Beginning Azure Functions

Building Scalable and Serverless Apps

Rahul Sawhney

Apress®

Beginning Azure Functions

Building Scalable and Serverless Apps

Rahul Sawhney

Apress®

Beginning Azure Functions: Building Scalable and Serverless Apps

Rahul Sawhney
Hyderabad, India

ISBN-13 (pbk): 978-1-4842-4443-2 ISBN-13 (electronic): 978-1-4842-4444-9
https://doi.org/10.1007/978-1-4842-4444-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Matthew Moodie
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4443-2. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to my parents, Ashwani Kumar Sawhney and Neha Sawhney. Without their sacrifices, I wouldn't have achieved what I have in life.

Also, I would like to dedicate this book to my wife, Kulpreet, for always standing by my side and supporting me during hard times.

Table of Contents

About the Author

Rahul Sawhney works as a software developer with Microsoft, India, and has more than five years of experience delivering cloud solutions using technologies such as .NET Core, Azure Functions, microservices, AngularJS, Web API, Azure AD, Azure Storage, ARM templates, App Service, Traffic Manager, and more.

He is a Microsoft Certified Azure Developer and Architect. He loves learning new technologies and is passionate about Microsoft technologies. In his free time, he loves playing table tennis, watching movies, and reading books.

You can reach Rahul at `rahulsawhney2206@gmail.com` or `www.linkedin.com/in/rahul-sawhney-2206`.

About the Technical Reviewer

Vidya Vrat Agarwal is a software architect, author, blogger, Microsoft MVP, C# Corner MVP, speaker, and mentor. He is a TOGAF Certified Architect and a Certified Scrum Master (CSM). Currently working as a principal architect at T-Mobile in the United States, he started working on Microsoft .NET with its first beta release. He is passionate about people, process, and technology, and he loves to contribute to the .NET community. He lives in Redmond, Washington, with his wife Rupali; two daughters, Pearly and Arshika; and a puppy girl, Angel. He blogs at `www.MyPassionFor.Net` and can be reached by e-mail (`vidya_mct@yahoo.com`) or on Twitter (`@dotnetauthor`).

Acknowledgments

I must start by thanking my girlfriend and now wife, Kulpreet, for always being there and supporting me during my struggling days and for always believing in me. I could not have written this book without her support and motivation.

My heartfelt thanks to Manas Mayank and Kidar Garg who introduced me to Microsoft Azure. They constantly mentored and guided me during my early days of learning cloud technologies. They helped me a lot by giving me complex work, and they always trusted in me. They not only changed my thought process but instilled a growth mind-set in me by encouraging me to try new technologies during this journey.

I am highly indebted to my younger brother, Sanjay, and my childhood friends, Saurabh Trivedi and Gajendra Raikwar, because they always trusted in my abilities and pushed me to work hard.

I would also like to thank my managers at Microsoft (Subhavya Sharma, Anil Emmadi, Manish Sanga, and Jaydeep Baliram Sawant) for always encouraging me to try new things and supporting and guiding me. They helped me shape my career as well as guided me on the right path.

I would also like to thank my colleagues at Microsoft, (Rishabh Verma, Mohit Garg, Subhendu De, Prashant Jain, Mehul Gardi, Binay Prasad, Sanyam Seth, Archit Shukla, Harshit Agarwal, Abhishek Somani, Sidharth Mittal, and Dinesh Kumar Reddy) for their zeal to learn new technologies. Each one of you has taught me something about new technologies, and the culture you create of learning and sharing is what makes work effortless.

Thanks to the team at Apress (Smriti Srivastava, Shrikant Vishwakarma, and Matthew Moodie) for giving me this wonderful opportunity and making this a memorable journey. Thanks to Vidya Vrat

and Matthew Moodie for providing their valuable technical reviews, which has helped me to improve the book.

Lastly, I would like to thank all the readers of this book. Please feel free to share your valuable feedback about this book, which will help me to deliver better content in the future. I look forward to all your feedback and suggestions.

Introduction

Get ready to create highly scalable apps and monitor functions in production using Azure Functions 2.0!

The book starts by taking you through the basics of serverless technology and Azure Functions and then covers the different pricing plans of Azure Functions. After that, you will dive into how to use Azure Functions as a serverless API. Then, you will learn about the Durable Functions model and about disaster recovery and georeplication.

Moving on, you will encounter lots of practical recipes with hands-on steps for creating different types of functions in Azure Functions using Azure Portal and Visual Studio Code. Finally, I will discuss DevOps strategy as well as how to deploy Azure Functions and get Azure Functions production-ready.

By the end of this book, you will have all the skills needed to work with Azure Functions, including creating durable functions, deploying functions, and making them production-ready by using telemetry and authentication/authorization.

What This Book Covers

Chapter 1 goes through the basics of serverless computing and talks about Azure Functions. It compares Azure Functions to WebJobs so you understand the difference between them. I also talk about the different pricing plans of Azure Functions.

In Chapter 2, you will create first Function using Azure Portal and then using Visual Studio Code. I will also talk about the Azure Functions file hierarchy, configuration, and settings.

In Chapter 3, you'll learn about triggers and bindings. I will also discuss changes to Azure Functions 2.0 bindings. You will create Blob Storage–triggered Azure Functions.

Chapter 4 will go through the differences between monolithic applications vs. microservices. Then, I will talk about how you can convert a monolithic application to microservices using Azure Functions. You will create some functions and then learn about proxies.

In Chapter 5, you will start with overview of the Durable Functions pattern and bindings. You'll also learn about performance and scaling in a durable function. You will create your first durable function and learn about disaster recovery and geo-replication.

In Chapter 6, you will look at deploying functions to Azure, first using a CI/CD pipeline and then using ARM templates.

In Chapter 7, you will look at the built-in logging capabilities of Azure Functions. Then, you will look at Application Insights and how it can be used to monitor Azure Functions. Then, I will talk about securing Azure Functions using Azure Active Directory and how to configure cross-origin site scripting (CORS) in Azure Functions.

Let's get started!

CHAPTER 1

Introduction to Azure Functions

In the software industry, we are now in an era where everything we develop is oriented toward the cloud. To help developers achieve more productivity, cloud platforms such as Microsoft Azure, Amazon Web Services, Google Cloud Platform, and so on, implement a concept known as *serverless computing*. With serverless computing, companies and developers can concentrate on developing products rather than worrying about the maintenance and administration of the server.

Azure Functions is one such product for serverless computing. Before going into Azure Functions, I'll talk about serverless computing and what it means.

In this chapter, I will cover the following topics:

- Overview of serverless computing
- Overview of Azure Functions
- Azure Functions vs. Azure WebJobs
- Azure Functions pricing options

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_1

Overview of Serverless Computing

Serverless computing is also known as *function as a service* (FaaS) and is one of the current buzzwords of the tech industry. Serverless computing does not mean your code runs without a server; it means you don't have to take care of the server maintenance, including patching, upgrading, and so on. The servers will be managed by a cloud service provider such as Amazon, Microsoft, Google, and so on, and you have to take care of managing your code/application. With serverless computing, you pay only for the time your code runs or executes. Also, the cloud service provider takes care of scaling and load balancing, which is a win-win situation for both the cloud service provider and you because you can dedicate the majority of your time to doing what's most important: developing the code/application. The cloud service provider maintains and owns the server and bills you for the use of it.

Serverless computing is a paradigm shift in computing. Deploying applications or code used to take months with physical machines. With serverless computing, deploying takes just a millisecond. This has changed the IT world drastically.

Now, what is Azure Functions?

Overview of Azure Functions

With Azure Functions, you can start writing your application code without worrying about the application architecture and infrastructure required to run the application. Azure Functions also provides the capability to scale as needed. So, if the load is high, you can expect Azure Functions to scale and cater to the high load. Also, with Azure Functions you pay only for the time your code runs, so if the load on the application is low, you pay less.

Here are some important features of Azure Functions:

- **Browser-based interface**: You can write and test your code directly in the interface without using any integrated development environment (IDE).
- **Programming languages**: Azure Functions supports many languages such as C#, JavaScript, F#, Java, Python, TypeScript, PHP, Batch, Bash, PowerShell, and a few other experimental languages.
- **Seamless integration with third-party apps**: Azure Functions integrates seamlessly with third-party apps such as Facebook, Google, Twitter, Twilio, and other Azure services like CosmosDB, Azure Storage, Azure Service Bus, and more. You can also integrate existing apps using triggers and events.
- **Continuous deployment**: Azure Functions supports continuous deployment through Azure DevOps (VSTS), GitHub, Xcode, Eclipse, and IntelliJ IDEA.

As you can see, Azure Functions possesses some unique capabilities that not only enhance your productivity but provide lots of different options for developers to choose from. Still, I have heard developers getting confused about when to use Azure Functions and when to use

Azure WebJobs. The primary reason for this confusion is that developers have traditionally reduced the load on the application by doing extensive and time-consuming computations in Azure WebJobs.

In the next section, you will learn about the differences between Azure Functions and Azure WebJobs and in which scenario you should use each of them.

Azure Functions vs. Azure WebJobs

Azure Functions and Azure WebJobs are both code-first integration services that were designed for developers. Both support features such as authentication, Application Insights, and source control integration.

Azure Functions has the following features that Azure WebJobs does not:

- Serverless app model with auto scaling
- Development and testing in the browser
- Azure Logic Apps integration
- Pay-per-use pricing model
- Many triggers in version 2.0 such as Queue, Event Grid, HTTP, Timer, and so on

For most scenarios, Azure Functions is the best choice because it offers many programming languages, many pricing options, and greater developer productivity. However, the following are two scenarios where you should use WebJobs instead:

- You have an App Service environment where you want to run some code snippet and maintain the same DevOps pipeline and environment.

- You want to customize JobHost behavior in the `host.json` file such as having a custom retry policy for Azure Storage.

Azure WebJobs runs under the Azure App Service model, whereas for Azure Functions you have different pricing models that give you more control over pricing. You'll learn about pricing next.

Azure Functions Pricing Plan

Azure Functions supports two pricing plans.

- Consumption Plan
- App Service Plan

Let's look in detail at both plans.

Consumption Plan

With Azure Functions' Consumption Plan, you pay only when your code is executing. This helps you save significantly over the App Service Plan or when using a virtual machine. For example, if you have a weekly newsletter for your web site, instead of using WebJobs, you can use Azure Functions and save an enormous amount of money.

The metric used for calculating price in Azure Functions is gigabyte-second (GB-s). This metric calculates the memory usage and total execution time for billing. It is billed based on per-second resource executions and consumptions.

In the Consumption Plan, you are granted 1 million requests and 400,000 GB-s of resource consumption for free per month per subscription across all Azure Functions apps in that subscription.

App Service Plan

Azure Functions' App Service Plan utilizes the same App Service Plan used for hosting a web site, the Web API, and so on. With the App Service Plan for Azure Functions, instead of paying for the duration when a function is executing, you pay for the reserved resources of the underlying virtual machine (VM). This makes the App Service Plan costlier than the Consumption Plan.

Why do some companies use the App Service Plan? The reason is that in the Consumption Plan, functions have a time limit of five minutes, so if your Azure Functions code runs for more than five minutes in a single execution, it will be timed out, whereas there is no time limitation for Azure Functions in the App Service Plan. So, in the App Service Plan, Azure Functions is as good as WebJobs.

Also, when you are billed on the App Service Plan, it is easier to maintain the monthly quotas of your company because all the resources are under the same App Service Plan.

However, if you have a piece of code that is resource hungry, then having it on the same App Service Plan as the rest of your company would actually make your other applications vulnerable because Azure Functions would be using the same shared resource and thus would make the other applications run slowly.

With this we have now come to the end of Chapter 1 and we now have basic understanding of Serverless, Azure Function and App Pricing. Let's now move to Chapter 2 where we will build onto the learnings of this chapter.

CHAPTER 2

Creating Functions in Azure Functions

In this chapter, you will start using Azure Functions. Microsoft has recently released Azure Functions version 2.0, so I will be using Azure Functions 2.0 throughout the book's examples.

In this chapter, I will cover the following topics:

- Creating functions using Azure Portal
- Creating functions using Visual Studio Code
- Checking out the file hierarchy, configuration, and settings of Azure Functions

Let's look at both ways you can create functions with Azure Functions.

Creating an Azure Function Using Azure Portal

In this section, you will create your first function using Azure Portal. Here you will be creating a "Hello, world" application completely using Azure Portal.

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_2

Creating an Account on Azure Portal or Logging into Azure Portal

First you need to log into Azure Portal. You can go to http://portal.azure.com to sign in. If you don't have an Azure subscription or you are first-time Azure user, you can get an Azure account for free for 12 months. Visit https://azure.microsoft.com/en-us/free/ to get started on Azure.

Creating Your First Function App Using Azure Portal

Once you have completed the sign-up/sign-in process, you will be taken to the Azure Portal Dashboard. Now you can create a function app.

1. On the Dashboard, click "Create a resource" in the left panel. In the menu that opens, click Compute and then Function App, as shown in Figure 2-1.

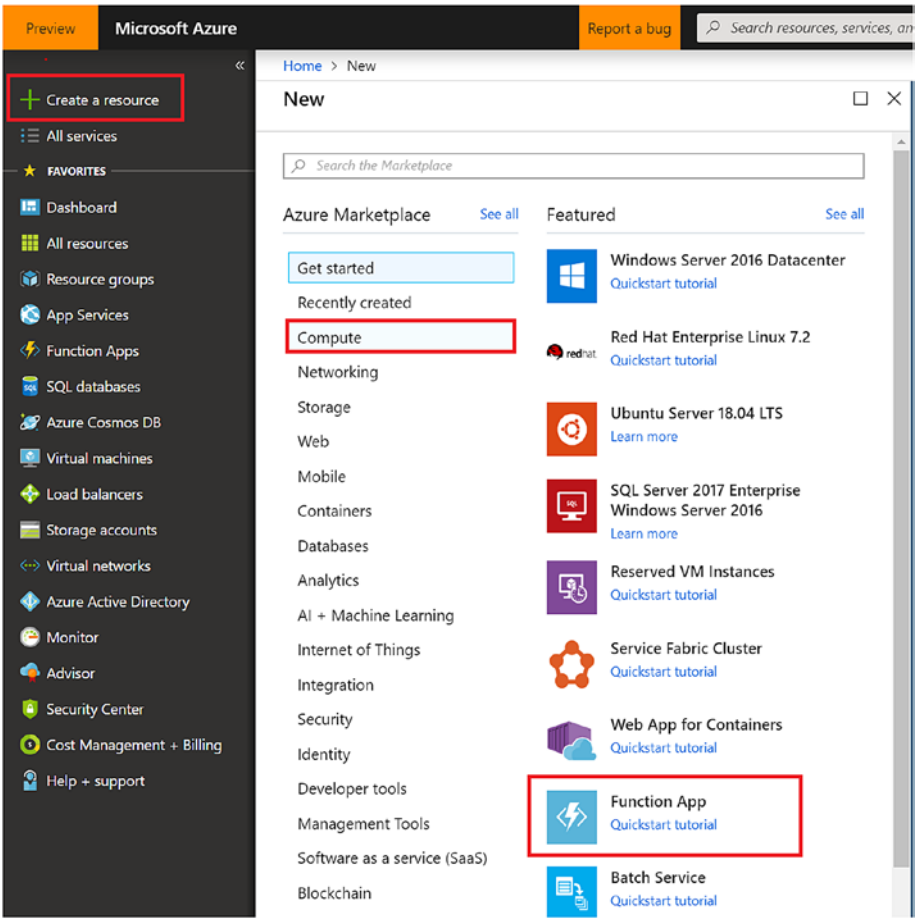

Figure 2-1. *Creating Function App resource*

2. Once you have clicked Function App, you will be asked to provide certain details such as the app name, resource group, OS, hosting plan, and so on. Refer to Figure 2-2 to fill in these settings.

    - **App Name**: This is the name of the function app. The app name in this example is `building-azure-function`, so the URI will be `building-azure-function.azurewebistes.net`.

- **Subscription**: This is the subscription under which the function app will be created. An Azure account can have multiple subscriptions that are used for maintenance and billing purposes.
- **Resource Group**: You can create a new resource group or use an existing one. In Figure 2-2, I am creating a new resource group named `building-azure-function`. A resource group is like a container that holds the resources required to run that solution.
- **Hosting Plan**: By default, Consumption Plan is selected, but you can choose App Service Plan. To better use Azure Functions and get a lower cost, go for the Consumption Plan.
- **Location**: Always try to choose the nearest location to where you expect the majority of the traffic to come for your function app.
- **Storage**: Every function app requires storage. You can either create new Azure storage or select existing storage. In Figure 2-2, I am creating new Azure storage.

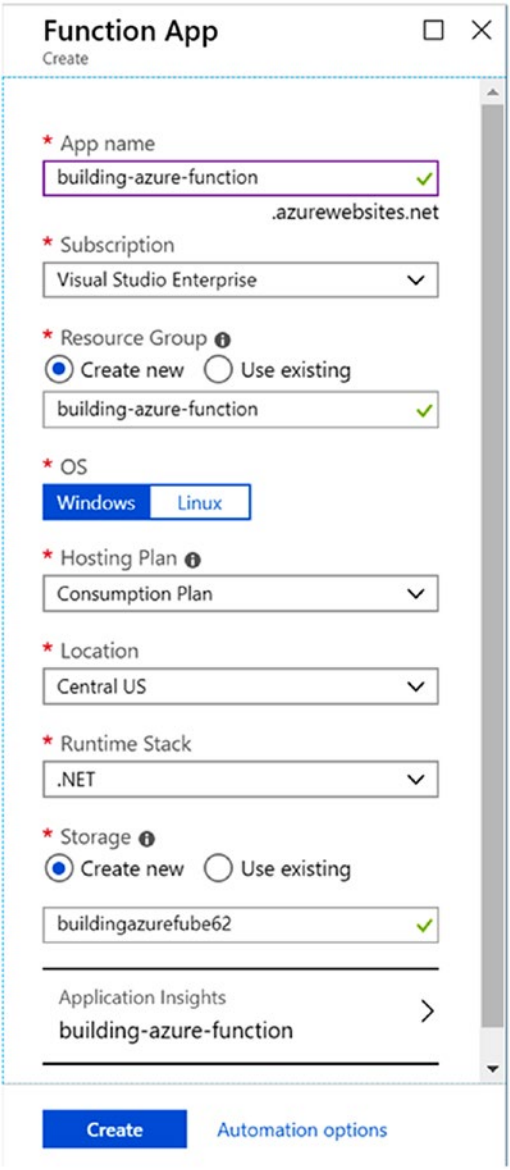

Figure 2-2. *Create Function App*

3. Click the Create button to provision the new function.

4. You can check the status of your function by clicking the bell icon at the top, as shown in Figure 2-3.

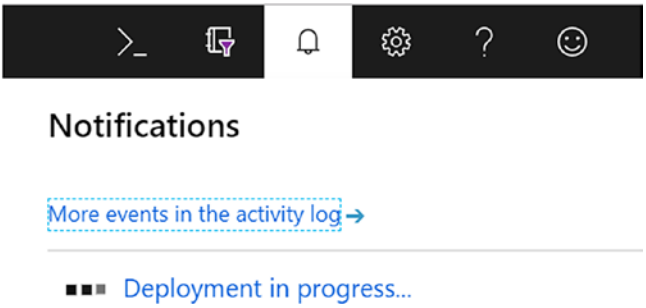

Figure 2-3. *Checking the status*

5. Once the deployment is completed, your first Azure Functions app, called `building-azure-function`, is ready and has been deployed, and you can now start coding for it. To go to the function app you created in the previous steps, please refer to Figure 2-4.

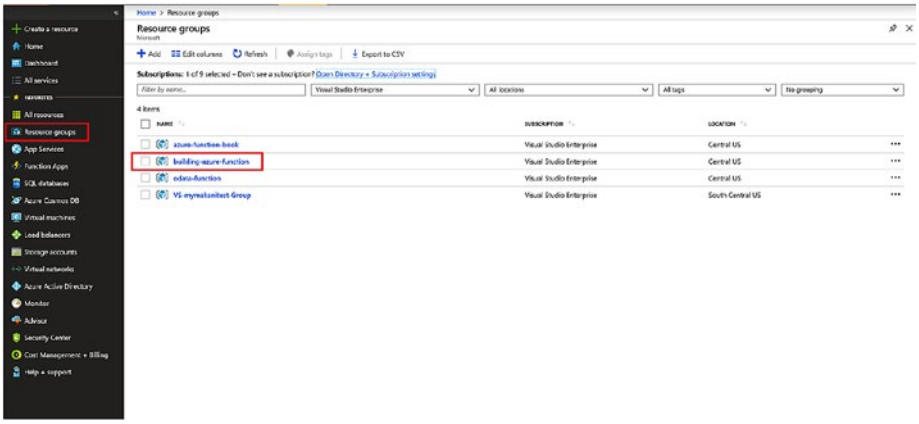

Figure 2-4. *Select the Resource Group you created*

Creating Your First Function in the Function App

Your function app is ready and deployed, but it is not usable yet because you don't have any function inside the function app. A function is the core piece of the function app where you code your logic.

1. To create your first function, click `building-azure-function`, as shown earlier in Figure 2-4. Then click the `building-azure-function` app service, as shown in Figure 2-5.

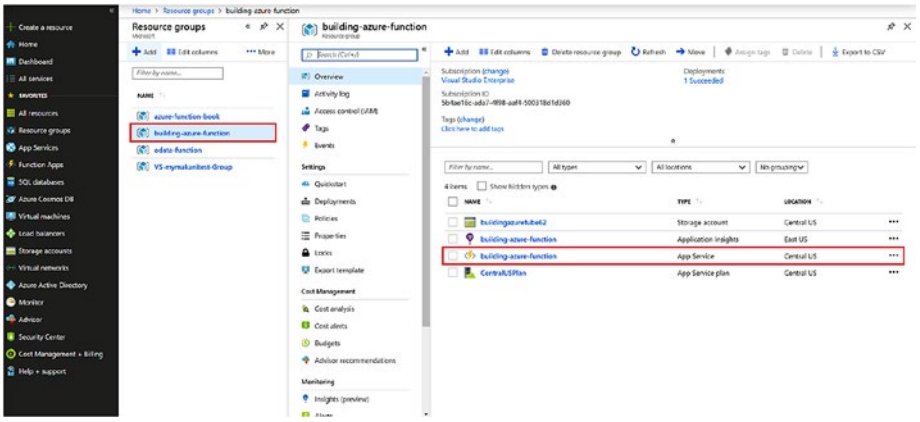

Figure 2-5. *Opening the app service*

2. Click the + icon beside Functions and select "in-portal," as shown in Figure 2-6.

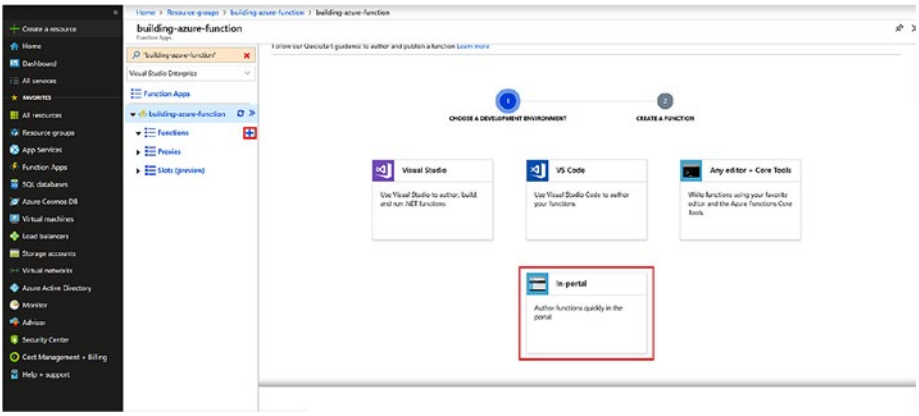

Figure 2-6. *Choose Development Environment as "In-Portal"*

3. Click the Continue button at the bottom, click Webhook + API, and click Create, as shown in Figure 2-7.

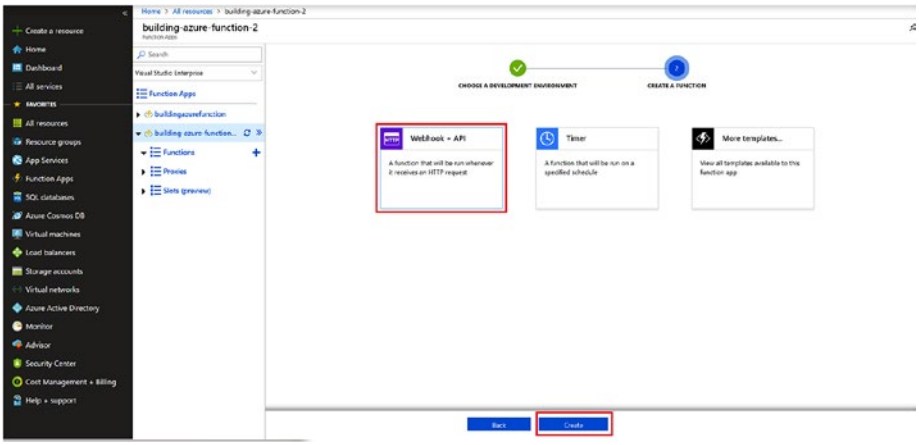

Figure 2-7. *Clicking Webhook + API and then Create*

4. Your first HTTP-triggered function will be created. As you can see in Figure 2-8, it comes with some basic code. This code should be good enough for you to test your function. So, click Save.

5. Once the function is saved, click Run and then click Get Function URL, as shown in Figure 2-8.

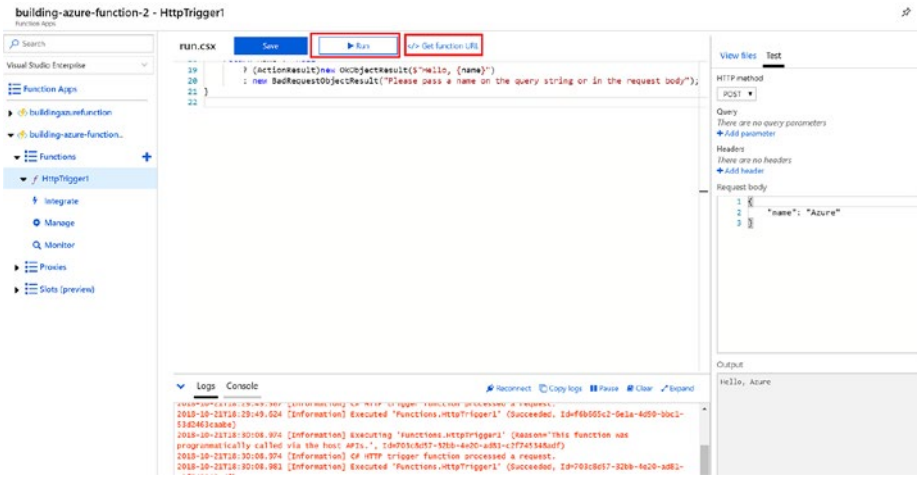

Figure 2-8. *Running your Azure Function*

6. A pop-up will open. Copy the URL shown in the pop-up, and at the end of the URL add a query parameter like &name={name}, such as `https://building-azure-function.azurewebsites.net/api/HttpTrigger1?code=LYXZApwCTtfGjzYOcuGSEc/1SGgEaFFq9Bp6AX6z8ZKfPEU34Dazdw==&name=TestUser`. Paste this URL in the address bar of a browser like Edge or Chrome and click Enter. You will see a message on the screen saying "Hello, {name}," as shown in Figure 2-9.

Figure 2-9. *Viewing the function in the browser*

7. When your function runs, the trace information is written to logs. To see the output of the trace, go back to Azure Portal and click the arrow at the bottom, as shown in Figure 2-10.

Figure 2-10. *Trace log*

With this you have created your first running Azure Functions app. Now, let's create the same thing in Visual Studio Code.

Creating an Azure Function Using Visual Studio Code

In this section, I will take you through the steps for creating functions using Visual Studio Code, which is a popular IDE.

In this section, you will learn how to create functions using the Azure Functions extension for Visual Studio Code and publish the same function as earlier to Azure using Visual Studio Code.

These are the prerequisites:

1. Install Visual Studio Code from `https://code.visualstudio.com/`.
2. Install .NET Core 2.1 for Windows from `https://www.microsoft.com/net/download`.
3. Install Node.js, which consists of NPM, from `https://docs.npmjs.com/getting-started/installing-node#osx-or-windows`. Install the 8.5+ version.
4. Install the Core Packages tool by running `npm install -g azure-functions-core-tools` in the Visual Studio Code terminal. (To open Terminal, go to the Terminal menu at the top and select New Terminal. Once the terminal opens, paste in the code and hit Enter.)

Once you are done with these steps, you are ready to create your first function app using Visual Studio Code.

Creating Your First Function App Using Visual Studio Code

Follow these steps:

1. You need to install the Azure Functions extension in Visual Studio Code. To do that, go to Extensions in Visual Studio Code and search for *Azure Functions*. Then click Install. Refer to Figure 2-11 to understand the steps.

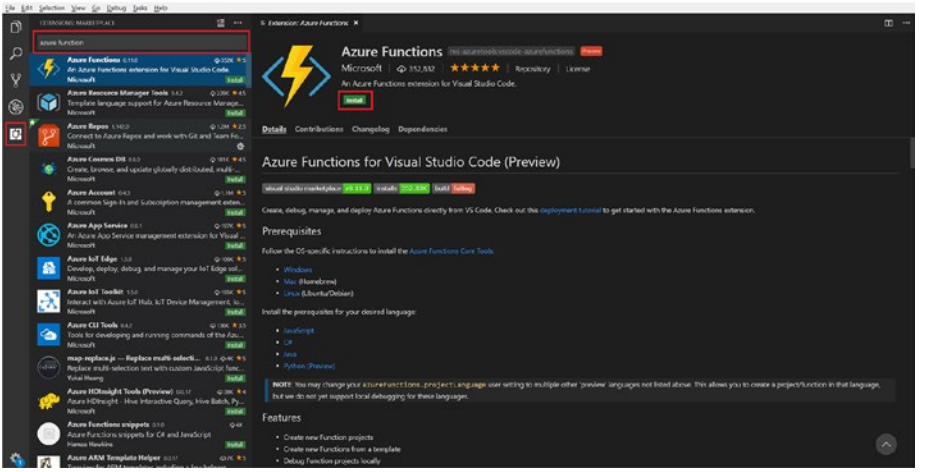

Figure 2-11. *Installing the extension*

2. Once the installation is complete, click the Reload to Activate button or restart Visual Studio Code for the new extension to appear in Visual Studio Code.

3. Click the Azure logo in the vertical menu and then click the folder icon. Select Folder or Create New Folder for the project. Then, select the language you want to code your function in, as shown in Figure 2-12.

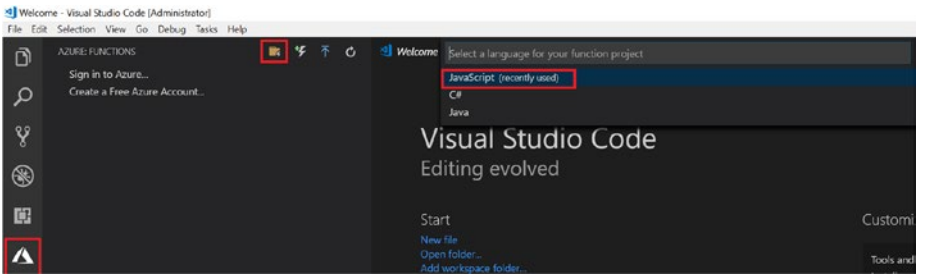

Figure 2-12. *Selecting a language*

4. Select how you would like to open the function app, as shown in Figure 2-13.

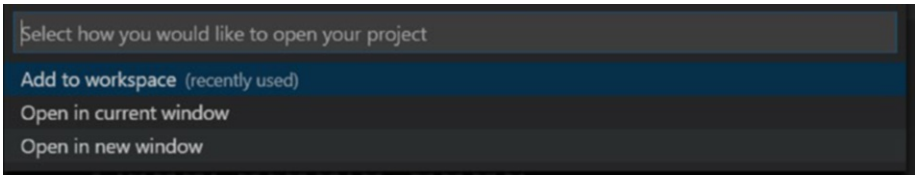

Figure 2-13. *Setting how you would like to open your project*

Creating Your First Function in the Function App

In the previous section, you created your first function app, but a function app without any function is of no use. In this section, you will create your first function.

1. Click the file icon in the vertical menu, and you will see the function app created but with no function. So, click the Azure logo in the vertical menu and click Function. Then select the current project, as shown in Figure 2-14.

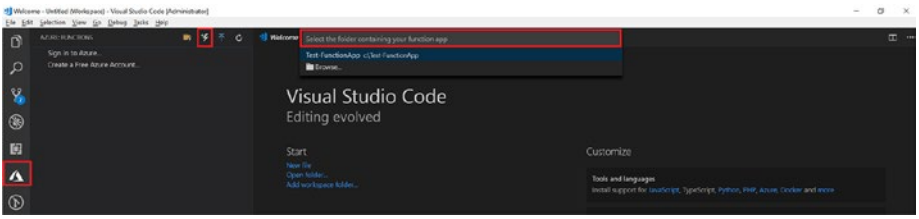

Figure 2-14. *Selecting the current project*

2. Select the HTTP Trigger function template and provide a name for the template. Select Anonymous for the Authorization Type field, as shown in Figure 2-15.

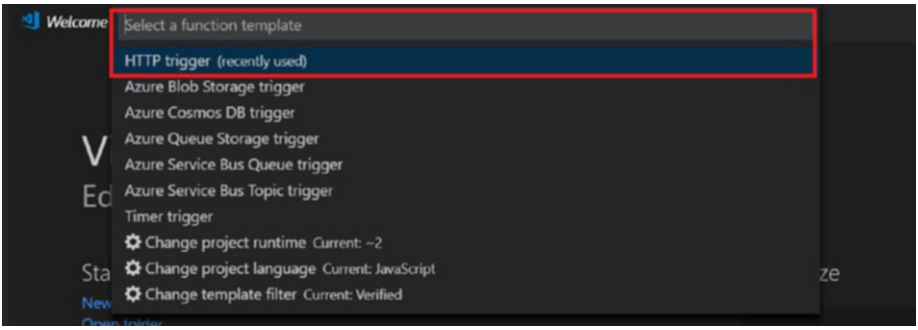

Figure 2-15. *Selecting the type of trigger*

3. You will see that the `index.js` file has been loaded. Press Ctrl+F5 to start the function. Once the function is running (as shown in Figure 2-16), you will get a URL in green. Copy the URL and append `?name={name}` to it (replacing `{name}` with the actual name), and it will show "Hello, {name}."

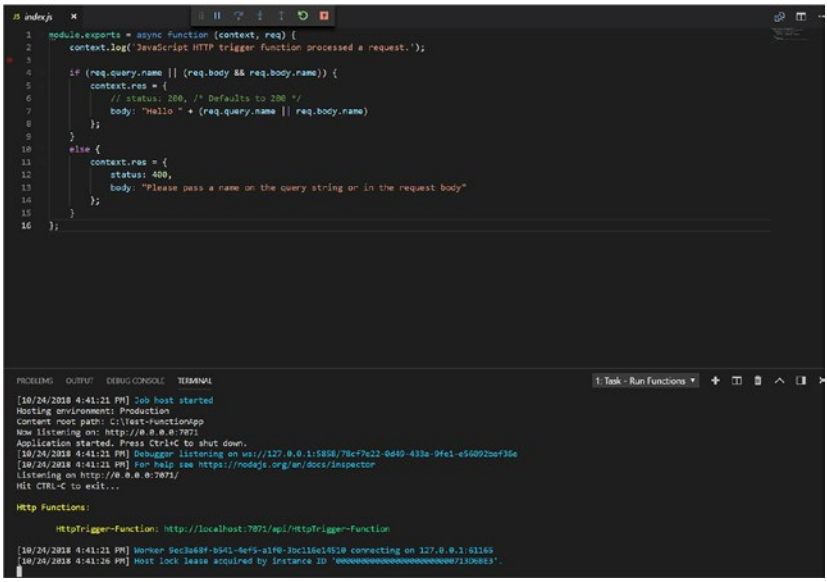

Figure 2-16. *Copying the URL*

4. Click the Azure logo in the vertical menu and click Sign in to Azure. Then click Deploy to Azure and select the subscription, as shown in Figure 2-17.

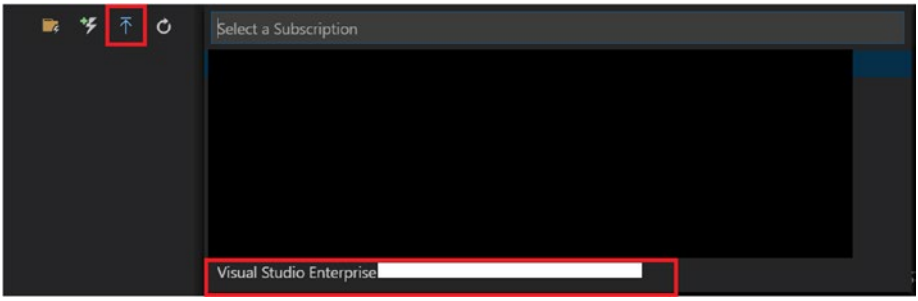

Figure 2-17. *Selecting a subscription*

5. Select Create New Function App in Azure, as shown in Figure 2-18. Then provide a unique name for the function app and press Enter, as shown in Figure 2-19.

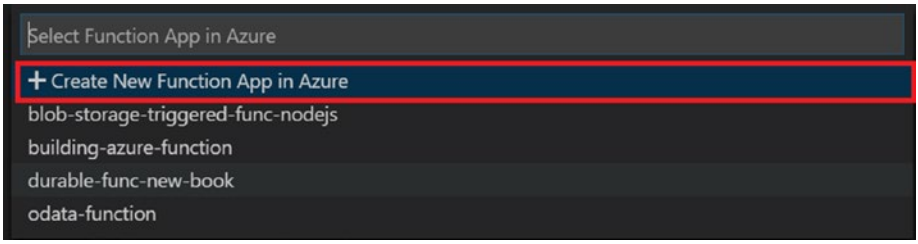

Figure 2-18. *Creating a new function app*

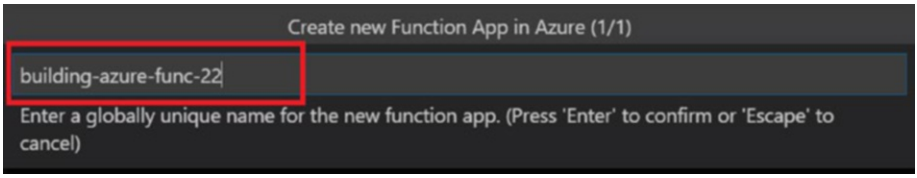

Figure 2-19. *Naming the app*

6. This will start creating a new Azure Functions function app in the selected subscription. You can check it out in the left menu, as shown in Figure 2-20.

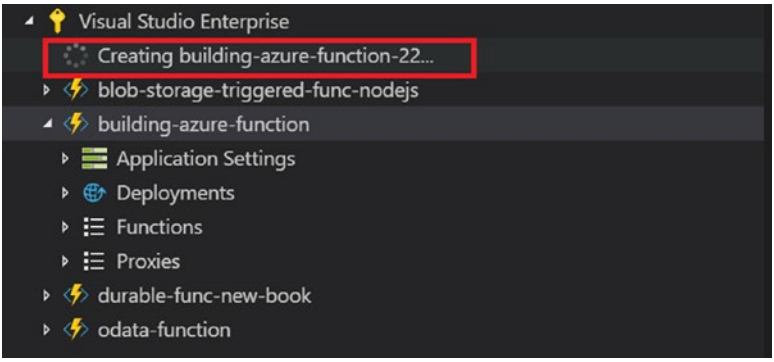

Figure 2-20. *Checking on the creation process*

You have now created two functions, one in Azure Portal directly using C# and another in Visual Studio Code using JavaScript. Let's now look at the settings and hierarchy of Azure Functions.

File Hierarchy, Configuration, and Settings in Azure Functions

It's time to go back to the explorer in Visual Studio Code and take a look at the file hierarchy of Azure Functions. It is important as a developer for you to know which files reside where in Azure Functions. Figure 2-21 shows the file hierarchy.

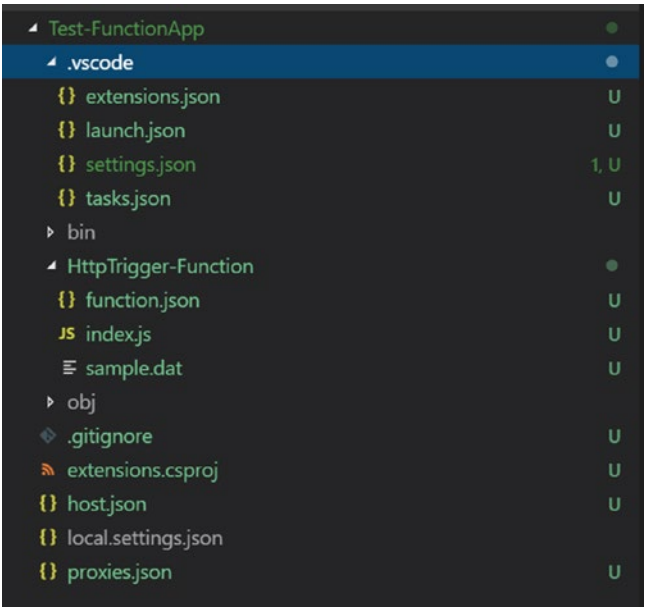

Figure 2-21. *File hierarchy*

Note that the function host will throw an exception if the `host.json` file is missing the `"version": "2.0"` property. Also, `version` requires a string for the value, so `"version": 2.0` will not work.

All the application-level extensions such as CosmosDB, HTTP Trigger, Queues, and so on, reside under the `extensions` object and not in the root of the `json` object, as shown in Figure 2-22.

```
"extensions": {
    "cosmosDB": {
        "connectionMode": "Gateway",
        "protocol": "Https",
        "leaseOptions": {
            "leasePrefix": "prefix1"
        }
    },
    "sendGrid": {
        "from": "Azure Functions <samples@functions.com>"
    },
    "http": {
        "routePrefix": "api",
        "maxConcurrentRequests": 5,
        "maxOutstandingRequests": 30
    },
    "queues": {
        "visibilityTimeout": "00:00:10",
        "maxDequeueCount": 3
    },
    "eventHubs": {
        "batchCheckpointFrequency": 5,
        "eventProcessorOptions": {
            "maxBatchSize": 256,
            "prefetchCount": 512
        }
    }
}
```

Figure 2-22. *Application-level extensions*

All the logging-level settings for Azure Functions reside under the `logging` object, as shown in Figure 2-23.

```
"logging": {
   "fileLoggingMode": "debugOnly"
},
```

Figure 2-23. *Logging-level extensions*

In this chapter, you created your first function app and function. Also, you learned about the Azure Functions file hierarchy, configuration, and settings.

CHAPTER 3

Understanding Azure Functions Triggers and Bindings

This chapter covers the following topics:

- Overview of triggers and bindings
- Azure Functions 2.0's changes to bindings
- Creating a Blob Storage–triggered function

Overview of Triggers and Bindings

Azure Functions is like WebJobs and the Web API in that it needs to be invoked either by using Scheduler or by calling endpoints. In the case of Azure Functions, a *trigger* is what invokes a function to run. A trigger defines how a function is invoked, and each function in Azure Functions must have only one trigger. Triggers usually have associated data, which is nothing but the payload that triggers the function.

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_3

Different types of triggers are available.

- **BlobTrigger**: This trigger gets fired when a new blob or a blob update is detected. The blob contents are provided as input to the function.
- **QueueTrigger**: This trigger gets fired when a new message arrives in the Azure storage queue.
- **EventHubTrigger**: This trigger gets fired when any event is delivered to the Azure Event Hub service.
- **TimerTrigger**: This trigger is called on a scheduled basis. You can set the time to execute the function using this trigger.
- **HTTPTrigger**: This trigger gets fired when the HTTP request comes. In Chapter 2, you created an HTTP-triggered function using Visual Studio Code.
- **Service Bus Trigger**: This trigger gets fired when a new message comes in to an Azure Service Bus topic or queue.
- **Generic Webhook**: This trigger gets fired when a webhook HTTP request comes from any service that supports webhooks.
- **GitHub Webhook**: This trigger gets fired when any event such as Create Branch, Delete Branch, Issue Comment, or Commit Comment occurs in your GitHub repository.

Let's now discuss bindings in Azure Functions. Azure Functions *bindings* are a declarative way of connecting another resource to a function. Bindings can be connected as input bindings, output bindings, or both. Data from these bindings is provided to the function as parameters.

Azure Functions 2.0 has the following bindings:

- Blob Storage
- Cosmos DB
- Event Grid
- Event Hubs
- HTTP & Webhooks
- Microsoft Graph Events
- Microsoft Graph Excel tables
- Microsoft Graph Outlook e-mail
- Microsoft Graph OneDrive files
- Microsoft Graph Auth Tokens
- Queue Storage
- Table Storage
- Service Bus
- Timer
- Webhooks
- SendGrid
- SignalR
- Twilio

Bindings are optional in Azure Functions, and you can have multiple input and output bindings. In Azure Functions 2.0, all the bindings must be registered except HTTP and Timer.

Azure Functions triggers and bindings are configured in the `functions.json` file, and they help you avoid putting hard-coded values in the code.

Note You can find all the supported bindings in Azure Functions 2.0 by visiting https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-bindings.

Azure Functions 2.0 Changes

Azure Functions 2.0 now supports .NET Core 2.*x*, which means Azure Functions 2.0 supports cross-platform development. That means Azure Functions 2.0 now runs in more environments than just Mac and Linux machines. Developers can develop functions on all major platforms including Windows, Linux, and Mac.

Azure Functions 2.0 also supports non-.NET languages by using the language worker model and now supports both Node 8 and Node 10. Azure Functions 2.0 is faster and more performant than 1.0 as it now runs on a modern language runtime.

In the Azure Functions 2.0 runtime, a new binding model was introduced by Microsoft in which the bindings are no longer referenced by the runtime by default except the few core bindings like HTTP and Timer.

With this new model, the runtime is decoupled from the extensions, which provides additional flexibility and reduces the load by loading only the extensions referenced in the function app. This also means that the runtime now has no knowledge of the extensions, so they must be registered before use.

Extensions are now distributed as NuGet packages, and the registration of these extensions is done by installing the required NuGet package of the extension.

Installing Extensions Using the Azure Functions Core Tools

With Azure Functions 2.0, the Azure Functions Core Tools (CLI) has also been enhanced to support the new extension model. With 2.0, a new extension context has been added to allow you to manage the extensions.

- **func extension install**: With this command **func extension install** you can install an extension and register it to the function.
- **func extension sync**: This command **func extension sync** allows you to install or uninstall the extensions that are referenced in a function.

The following is the example of installing an extension:

```
func extension install –package Microsoft.Azure.WebJobs.
Extensions.Storage –version 3.0.1
```

This command installs the Blob extension to Azure Functions 2.0, which will allow you to configure your function for a blob trigger.

Installing Extensions Using the Azure Functions Visual Studio Tools

With Visual Studio Code, you will be referencing the extensions package directly from the project. So, Visual Studio Code handles the installation of the extensions, but you still need to register the extensions.

Extension registration is handled by a custom build task added by the NuGet package called `Microsoft.Azure.WebJobs.Script.ExtensionsMetadataGenerator`, which needs to be explicitly referenced for now. In a future release, this will be part of the SDK or Visual Studio Tools.

The following are the steps that you need to perform to use the Blob extension:

1. Add a reference to the `Microsoft.Azure.WebJobs.Extensions.Storage` NuGet package.
2. Add a reference to `Microsoft.Azure.WebJobs.Script.ExtensionsMetadataGenerator`.
3. Build the project.

Note With any of the installation steps mentioned, if you install and register the extension, a metadata file named `extensions.json` will be generated in the `bin` folder inside the function's app root folder. Only the extensions registered in this file will be used by the runtime.

Creating a Blob Storage–Triggered Function

In this section, you will learn how to create a Blob Storage–triggered function using both C# and Node.js. I will take you through the process one by one. You will be using Visual Studio Code to create the function. To set up the system to create a function, you can check Chapter 2's "Creating an Azure Function Using Visual Studio Code" section.

In this function, you will try to resize the image once it is uploaded on the blob by using a Blob trigger in Azure Functions.

Let's start by first creating a function using C#.

Creating a Blob-Triggered Function Using C#

Set up the machine as mentioned in Chapter 2. Once the machine is set up, open Visual Studio Code, go to the Extensions section, and install the C# extension. Make sure your Azure Functions extension version is 0.16.0. Once this is done, go to the Azure Function menu and add a new function with the following steps:

1. Open Visual Studio Code and then click the Azure logo in the left menu, as shown in Figure 3-1.

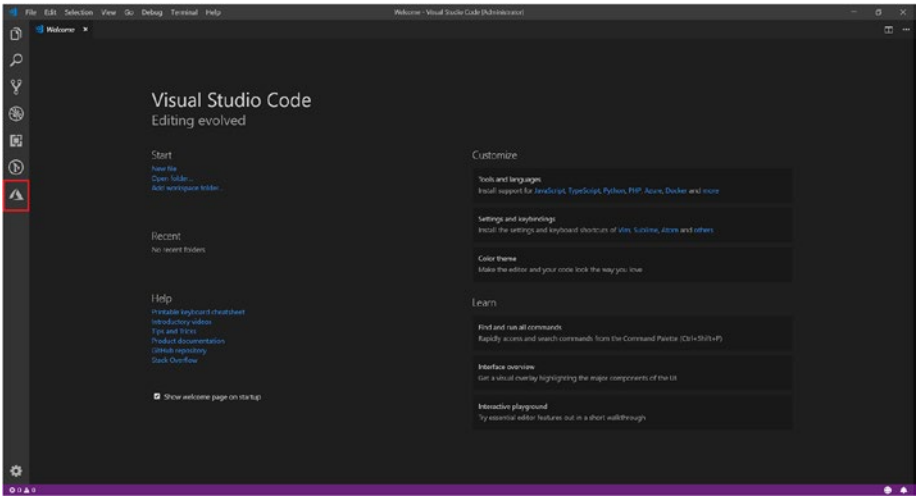

Figure 3-1. *Clicking the Azure logo*

2. Click the folder icon and create a new folder, as shown in Figure 3-2.

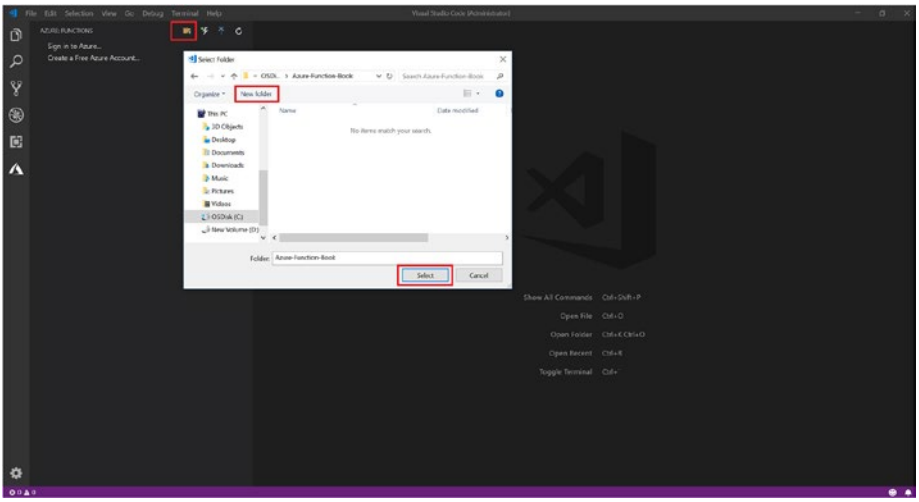

Figure 3-2. *Creating a new folder*

3. Once a new folder is created, select the language in which you want to code your function. In this case, I am selecting C#, as shown in Figure 3-3.

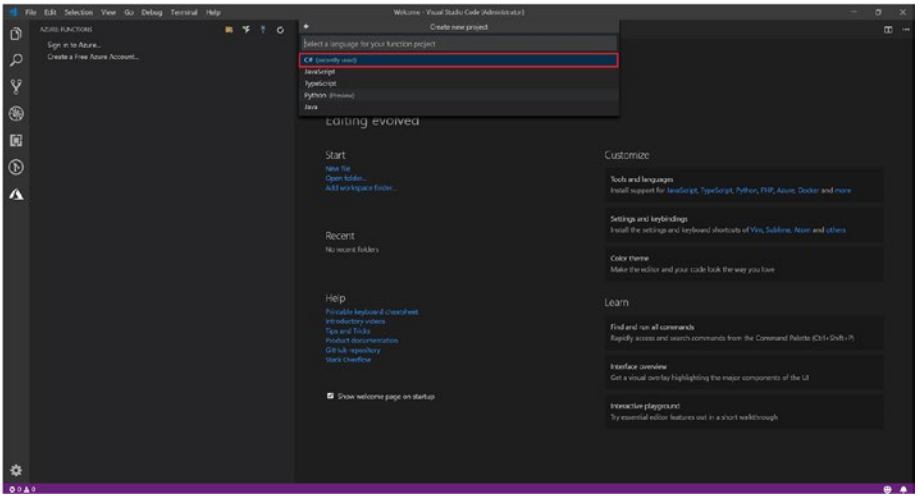

Figure 3-3. *Selecting the language*

4. Select the template for the function. In this case, select BlobTrigger as the template, as shown in Figure 3-4.

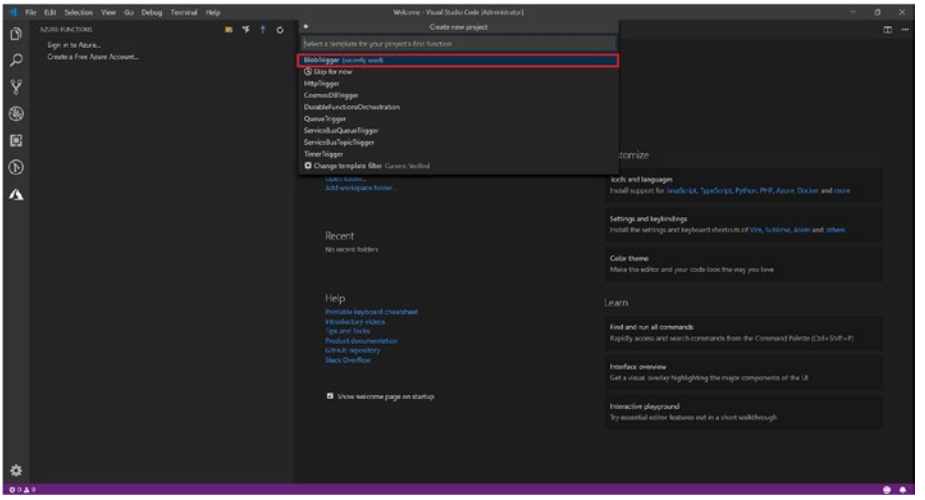

Figure 3-4. *Selecting BlobTrigger*

5. Provide the name of the function. By default, it will become BlobTriggerCSharp. You can leave it as is, or you can type any name you want, as shown in Figure 3-5.

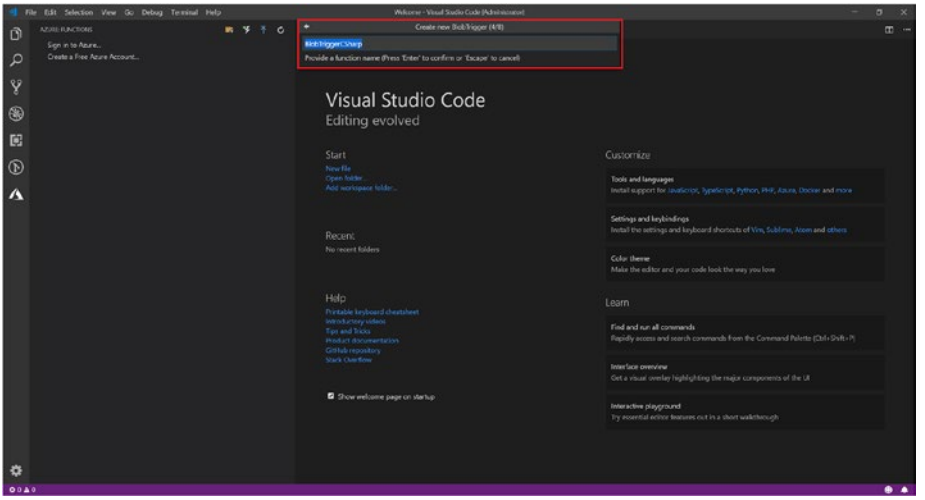

Figure 3-5. *Naming the function*

6. Provide the namespace. By default it will be `Company.Function`. For this demo, set it as `AzureFunctionV2Book.Function`, as shown in Figure 3-6.

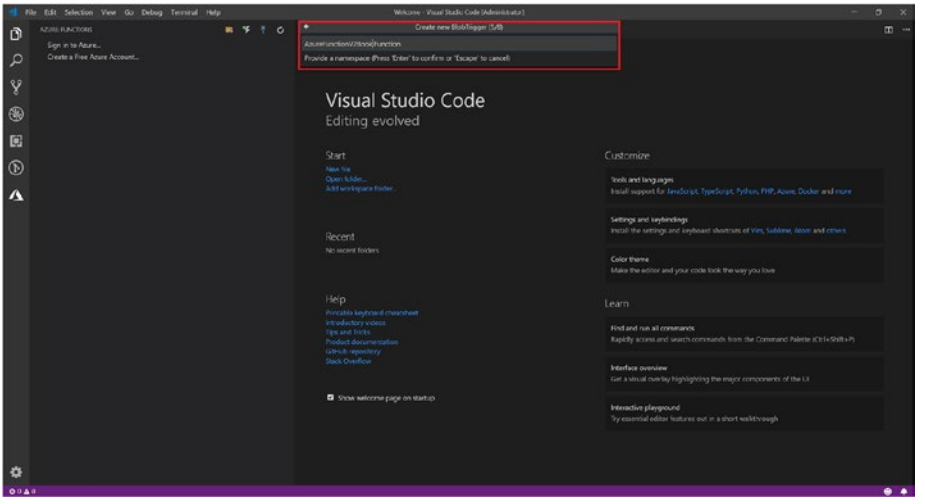

Figure 3-6. *Providing the namespace*

7. Click "Create new local app setting," as shown in Figure 3-7.

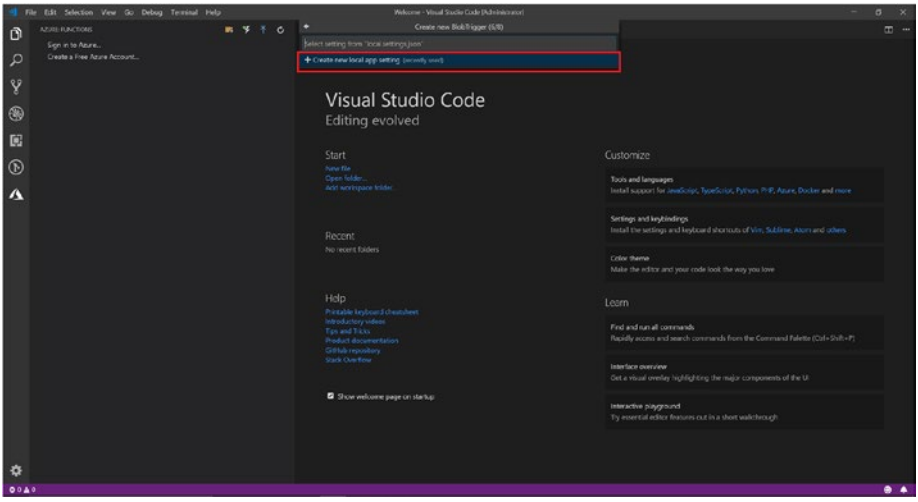

Figure 3-7. *Creating a new local app setting*

8. The screen will ask you to sign in to Azure, as shown in Figure 3-8, or it will show you the subscriptions if you are already signed in. Select "Sign in to Azure" if you already have account or select "Create a free Azure Account" if you don't have one.

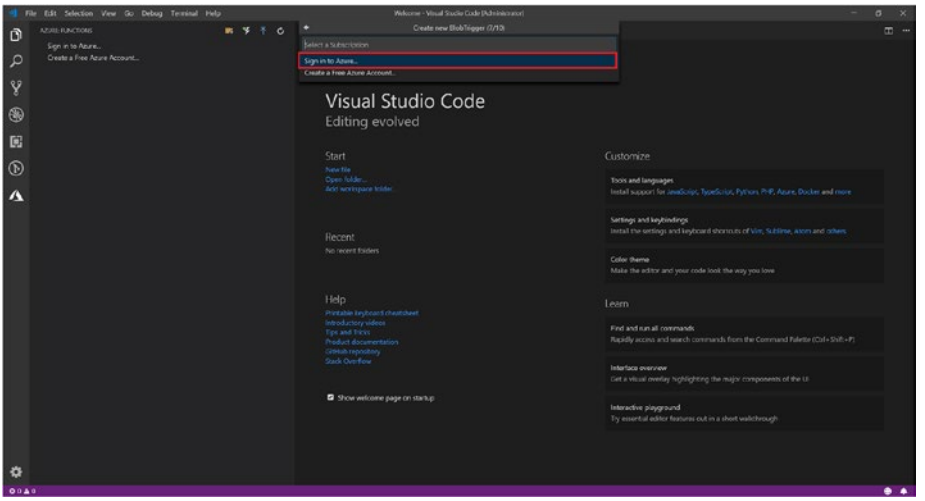

Figure 3-8. *Signing in*

9. Once you have selected "Sign in to Azure," it will take you to your browser to sign in. Then, it will load all the subscriptions that you have in Visual Studio Code. Select the subscription, as shown in Figure 3-9.

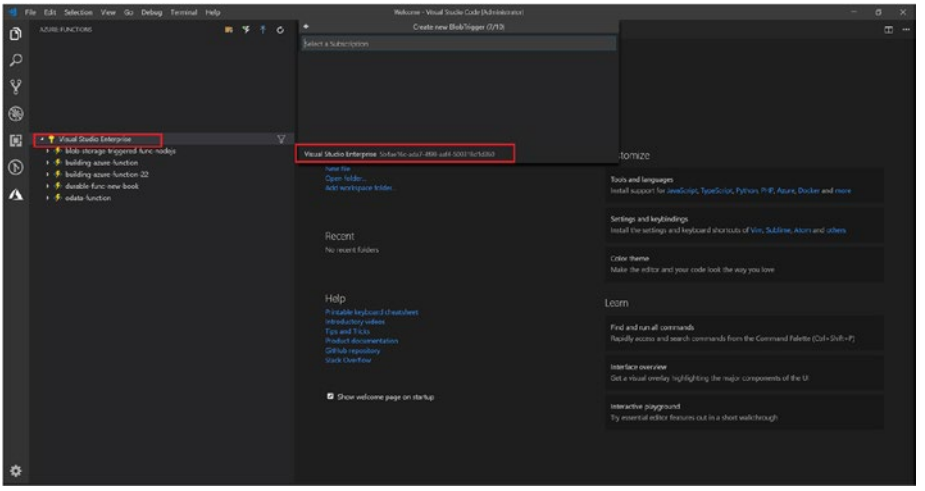

Figure 3-9. *Selecting the subscription*

10. You can select the Azure Storage account that already exists, or you can create a new storage account. In this case, you will select the existing one, as shown in Figure 3-10.

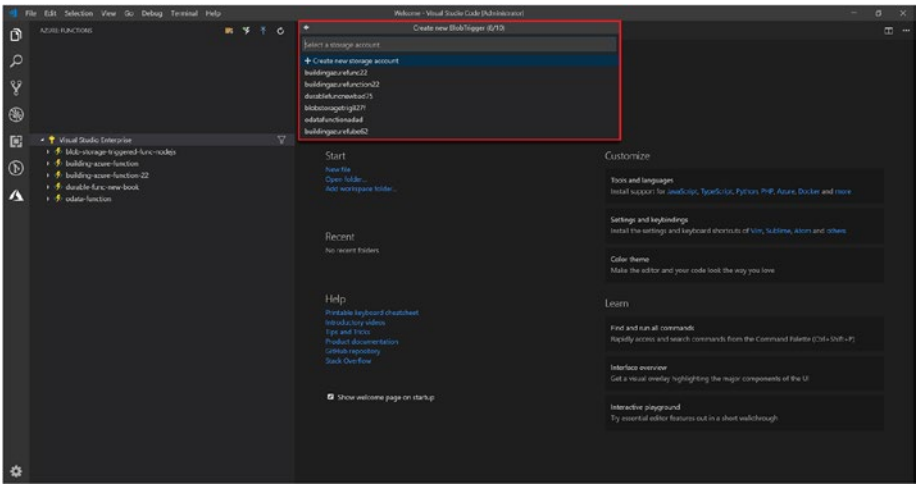

Figure 3-10. *Selecting an existing storage*

11. Once the storage account is set up, provide a name for the blob trigger. This is the path that the trigger will monitor. By default, it shows as `samples-workitems`. For this function, you are setting it to `function-v2-book`, as shown in Figure 3-11.

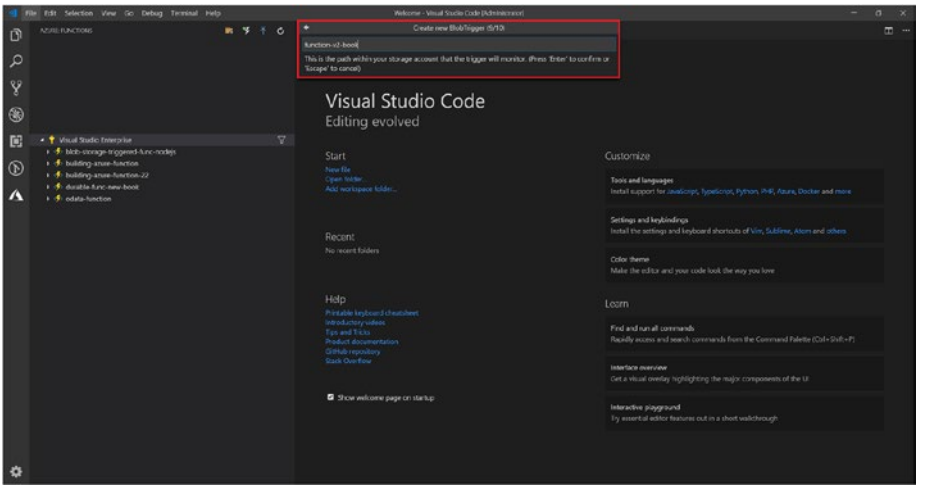

Figure 3-11. *Setting it to function-v2-book*

12. Click "Add to workspace," and your function will be ready, as shown in Figure 3-12.

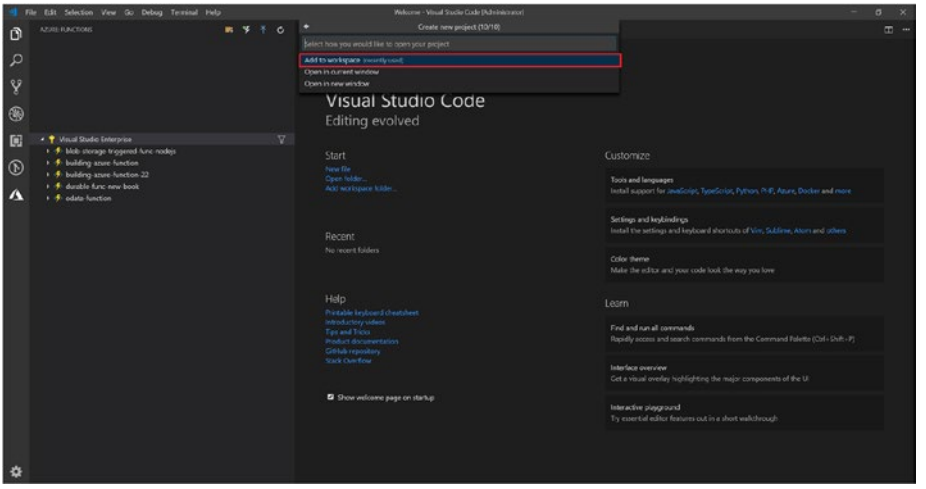

Figure 3-12. *Adding to the workspace*

13. Your function is all set up, as shown in Figure 3-13.

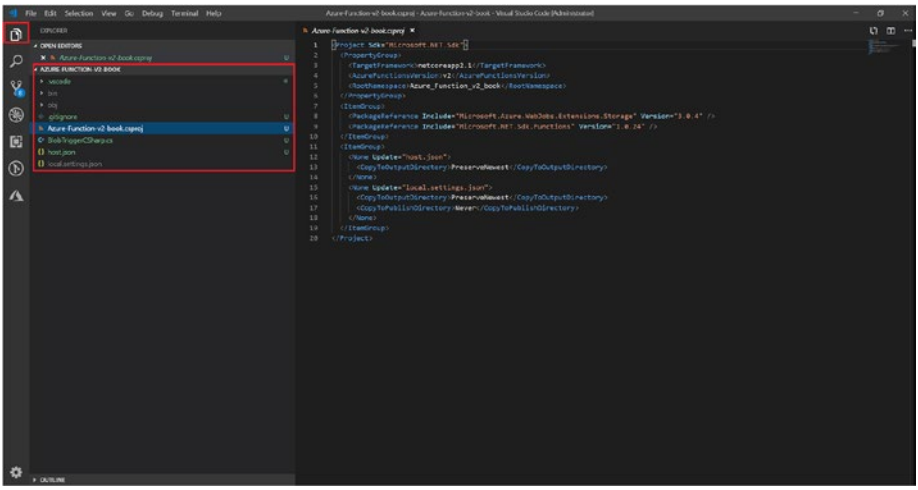

Figure 3-13. *Completed function*

14. Now you need to add a package named `SixLabors.ImageSharp` to the project. To do that, type `dotnet add package SixLabors.ImageSharp -v 1.0.0-beta0006` in the Terminal, and it will install the package.

15. Once the package is installed, paste the following code into the `.cs` file:

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.Blob;
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats.Png;
using SixLabors.ImageSharp.PixelFormats;
using SixLabors.ImageSharp.Processing;
```

```
namespace AzureFunctionv2Book.Function
{
    public static class BlobTriggerCSharp
    {
        [FunctionName("BlobTriggerCSharp")]
        public static async Task Run([BlobTrigger
        ("image-blob/{name}", Connection = "AzureWeb
        JobsStorage")]Stream myBlob, string name,
        [Blob("output-blob/{name}", FileAccess.
        ReadWrite, Connection = "AzureWebJobsStorage")]
        CloudBlockBlob outputBlob, ILogger log)
        {
            log.LogInformation($"C# Blob trigger
            function Processed blob\n Name:{name} \n
            Size: {myBlob.Length} Bytes");

            var width = 100;
            var height = 200;
            var encoder = new PngEncoder();
            using (var output = new MemoryStream())
            using (Image<Rgba32> image = Image.Load
            (myBlob))
            {
                image.Mutate(x => x.Resize(width,
                height));
                image.Save(output, encoder) ;
                output.Position = 0;
                await outputBlob.UploadFromStream
                Async(output);
            }
        }
    }
}
```

16. In the top menu, click Debug and then click Start Without Debugging to get the function up and running. Now, go to Azure Storage and create two containers named `image-blob` and `output-blob`. If you have never created a container before, go to the following link to see how to create containers: https://docs.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-portal.

    Once the container is created, upload the blob as shown in the previous link in the `image-blob` container. Now you will see the function getting triggered, and once the function runs, the resized image will appear in `output-blob`, as shown in Figure 3-14.

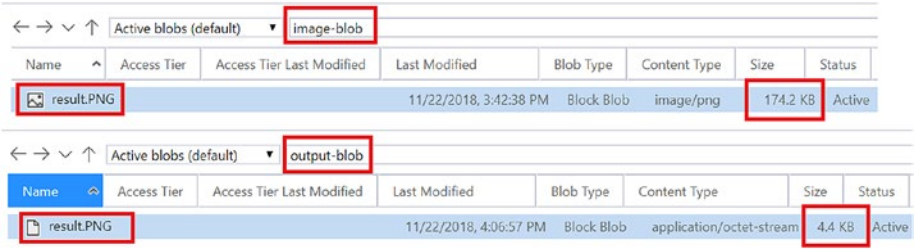

Figure 3-14. *Resized image*

With this you have created a blob-triggered function using C#. Now, let's look at creating a blob-triggered function using Node.js.

Blob-Triggered Function Using Node.js

Let's try to implement the same functionality of image resizing using Node.js.

1. Set up the machine for Node.js by installing the latest version of Node.js. Once that is done, restart Visual Studio Code and go to the function. The first two steps are the same as what you did while creating a blob-triggered function using C#. So, you will start from step 3.

2. Select JavaScript as the language, as shown in Figure 3-15.

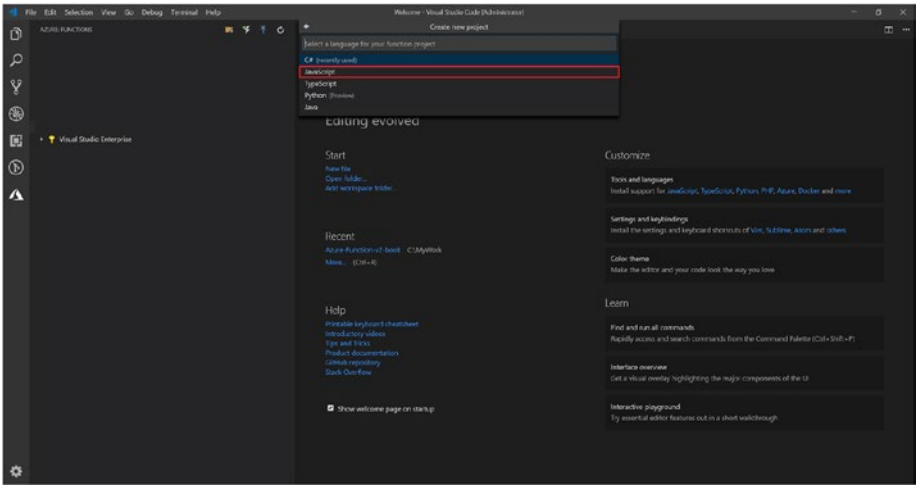

Figure 3-15. *Selecting the language*

3. Now, select the template Azure Blob Storage Trigger, as shown in Figure 3-16.

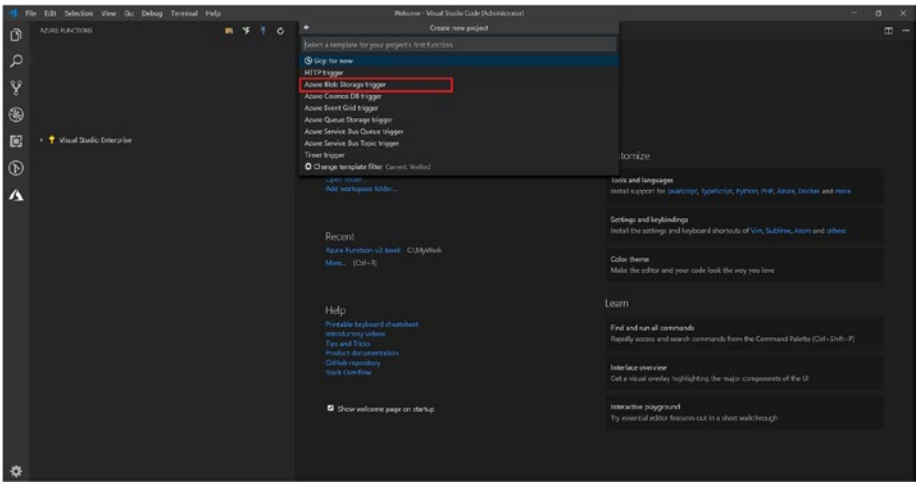

Figure 3-16. *Selecting the template*

4. Provide the function name. By default, it will be `BlobTrigger`. For this function, set the function name to `BlobTriggerJs`, as shown in Figure 3-17.

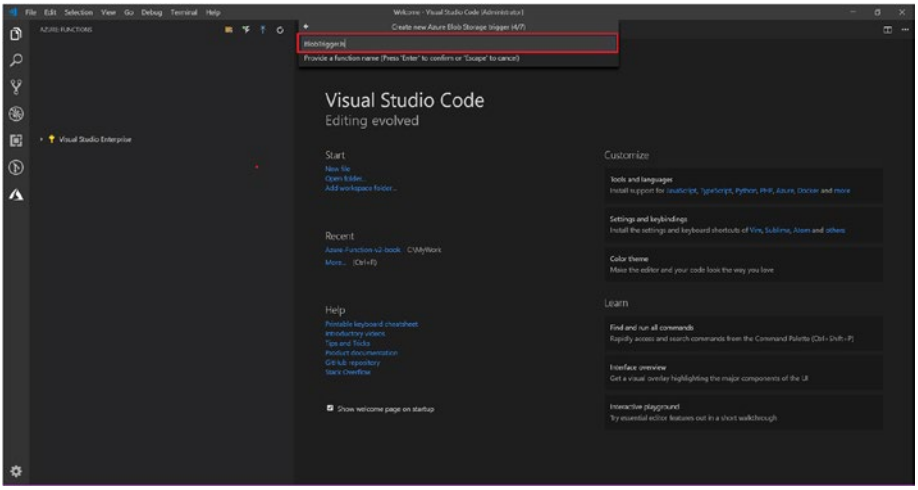

Figure 3-17. *Naming the function*

5. Click "Create new local app setting," as shown in Figure 3-18.

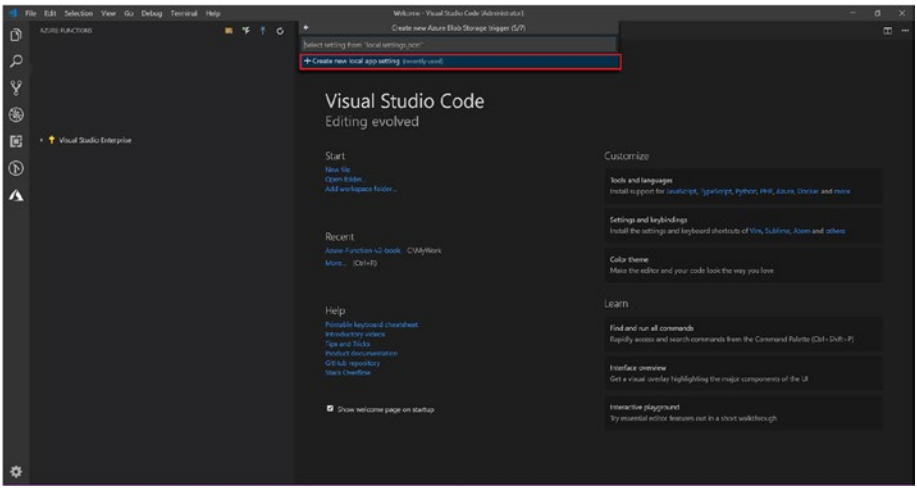

Figure 3-18. *Creating a new local app setting*

6. Since you have already connected to Azure in the previous section, you should now directly see all the subscriptions available in Azure. Select the Azure subscription under which you want to create this function, as shown in Figure 3-19.

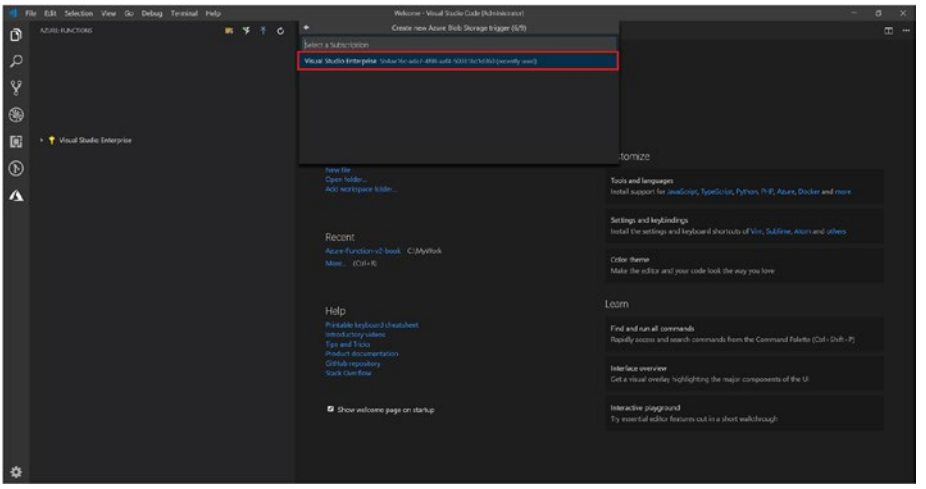

Figure 3-19. *Selecting a subscription*

7. Select the Azure Storage account that you want this function to connect with, as shown in Figure 3-20.

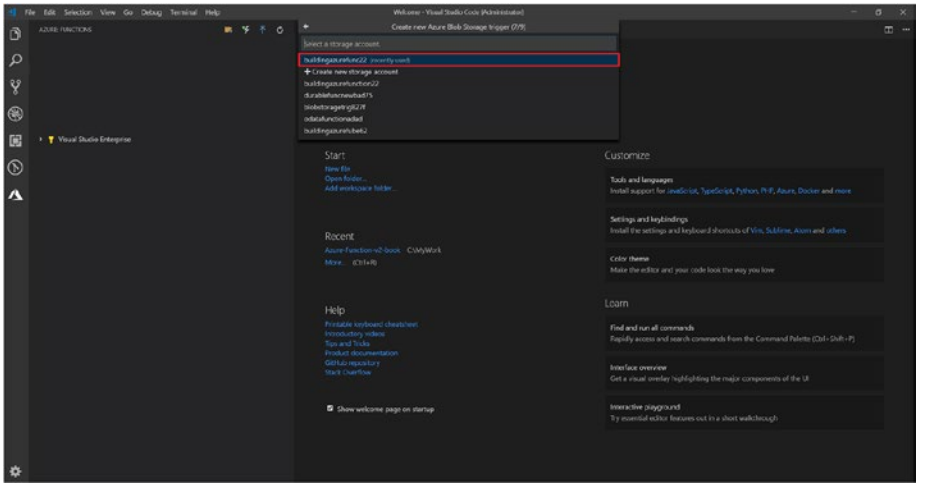

Figure 3-20. *Selecting the Azure Storage account*

8. After selecting the app setting name, the function will ask you to provide the name of the blob that will trigger this function. Provide the name of your blob, as shown in Figure 3-21.

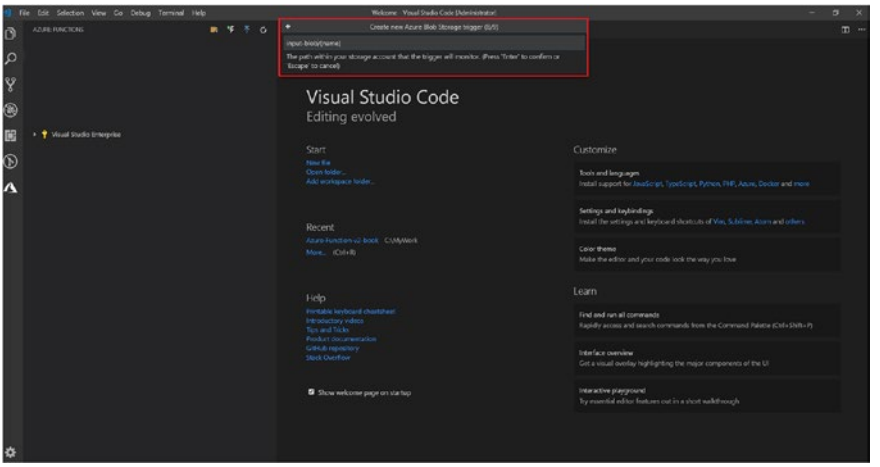

Figure 3-21. *Naming your blob*

9. Select "Add to workspace," as shown in Figure 3-22.

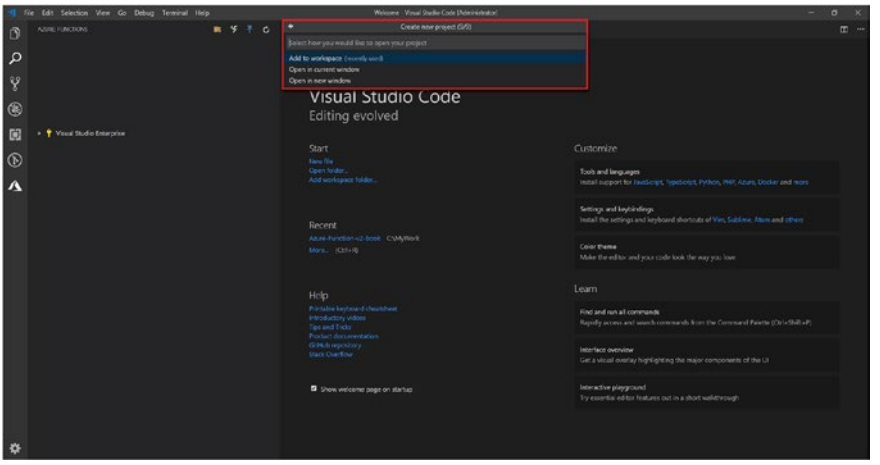

Figure 3-22. *Adding to the workspace*

10. Your function is ready. Click the file icon, and you will see all the function files, as shown in Figure 3-23.

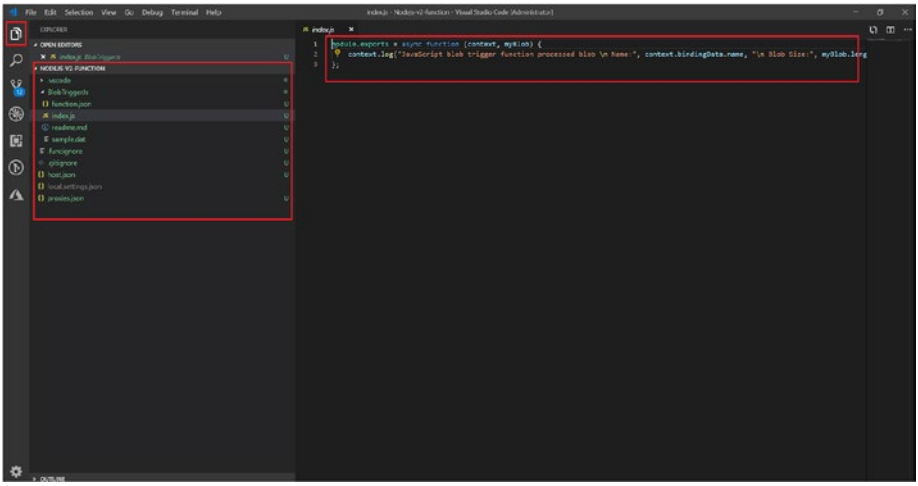

Figure 3-23. *Function files*

11. To resize the image, you need to add a few NuGet packages. The packages that you need to install are `azure-storage`, `urijs`, `stream`, `jimp`, and `async`. You can install these packages using `npm i <package name>`.

12. Once the packages are installed, copy the following code and paste it in:

```
var storage = require('azure-storage');
var URI = require('urijs');
const stream = require('stream');
const Jimp = require('jimp');
var async = require('async');
```

```
module.exports = async function (context, myBlob) {
    context.log("JavaScript blob trigger function
    processed blob \n Name:", context.bindingData.name,
    "\n Blob Size:", myBlob.length, "Bytes");

    var blobService = storage.createBlobService
    (process.env.AzureWebJobsStorage);
    var blockBlobName = context.bindingData.name;
    const widthInPixels = 60;
    const heightInPixels = 60;
    const blobContainerName = 'output-blob';
    async.series(
      [
        function (callback) {
          blobService.createContainerIfNotExists(
            blobContainerName,
            null,
            (err, result) => {
              callback(err, result)
            })
        },
        function (callback) {
          var readBlobName = generateSasToken
          ('input-blob', blockBlobName, null)
          Jimp.read(readBlobName.uri).then
          ((thumbnail) => {

            thumbnail.resize(widthInPixels,
            heightInPixels);

            thumbnail.getBuffer(Jimp.MIME_PNG, (err,
            buffer) => {
```

```
            const readStream = stream.PassThrough();
            readStream.end(buffer) ;

            blobService.createBlockBlobFromStream
            (blobContainerName, blockBlobName,
            readStream, buffer.length, null,
            (err, blobResult) => {
              callback(err, blobResult);
            });
          });
        });
      }
    ],
    function (err, result) {
      if (err) {
        callback(err, null);
      } else {
        callback(null, result);
      }
    }
  );
};

function generateSasToken(container, blobName,
permissions) {
    var connString = process.env.AzureWebJobsStorage;
    var blobService = azure.createBlobService
    (connString);

    // Create a SAS token that expires in an hour
    // Set start time to five minutes ago to avoid
       clock skew.
```

```
    var startDate = new Date();
    startDate.setMinutes(startDate.getMinutes() - 5);
    var expiryDate = new Date(startDate);
    expiryDate.setMinutes(startDate.getMinutes() + 60);

    permissions = permissions || storage.BlobUtilities.
    SharedAccessPermissions.READ;

    var sharedAccessPolicy = {
        AccessPolicy: {
            Permissions: permissions,
            Start: startDate,
            Expiry: expiryDate
        }
    };

    var sasToken = blobService.generateSharedAccessSign
    ature(container, blobName, sharedAccessPolicy);

    return {
        token: sasToken,
        uri: blobService.getUrl(container, blobName,
        sasToken, true)
    };
}
```

13. Once you run the code and upload the image, you should see the image getting resized.

Running the Example

With these examples, you created two blob-triggered functions running on the 2.0 framework, one with C# and another one with JavaScript/Node.js. The main thing to note here is that the file `host.json` is important. It stores the version of the function and lets the framework know on what version you are running your function.

```
{
  "version": "2.0"
}
```

Now you understand the concept of Azure triggers and bindings, and you have created a blob-triggered function. In the next chapter, you will look at creating serverless APIs using Azure Functions.

CHAPTER 4

Serverless APIs Using Azure Functions

Before you start creating APIs with Azure Functions, it is imperative for you to understand where Azure Functions as a serverless API will fit into the current system architecture that you are planning to use for building your product or applications.

Traditionally, applications were based on a monolithic architecture because developers wanted all the APIs to be a single deployable unit. Setting up an individual API for the business case was a mammoth task, so with the advent of cloud computing and the agile process, the monolithic approach became less desirable. Developers started looking at microservice architecture because cloud giants such as Microsoft, Amazon, and Google made microservices easy.

In this chapter, I will cover the following topics:

- Monolithic architecture vs. microservice architecture
- Converting monolithic applications to highly scalable APIs using Azure Functions
- Creating an HTTP-triggered function
- Overview of proxies in Azure Functions

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_4

Let's look in detail at the monolithic and microservice architectures in the next section and try to understand which architecture to use in specific circumstances and where Azure Functions fits in.

Monolithic Architecture vs. Microservice Architecture

The *monolithic* approach used to be one of the most popular approaches to building applications, where the complete application resides in one codebase consisting of client-side applications, server-side applications, and database code.

But with time, these monolithic applications become complex and difficult to maintain, and compared to the agile development model, monolithic applications are vulnerable to bugs and deployment issues. For example, if there is a bug in the client-side code, you still have to deploy all the code after fixing the bug since everything resides in one codebase. This includes the server-side code, which can create issues if the server-side code was not tested properly.

Also, with most applications now moving to the cloud, monolithic architecture makes it difficult (and more expensive) for applications to scale. In addition, DevOps becomes slow and complex, and the time to deploy features, bugs, hotfixes, and so on, keeps increasing. This is where the microservice architecture comes to the rescue.

The *microservice* architecture is the idea of breaking this complex monolithic application into small and independent applications. With a microservice architecture, it becomes easy and less expensive to deploy and scale individual applications and makes DevOps less time-consuming. If you further break down microservices, it is called *nano services*.

The benefits of microservices are as follows:

- The small and independent codebase is easy to understand and maintain.
- Onboarding new developers becomes easy.
- On the cloud, each application can be scaled individually based on their consumption.
- Multiple teams can work in parallel on different microservices.
- The language barrier can be avoided as each microservice can be written in a different coding language based on which language best suits the business scenario. This practice of writing code in multiple languages to capture additional functionality and efficiency that is not available in a single language is known as *polyglot programming*.

But, with the benefits, there are also trade-offs when using microservices.

- Writing test cases becomes difficult for each individual application.
- Communication within the APIs can become slow if not developed properly.
- If DevOps is not properly set up, deployment can become messy and can create a lot of issues (but if done properly, it becomes easy to maintain). A complete enterprise application can have more than 10 to 12 microservices, so it is imperative for you to have a stable CI/CD pipeline for each; otherwise, deploying these microservices can end up being your biggest blocker.

Figure 4-1 illustrates the differences between a monolithic architecture and a microservice architecture.

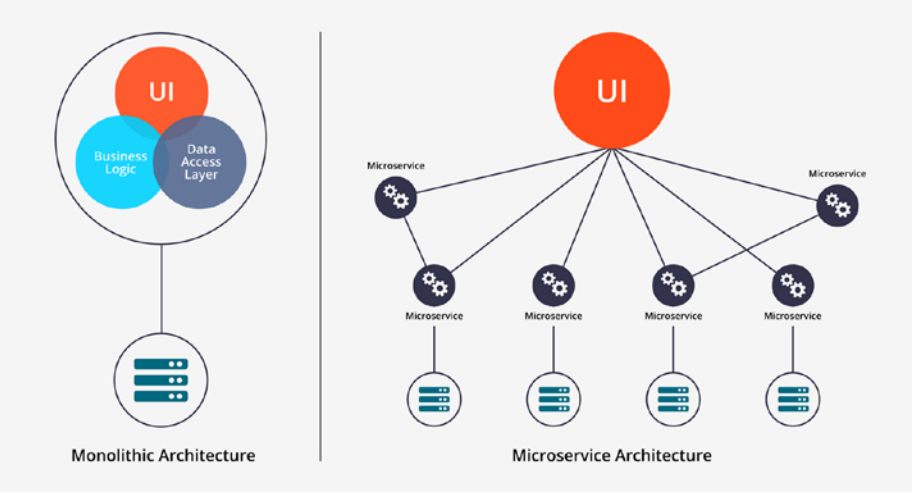

Figure 4-1. *Monolithic architecture vs. microservice architecture*

Converting Monolithic Applications to Highly Scalable APIs Using Azure Functions

Let's now look at converting a monolithic e-commerce web site to microservices. A basic e-commerce web site comes with a client interface, a customer profile, product details, checkout and payment functions, and inventory management.

By looking at these, you can easily see that each is an individual business scenario and can be converted to a microservices architecture. You would have the following microservices:

- Customer service, which will include customer details, orders, and so on
- Product service

- Payment and checkout service
- Inventory management service

You can expose each of these services as an API that can be easily consumed by your front-end user interface.

Now, let's see how microservices will help you in scaling the application with a minimum cost. Let say you have a sale coming up and your estimate is that you will have double the amount of traffic on the web site during this time.

After some analysis, you find that the product service and the payment and checkout services will have the most load, and there won't be much change in load on the customer and inventory management service. With microservices, you can scale out only those two services (product and payment and checkout) and leave the other services as is, whereas if you had a monolithic application, you would have to scale out the complete application, and that would increase your application costs a lot.

Since now you are aware why you would want to convert a monolithic application to microservices, let's understand how Azure Functions can help you achieve that in a simpler and more cost-efficient way.

Azure Functions allows you to write and deploy small pieces of code. With the help of Azure Functions, you can divide the microservices into small parts and write a function that performs a specific task. For example, the customer service microservice would have different activities such as Update Profile, View My Orders, Cashback Amount, Card Details, and so on. You could write each one of them as a separate function, and on days where you have a big sale, you can scale up the individual functions.

With Azure Functions, the infrastructure maintenance is taken care of by the cloud service provider, and you won't have to worry about scaling up, upgrading the software, and so on. This in a way reduces the load on the team and helps them concentrate on the business scenario.

Azure Functions provides you with a free grant of 400,000 GB-s of execution time and 1 million total executions per month, which should be enough to run a medium-sized application on Azure Functions at no cost.

With Azure Functions, each function is completely isolated, so if a bug or issue is fixed in one function, you do not have to deploy the complete microservice or application. This is where Azure Functions also makes DevOps a lot easier.

As you can see in Figure 4-2, you can actually create functions for each of the microservices discussed earlier, and you can expose them as REST APIs.

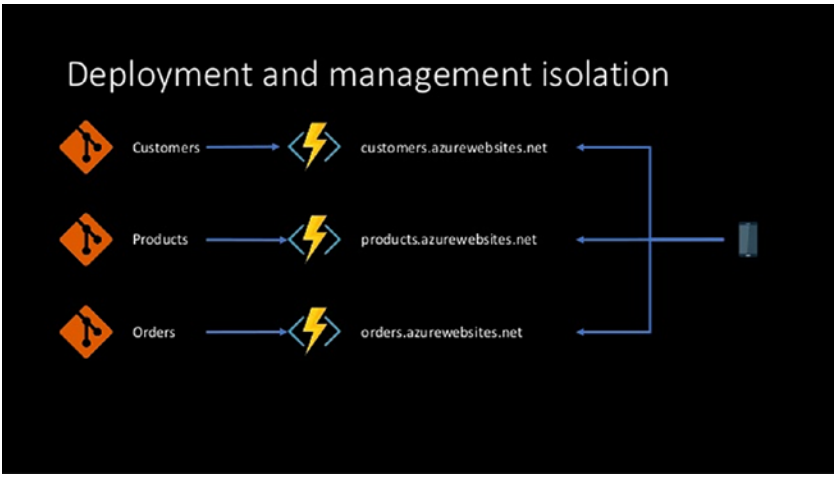

Figure 4-2. *Isolated functions*

To expose functions as REST APIs, you have to create HTTP-triggered functions so that you can use them. HTTP-triggered functions start working like any other API where you call an endpoint and it returns you the result. Let's create an HTTP-triggered function in the next section.

Creating an HTTP-Triggered Function with SQL Server Interaction

Before you start creating the HTTP-triggered function in this section, let's first create the Azure SQL Server database with AdventureWorks content so that you can fetch and modify the data using the HTTP-triggered function.

Creating a SQL Server Instance with Sample Data

Let's get started.

1. Log in to Azure Portal and click "Create a resource." Select Databases from the vertical pane and then select SQL Database, as shown in Figure 4-3.

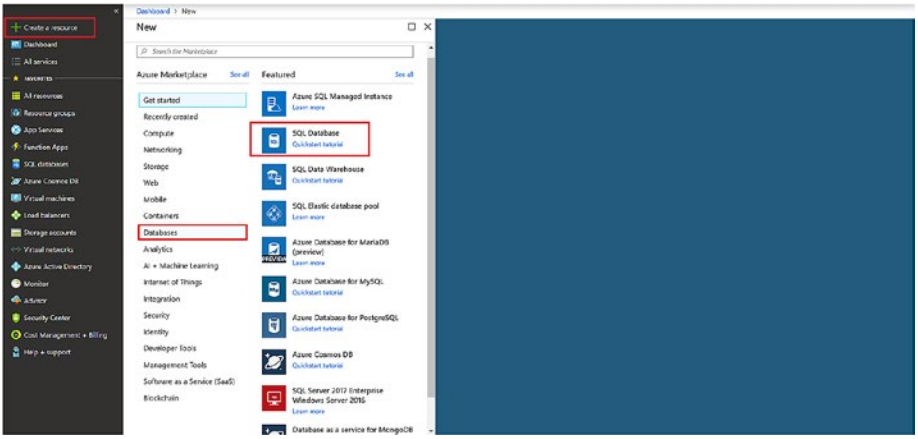

Figure 4-3. *Selecting SQL Database*

2. Provide the necessary details and set "Select source" to Sample (AdventureWorksLT). Create a server if it does not exist and then click Create to create the database with the content, as shown in Figure 4-4.

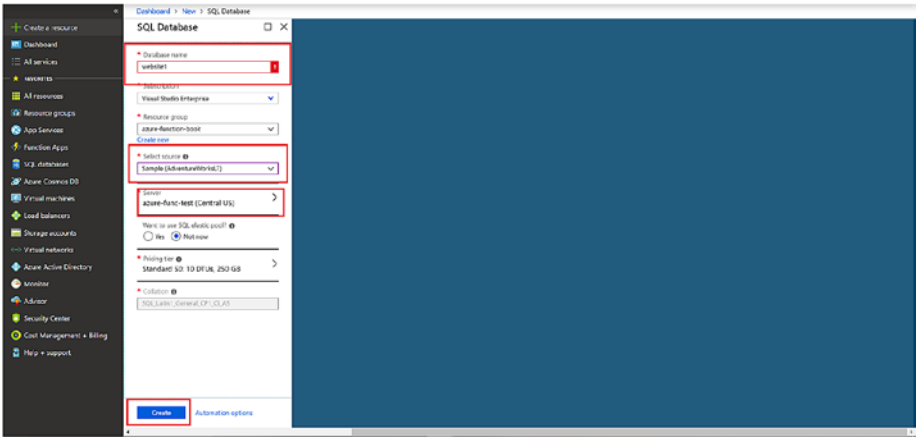

Figure 4-4. *Creating the database*

3. It will take some time for SQL Server and the database to be ready. Once it is ready, you will see it under the resource group you selected, as shown in Figure 4-5.

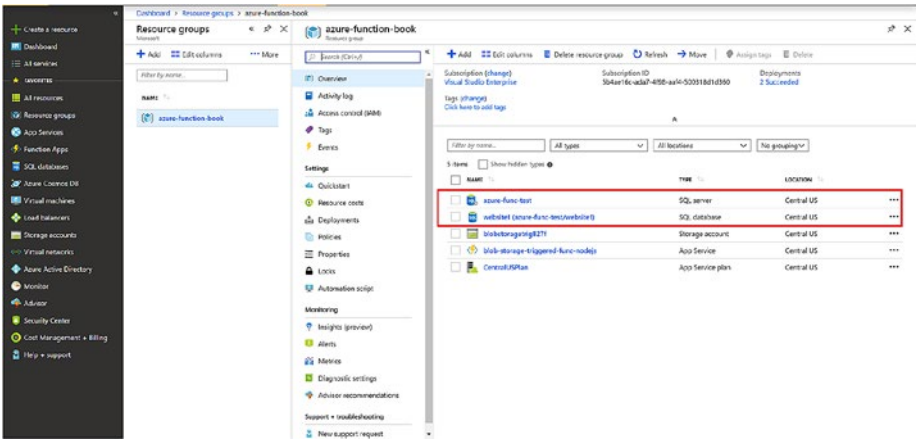

Figure 4-5. *Database created*

4. In the left panel, click "Query editor," provide your password to connect, and click OK, as shown in Figure 4-6.

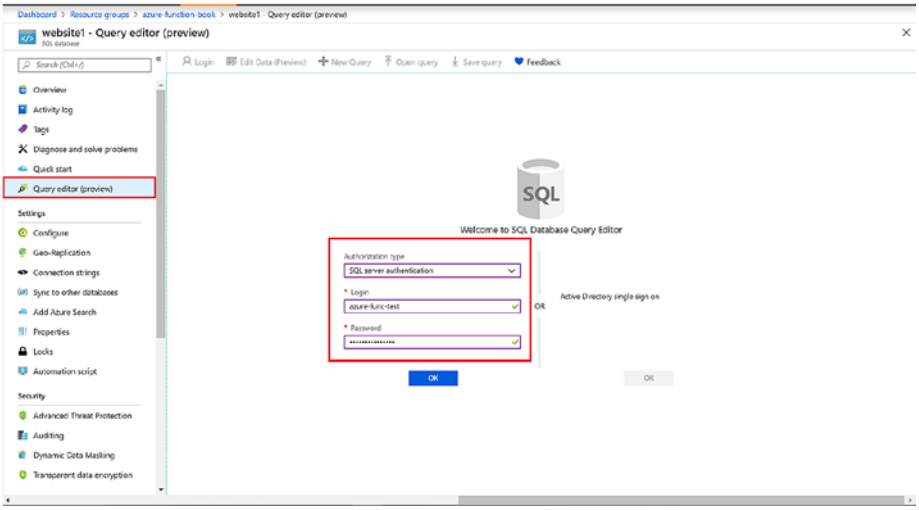

Figure 4-6. *Connecting to the database*

5. Once you have successfully logged in, you will see the Query Editor. In the left menu, click Tables, and you will see the tables being created, as shown in Figure 4-7.

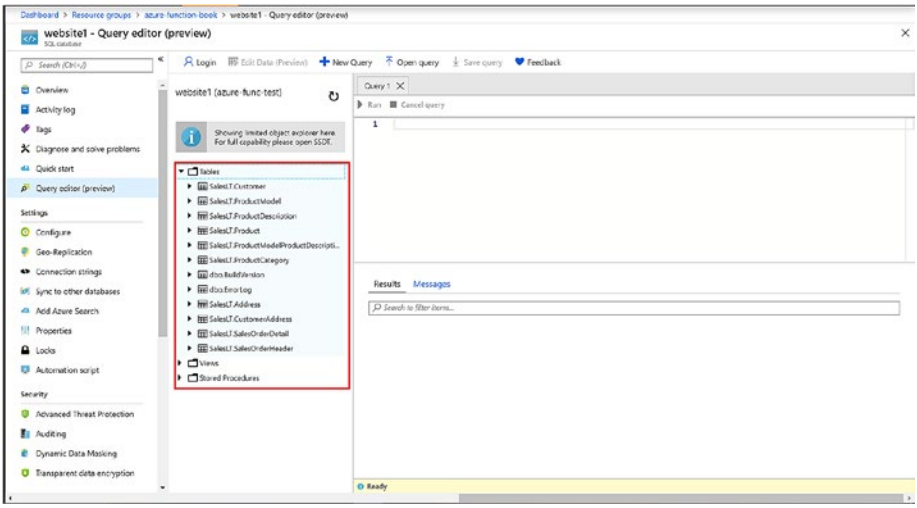

Figure 4-7. *Tables being created*

6. You can query the tables in a similar way as you do in SQL Server Management Studio.

Your database has been created with some initial data. Let's now create an HTTP-triggered function for it.

Creating an HTTP-Triggered Function Using C#

Now it's time to code.

1. In Chapter 3, you set up your machine to run Azure Functions using Visual Studio Code. If you have not set that up yet, please follow the steps in Chapter 3.

2. Go to Visual Studio Code and click the Azure icon (make sure that your Azure Functions extension version is 0.16.0) in the menu and then click New Folder. For this function, set the folder name to `HTTP-Triggered-Function` and then click Select, as shown in Figure 4-8.

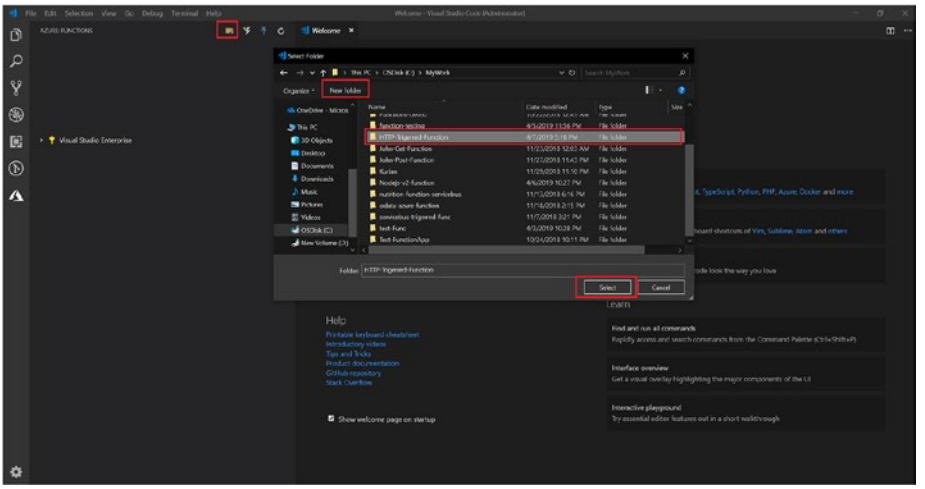

Figure 4-8. *Naming the function*

3. After clicking Select, you can select a language to be used for Azure Functions. Select C# as the language for this example, as shown in Figure 4-9.

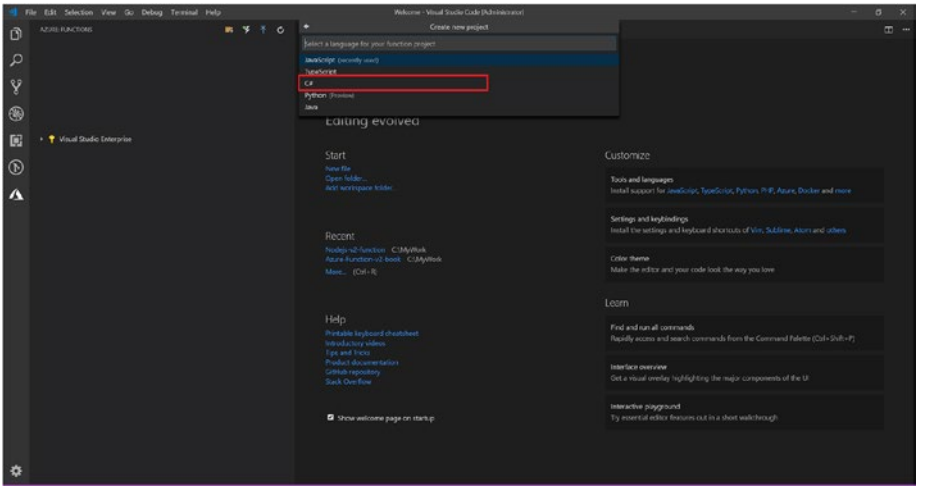

Figure 4-9. *Selecting the language*

4. Select HttpTrigger as the template for the function, as shown in Figure 4-10.

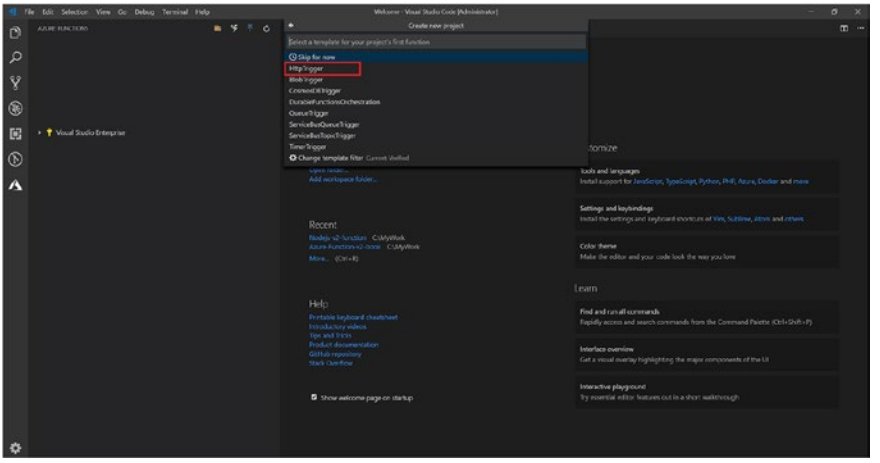

Figure 4-10. *Selecting the HttpTrigger template*

5. Provide a name for the function. By default, it is HttpTriggerCSharp. For this function, I am using the default value, as shown in Figure 4-11.

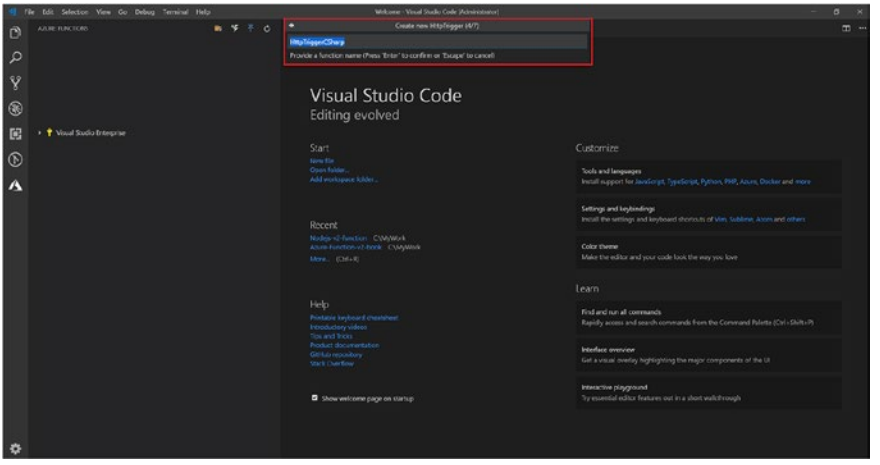

Figure 4-11. *Naming the function*

6. Provide the namespace of the function. By default, it is `Company.Function`, but for this function you will set it to `AzureFunctionBook.Function`, as shown in Figure 4-12.

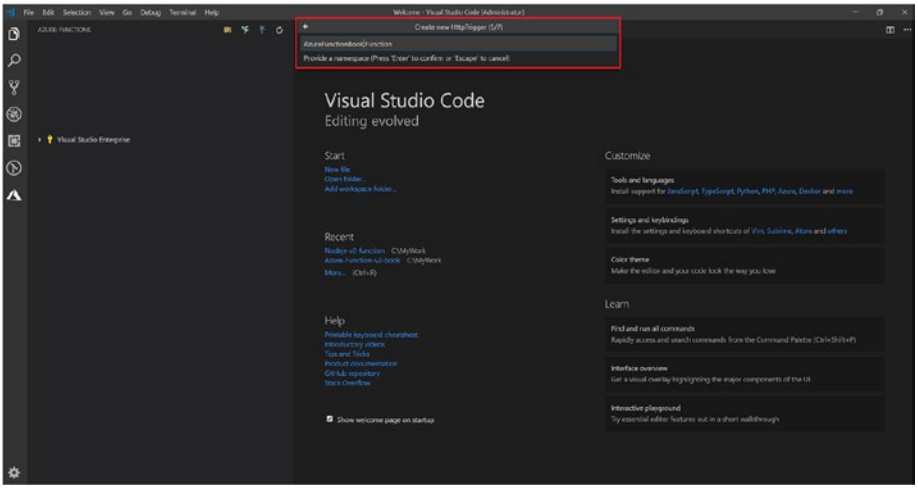

Figure 4-12. *Providing the namespace*

7. Set the access rights for this function. You will see three options: Anonymous, Function, and Admin. These different access rights determines what keys are required to invoke the function.

   -**Anonymous**: This means no API key is required.

   -**Function**: This is the default setting if nothing is selected. This means a function-specific API key is required.

   - **Admin**: This means a master key is required.

   For this function, use Anonymous as the access right, as shown in Figure 4-13.

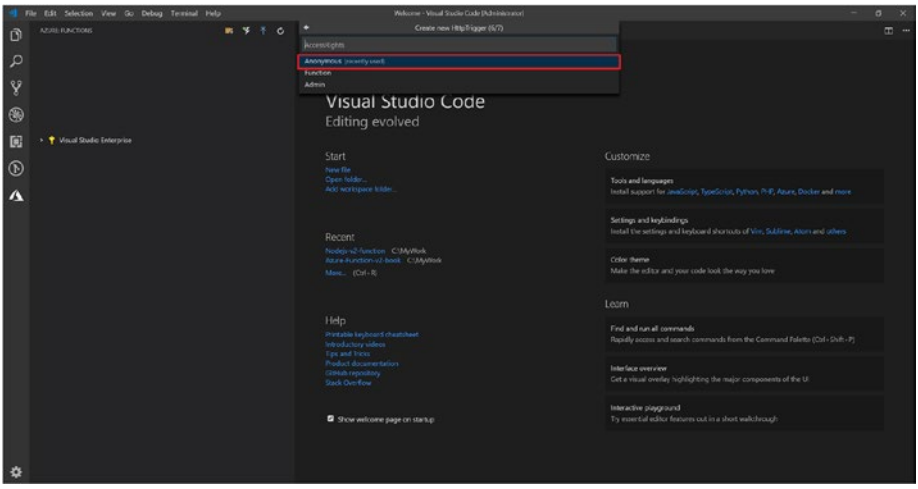

Figure 4-13. *Setting the access rights*

8. Let's add the `SqlClient` package to the solution. To do this, go to the Terminal, and in the top menu, select New Terminal. Type `dotnet add package System.Data.SqlClient --version 4.5.1` and press Enter. This will install the required package to your solution. To verify, go to the `.csproj` file, and you will see the package added, as shown in Figure 4-14.

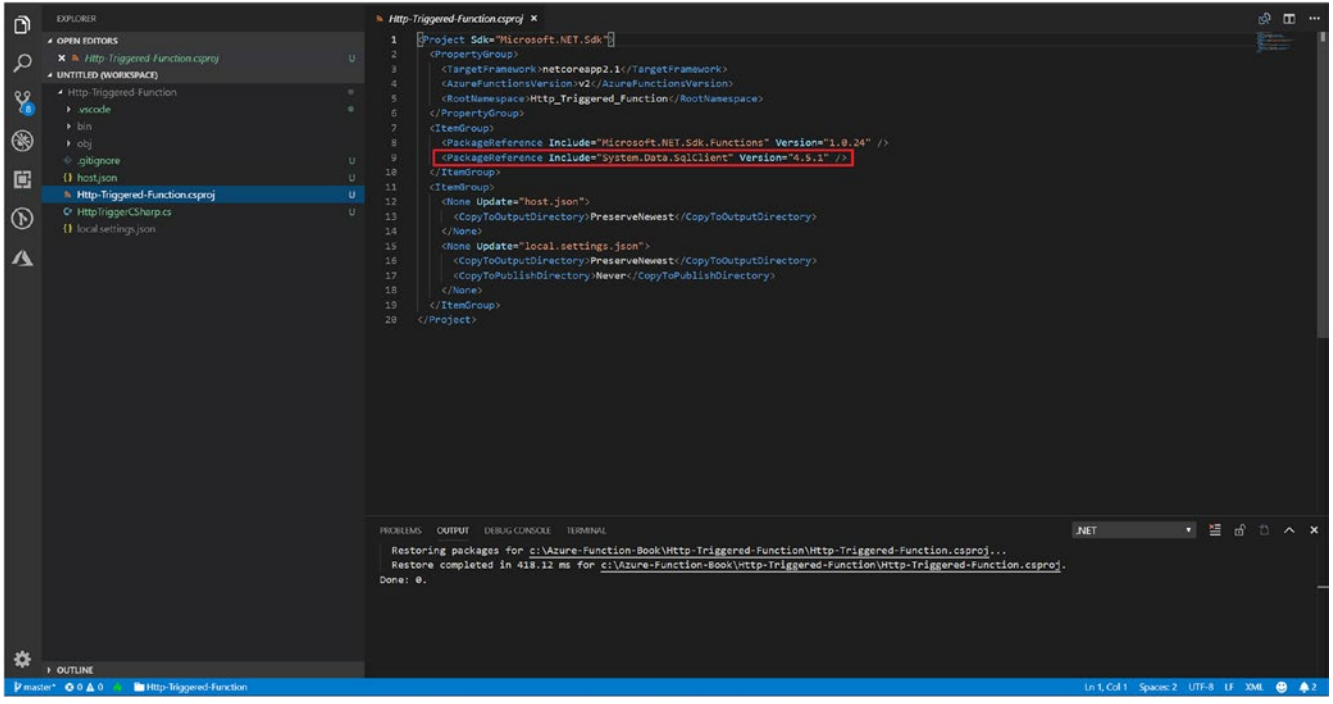

Figure 4-14. *Package added*

9. Now everything is set up. You will follow a code structure similar to what you would follow in normal projects. Create two folders, named `Helper` and `Models`. In `Models`, create the file `CustomerModel.cs`, and in `Helper` create the file `SqlClientHelper.cs`. You will use them for your function, as shown in Figure 4-15.

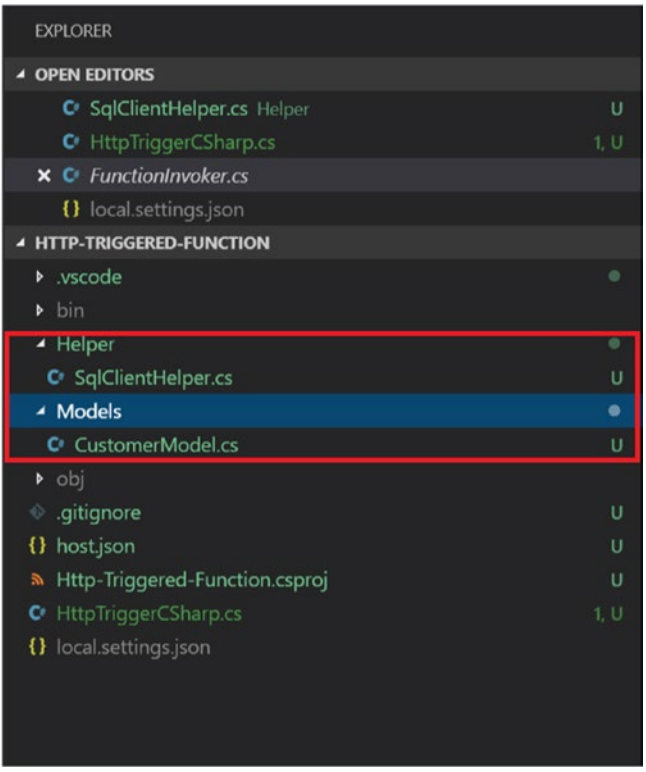

Figure 4-15. *Folders created*

10. Click the `CustomerModel.cs` file and paste the following code:

```
namespace Company.Function.Models
{
    public class CustomerModel
    {
        public int CustomerID { get; set; }

        public int NameStyle { get; set; }

        public string Title { get; set; }

        public string FirstName { get; set; }

        public string MiddleName { get; set; }

        public string LastName { get; set; }

        public string CompanyName { get; set; }
    }
}
```

11. Click the `SqlClientHelper.cs` file and paste the following code. The following code will make a call to the SQL Server database and get the customer details from the `SalesLT.Customer` table.

```
using System;
using System.Data;
using System.Data.SqlClient;
using Company.Function.Models;

namespace Company.Function.Helper
{
    public static class SqlClientHelper
```

```
{
    public static CustomerModel GetData(int
    customerId)
    {
        var connection = Environment.GetEnvironment
        Variable("coonectionString");
        CustomerModel customer = new
        CustomerModel();
        using (SqlConnection conn = new
        SqlConnection(connection))
        {
            var text = "SELECT CustomerID,
            NameStyle, FirstName, MiddleName,
            LastName, CompanyName FROM SalesLT.
            Customer where CustomerID=" +
            customerId;
            SqlCommand cmd = new SqlCommand
            (text, conn);
            // cmd.Parameters.AddWithValue
               ("@CustomerId", customerId);

            conn.Open();
            using (SqlDataReader reader = cmd.
            ExecuteReader(CommandBehavior.
            SingleRow))
            {
                while (reader.Read() && reader.
                HasRows)
                {
                    customer.CustomerID = Convert.
                    ToInt32(reader["CustomerID"].
                    ToString());
```

```
                    customer.FirstName = reader
                    ["FirstName"].ToString();
                    customer.MiddleName = reader
                    ["MiddleName"].ToString();
                    customer.LastName =
                    reader["LastName"].ToString();
                    customer.CompanyName = reader
                    ["CompanyName"].ToString();
                }

                conn.Close();
            }
        }

        return customer;
    }
}
}
```

12. Go to the main file `HttpTriggerCSharp.cs` and paste the following code. The following code is the function that will be triggered when you call it. It is first trying to get the `CustomerId` value from the query and convert it to `Int`. Then, it calls the `SQLClientHelper.GetData` method by passing the `CustomerId` value and returning the result.

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Company.Function.Helper;

namespace Company.Function
{
    public static class HttpTriggerCSharp
    {
        [FunctionName("HttpTriggerCSharp")]
        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.");

            int customerId = Convert.ToInt32(req.Query
            ["customerId"]);

            return (ActionResult) new OkObjectResult
            (SqlClientHelper.GetData(customerId));
        }
    }
}
```

If you look at the code that you have, you'll see you have created a basic customer profile function where you will get customer details based on `CustomerID`.

13. Select Debug in the top menu and click Start Debugging. You will see the function compiling in the Terminal, and once the function is compiled, you will see the local URL of the function, as shown in Figure 4-16.

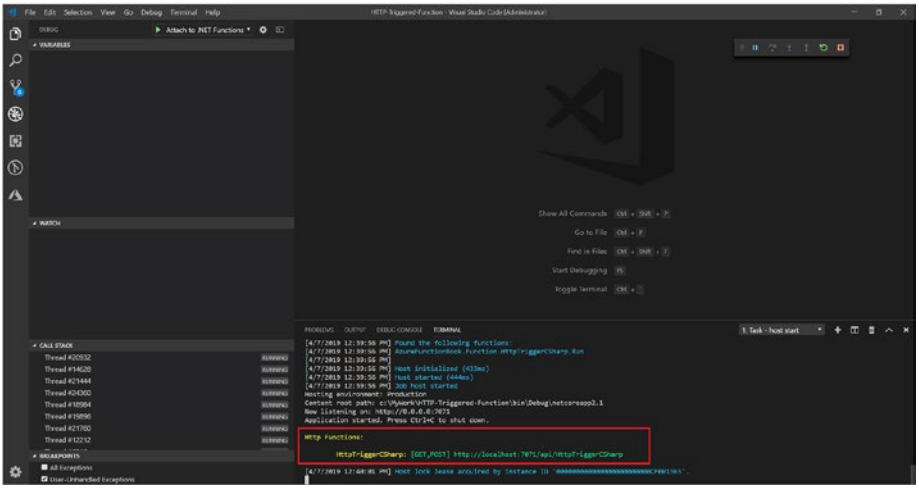

Figure 4-16. *Local URL*

14. Copy this URL, append `?customerId=1` to it, and hit Enter. You will get the output shown in Figure 4-17.

Figure 4-17. *Output*

Now you understand how you can create an HTTP-triggered API using Azure Functions. In the next section, you will look at how you can use Azure Functions as an OData API to access SQL Server.

Creating an HTTP-Triggered OData API for SQL Server Using Azure Functions

Before you start creating functions, you should first understand what OData is. OData stands for Open Data Protocol and defines a set of best practices for consuming and building web APIs.

Note For more information about OData, you can visit the official page at https://www.odata.org/.

To create a function, follow steps 1 to 6 in the previous section. The only change here is that instead of C# as the language, you will be using JavaScript as the language for the function.

1. Once the function is created, install the following `npm` packages:
   - `Azure-odata-sql`
   - `Async`
   - `Tedious`
   - `Tedious-connection-pool`
2. To install the packages, open Terminal and type the following, which will install the package for you:

   ```
   npm install <package name>
   ```

3. Once the packages are installed, create a file named `functions.js` and paste the following code. It basically connects to the SQL Server database, runs the query, and returns the data. In the following code, you are first creating the pool of SQL

connections using poolConfig. Then, you write the getSqlResults method, which fetches the records from the table.

```
// TEDIOUS
var ConnectionPool = require('tedious-connection-
pool');
var Connection = require('tedious').Connection;
var Request = require('tedious').Request;
var TYPES = require('tedious').TYPES;

// Pool Connection Config
var poolConfig = {
    min: 1,
    max: 10,
    log: true
};

//Connection Config
var config = {
    userName: process.env.databaseUser,
    password: process.env.databasePassword,
    server: process.env.databaseUrl,
    options: {
        database: process.env.databaseName,
        encrypt: true,
        requestTimeout: 0,
    }
};

//create the pool
var pool = new ConnectionPool(poolConfig, config);
```

```
pool.on('error', function (err) {
    console.error(err);
});

function getSqlResult(sqlObject, callback) {
    var result = []
    pool.acquire(function (err, connection) {
        if (err) {
            callback(err, null);
        }

        var request = new Request(sqlObject.sql,
        function (err, data) {
            if (err) {
                callback(err, null);
            }

            console.log(data);
            connection.release();
            callback(null, result);
        });

        sqlObject.parameters.forEach(element => {
            request.addParameter(`${element.name}`,
            TYPES.NVarChar, `${element.value}`);
        });

        request.on('row', (columns) => {
            var rowdata = new Object();
            columns.forEach((column) => {
                rowdata[column.metadata.colName] =
                column.value;
            });
```

```
            result.push(rowdata);
        });

        connection.execSql(request);
    });
}

module.exports = {
    getSqlResult: getSqlResult,
}
```

4. Go to the main file `index.js` and paste the following code. In the following code, you are first configuring the table and schema that will be used in this OData API. Then you are fetching the `pageSize`, filters, selection, ordering, and so on, from the query parameters. Once you get all this, you prepare the query and call `azureOdata.format` to convert this to a proper SQL query.

```
var azureOdata = require('azure-odata-sql');
var async = require('async');

var tableConfig = {
    name: 'Customer',
    schema: 'SalesLT',
    flavor: 'mssql',
};

var defaultPageSize = 30;

module.exports = function(context, req) {
    var module = require('./functions');
```

```
var pageSizeToUse = req.query !== null && req.
query.$pageSize !== null && typeof req.query.
$pageSize !== "undefined" ? req.query.$pageSize :
defaultPageSize
var getSqlResult = module.getSqlResult;
var query = {
    table: 'Customer',
    filters: req.query !== null &&
    req.query.$filter !== null && typeof
    req.query.$filter !== "undefined" ?
    req.query.$filter : '',
    inlineCount: "allpages",
    resultLimit: pageSizeToUse,
    skip: req.query !== null && req.query.$page !==
    null && typeof req.query.$page !== "undefined"
    ? pageSizeToUse * (req.query.$page -1): '',
    take: pageSizeToUse,
    selections: req.query !== null && req.query.
    $select !== null && typeof req.query.$select
    !== "undefined" ? req.query.$select : '',
    ordering: req.query !== null && req.query.
    $orderby !== null && typeof req.query.$orderby
    !== "undefined" ? req.query.$orderby :
    'CustomerID',
};

var statement = azureOdata.format(query,
tableConfig);
var calls = [];
var data = [];
async.series([
    function (callback) {
        getSqlResult(statement[0], (err, result) => {
```

```
            if (err)
                throw err;

            data.push(result);
            callback(err, result);
        });
    },
    function (callback) {
        getSqlResult(statement[1], (err, result) => {
            if (err)
                throw err;

            data.push(result);
            callback(err, result);
        });
    }
],
    function (err, result) {
        if (err) {
            console.log(err);
        } else {
            var count = result[0].length;
            context.res = {
                status: 200,
                body: {
                // '@odata.context': req.protocol
                   + '://' + req.get('host') + '/
                   api/$metadata#Product',
                'value': result[0],
                'total': result[1][0].count,
                'count': count,
```

```
                    'page': req.query !== null && req.
                    query.$page !== null ? req.query.
                    $page : 1
                    },
                    headers: {
                        'Content-Type': 'application/
                        json'
                    }
                };
            }

            context.done();
        });
}
```

5. You first prepared the query inside `var query = {}`. Now, you will convert this into a SQL-understandable query by calling `azureOdata.format(query, tableConfig)`. Once the query is converted to a SQL query, you will pass this to the function you wrote in `functions.js`, which will run the SQL statements and return the data.

6. Run Azure Functions by going to the top menu and clicking Debug and then Start Debugging. Once the function starts, you will make an HTTP call.

   ```
   http://localhost:7071/products?$filter=CustomerID
   eq 1&$select=CustomerID,FirstName,LastName
   ```

   If you look at this URL, you will see two query parameters. One is `$filter`, which gets converted to a `WHERE` clause, and one is `$select`, which gets converted to a `SELECT` statement.

Once you run the previous query, you will get the result shown in Figure 4-18.

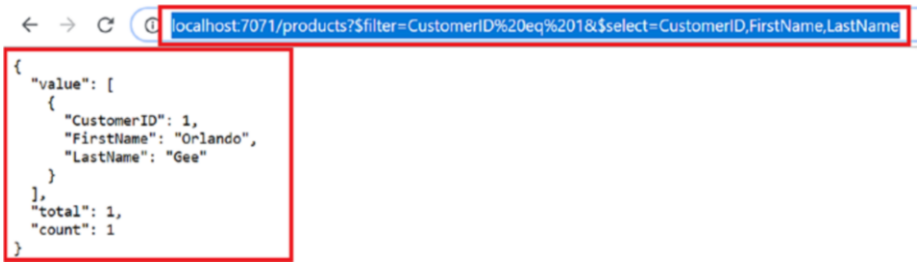

Figure 4-18. *Output of query*

With this you have created two HTTP-triggered functions; one is a normal HTTP-triggered function with C# and another one is an advanced HTTP-triggered function using OData with NodeJs.

Let's look at proxies in the next section.

Overview of Proxies in Azure Functions

Proxies are one of the most important features of Azure Functions. With the help of proxies, you can divide a large API into small functions, but for the end customer, it still shows as a single API with one endpoint.

This not only simplifies the use of the API by other customers but also reduces the burden on the customers to call individual APIs with different URLs.

The following are features of an Azure Functions proxy:

- You can modify request and response queries using variables.
- You can modify request and response queries by referencing application settings.
- You can troubleshoot an Azure Functions proxy.

Now let's try to create a proxy from Azure Portal.

Creating a Proxy Using Visual Studio Code

Let's get started.

1. Go to the previous OData function and click the vertical menu. You will see a file named `proxies.json`, as shown in Figure 4-19.

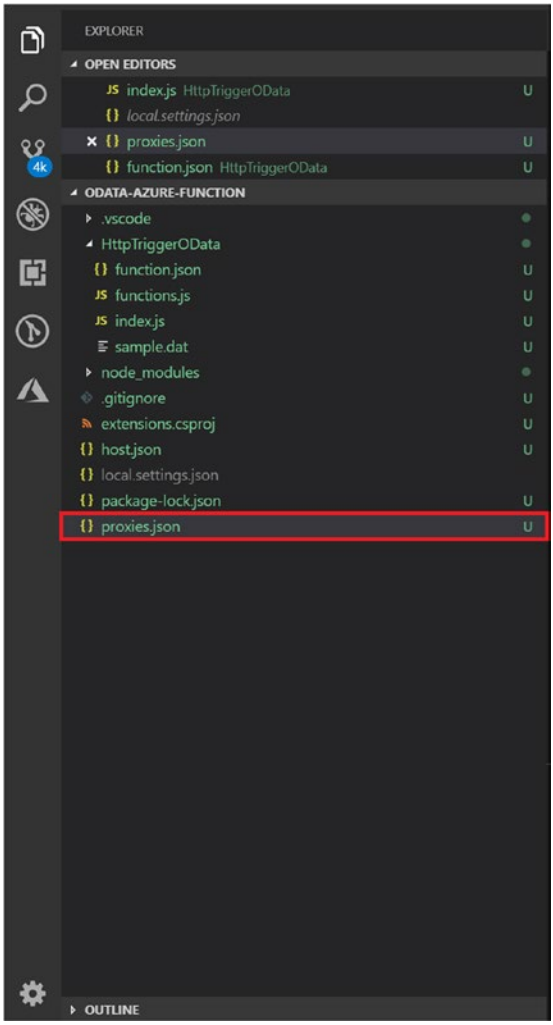

Figure 4-19. *Listing of files*

2. Click the `proxies.json` file and paste the following code. Internally you are calling the same functions, but you can expose different URLs to the customer so no one is aware of the exact function app name and location. You can change the `backendUri` value to call another function.

```
{
{
  "$schema": "http://json.schemastore.org/proxies",
  "proxies": {
    "proxy1": {
      "matchCondition": {
        "methods": [
          "GET"
        ],
        "route": "/api/customer"
      },
      "backendUri": "http://localhost:7071/api/Http
      TriggerOData"
    }
  }
}
```

3. Run Azure Functions, and in the Terminal you will see that Azure Functions is now providing two endpoints, as shown in Figure 4-20.

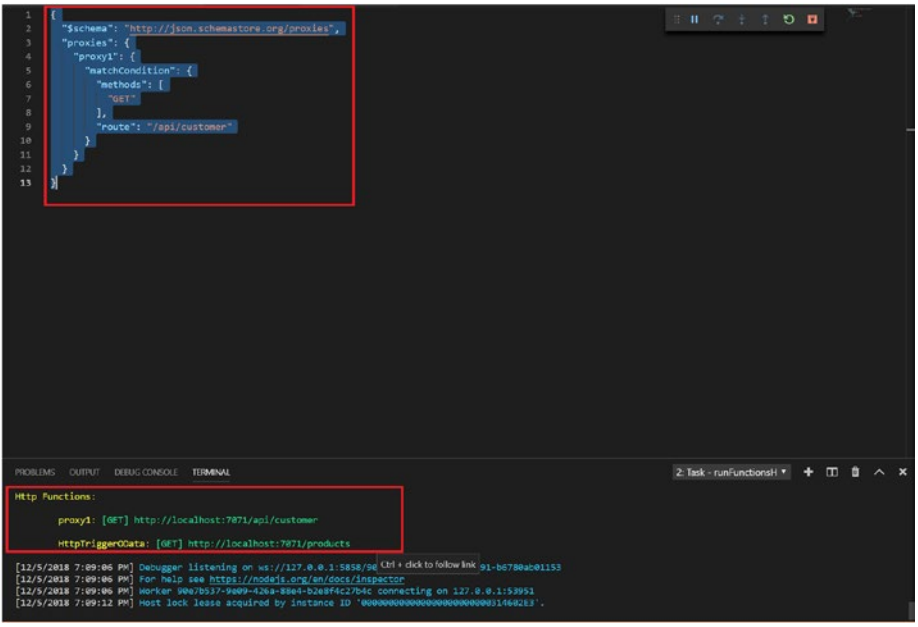

Figure 4-20. *Two endpoints*

4. Let's hit the proxy from the browser and check the output. The output is shown in Figure 4-21.

Figure 4-21. *Output of proxy*

As you can see, your proxy is working fine, and the output is the same as you got in the previous section. So, that's how you can create a proxy by using Visual Studio Code. Let's look at how you can do the same thing from Azure Portal.

Creating a Proxy Using Azure Portal

Here is the process:

1. Go to Azure Portal and click Function Apps, as shown in Figure 4-22.

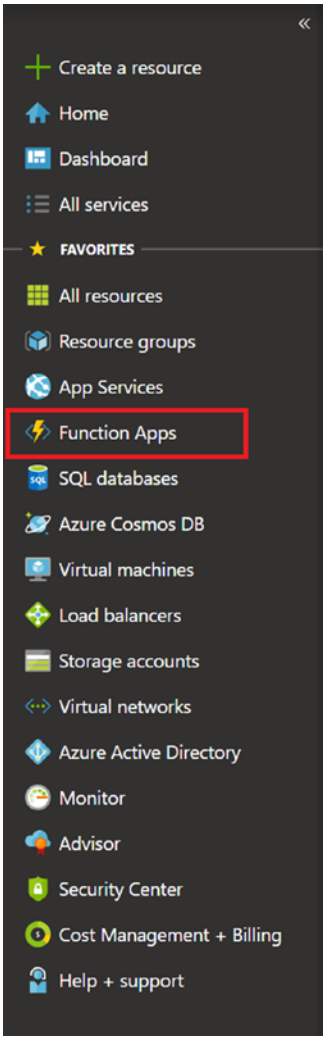

Figure 4-22. *Selecting Function Apps*

2. Select the function app and click Proxies, as shown in Figure 4-23.

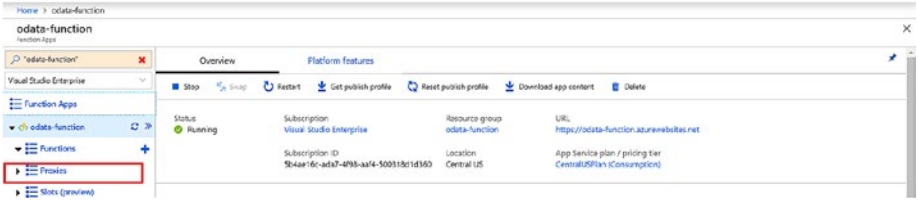

Figure 4-23. *Clicking Proxies*

3. Click + near the Proxies tab and provide details for the proxy. In the back-end URL, provide the URL of the function where you want the request to navigate to, as shown in Figure 4-24.

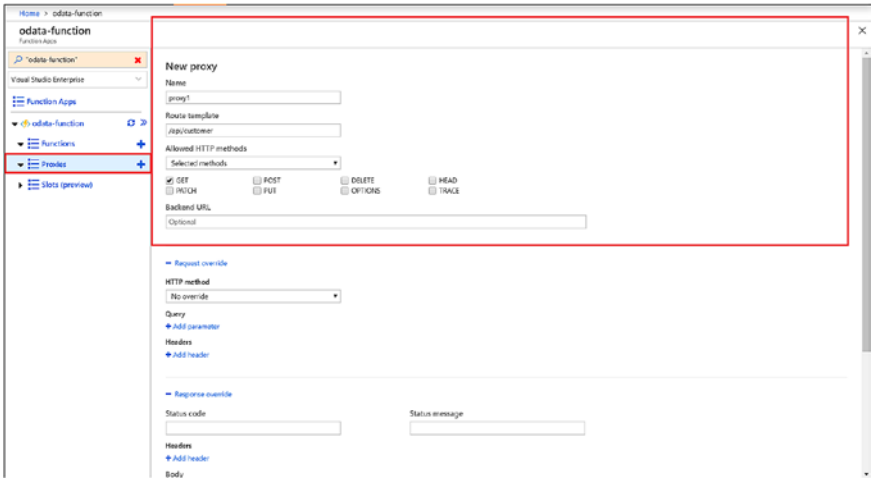

Figure 4-24. *URL of function*

This is how you create a proxy from Azure Portal. This brings us to the end of the chapter. In the next chapter, you will look at the Durable Functions extension and how you can use durable functions for long-running tasks.

CHAPTER 5

Azure Durable Functions

In this chapter, I will cover the following topics:

- Overview of the Durable Functions extension
- Bindings for the Durable Functions extension
- Performance and scale of durable functions
- Creating durable functions using Azure Portal
- Disaster recovery and geodistribution of durable functions

Overview of Durable Functions

Durable Functions is an extension to Azure Functions that allows you to write stateful functions in a serverless environment by managing checkpoints, state, and restarts for you.

Durable Functions uses a new type of function called an *orchestrator function*, which lets you define the stateful workflows in code and allows you to call other functions both synchronously and asynchronously.

The main use case of Durable Functions is to simplify stateful coordination problems in the serverless world.

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_5

Types of Functions

The Durable Functions extension allows the stateful orchestration of functions. Each function is made up of combination of different functions. Each of these functions plays a different role in orchestration, as shown in Figure 5-1.

Figure 5-1. *The three types of function*

There are basically three types of functions.

- **Client functions**: These types of functions are the entry point for creating an instance of a durable function. They are triggered functions that create a new instance of an orchestration process. Client functions can be triggered by any available trigger in Azure Functions. Also, client functions have an orchestration binding that allows them to manage durable orchestrations.
- **Orchestrator functions**: These are the heart of durable functions and describe the order in which actions are executed. An orchestrator function must be triggered by an orchestration trigger (a client function with orchestration binding). Each instance of an orchestrator has an instance identifier that can be autogenerated or user-generated and is used to manage instances of orchestration.

- **Activity functions**: These are the basic unit of work in Durable Functions orchestration and are the functions and tasks that are being orchestrated or ordered in the process. For example, you can create a durable function for order cancellation to handle canceling the shipment, updating the inventory, and refunding the payment. Each of these tasks will be an activity function, and the output of one function can be used as the input of another. An activity function must be triggered by an activity trigger.

Durable Function Patterns

The Durable Functions extension basically caters to five application patterns.

- Function Chaining
- Fan-Out/Fan-In
- Async HTTP APIs
- Monitoring
- Human Interaction

Function Chaining

Function Chaining is a pattern where you execute functions in a sequential order. Also, you use Function Chaining when the output of one function has to be used as the input of another function.

Let's see an example of e-commerce order processing. First, a customer orders a product, and after that, internally you process the order and notify the dealer. Once the dealer confirms that the product is ready to

be shipped, you notify the delivery service to pick up the order and ship it. Once the product is shipped, you notify the customer. This whole process can be done using Function Chaining, as shown in Figure 5-2.

Figure 5-2. *Function Chaining example*

The following simple code will call the Function Chaining pattern using C#:

```
public static async Task<object>
Run(DurableOrchestrationContext context)
{
    try
    {
        var orderProcessedResult = await context.CallActivity
        Async<object>("ProcessOrder");
        var dealerNotificationResult = await context.CallActivity
        Async<object>("NotifyDealer", orderProcessedResult);
        var deliveryServiceResult = await context.CallActivity
        Async<object>("NotifyDeliveryService", dealer
        NotificationResult);
        return  await context.CallActivityAsync<object>("Notify
                Customer", deliveryServiceResult);
    }
    catch (Exception ex)
    {
```

```
        // This will be the 5th function which will rollback
           all the operations before the function which caused
           the error
        await context.CallActivityAsync<object>("Rollback", null);

        context.log("Error cannot be processed");
    }
}
```

Fan-Out/Fan-In

The Fan-Out/Fan-In pattern refers to executing multiple functions in parallel and then waiting for all of them to execute. Usually some aggregation work is done on the result returned by multiple functions.

With normal functions in Azure Functions, fanning out can be done by publishing multiple messages to the queue. But the fanning in part is complicated because you have to keep track of when the message is picked up and processed and store the result. This is a difficult task to achieve in Azure Functions, but the Durable Functions extension handles this pattern quite easily.

Let's look at an example where you have replenished the stock and want to notify all the customers who selected "notify me once the product is available," as shown in Figure 5-3.

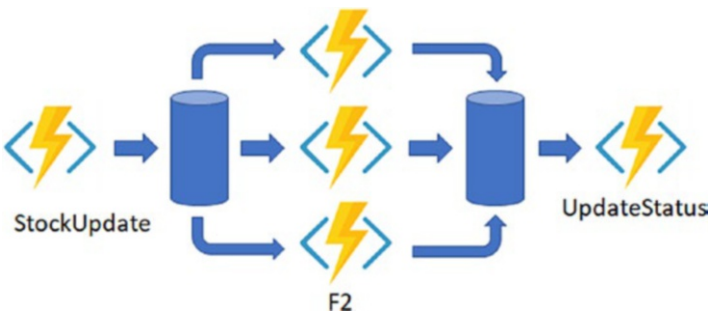

Figure 5-3. *Fan-Out/Fan-In example*

This first function updates or replenishes the stock, i.e., products. Then you call the F2 function for each product that was out of stock and call multiple functions. One function will send an e-mail, the other one will send an SMS message to the customer, and one will stack the product based on the user interest shown as per the "notify me once the product is available" selection. Once you get a response from all three functions, you call the `UpdateStatus` function, which will update the notification status corresponding to each user who opted for notification.

```
public static async Task Run (Durableorchestrationcontext ctx)
{
var parallelTasks = new List<Task<int>>();

object []workBatch = await ctx.CallFunctionAsync<object[]>
("StockUpdate");
for (int i = 0; i < workBatch.Length; i++)
{
Task<int> task = ctx.CallFunctionAsync <int> ("F2", workBatch [i]);
parallelTasks.Add (task);
}
await Task.WhenAll(parallelTasks);
//aggregate result of all tasks and send result to UpdateStatus
int sum = parallelTasks.Sum(t=> t.Result);
await ctx.CallFunctionAsync ("UpdateStatus", sum);
}
```

Async HTTP APIs

The Async HTTP APIs pattern takes care of the problem of keeping the state of long-running processes with the external clients. The common way to implement this pattern is to trigger the long-running job with the HTTP client and then redirect the external client to another page, which keeps on polling the state of the long-running job.

The Durable Functions extension provides you with a built-in capability that simplifies the code you will write for interacting with long-running processes. Since the Durable Functions runtime manages the state, you don't have to implement your own state-tracking mechanism.

Let's look at an example of a food-ordering app. You order your food, and the app takes you to a page where you track the status of the order. The first state is whether the order is accepted by the restaurant. Once the order is accepted, it starts showing you the time it will take for the order to be prepared by the restaurant, and then once the order is ready and picked up by the delivery person, it shows you a map with the location of the delivery person. You can implement this with the help of Durable Functions, as shown in Figure 5-4.

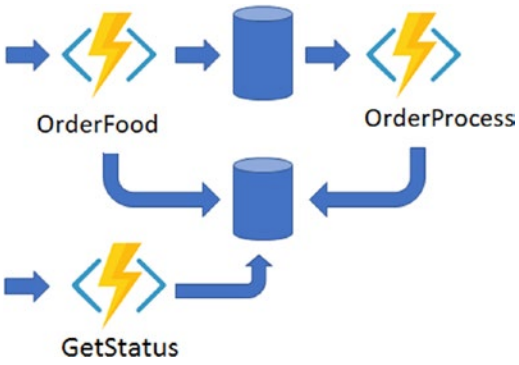

Figure 5-4. *Async example*

As you can see in Figure 5-4, the `OrderFood` function will act as an HTTP API that will be called once the end user clicks Order Food. The `OrderFood` function will check for the validity of the order and will call the `OrderProcess` function. This function will keep on updating the status of the order.

You will redirect the end user to the order-tracking page, which will poll the `GetStatus` function and will show the order status.

The following is the demo code depicting the creation of an orchestrator function for `OrderProcess`. The following code is part of the HTTP-triggered `OrderFood` function.

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

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

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

Monitoring

The Monitoring pattern is used when you need polling until a condition is met. Normally a regular timer trigger (a timer trigger lets you run a function on a specified schedule) can be used for scenarios such as a cleanup job, but the problem with this is that the time interval is static, so managing the lifetime of the instances becomes complex.

The Durable Functions runtime, on the other hand, comes with flexible intervals and lifetime management of tasks. It allows you to create multiple monitor processes from a single orchestration. See Figure 5-5.

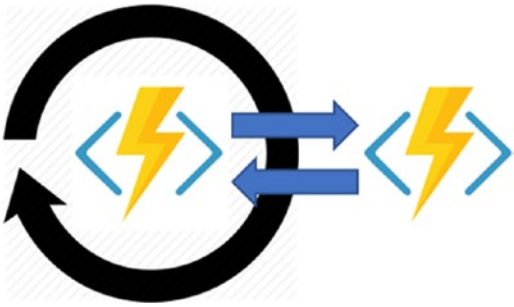

Figure 5-5. *Monitoring example*

Human Interaction

Usually you will automate processes that require no human intervention because people are not as highly available and responsive as cloud services. But, in certain scenarios that require approval, human intervention is required, so the automated processes must account for that.

Automated processes generally do this by using timers and compensation logic. Let's look at a "leave approval" workflow as an example. In this case, an employee applies for a leave. The notification goes to the manager to approve it. Here you can have two scenarios. One is if the manager does not approve it within 48 hours, the leave will be automatically approved. The other scenario is if the manager does not approve it within 48 hours, then it is escalated to the manager's manager. See Figure 5-6.

Figure 5-6. *Human Interaction pattern example*

Here is the example code:

```
public static async Task Run(DurableOrchestrationContext context)
{
      await context.CallActivityAsync("SubmitLeaveRequest");
      using (var timeoutCts = new CancellationTokenSource())
      {
            DateTime dueTime = context.CurrentUtcDateTime.
            AddHours(48);
            Task durableTimeout = context.CreateTimer(dueTime,
            timeoutCts.Token);
            Task<bool> approveLeaveEvent = context.WaitFor
            ExternalEvent<bool>("ApproveLeaveEvent");
            if (approveLeaveEvent == await Task.WhenAny(approve
            LeaveEvent, durableTimeout))
            {
                  timeoutCts.Cancel();
                  await context.CallActivityAsync("ProcessLeave
                  Approval", approveLeaveEvent.Result);
            }
```

```
            else
            {
                  await context.CallActivityAsync
                  ("EscalateEvent");
            }
      }
}
```

Bindings for Durable Functions

The Durable Functions extension introduces two new trigger bindings that control the execution of orchestrator and activity durable functions. The Durable Functions extension also introduces one output binding that acts as a trigger for the Durable Functions runtime.

Activity Triggers

An activity trigger enables you to author functions that are called by orchestrator functions. Activity functions are like any other normal function. The only difference is that you will have `ActivityTrigger`, which is triggered from the orchestrator function.

Internally, the following activity trigger binding keeps polling a series of queues in the default storage account of the function app. The queues are internal implementations of the extension, so that's why they are not part of the orchestrator trigger binding.

The activity trigger is defined by the following JSON object in the bindings array:

```
{
      "name": "Input parameter name",
      "activity": "<Optional parameter. Name of the activity",
      "type": "activityTrigger",
      "direction": "in"
}
```

Here is the trigger behavior:

- **Threading**: An activity trigger is like any other function in Azure Functions that you code and has no limitation on threading or I/O.
- **Message visibility**: The messages are dequeued and kept invisible for a configurable amount of time. As long as the function app is running and is in a healthy state, the visibility of the messages is renewed automatically.
- **Return values**: The return values are JSON serialized and are persisted in the Azure Storage orchestration history table.

The following is the basic code for an activity trigger:

```
[FunctionName("City_Travel")]
public static string Run([ActivityTrigger] string cityName,
TraceWriter log)
{
      log.Info($"I am travelling to {cityName}.");
      return $"I am travelling to {cityName}!";
}
```

Orchestration Triggers

As the name suggests, an orchestration trigger enables you to author orchestrator functions. The trigger allows you to start new instances of orchestrator functions and also allows you to resume existing instances of orchestrator functions that are awaiting a task.

Behind the scenes, the following orchestrator trigger binding keeps polling a series of queues in the default storage account of the function app. The queues are internal implementations of the extension, so that's why they are not part of the orchestrator trigger binding.

The orchestrator trigger is defined by the following JSON object in the bindings array:

```
{
      "name": "Input parameter name",
      "orchestration": "Optional parameter - Name of
      orchestration",
      "type": "orchestrationTrigger",
      "direction": "in"
}
```

Here is the trigger behavior:

- **Single threading**: For all orchestrator functions running on a single host instance, a single dispatcher thread is used. For this reason, the orchestrator function code should not perform any I/O. Also, this thread should not do any async work except when awaiting on Durable Functions–specific task types. JavaScript orchestrator functions should never be declared async.

- **Message visibility**: The messages are dequeued and kept invisible for a configurable amount of time. As long as the function app is running and is in a healthy state, the visibility of the messages is renewed automatically. Orchestration triggers do not support poison message handling.
- **Return values**: The orchestrator return values are JSON serialized and are persisted in the Azure Storage orchestration history table.

The following is the basic code for an orchestrator trigger:

```
[FunctionName("Orchestrator_City")]
public static async Task<List<string>> Run(
[OrchestrationTrigger] DurableOrchestrationContext context)
{
      var outputs = new List<string>();

      outputs.Add(await context.CallActivityAsync<string>
      ("City_Travel", "Hyderabad"));
      outputs.Add(await context.CallActivityAsync<string>
      ("City_Travel", "New York"));
      outputs.Add(await context.CallActivityAsync<string>
      ("City_Travel", "Delhi"));

      // returns
      // "I am travelling to Hyderabad"
      // "I am travelling to New York"
      // "I am travelling to Delhi"

      return outputs;
}
```

As you can see, the previous orchestrator function is calling the activity function `City_Travel` and is passing the name of the city to it. From the way it is written, it looks like the orchestrator function is calling the `City_Travel` activity function directly, but actually it is sending a message to a work-item queue. The activity function `City_Travel` polls the queue, and as soon as it receives the message in the queue, it executes the logic.

Once the activity function completes the logic execution, it sends the response message to the control queue that the orchestrator function is polling. As the orchestrator function `Orchestrator_City` receives the message via `OrchestrationTrigger`, it shows the response. This is the behavior of the durable function.

Once you start the durable function, it creates four control queues and one `workitems` queue, as shown in Figure 5-7.

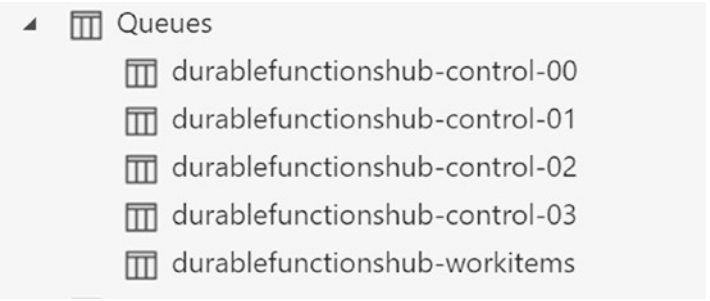

Figure 5-7. *Durable function queues*

It also creates two Azure Storage tables, named `DurableFunctionsHubHistory` and `DurableFunctionsHubInstances`.

Orchestration Client

The orchestrator client is responsible for starting/stopping the orchestrator function. It is also used to query the status, send events, and purge instances of the history of the orchestrator function.

The orchestrator client binding actually allows you to write functions in Azure Functions that interact with orchestrator functions.

The orchestrator client trigger is defined by the following JSON object in the bindings array:

```
{
      "name": "Name of Input Parameter",
      "taskHub": "Optional Parameter. name of the task hub",
      "connectionName": "Optional Parameter. Name of the
      connection string in the app settings",
      "type": "orchestrationClient",
      "direction": "in"
}
```

The following is the basic code for the orchestration client:

```
[FunctionName("OrchestrationClient_Start")]
public static async Task<HttpResponseMessage> HttpStart(
[HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")]
HttpRequestMessage req,
[OrchestrationClient]DurableOrchestrationClient starter,
TraceWriter log)
{
      string instanceId = await starter.StartNewAsync
      ("Orchestrator_City", null);
      log.Info($"Running orchestration with ID =
      '{instanceId}'.");
      return starter.CreateCheckStatusResponse
      (req, instanceId);
}
```

Performance and Scaling of Durable Functions

The Durable Functions extension has unique scaling characteristics that need to be understood to be able to scale and improve performance. To understand the scaling behavior, you have to first understand some of the underlying details of the Azure Storage provider.

History Table

The history table contains the history events for all the orchestration instances running within a task hub. The name of the table is in the format `TaskHubNameHistory`. The partition key of this table is derived from the instance ID of the orchestration function. Since the instance ID is generated randomly, it ensures optimal distribution of internal partitions in an Azure Storage table. As the orchestrator function instances run, new rows are added to this table.

When an orchestration instance runs, first the appropriate rows of the history table are loaded into the memory. These history events are then replayed in the orchestrator function to get back to the previous checkpoint state. This is influenced by the Event Sourcing pattern.

Instance Table

This table contains the statuses of all the orchestrations running within a task hub. The orchestration function instance ID is the partition key of this table, and the row key is a fixed constant. There is one row per orchestration function instance.

This table is consistent with the content of the history table. This table is used by the `GetStatusAsync` (.NET) API and the `getStatus` (JavaScript) API. Also, it is used by the HTTP status query API.

Using a separate table to efficiently satisfy the instance query operation in this way is influenced by the Command and Query Responsibility Segregation (CQRS) pattern.

Internal Queue Triggers

Activity functions and orchestrator functions are both triggered by the queues in the task hub of the Azure Functions app. This provides an "at-least-once" delivery guarantee of messages. There are two types of queues in Durable Functions.

- **Control queue**: In Durable Functions, there are multiple queues per task hub. Control queues are more sophisticated than work-item queues because control queues trigger the stateful orchestrator functions. Orchestrator messages are load balanced across the control queue. In a single poll, a message can dequeue as many as 32 messages, and if all those messages belong to a single orchestrator, they are processed as a batch.
- **Work-item queue**: Per task hub there is one work-item queue in Durable Functions. This queue behaves like a normal queue. This queue triggers the stateless activity functions by dequeueing a single message at a time. When a durable function scales out to multiple VMs, each VM competes to acquire work from the work-item queue.

Since you now have an understanding of the underlying mechanism, let's look at how to scale durable functions.

Orchestrator Scale-Out

Stateless functions like activity functions can be scaled out easily by adding more VMs, but stateful functions like orchestrator functions are partitioned across one or more queues for them to scale out. By default, a task hub can have at most 16 partitions, and by default the partition count is 4. The number of control queues is defined in the `host.json` file for a function running on the 2.0 runtime, as shown here:

```
{
"extensions": {
      "durableTask": {
            "partitionCount": 2
      }
  }
}
```

When you scale out the orchestrator function to multiple instances, each instance acquires a lock on one of the control queues, and this way it ensures that each orchestration instance runs on a single host instance at a time. In the previous example, a task hub will have two control queues, so an orchestration instance can be load balanced across as many as five VMs. Additional VMs can be added to increase the capacity of activity functions. Generally, orchestration functions are intended to be lightweight, so they should not require more computing power. It is therefore advisable to create not more than two to five control queues.

Figure 5-8 depicts how Azure Functions behaves in a scaled-out manner.

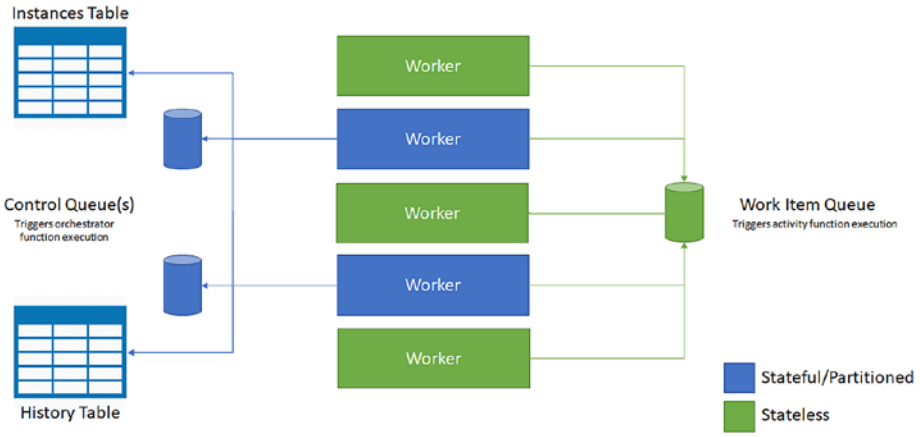

Figure 5-8. *Azure Functions behavior when scaling out*

As you can see in Figure 5-8, all instances compete for the work from the work-item queue, but only two instances at a time can acquire messages from the control queue, and each instance locks the single control queue.

Autoscaling

Durable Functions supports autoscaling via the scale controller. The scale controller monitors the rate of events and decides whether to scale in or scale out. In the case of Durable Functions, the scale controller monitors the latency of each queue by issuing a peek command. If the message latencies are higher than the threshold, then the scale controller will keep adding the instances until it reaches the partition count.

In the case of work-item queues, the scale controller will keep adding the VM instances if the message latencies exceed the threshold irrespective of the partition count. The maximum number of instances it can add is 200.

Concurrency Throttling

Azure Functions allows you to run multiple functions concurrently within a single app instance. The concurrency increases the parallel execution and reduces the number of "cold starts." But you should also be mindful of the fact that high concurrency results in high per-VM memory usage.

Orchestrator and activity functions both support concurrency, and their limits can be set in `host.json`. The setting for an activity function is `maxConcurrentActivityFunctions` and for an orchestrator function is `maxConcurrentOrchestratorFunctions`.

```
{
      "extensions": {
            "durableTask": {
                  "maxConcurrentActivityFunctions": 20,
                  "maxConcurrentOrchestratorFunctions": 20
            }
      }
}
```

By default the number of activity and orchestrator function executions is capped at ten times the number of cores on the VM.

Orchestrator Function Replay

As you know, orchestrator functions are stateful functions, and they replay to the checkpoint using the contents of the history table. The orchestrator function code is replayed every time a batch of messages is dequeued from the control queue by default.

Durable Functions provides an ability to decrease the aggressive behavior of the replay by using extended sessions. When you enable extended sessions, the function instances are held in memory for that time, and you can process message without a full replay.

Enabling extended sessions reduces the I/O against the Azure Storage table and thus increases the throughput. You can enable extended sessions by setting `extendedSessionsEnabled` to true. To control how long you will keep the idle session in the memory, you use the `extendedSessionIdleTimeoutInSeconds` setting in `host.json`, as shown here:

```
{
      "extensions": {
            "durableTask": {
                  "extendedSessionsEnabled": true,
                  "extendedSessionIdleTimeoutInSeconds": 30
            }
      }
}
```

But there are always two sides of a coin. So, when enabling extended session to increase throughput, there is a downside as well.

- It can increase function app memory usage.
- It can decrease throughput if there are many concurrent, short-lived orchestrator functions.

Performance Targets

If you are planning to use durable functions in a production application, you should consider the performance requirements early in the process because they will define the pattern you should use for your functions.

Table 5-1 shows the maximum throughput for various scenarios.

Table 5-1. *Maximum Throughput*

Scenario	Maximum Throughput
Sequential activity execution	5 activities per second, per instance
Parallel activity execution (fan-out)	100 activities per second, per instance
Parallel response processing (fan-in)	150 responses per second, per instance
External event processing	50 events per second, per instance

Creating Durable Functions Using Azure Portal

Now that you understand what a durable function is, let's create one.

Creating a Durable Function

Follow these steps:

1. Open Azure Portal and click "Create a resource." Select Compute and then Function App, as shown in Figure 5-9.

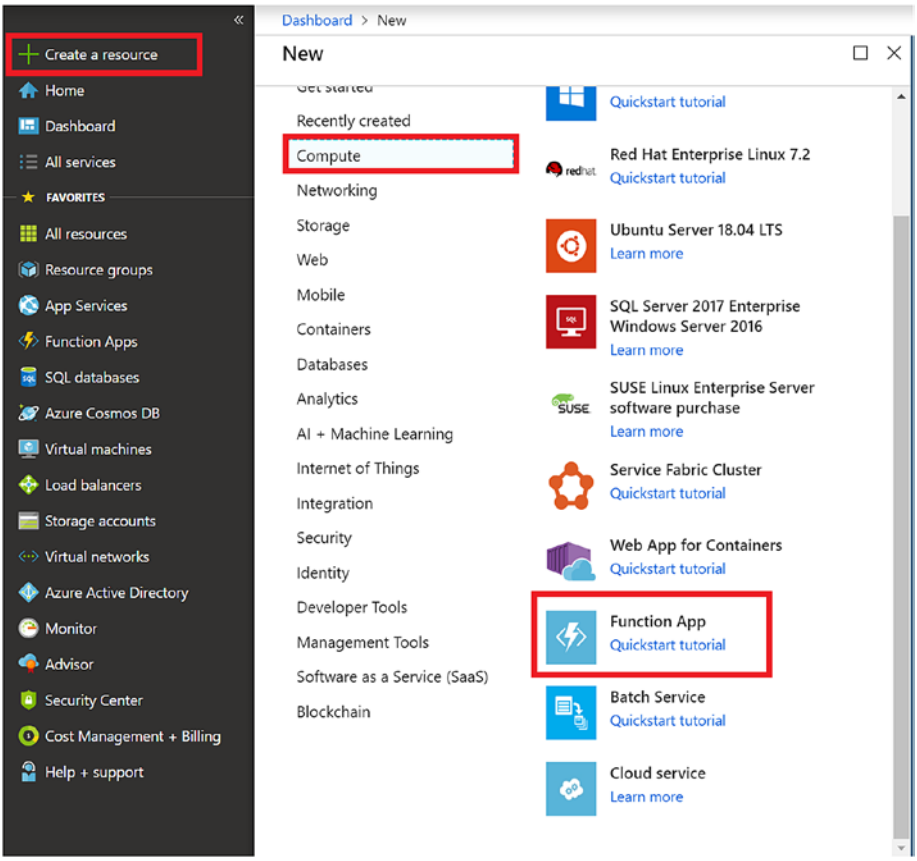

Figure 5-9. *Starting a durable function*

2. Provide the details shown in Figure 5-10 and click Create.

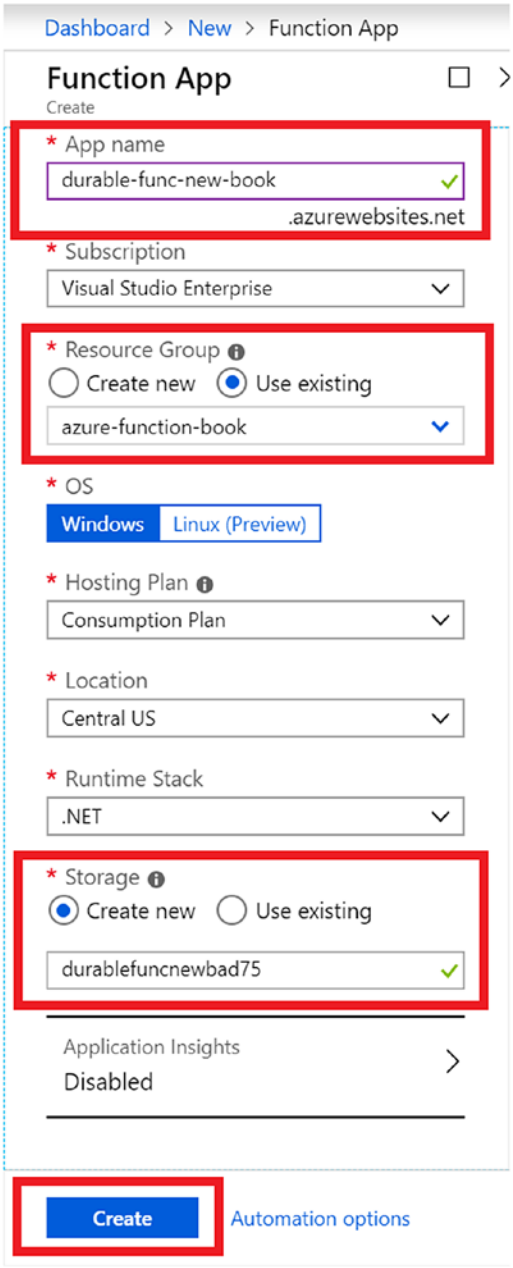

Figure 5-10. *App details*

3. Once the deployment succeeds, go to the resource and select the `durable-func-new-book` function, as shown in Figure 5-11.

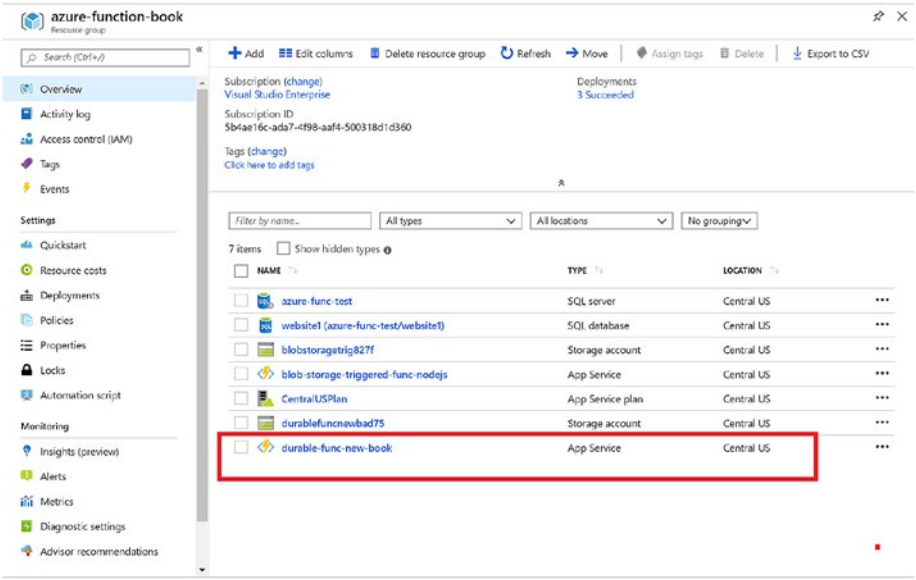

Figure 5-11. *Selecting the function*

4. Expand the function's app and click the + icon. Then, click the "In-portal" environment and continue, as shown in Figure 5-12.

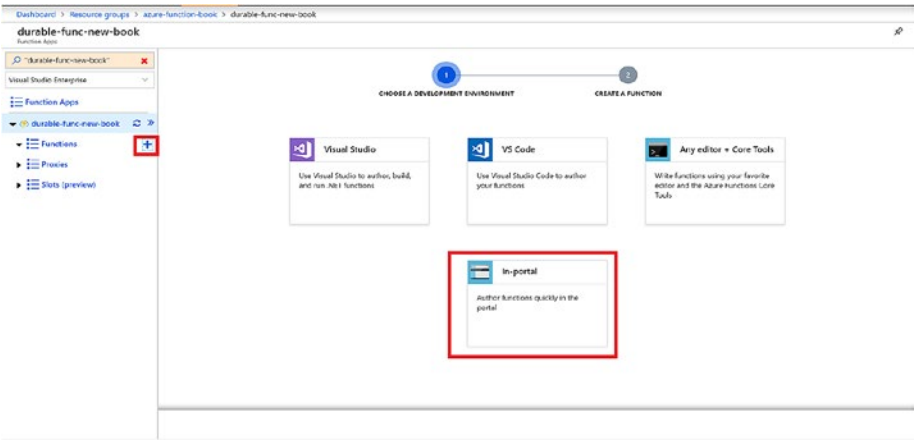

Figure 5-12. *Selecting the environment and continue, as shown in Figure 5-12.*

5. Click "More templates" and click "Finish and view templates and continue, as shown in Figure 5-12.," as shown in Figure 5-13.

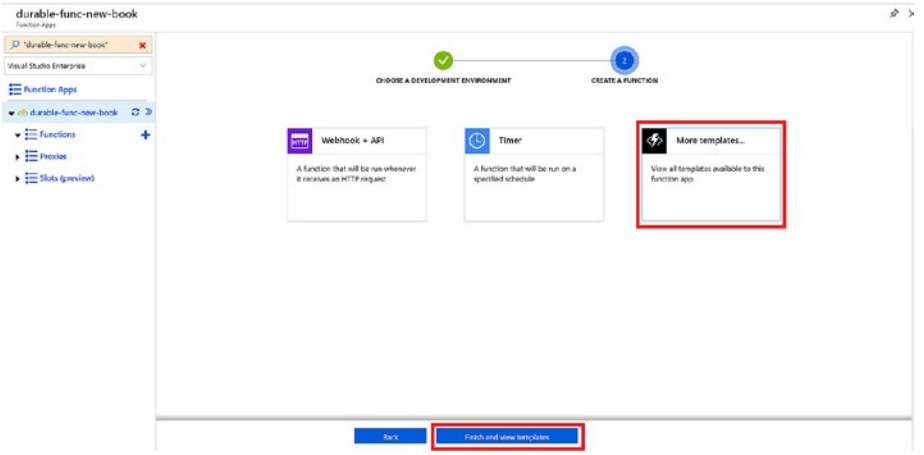

Figure 5-13. *Choosing more templates*

6. In the search field, type **durable** and select the "Durable Functions HTTP starter" template, as shown in Figure 5-14.

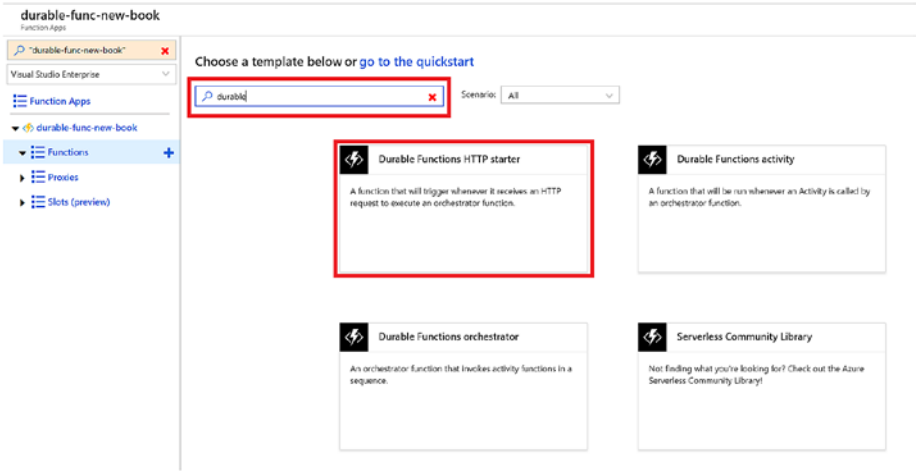

Figure 5-14. *Selecting the starter template*

7. Click Install to install the Durable Functions extension, as shown in Figure 5-15.

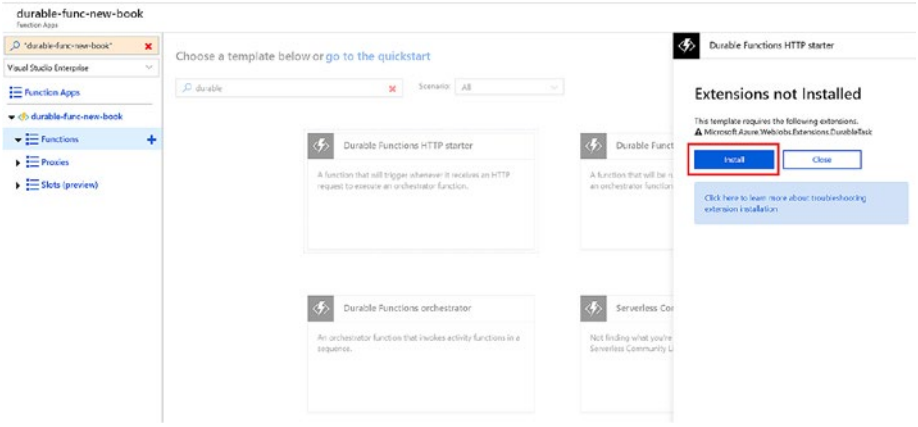

Figure 5-15. *Starting the installation*

8. Name the orchestrator client function `OrchestrationClient_Start`. Paste the following code and click Save:

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

    using System.Net.Http;
    using System.Threading.Tasks;
    using Microsoft.Azure.WebJobs;
    using Microsoft.Azure.WebJobs.Extensions.Http;
    using Microsoft.Azure.WebJobs.Host;
    using Microsoft.Extensions.Logging;

[FunctionName("OrchestrationClient_Start")]
public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous,
            "get", "post")]HttpRequestMessage req,
            [OrchestrationClient]DurableOrchestration
            Client starter, ILogger log)
        {
            string instanceId = await starter.StartNew
            Async("Orchestrator_City", null);
            log.LogInformation($"Running orchestration
            with ID = '{instanceId}'.");
            return starter.CreateCheckStatus
            Response(req, instanceId);
        }
```

9. Click the + icon again and type **durable**. Select "Durable Functions orchestrator," as shown in Figure 5-16.

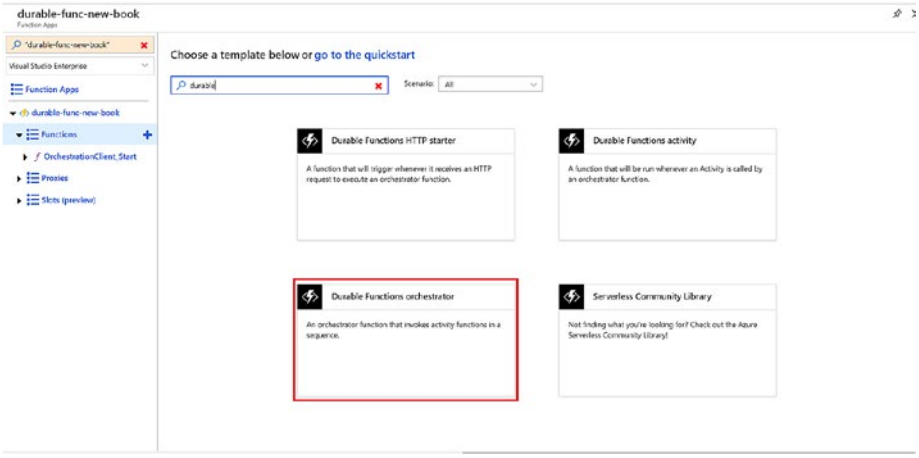

Figure 5-16. *Selecting the orchestrator*

10. Name the function `Orchestrator_City` and paste the following code:

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

        [FunctionName("Orchestrator_City")]
        public static async Task<List<string>> Run(
            [OrchestrationTrigger] DurableOrche
            strationContext context)
        {
            var outputs = new List<string>();

            outputs.Add(await context.CallActivity
            Async<string>("City_Travel", "Hyderabad"));
```

```
        outputs.Add(await context.CallActivity
        Async<string>("City_Travel", "New York"));
        outputs.Add(await context.CallActivity
        Async<string>("City_Travel", "Delhi"));

        // returns
        // "I am travelling to Hyderabad"
        // "I am travelling to New York"
        // "I am travelling to Delhi"

        return outputs;
    }
```

11. Click the + icon again and type **durable**. Select "Durable Functions activity," as shown in Figure 5-17.

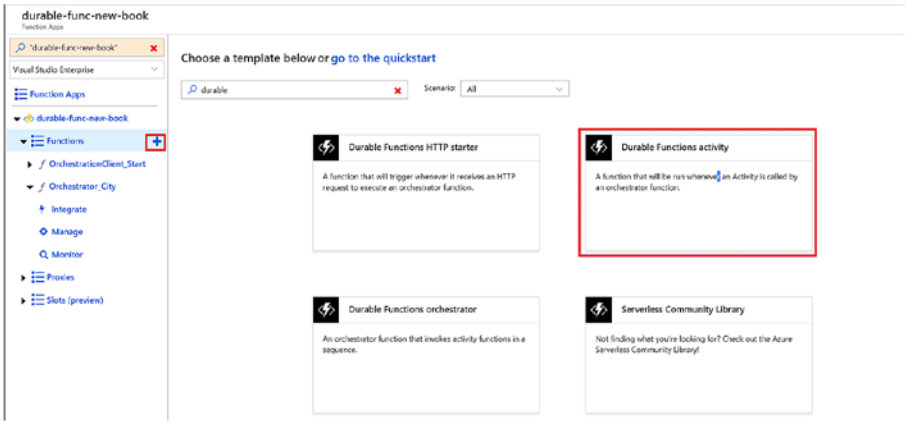

Figure 5-17. *Selecting the activity*

12. Name the function `City_Travel` and paste the following code:

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

[FunctionName("City_Travel")]
        public static string Run([ActivityTrigger]
        string cityName, ILogger log)
        {
            log.LogInformation($"I am travelling to
            {cityName}.");
            return $"I am travelling to {cityName}!";
        }
```

13. Click the function name and then click "Platform features." Once you are on the Platform features tab, select App Service Editor, as shown in Figure 5-18.

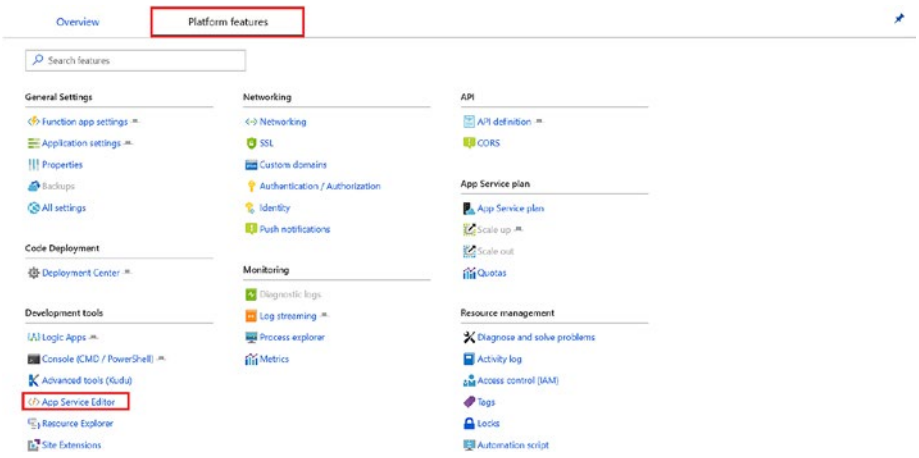

Figure 5-18. *Selecting the App Service Editor*

14. The App Service Editor will open in a new tab. Now, select the `host.json` file and copy and paste the following code in it:

```
{
    "version": "2.0",
    "logging": {
        "fileLoggingMode": "always",
        "logLevel": {
          "default": "Information",
          "Host.Results": "Information",
          "Function": "Information",
          "Host.Aggregator": "Trace"
        }
      }
}
```

15. There's no need to save the file. It autosaves, as shown in Figure 5-19.

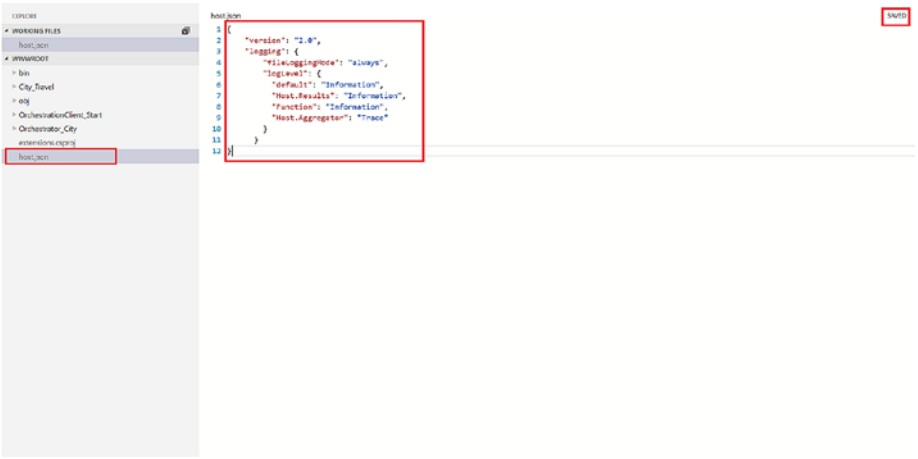

Figure 5-19. *The code*

16. Now, go back to the function app and go to the `OrchestrationClient_Start` function. Click Get Function Url and copy the URL. The URL will be in the format of `https://function-hhtp-instance/api/orchestrators/{functionName}?code=#code`. Replace `{functionName}` with the name of the orchestration HTTP trigger, which in this case is `OrchestrationClient_Start`. Now your URL is all set. Paste the URL in the browser and press Enter, and you should see the result, as shown in Figure 5-20.

Figure 5-20. *Results*

Disaster Recovery and Geodistribution of Durable Functions

Since you have deployed your Azure durable function and are wanting to use it in production, you should now look at how you can make it production-ready.

Whenever you want a solution to run on a cloud service provider such as Microsoft, Amazon, Google, and so on, you should specifically plan for disaster recovery and make sure your application is running in case there is any disaster and the region in which application is running in goes down.

Also, you should take care of the data that is going to be stored to make this application run successfully is properly georeplicated.

To enable disaster recovery for durable functions, you should first make sure that your durable function is stateless. Once you have done that, you can enable disaster recovery by leveraging another Microsoft service called Traffic Manager.

You can configure Azure Functions as an app service in Traffic Manager and use any routing strategy.

For geodistribution, you should always consider keeping a copy of the data in multiple regions. All the Azure data storage services such as Azure Storage, Azure Cosmos DB, and Azure SQL provide georeplication of data across Azure data centers. You can enable these georeplication services to make sure that your data is available in multiple regions. This will help you make your function available quickly in the case of a disaster.

In conclusion, this chapter covered how durable functions work and the patterns you'll use with the Durable Functions extension. Also, in this chapter, you created your first durable function running in Azure. You now understand how to manage your functions in the case of a disaster.

In the next chapter, you will look at deploying functions to Azure using a CI/CD pipeline and at how to configure the functions for Azure Functions.

CHAPTER 6

Deploying Functions to Azure

In this chapter, I will cover following topics:

- Deploying functions to Azure using continuous deployment
- Deploying functions to Azure using ARM templates

This chapter will walk you through the ways to deploy functions to Azure. By the end of this chapter, you should be able to deploy your functions in two different ways.

Deploying Functions Using Continuous Deployment

Azure Functions integrates seamlessly with continuous integration/continuous deployment (CI/CD) and the Azure pipeline, which allows you to continuously deploy your functions to production. Continuous deployment makes it easier to deploy code bits in a project where multiple people are working and when changes in the code repository are frequent.

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_6

Using App Service continuous integration, you can easily deploy a function app. Azure Functions integrates seamlessly with the following deployment sources:

- Azure DevOps (a.k.a. VSTS)
- OneDrive
- GitHub
- Dropbox
- Bitbucket
- Git local repository
- External repositories such as Mercurial and Git

Continuous deployment is configured on a per-function app basis, and once the continuous deployment is enabled, the access to the function app is set to read-only in Azure Portal.

Setting Up a Code Repository for Continuous Deployment

Before you set up continuous deployment for your function app, you should arrange your source code properly. The name of the directory is the name of the function app. The `host.json` file resides in the parent or top folder. Each subfolder in the function app consists of separate functions. A `bin` folder contains library files and packages required by the function to run, as shown in Figure 6-1.

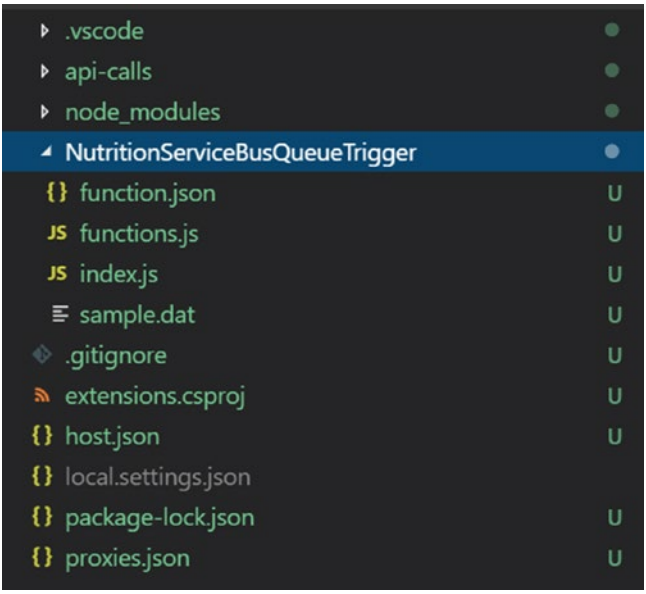

Figure 6-1. *Project organization*

All the functions in the function app should have the same language worker.

Now that your code repository is ready, let's set up your function for continuous deployment.

Setting Up an Azure DevOps Account

Before you can set up continuous deployment for your Azure function, you need to set up your Azure DevOps account so that you can connect it to the Azure Functions service.

Set up your Azure DevOps account by following these steps:

1. Go to Azure Portal (`https://ms.portal.azure.com/`) and click Services. Then search for *azure devops* and select "Azure DevOps organizations," as shown in Figure 6-2.

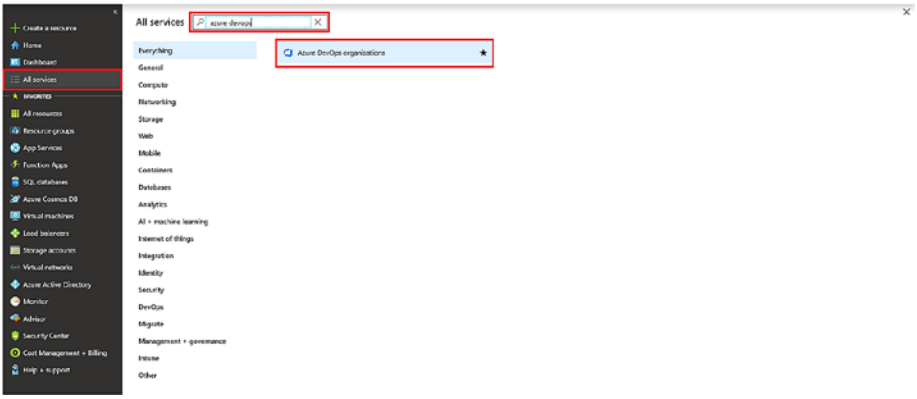

Figure 6-2. *Finding Azure DevOps organizations*

2. You should see the list of organizations, as shown in Figure 6-3.

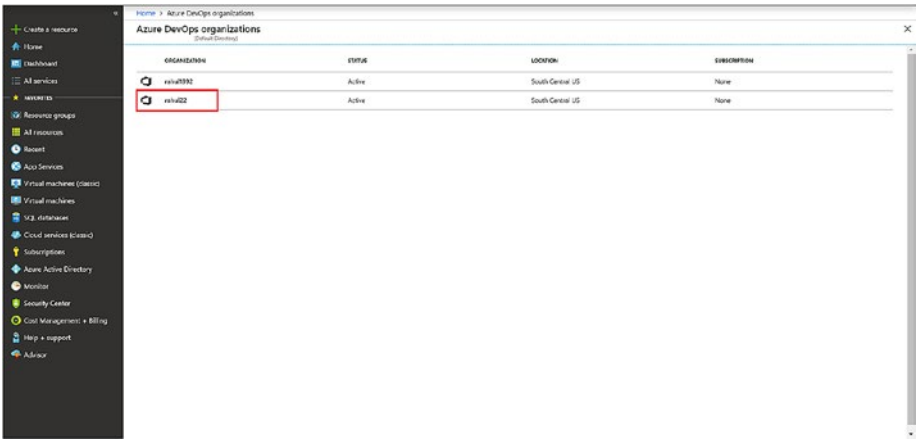

Figure 6-3. *Organizations list*

3. Once you click the organization, a blade should open on the right side, as shown in Figure 6-4. Select "Set up billing" in the menu.

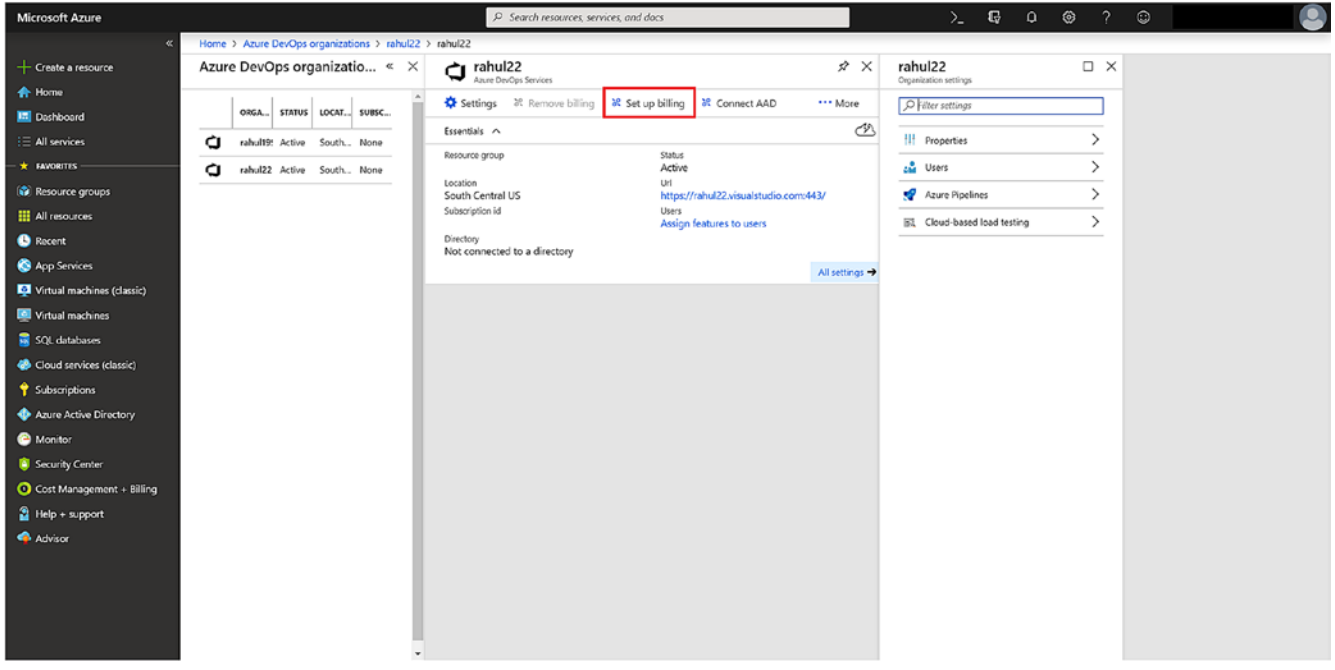

Figure 6-4. *Setting up billing*

4. A vertical blade will open with the available Azure subscriptions. Select a suitable one and click Link, as shown in Figure 6-5.

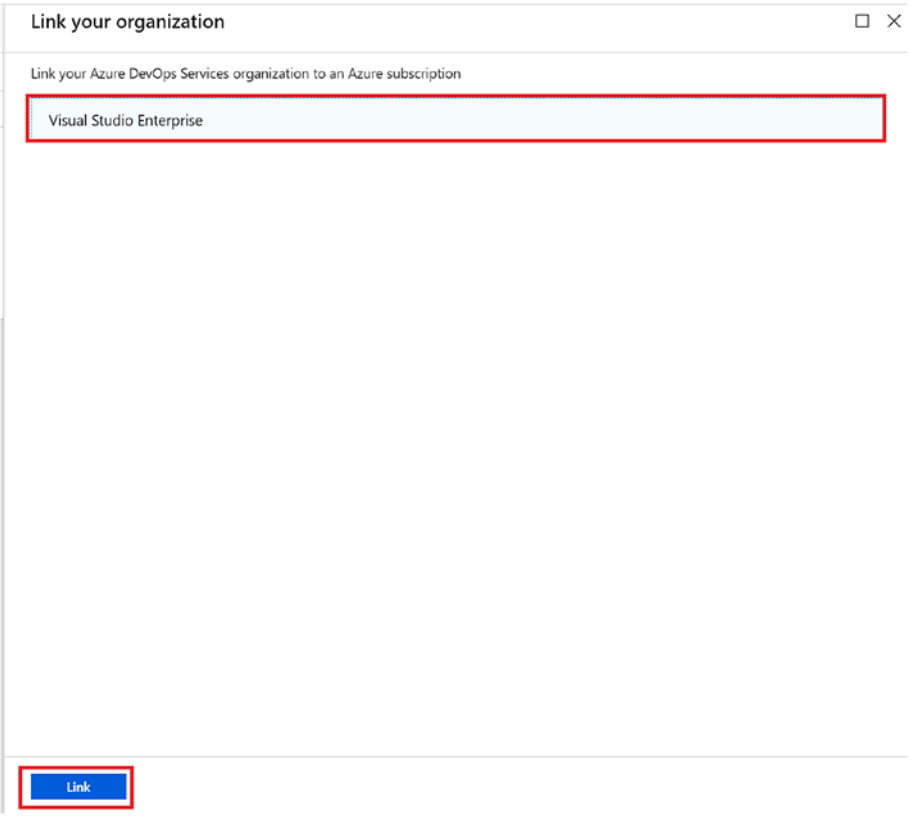

Figure 6-5. *Linking to an Azure subscription*

5. Once you get a notification that the subscription is linked, you are good to go, as shown in Figure 6-6.

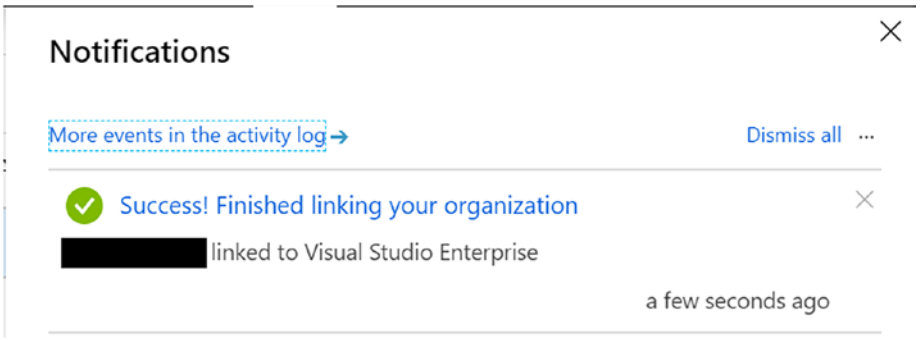

Figure 6-6. *Success*

Setting Up Continuous Deployment for Azure Functions

You have linked your Azure DevOps organization with an Azure subscription, so you can now set up continuous development for Azure Functions.

1. Go to Azure Portal and select all the resources and functions you want to set up for continuous deployment, as shown in Figure 6-7.

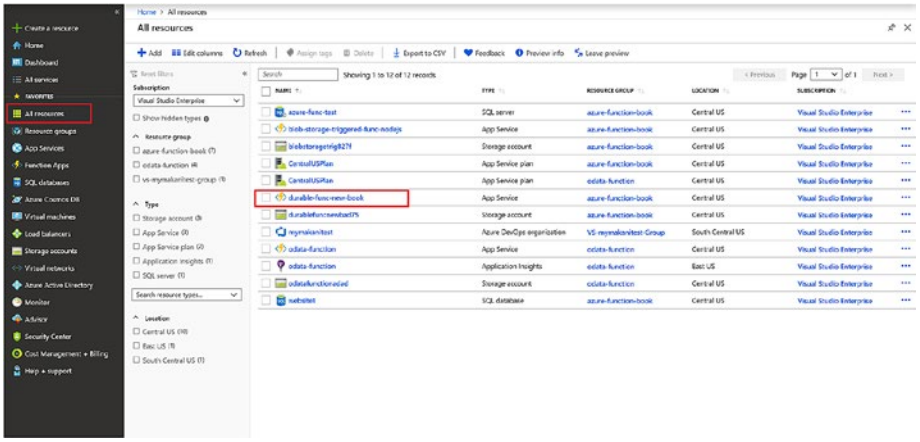

Figure 6-7. *Choosing the functions*

2. Once a function loads, click "Platform features" in the top menu, as shown in Figure 6-8.

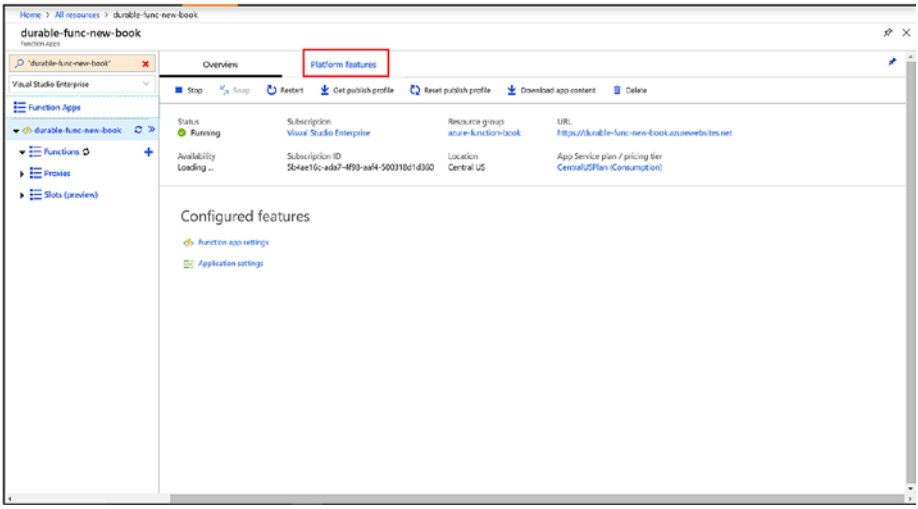

Figure 6-8. *Selecting platform features*

3. In "Platform features," you will see the Code Deployment section. Select Deployment Center, as shown in Figure 6-9.

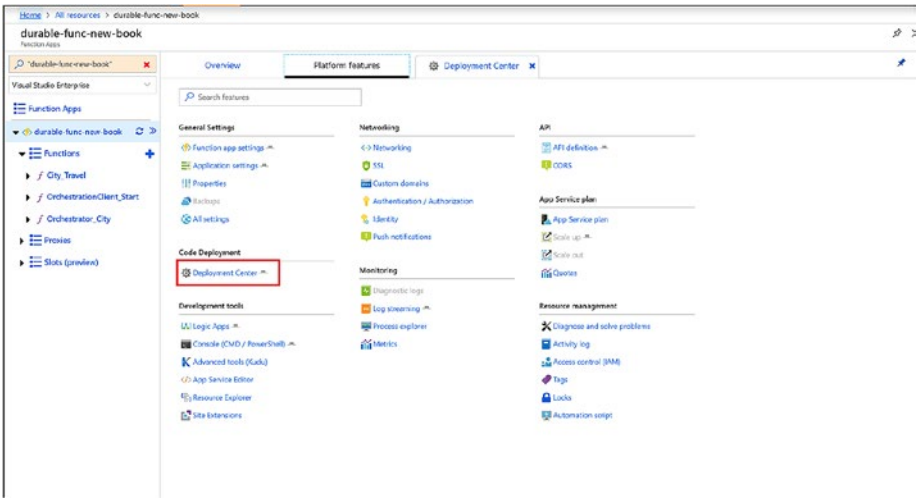

Figure 6-9. *Choosing Deployment Center*

4. Now you will see lot of options such as Azure Repos, GitHub, Bitbucket, and so on. For this, select Azure Repos and click Next, as shown in Figure 6-10.

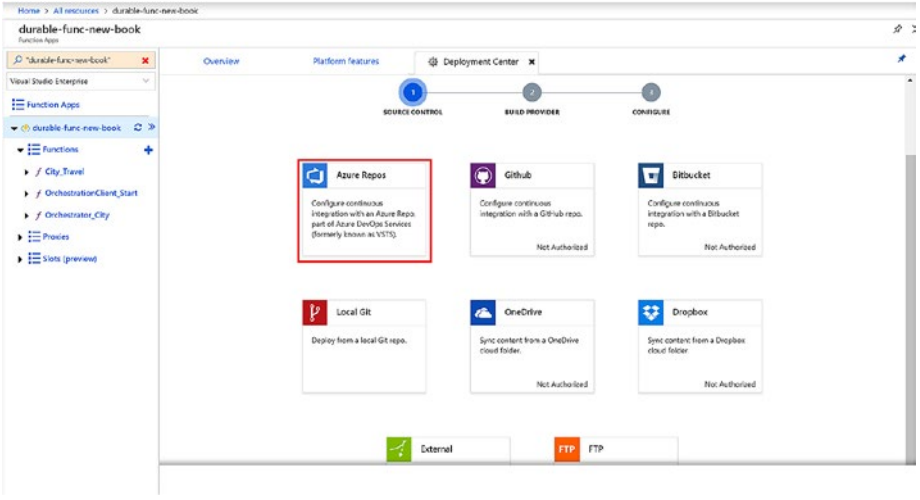

Figure 6-10. *Choosing Azure Repos*

5. Select Azure Pipelines because it is the preferred way to configure continuous deployment. It is still in preview but is good enough to use. Once you have selected Azure Pipelines, as shown in Figure 6-11, click Continue.

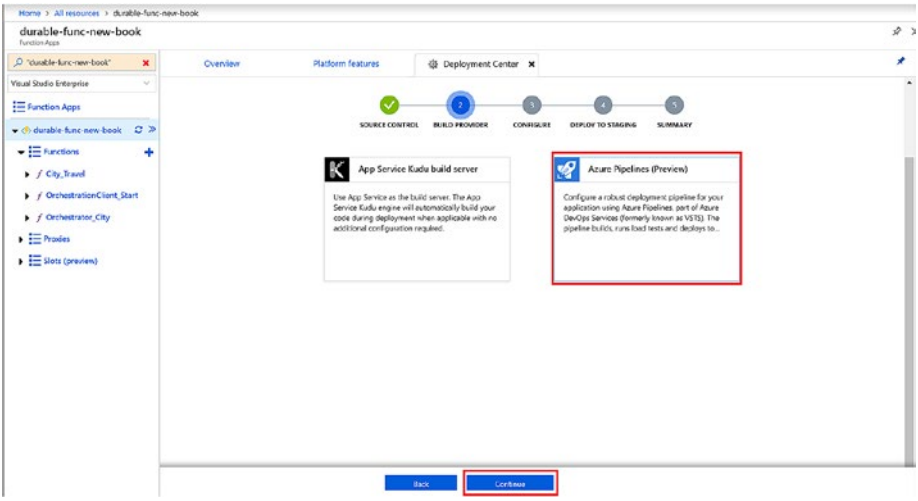

Figure 6-11. *Choosing Azure Pipelines*

6. Select the Azure DevOps configuration that you configured and provide details such as the project name, repository, and branch. Then click Continue, as shown in Figure 6-12.

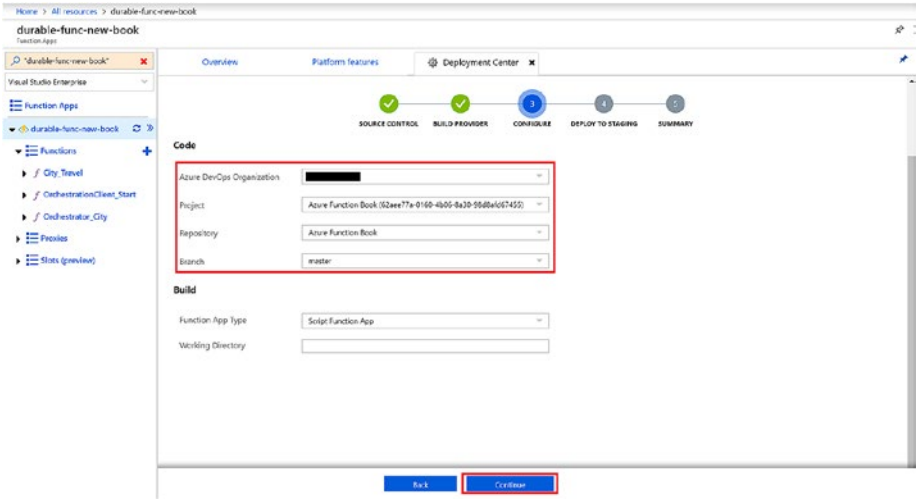

Figure 6-12. *Adding the Azure DevOps configuration*

7. Set up a deployment slot, which is just a staging area. A staging slot is where you can deploy your app and test it, and if everything looks good, you can just swap the staging slot with the production slot without any downtime. This will help you in two ways.
   - The app can be thoroughly tested before being released to production.
   - If the new deployment after the slot swap misbehaves in production or has a bug, you have your last working code already available in the staging slot, so you can swap it back to production.
8. Provide the details, as shown in Figure 6-13, and click Continue.

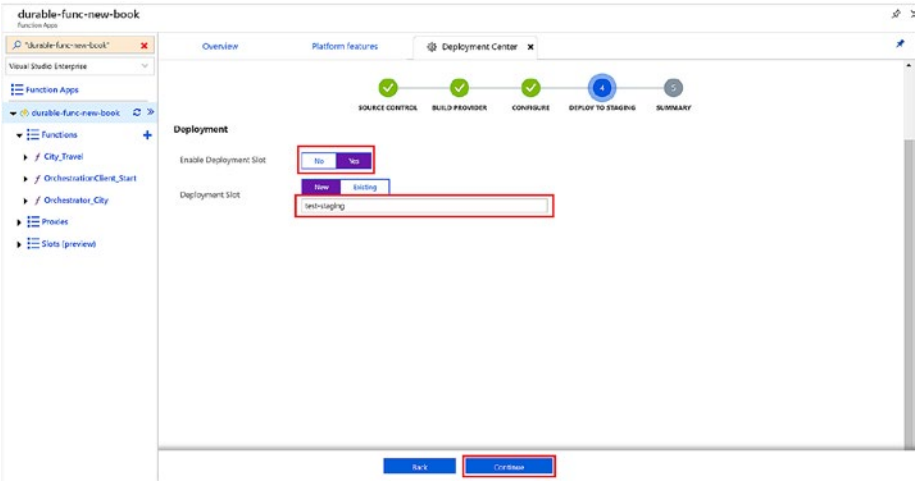

Figure 6-13. *Setting up the deployment slot*

9. Check the summary and click Finish, and your continuous deployment is now ready and set up, as shown in Figure 6-14.

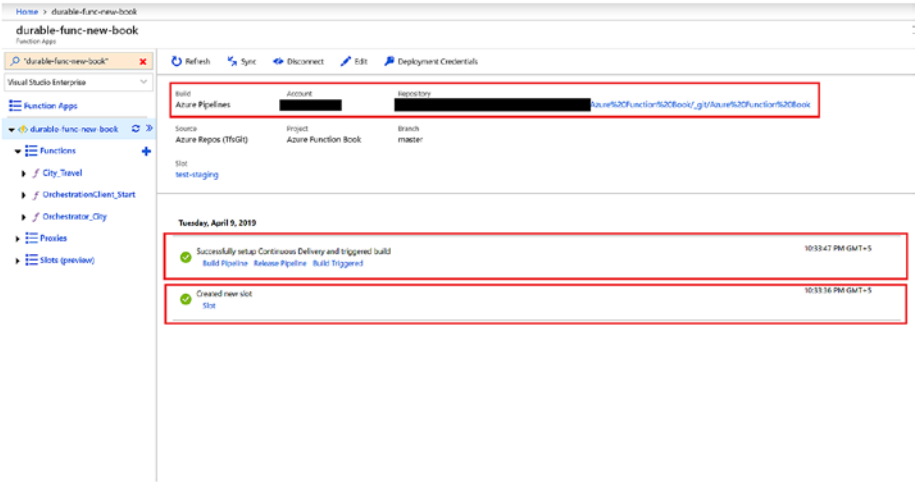

Figure 6-14. *CD now ready*

10. Click Build Pipeline, as shown in Figure 6-14. Click the link, and you will be taken to the build pipeline of your function, as shown in Figure 6-15.

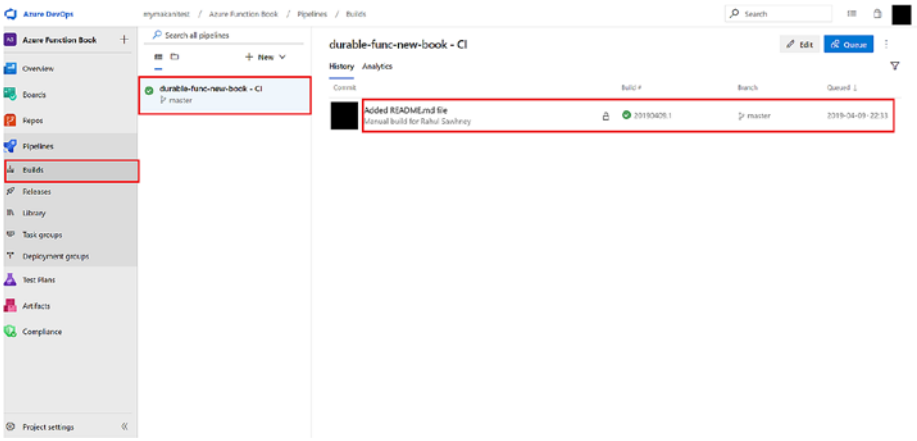

Figure 6-15. *Build pipeline*

11. To check the release pipeline, click the Release Pipeline link, as shown in Figure 6-14. You will be taken to the release pipeline of the function in Visual Studio Team Service (VSTS), as shown in Figure 6-16.

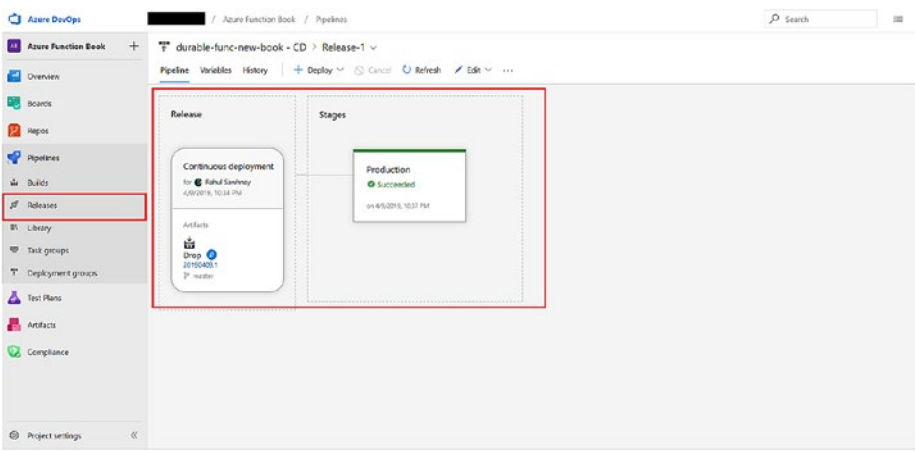

Figure 6-16. *Release pipeline*

Deploying Azure Functions Using ARM Templates

One of the most popular ways of deploying anything on Azure has been Azure Resource Manager (ARM) templates. Functions can also be deployed using ARM templates. In this section, you will look at the required parameters and resources that will enable you to deploy functions with ARM templates.

Basically, you need the following resources to start deploying functions using ARM templates:

- Azure Storage account
- Hosting plan
- Function app

Let's set up these using ARM templates.

For any deployment on Azure, you require an Azure Storage account, which is the case here. ARM templates will basically first copy the zip file to an Azure Storage blob and then use that zip file to deploy the required resource.

The following code snippet will create an Azure Storage account using an ARM template:

```
{
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[variables('storageAccountName')]",
      "apiVersion": "2016-12-01",
      "location": "[parameters('location')]",
      "kind": "Storage",
      "sku": {
        "name": "[parameters('storageAccountType')]"
      }
}
```

This code is looking for the `storageAccountType` parameter, which can be set up in the parameters section in the ARM template, as shown here:

```
"storageAccountType": {
      "type": "string",
      "defaultValue": "Standard_LRS",
      "allowedValues": ["Standard_LRS", "Standard_GRS",
      "Standard_RAGRS"],
      "metadata": {
        "description": "Storage Account type"
      }
}
```

The Azure Storage account is set up, so let's look at setting up the hosting plan. Here you have two types of hosting plans: the Consumption Plan and the App Service Plan. Let's first look at the Consumption Plan.

Deploying a Function App on the Consumption Plan

The Consumption Plan allows you to make the best use of Azure Functions. The Consumption Plan dynamically allocates compute power when your code is running. It scales out to handle extra load and then returns to normal when the load lessens. So, if Azure Functions is not running, you are not paying anything for idle VMs. Also, you don't have to worry about peak load in advance because the Consumption Plan will take care of it.

The Consumption Plan is a special type of `serverfarm` resource, and in ARM templates you specify it by setting the `Dynamic` value for the `computeMode` and `sku` properties.

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

In addition, two more settings, `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` and `WEBSITE_CONTENTSHARE`, are required by the Consumption Plan. These properties configure the storage account and file path where the function app and configuration are stored.

```
"properties": {
        "serverFarmId": "[resourceId('Microsoft.Web/
        serverfarms', variables('hostingPlanName'))]",
        "siteConfig": {
          "appSettings": [
            {
              "name": "WEBSITE_CONTENTAZUREFILE
              CONNECTIONSTRING",
              "value": "[concat('DefaultEndpointsProtocol=
              https;AccountName=', variables('storageAccount
              Name'), ';AccountKey=', listKeys(variables('storage
              Accountid'),'2015-05-01-preview').key1)]"
            },
            {
              "name": "WEBSITE_CONTENTSHARE",
              "value": "[toLower(variables('functionAppName'))]"
            }
          ]
        }
}
```

The complete ARM template to deploy Azure Functions on the Consumption Plan is as follows:

```
{
  "$schema": "https://schema.management.azure.com/schemas/
  2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "appName": {
      "type": "string",
```

```
      "metadata": {
        "description": "Function App Name"
      }
    },
    "storageAccountType": {
      "type": "string",
      "defaultValue": "Standard_LRS",
      "allowedValues": ["Standard_LRS", "Standard_GRS",
      "Standard_RAGRS"],
      "metadata": {
        "description": "Storage Account type"
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for all resources."
      }
    },
    "runtime": {
      "type": "string",
      "defaultValue": "node",
      "allowedValues": ["node", "dotnet", "java"],
      "metadata": {
        "description": "The language worker runtime to load in
        the function app."
      }
    }
  },
```

```
"variables": {
  "functionAppName": "[parameters('appName')]",
  "hostingPlanName": "[parameters('appName')]",
  "applicationInsightsName": "[parameters('appName')]",
  "storageAccountName": "[concat(uniquestring(resource
  Group().id), 'azfunctions')]",
  "storageAccountid": "[concat(resourceGroup().id,'/providers/',
  'Microsoft.Storage/storageAccounts/', variables('storage
  AccountName'))]",
  "functionWorkerRuntime": "[parameters('runtime')]"
},
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "name": "[variables('storageAccountName')]",
    "apiVersion": "2016-12-01",
    "location": "[parameters('location')]",
    "kind": "Storage",
    "sku": {
      "name": "[parameters('storageAccountType')]"
    }
  },
  {
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2015-04-01",
    "name": "[variables('hostingPlanName')]",
    "location": "[parameters('location')]",
    "properties": {
      "name": "[variables('hostingPlanName')]",
      "computeMode": "Dynamic",
      "sku": "Dynamic"
    }
  },
```

```
{
  "apiVersion": "2015-08-01",
  "type": "Microsoft.Web/sites",
  "name": "[variables('functionAppName')]",
  "location": "[parameters('location')]",
  "kind": "functionapp",
  "dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables
    ('hostingPlanName'))]",
    "[resourceId('Microsoft.Storage/storageAccounts',
    variables('storageAccountName'))]"
  ],
  "properties": {
    "serverFarmId": "[resourceId('Microsoft.Web/
    serverfarms', variables('hostingPlanName'))]",
    "siteConfig": {
      "appSettings": [
        {
          "name": "AzureWebJobsDashboard",
          "value": "[concat('DefaultEndpointsProtocol=
          https;AccountName=', variables('storageAccount
          Name'), ';AccountKey=', listKeys(variables
          ('storageAccountid'),'2015-05-01-preview').key1)]"
        },
        {
          "name": "AzureWebJobsStorage",
          "value": "[concat('DefaultEndpointsProtocol=
          https;AccountName=', variables('storageAccount
          Name'), ';AccountKey=', listKeys(variables
          ('storageAccountid'),'2015-05-01-preview').key1)]"
        },
```

```
        {
          "name": "WEBSITE_CONTENTAZUREFILE
          CONNECTIONSTRING",
          "value": "[concat('DefaultEndpointsProtocol=
          https;AccountName=', variables('storageAccount
          Name'), ';AccountKey=', listKeys(variables
          ('storageAccountid'),'2015-05-01-preview').key1)]"
        },
        {
          "name": "WEBSITE_CONTENTSHARE",
          "value": "[toLower(variables('functionAppName'))]"
        },
        {
          "name": "FUNCTIONS_EXTENSION_VERSION",
          "value": "~2"
        },
        {
          "name": "WEBSITE_NODE_DEFAULT_VERSION",
          "value": "8.11.1"
        },
        {
          "name": "APPINSIGHTS_INSTRUMENTATIONKEY",
          "value": "[reference(resourceId('microsoft.
          insights/components/', variables('application
          InsightsName')), '2015-05-01').InstrumentationKey]"
        },
        {
          "name": "FUNCTIONS_WORKER_RUNTIME",
          "value": "[variables('functionWorkerRuntime')]"
        }
      ]
    }
```

```
      }
    },
    {
      "apiVersion": "2018-05-01-preview",
      "name": "[variables('applicationInsightsName')]",
      "type": "microsoft.insights/components",
      "location": "East US",
      "tags": {
        "[concat('hidden-link:', resourceGroup().id, '/
        providers/Microsoft.Web/sites/', variables('application
        InsightsName'))]": "Resource"
      },
      "properties": {
        "ApplicationId": "[variables('applicationInsightsName')]",
        "Request_Source": "IbizaWebAppExtensionCreate"
      }
    }
  ]
}
```

Deploying a Function App on the App Service Plan

With this plan, Azure Function runs on dedicated VMs similar to web apps.

You can set up the App Service Plan in an ARM template as follows:

```
{
      "type": "Microsoft.Web/serverfarms",
      "apiVersion": "2016-09-01",
      "name": "[variables('hostingPlanName')]",
      "location": "[parameters('location')]",
      "properties": {
```

```
        "name": "[variables('hostingPlanName')]",
        "sku": "[parameters('sku')]",
        "workerSize": "[parameters('workerSize')]",
        "hostingEnvironment": "",
        "numberOfWorkers": 1
      }
}
```

Here, `workerSize` is the size of the VM, which is small (0), medium (1), or large (2). You can set up the worker size in the ARM template in the parameters section, as shown here:

```
"workerSize": {
      "type": "string",
      "allowedValues": [
        "0",
        "1",
        "2"
      ],
      "defaultValue": "0",
      "metadata": {
        "description": "The instance size of the hosting plan"
      }
}
```

The complete ARM template for Azure Functions is shown here:

```
{
  "$schema": "https://schema.management.azure.com/
  schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "appName": {
      "type": "string",
```

```
      "metadata": {
        "description": "Function App name"
      }
    },
    "sku": {
      "type": "string",
      "allowedValues": [
        "Free",
        "Shared",
        "Basic",
        "Standard"
      ],
      "defaultValue": "Standard",
      "metadata": {
        "description": "The pricing tier for the hosting plan."
      }
    },
    "workerSize": {
      "type": "string",
      "allowedValues": [
        "0",
        "1",
        "2"
      ],
      "defaultValue": "0",
      "metadata": {
        "description": "The instance size of the hosting plan"
      }
    },
    "storageAccountType": {
      "type": "string",
      "defaultValue": "Standard_LRS",
```

```
      "allowedValues": [
        "Standard_LRS",
        "Standard_GRS",
        "Standard_RAGRS"
      ],
      "metadata": {
        "description": "Storage Account type"
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for all resources."
      }
    }
  },
  "variables": {
    "functionAppName": "[parameters('appName')]",
    "hostingPlanName": "[parameters('appName')]",
    "storageAccountName": "[concat(uniquestring(resource
    Group().id), 'functions')]"
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[variables('storageAccountName')]",
      "apiVersion": "2018-02-01",
      "location": "[parameters('location')]",
      "kind": "Storage",
```

```
      "sku": {
        "name": "[parameters('storageAccountType')]"
      }
    },
    {
      "type": "Microsoft.Web/serverfarms",
      "apiVersion": "2016-09-01",
      "name": "[variables('hostingPlanName')]",
      "location": "[parameters('location')]",
      "properties": {
        "name": "[variables('hostingPlanName')]",
        "sku": "[parameters('sku')]",
        "workerSize": "[parameters('workerSize')]",
        "hostingEnvironment": "",
        "numberOfWorkers": 1
      }
    },
    {
      "apiVersion": "2016-08-01",
      "type": "Microsoft.Web/sites",
      "name": "[variables('functionAppName')]",
      "location": "[parameters('location')]",
      "kind": "functionapp",
      "properties": {
        "name": "[variables('functionAppName')]",
        "serverFarmId": "[resourceId('Microsoft.Web/
        serverfarms', variables('hostingPlanName'))]",
        "hostingEnvironment": "",
        "clientAffinityEnabled": false,
        "siteConfig": {
          "alwaysOn": true
        }
      },
```

```
"dependsOn": [
  "[resourceId('Microsoft.Web/serverfarms', variables
  ('hostingPlanName'))]",
  "[resourceId('Microsoft.Storage/storageAccounts',
  variables('storageAccountName'))]"
],
"resources": [
  {
    "apiVersion": "2016-08-01",
    "name": "appsettings",
    "type": "config",
    "dependsOn": [
      "[resourceId('Microsoft.Web/sites', variables
      ('functionAppName'))]",
      "[resourceId('Microsoft.Storage/storageAccounts',
      variables('storageAccountName'))]"
    ],
    "properties": {
      "AzureWebJobsStorage": "[concat('DefaultEndpoints
      Protocol=https;AccountName=',variables('storage
      AccountName'),';AccountKey=',listkeys(resourceId
      ('Microsoft.Storage/storageAccounts', variables
      ('storageAccountName')), '2015-05-01-preview').
      key1,';')]",
      "AzureWebJobsDashboard": "[concat('DefaultEndpoints
      Protocol=https;AccountName=',variables('storage
      AccountName'),';AccountKey=',listkeys(resourceId
      ('Microsoft.Storage/storageAccounts', variables
      ('storageAccountName')), '2015-05-01-preview').
      key1,';')]",
```

```
            "FUNCTIONS_EXTENSION_VERSION": "~1"
          }
        }
      ]
    }
  ]
}
```

Once the function is deployed using the CI/CD pipeline and you have set up the staging slot, the function will be deployed to the staging slot. To go to the staging slot, follow these steps:

1. Go to Azure Portal and click Function Apps in the menu, as shown in Figure 6-17.

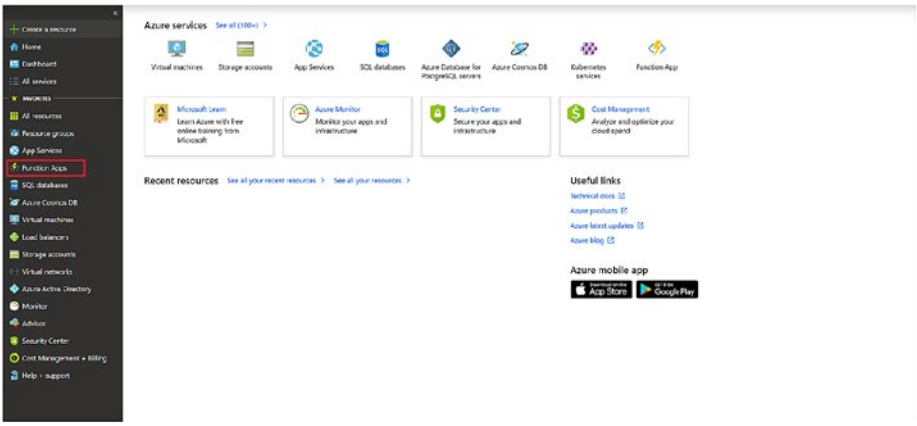

Figure 6-17. *Selecting Function Apps*

2. Select the function for which you created the CI/CD pipeline. I have created a pipeline for `durable-func-new-book`, as shown in Figure 6-18.

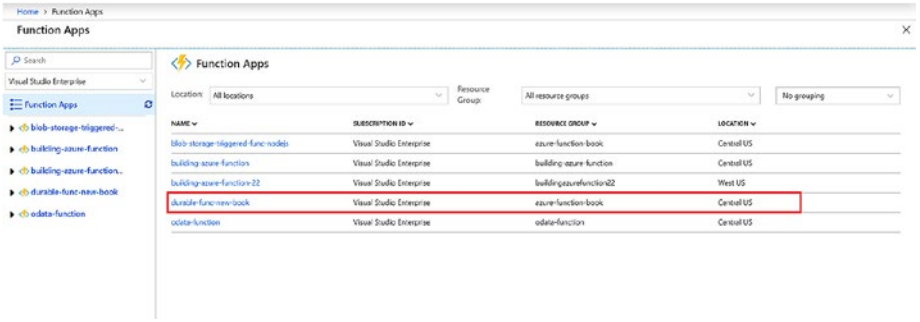

Figure 6-18. *Pipeline*

3. Click "Platform features" and select Deployment Center, as shown in Figure 6-19.

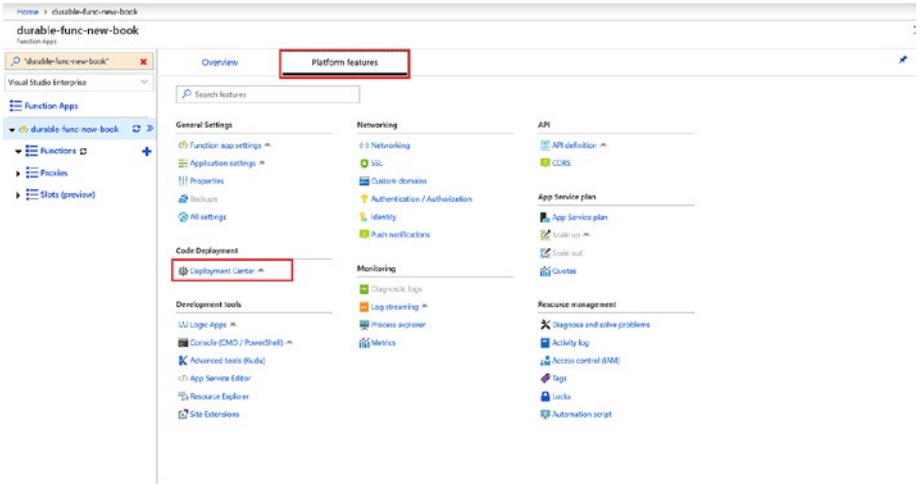

Figure 6-19. *Selecting Deployment Center*

4. Once you are in the Deployment Center, click the slot, as shown in Figure 6-20.

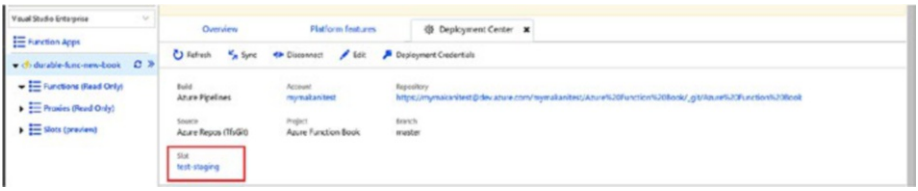

Figure 6-20. *Selecting the slot*

5. Once you are in the slot, you will see the URL of the staging slot and the Swap option, as shown in Figure 6-21. Now, you can test your function in the staging slot. Once you are satisfied that things are running fine, you can swap the slot.

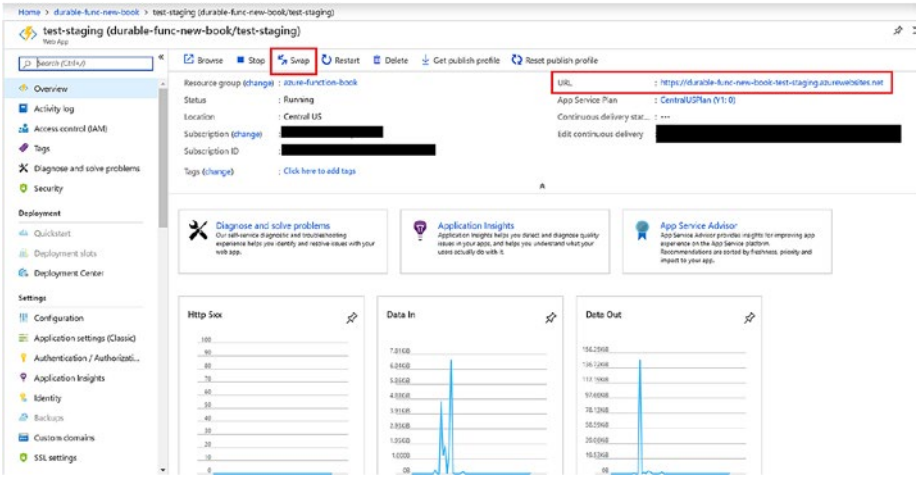

Figure 6-21. *Testing*

6. To swap the slot, click the Swap button, as highlighted in Figure 6-21. When you click Swap, the vertical screen will open with option to swap. Once you are satisfied with the values, click Swap again, as shown in Figure 6-22.

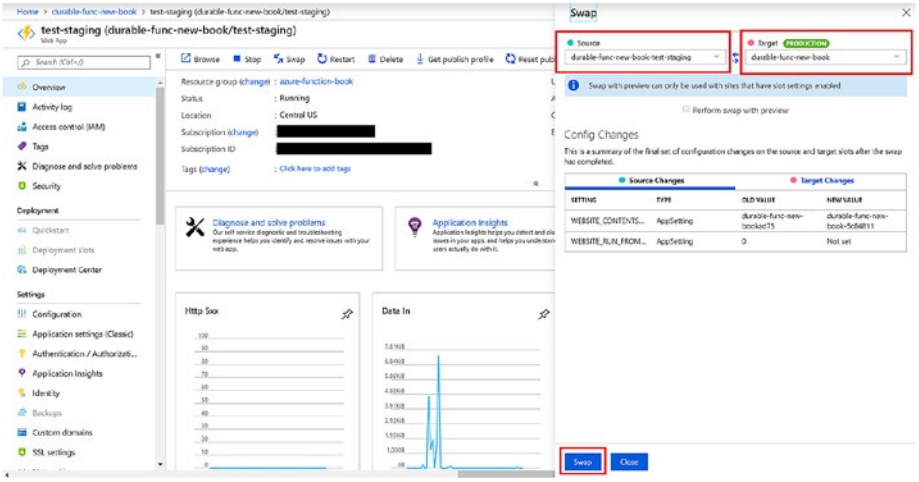

Figure 6-22. *Clicking Swap*

You can get the Azure Quick Start template at `https://azure.microsoft.com/en-us/resources/templates/` or from GitHub at `https://github.com/Azure/azure-quickstart-templates/`.

You have now configured the CI/CD pipeline of your function, so your function app is all set for production. In the next chapter, you will look at what's required to make functions production-ready.

CHAPTER 7

Getting Functions Production-Ready

In this chapter, I will cover following topics:

- Using built-in logging
- Using Application Insights to monitor functions
- Securing functions
- Configuring CORS in Azure Functions

Using Built-in Logging

The first thing that comes to mind when talking about monitoring functions is error logging. You'll want to log errors in Azure Functions so that you know what went wrong and can fix it.

By default, Azure Functions comes with a logger instance that logs errors to Azure File Storage. The logger is passed to the function along with the invocation, as shown in Figure 7-1.

```
[FunctionName("HttpTriggerCSharp")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRequest req,
    ILogger log)
```

Figure 7-1. *The logger is passed to the function*

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9_7

As you can see in Figure 7-1, an instance of `ILogger` is passed as an argument to the function invocation. You can use the extension method from `Microsoft.Extensions.Logging` to log events. The events that are exposed are `LogDebug`, `LogInformation`, `LogError`, `LogWarning`, and `LogCritical`.

For a JavaScript function, it looks like Figure 7-2.

Figure 7-2. *The JavaScript function*

The context passed in Figure 7-2 has a `log` function, and you can use it to log at different levels. The `log` function has similar levels, such as Trace, Debug, Information, Warning, Error, and Critical.

The host and function logs of Azure Functions is kept in `/LogFiles/Application/Functions`.

Using Application Insights to Monitor Azure Functions

Azure Functions offers built-in integration with Application Insights. Using Application Insights, you can monitor Azure Functions easily because Application Insights not only provides error details but also provides details such as server requests, timer functions, and much more.

Application Insights Settings for Azure Functions

To connect Azure Functions to Application Insights, Azure Functions needs to know the Application Insights instrumentation key. The key `APPINSIGHTS_INSTRUMENTATIONKEY` must be set in the app settings of Azure Functions.

You can integrate Application Insights with Azure Functions in two ways.

- Automatically integrating during new function creation
- Manually connecting to the existing Application Insights service

Integrate Application Insights During New Azure Function Creation

Let's see how this is done.

1. Go to Azure Portal and click "Create a resource." Then, click Compute and click Function App, as shown in Figure 7-3.

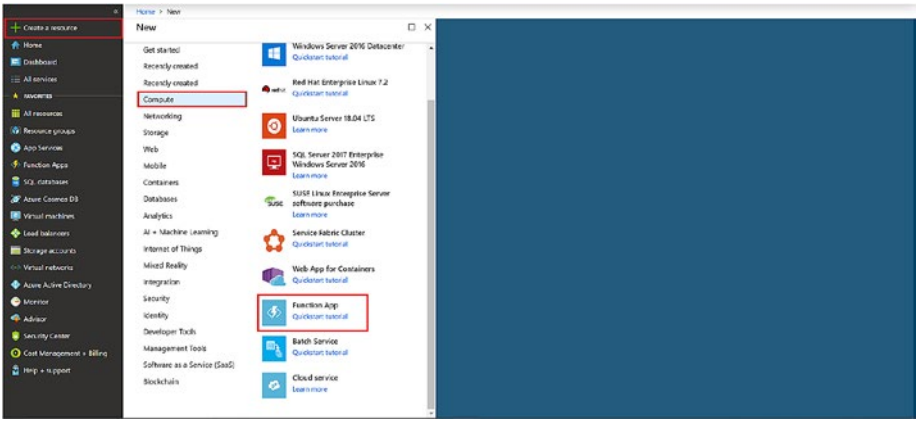

Figure 7-3. *Starting the function app*

2. A blade will open. Scroll down and click Application Insights, as shown in Figure 7-4.

Figure 7-4. *Finding Application Insights*

3. Once you click it, the Application Insights setup will open. Click Enable, select "Create new resource" or "Select existing resource," and set up Application Insights, as shown in Figure 7-5.

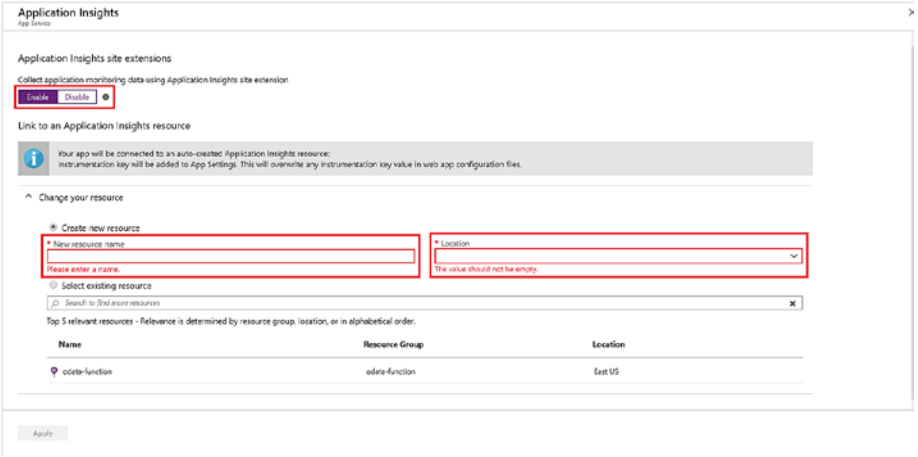

Figure 7-5. *Setting up Application Insights*

4. Provide the proper details such as the resource name and location and click Apply. Your new function is now integrated with Application Insights.

Manually Connecting Application Insights to Azure Functions

Let's do it manually now. Follow these steps:

1. Go to Azure Portal, click "Create a resource," and search for *Application Insights*. Then click Create, as shown in Figure 7-6.

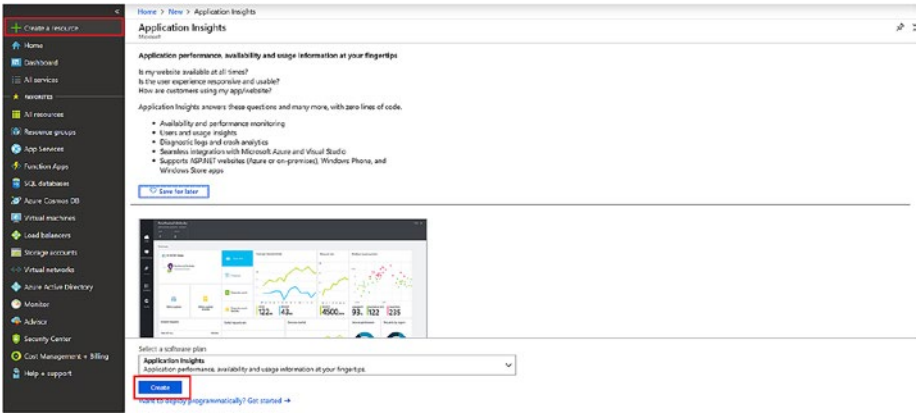

Figure 7-6. *Starting the manual process*

2. Once you are on the Application Insights creation page, provide details such as the name, application type, resource group, and location, as shown in Figure 7-7. Click Create.

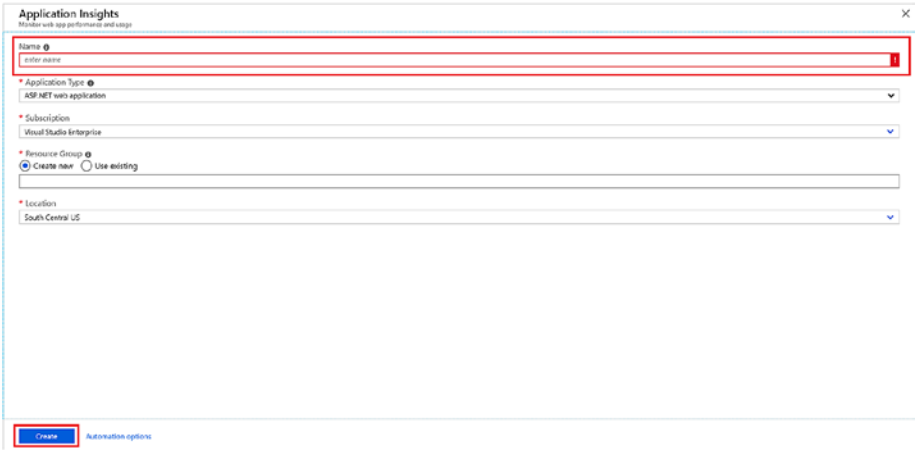

Figure 7-7. *Application Insights properties*

3. Once Application Insights is ready, go to the Dashboard and copy the integration key, as shown in Figure 7-8.

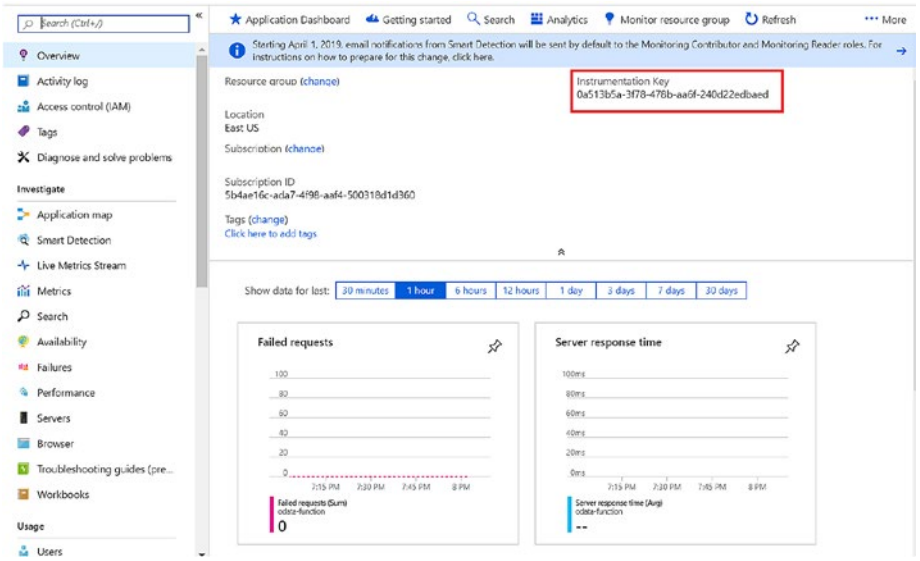

Figure 7-8. *Locating the instrumentation key*

4. Go to Azure Functions and select "Platform features" and then Application Settings. Click Add New Setting and add `APPINSIGHTS_INSTRUMENTATIONKEY`. See Figure 7-9.

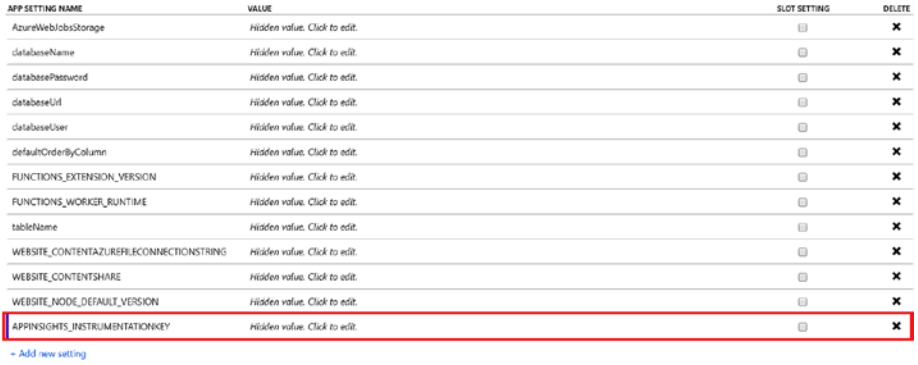

Figure 7-9. *Adding the key*

The Azure Functions app is integrated with Application Insights, and you can create custom telemetry events and other metrics in your Azure app function.

Disabling Built-in Logging

Because you have enabled Application Insights for your Azure Functions app, it is imperative to disable the built-in logging of Azure Functions that uses Azure Storage. The built-in logging is good for light-weight workloads such as testing in lower environments but is not intended for use in production. The reason for discouraging the use of built-in logging is that if the workload is high, then the logs might be incomplete because of Azure Storage's throttling.

To disable the built-in logging, you need to delete the `AzureWebJobsDashboard` setting from the app settings. Just make sure that this key is not being used in any applications.

Configuring Categories and Log Levels

Application Insights is like a plug-and-play service for Azure Functions, but if you use the default configuration, it can result in high-volume data, and you will end up hitting your data cap for Application Insights.

To avoid that, you can customize the configuration and send only the logs you require. To do that, you need to first understand the categories of logs in Azure Functions.

- The function runtime creates logs with a category that begins with `Host`.
- The "Function started," "function executed," and "function completed" logs have the category `Host.Executor`.
- The logs that you write in your function have the category "`Function`".

You can configure which log level to go to Application Insights for the previous categories in the `host.json` file.

```
{
      "logging": {
            "fileLoggingMode": "always",
            "logLevel": {
                  "default": "Information",
                  "Host.Results": "Error",
                  "Function": "Error",
                  "Host.Aggregator": "Trace"
            }
      }
}
```

In the previous settings, you are setting the following:

- For the categories `Host.Results` and `Function`, you will send logs with a log level of Error or higher to Application Insights.
- For the category `Host.Aggregator`, you will send logs with level Trace or Verbose and higher.
- For all other logs, you will send logs with a log level of Information or higher.

So, now you are done, and your function is ready to be monitored properly in production. Let's see it in action in Figure 7-10.

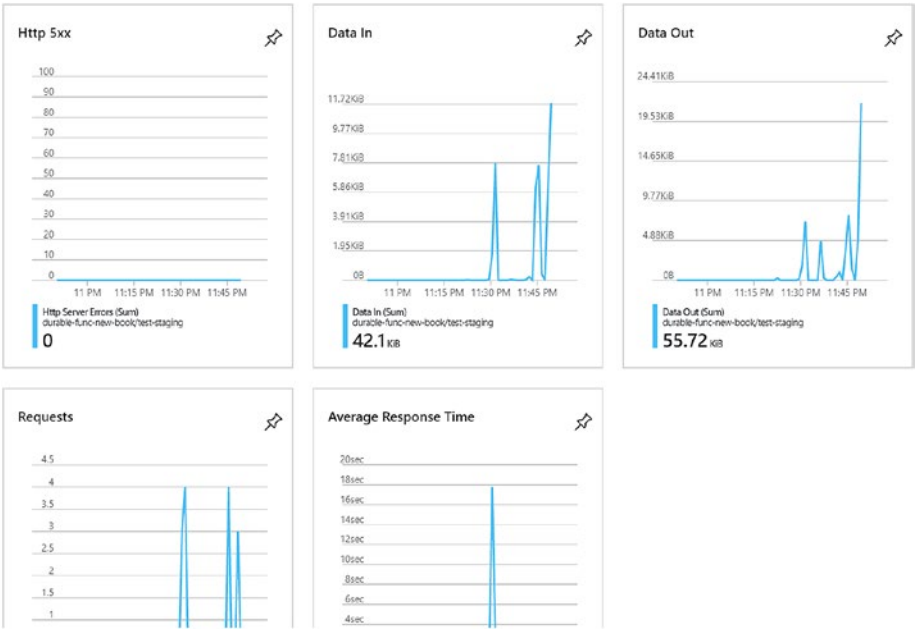

Figure 7-10. *Function in action*

Monitoring functions in production is a necessity. This enables you to monitor load, errors, and requests, and also lets you debug issues in production.

Securing Azure Functions

To make your functions production-ready, you have to secure them so that unauthorized access can be reduced. In today's world, securing your functions should be one of the most important tasks as there are lot of data breaches, and any data breach reduces people's trust of the company and its web sites. So, it is paramount for you to secure Azure Functions.

The good thing about Azure Functions is it provides you with an easy-to-use configuration to secure your functions. Let's go back to the HTTP-triggered function you created in this book. Let's copy the function URL, as shown in Figure 7-11.

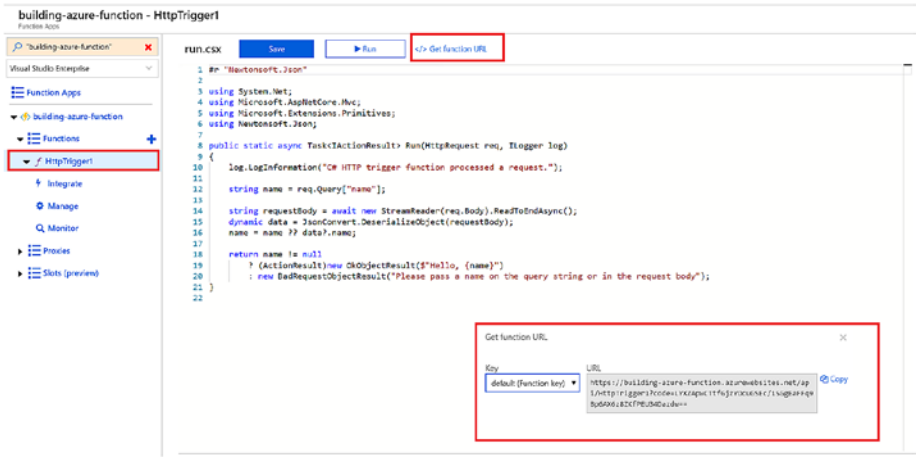

Figure 7-11. *Copying the function*

Now, copy this URL to a browser and add `&name={provideYourName}`, replacing `YourName` with any name, as shown in Figure 7-12.

Figure 7-12. *Replacing YourName*

As you can see, anyone who has your URL can access the function. Since this is a basic function that does not interact with your database, it's OK. But consider a function like the OData API function that you created in Chapter 4. Now, if your endpoint is not secured (i.e., it does not require any authentication/authorization), then you are actually inviting hackers to easily get your data.

To avoid this, you need to make sure that only authenticated users can access your function. Let's enable authentication/authorization for your function using Active Directory.

1. Go to the function that you want to secure and click "Platform features" and then Authentication/Authorization, as shown in Figure 7-13.

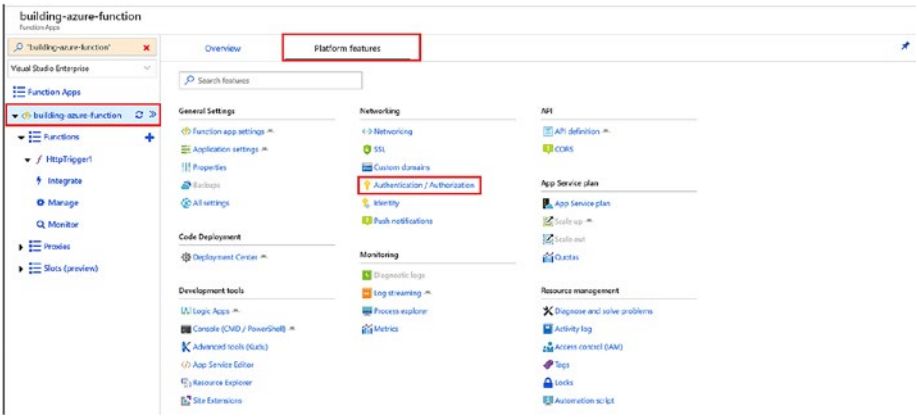

Figure 7-13. *Clicking Authentication/Authorization*

2. Set App Service Authentication to On and then set the "Action to take when request is not authenticated" drop-down to "Log in with Azure Active Directory," as shown in Figure 7-14.

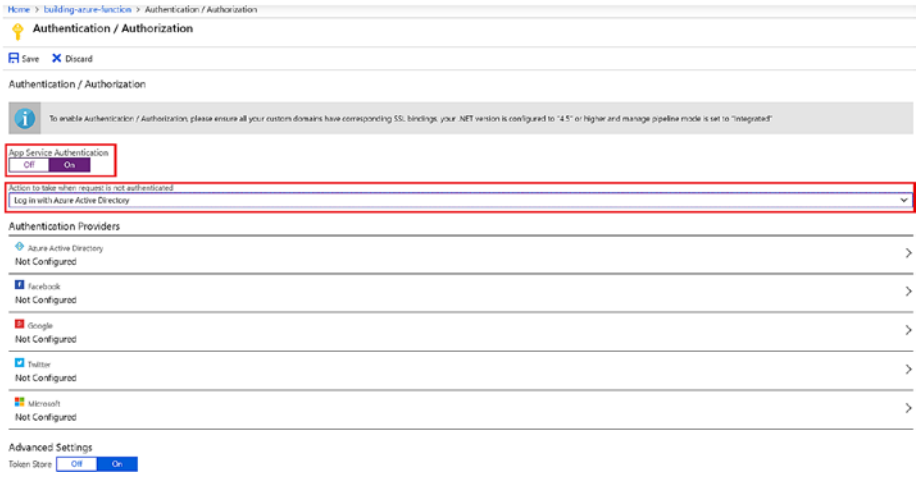

Figure 7-14. *Setting "Action to take when request is not authenticated"*

3. In the Authentication Providers section, click Azure Active Directory and set Management Mode to Express. Then select Create New AD App. Provide the name of the Active Directory and click OK. This will create the AD app and enable authentication/authorization for this function, as shown in Figure 7-15.

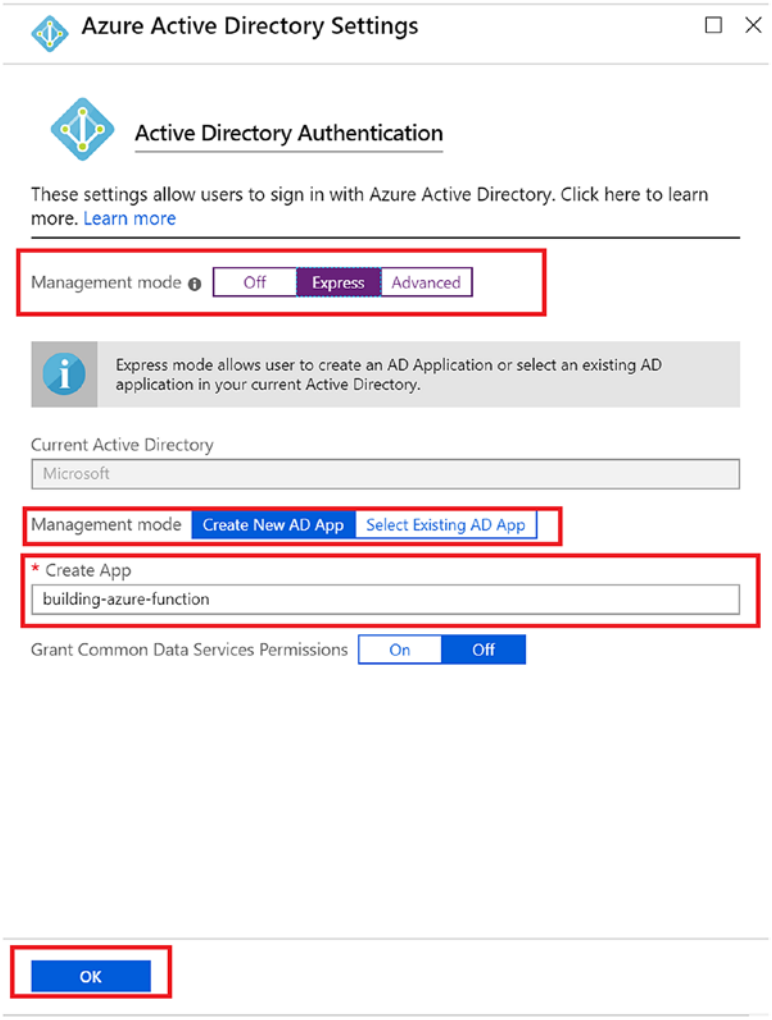

Figure 7-15. *Creating the AD app ID*

4. Let's try to hit the same URL that you did in Figure 7-12. You will see that now it asks you to log in before showing the result, as shown in Figure 7-16.

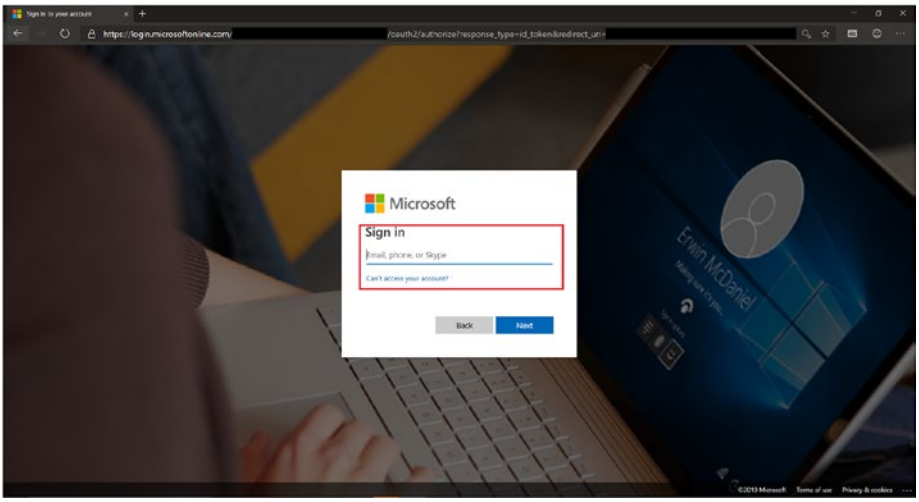

Figure 7-16. *Requesting a login*

You have now secured your function, and only the users who are in your AD application will be able to access this function.

Configuring CORS on Azure Functions

In most cases where you want to use a function as an API, you will be running Azure Functions and your UI or service that will call Azure Functions in different domains.

If that's the case, you will have to enable cross-origin site scripting (CORS) for your function so that you can access it from different domains.

To do that, let's follow these steps:

1. Let's go back to the function and click "Platform features." Then select CORS within the API section, as shown in Figure 7-17.

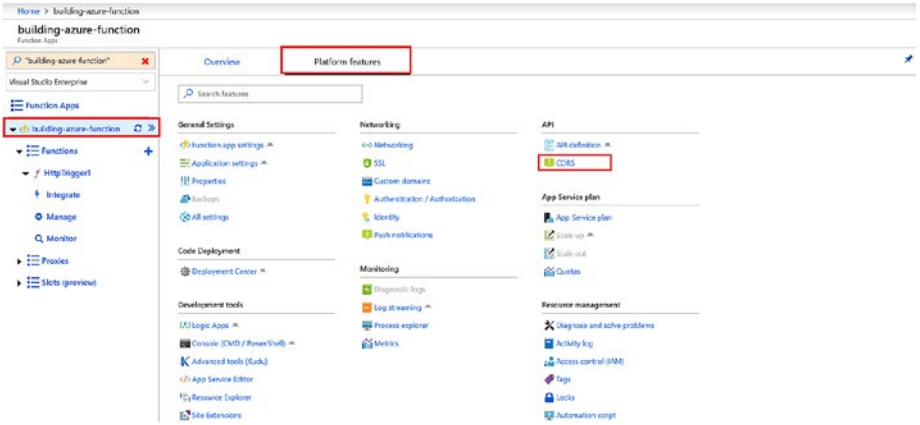

Figure 7-17. *Selecting CORS*

2. Select Enable Access-Control-Allow-Credentials, as shown in Figure 7-18. Let's say you have an application running locally on `http://localhost:5000` and you want to access this function from this application. In Allowed Origins, set this URL and click Save, as shown in Figure 7-18.

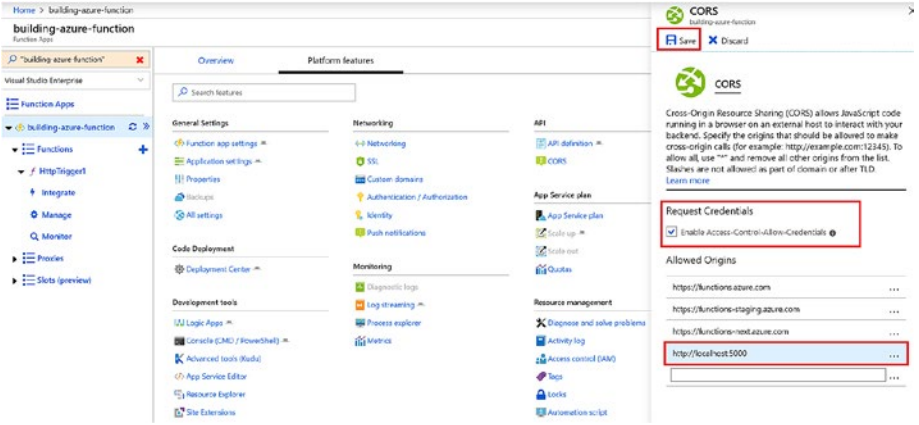

Figure 7-18. *Setting the URL*

With this you have enabled CORS on Azure Functions, and your function can now be accessed from all the domains that you have provided in Allowed Origins. To enable all the URLs, set it to *.

With this, you have come to the end of the book. I hope this book is just the beginning of your learning about the Azure Functions service. The more you dig into it, the more you will learn about Azure Functions.

Index

A

R. Sawhney, *Beginning Azure Functions*, https://doi.org/10.1007/978-1-4842-4444-9

B

C

D

E

F

G

H

I

J, K, L

M

N

O

P, Q

R

S

T, U

V

W, X, Y, Z