector** if you would like to store multiple messages synchronously. These two interfaces contain the methods that can accept a collection of messages and can create them as queue messages into the queue.

In this recipe, 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 being added using Azure Storage Explorer.

Let's move on to the next recipe to understand how to implement defensive applications using Azure functions.

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 an application might fail. Developers and architects cannot predict all unexpected inputs throughout the lifespan of an application being used by business users or general users. So, it's good practice to make sure that your application alerts you if there are any errors or unexpected issues with the application.

In this recipe, we'll learn how Azure functions help us handle (and receive alerts about) errors with minimal code.

Getting ready

Before starting the recipe, please make sure you have done the following:

- Create a storage account using the Azure portal if you have not created one.
- Install Azure Storage Explorer from http://storageexplorer.com/ if you have not installed it yet.

How to do it...

In this section, we'll perform the following steps:

1. Develop a console application using C# that connects to the storage account and creates queue messages in the queue named **myqueuemessages**.
2. 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 to create messages in the queue using the console application:

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.
2. Install the **Configuration** and **Queue Storage NuGet** packages using the following commands:

```
Install-Package Microsoft.Azure.Storage.Queue
Install-Package System.Configuration.ConfigurationManager
Install-Package Microsoft.Extensions.Configuration
Install-Package Microsoft.Extensions.Configuration.Json
```

3. Add the following namespaces to the **program.cs** file:

```
using Microsoft.Azure.Storage;
using Microsoft.Azure.Storage.Queue;
using System.Configuration;
using Microsoft.Extensions.Configuration;
using System.IO;
```

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()
{
var builder = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory()).
AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
            IConfigurationRoot configuration = builder.Build();
    CloudStorageAccount storageAccount =
    CloudStorageAccount.Parse(configuration.
    GetConnectionString("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);
    }
}
```

We are done with the console application that creates the messages. We'll move on to the next section.

Developing the Azure function–queue trigger

In this section, we'll learn how to develop a queue trigger to read the messages created in the previous section. Perform the following steps:

1. Create a new Azure function named **ProcessData** using the queue trigger template and provide **myqueuemessages** as the **Queue name**. This is how the **Integrate** tab should look after you have created the function:

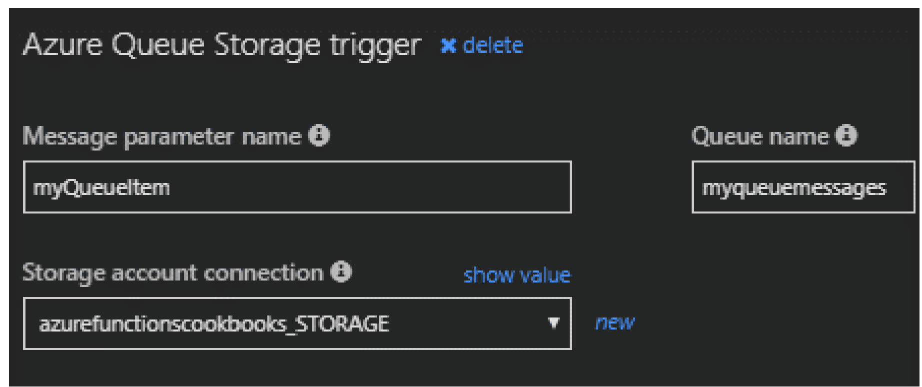


Figure 10.5: Azure Queue Storage output binding—configuration

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 all the messages whose content is greater than **50**.

Let's now run the console application that we built in the previous section.

Running tests using the CreateQueueMessage console application

In this section, we'll test the functionality using 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.

> **Note**
>
> Don't be surprised if the messages in **myqueuemessage** are vanishing. It's expected that as soon as a message is read successfully, the message is locked from the queue.

3. As shown here, you should also see a new (**poison**) 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:

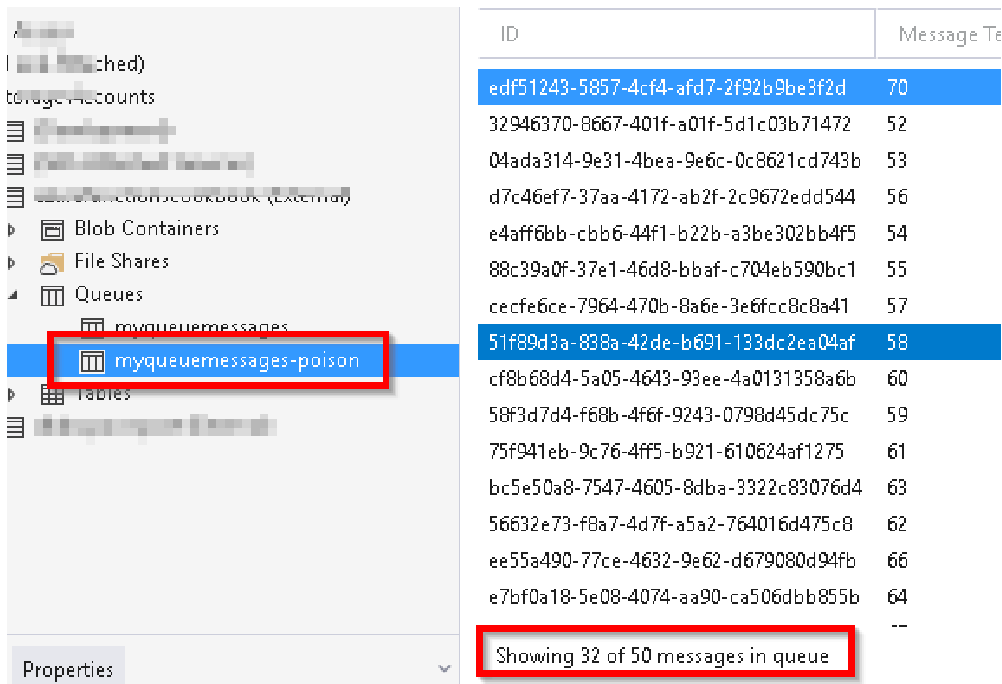


Figure 10.6: Storage Explorer—poison messages in Queue storage

So, in this section, we have learned that the Azure function runtime will ensure that unprocessed messages are automatically stored in the poison queue. It's the developer's responsibility to process messages from the poison queues as well.

There's more...

Before pushing a queue message to the poison queue, the Azure function runtime tries to pick the message and process it 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.

We have created a console application that creates messages in 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 throw an error if the value in the queue message content is greater than **50**.

Azure functions will take care of creating a new (**poison**) 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 the necessary actions to fix errors in their applications.

> **Note**
>
> The Azure function runtime will take care of deleting the queue message after the 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.

In this recipe, we have learned how Azure Functions automatically understands if there are any errors while processing the queue messages. If there is any problem in processing the message, then it creates that message in the poison queue. Let's move on to the next recipe.

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 three hosting plans:

- App Service plan
- Consumption plan
- Premium plan

One of the benefits of being serverless is the fact that you are charged based on the number of executions. This benefit is available 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 requests when there have been no requests for quite some time. To learn more about this topic, go to azure.microsoft.com/blog/understanding-serverless-cold-start/?ref=msdn.

> **Note**
>
> The Premium plan and App Service plan have a dedicated instance reserved for us and they can always be warm even if there are no requests for quite a while. Having a dedicated instance always running can be expensive at times.

In this recipe, we'll learn a technique that can be used to always keep your instance live and warm so that all requests are served properly.

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 five minutes and makes an HTTP request to the **HttpAlive** HTTP trigger

If we have clearly understood what a cold start is, then it will be clear that there will be no concerns if our application has traffic regularly during the day. So, if we can ensure that our application has traffic all day, then the Azure Functions instance will not be deprovisioned and so there won't be any concerns about the Consumption plan.

How to do it...

In this recipe, we'll 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 always be in the provisioned state.

Creating an HTTP trigger

Create a new HTTP trigger named **`HttpAlive`** and replace the default code with the following code, which 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");

}
```

We have created a simple HTTP trigger. Let's move on to the next section to create a timer trigger.

Creating a timer trigger

Create a timer trigger named **KeepFunctionAppWarm** that runs every five minutes and makes an HTTP request to the **HttpAlive** HTTP trigger by performing 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 five 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/api/HttpALive");
}
}
```

In this recipe, we have learned how to overcome the cold-starts limitation of Azure functions. Let's move on to the next recipe to learn how to share code across the Azure functions.

Sharing code across Azure functions using class libraries

Let's say that we have developed a common library across various applications being used in our project, such as a web app or a **Windows Presentation Foundation** (**WPF**) application, and now we would like to re-use some functionality in an Azure function app. It's definitely possible to re-use it. In this recipe, we'll develop and create a new `.dll` file and we'll learn how to use the classes and their methods in Azure functions.

How to do it...

Let's create a class library by performing the following steps:

1. Create a new **Class Library** application using Visual Studio as shown in *Figure 10.7*:

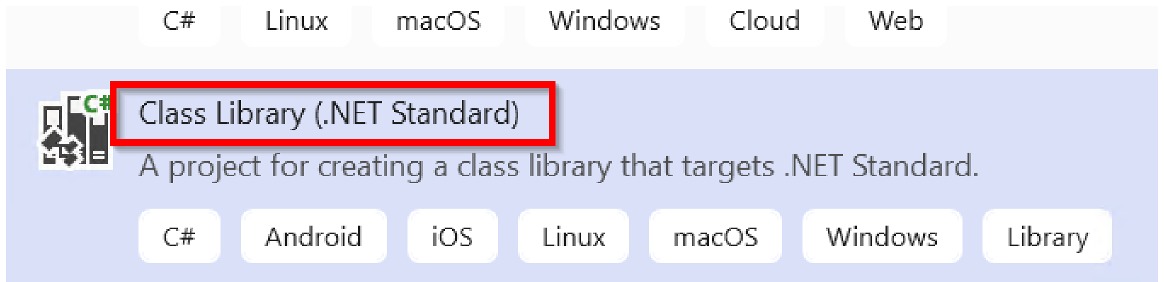


Figure 10.7: Visual Studio—creating a class library project

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 Reusable 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 (in which you would like to use the library) by clicking on the **App Service Editor** button, which is available under the **Development tools** section of the **Platform Features** tab.
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 we'll 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 (in my case the `.dll` name was `Reusability.dll`, which might change in your case depending on the name of the project that you provide for the class library) to the `bin` folder:

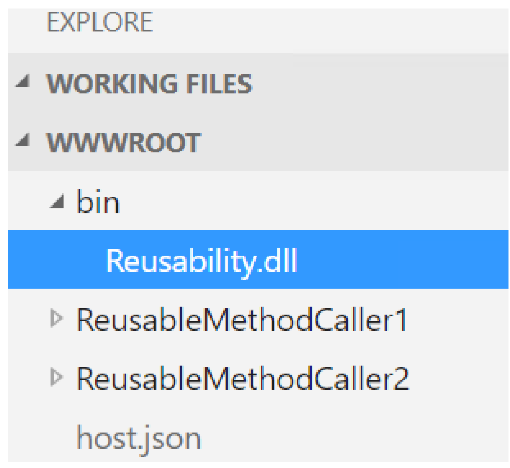


Figure 10.8: Azure Function app—App Service editor

9. Navigate to the Azure function in which you would like to use the shared method. To demonstrate, I have created two Azure functions (one HTTP trigger and one timer trigger):

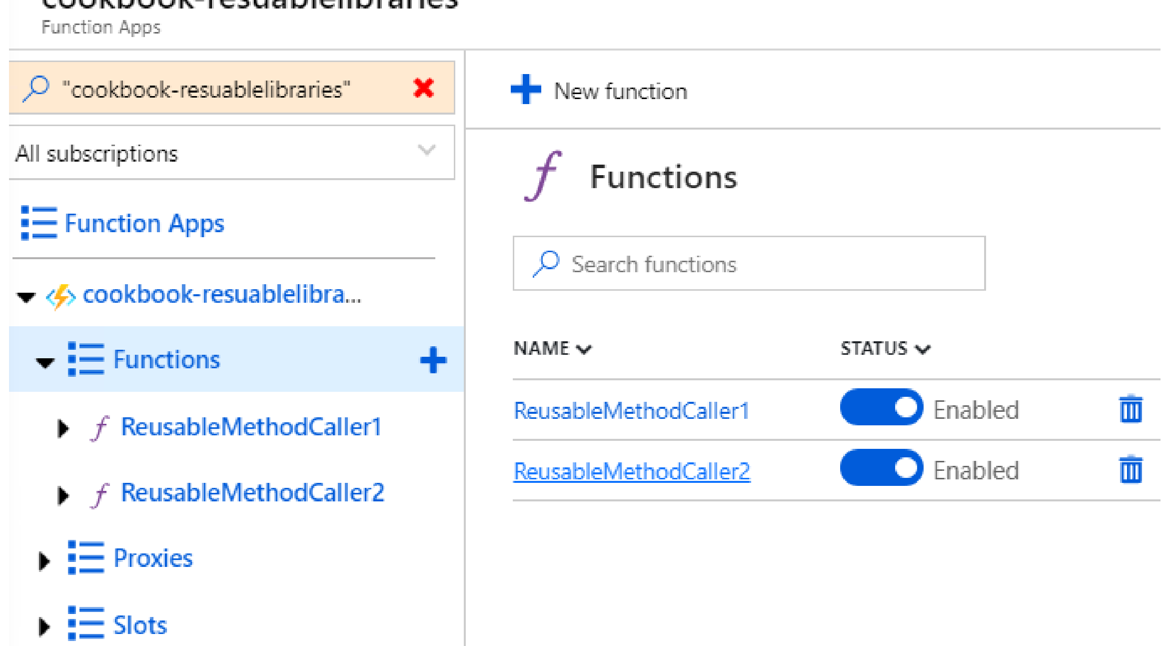


Figure 10.9: Function app—Functions list

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/Reusability.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:

```
#r "../bin/Reusability.dll"
using Utilities;
public static async Task Run(HttpRequest req, ILogger log)
{
   log.LogInformation(Helper.GetReusableFunctionOutput());
}
```

12. When you run the application, you should see the following message in the logs:

```
2020-04-19T06:27:27.081 [Information] Executing 'Functions.ReusableMethodCaller1' (Reason='This function was
programmatically called via the host
2020-04-19T06:27:27.298 [Information] This is an output from a Resuable Library across functions
2020-04-19T06:27:27.301 [Information]
4b82-a5b2-26829a691384)
```

Figure 10.10: Azure Functions—console logs

13. Repeat the same steps of adding the reference and the namespace of the **utilities** library for the second Azure function, **ReusableMethodCaller2**. If you have made the changes successfully, you should see something like what follows:

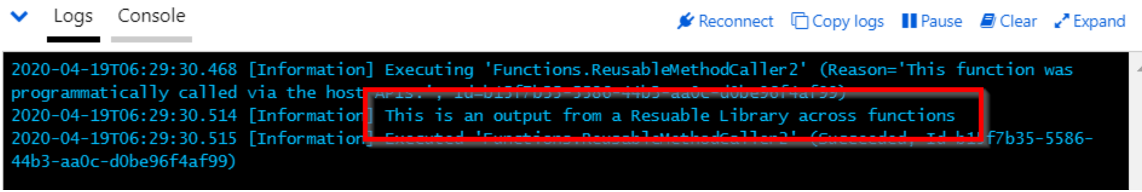


Figure 10.11: Azure functions—console logs

We have learned how to create and consume a reusable class library in Azure functions.

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.

> **Note**
>
> Another major advantage of using class libraries is that it improves performance, as they are already compiled and ready for execution.

We have created a `.dll` file that contains reusable code and can be used in any Azure function that requires 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**
>
> We have added the **`bin`** folder to the **WWWROOT** so that it is available to all the Azure Functions available in the function app.

In this recipe, we have learned how to reuse an existing component in our Azure Functions. Let's move on to the next recipe.

Migrating C# console application to Azure functions using PowerShell

Currently, many business applications are being hosted in private clouds or on-premises datacenters. Some of them have started migrating their applications to Azure using various methods.

The following are just a few methods for quick migration to Azure:

- **Lift and shift the legacy application to the Infrastructure as a Service (IaaS) environment**: This method should be straightforward, as you have complete control over the virtual machines that you create. You could host all your web applications, schedulers, databases, and so on without making any changes to your application code. You can even install any third-party software or libraries. Though this option provides full control for your application, it would be expensive in most cases as the background application might not be running all the time.
- **Convert legacy applications to a Platform as a Service (PaaS)–compatible environment**: This method could be complex, depending on how many dependencies your applications have on other third-party libraries that are not compatible with the Azure PaaS environment. You would need to make code changes to your applications so that they are stateless and are not dependent on any of the resources of the instances where they are hosted. This option is very cost-effective as you just need to pay for the execution time of your applications.

In this recipe, we'll look at one of the easiest ways of migrating our existing background job applications developed using C# classes and console applications without making many changes to the existing application code.

Getting ready

The code provided in the recipe works well with any of the previous versions of Visual Studio. Let's use the latest version of Visual Studio 2019.

How to do it...

In this recipe, we'll do the following to migrate an existing background job to Azure Functions timer triggers using PowerShell:

- Create a .NET Framework–based application to simulate a background job.
- Create a timer trigger to execute the console application on a certain frequency.

Let's start with the development of the console application.

Developing a console application

In this section, we'll create a background job using a console application. Let's follow these steps:

1. Create a new .NET Framework console application and name it **`BackgroundJob`** using Visual Studio.
2. In the **`BackgroundJob`** project, create a new class called **`UserRegistration`** and replace the default code with the following code:

```
using System;
namespace BackgroundJob
{
    class UserRegistration
    {
        public static void RegisterUser()
        {
            Console.WriteLine("Register User method of
            UserRegistration has been called.");
        }
    }
}
```

3. Create a new class called **OrderProcessing** and replace the default code with the following code:

```
using System;
namespace BackgroundJob
{
    class OrderProcessing
    {
        public static void ProcessOrder()
        {
            Console.WriteLine("Process Order method of
            OrderProcessing class has been called");
        }
    }
}
```

4. In the **Program.cs** file, replace the existing code with the following code:

```
using System;
namespace BackgroundJob
{
        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Main method execution has
                been started");
                Console.WriteLine
                ("======================================");
                UserRegistration.RegisterUser();
                OrderProcessing.ProcessOrder();
                Console.WriteLine
                ("======================================");
                Console.WriteLine("Main method execution
                has been completed");
            }
        }
}
```

Build the application to create the **.exe** file. You can configure it to run in either debug or release mode. It is recommended that you deploy the **.exe** file in the release mode in your production environments. In this section, we have created a console application. Let's move on to the next section:

1. Create a new **function app** using **Power Shell 6** by choosing **Power Shell Core** in the **Runtime stack** dropdown while creating the function app, as shown in *Figure 10.12*:

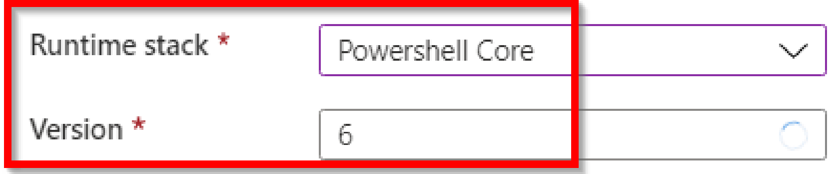


Figure 10.12: Azure Function app—runtime stack and PowerShell version

2. Once the function app is created, navigate to the **Functions** blade and click on **Add**, as shown in *Figure 10.13*:

Figure 10.13: Adding a new Azure function

3. In the **New function** blade, choose the **Timer trigger** template, as shown in *Figure 10.14*:

New Function

Create a new function in this Function App. Start by selecting a template below.

Templates Details

Search by template name

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

Timer trigger
A function that will be run on a specified schedule

Figure 10.14: Selecting the Timer trigger template

4. In the **Details** view, provide the name and the schedule and click on **Create function**, as shown in *Figure 10.15*, to create the timer trigger function:

New Function

Create a new function in this Function App. Start by selecting a template below.

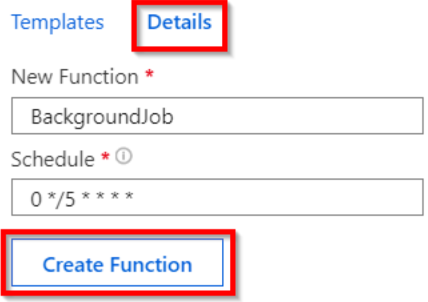


Figure 10.15: Providing Azure timer trigger details

5. Now, we need to upload the console application executable files to **Azure function timer trigger**. We can upload using the **App Service Editor**. Click on the **App Service Editor**, as shown in *Figure 10.16*:

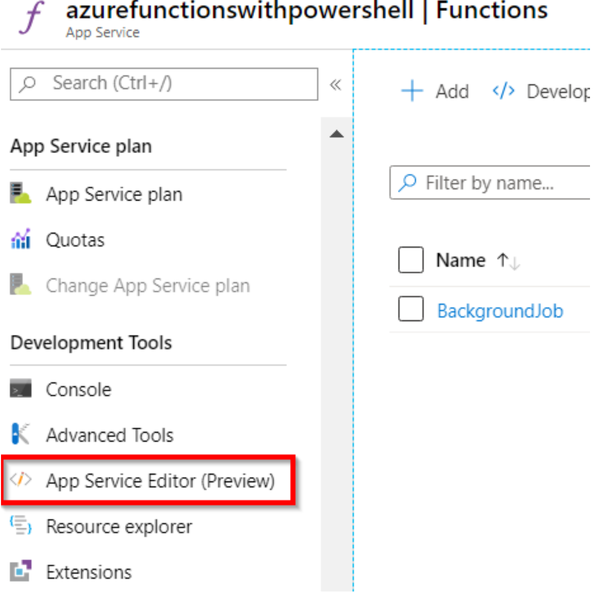


Figure 10.16: Azure Functions—App Service Editor

6. Clicking on the **Go** button on the next page will open up a new browser tab where you can see the **App Service Editor**. As shown in *Figure 10.17*, right-click on the **BackgroundJob** folder and create a **New Folder** named **bin**:

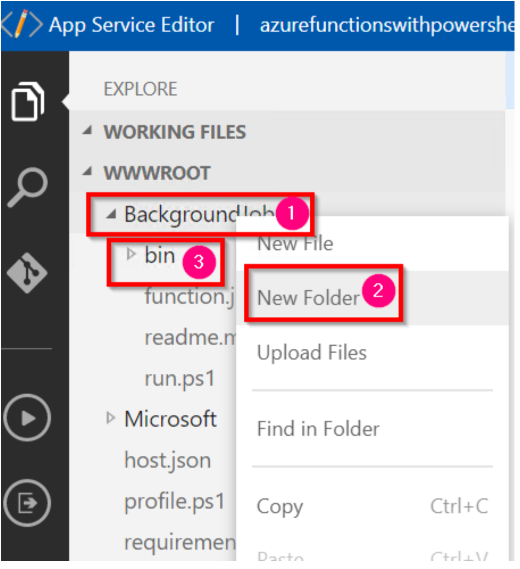


Figure 10.17: Azure Functions—App Service Editor—New Folder

7. Now, let's upload the **.exe** file along with any other dependencies, if any. In this recipe, we just have the **.exe** file, as shown in *Figure 10.18*:

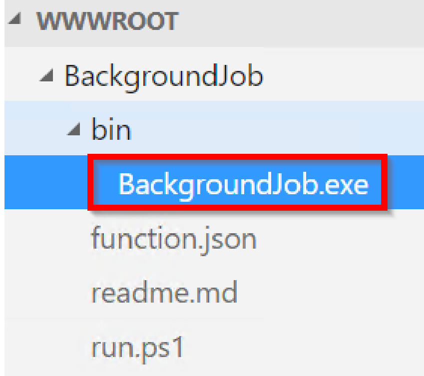


Figure 10.18: Azure Functions—App Service Editor—uploading the .exe file

8. Let's navigate to the timer trigger's **Code / Test** window and add the code to invoke the **BackgroundJob.exe**, as shown in *Figure 10.19*. In the following code, we are first setting the path of the executable folder and then running the **.exe** file:

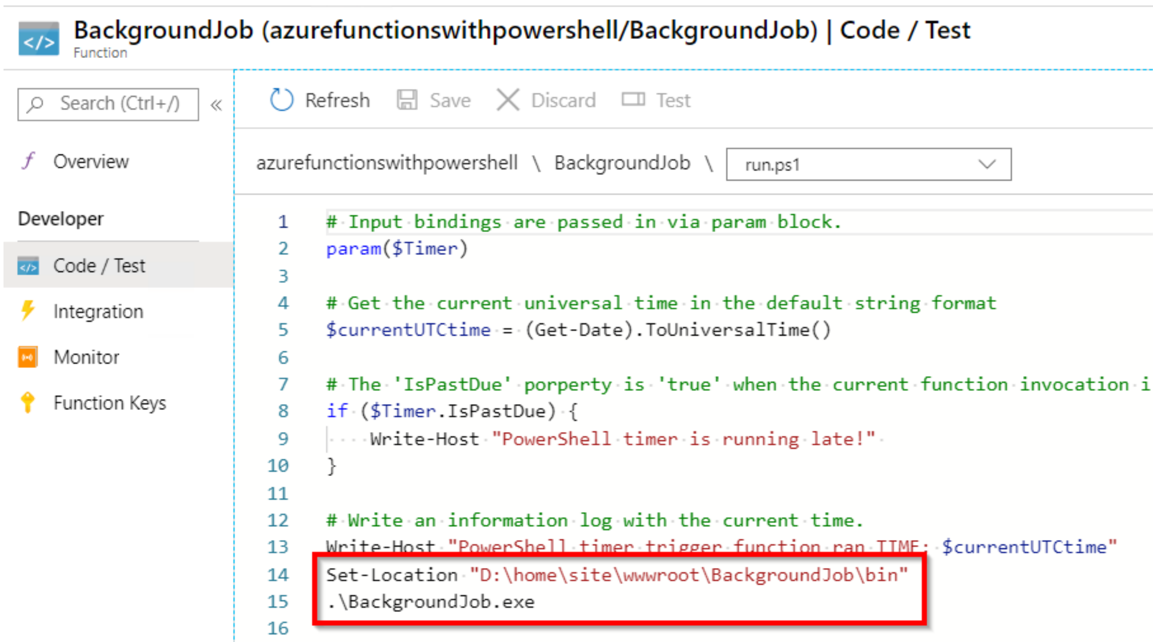


Figure 10.19: Azure Functions—invoking .exe using PowerShell

9. That's it. From now on, the timer trigger runs every five minutes. *Figure* 10.20 shows the output of one of its executions:

```
Logs   Console
SCM_LOGSTREAM_TIMEOUT (in seconds).
2020-04-12T08:49:00  No new trace in the past 1 min(s).
2020-04-12T08:50:01  No new trace in the past 2 min(s).
2020-04-12T08:49:59.995 [Information] Executing 'Functions.BackgroundJob' (Reason='Timer fired at 2020-04-12T
4e94-9c87-093aee881e89)
2020-04-12T08:50:01.176 [Information] INFORMATION: PowerShell timer trigger function ran TIME: 04/12/2020 08:
2020-04-12T08:50:02.292 [Information] OUTPUT: Main method execution has been started
2020-04-12T08:50:02.293 [Information] OUTPUT: ======================================
2020-04-12T08:50:02.502 [Information] OUTPUT: Register User method of UserRegistration has been called.
2020-04-12T08:50:02.504 [Information] OUTPUT: Process Order method of OrderProcessing class has been called
2020-04-12T08:50:02.504 [Information] OUTPUT: ======================================
2020-04-12T08:50:02.504 [Information] OUTPUT: Main method execution has been completed
2020-04-12T08:50:02.504 [Information] Executed 'Functions.BackgroundJob' (Succeeded, Id=e059a9b5-6c16-4e94-9c
```

Figure 10.20: Azure Functions—console logs

In this recipe, we have learned how to migrate a .NET Framework–based console application to Azure Functions using timer triggers that run every five minutes.

Implementing feature flags in Azure functions using App Configuration

Usually, when we are working on enterprise projects, we are working on multiple large applications where we have individual app settings stores for every application. The app settings would be either specific to an application or common across all applications.

For example, if we have one database that is used by multiple applications, then we have to have the same connection string in each of those applications. If we have to change something (such as a password) in the connection string, we would need to change it in all the configurations of all the projects.

In order to solve this problem, Azure provides a service called App Configuration, which can be used to externalize configuration items. When we take configurations out of the scope of the individual project, we can use them in multiple applications.

In this recipe, we'll learn how to do the following:

- Externalize app configurations.
- Manage functionality dynamically without code deployment.

Getting ready

Please create an Azure function app if you have not yet done so.

How to do it...

In this recipe, we'll do the following:

- Create the App Configuration service.
- Create a configuration key and feature management keys.
- Develop an Azure function HTTP trigger to control the application features using feature flags.

Let's start creating the App Configuration service.

Create the App Configuration service

In this section, we'll create the **App Configuration** service to externalize our application configurations to reduce the downtime required and also have a one-stop solution to store all the common settings related to multiple applications. Please follow these steps:

1. Navigate to the Azure portal, click on **Create a resource**, search for **App Configuration**, and click on the **Create** button, as shown in *Figure 10.21*:

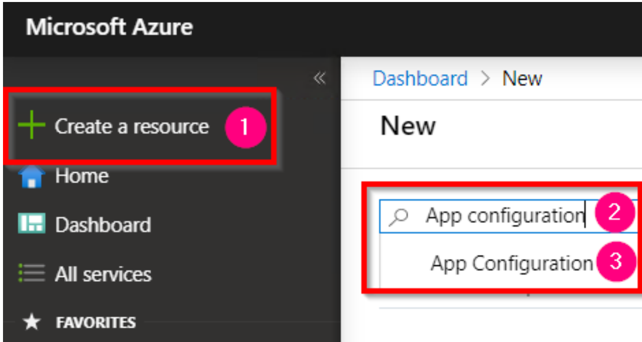


Figure 10.21: Searching for App Configuration

2. In the **App Configuration** blade, provide a name, choose a **Pricing tier**, and click on the **Create** button, as shown in *Figure 10.22*:

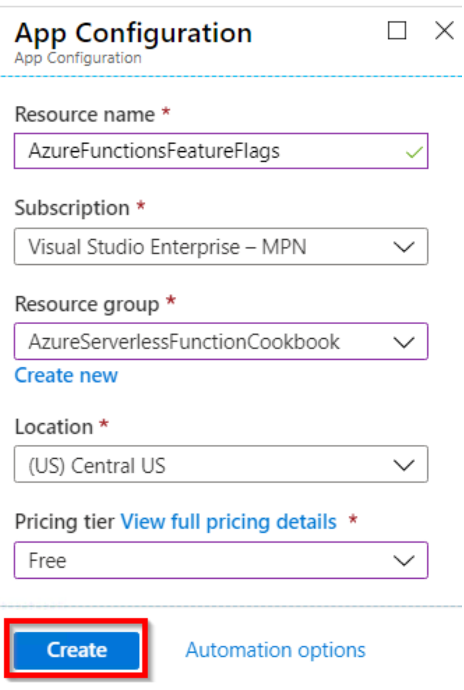


Figure 10.22: Creating a new App Configuration

3. Clicking on the **Create** button will create a new **App Configuration** service as shown in *Figure 10.23*:

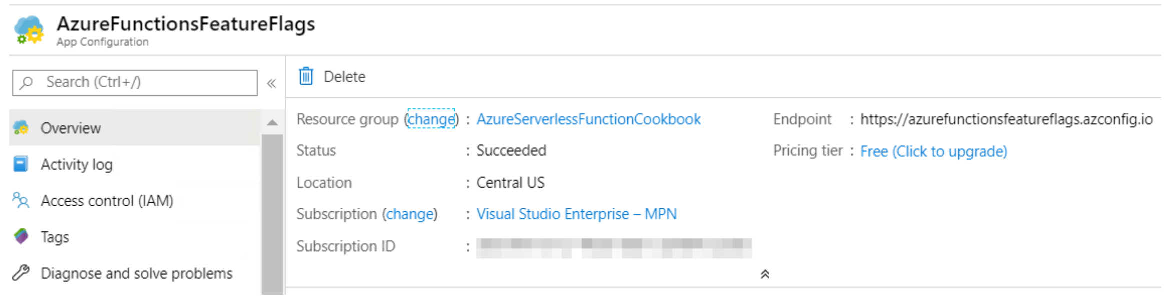


Figure 10.23: App Configuration—Overview blade

We have created the **App Configuration** service. In the next section, we'll learn about creating the configuration key (a key-value pair) and feature management keys.

Creating a configuration key and feature management keys

In this section, we'll create a **key-value** pair using **Configuration explorer** and also create feature flags:

1. Navigate to the **Configuration explorer** blade and click on the **Key-value** button, which is available under the **Create** button, as shown in *Figure 10.24*:

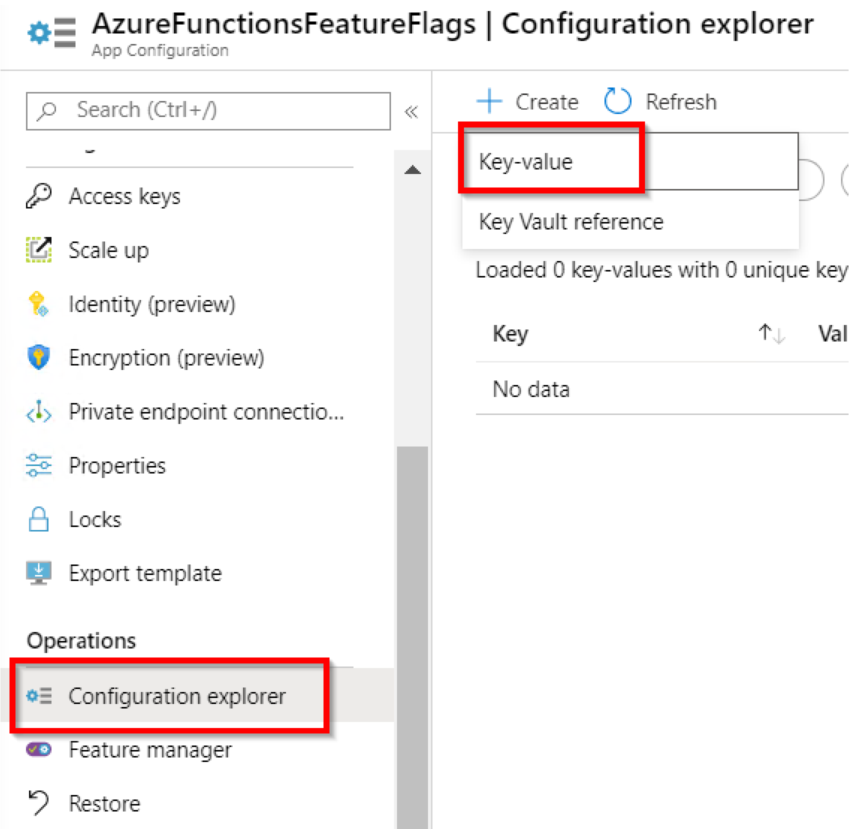


Figure 10.24: App Configuration—Configuration explorer

2. Clicking on **Key-value** will open up a new blade where you can create a key-value pair, as shown in *Figure 10.25*:

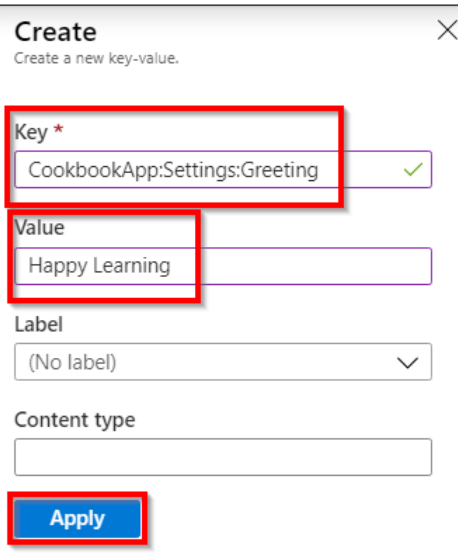


Figure 10.25: App Configuration—creating a new key-value pair

3. When you click on **Apply** in *Figure 10.25*, a key-value pair will be created, as shown in *Figure 10.26*:

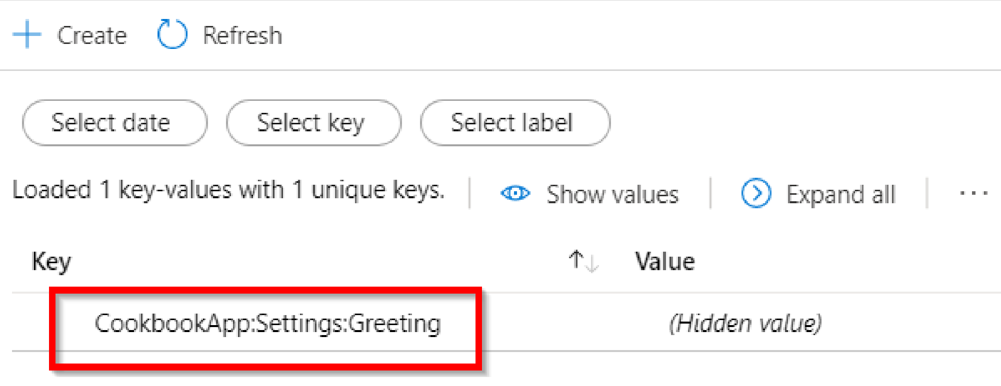


Figure 10.26: App Configuration—list of key-value pairs

4. Navigate to the **Feature manager** blade and click on **Add**, as shown in *Figure 10.27*:

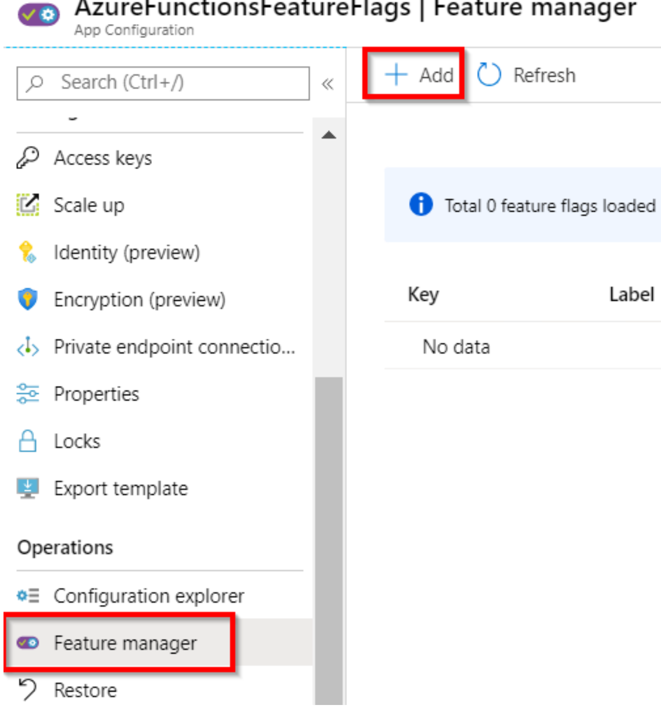


Figure 10.27: App Configuration—Feature manager

5. That opens up a blade where you can add a new feature flag. Select **On** and provide a key, as shown in *Figure 10.28*:

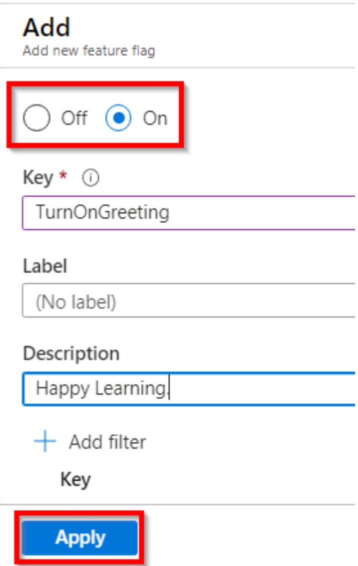


Figure 10.28: App Configuration—Feature manager—adding a new feature flag

6. Once you click on **Apply** in *Figure 10.28*, it will create a feature flag as shown in *Figure 10.29*:

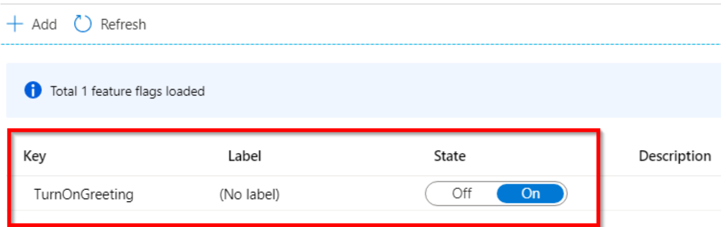


Figure 10.29: App Configuration—Feature manager —list of feature flags

7. Note that the feature flags are also configuration items. So, the **`TurnOnGreeting`** feature flag is also shown in the **Configuration explorer**, as shown in *Figure 10.30*:

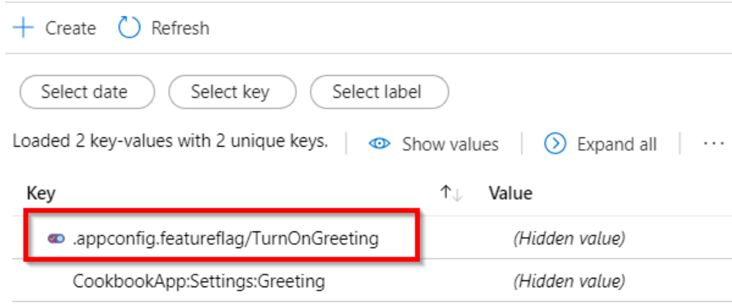


Figure 10.30: App Configuration—list of configurations

In this section, we have created configuration keys and feature flags. Let's move on to the next section.

Developing an Azure function HTTP trigger to control the application features using feature flags

In this section, we'll develop an Azure function HTTP trigger and learn how to use these configuration keys and feature flags.

In this section, we'll do the following:

1. Develop the HTTP trigger.
2. Load the feature flags and key-value pairs from App Configuration using **`Startup`**.
3. Inject the feature flags and key-value pairs using dependency injection.
4. Access the feature flags and key-value pairs in the HTTP trigger.

Developing the HTTP trigger

In this section, we'll develop a function app named **FeatureFlags**, create an HTTP trigger, and configure the connection string of the **App Configuration** service:

1. Open Visual Studio and create an HTTP trigger with the name **`DisplayGreeting`**. Please make the class non-static as we will be passing parameters to it later.
2. Install the following NuGet packages:

```
Install-Package Microsoft.Extensions.Configuration.AzureAppConfiguration
Install-Package Microsoft.FeatureManagement
Install-Package Microsoft.Azure.Functions.Extensions
```

3. Navigate to **App Configuration** and copy the **Connection string** from the **Read-only keys** tab available in the **Access keys** blade, as shown in *Figure 10.31*:

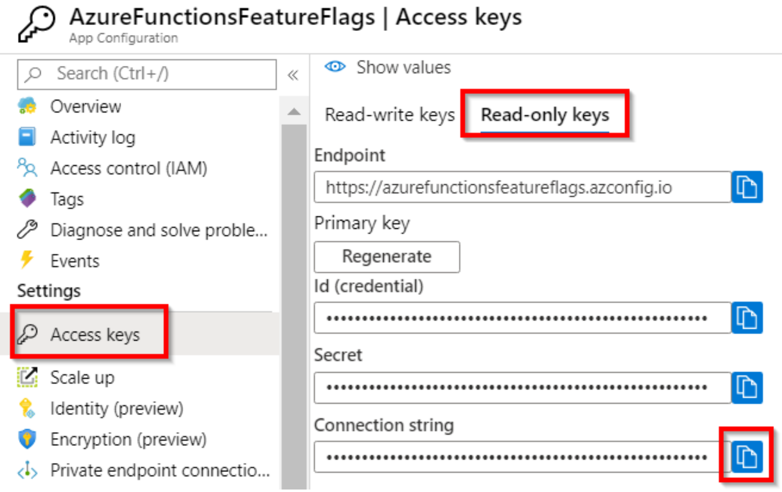


Figure 10.31: App Configuration—read-only access keys

4. Open the **`local.settings.json`** configuration file, create a connection string named **`AppConfigurationConnectionString`**, and paste the connection string. Once you configure the connection string, it should look something like *Figure 10.32*:

```
local.settings.json
Schema: <No Schema Selected>
1 {
2     "IsEncrypted": false,
3   "Values": {
4     "FUNCTIONS_WORKER_RUNTIME": "dotnet",
5     "AppConfigurationConnectionString": "Endpoint=https://azu
       s0:071JN09PMK1S
6   }
7 }
```

Figure 10.32: Visual Studio—local configuration file—creating the App Configuration connection string

Let's move on to the next section to develop the **Startup** class, which can be used to load the configurations.

Loading feature flags and key-value pairs from App Configuration using Startup

In this section, we'll develop the **Startup** class, which is used to connect to the App Configuration service and load the configurations. You can learn more about it at docs.microsoft.com/azure/azure-functions/functions-dotnet-dependency-injection:

1. Create a new class named **Startup** and replace the default code with the following code. This code connects to the App Configuration service and loads both the feature flags as well as the configuration key-value pairs:

```
using System;
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.FeatureManagement;

[assembly: FunctionsStartup(typeof(FeatureFlags.Startup))]

namespace FeatureFlags
{
    class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            ConfigurationBuilder configurationBuilder = new
ConfigurationBuilder();
            configurationBuilder.AddAzureAppConfiguration(options =>
            {
                options.Connect(Environment.
GetEnvironmentVariable("AppConfigurationConnectionString"))
                       .UseFeatureFlags();
            });

            IConfiguration configuration = configurationBuilder.Build();
            builder.Services.Configure<Settings>(configuration.
GetSection("CookbookApp:Settings"));
            builder.Services.AddFeatureManagement(configuration);
        }
    }
}
```

> **Note**
>
> In your projects, you might have multiple config items. You will need to create a separate property in this **Settings** class for each of the config items. It's called the options pattern. Learn more about it at docs.microsoft.com/aspnet/core/fundamentals/configuration/options?view=aspnetcore-3.1.

2. Add a new class named **Settings** and paste the following code. This class has only one property, named **Greeting**, as we have only one key-value pair in our App Configuration.

```
namespace FeatureFlags
{
    public class Settings
    {
        public string Greeting { get; set; }
    }
}
```

We have developed the **Startup** class. Let's move on to the next section to inject the feature flags into the Azure function HTTP trigger.

Injecting the feature flags and key-value pairs using dependency injection

In this section, we'll learn how to inject the feature flags and key-value configurations to the HTTP trigger so that we can use them in the HTTP trigger's code:

1. Create the following variables in the **DisplayGreeting** class that we have created:

```
private readonly IFeatureManagerSnapshot _featureManagerSnapshot;
private readonly Settings _settings;
private readonly IConfiguration _configuration;
```

2. Add the following namespaces:

```
using Microsoft.FeatureManagement;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Options;
```

3. Add the constructor and inject the dependencies of **FeatureManagement** and **Configurations** as follows:

```
public DisplayGreeting(IFeatureManagerSnapshot featureManagerSnapshot,
IOptionsSnapshot<Settings> settings, IConfiguration configuration)
        {
            _featureManagerSnapshot = featureManagerSnapshot;
            _settings = settings.Value;
            _configuration = configuration;
 }
```

As we have configured everything, we can now go ahead and access the feature flags and the key-value pairs in the HTTP trigger. Let's move on to the next section to learn how to do that.

Accessing the feature flags and key-value pairs in the HTTP trigger

1. Please remove the **Static** keyword from the HTTP trigger definition.
2. In order to access the feature flag, add the following line of the code to get the status of the flag named **TurnOnGreeting**:

```
bool featureEnabled = await _featureManagerSnapshot.
IsEnabledAsync("TurnOnGreeting");
```

3. The next step is to retrieve the value of the key-value configuration named **Greeting**. We can do so by accessing **Settings** as follows:

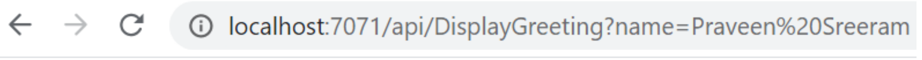
```
if (featureEnabled)
            {
                return new OkObjectResult($"Hello, {name}. {_settings.
Greeting}");
            }
            else
            {
                return new OkObjectResult($"Hello, {name}.");

    }
```

The preceding code displays the greeting **Happy Learning** only if the feature flag is turned on.

4. The following is the complete code of the HTTP trigger. Once we have verified the code, we can execute the HTTP trigger:

```
[FunctionName("DisplayGreeting")]
        public 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;

            bool featureEnabled = await _featureManagerSnapshot.
IsEnabledAsync("TurnOnGreeting");

            if (featureEnabled)
            {
                return new OkObjectResult($"Hello, {name}. {_settings.
Greeting}");
            }
            else
            {
                return new OkObjectResult($"Hello, {name}.");

            }
        }
```

5. As shown in *Figure 10.33*, you will see the greeting **`Happy Learning`** as the feature flag is turned on in App Configuration:




```
Hello, Praveen Sreeram. Happy Learning
```

Figure 10.33: Azure Functions HTTP trigger—output when the feature flag is on

6. Let's say, for some reason, you would like to turn the flag off. To do so, you navigate to **App Configuration** and set the feature flag to **Off**, as shown in *Figure 10.34*:

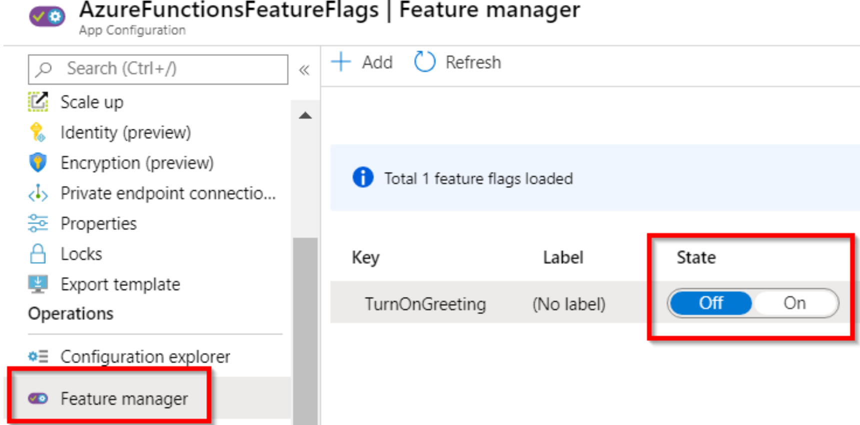
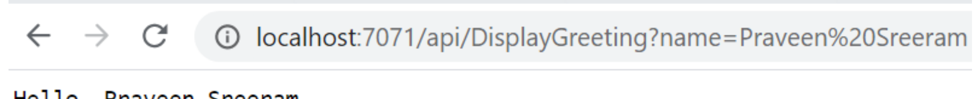


Figure 10.34: Azure Functions HTTP trigger—feature flag turned off

7. After turning it off, the next time you run and access the HTTP trigger, you will not see the greeting, as shown in *Figure 10.35*:




```
Hello, Praveen Sreeram.
```

Figure 10.35: Azure Functions HTTP trigger—output when the feature flag is off

In this section, we have learned how to leverage feature flags in Azure functions.

In this recipe, we have used only one of the few services that App Configuration provides. We can also use App Configuration to externalize our application configurations to reduce the downtime required and have a one-stop solution to store all the common settings related to multiple applications that we might be working with. Here are some other features of App Configuration:

- We can compare configurations.
- We can import and export the keys from an existing configuration file easily.
- We can reload or refresh configuration changes.

Learn more about these features at docs.microsoft.com/azure/azure-app-configuration/overview.

In this chapter, we have learned some of the best practices that help to improve the performance of our applications. We have also learned how to migrate the services from on-premises to Azure.

11

Configuring serverless applications in the production environment

In order to learn how to deploy a function application efficiently and move configurations without making any mistakes, we will be covering the following recipes in this chapter:

- Deploying Azure functions using the Run From Package feature
- Deploying Azure functions using ARM templates
- Configuring a custom domain for Azure functions
- Accessing application settings
- Breaking down large APIs into smaller subsets using proxies
- Moving configuration items from one environment to another using resources

Introduction

After spending days (or months) developing the code for your serverless applications, you then need to deploy them to Azure so that other applications can access them.

As an architect or administrator, you may encounter various challenges (depending on the requirements) in deploying or promoting your function app's project files, dependencies, and related configurable items to various environments.

This chapter focuses on the configurations that we need to make in a non-development environment (such as staging, UAT, and production).

Deploying Azure functions using the Run From Package feature

We have been learning about different techniques for developing Azure functions and deploying them to the cloud.

As you may already know, each function app can have multiple functions hosted within it. All the code related to these functions is located in the **`D:\home\site\wwwroot`** folder. We'll use the Kudu app to view the binaries.

Kudu is an open-source application that lets us deploy binaries to an App Service, view the environment variables, and view processes running on the App Service's hosts. Navigate to Kudu with the URL **`https://<<yourfunctionappname>>.scm.azurewebsites.net`**.

In *Figure 11.1*, you can see all the binaries of the Kudu web app:

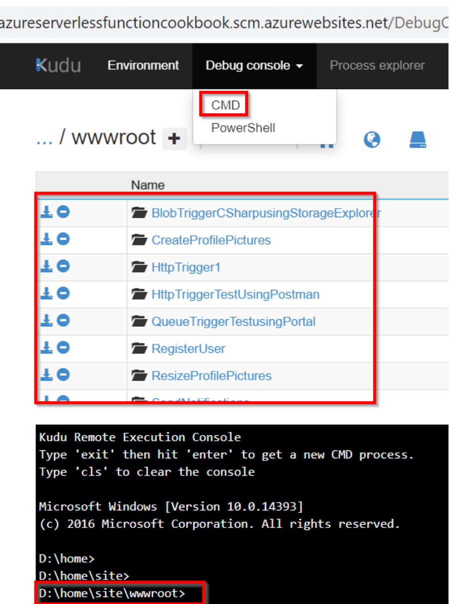


Figure 11.1: KUDU app—WWW root folder structure

`D:\home\site\wwwroot` is the location where the runtime would look for the binaries and all the configuration files that are required to execute the application.

In this recipe, we'll learn another new technique, called Run From Package (previously 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.

This **Run From Package** method definitely reduces the risk of file locks when copying files. Learn more about this method at https://docs.microsoft.com/azure/azure-functions/run-functions-from-deployment-package.

Getting ready

Perform the following steps to get ready for this recipe:

1. Create one or more Azure functions using Visual Studio. For this example, I have created one HTTP trigger and one timer trigger:

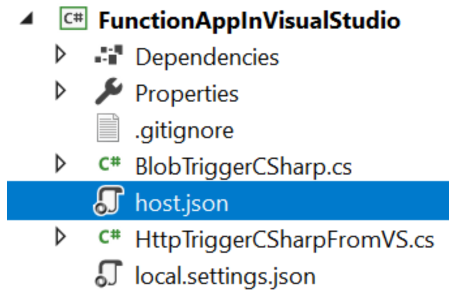


Figure 11.2: Visual Studio—function app solution explorer

2. Create an empty function app with .NET Core as the runtime stack using the Azure portal:

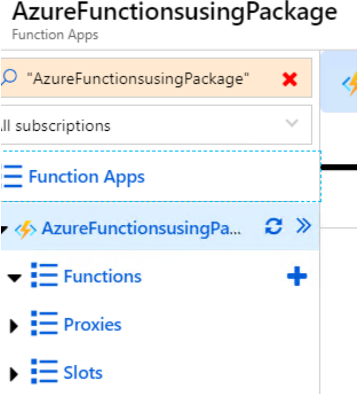


Figure 11.3: A new Azure function app in the portal

3. Create a new, or use an existing, storage account. This storage account will be used to upload the package file.

How to do it...

Perform the following steps:

1. Create a package file for the application by clicking on **Publish** and choosing a folder, as shown in *Figure 11.4*. We will make use of the same application that we created in *Chapter 4*, *Developing Azure functions using Visual Studio*:

Pick a publish target

Azure Functions Consumption Plan

Azure Functions Premium Plan

Azure App Service Plan

Azure App Service Plan Linux

Container Registry

Folder

Folder or File Share

Publish your app to a folder or file share

Choose a folder

bin\Release\netcoreapp3.0\publish\

Browse...

Import Profile...

Create Profile

Cancel

Figure 11.4: Visual Studio—picking a publish target

2. Navigate to the `bin` folder location that contains other files related to your functions. Create a `.zip` file of the files, which is highlighted in *Figure 11.5*:

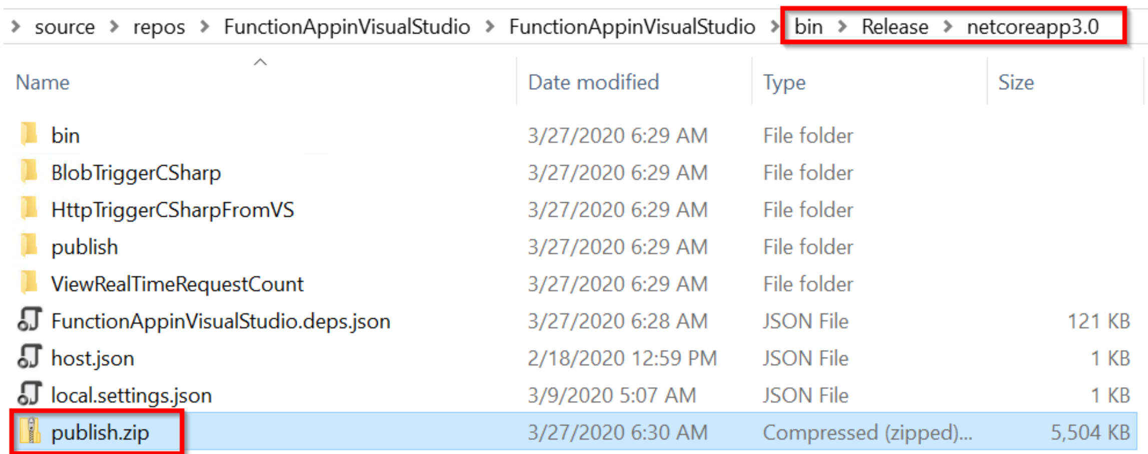


Figure 11.5: Windows Explorer—creating a .zip file from the binaries

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 with read permissions for the blob so that the Azure function runtime has the permission required to access the files located in the container. You can generate an SAS token by clicking on the **Generate SAS** button, as shown in *Figure 11.6*:

Figure 11.6: Storage blob—generating an SAS token

You can learn more about SAS at https://docs.microsoft.com/azure/storage/common/storage-sas-overview.

5. Here is the generated URL along with the SAS token:

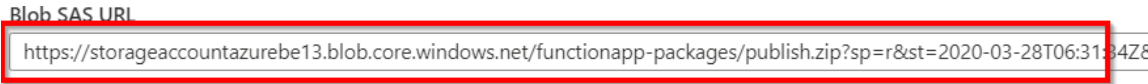


Figure 11.7: Storage blob—generated URL with an SAS token

6. Navigate to the **Configuration** pane's **Application settings** of the function app that you created. Create a new app setting with the `WEBSITE_RUN_FROM_PACKAGE` key and set the value to be the **Blob SAS URL** that you created in the previous step. Click on **Save** to save the changes:

Figure 11.8: Package location in the app settings

7. That's it! After the preceding configuration, you can test the function:

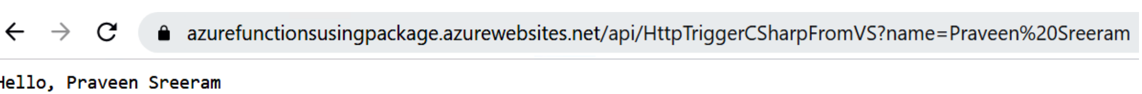


Figure 11.9: HTTP trigger—output

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 corresponding storage account. So, on the fly, the runtime downloads the files and uses them to launch the application.

In this recipe, we have learned how to deploy the Azure functions using Run From Package options. Let's now move on to the next recipe.

Deploying Azure functions using ARM templates

So far, we have been manually provisioning Azure functions using the Azure portal. Although it's easy to work with the portal, this approach has a number of disadvantages:

1. It is not easy to view the history of all the changes made to any service.
2. In large projects with hundreds of services, replicating the infrastructure across new environments is not easy (in one of my engagements, we have more than 500 services). If customers ask to create a new environment (for instance, an Alpha environment), which should be similar to our production, then it might take weeks to create.

In order to resolve these challenges, it's a best practice to automate the process of infrastructure provisioning. Azure has a solution for this in the form of **Azure Resource Manager (ARM)** templates.

ARM templates are JSON-based files where you can define the resources that you want to be created. You can add these ARM templates to source control repositories (such as Git) so that multiple team members can collaborate using them and you can view the history of changes that you or your team has made.

In this recipe, we'll learn how to automate the process of provisioning Azure functions using ARM templates.

Getting ready

Before we start authoring the ARM templates, we need to understand the other Azure services upon which the Azure function depends. The following services are automatically created when we create a function app:

- **App Service plan**: This could either be a regular App Service plan or a consumption plan.
- **Storage account**: An Azure function runtime uses a storage account to log diagnostic information that we can use for troubleshooting.

- **Application Insights**: An Application Insights account is optional. 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 the Azure Table storage service to log diagnostic information.

Along with these services, we will obviously need to have a resource group. In this recipe, we'll 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 begin by authoring the ARM template using Visual Studio:

1. Create a new project by choosing **Azure** and then **Azure Resource Group**:

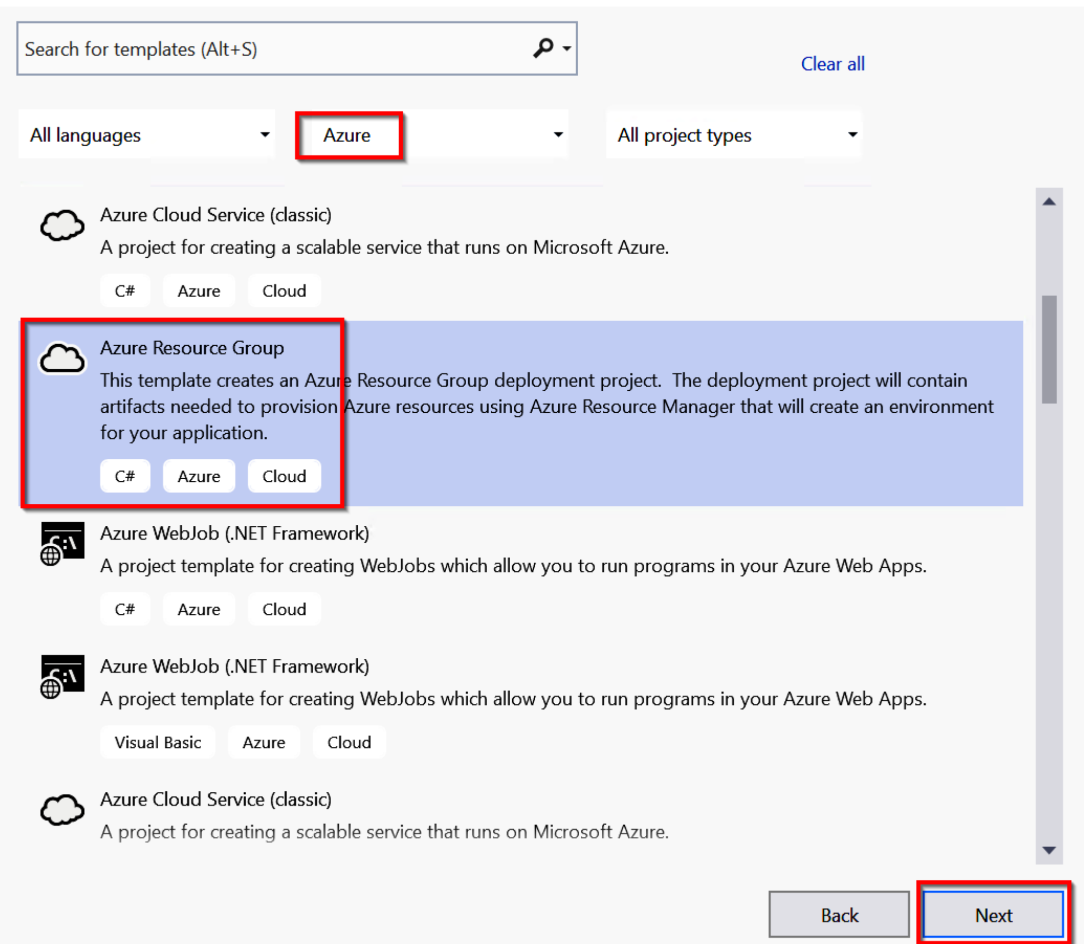


Figure 11.10: Visual Studio—creating a new Azure Resource Group project

2. Clicking on the **Next** button in the previous step will open up the **Configure your new project** pane, where you can provide a name for your project. Provide a meaningful name for the project and click on the **Create** button to create it. In the **Select Azure Template** step, choose the **Azure QuickStart (github.com/Azure/azure-quickstart-templates)** template:

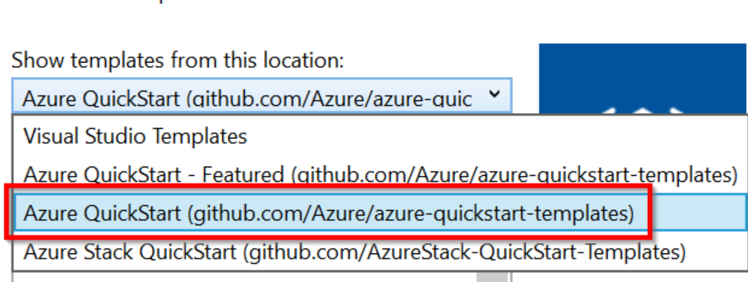


Figure 11.11: Visual Studio—selecting Azure Quickstart templates from GitHub

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:

Figure 11.12: Visual Studio—selecting ARM templates from GitHub

4. The required JSON template will be created in Visual Studio. Learn more about the JSON content at https://docs.microsoft.com/azure/azure-functions/functions-infrastructure-as-code.

5. Deploy the ARM to provision the function app and its dependent resources. You can deploy it by right-clicking on the project name (in my case, `FunctionAppusingARMTemplate`), clicking on **Deploy**, and then clicking on the **New** button:

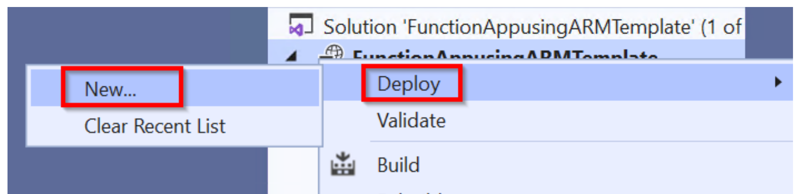


Figure 11.13: Visual Studio Azure Resource Group—new deployment

6. Choose **Subscription**, **Resource group**, and other parameters to provision the function app. Choose all the mandatory fields and click on the **Deploy** button:

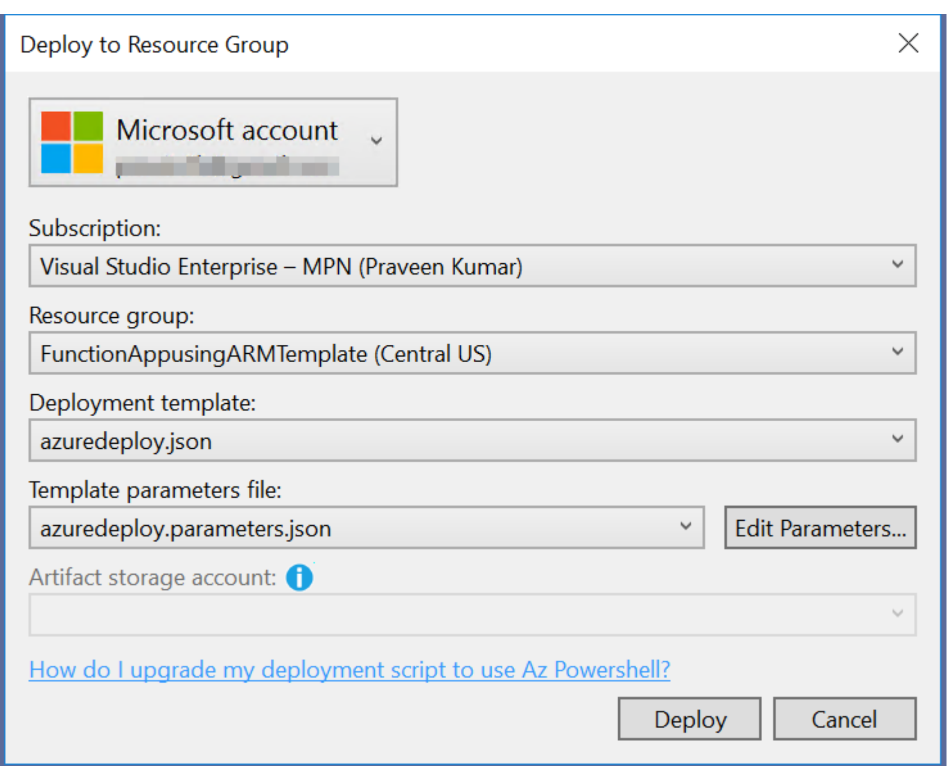


Figure 11.14: Visual Studio—Azure Resource Group—new deployment

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:

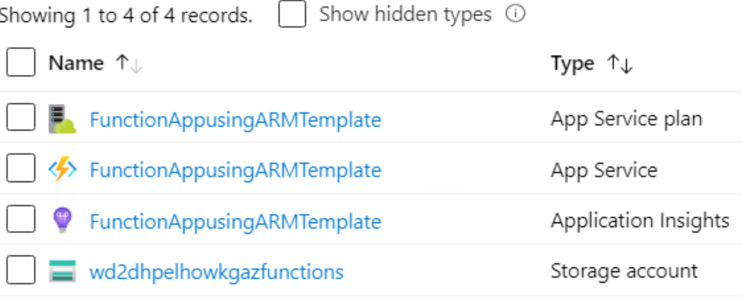


Figure 11.15: Azure portal—resources in the resource group

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 to 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 quickly.
- With the ARM templates, we can automate the process of provisioning the infrastructure to multiple environments using **Continuous Integration/ Continuous Deployment** (**CI/CD**) pipelines

In this recipe, we have learned how to automate the process of creating an Azure function using ARM templates. Let's now move on to the next recipe.

Configuring a custom domain for Azure functions

Looking at the default URL in the `functionappname.azurewebsites.net` format of the Azure function app, you may be wondering whether it's possible to have a separate domain instead of the default domain, as customers might have their own domains. Yes–it's possible to configure a custom domain for function apps. Let's learn how to do that in this recipe.

Getting ready

Create a domain with any of the domain registrars. You can also purchase a domain from the portal directly using the **Buy Domain** button, which is available in the **Custom Domains** pane:

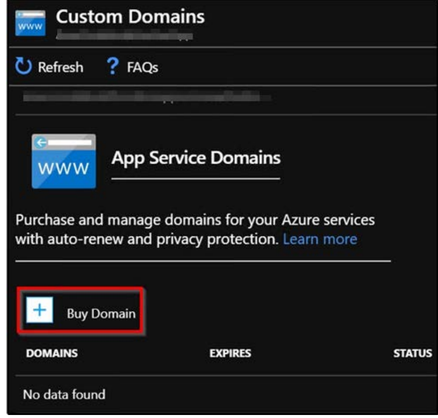


Figure 11.16: Azure Functions—purchasing a new domain

Once the domain is ready, create the following DNS records using the domain registrar:

- A record
- A CName record

How to do it...

In this section, we'll configure the custom domain for the Azure function app by performing the following steps:

1. Navigate to the **Custom Domains** pane of the Azure function app for which you would like to configure a domain and make a note of the **IP address** along with the default URL of the Azure function app, as shown in *Figure 11.17*:

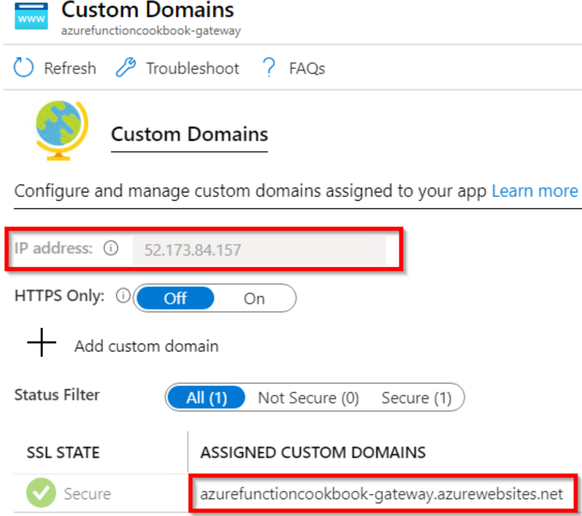


Figure 11.17: Azure Functions—custom domain details

2. Navigate to the **App Service Domain**, as shown in *Figure 11.18*:

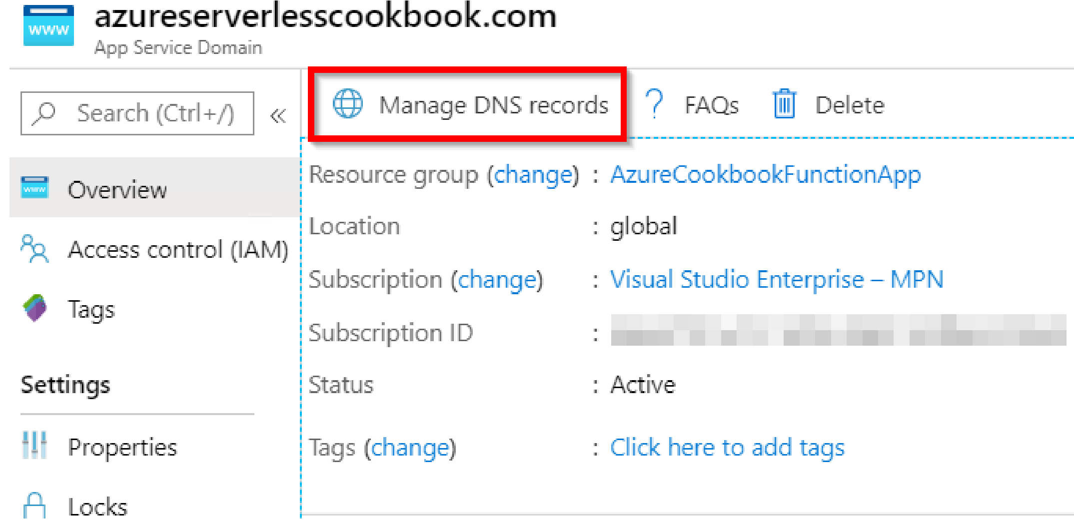


Figure 11.18: Azure App Service Domain overview

3. By clicking on the **Manage DNS records** button shown in *Figure 11.18*, you will be taken to the page shown in *Figure 11.19*:

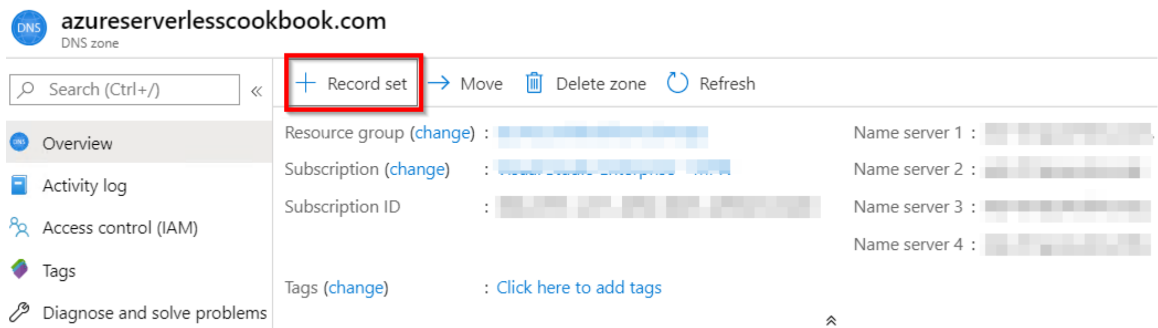


Figure 11.19: App Service Domain—DNS zone overview

4. Now, click on the **Record set** button to add a new **CName** record, as shown in *Figure* 11.20. You need to provide the default URL of your function app in the **Alias** text box:

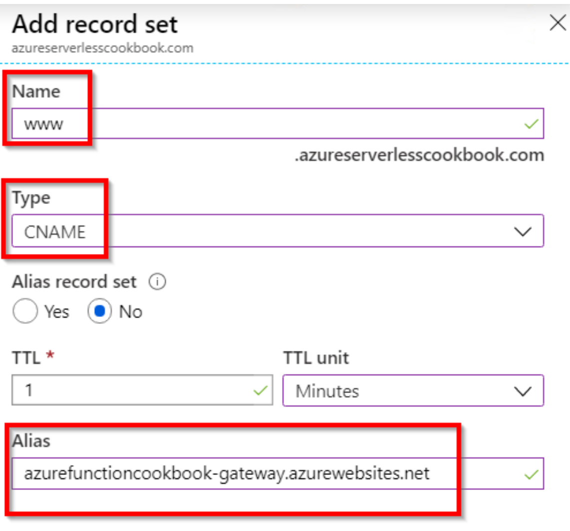


Figure 11.20: App Service Domain—adding a record set to the DNS zone

5. Once you have added the **CName** record, navigate to the **Custom Domains** pane of your function app and click on **Add custom domain**, as shown in *Figure* 11.21:

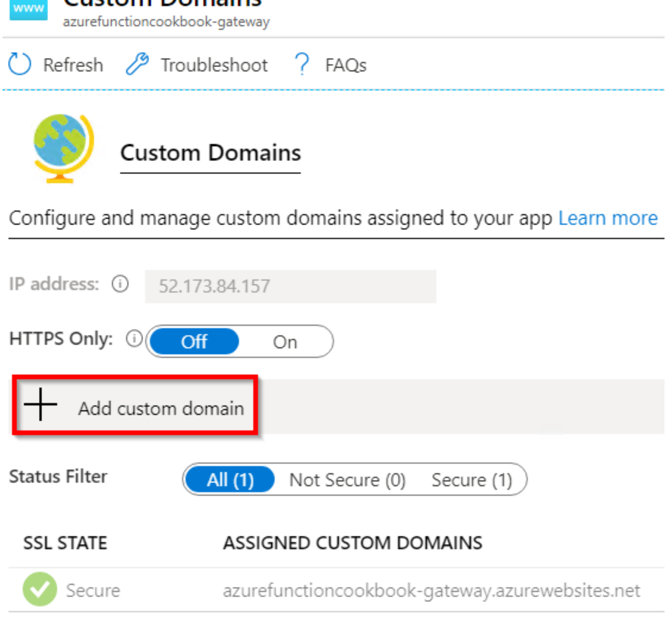


Figure 11.21: Azure Functions—custom domain details

6. This opens an **Add custom domain** pop-up window where you are prompted to provide the **Custom domain** name that you want to associate with. Provide the name of the domain and click on **Validate**, as shown in *Figure 11.22*:

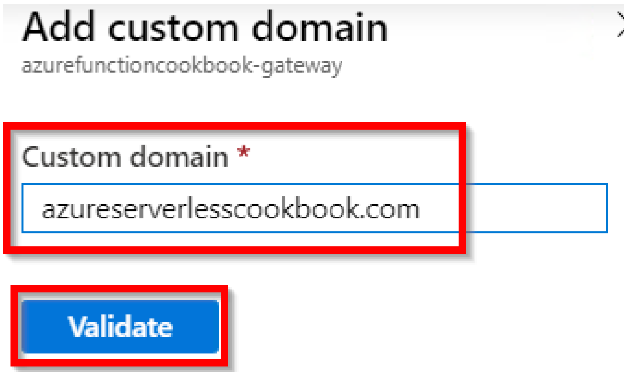


Figure 11.22: Azure Functions—adding a custom domain

7. After clicking on **Validate**, choose **CName** in the **Hostname record type** drop-down menu and click on the **Add custom domain** button, as shown in *Figure 11.23*:

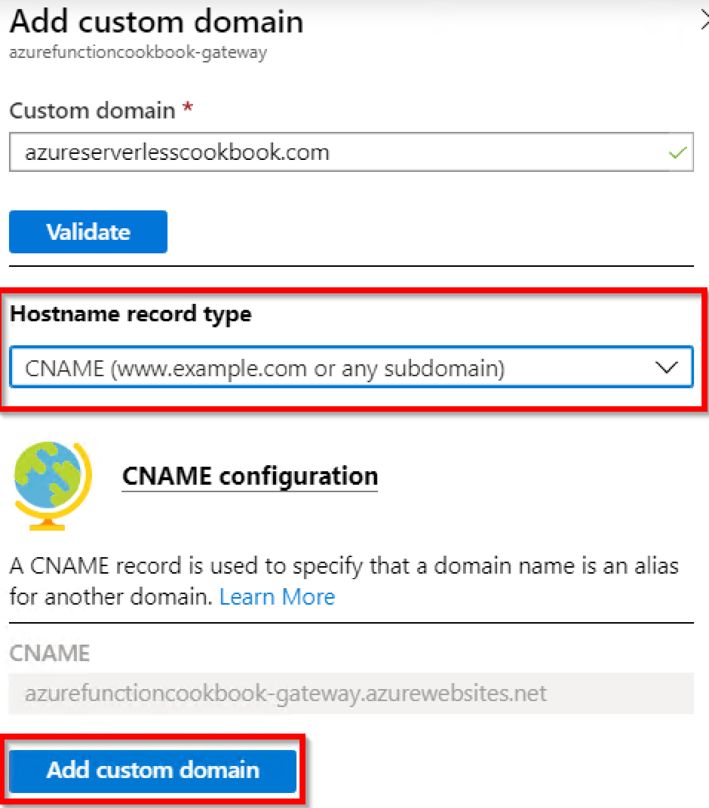


Figure 11.23: Azure Functions—adding a custom domain

8. That's it! You have successfully configured a custom domain for a function app, as shown in *Figure 11.24*:

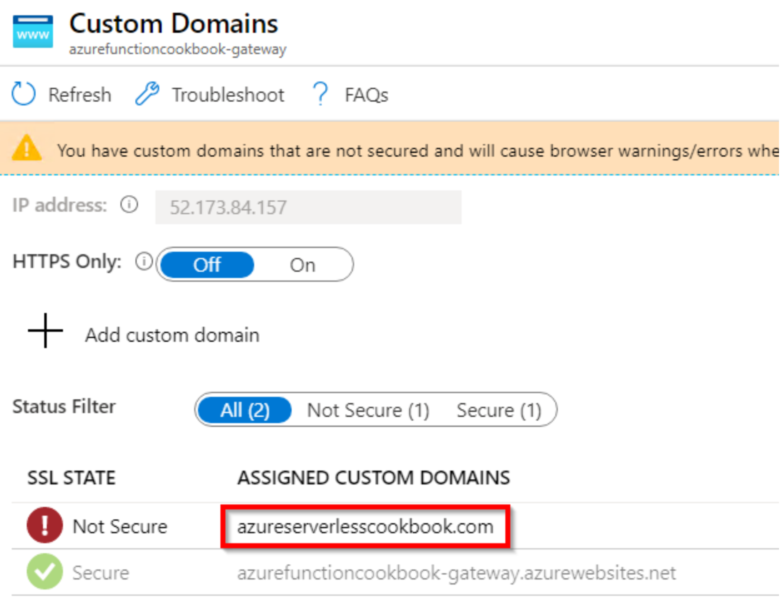


Figure 11.24: Azure Functions—Custom Domains

9. Now, open a new browser tab and access the custom domain (in my case, it is **`azureserverlesscookbook.com`**); this should show the function app page, as shown in *Figure 11.25*:

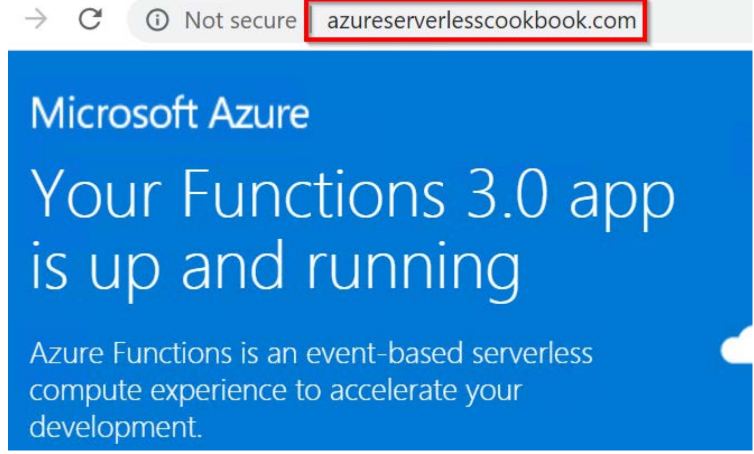


Figure 11.25: Accessing Azure Functions using a custom domain

In this recipe, we have learned how to create and configure the custom domain for a function app in order to access all the functions that you have created therein. Let's now move on to the next recipe.

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 may want them to change in the future, after the application goes live, without touching the code.

In general, these configuration items can be classified into two categories:

- Some of the configuration items might be different across environments, for example, the connection strings of the database and the 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 the possible use of the configuration value, you need to have a place to store configuration values that need to be accessed by the application.

In this recipe, we'll 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 V3 Functions runtime if one has not already been created. We will use the function app that was created in *Chapter 4*, *Developing Azure functions using Visual Studio*.

How to do it...

In this recipe, we'll look at a few ways of accessing the configuration values.

Accessing application settings and connection strings in the Azure function code

In this section, we'll learn how to access the configuration values using the **ConfigurationBuild** class by performing the following steps:

1. Create a configuration item with the **MyAppSetting** key and a **ConnectionStrings** with the **sql_dbconnection** key in the **local.settings.json** file. The **local.settings.json** file should look something like *Figure 11.26*:

Figure 11.26: Visual Studio—local configuration file

2. Replace the existing code with the following code. We have added a few lines that read the configuration values and the connection strings:

```
configuration.GetConnectionStringOrSetting("MyAppSettings")
```

The **GetConnectionStringOrSetting** method could be used to either get the value of an app setting or the value of a connection string.

The **configuration["MyAppSettings"]** indexer can be used to get the value of an app setting:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;
```

```
namespace FunctionAppinVisualStudio
{
    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("MyAppSettings");
            logger.LogInformation("Get Connection String Or Setting -
MyAppSettings = " + ValueFromGetConnectionStringOrSetting);

            var ValueFromConfigurationIndex =
configuration["MyAppSettings"];
            logger.LogInformation("Value From Configuration Index -
MyAppSettings = " + ValueFromConfigurationIndex);

            var ValueFromConnectionString = configuration.
GetConnectionStringOrSetting("connectionStrings:sql_dbconnection");
            logger.LogInformation("ConnectionStrings: sql_dbconnection = "
+ 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 clicking on **Publish** in the menu.

4. Add the configuration key and the connection string in the **Configuration** pane. Add the app setting as shown in *Figure 11.27*:

Figure 11.27: Azure Functions—adding an app setting using the configuration pane

5. Add the connection string as shown in *Figure 11.28*:

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel. Settings. Learn more

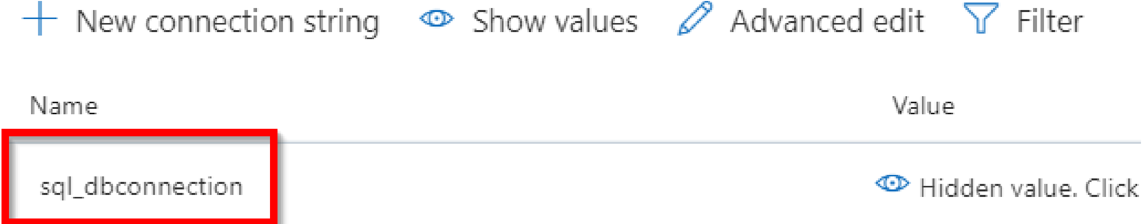


Figure 11.28: Azure Functions—adding a connection string using the configuration pane

6. Run the function by clicking on the **Run** button, which logs the output in the **Output** window:

Logs Console Reconnect Copy logs Pause Clear Expand

```
2020-03-28T11:29:34  Welcome, you are now connected to log-streaming service. The default timeout is 2 hours. Change the timeout with
the App Setting SCM_LOGSTREAM_TIMEOUT (in seconds).
2020-03-28T11:30:09.199 [Information] Executing 'HttpTriggerCSharpFromVS' (Reason='This function was programmatically called via the
host APIs.', Id=f2ef3d4e-9def-4268-8d92-9f4e6bb31ad6)
2020-03-28T11:30:09.228 [Information] Get Connection String Or Setting - MyAppSettings = Hello..! I'm a value from app setting
2020-03-28T11:30:09.228 [Information] Value From Configuration Index - MyAppSettings = Hello..! I'm a value from app setting
2020-03-28T11:30:09.228 [Information] ConnectionStrings: sql_dbconnection = connection_string_here
2020-03-28T11:30:09.236 [Information] Executed 'HttpTriggerCSharpFromVS' (Succeeded, Id=f2ef3d4e-9def-4268-8d92-9f4e6bb31ad6)
```

Figure 11.29: Azure Functions—viewing the app settings and connection string in the console logs

In this section, we have learned how to access configuration items using `ConfigurationBuilder` in the code. Let's now move on to the next section to learn about binding expressions.

Application settings–binding expressions

In the previous section, we learned how to access configuration settings from the code. Sometimes, you might want to configure some of the declarative items, too. You could achieve that using binding expressions. You'll understand what I mean in a moment when we look at the code:

1. Open Visual Studio and make changes to the **Run** method to add a new parameter to configure the **QueueTrigger**:

```
public static IActionResult Run
    (
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route= null)]HttpRequest req,
    ILogger logger,
    [QueueTrigger("hardcodedqueuename")] string queue
    )
```

Figure 11.30: Azure Functions—QueueTrigger—"hardcodedqueuename"

2. The **hardcodedqueuename** parameter is the name of the queue in 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 expressions:

```
public static IActionResult Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]HttpRequest req,
    ILogger logger,
    [QueueTrigger("%queuename%")] string queue)
```

Figure 11.31: Azure Functions—QueueTrigger binding expression

3. The application setting key must be enclosed in `%...%` and a key with the name **queuename** should be created in **Application settings**.

In this recipe, we have learned how to access the configuration items using both the **ConfigurationBuilder** class and binding expressions.

Breaking down large APIs into smaller subsets using proxies

In recent times, one of the buzzwords in the industry has been microservices, where we develop our web components as microservices that can be managed (scaling, deployment, and so on) individually without impacting the other related components. Although the subject of microservices is itself a huge one, in this recipe, we'll try to build a few microservices that can be managed individually as independent function apps. However, we'll 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'll be implementing the following architecture:

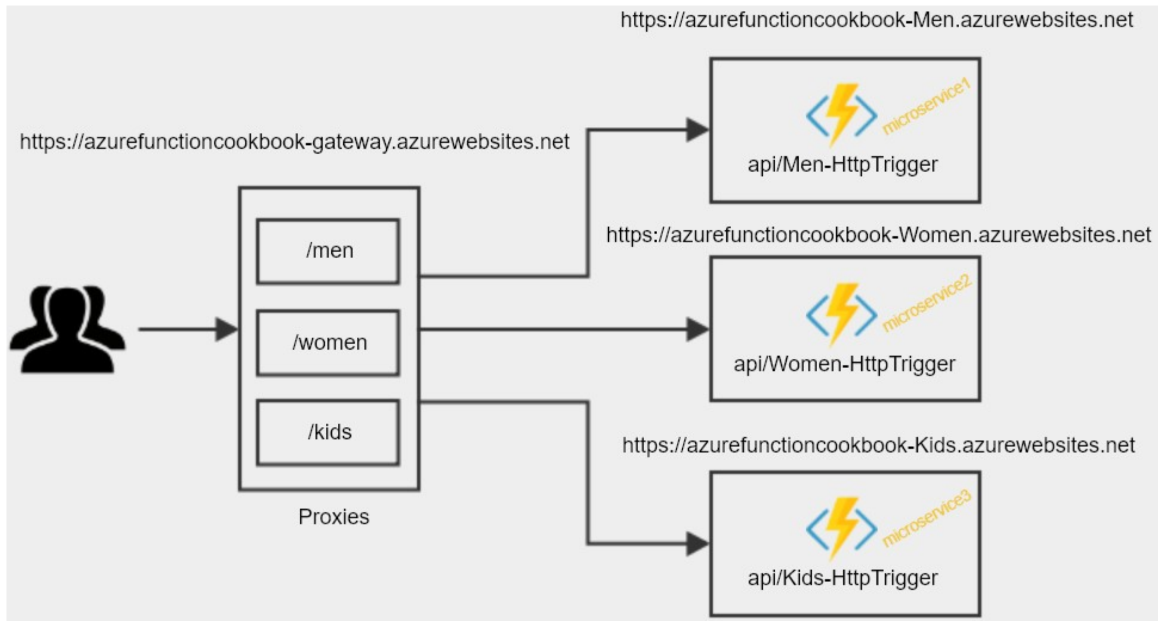


Figure 11.32: Azure function app with proxies—architecture

As depicted in the preceding architecture diagram, we are going to create three proxies in the gateway function app, which will be consumed by the client apps. Each proxy is responsible for the redirection of the request to the appropriate HTTP trigger based on the route template (**/men**, **/women**, and **/kids**).

Finally, the HTTP trigger is responsible for processing the request.

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 back-end APIs in a microservice architecture where each microservice is independent of the others.

In this recipe, we'll achieve this by creating the following function apps:

- A gateway component (function app) that is responsible for controlling the traffic to the correct microservice based on the route (**/men**, **/women**, or **/kids**). In this function app, we will 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'll be performing the following steps:

1. Creating all three microservices with one HTTP trigger in each of them
2. Creating, proxying, and configuring the respective microservice
3. Testing the proxy URL

Creating the microservices

In this section, we'll create the microservices by performing the following steps:

1. Create three function apps, one for each of the microservices, as well as the gateway function app that we have planned:

Showing 1 to 4 of 4 records.

Name	Status
azurefunctioncookbook-gateway	Running
azurefunctioncookbook-kids	Running
azurefunctioncookbook-men	Running
azurefunctioncookbook-women	Running

Figure 11.33: Creating four function apps (one gateway and three microservices)

2. Create the following anonymous HTTP triggers in each of the function apps, which display a message along the lines of what is shown in *Figure 11.34*:

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

Figure 11.34: Creating anonymous HTTP triggers

Creating the gateway proxies

Perform the following steps to create gateway proxies:

1. Navigate to the gateway function app and create a new proxy:

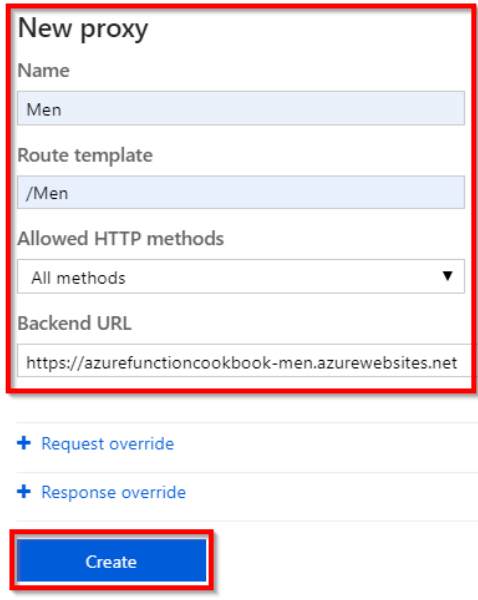


Figure 11.35: Azure function app—creating a proxy

2. You will then be taken to the details pane:

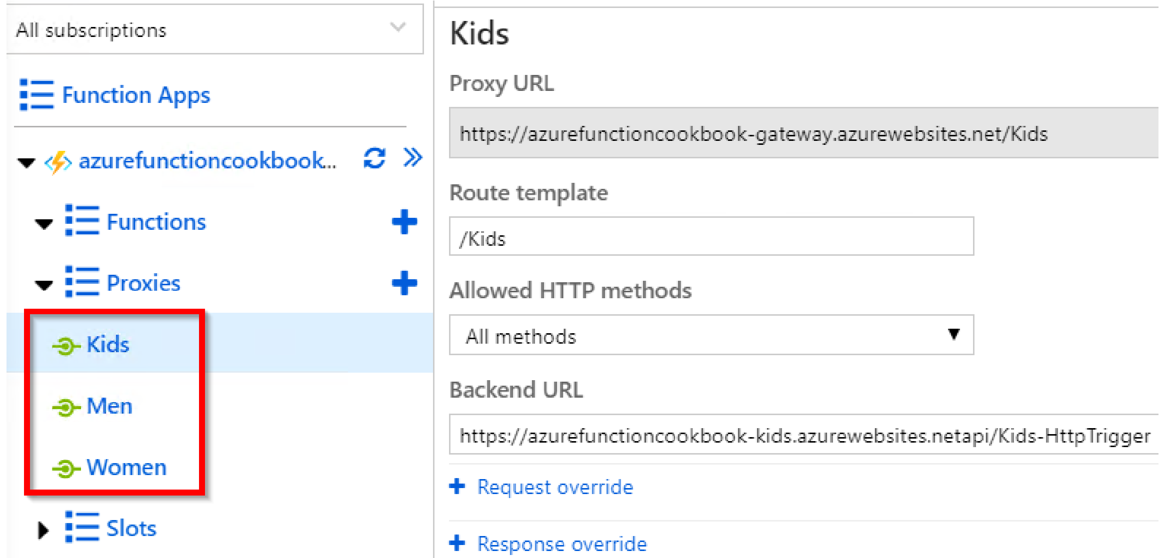


Figure 11.36: Azure function app—viewing proxies and details

3. Create the proxies for **Women** and **Kids**. Here are the details of all three proxies. Note that the back-end URLs (of the function apps) may be different based on the inputs:

Proxy name	Route template	Back-end 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`

Figure 11.37: Details of all three proxies

4. Once the three proxies have been created, the list will look something like this:

Proxies

NAME	BACKEND URL
Men	https://azurefunctioncookbook-men.azurewebsites.net/api/Men-HttpTrigger
Women	https://azurefunctioncookbook-women.azurewebsites.netapi/Women-HttpTrigger
Kids	https://azurefunctioncookbook-kids.azurewebsites.netapi/Kids-HttpTrigger

Figure 11.38: Azure Function app—viewing the proxies

5. In *Figure 11.38*, you can view three different domains. However, in order 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 **Proxies** 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, your HTTP triggers accept a required name parameter, and you need to pass the name query string to the proxy URL. Let's access the following URLs in the browser:

- Men:

https://azurefunctioncookbook-gateway.azurewebsites.net/Men?name=Praveen%20Sreeram

```
Hello, Praveen Sreeram
```

Figure 11.39: Azure function app proxy—output for the /men route template

- Women:

https://azurefunctioncookbook-gateway.azurewebsites.net/Women?name=Haritha%20Sreeram

```
Hello, Haritha Sreeram
```

Figure 11.40: Azure function app proxy—output for the /women route template

- Kids:

https://azurefunctioncookbook-gateway.azurewebsites.net/Kids?name=Baby%20Rithwika%20Sreeram

```
Hello, Baby Rithwika Sreeram
```

Figure 11.41: Azure function app proxy—output for /kids route template

Observe the URLs of the three preceding screenshots. You'll notice that they look like they are being served from a single application with different routes. However, they are three different microservices that can be managed individually.

There's more...

All the microservices created in this recipe are anonymous, which means they are publicly accessible. In order to make them secure, you need to follow either of the approaches recommended in *Chapter 10*, *Implementing best practices for Azure Functions*.

Azure function proxies also allow the interception of original requests and, if required, you can add new parameters and pass them to the back-end API. Similarly, you can add additional parameters and pass the response back to the client application. Learn more about Azure function proxies in the official documentation at https://docs.microsoft.com/azure/azure-functions/functions-proxies.

In this recipe, we have learned how to implement a microservice kind of architecture using the feature proxies in the Azure function app. Let's now move on to the next recipe.

Moving configuration items from one environment to another

Every application that you develop will have many configuration items (such as application settings and connection strings) stored in **`Web.Config`** files for all .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`**, this has 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 can still have the **`Web.Config`** files and they work as they used to in the traditional on-premises world. However, an Azure App Service provides us with an 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 will take quite a bit of time. In this recipe, we will learn the tip of exporting and importing these application settings from a lower environment (development, for example) to a higher environment (production, for example).

Getting ready

Perform the following steps:

1. Create a function app (say, **`MyApp-Dev`**) if one has not been created already.
2. Create some application settings:

Name	Value	Source
AppSetting0	Hidden value. Click show values button	App Config
AppSetting1	Hidden value. Click show values button	App Config
AppSetting2	Hidden value. Click show values button	App Config
AppSetting3	Hidden value. Click show values button	App Config
AppSetting4	Hidden value. Click show values button	App Config
AppSetting5	Hidden value. Click show values button	App Config
AppSetting6	Hidden value. Click show values button	App Config
AppSetting7	Hidden value. Click show values button	App Config
AppSetting8	Hidden value. Click show values button	App Config
AppSetting9	Hidden value. Click show values button	App Config

Figure 11.42: Azure function app—application settings in the configuration pane

3. Create another function app (say, **`MyApp-Prod`**).

This recipe showcases the ease of copying the application settings from one function to another. This technique will be handy when there are many application 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 **Resource Explorer**.
2. **Resource Explorer** will open, and from there you can traverse all the internal elements of a given service:

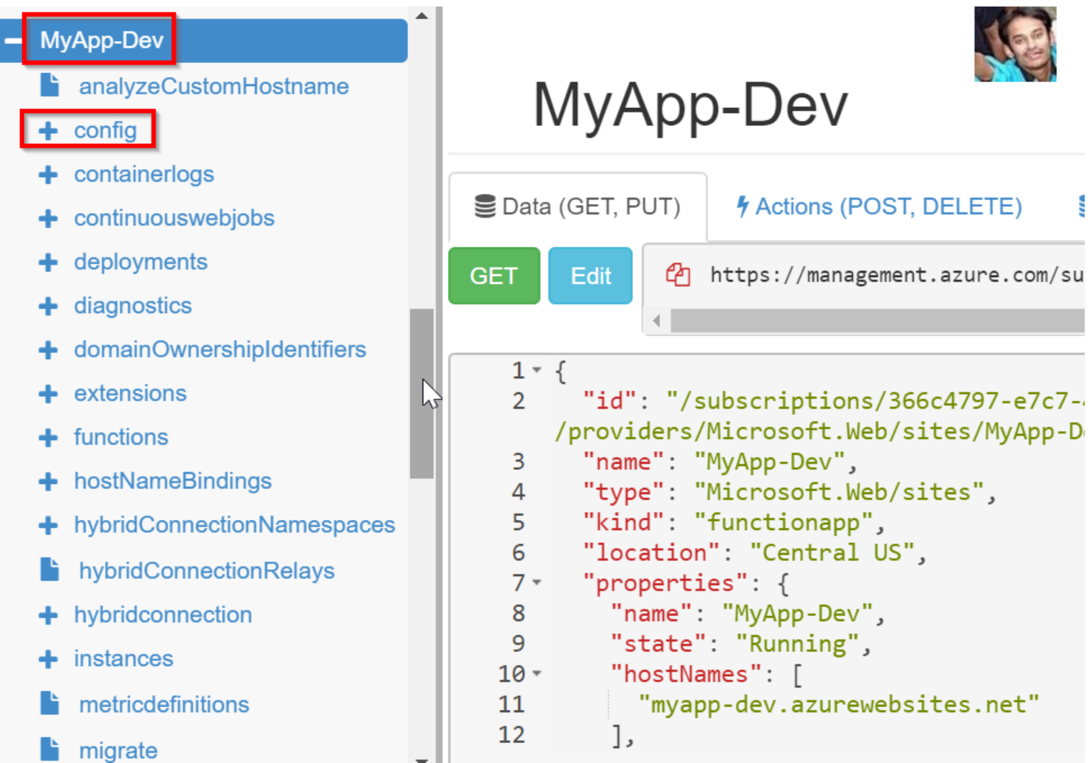


Figure 11.43: Azure resources view—selecting the config node

> **Note**
>
> Please be sure to open the setting in **Read/Write** mode (available in the top right-hand corner) in **Resource Explorer**.

3. Click on the **config** element, as shown in *Figure 11.43*, which opens all the items related to configurations:

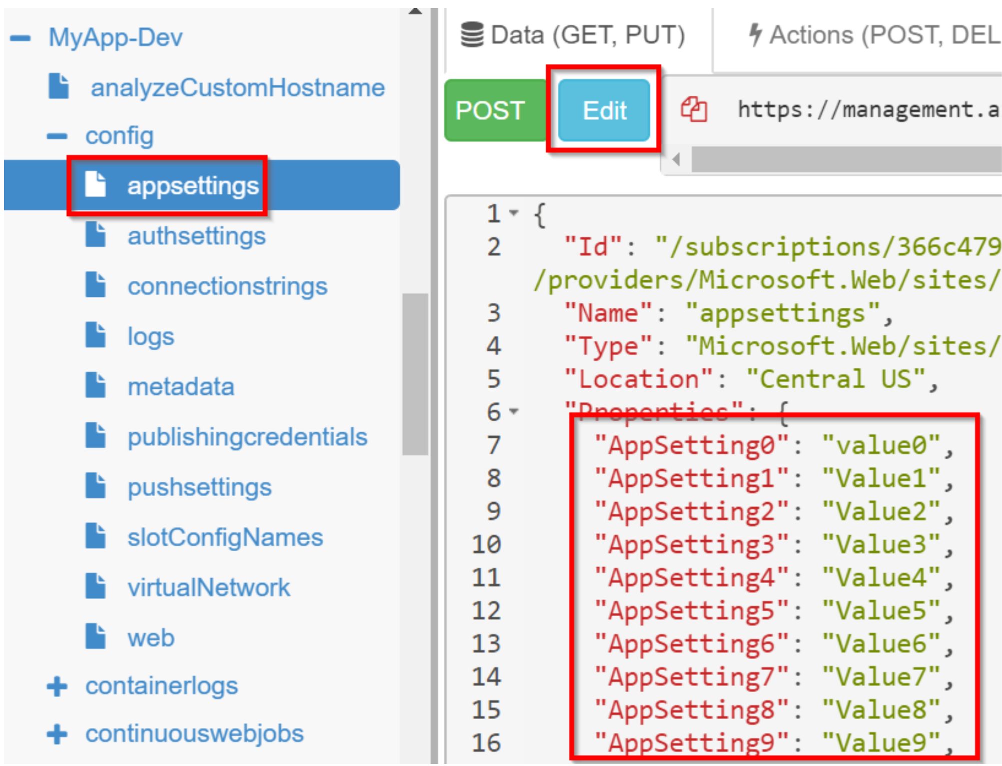


Figure 11.44: Azure resources view—editing the appsettings node

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 *Figure 11.44*, or you can copy all the application settings from **`AppSetting0`** to **`AppSetting9`**.

5. Navigate to the `MyApp-Prod` function app (which won't have the application settings highlighted in *Figure 11.44*), click on **Resource Explorer**, and then click on the **config | appsettings** elements to open the existing application settings. It should look something like this:

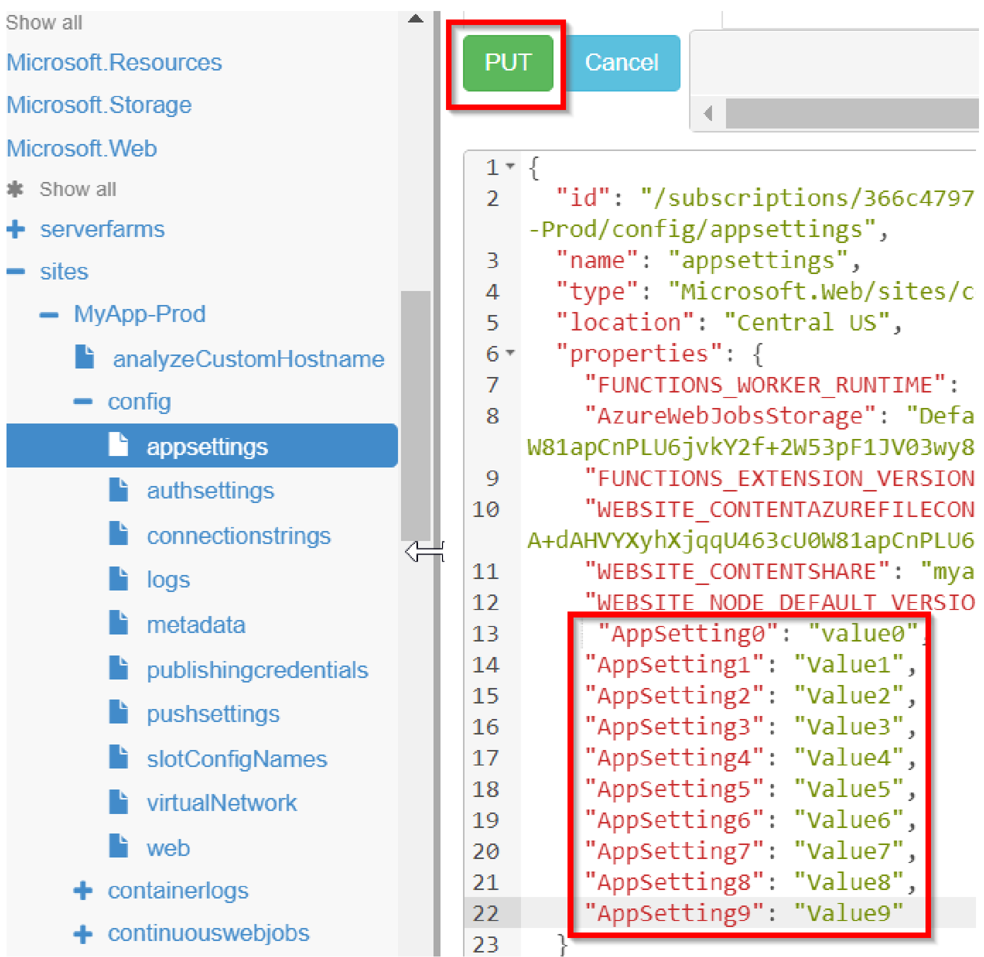


Figure 11.45: Azure resources view—updating appsettings

6. Click on the **Edit** button and paste the content that was copied earlier. After reviewing the settings, click on **PUT**, which is shown in *Figure 11.45*.

7. Navigate to the application settings pane of the **`MyApp-Prod`** function app:

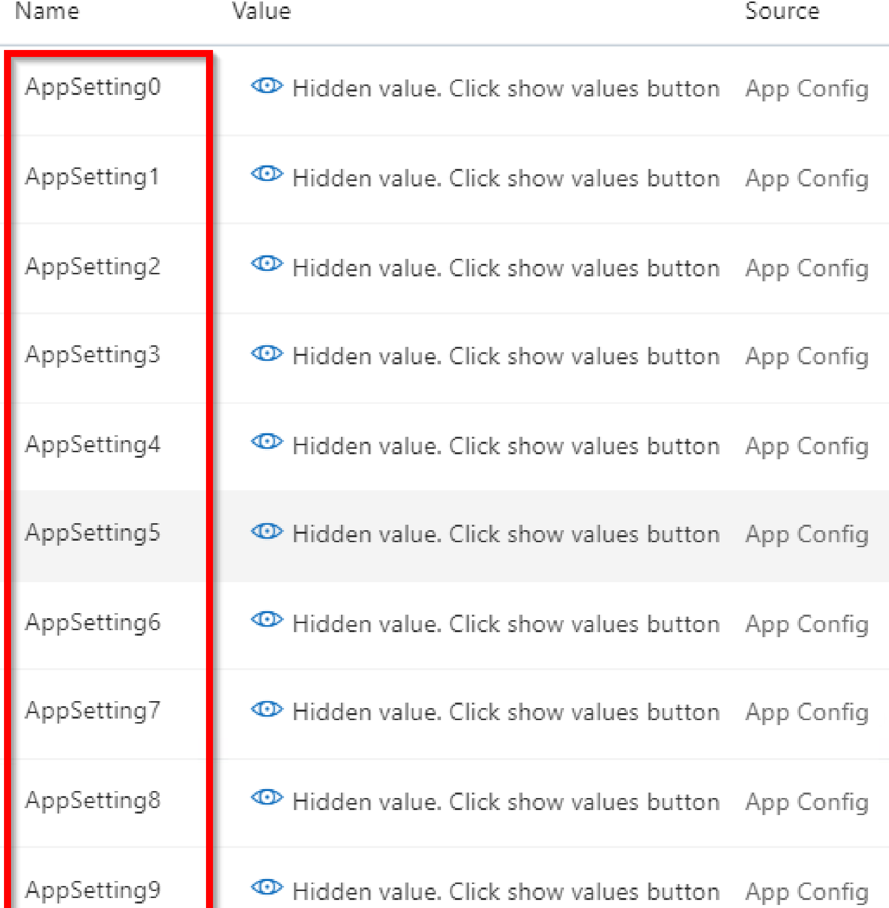


Figure 11.46: Azure function app—the configuration pane of the app settings

You should see all the application settings that we have created in **Resource Explorer**, as shown in *Figure 11.46*.

In this recipe, we have learned a quick way of copying the configuration items from one function app to another.

In this chapter, we have discussed some of the important techniques that will help a developer to improve their productivity, as well as some of the best practices that need to be followed in the production environment.

12

Implementing and deploying continuous integration using Azure DevOps

In this chapter, you'll learn about the following topics:

- Continuous integration–creating a build definition
- Continuous integration–queuing a build and triggering it manually
- Continuous integration–configuring and triggering an automated build
- Continuous integration–executing unit test cases in the pipeline
- Creating a release definition
- Triggering a release automatically
- Integrating Azure Key Vault to configure application secrets

Introduction

As a software professional, you may be aware of different software development methodologies that are followed across the industry. Irrespective of the methodology being followed, there will be multiple environments, such as development, staging, and production, where the application life cycle needs to be followed, with these critical stages related to development:

1. Develop the application based on the requirements.
2. Build the application and fix any errors.
3. Deploy/release the package to an environment (development/staging/production).
4. Test against the requirements.
5. Promote the release to the next environment (from development to staging and staging to production).

> **Note**
>
> For the sake of simplicity, the initial stages, such as requirement gathering, planning, designing, and architecture, are excluded, just to emphasize the stages that are relevant to this chapter.

For each change made 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 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.

To streamline and automate a few of the steps mentioned earlier in this chapter, we'll discuss some of the popular techniques that the industry uses in order to deliver software quickly, with minimal infrastructure.

> **Note**
>
> In previous chapters, most of the recipes provided us with a solution for an individual business problem. However, this chapter will try to provide a solution for the **continuous integration** (**CI**) and continuous delivery of business-critical applications.

The Azure DevOps team continuously adds new features to Azure DevOps (https://dev.azure.com), formerly known as VSTS (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 with what you see at https://dev.azure.com.

Prerequisites

Do the following if you haven't done so already:

1. Create an Azure DevOps organization at https://dev.azure.com and create a new project in that account. While creating the project, either choose Git or Team Foundation Version Control as the version control repository. Let's use **Git** version control for our project:

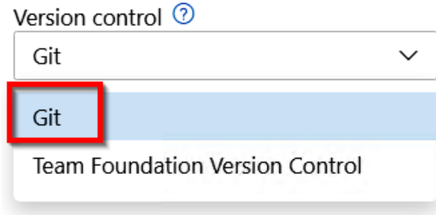


Figure 12.1: Creating an Azure DevOps project with Git version control

2. Configure the Visual Studio project that was developed in *Chapter 4*, *Developing Azure functions using Visual Studio*, to Azure DevOps. Go to https://docs.microsoft.com/azure/devops/organizations/accounts/set-up-vs?view=azure-devops to follow the step-by-step process of creating a new account and project using Azure DevOps.

> **Note**
>
> I will be making some small changes to the response messages embedded within the code to show different outputs. Make sure that you modify the unit tests accordingly. Otherwise, the build will fail.

Continuous integration—creating a build definition

In this recipe, we will learn how to configure continuous integration by creating a build definition. A build definition is a set of tasks that are required to configure an automated build of 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 **Git**, as shown in *Figure 12.2*:

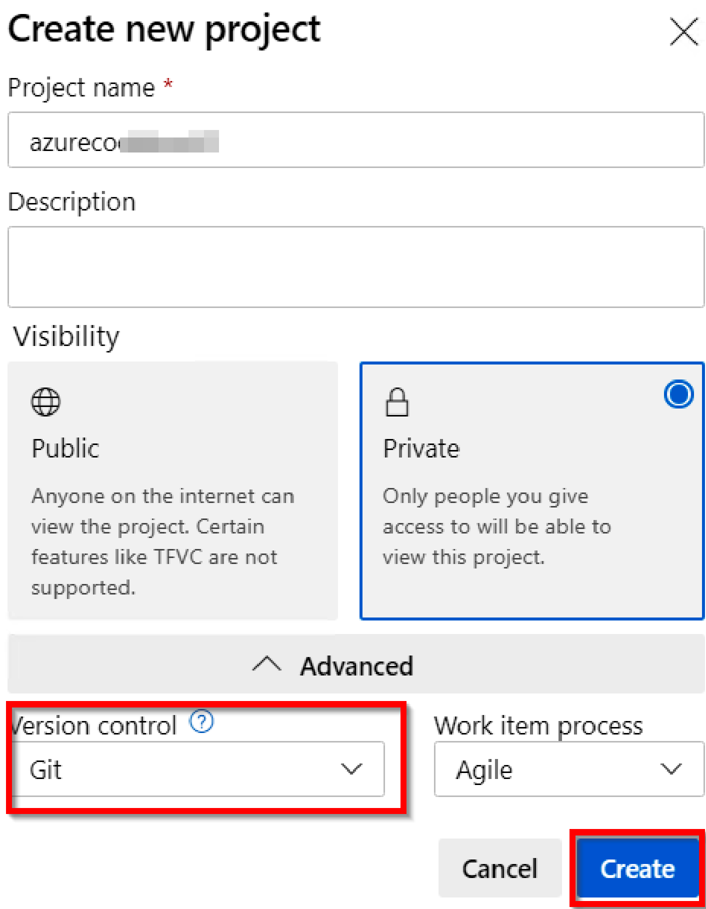


Figure 12.2: Creating a private Azure DevOps project with Git version control

How to do it...

In order to create the build definition, we'll have to perform the following steps:

1. Navigate to the **Pipelines** tab in the **Azure DevOps** account, click on **Pipelines**, and choose **Create Pipeline** to start the process of creating a new build definition, as shown in *Figure 12.3*:

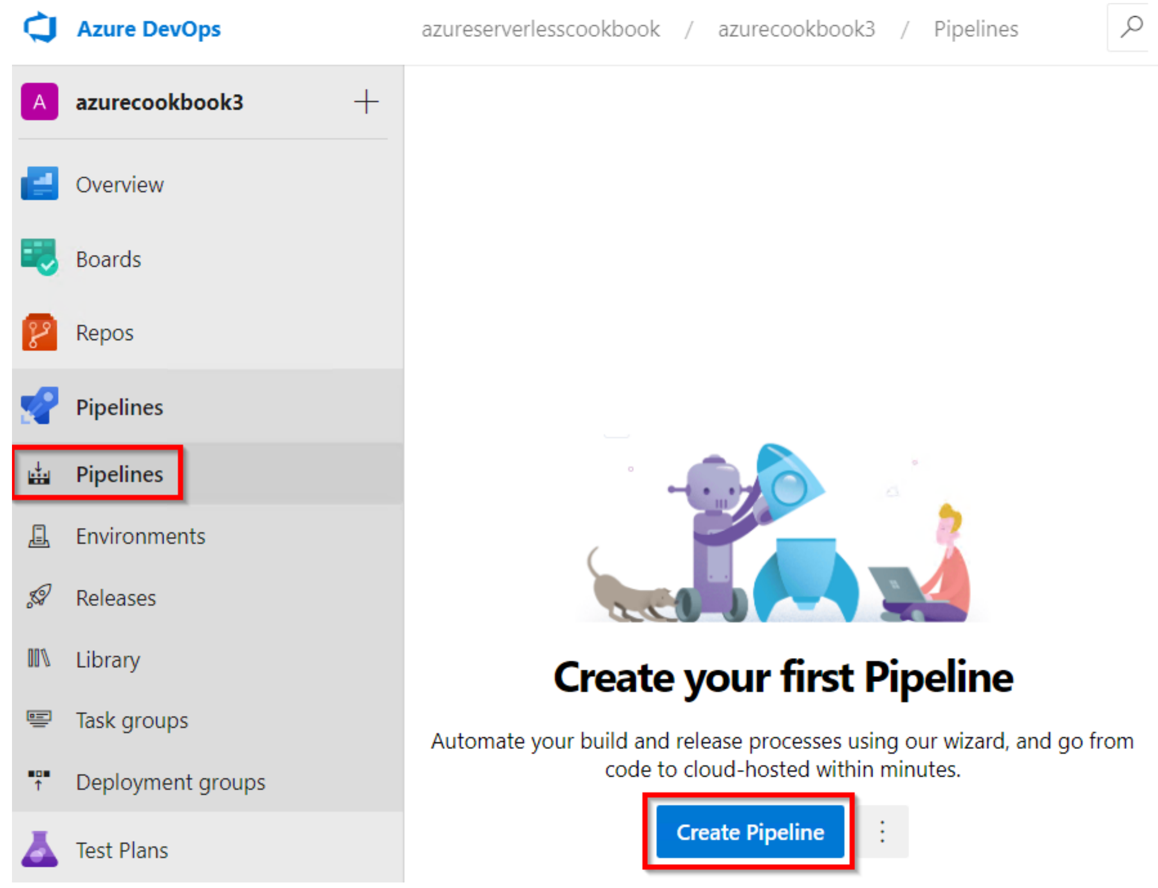


Figure 12.3: Azure DevOps—the Create Pipeline button

2. In the next step, click on the **Use the classic editor** link, as shown in *Figure 12.4*:

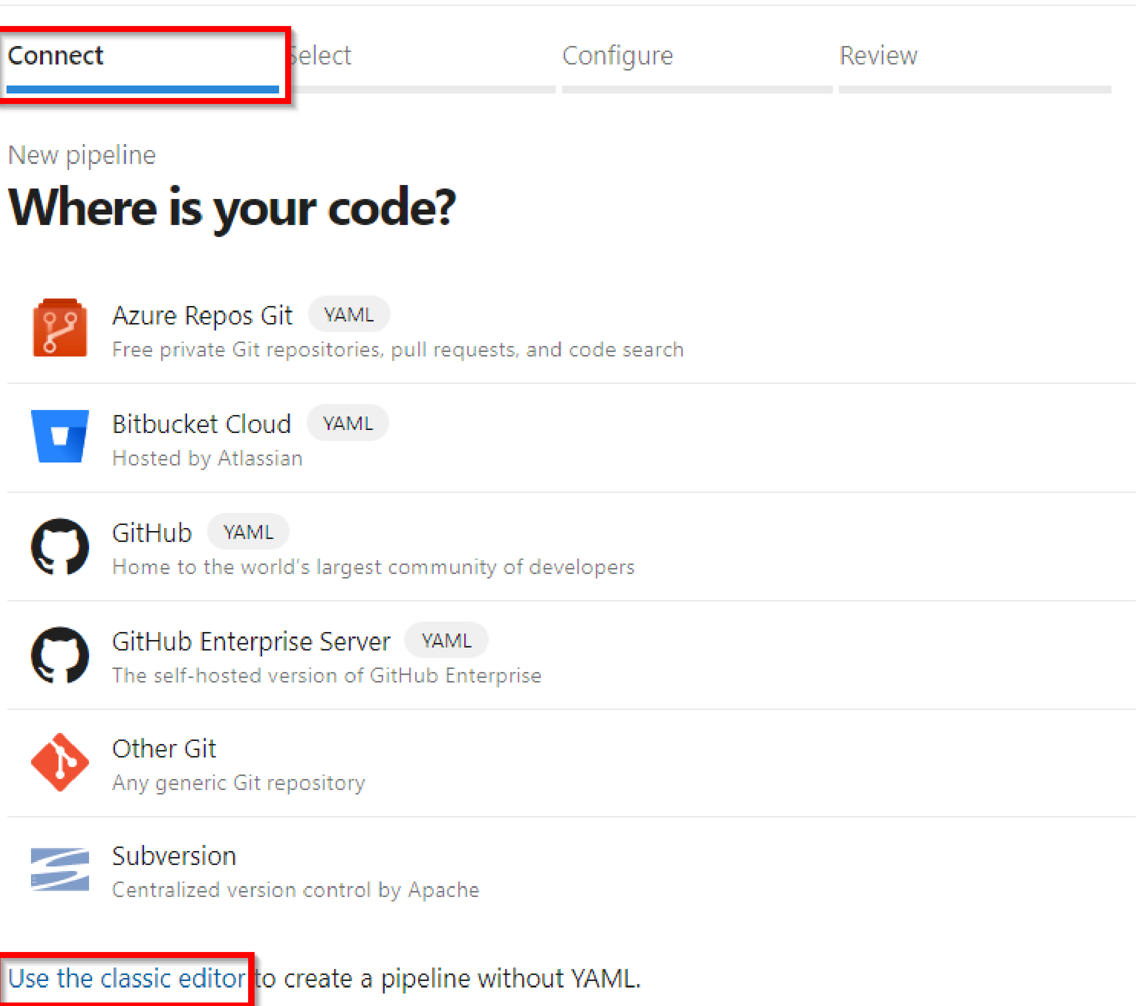


Figure 12.4: Azure DevOps—using the classic editor to create a pipeline

3. You will be taken to the **Select a source** screen, where you can choose your repository. For this example, ours is Git. As shown in *Figure 12.5*, select **Azure Repos Git** and click on **Continue**. Make sure that you have chosen your project, which in this case is `azurecookbook3`, and the `azurecookbook3` repository:

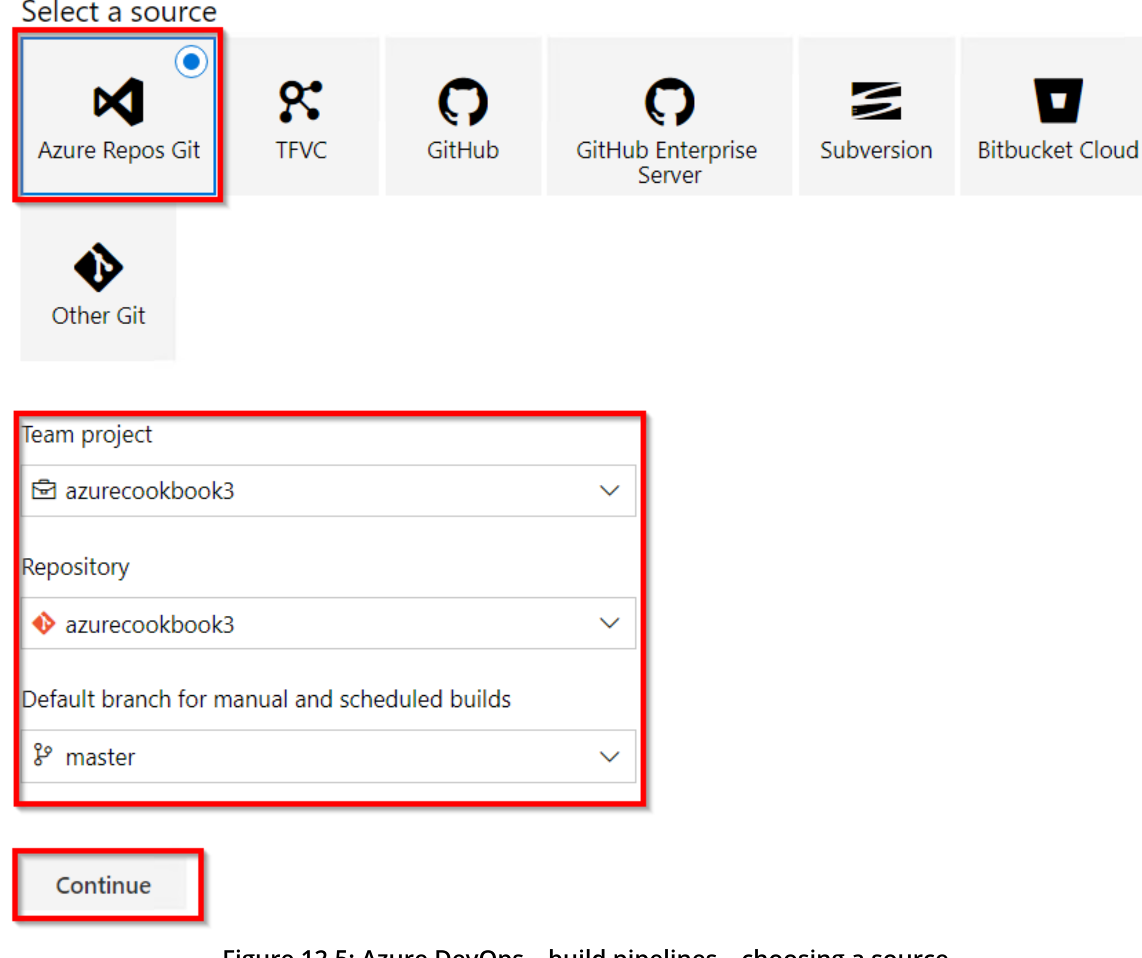


Figure 12.5: Azure DevOps—build pipelines—choosing a source

4. You will be taken to the **Select a template** step, where you can choose the template required for your application. For this recipe, let's choose **Azure Functions for .NET**, as shown in *Figure* 12.6, by clicking on the **Apply** button:

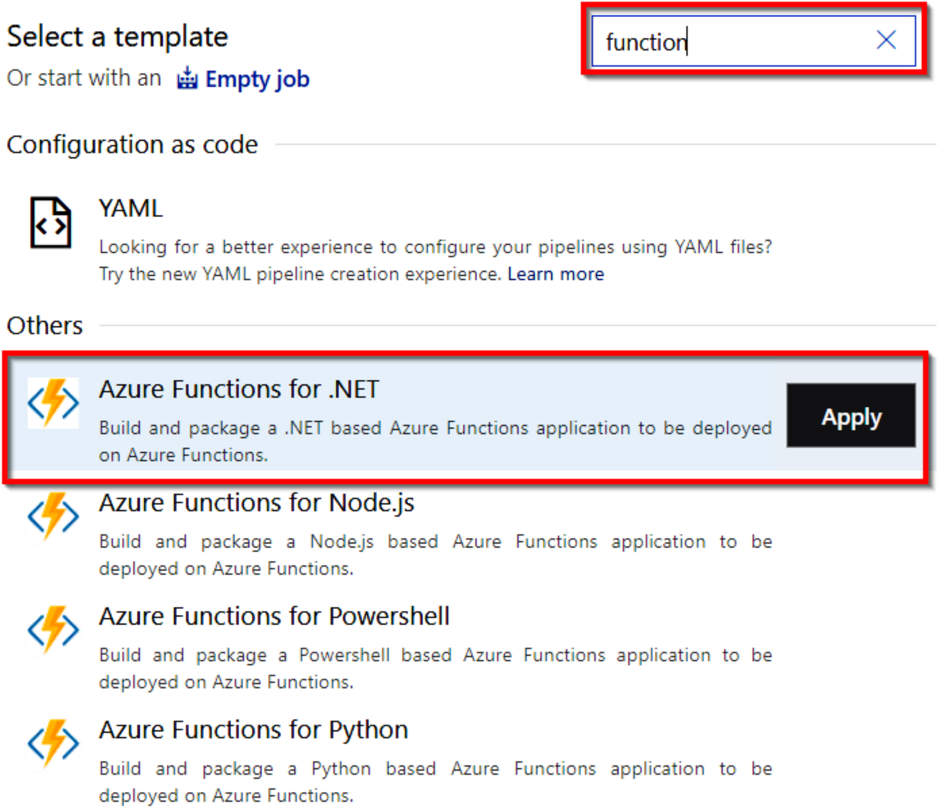


Figure 12.6: Azure DevOps—build pipelines—selecting a template

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 we provide inputs for each of those fields based on our requirements. Let's start by providing a meaningful name in the pipeline step. Be sure to choose **vs2017-win2016** in the **Agent Specification** drop-down list, as shown in *Figure 12.7*:

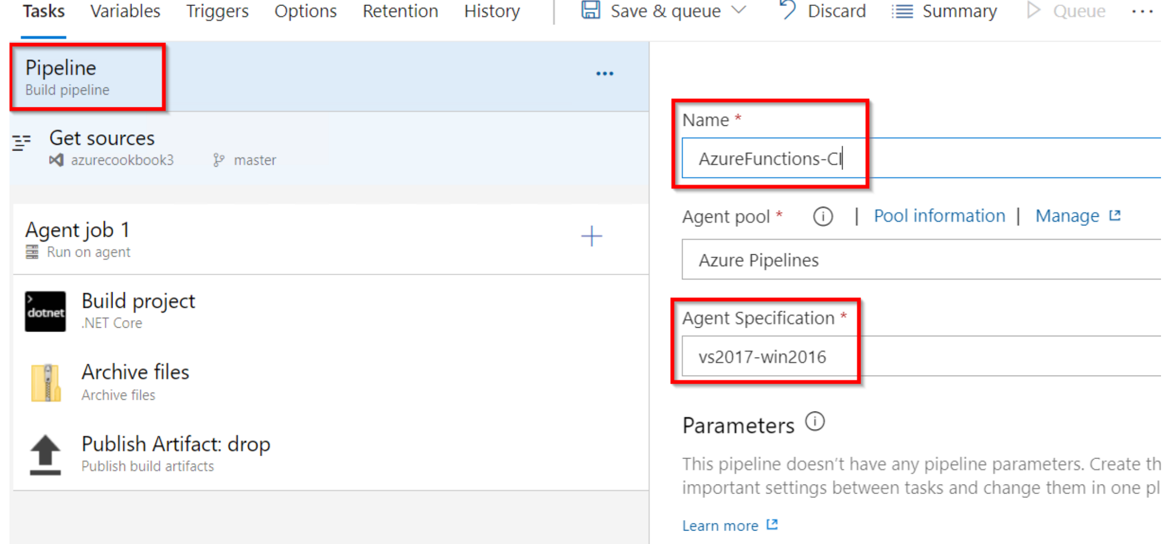


Figure 12.7: Azure DevOps—build pipelines—configuring the pipeline

> **Note**
>
> Agent Specification defines the agent (a virtual machine) to be used. An agent in the current context is a virtual machine that has the required tools and software pre-installed.

6. In the **Get sources** step, ensure that the following are done as shown in *Figure 12.8*:

   Select the version control system based on the project's requirements.

   Choose the repository that we want to build. In this example, we have chosen **azurecookbook3**:

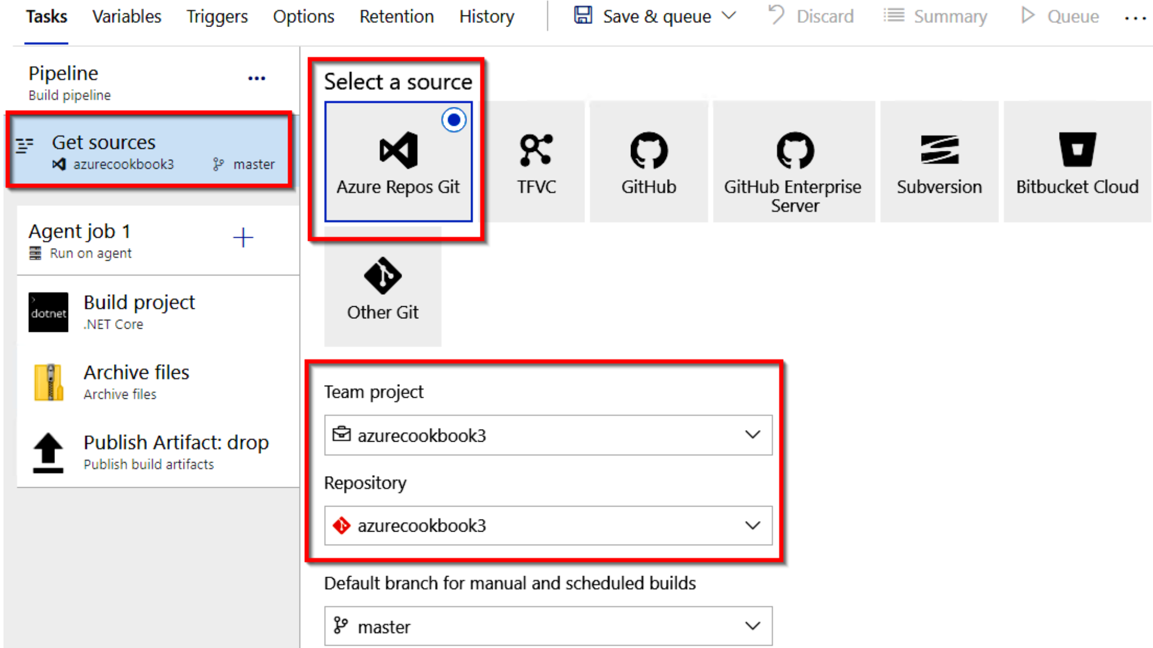


Figure 12.8: Azure DevOps—build pipelines—viewing and editing the repository

7. Once all the values in all the fields are reviewed, click on **Save**, as shown in *Figure 12.9*, and click on **Save** again in the **Save build pipeline** pop-up window:

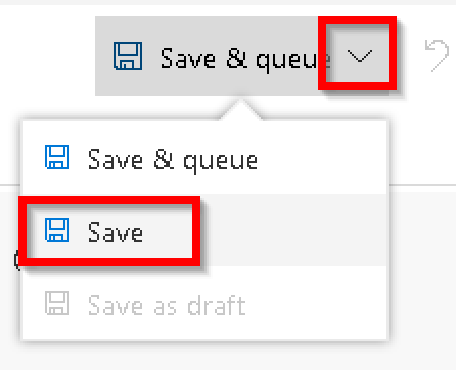


Figure 12.9: Azure DevOps—build pipelines—save pipeline

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 we 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 they have been configured. The steps can also be rearranged 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 it starts the process of building the package. Once that process is complete, the build process 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).

> **Note**
>
> `Build.ArtifactStagingDirectory` is a predefined variable that contains the local path on the agent where any artifacts are copied before being pushed to their destination.

Learn more about pre-defined variables at https://docs.microsoft.com/azure/devops/pipelines/build/variables?view=azure-devops&tabs=yaml.

There's more...

Azure DevOps provides many tasks. Choose a new task for a template by clicking on the **Add Task** (+) button.

If we don't find a task that meets our requirements, we can search for a suitable one in the marketplace by clicking on the **Marketplace** button shown in *Figure 12.10*:

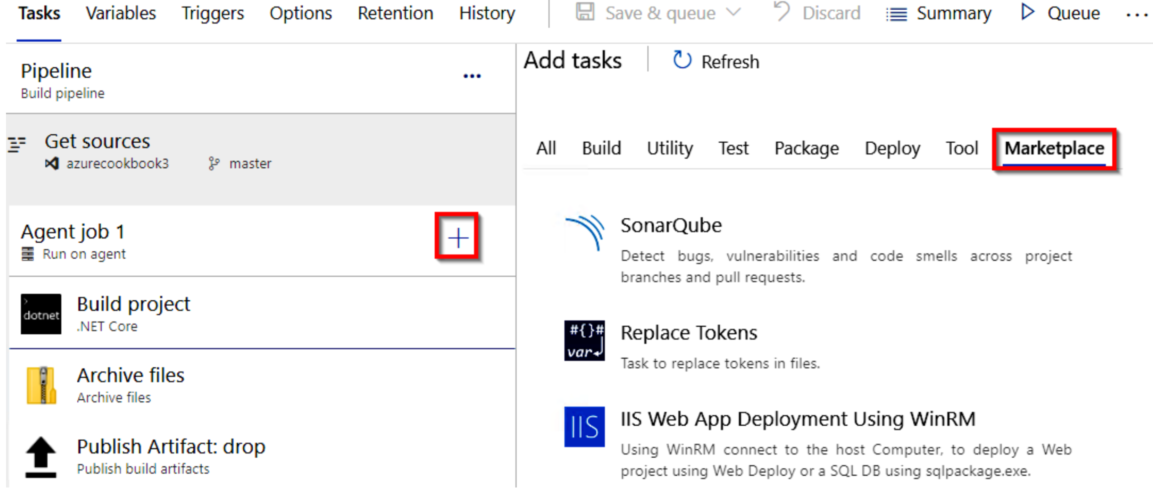


Figure 12.10: Azure DevOps—build pipelines—adding a task from Marketplace

In this recipe, we have learned how to create a build pipeline. Let's move on to the next recipe.

Continuous integration—queuing a build and triggering it manually

In the previous recipe, you learned how to create and configure a build definition. In this recipe, you will learn how to trigger a build manually and understand the process of building an application.

Getting ready

Before we begin, make sure that you have done the following:

- Configured the build definition as mentioned in the previous recipe.
- Checked all of your source code into the Azure DevOps team project.

How to do it...

Perform the following steps:

1. Navigate to the build definition named **AzureFunctions-CI**, click on the **Edit** button, and then click on the **Queue** button available on the right-hand side, as shown in *Figure 12.11*:

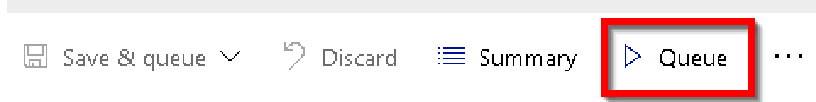


Figure 12.11: Azure DevOps—build pipelines—the Queue button

2. In the **Azure Pool for AzureFunctions-CI** pop-up window, make sure that the **vs2017-win2016** option is chosen in the **Agent Specification** drop-down list in Visual Studio 2017 or 2019 and click on the **Queue** button, as shown in *Figure 12.12*:

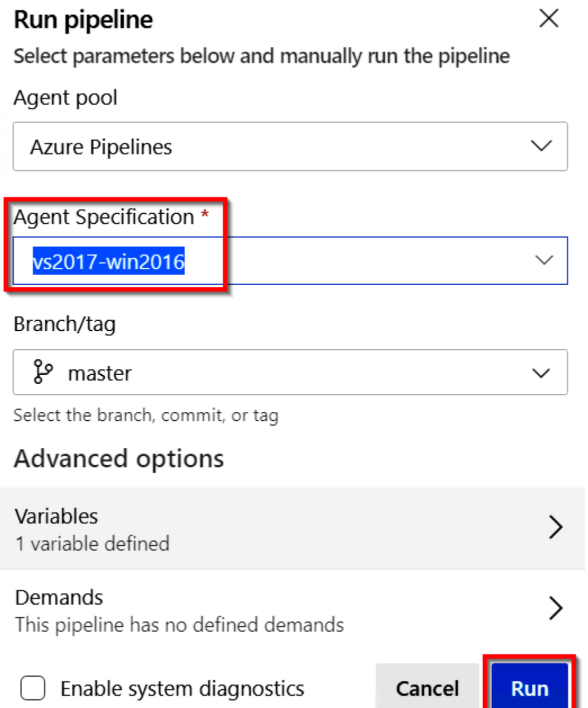


Figure 12.12: Azure DevOps—build pipelines—running a pipeline

> **Note**
>
> At the time of writing, the VS 2019 option is not available. While reading, if the VS 2019 option becomes available, feel free to choose that.

3. In just a few moments, the build will be queued and the message will be displayed, as shown in *Figure 12.13*:

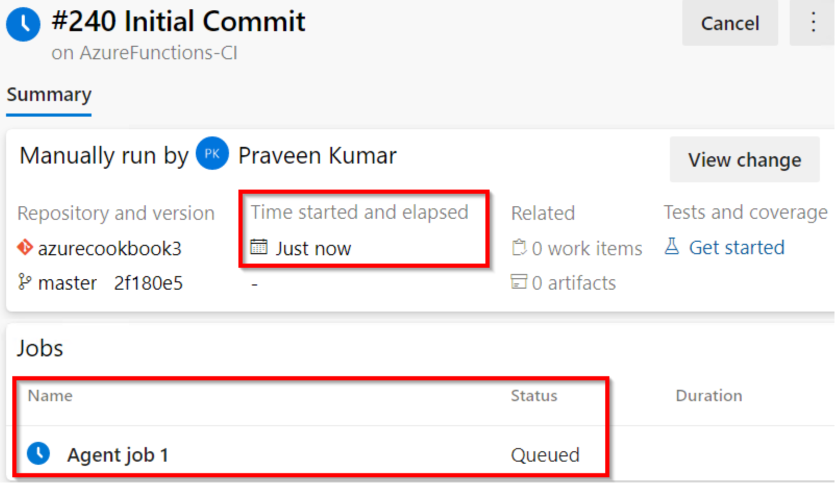


Figure 12.13: Azure DevOps—build pipelines—viewing progress

4. 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 by clicking on **Agent job 1** in the preceding step. You will see the status of all the tasks, as shown in *Figure 12.14*:

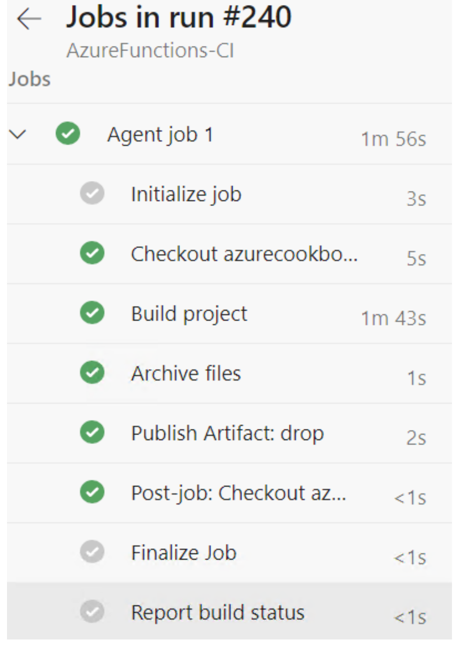


Figure 12.14: Azure DevOps—build pipelines—viewing the job summary

5. To view the output of the build, click on the **published** button highlighted in *Figure 12.15*:

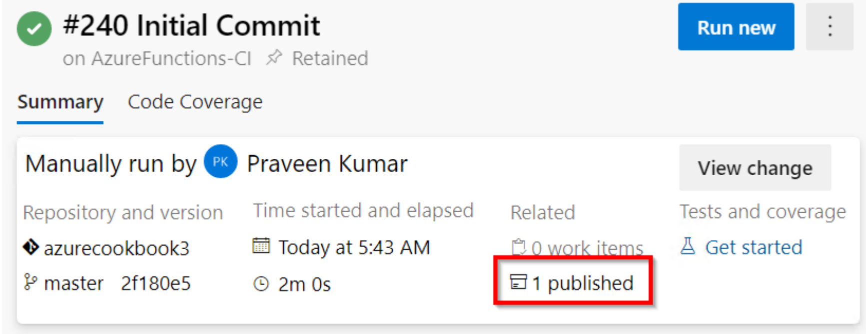


Figure 12.15: Azure DevOps—build pipelines—viewing the published artifact

6. Download the files by clicking on the download button, as shown in *Figure 12.16*:

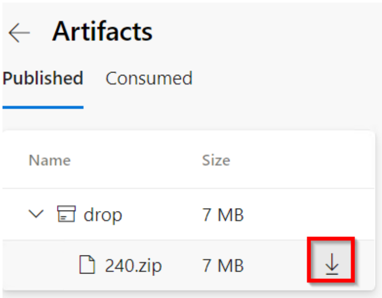


Figure 12.16: Azure DevOps—build pipelines—downloading the published artifact

In this recipe, we have configured the pipeline and also triggered it manually to test whether the pipeline is configured properly. Let's move on to the next recipe.

Continuous integration—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 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 the team project and also trigger an automated build by making a change to the code of the HTTP trigger Azure function that we created in *Chapter 4*, *Developing Azure functions using Visual Studio*.

How to do it...

Perform the following steps:

1. Navigate to the **AzureFunctions-CI** build definition by clicking on the **Edit** button, as shown in *Figure 12.17*:

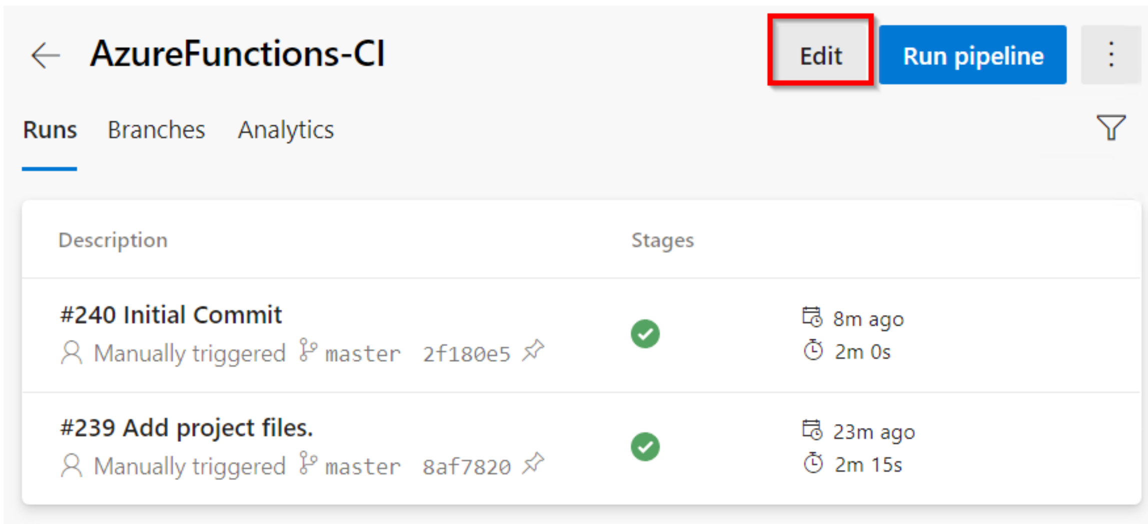


Figure 12.17: Azure DevOps—build pipelines—editing a pipeline

2. Once inside the build definition, click on the **Triggers** menu, as shown in *Figure 12.18*:

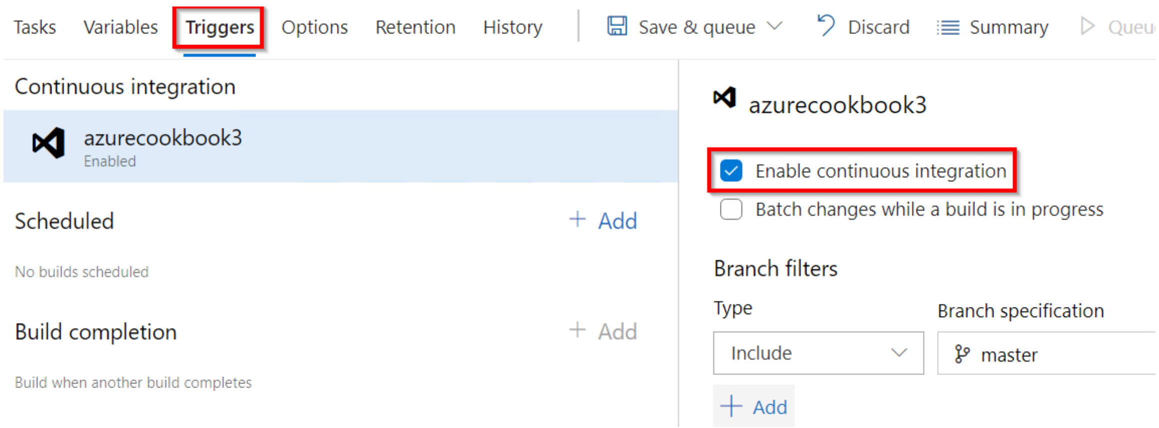


Figure 12.18: Azure DevOps—build pipelines—enabling continuous integration

3. Now, click on the **Enable continuous integration** checkbox to enable the automated build trigger.
4. Save the changes by clicking on the arrow sign available beside the **Save & queue** button and click on the **Save** button available in the drop-down menu shown in *Figure 12.19*:

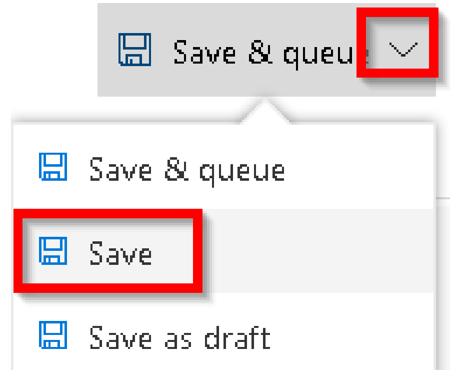


Figure 12.19: Azure DevOps—build pipelines—saving the pipeline

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 here. We will just replace the word **`hello`** with **`Automated Build Trigger test`**, as follows:

```
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 in *Figure 12.20*, click on **Commit All** to commit the code and then, in the next step, push all the changes:

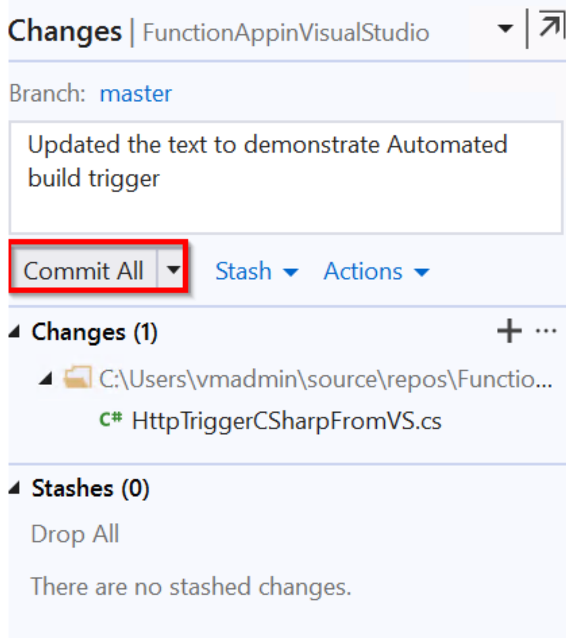


Figure 12.20: Visual Studio—Team Explorer—committing changes to Git

7. Now, immediately navigate back to the Azure DevOps build definition to see that a new build was triggered automatically and is in progress, as shown in *Figure 12.21*:

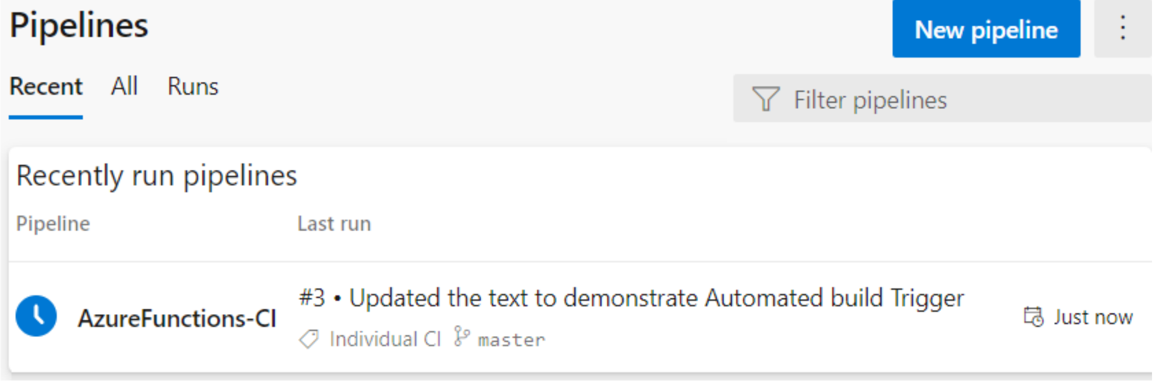


Figure 12.21: Azure DevOps—build pipelines—pipeline triggered automatically

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 code base and checked it into Git.
3. Automatically, a new build was triggered in Azure DevOps. The build was triggered immediately after the code was committed to Git.

In this recipe, we have learned how to configure continuous integration for Azure Functions using Azure DevOps build pipelines. Let's move on to the next recipe.

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 to validate the correctness of our code. It is also important that we run these unit tests every time the developer commits 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 with HTTP triggers* recipe of *Chapter 5*, *Exploring testing tools for Azure functions*.

How to do it...

In this recipe, we are going to add a new task to the pipeline that runs the unit test cases. Perform the following steps:

1. Edit the **AzureFunctions-CI** build definition and add the **.NET Core** task as shown in *Figure 12.22*:

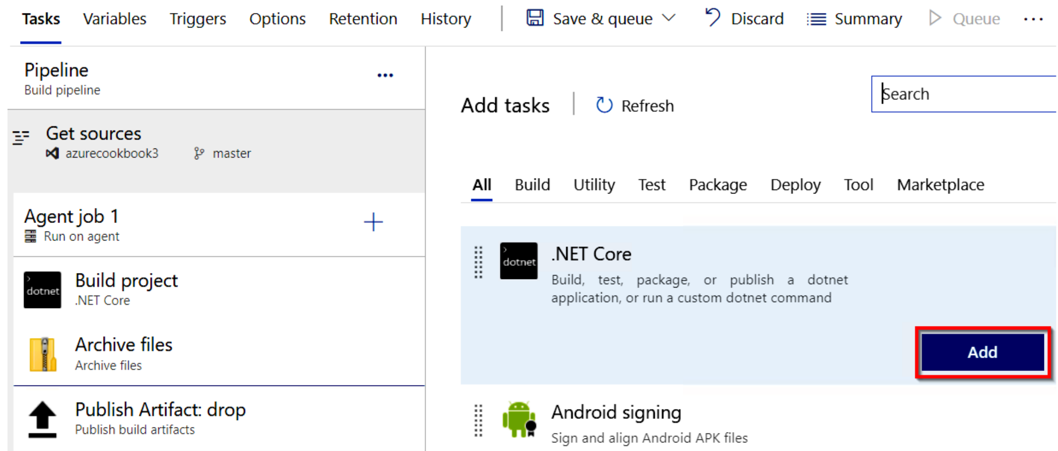


Figure 12.22: Azure DevOps—build pipelines—adding a new task

2. Once the task is added, change the following attributes of the task:

   Display Name: The name of the task. Change it to **`Test`**.

   Command: This is the command to run. Please choose the **`test`** option. The command will take care of running the unit test cases.

   Path to Project(s): This is the name of the unit test project. Provide **`**/*Test*.csproj`** or the actual name of the unit test project.

   Arguments: The arguments for building the application. Provide **`--output publish_output --configuration release`** as the command.

3. Once all these changes are made, the **Test** task should look something like *Figure 12.23*. After reviewing everything, click on **Save** to save the changes:

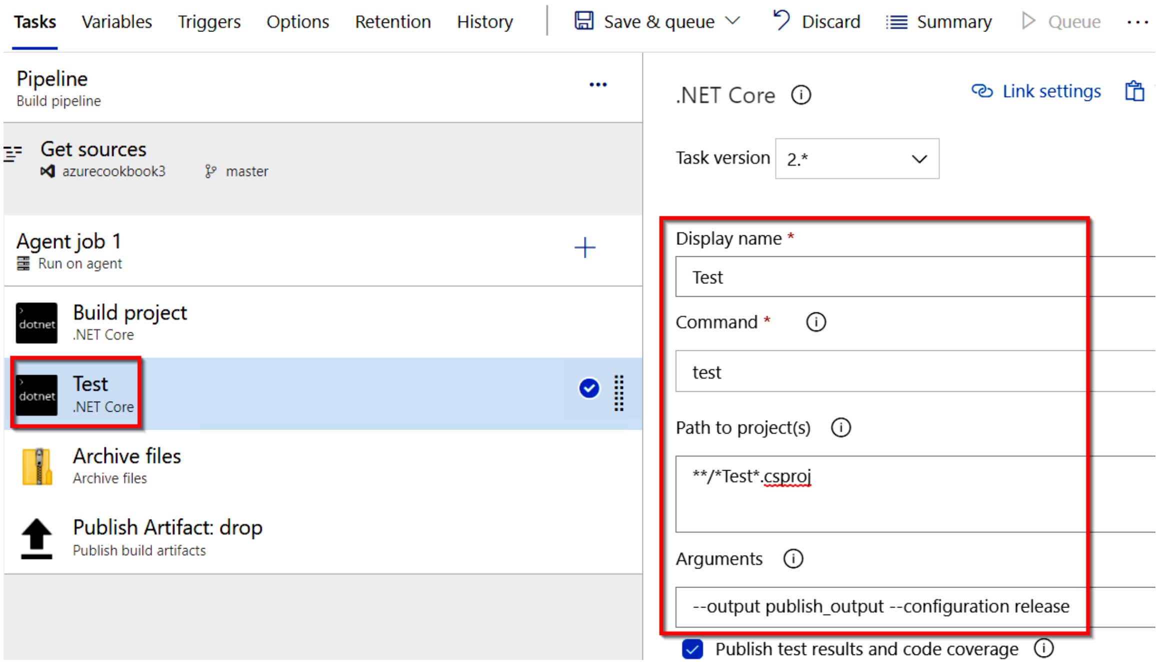


Figure 12.23: Azure DevOps—build pipelines—configuring the .NET Core Test task

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 be passed without any warnings, as shown in *Figure 12.24*:

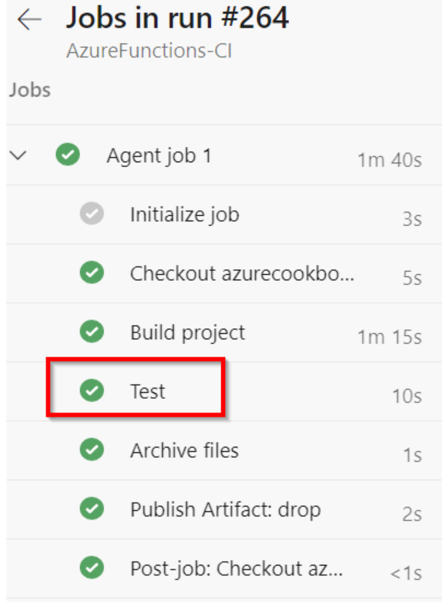


Figure 12.24: Azure DevOps—build pipelines—viewing the job status

5. Here is a summary of the test cases. *Figure 12.25* is a chart that shows the percentage of the test cases that have passed and failed:

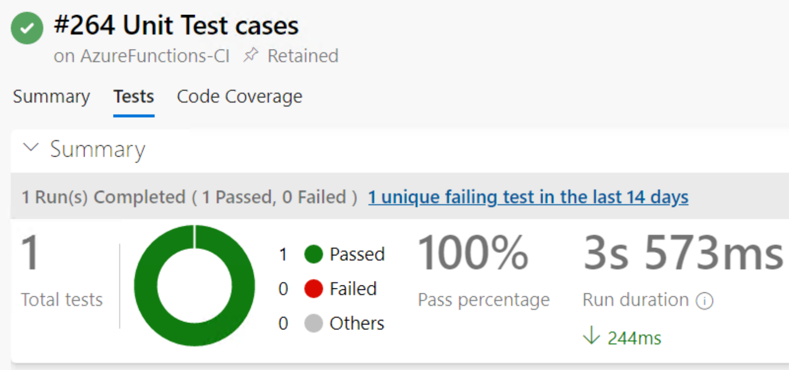


Figure 12.25: Azure DevOps—build pipelines—viewing the test case summary

There's more...

If all the naming conventions were followed as per the instructions, then we won't face any issues with this recipe. However, we may come across issues if we have used a different name for the unit project and haven't used the word `test` somewhere in the name of the project (which is the same name as the generated `.dll` file).

In the recipe, we used `*` and `**`, which are called file matching patterns. Learn more about file matching patterns at https://docs.microsoft.com/azure/devops/pipelines/tasks/file-matching-patterns?view=azure-devops&viewFallbackFrom=vsts.

In this recipe, we have learned how to create and configure a task for running unit test cases. Let's move on to the next recipe.

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, we need to create a release definition in the same way that we created the build definitions.

Getting ready

Before working on this recipe, please make sure you have created the build definition and also ensure that you have run it successfully at least once.

How to do it...

To release and deploy the package to an environment, we'll perform the following steps:

1. Navigate to the **Releases** tab, as shown in *Figure* 12.26, and click on the **New pipeline** button:

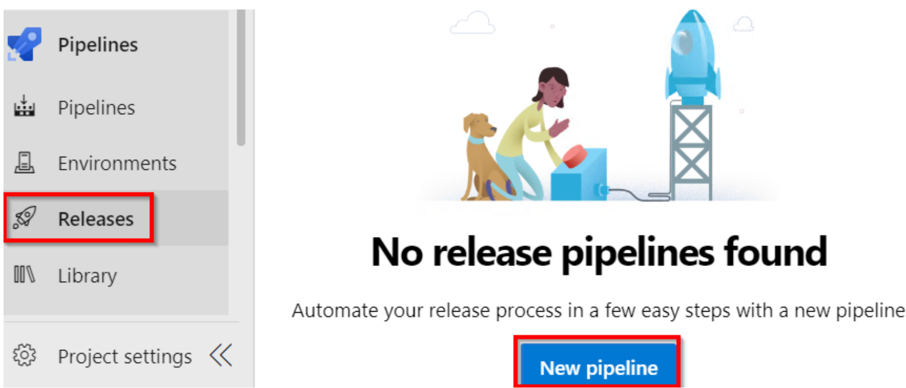


Figure 12.26: Azure DevOps—release pipelines—New Pipeline

2. The next step is to choose a release definition template. In the **Select a template** pop-up window, select **Deploy the function app to Azure Functions** and click on the **Apply** button, as shown in *Figure 12.27*. Immediately after clicking on the **Apply** button, a new environment (stage) pop-up window will be displayed. For now, just close the **Environment** pop-up window:

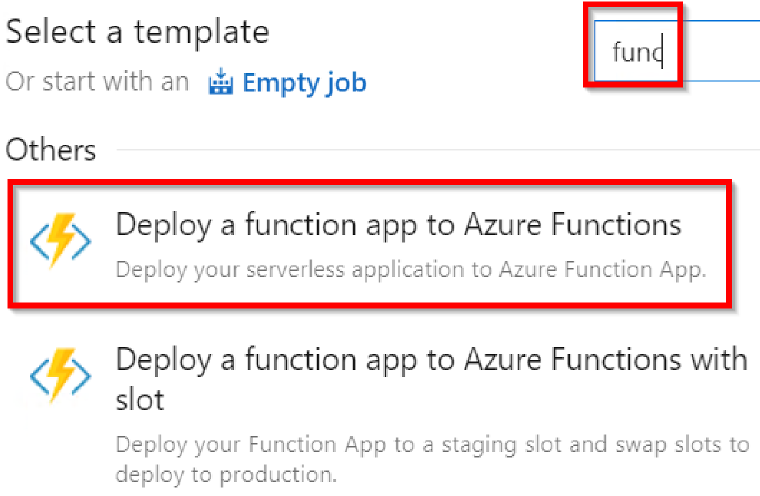


Figure 12.27: Azure DevOps—release pipelines—choosing the Deploy a function app to Azure Functions app template

3. Click on the **Add** button available in the **Artifacts** box to add a new artifact, as shown in *Figure 12.28*:

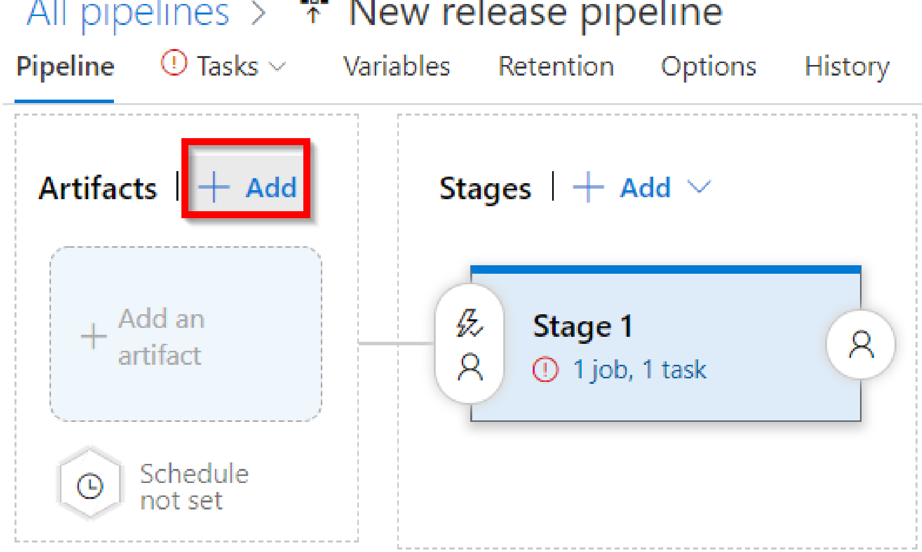


Figure 12.28: Azure DevOps—release pipelines—adding a new stage

4. In the **Add an artifact** pop-up window, make sure to choose the following:

   Source type: Build

   Project: The team project your source code is linked to

   Source (build pipeline): The build pipeline name where your builds are created

   Default version: Latest:

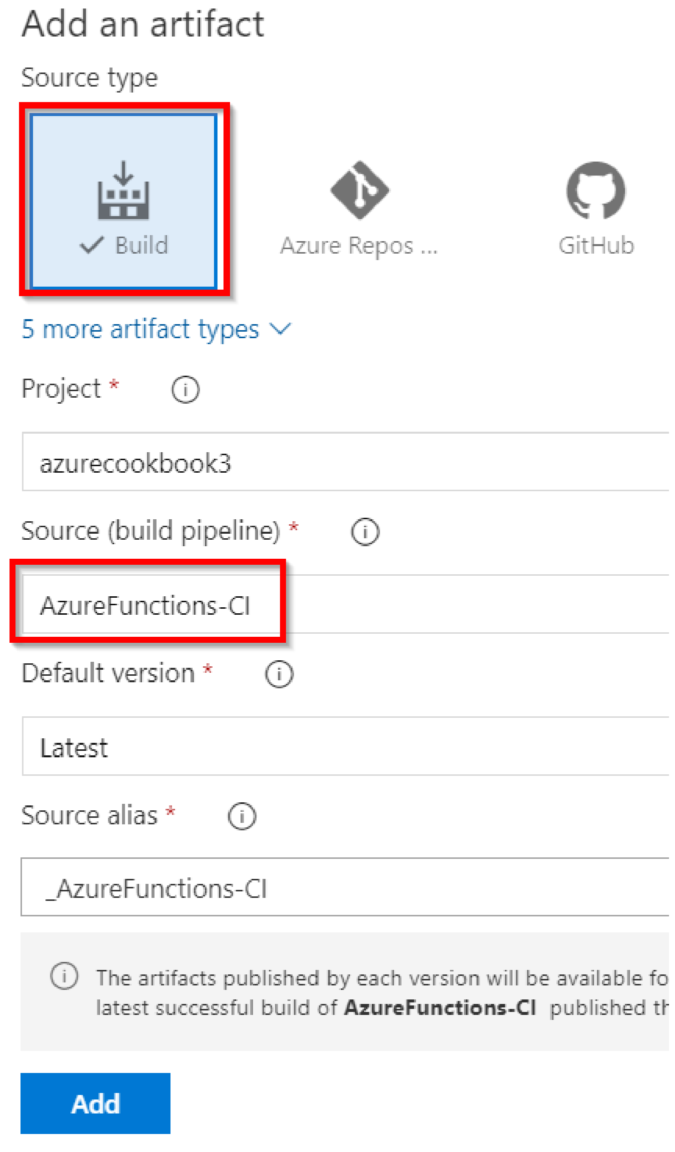


Figure 12.29: Azure DevOps—release pipelines—adding an artifact

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 job, 1 task** link, shown in *Figure* 12.30. Also, change the name of the release definition to **`release-def_stg`**:

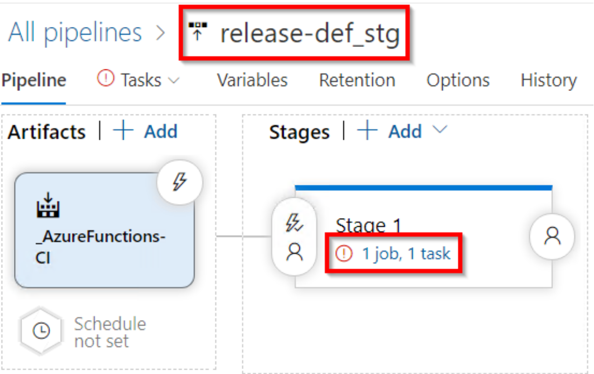


Figure 12.30: Azure DevOps—release pipelines—the configuration stage

7. You will be taken to the **Tasks** tab, shown in *Figure* 12.31. Provide a meaningful name in the **Stage name** field. I have provided the name **`Staging Environment`** for this example. Next, choose the Azure subscription to which you would like to deploy the Azure function. You will need to click on the **Authorize** button to provide the permissions:

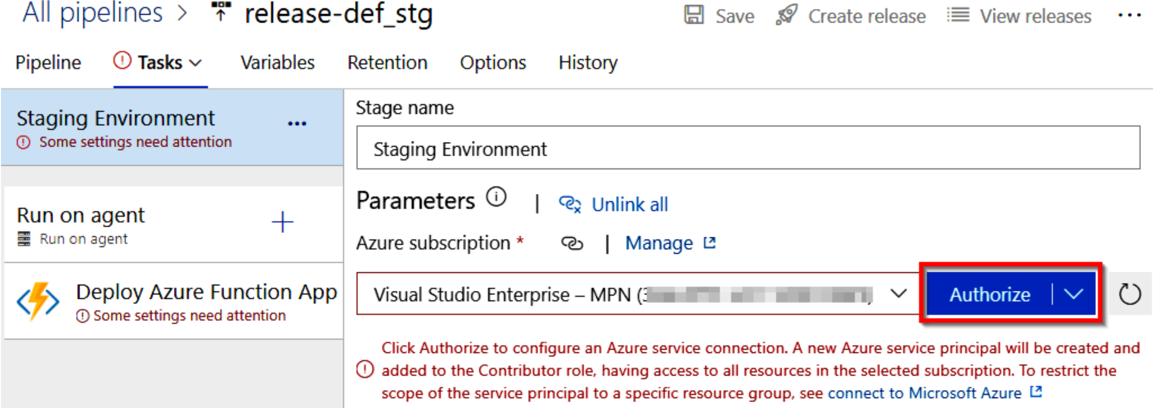


Figure 12.31: Azure DevOps—release pipelines—authorizing an Azure subscription

8. Once you authorize the account, you will see all the available Azure functions in the **App Service name** drop-down list, as shown in *Figure 12.32*. You can choose the one in which you would like to deploy the function app project:

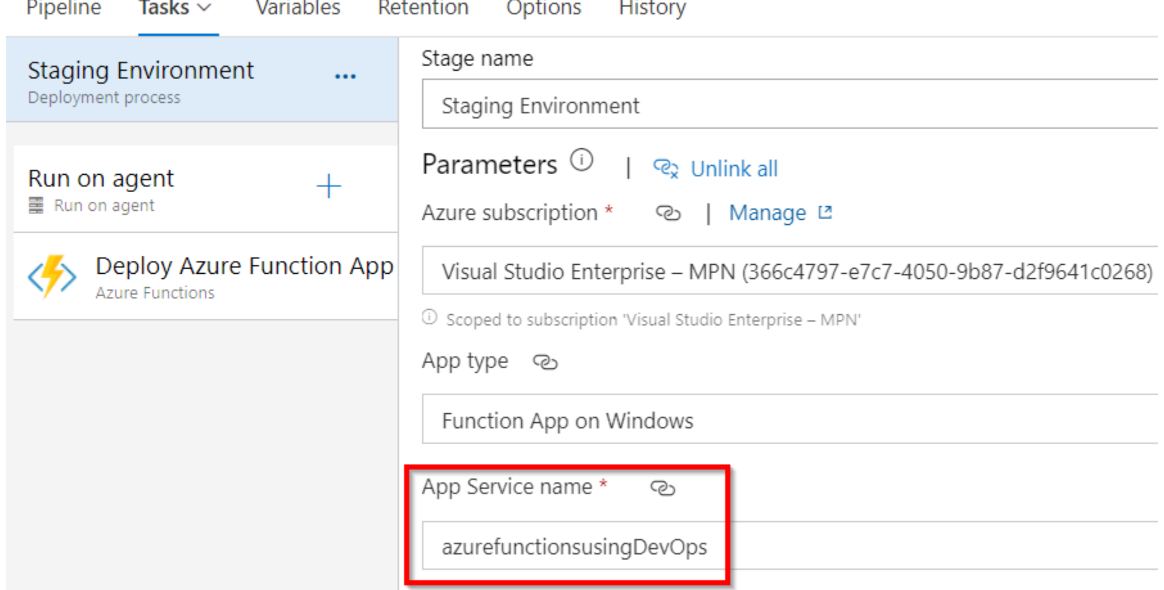


Figure 12.32: Azure DevOps—release pipelines—choosing the Azure Function App

9. Click on the **Save** button to save the changes. Now, let's use this release definition and try to create a new release by clicking on **Create release**, as shown in *Figure 12.33*:

Figure 12.33: Azure DevOps—release pipelines—Create release

10. Next, you will be taken to the **Create a new release** pop-up window 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 in *Figure* 12.34. Once you have reviewed it, click on the **Create** button to queue the release:

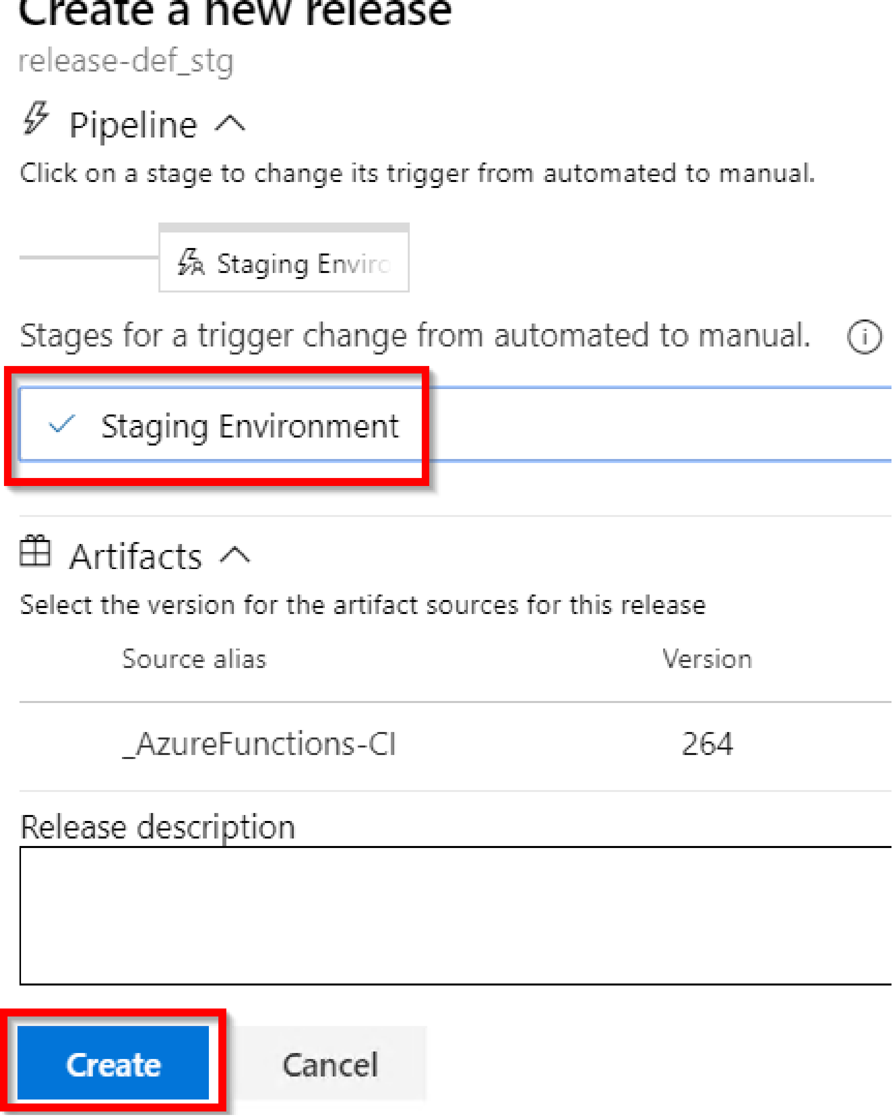


Figure 12.34: Azure DevOps—release pipelines—creating and configuring a release

11. Once the release is created, navigate to the **Pipeline** tab, as shown in *Figure* 12.35. Now, click on the **Deploy** button as shown here to initiate the process of deploying the release:

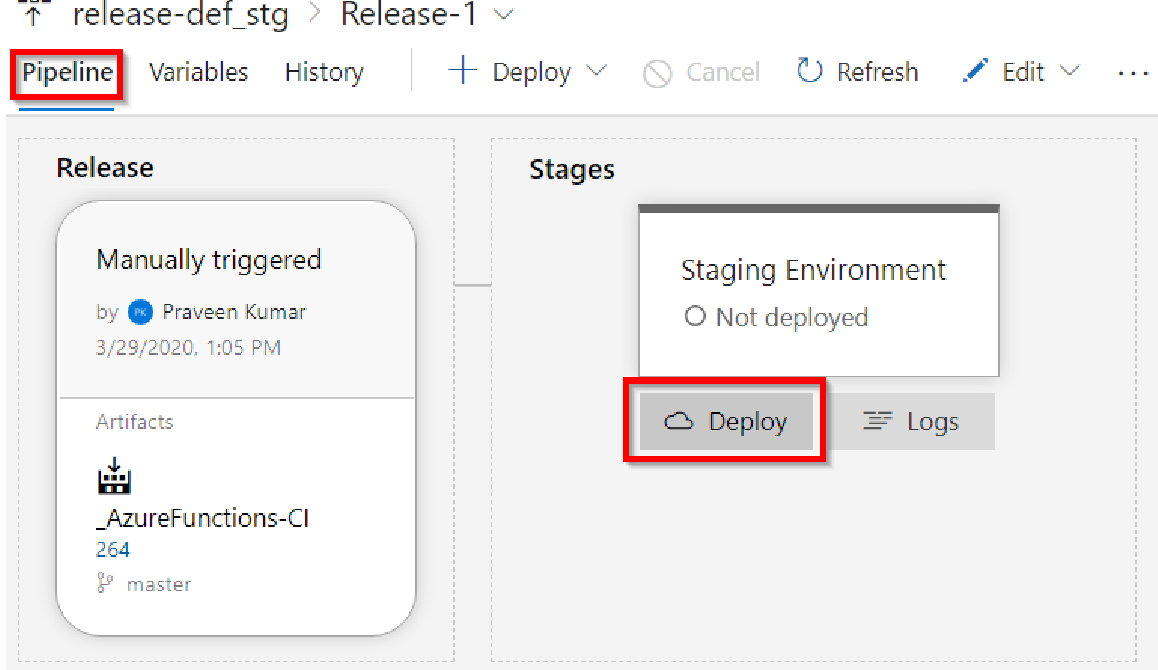


Figure 12.35: Azure DevOps—release pipelines—deploying the release

12. You will now be prompted to review the associated artifacts. Upon review, click on the **Deploy** button, as shown in *Figure 12.36*:

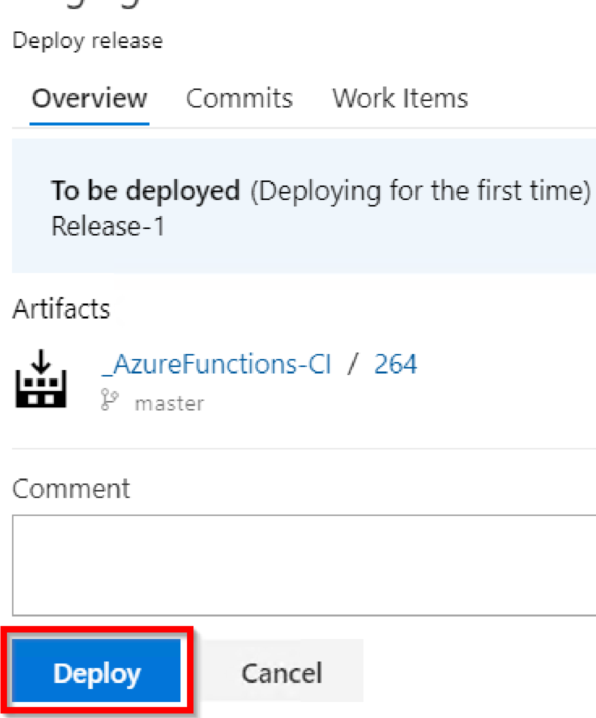


Figure 12.36: Azure DevOps—release pipelines—deploying and reviewing the release

13. Immediately, the process will start, and it will show **In Progress** to indicate the progress of the release, as shown in *Figure 12.37*:

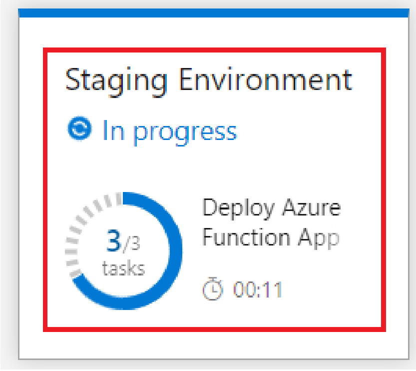


Figure 12.37: Azure DevOps—release pipelines—deployment progress

14. Click on the **In Progress** link shown in *Figure* 12.37 to review the real-time progress. As shown in *Figure* 12.38, the release process succeeded:

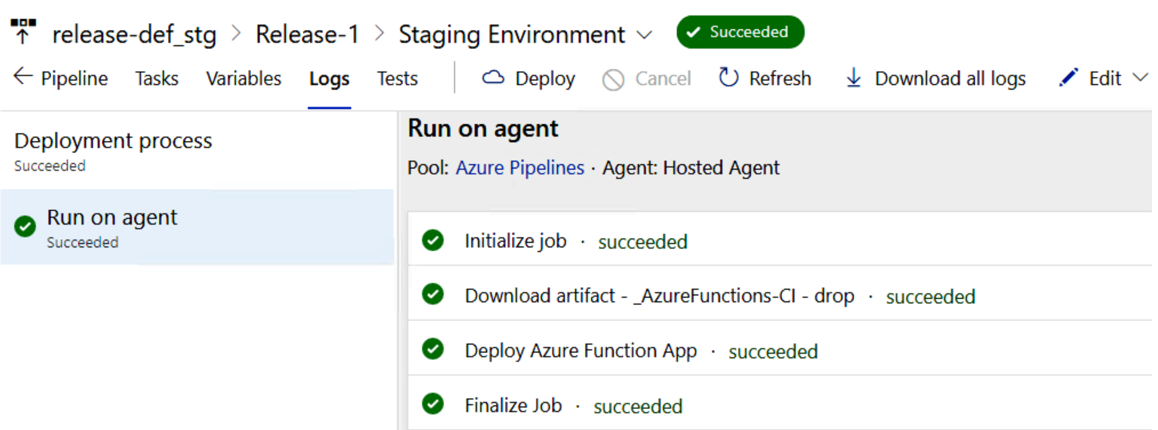


Figure 12.38: Azure DevOps—release pipelines—release summary

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, Developing Azure functions using Visual Studio*.

There's more...

While configuring continuous deployment for the first time, we may come across a button with the text **Authorize** in the **Azure App Service deployment** step. Clicking on the **Authorize** button will open a pop-up window, prompting for the Azure portal's credentials.

In this recipe, we have learned how to create and configure a release pipeline. Let's move on to the next recipe.

Triggering a release automatically

In this recipe, you will learn how to configure continuous deployment for an environment. In your project, you can configure development, staging, or any other pre-production environment and configure continuous deployment to streamline the deployment process.

In general, it is not recommended to configure continuous deployment for a production environment. However, this might depend on various factors and requirements. Be cautious and think about various scenarios before configuring continuous deployment for a production environment.

Getting ready

Download and install the Postman tool if it's not installed yet.

How to do it...

To configure continuous deployment, we'll 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** button, as shown in *Figure* 12.39:

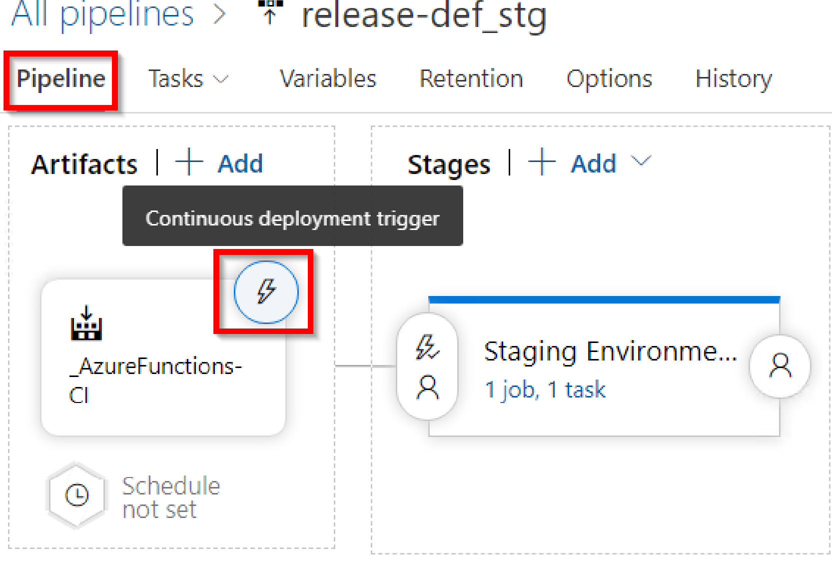


Figure 12.39: Azure DevOps—release pipelines—clicking on the Continuous deployment trigger button

2. As shown in *Figure 12.40*, enable the continuous deployment trigger and click on **Save** to save the changes:

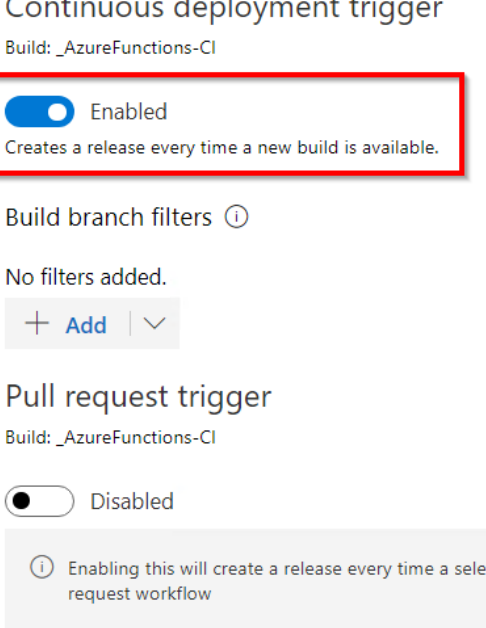


Figure 12.40: Azure DevOps—release pipelines—enabling the continuous deployment trigger

3. Navigate to Visual Studio and make some code changes, as follows:

```
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, commit the code with a comment – **Continuous Deployment**, to commit the changes to Azure DevOps. Soon after checking in the code, navigate to the **Builds** tab to see a new build get triggered, as shown in *Figure 12.41*:

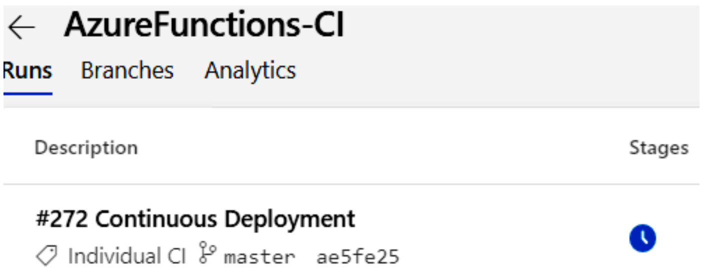


Figure 12.41: Azure DevOps—build pipelines—new build triggered automatically

5. Navigate to the **Releases** tab after the build is complete to see that a new release got triggered automatically, as shown in *Figure 12.42*:

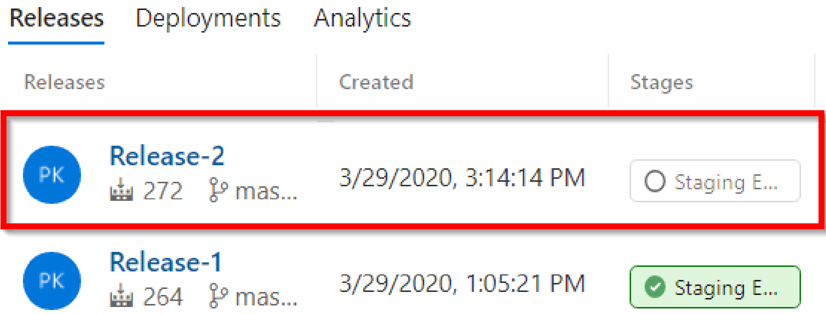


Figure 12.42: Azure DevOps—release pipelines—new release triggered automatically

6. Once the release process is complete, changes can be reviewed by making a request to the HTTP trigger using the Postman tool:

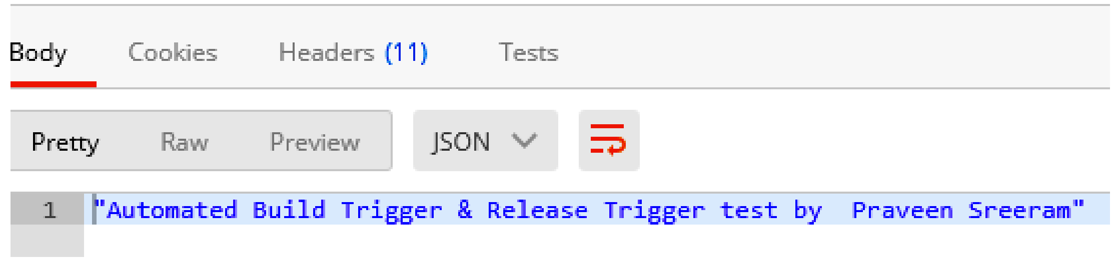


Figure 12.43: Azure DevOps—release pipelines—output in Postman

How it works...

In the **Pipeline** tab, we have enabled the continuous deployment trigger.

Every time a build associated with **AzureFunctions-CI** is triggered, `the release-def_stg` 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...

We can also create multiple environments and configure the definitions to release the required builds to those environments.

In this recipe, we have learned how to configure continuous deployment for Azure functions using release pipelines.

Integrating Azure Key Vault to configure application secrets

One of the major parts of any project is handling secrets in an efficient manner to adhere to organization-wide security guidelines. It's not advised to maintain secrets (such as passwords) in code or in files that are accessible to developers or any other stakeholders. In fact, these days, all the production environment details are only accessible by a few people and the secrets are managed by various systems. One such system in Azure is Key Vault. In this recipe, we'll learn how to leverage Key Vault to manage a secret that can be accessed by the code in a function app.

How to do it…

In this recipe, we will work on the following steps:

1. Creating a secret in the Key Vault service.
2. Configuring the Azure DevOps release pipeline.
3. Configuring the access policy.

Creating a secret in the Key Vault service

In this section, we will create a Key Vault service that can be used to manage secrets:

1. Create a Key Vault service as shown in *Figure 12.44*:

Figure 12.44: Creating a new Key Vault service

2. Once the Key Vault service is created, navigate to the **Secrets** blade, as shown in *Figure 12.45*:

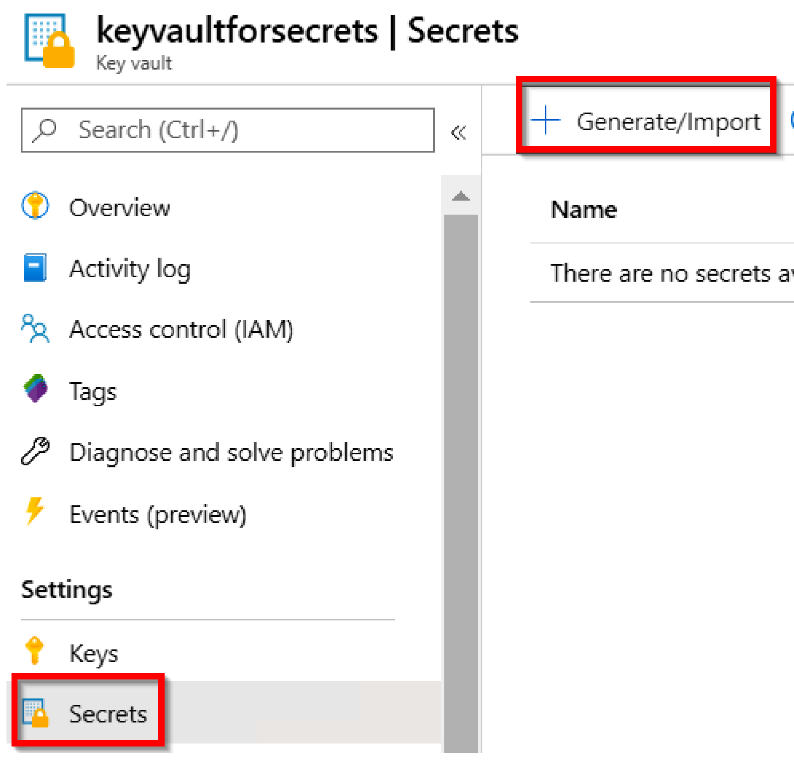


Figure 12.45: Key Vault—generating secrets

3. Click on the **Generate/Import** button to create a new secret.

4. Now, provide the **Name/Value** pair for the secret. The **Name** is the variable name that is used to refer to the secret and the **Value** is our actual secret to be used in the application. For example, the value would be a password or some other confidential value that needs to be stored in a secure place. As shown in *Figure 12.46*, provide the values, create a secret, and click on the **Create** button:

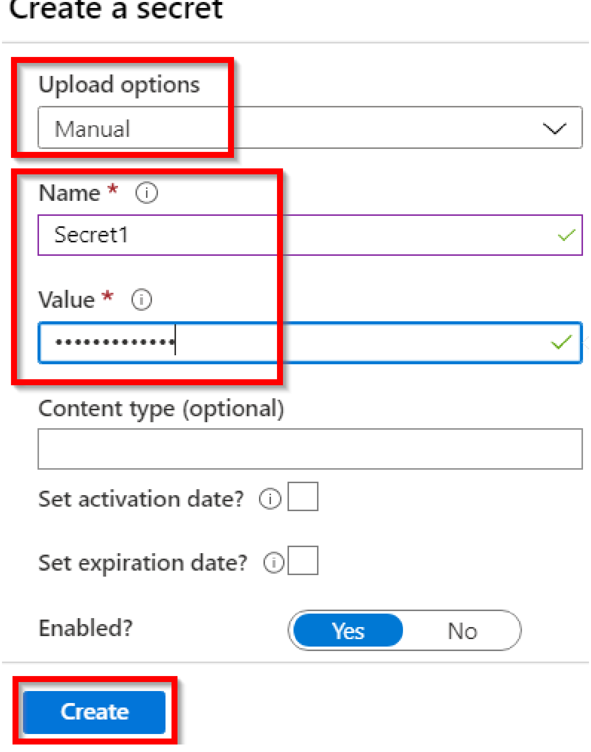


Figure 12.46: Key Vault—creating a secret

That's it. We have created a key vault service and also a secret value. We will be referring to these later in this recipe when we create an app setting in the Azure portal via the Azure DevOps release pipeline.

Configuring the Azure DevOps release pipeline

In this section, we'll learn how to download the secret(s) from the Key Vault service into the **Release** pipeline.

Perform the following steps in order to download the secrets from the Key Vault service:

1. Navigate to your **Release** pipeline and click on the **Edit** button:

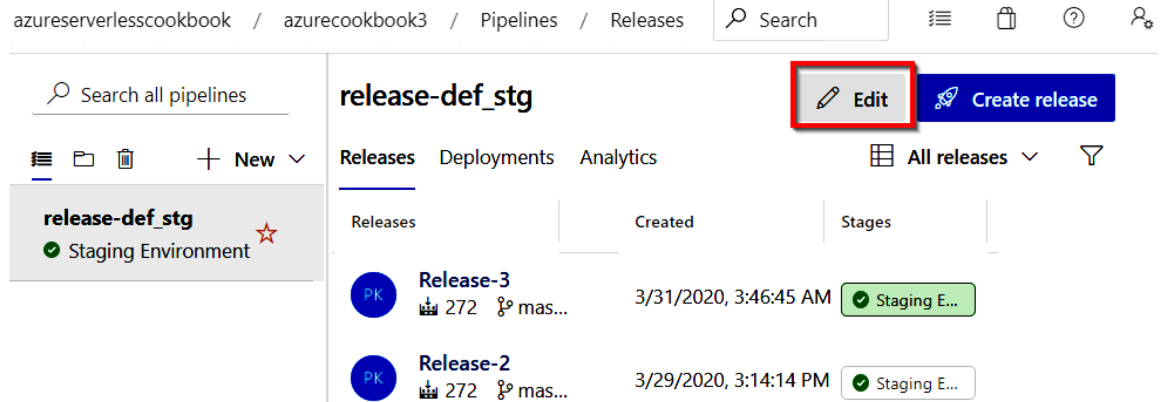


Figure 12.47: Azure DevOps—release pipelines—editing the release pipeline

2. In the **Pipeline** tab, click on the link to navigate to the **Tasks** tab:

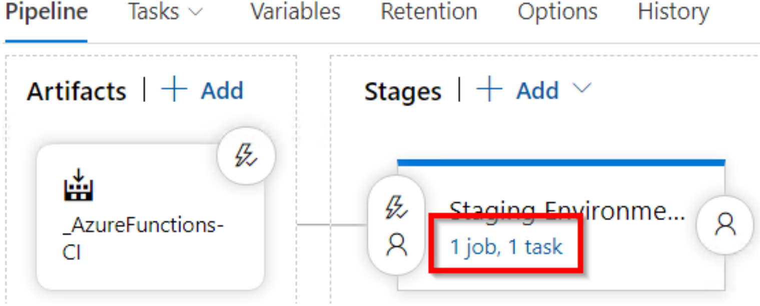


Figure 12.48: Azure DevOps—release pipelines—the editing stage

3. In the **Tasks** tab, click on the **Add** button to add the task to the pipeline as shown in *Figure* 12.49. This task downloads all the secrets from the Key Vault service. These downloaded values will be created as release variables, which can be referred to in any of the release steps. The variable names are the same as the names of the secrets. For example, we have created a secret with the name `Secret1`. So, we can refer to it as `$(Secret1)` in any of the steps in the release pipeline:

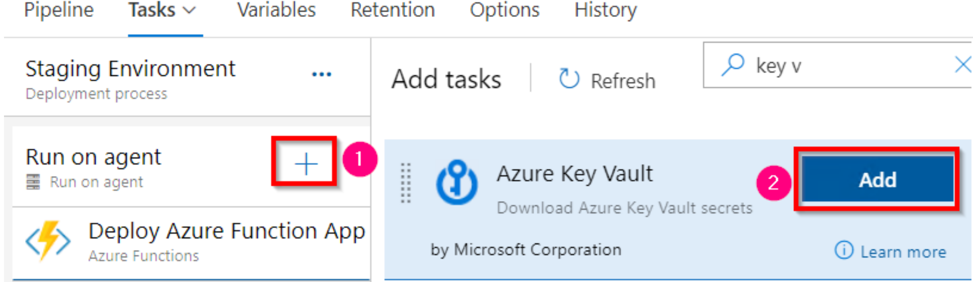


Figure 12.49: Azure DevOps—release pipelines—adding an Azure Key Vault secrets task

4. Now, in the **Azure Key Vault** task, choose the service connection (in the **Azure subscription** field) as shown in the following figure and also choose the name of the key vault service, and then provide a filter as per the project's requirements. Providing `*` would download all the secrets from the Key Vault service. Please make sure you change the order of the **Azure Key Vault** task to run before the **Deploy Azure Function App** task as shown in *Figure* 12.50:

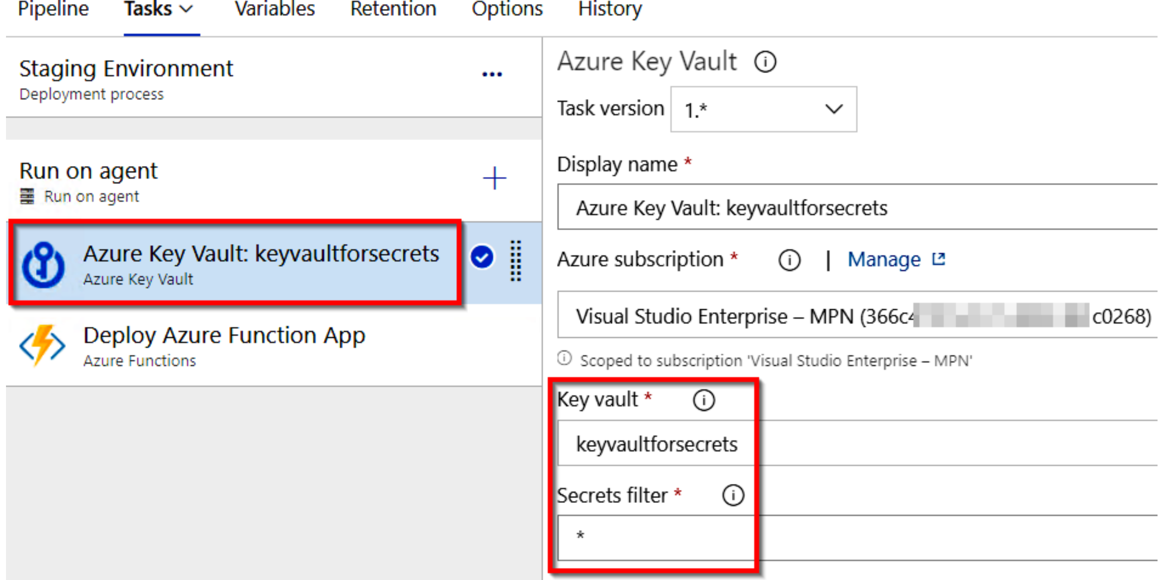


Figure 12.50: Azure DevOps—release pipelines—configuring the Azure Key Vault secrets task

5. Now, our goal is to use the secret variable and create an app setting inside the Azure Function app. Let's select the Azure function task and navigate to the **App Settings** section, then add a new key-value pair as shown in *Figure 12.51*. It will create a new app setting named `SecretKeyName` with the value that you have in the secret:

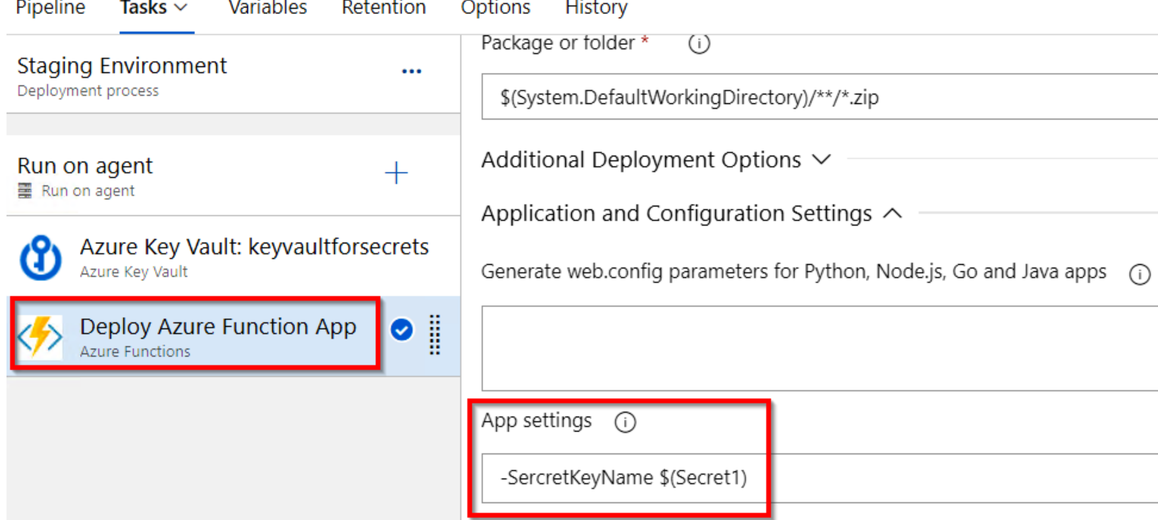


Figure 12.51: Azure DevOps—release pipelines—creating app settings in an Azure Function app

6. After reviewing the changes, please save them.

We are now done with integrating our DevOps pipeline with the Azure Key Vault service. However, it won't work as Key Vault is secured by default. So, we need to configure DevOps to access the Key Vault service explicitly. Let's do that now.

Configuring the access policy

In this section, we will learn how to configure the access policy:

1. Navigate to the **Access policies** blade and view the current policy that has access to the key vault. As you can see here, the username **Praveen** has access to the Key Vault service. Now, we need to provide access to the Azure DevOps service connection that we have created for our release pipeline:

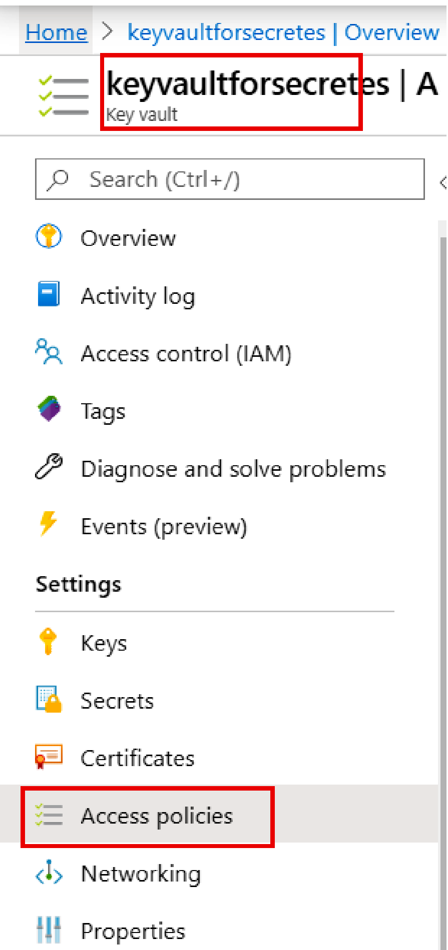


Figure 12.52: Key Vault—Access policies

2. In the **Access policies** blade, click on **Add Access Policy**, which takes you to another page for choosing the required policies:

   In the **Secret Permissions** field, choose **Get** and **List**.

   Click on **Select principal**.

3. In the **Principal** pop-up window, you have to search for the principal name as shown in *Figure* 12.53. It will be in the format `<azure devops organization name>-<Project name>-<Azure Subscription Id>`:

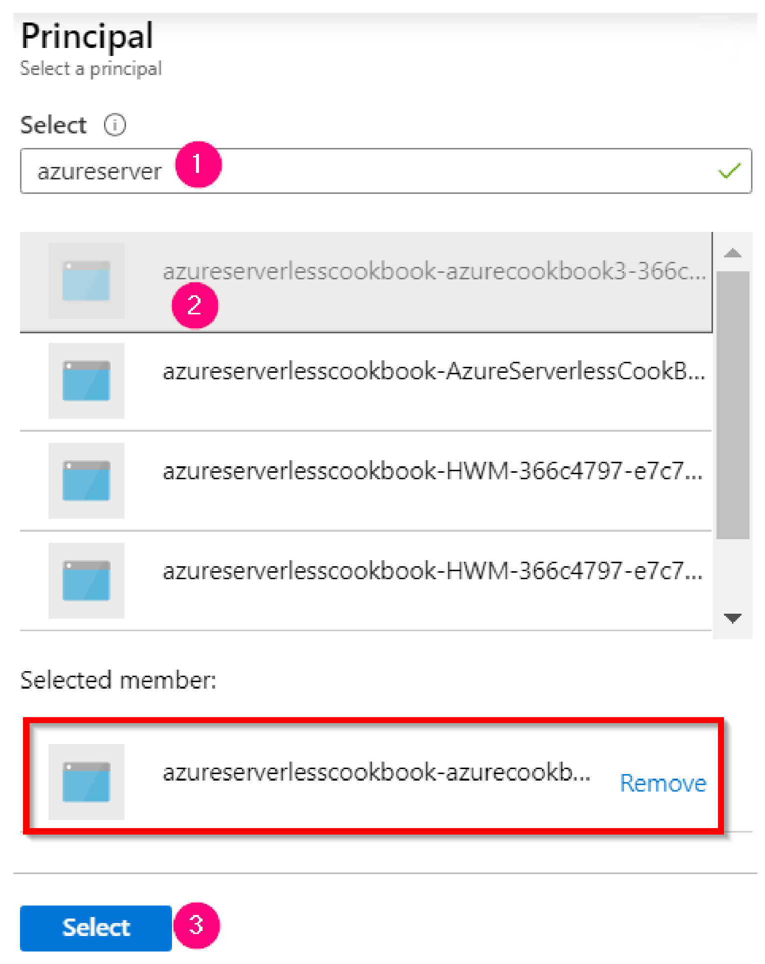


Figure 12.53: Key Vault—configuring access policies for the function app

4. After clicking on **Select**, the selected permission will be displayed, as shown in *Figure 12.54*:

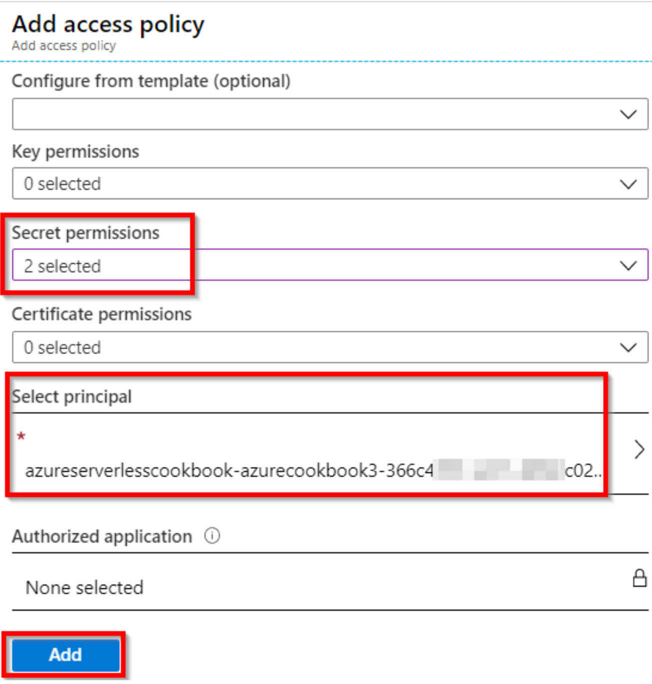


Figure 12.54: Key Vault—configuring access policies for the function app with permissions

5. In the preceding step, click on **Add**, which will configure the permissions between Azure DevOps and the Key Vault service as shown in *Figure 12.55*. Click on the **Save** button to save the changes:

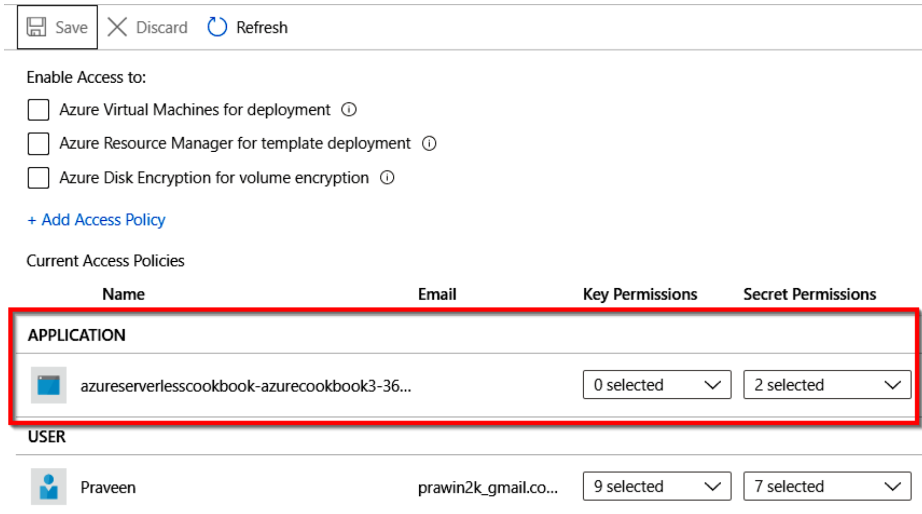


Figure 12.55: Key Vault—configuring access policies—viewing a function app with permissions

6. That's it. We are now ready to run the release pipeline. Go ahead to the release pipeline and run it. Now, navigating to the **Application settings** tab available in the **Configuration** blade of the Azure Function app should show the secret setting configured, as shown in *Figure 12.56*:

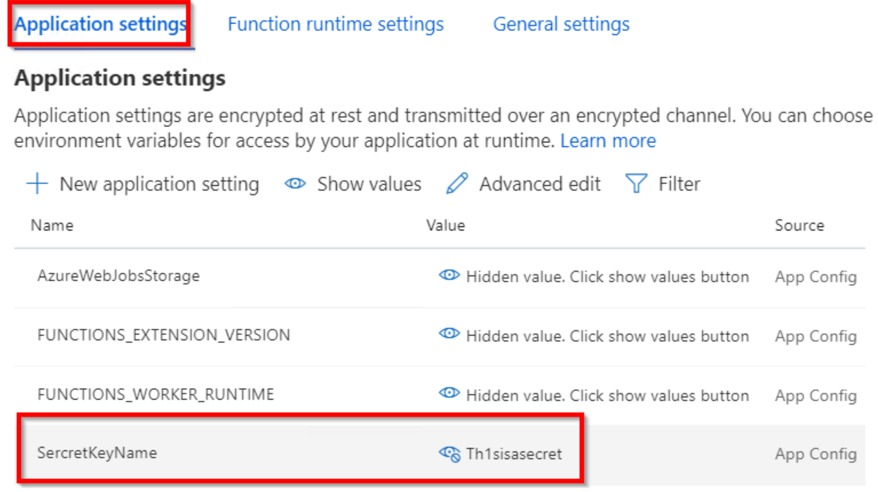


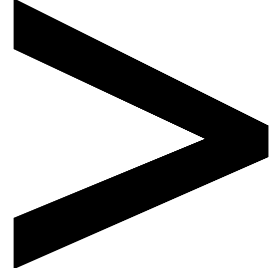
Figure 12.56: Azure Function app—app settings

How it works...

In this recipe, we have done the following:

1. We created a Key Vault service.
2. We created a secret in the Key Vault service.
3. In the Azure DevOps pipeline, we created a new task called **Azure Key Vault** that is capable of downloading secrets from the key vault depending on the filters specified in the task.
4. In the Azure Key Vault service, we configured an access policy by providing the read (**Get** and **List**) permissions to the service principal of the Azure DevOps service connection.
5. We created a key that refers to a secret variable, which is downloaded in the previous task (the **Azure Key Vault** task).
6. When we run the release pipeline, the new app settings get created in the **Configuration** blade of the Azure Function app.

In this chapter, we learned how to create a build pipeline and a release pipeline, and we learned how to configure continuous integration and continuous deployment for Azure functions using Visual Studio and Azure DevOps.



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

A

ad-admin: 307
addasync: 16-17, 34, 40, 61, 319
addcontent: 41-42, 44, 209
address: 29, 37-38, 42, 44, 118, 173, 177, 214, 292, 296-297, 299, 309, 311, 363
aiappid: 197-199, 208, 210, 220-221
aiappkey: 197-198, 208, 210, 220-221
algorithms: 56
allocated: 86, 266
analytics: 183, 193-195, 197-198, 204, 206, 212, 214, 219, 221-222
apipath: 198, 210, 221
artifact: 395, 399, 407-409
aspnet: 348
assemblies: 252
auditing: 55, 79
authlevel: 230
authoring: 92, 358-359
autopilot: 266
azure-: 360
Azure Blob Storage 1, 2, 3, 17, 18, 19, 21, 27, 43, 44, 45, 46, 58, 92, 98, 149
Azure Table Storage 1, 2, 3, 9, 10, 11, 13, 14, 15, 47, 56, 59, 60, 64, 359

B

bearer: 290-292
binaries: 354-356
binding: 13, 15, 17-18, 20-21, 29, 33-34, 36-37, 49, 59-60, 268, 318-319, 323, 371-372
blobname: 260-261

C

classes: 120, 328, 332
client: 3, 20, 28, 61, 88, 114, 147, 199, 209, 211, 227-228, 231, 233-234, 247-249, 254-258, 268, 270, 274, 277, 279, 281, 283, 285-288, 290, 292, 296, 298, 316, 373, 376, 378
cloudqueue: 322
clustered: 307
cognitive: 55-58, 60-61, 63-65, 109
config: 255, 348, 378, 380-382
connector: 68-69, 71, 73-74, 76
container: 15, 17, 19-20, 43, 45, 58, 62, 79-82, 84-86, 88-89, 100, 111, 121-122, 124, 128-130, 132-135, 137, 139-140, 142-143, 149, 151, 248-249, 251-254, 258, 260, 267, 270-271, 357
cosmosdb: 267-268, 270
countif: 220
csharp: 226, 246, 259
csvimport: 248-249, 252, 254-256, 258, 260, 263-264, 267-269

D

database: 79-80, 82, 86, 161, 239, 267, 271, 273, 300-305, 308-309, 339, 368
dataset: 97-101, 214-219, 222-223
debugger: 118, 124, 128, 131
deploy: 65, 124-125, 127-129, 132-133, 135, 139, 143, 163, 184, 334, 353-355, 358, 361, 386, 406-407, 409-410, 412-413, 417, 422
detect: 56, 124
devops: 112, 385, 387-418, 420-427
diagnose: 184, 186, 188
dimensions: 21, 24
directory: 249-250, 260, 273, 281-282, 284-286, 290, 292, 305, 313, 322, 395
docker: 132-140, 142-143
domain: 2, 226, 353, 362-367
dotnet: 136, 164
durable: 225-243, 245-247, 255-257, 259-260, 271

E

easyauth: 281
encrypt: 301, 378
endpoint: 57-58, 80, 143, 290
exception: 24, 166, 185-186, 188, 199, 211, 250, 302, 323

F

foreach: 60, 88, 96, 102-106, 270
function: 3-4, 6-7, 9-13, 15-22, 24-25, 29, 33-47, 49, 51, 53, 56, 58-60, 63-65, 75-79, 82-86, 88-90, 94-96, 105-106, 108, 111-125, 127-133, 135-137, 139-143, 145-148, 152, 154-158, 160, 162-167, 169, 174-175, 177-181, 184-192, 195-197, 199-202, 205, 208-209, 212, 219-220, 222, 227-229, 231-243, 246, 249-250, 252-260, 263-265, 268-270, 273-285, 289-291, 294-296, 299-301, 303, 306, 309-310, 313, 315-317, 319-332, 335-337, 340, 345-346, 348, 350, 353-355, 357-363, 365, 367-368, 371-380, 382-383, 399, 401, 407, 409-410, 418, 422-423, 425-427

G

gateway: 231, 300, 373-375
getasync: 198, 210, 221, 327
getbytes: 47, 52, 221
getinput: 240, 264-265
getpostman: 146, 227, 235, 237, 274
getsection: 347
getstring: 261
github: 177, 234, 360

H

hosting: 133, 141, 325
httpalive: 326-327
httpclient: 61, 198, 210, 220-221, 327
httpstart: 229, 231, 234-235, 256

I

ibinder: 46, 52
icollector: 320
identity: 273, 281, 300, 305-307, 309
ilogger: 7, 12, 16, 19, 23, 34, 38, 41, 44, 46, 52, 60, 84, 147, 178, 180, 191, 197, 208, 210, 220, 229, 240, 258, 263, 270, 301, 317, 319, 323, 327, 331, 350, 370
images: 1, 3, 17, 20, 25, 55-56, 58, 62-63, 133-134, 137, 240
import: 245-247, 255, 266, 294, 296, 351, 419
inbound: 292, 296-300
inject: 345, 348-349
inputjson: 7, 12, 16, 39, 41, 44, 47
insights: 145, 166-168, 171, 174-175, 183-184, 188-196, 199-209, 212-214, 219, 222-223, 359
instance: 45, 57, 88, 90, 94, 96, 108, 167, 191-192, 206, 226, 234-235, 259, 284, 294, 299, 326, 358
invoke: 3, 20, 38, 64, 74, 88, 171, 234, 255-257, 259-260, 264, 269, 338
isdeleted: 86
ispastdue: 220

J

json-based: 358
jtoken: 198, 210, 221

K

key-value: 9, 341-343, 345, 347-349, 423

L

leases: 86
libraries: 20, 22, 315, 328, 331-332
localhost: 118, 137, 286
logerror: 24, 198-199, 211
logging: 27, 43, 46, 138, 180, 196, 208, 219, 258, 325, 369
lookup: 96-104

M

methods: 148, 155, 230, 266, 320, 328, 332
metric: 194-196, 198-199, 202-205, 221
migrate: 316, 333, 339, 351
msgcontent: 209
mutate: 24

N

nodejs: 163

O

object: 14, 37, 39-40, 46-47, 180, 229, 232, 260, 262, 306-307
outbound: 296, 313
outputblob: 18-19, 44, 47

P

parameter: 11, 13, 15-18, 20-21, 37, 39, 42, 44, 58-59, 64, 71, 79, 103, 147, 166, 191-193, 231, 275-276, 298, 317-318, 320, 325, 372, 377
pipeline: 87-89, 91-92, 96, 102, 105, 107-109, 385, 389-390, 393-397, 399-403, 405-406, 408, 412, 414-415, 417-418, 420-424, 427
plugging: 87
plumbing: 112, 115
poison: 324-325
postasync: 61, 221
postdata: 221
postman: 146-148, 227, 235-237, 241, 274-276, 284-287, 290, 308, 317, 415, 417
powerbi: 214
powershell: 315, 332-333, 335, 338
proxies: 353, 372-373, 375-376, 378

Q

queues: 1, 3, 14-15, 18-19, 146, 153, 316, 325

R

rand-guid: 18, 20
readblob: 260, 263
registry: 133-135, 142-143
release: 129, 136, 329, 334, 385-386, 404, 406-418, 420-424, 427
repository: 133-134, 387, 391, 394-395
resultjson: 198, 210-211, 221
rowkey: 12, 46-47, 60

S

sendgrid: 27-33, 36-39, 41-42, 45-46, 48, 88, 108, 205-206, 208, 212
serverless: 2-3, 55, 74, 79, 87, 133, 209, 225-226, 231, 246-247, 316, 326, 353-354
simulate: 193, 237, 241, 317, 333
sqlclient: 301, 303
staging: 145, 155, 157-160, 162, 354, 386, 409, 414-415
status: 8, 13, 16, 120, 148, 175, 234-236, 241-242, 259, 305, 349, 398, 405
subsets: 353, 372
swapping: 156, 159

T

telemetry: 166, 183-184, 188, 190, 193, 195, 199-200, 205, 209, 212-213
template: 6, 18, 21, 35, 59, 113, 149, 152, 164, 231, 233, 235, 238-240, 252, 256, 323, 336, 359-361, 373, 377, 388, 392-393, 395-396, 407
testappid: 198-199
text-align: 209-210
threading: 180, 197, 208, 219, 249, 261, 303
throttling: 273, 292, 296-297
timestamp: 192, 197, 202, 207, 211, 220
traces: 191-193
tracking: 166, 188
traffic: 166, 190, 326-327, 373

trigger: 1, 3, 6-10, 12-13, 15, 17-18, 20-21, 25, 28, 33-36, 38, 40-41, 58, 60, 66, 68, 73, 75, 78, 82-86, 88-89, 107, 114-116, 118-124, 129-130, 132, 136, 146-149, 152-156, 164, 166, 168, 177-179, 185, 187, 190-193, 196-197, 203, 205, 208-209, 212-213, 219-220, 223, 229, 231, 238, 245-247, 252, 254-260, 263-265, 268-270, 274-276, 278, 284-285, 291, 294, 298, 300-301, 303, 308-309, 312-313, 316-317, 320-321, 323, 325-327, 330-331, 333, 336-340, 345-346, 348-351, 355, 357, 373-374, 396, 399, 401, 403, 406, 415-417
tweets: 65-66, 68, 73-74
twilio: 27-28, 48-53, 108
twitter: 65-66, 68, 72-74, 78

U

uploadblob: 249-250
urifactory: 268, 270
utilities: 115, 328, 330-331

V

validating: 145-146, 166, 188, 207
variable: 39, 44, 46, 78, 119, 129, 395, 420, 422-423, 427

W

webclient: 19
webjobs: 46, 52, 124, 196, 208, 219, 229, 232-233, 238-240, 258, 267, 303, 369
website: 20, 29, 31, 44, 166, 195, 212-213, 243, 278, 316, 357-358
whitelist: 309, 311, 313
wwwroot: 136, 329, 332, 354-355

X

x-api-key: 198, 210, 221
x-ms-app: 198, 210, 221

Y

youtube: 53