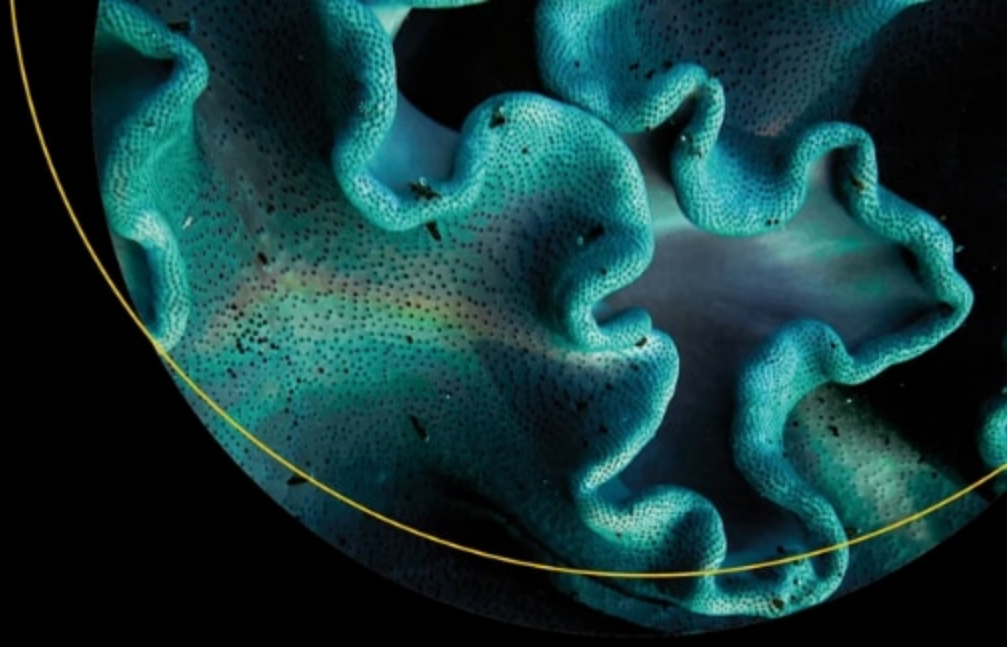

Beginning Azure Functions

Building Scalable and Serverless Apps

—

Second Edition

—

Rahul Sawhney
Kalyan Chanumolu

Apress®

Beginning Azure Functions

Building Scalable and Serverless Apps

Second Edition

Rahul Sawhney
Kalyan Chanumolu

Apress®

Beginning Azure Functions: Building Scalable and Serverless Apps

Rahul Sawhney
Hyderabad, India

Kalyan Chanumolu
Hyderabad, India

ISBN-13 (pbk): 978-1-4842-9202-0
ISBN-13 (electronic): 978-1-4842-9203-7
https://doi.org/10.1007/978-1-4842-9203-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Coordinating Editor: Mark Powers
Copyeditor: Kim Burton

Cover designed by eStudioCalamar

Cover image by Hushaan @fromtinyisles on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

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 http://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 (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Authors

Rahul Sawhney works as a software developer with Microsoft, India. He 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, Azure App Service, Azure 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.

Kalyan Chanumolu is a Senior Technical Program Manager at Microsoft. He works on building the engineering systems that power Azure—the world's computer. He is extremely passionate about distributed systems and cloud computing and has served as a technical reviewer for books on microservices, Azure, ASP.NET, and more. He has vast experience in software development, consulting, and migrating enterprise workloads to the cloud. He loves cycling, swimming, and reading books.

About the Technical Reviewers

Carsten Thomsen is a back-end developer primarily but works with smaller front-end bits as well. He has authored and reviewed several books and created numerous Microsoft Learning courses, all related to software development. He works as a freelancer/contractor in various European countries; Azure, Visual Studio, Azure DevOps, and GitHub are some of the tools he regularly uses. He is an exceptional troubleshooter and enjoys working with architecture, research, analysis, development, testing, and bug fixing. Carsten is a great communicator with mentoring and team-lead skills and enjoys researching and presenting new material.

Kapil Bansal is a lead DevOps engineer at S&P Global Market Intelligence, India. He has more than 14 years of experience in the IT industry, having worked on Azure cloud computing (PaaS, IaaS, and SaaS), Azure Stack, DevSecOps, Kubernetes, Terraform, Office 365, SharePoint, release management, application lifecycle management (ALM), Information Technology Infrastructure Library (ITIL), and Six Sigma. He has worked with companies such as IBM India Pvt Ltd, HCL Technologies, NIIT Technologies, Encore Capital Group, and Xavient Software Solutions, Noida. He has served multiple clients based in the United States, the United Kingdom, and Africa, such as T-Mobile, World Bank Group, H&M, WBMI, Encore Capital, and Bharti Airtel (India and Africa).

Introduction

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

We'll start by taking you through the basics of serverless technology and Azure Functions and then cover the different pricing plans of Azure Functions. After that, we'll dive into how to use Azure Functions as a serverless API. We will then walk you through the Durable Functions model, disaster recovery, and georeplication.

Moving on, we provide practical recipes for creating different functions in Azure Functions using the Azure portal and Visual Studio Code. Finally, we discuss DevOps strategy and 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 working with Durable Functions and 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 their differences. We also talk about the different pricing plans of Azure Functions.

In Chapter 2, you create the first function using the Azure portal and then Visual Studio Code. We also discuss the Azure Functions file hierarchy, configuration, and settings.

In Chapter 3, you learn about triggers and bindings. We also discuss changes to Azure Functions 2.0 bindings, and you create Azure Blob Storage–triggered functions.

Chapter 4 goes through the differences between monolithic applications vs. microservices. Then, we talk about converting a monolithic application to microservices using Azure Functions. You create some functions and then learn about proxies.

Chapter 5 starts with an overview of the Durable Functions pattern and bindings. You also learn about performance and scaling in a durable function. You create your first durable function and learn about disaster recovery and georeplication.

Chapter 6 looks at deploying functions to Azure, using a CI/CD pipeline and ARM templates.

Chapter 7 covers the built-in logging capabilities of Azure Functions and looks at Application Insights and how it is used to monitor Azure Functions. We also talk about securing Azure Functions using Azure Active Directory and how to configure cross-origin site scripting (CORS) in Azure Functions.

Source Code

The source code for this book can be downloaded from `github.com/apress/beginning-azure-functions-2e`.

Let's get started!

CHAPTER 1

Introduction to Azure Functions

The software industry is now in an era where apps and services increasingly rely on the cloud. Applications are globally available and need to scale at a moment's notice. Deployments need to be seamless without any perceivable downtime to the end customer. Periodic maintenance windows for making infrastructure upgrades are a thing of the past.

To help developers be more productive and focus on developing software rather than worrying about provisioning and maintaining servers, and orchestrating deployments, cloud platforms such as Microsoft Azure, Amazon Web Services, and Google Cloud Platform, provide *serverless compute* offerings.

Azure Functions is one such product for *serverless computing*. With Azure Functions, developers can write less code and manage less infrastructure, and organizations can bring applications to market quickly with fewer upfront costs. Before diving into Azure Functions, let's discuss serverless computing and what it means.

This chapter covers the following topics.

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

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_1

Overview of Serverless Computing

Serverless computing, also known as *function as a service* (FaaS), is an event-driven application design and deployment model that allows developers to build and run applications without managing servers. 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 are managed by a cloud service provider such as Amazon, Microsoft, or Google, and you have to manage 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 much of your time to doing what's most important: developing the code/application. The cloud service provider maintains and owns the server and charges you for the usage.

Serverless computing is a paradigm shift in computing. Deploying applications on physical machines used to take months; with serverless computing, deploying takes just a few minutes or less.

Figure 1-1 shows how the underlying infrastructure is abstracted from the developer.

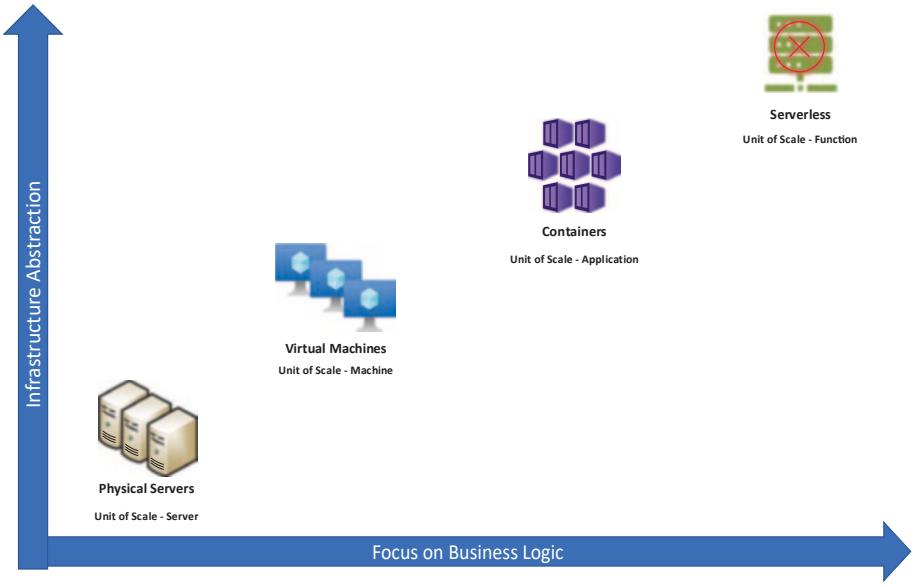

Figure 1-1. *Infrastructure abstraction in cloud computing*

Overview of Azure Functions

With Azure Functions, you can start writing code for your application without worrying about the infrastructure required to run and scale 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.

The following are some important Azure Functions features.

- **Browser-based interface**: You can write and test your code directly in the Azure portal 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, and Twilio, and other Azure services, like Cosmos DB, 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 enhance your productivity and provide lots of different options for developers to choose from. Still, we have heard developers becoming 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.

The next section discusses the differences between Azure Functions and Azure WebJobs and in which scenario you should use each.

Azure Functions Runtime Versions

The Azure Functions runtime provides a hosting platform for functions written in many different languages and supports a wide variety of triggers and bindings. Support for new languages and integration with other cloud services are distributed as updates to the runtime.

Azure Functions currently has four versions of the runtime host. Table 1-1 describes when these versions should be used.

Table 1-1. *Azure Functions Runtime Host versions*

Version	Description
4.x	The latest and recommended runtime version for functions in all languages. This version supports running C# functions on .NET 6.0 and above.
3.x	This runtime version supports all languages. This version supports running C# functions on .NET Core 3.1 and .NET 5.0.
2.x	Currently, this version is in maintenance mode. All the enhancements are provided in newer versions only. This version supports running C# functions on .NET Core 2.1
1.x	This version is in maintenance mode and should only be used for C# apps that must use .NET Framework. It only supports development locally on Windows, the Azure Stack Hub portal, and the Azure portal.

Going forward in this book, functions are created using version 4.x.

Azure Functions vs. Azure WebJobs

Azure Functions and Azure WebJobs are code-first integration services designed for developers. Both support features such as authentication, Application Insights, and source control integration. Table 1-2 notes some differences between Azure Functions and WebJobs.

Table 1-2. *Azure Functions and WebJobs Differences*

	Azure Functions	**Azure WebJobs**
Trigger	Azure Functions can be triggered with many triggers, as discussed in the upcoming chapters. Azure Functions don't run continuously.	There are two types of WebJobs: triggered and continuous.
Supported Languages	Azure Functions support the following languages: C#, Java, JavaScript, PowerShell, Python, and TypeScript.	Azure Web Jobs also supports languages like JavaScript, C#, and F#.
Deployment	Azure Functions run using a classic\ dynamic App Service plan.	It runs as a background process in app services, such as API apps, mobile apps, and web apps.

Azure Functions is a great choice in most scenarios because it offers many programming languages, flexible pricing options, and increased developer productivity. However, the following are two scenarios where you should use WebJobs instead.

- You have an Azure App Service environment where you want to run some code snippets and maintain the same DevOps pipeline and environment.
- You need to customize the behavior of the host that listens for triggers and calls functions, such as having custom retry policies or error handling.

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

Azure Functions vs. Azure Logic Apps

Azure Logic Apps is a designer-first low-code/no-code cloud platform where you can create and run automated workflows using a visual designer and selecting operations from prebuilt connectors. You can quickly build workflows by dragging and dropping API connectors onto the Logic Apps designer, specify triggers, configure inputs/outputs and connect one action to another. All from a user interface without having to write complex code to integrate with other Azure services. Table 1-3 notes some differences between Azure Functions and Logic Apps.

Table 1-3. *Azure Functions Logic Apps Differences*

	Azure Functions	**Logic Apps**
Trigger	An Azure Function can have only one trigger.	Azure Logic Apps can have up to ten triggers.
Supported Languages	Azure Functions supports the following languages: C#, Java, JavaScript, PowerShell, Python, and TypeScript.	The source code generated by the graphic designer can be accessed as a JSON template.
Development	Azure Functions can be developed on the Azure portal using Visual Studio or any supported IDE and have support for debugging. They can be deployed using Visual Studio, Azure Pipelines, or GitHub Actions.	Logic Apps are authored using the Azure portal or Visual Studio using an extension and produce a JSON template that can be checked into source control and deployed using Azure DevOps, Visual Studio, or FTP.

(*continued*)

Table 1-3. (*continued*)

	Azure Functions	Logic Apps
Hosting Plans	Azure Functions runs using a classic\dynamic App Service plan.	Logic Apps run in a Standard plan (Single tenant)/ Consumption plan (Multitenant).
Billing	Billing is based on resource consumption and executions per second.	Billing is based on the number of times triggers, actions, or connectors are executed.

Logic Apps are especially useful when you coordinate simple actions across multiple systems and services to automate business processes. They provide rich capabilities for control flow operations such as conditions, switches, loops, and scopes. You can also implement error and exception handling in your workflows. Azure Functions are better suited for workflows with complex business logic and data transformations, reuse of components and libraries across other projects and scenarios that require more control on security, reliability, and availability. Azure Functions and Logic Apps can call each other, allowing you to design a solution that can take advantage of both technologies as appropriate for your scenario.

Azure Functions Pricing Plans

Azure Functions supports three pricing plans.

- Consumption plan
- App Service plan
- Premium plan

Let's look at each plan in detail.

Consumption Plan

Azure Functions' Consumption plan allows you to 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 website, instead of using WebJobs, you can use Azure Functions and save a significant amount of money.

The metric for calculating price in Azure Functions is gigabytes/seconds (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.

The Consumption plan grants one 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 is the same plan used for hosting a website, 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 times out. In contrast, 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, having it on the same App Service plan as the rest of your company would make your other applications vulnerable because Azure Functions would be using the same shared resources and thus would make the other applications run slower.

Premium Plan

Azure Functions' Premium plan is a dynamic scale hosting plan for function apps. Azure Functions Premium plan offers the following benefits.

- More predictable pricing compared to the Consumption plan
- Premium instance sizes, such as one, two, and four core instances
- Unlimited execution duration, with 60 minutes guaranteed
- Virtual network connectivity
- Perpetually warm instances, eliminating one of the biggest serverless problems of cold start

With the Premium plan, there is no execution charge. The billing is based on the number of cores and memory allocated across instances, whereas in the Consumption plan, you are billed per execution and memory used.

In the Premium plan, at least one instance must be allocated at all times. Also, all function apps in a Premium plan share allocated instances. This results in a minimum monthly cost per active plan regardless of whether the function is alive or idle.

How the Premium Plan Solves a Cold Start Problem

A *cold start* refers to a phenomenon where an application that isn't used for a while takes longer to start up. In Azure Functions, latency is the time a user must wait for their function to execute.

In Azure Functions, a cold start results in higher latency for a function that hasn't been called recently. Let's now look at what happens under the hood when you write a function.

- Azure allocates a server with function and a server with capacity.
- The function runtime then starts up on that host.
- Your code then executes.

In the Consumption plan, if an event or execution doesn't occur, your app may scale down to zero instances. So, after that, the first call the function receives goes through the three steps and results in higher latency for the first few calls. Figure 1-2 shows how a cold start differs from a warm start.

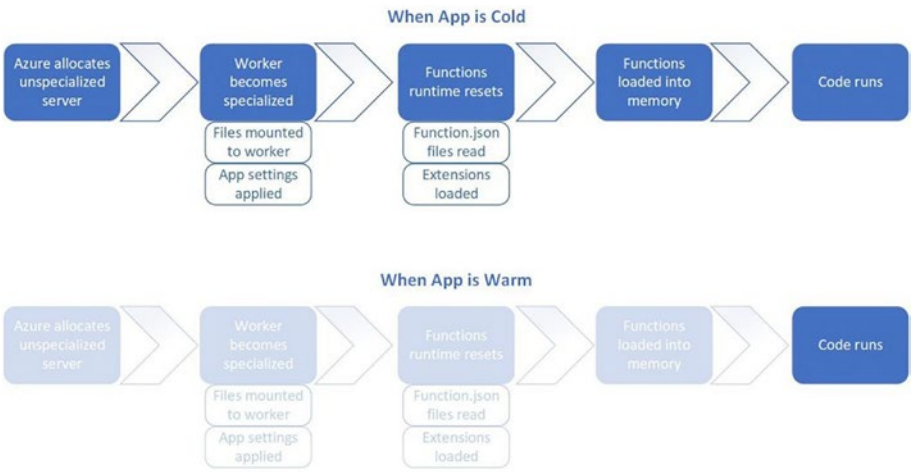

Figure 1-2. *Cold start vs. warm start*

On the other hand, in the Premium plan, there are two ways to solve a cold start problem. First, always ready instances and pre-warmed instances. The Premium plan lets you configure up to 20 instances as *always ready* instances. When an event triggers a function app, the execution is routed to an *always ready* instance. Additional instances are warmed as a buffer as the function gets more traffic. These buffer instances are then used during scale to prevent a cold start. These buffer instances are called *pre-warmed instances*.

Summary

You should now have a basic understanding of serverless computing, Azure Functions, and app pricing. Chapter 2 builds upon this chapter and gets started with creating Azure Functions.

CHAPTER 2

Creating Azure Functions

In this chapter, you will learn how to create an Azure function using the Azure portal and Visual Studio Code. You will also gain insight into a function's file hierarchy, configuration, and settings.

In this chapter, you start using Azure Functions. The latest runtime version is 4.0, which is used throughout the book's examples. Over the course of this chapter, we will cover the following topics:

- Creating an Azure portal account
- Creating functions using the Azure portal
- Azure Functions file hierarchy, configuration, and settings

Let's examine the two ways you can create functions with Azure Functions.

Creating an Azure Function Using the Azure Portal

In this section, you will create your first function using the Azure portal: a boilerplate application.

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_2

Creating an Azure Account

First, you need to log in to the Azure portal. You can go to `http://portal.azure.com` to sign in. If you don't have an Azure subscription or are a 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.

To get started with Azure Functions, visit `https://functions.azure.com/` and click the Free Account button.

Creating Your First Function App Using the Azure Portal

Once you have completed the sign-up/sign-in process, you are taken to the Azure portal dashboard. Now you can create a function app.

On the dashboard, click "Create a resource" in the left panel. In the menu, click Compute, search for "Function", and select Function App, as shown in Figure 2-1.

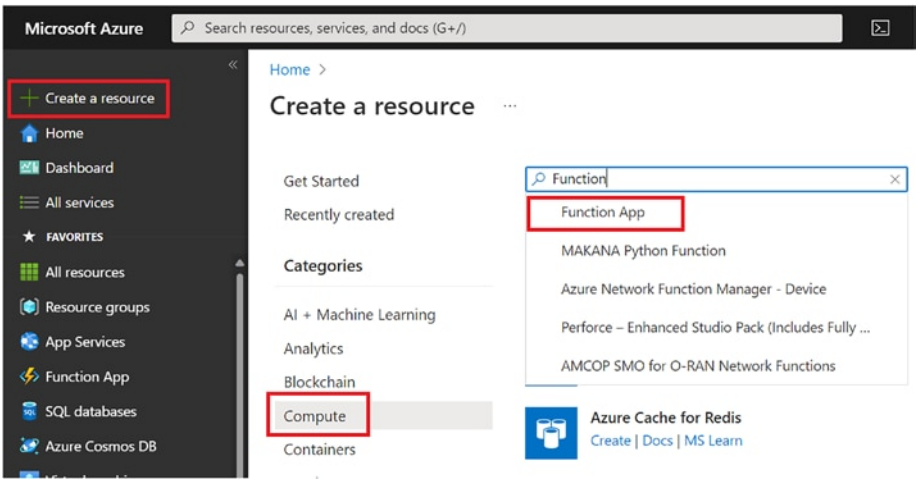

Figure 2-1. *Creating a function app resource*

You are asked to provide details such as the app name, resource group, OS, hosting plan, and so on. Refer to Figure 2-2 to enter the settings.

- **Function App Name**: This is the name of the function app. The app name in this example is `BuildingAzureFunction`, so the URI to access the function app is `BuildingAzureFunction.azurewebsites.net`.
- **Subscription**: This is the subscription under which the function app is created. An Azure account can have multiple subscriptions used for maintenance and billing purposes.
- **Publish:** There are two options to publish this function: Code and Docker Container. In Code Publish, you can choose among the Consumption, App Service, or Premium plans; but the Consumption plan is not available in Docker Container.
- **Plan Type**: By default, the Consumption plan is selected. This plan is ideal for most scenarios, and you are billed only for the duration your function executes. But you can choose the App Service plan or Premium as well.
- **Region**: Always choose the location nearest to where you expect the majority of the traffic to come for your function app.
- **Storage**: Every function app requires storage for logging function executions and managing schedules for function triggers. To use an existing storage account, go to Hosting and select Azure Storage Account. By default, it creates a new storage account.

- **Operating System**: Azure Functions supports both Linux and Windows. You can select either of them based on your preferred runtime stack.

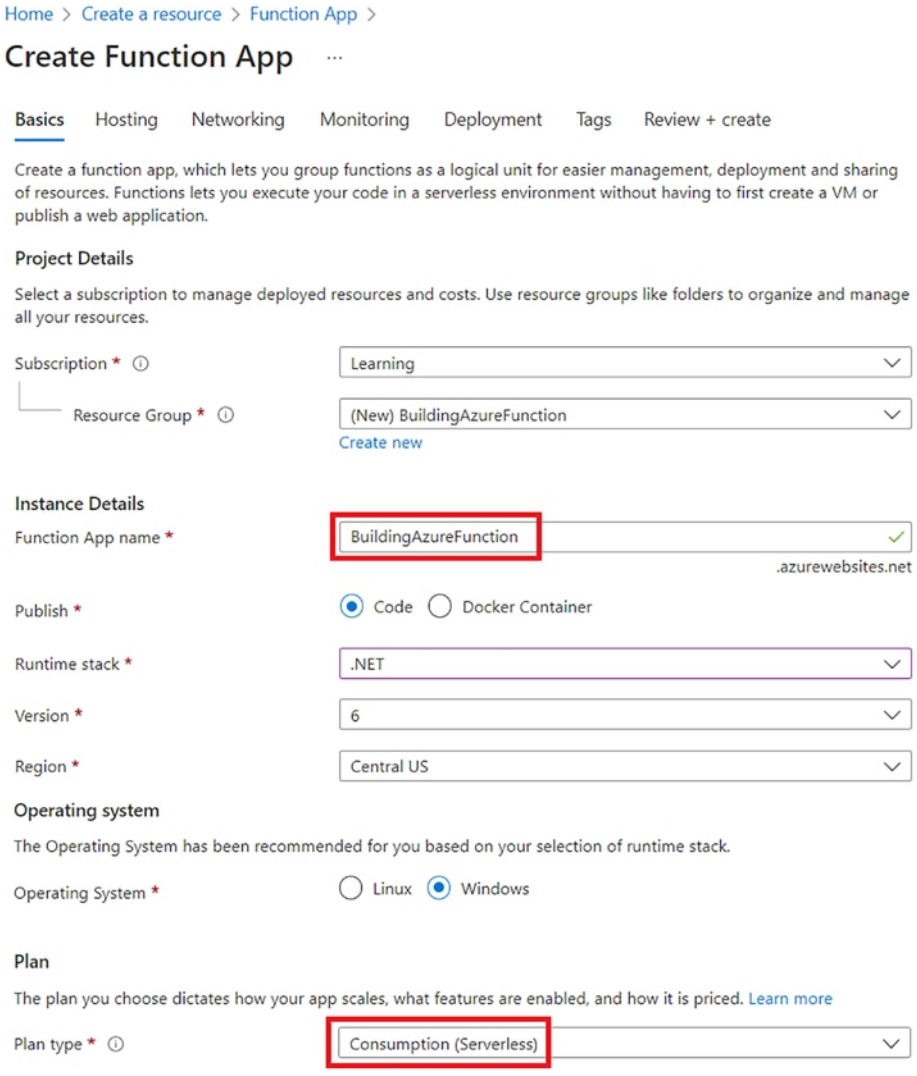

Figure 2-2. *Create function app*

Click the Review + Create button to get the summary of the function, and then click the Create button, as shown in Figure 2-3.

Figure 2-3. *Summary view of the function app*

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

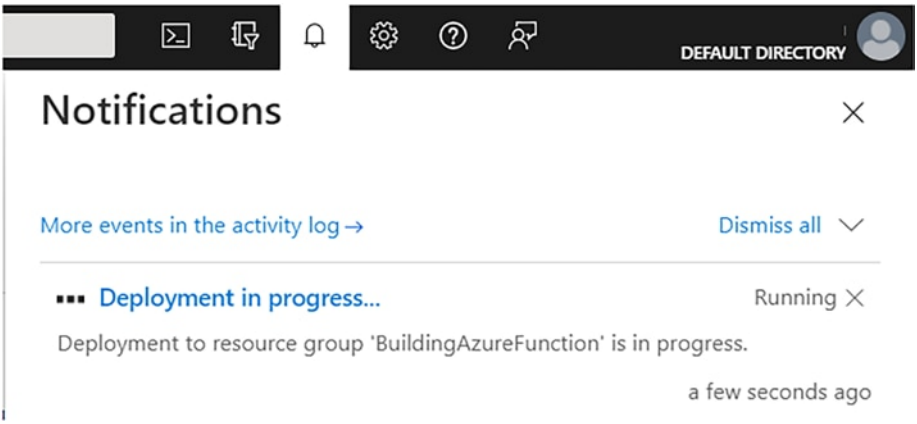

Figure 2-4. *Checking the status*

Once the deployment is completed, your first Azure Functions app, named `BuildingAzureFunction`, is ready, and you can now start coding for it. To go to the function app you created in the previous steps, navigate to the resource group. Please refer to Figure 2-5.

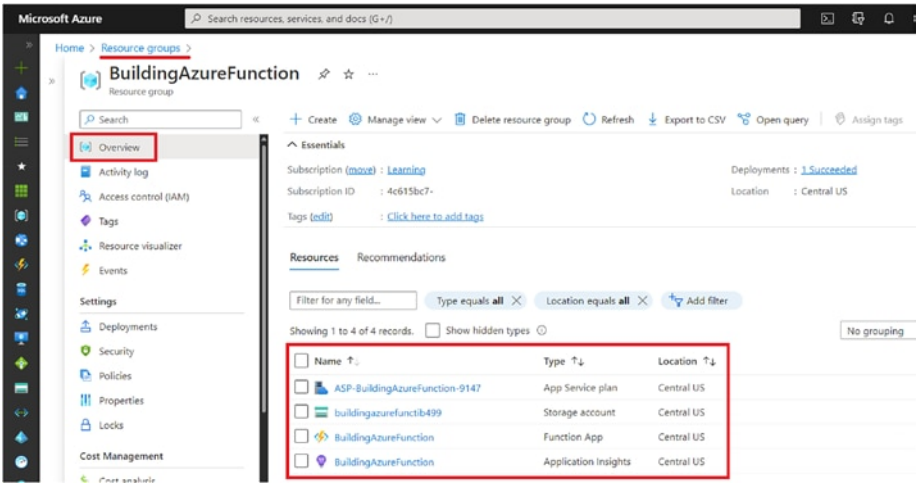

Figure 2-5. *Select the resource group you created*

Creating Your First Function in the Function App

Your function app is ready and deployed, but it is not usable because you haven't created any functions yet. A *function app* is like a container that can hold multiple functions. A *function* is the component of the function app where you code your logic.

To create your first function, click `BuildingAzureFunction`, as shown in Figure 2-5. Then click the Function menu inside the Functions subdirectory, and click Create, as shown in Figure 2-6.

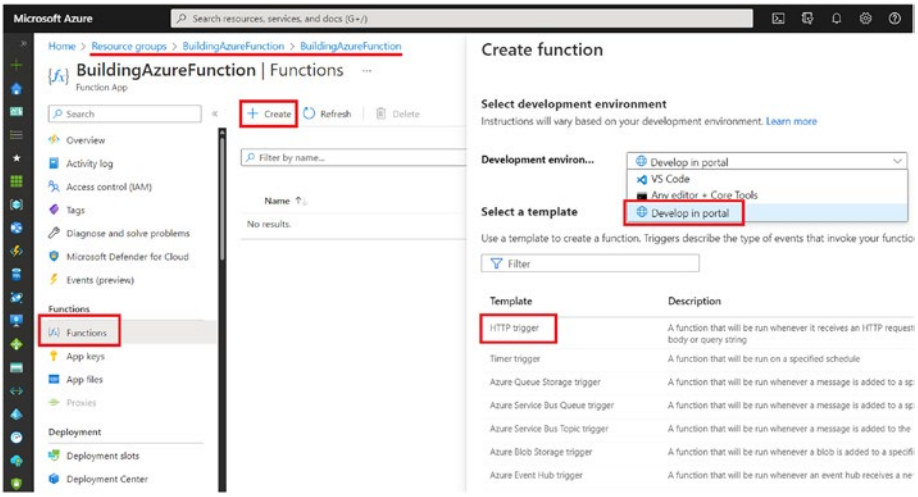

Figure 2-6. *Create function app*

For this example, choose "Develop in portal". However, you can use VS Code or any other editor that you are comfortable with. Several trigger templates can trigger your Azure Function. Select the HTTP trigger template for this example and provide your function app's details and the authorization level, as shown in Figure 2-7.

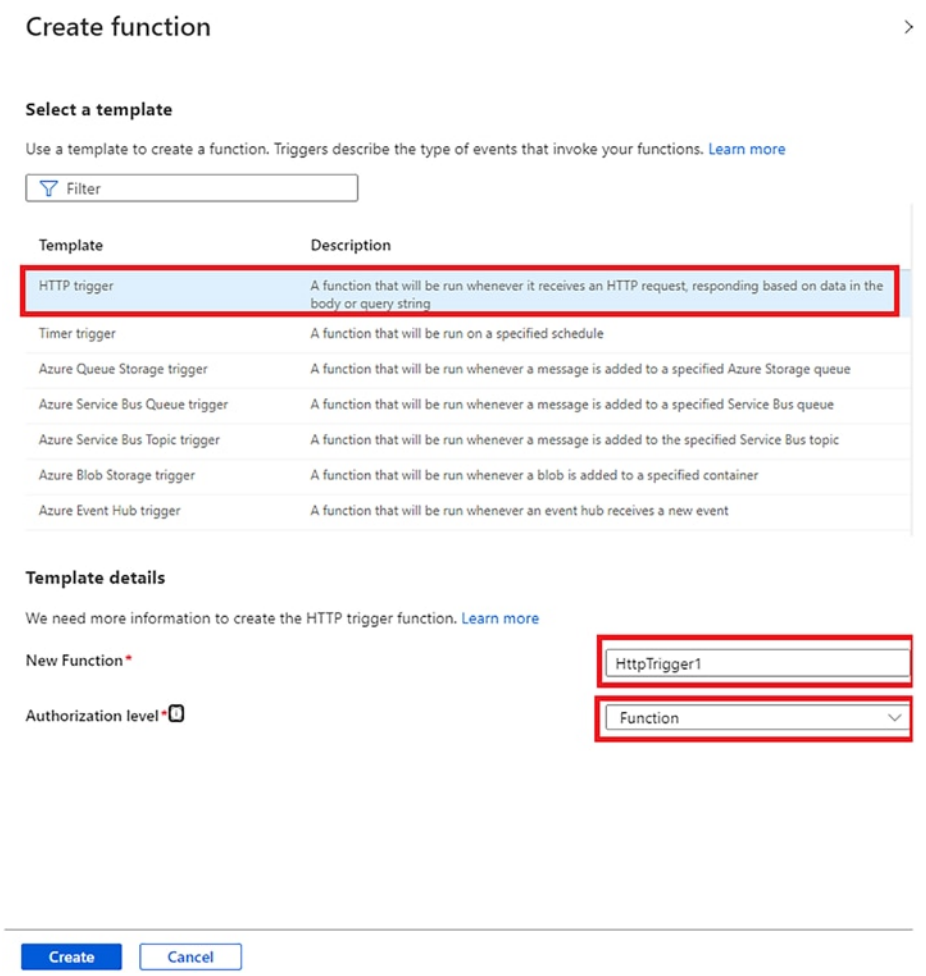

Figure 2-7. *Create HTTP triggered function*

The following describes the three authorization levels.

- **Function**: The authorization level function uses the function key. The function key can be found on the Keys Management panel on the portal.

- **Admin**: Admin authorization level uses the master key. Again the master key can be found in the Keys management panel on the portal.
- **Anonymous**: No API key is needed to access the function.

Click the Create button, and your first HTTP trigger function is created. It comes with basic boilerplate code. This code should be good enough for you to test your function. Click the Test/Run button, as shown in Figure 2-8.

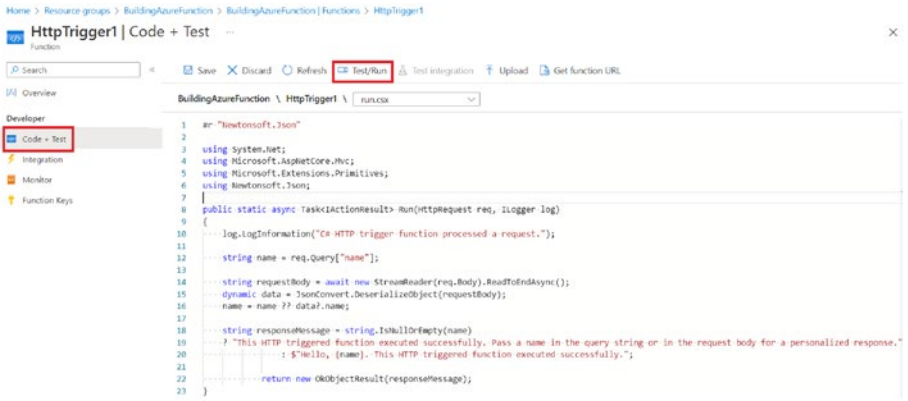

Figure 2-8. *HTTP Function boilerplate code*

In the Test panel, select Key as the default (Function key), update the body parameter as needed, and click Run, as shown in Figure 2-9.

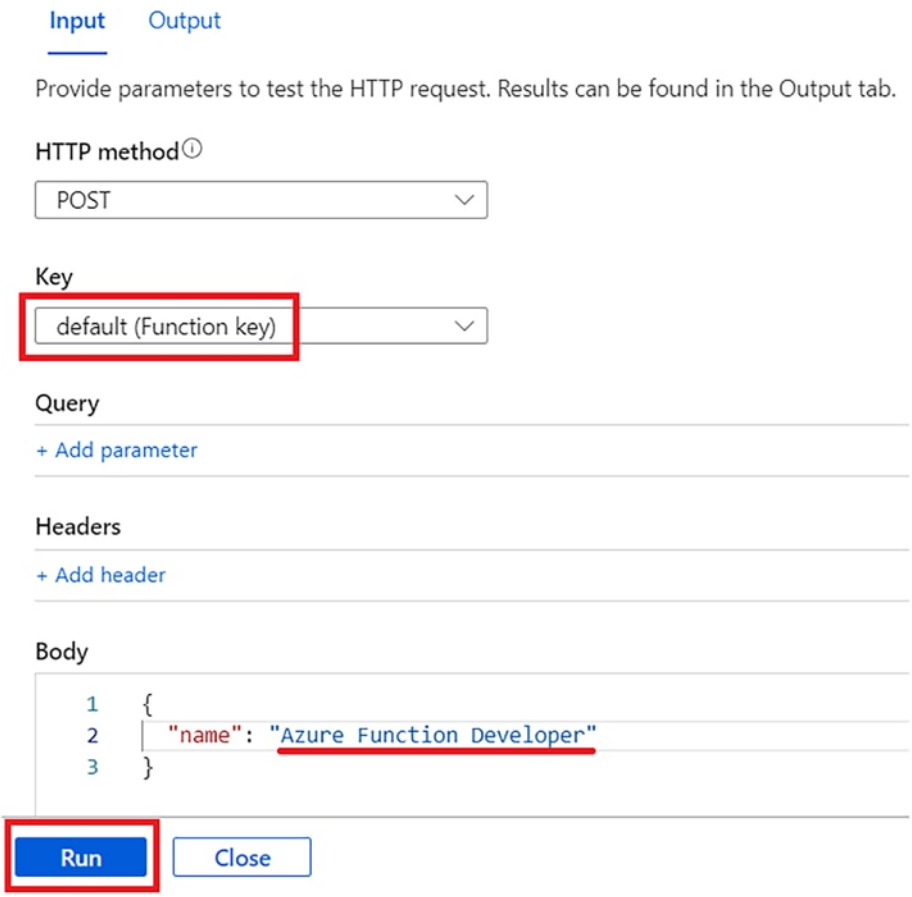

Figure 2-9. *Running your Azure Function*

The output is shown in the Output tab, as seen in Figure 2-10.

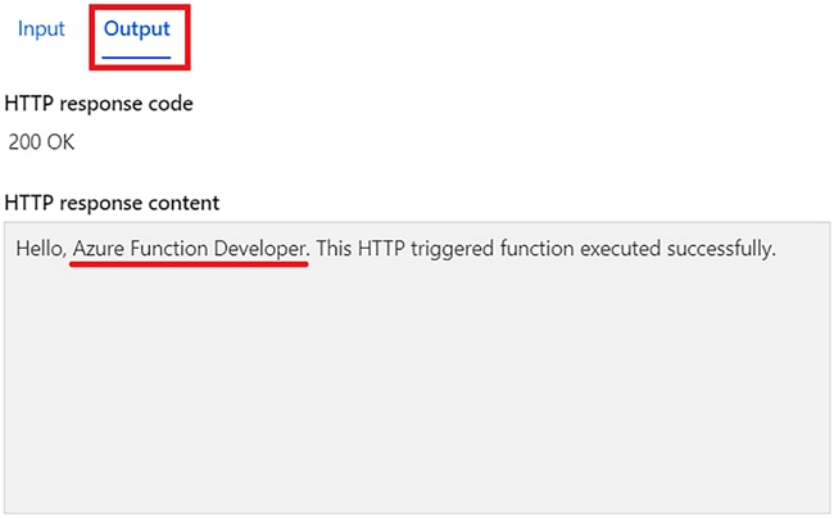

Figure 2-10. *Azure Function output*

After validating that the function is running as expected, copy the function URL, as shown in Figure 2-11.

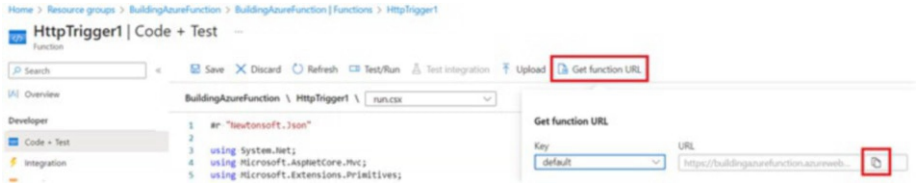

Figure 2-11. *Copy function URL*

Create another HTTP trigger function, HttpTrigger2, within the same function app.

You have created your first running Azure Functions app.

Next, let's look at file hierarchy, configuration, and settings in Azure Function.

File Hierarchy, Configuration, and Settings in Azure Functions

The code for all the functions in a specific function app is located in the `/wwwroot/` folder. The hierarchy of the function created is shown in Figure 2-12.

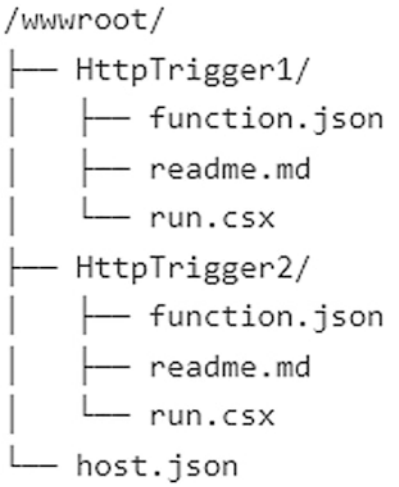

Figure 2-12. *File hierarchy*

Note All functions in a function app must share the same language stack in version 2.x and higher of the Azure Functions runtime.

You can also view the hierarchy of your function. Go to the Development Tools menu, select Advanced Tools, and click the Go button, as shown in Figure 2-13. This opens Kudu, an engine behind git deployments in Azure. You can read more about it at `https://github.com/projectkudu/kudu/wiki`.

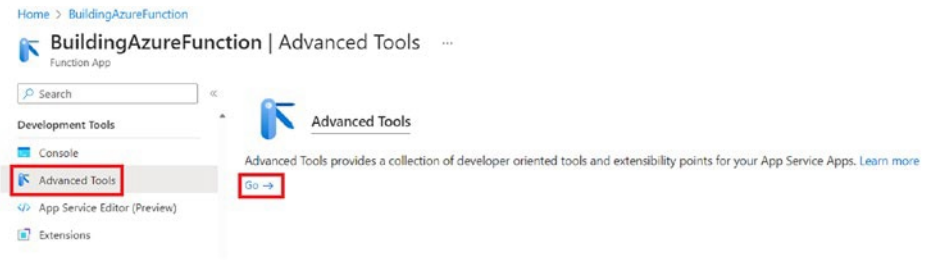

Figure 2-13. *Advanced Tools in Functions*

In the Kudu back end, you can now navigate to Tools ➤ Zip Push Deploy to see the file hierarchy, as shown in Figure 2-14.

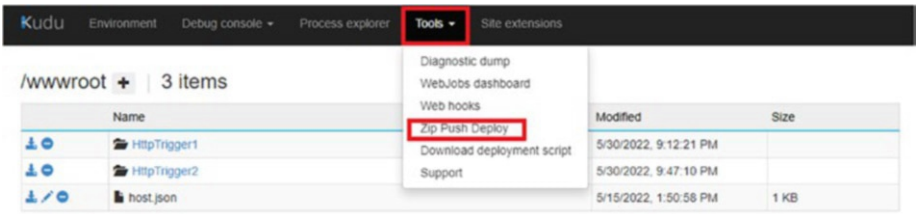

Figure 2-14. *File hierarchy in Kudu*

The `host.json` file is at the root of the function app folder and contains configuration options that affect all functions in a function app instance. In this file, you can specify the extension bundle version and the default `FunctionTimeout`.

The `function.json` file is inside each function folder and defines the function's trigger, bindings, and other configuration settings. The runtime uses this file to determine the events to monitor and how to pass data into and return data from a function execution. For compiled languages like C#, this file is generated automatically. However, for scripting languages like JavaScript and PowerShell, you must author this file yourself.

All the application-level extensions, such as Cosmos DB, HTTP triggers, queues, and so on, reside under the extensions object and not in the root of the function.json object, as shown in Figure 2-15.

```
"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 2-15. *Application-level extensions*

All the logging-level settings for Azure Functions reside under the `logging` object, as shown in Figure 2-16.

Figure 2-16. *Logging-level extensions*

Summary

In this chapter, you created your first function app and function. Also, you learned about the Azure Functions file hierarchy, configuration, and settings. In the next chapter, you'll learn how functions are triggered and how other Azure services can be declaratively connected to Azure Functions.

CHAPTER 3

Understanding Azure Functions Triggers and Bindings

This chapter teaches you two main components of Azure Functions: triggers and bindings.

This chapter covers the following topics.

- Overview of triggers and bindings
- Register binding extensions
- Binding expression patterns
- Creating an Azure Blob Storage–triggered function

Overview of Triggers and Bindings

A *trigger* defines the external event that can invoke the Azure function. Each function can have only one trigger. Triggers usually carry data associated with the event, which is the payload that triggers the function.

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_3

Azure Functions support several triggers, and Microsoft constantly adds support for more services. You can write a custom trigger of your own if you wish to. The following describes some common triggers.

- **HTTPTrigger**: This trigger is fired when the Azure function detects an incoming HTTP request. The request body and query parameters are provided as input parameters to the function. In Chapter 2, you created an HTTP-triggered function using Visual Studio Code.
- **BlobTrigger**: This trigger gets fired when a blob is created or updated on an Azure storage account or a blob. The contents of the blob are provided as an input parameter to the function.
- **QueueTrigger**: This trigger gets fired when a new message arrives in Azure Queue Storage.
- **EventHubTrigger**: This trigger gets fired when an event is delivered to the Azure Event Hubs event stream.
- **TimerTrigger**: This trigger is called on a scheduled basis. You can set the time to execute the function using this trigger.
- **Service Bus Trigger**: This trigger gets fired when a new message enters an Azure Service Bus topic or queue.
- **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.

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 has the following supported bindings, and the list is ever-growing.

- Blob Storage
- Cosmos DB
- Event Grid
- Event Hubs
- HTTP and webhooks
- Queue Storage
- Table Storage
- Service Bus
- SendGrid
- SignalR
- Twilio

Bindings are optional in Azure Functions, and you can have multiple input and output bindings. 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 by visiting https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-bindings.

Register Binding Extensions

Extensions are a way of creating and distributing triggers and bindings for Azure Functions. In the Azure Functions 2.0 runtime, a new binding model was introduced where the runtime is decoupled from the extensions except for a few core bindings, like HTTP and timer. It provides additional flexibility and reduces the load time by loading only the extensions referenced in the function app. It also means that the runtime has no knowledge of the extensions, so they must be registered before use.

Extensions are distributed as NuGet packages, and these extensions are registered by installing the required NuGet package for the extension.

Binding Expression Patterns

Binding Expressions is a very useful capability of triggers and bindings. In your function code, function parameters, and function.json file, you can use expressions that resolve to values from various sources during runtime instead of specifying values during compilation or deployment time.

For example, you have a function with blob output binding, and you need to create a file with the current date and time as the file name. You could use the {DateTime} binding expression in the function.json file, as follows.

```
{
  "type": "blob",
  "name": "blobOutput",
  "direction": "out",
  "path": "my-output-container/{DateTime}.txt"
}
```

Let's look at another example of using function app settings in binding expressions. Assume that you have a timer-triggered function, and you do not want to hardcode the timer schedule in your code. You could create a new app setting called ScheduleConfiguration to specify the schedule and reference the app settings, as follows. App setting binding expressions are wrapped in percent signs rather than curly braces.

```
[Function("RecurringOrdersProcessor")]
public async Task Run([TimerTrigger("%ScheduleConfiguration%",
RunOnStartup = false, UseMonitor = false)] MyInfo myTimer)
{

      // Function App code

}
```

The following are types of binding expressions.

- App settings
- Trigger file name
- Trigger metadata
- JSON payloads
- New GUID
- Current date and time

Note You can find all the supported binding expressions by visiting https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns.

Installing Extensions Using Visual Studio Tools

With Visual Studio Code or Visual Studio, you are referencing the extensions package directly from the project. The tool handles the installation and registration of the extensions.

Let's look at an example of installing an extension.

In the Visual Studio Code terminal window, run the following command to install and register the Azure Blob Storage bindings and trigger to your function.

```
Dotnet add package Microsoft.Azure.Functions.Worker.Extensions.
Storage.Blobs --version 5.0.0
```

Installing Extensions Using the Azure Functions Core Tools

Azure Functions Core Tools lets you develop and test your functions on your local computer from the command prompt or terminal. It is available as a download for Windows, macOS, and Linux. You can download and install it on your developer machine and use the `func extension install` command for managing extensions. This command lets you install an extension and register it to the function.

The following is an example of installing an extension.

```
func extensions install –package Microsoft.Azure.Functions.
Worker.Extensions.Storage.Blobs –version 5.0.0
```

This command installs the blob extension to Azure Functions, allowing you to configure your function for a blob trigger.

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

Creating an Azure Blob Storage–Triggered Function

This section teaches you how to create an Azure Blob Storage–triggered function using C# and Node.js. We will walk you through each process step-by-step. Visual Studio Code is used to create the function. To set up the system to create a function, review the "Creating an Azure Function Using Visual Studio Code" section in Chapter 2.

This function resizes the image once uploaded to 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 your machine as covered in Chapter 2. Once the machine is set up, open Visual Studio Code, go to the Extensions section, and install the latest versions of Azure Functions and C# extensions. Once this is done, go to the Azure Functions menu and add a new function with the following steps.

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

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

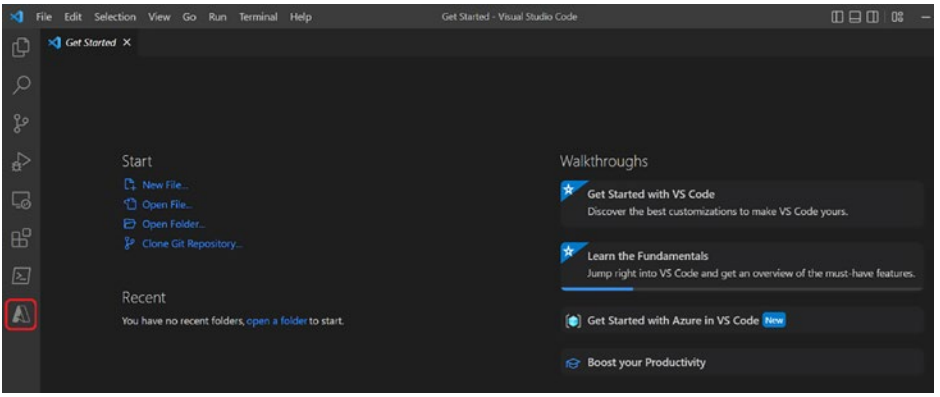

Figure 3-1. *Clicking the Azure logo*

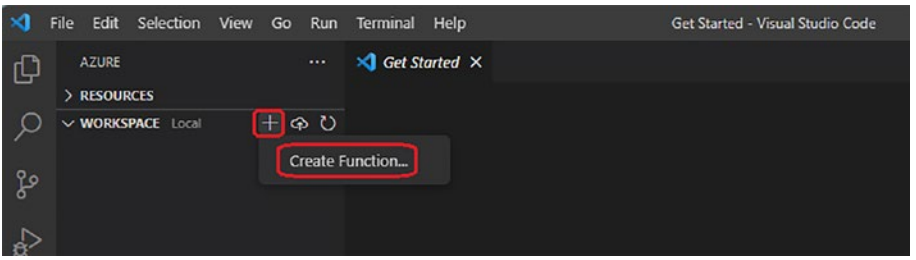

Figure 3-2. *Creating a new function app*

Create a new project on the next dialog and select a folder to save the project files.

Select the language in which you want to code your function. In this example, C# was selected, as shown in Figure 3-3.

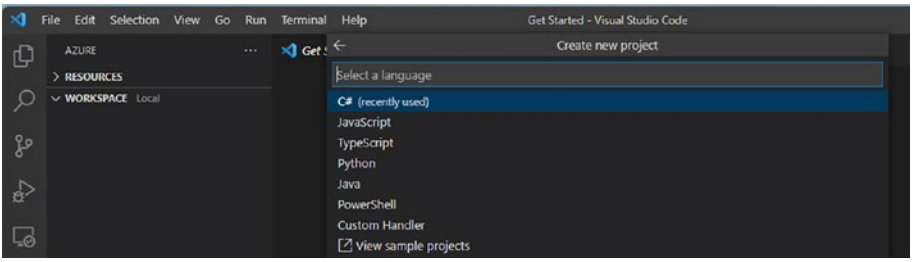

Figure 3-3. *Selecting the language*

Select the .NET 6.0 Isolated LTS runtime version, as shown in Figure 3-4. (This programming model was discussed in Chapter 2.)

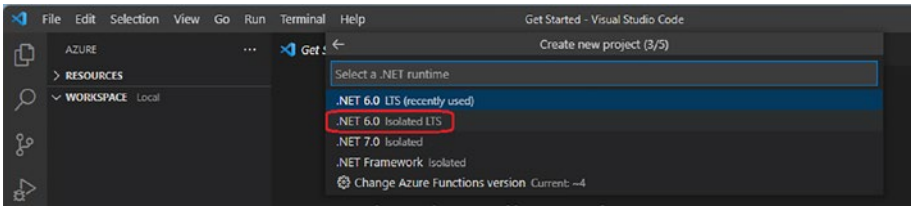

Figure 3-4. *Selecting the .NET runtime*

Select "Azure Blob Storage trigger" as the project template, as shown in Figure 3-5.

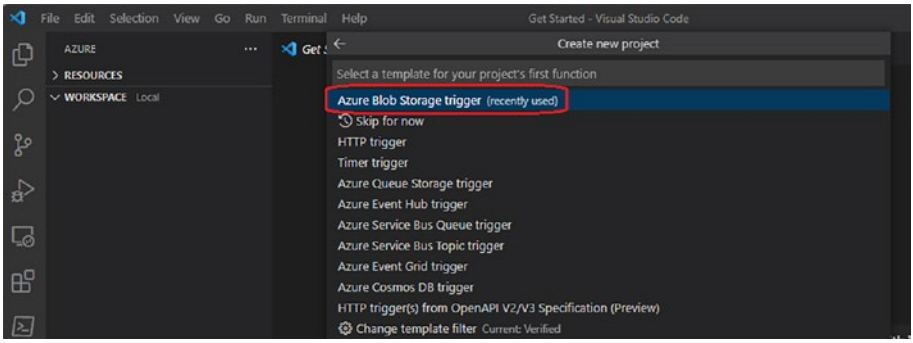

Figure 3-5. *Selecting the template for the function trigger*

Provide a name for your function app, as shown in Figure 3-6.

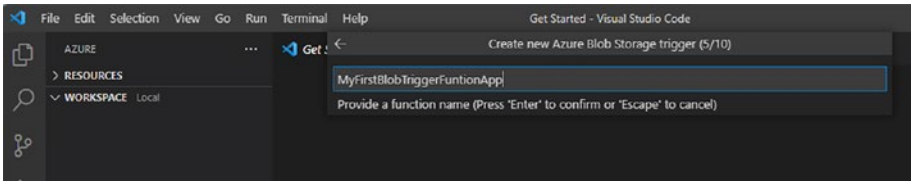

Figure 3-6. *Naming the function*

Provide the namespace. By default, it is `Company.Function`. For this demo, set it as `AzureFunctionV2Book.Function`, as shown in Figure 3-7.

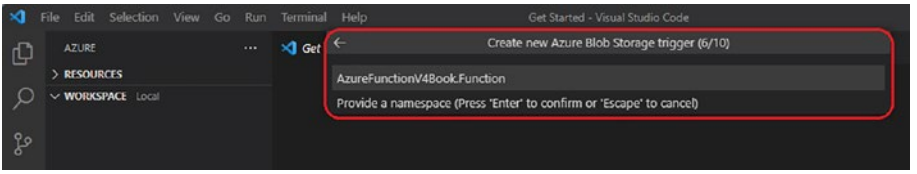

Figure 3-7. *Providing the namespace*

Click "Create new local app setting," as shown in Figure 3-8.

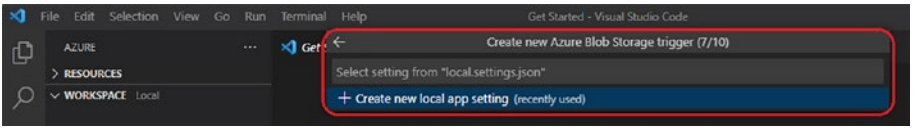

Figure 3-8. *Creating a new local app setting*

The next screen asks you to sign in to Azure. Select "Sign in to Azure" if you are not signed in. All the subscriptions associated with your Azure account load. Select the subscription, as shown in Figure 3-9. The subscription name in this example is VSEng.

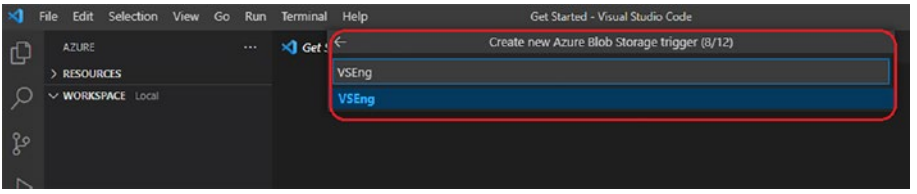

Figure 3-9. *Selecting the subscription*

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

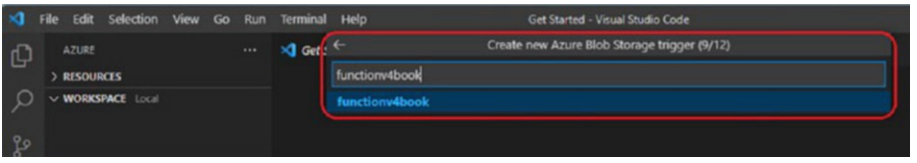

Figure 3-10. *Selecting an existing storage*

Once the storage account is set up, provide a name for the blob trigger. This is the path that the trigger monitors. By default, it shows as `samples-workitems`. For this function, you are setting it to `function-v4-book`, as shown in Figure 3-11.

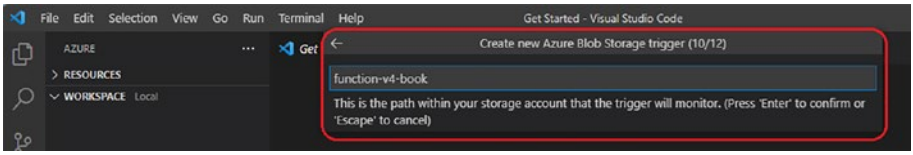

Figure 3-11. *Setting it to function-v4-book*

If you get a prompt, as shown in Figure 3-12, select "Use local emulator" and continue.

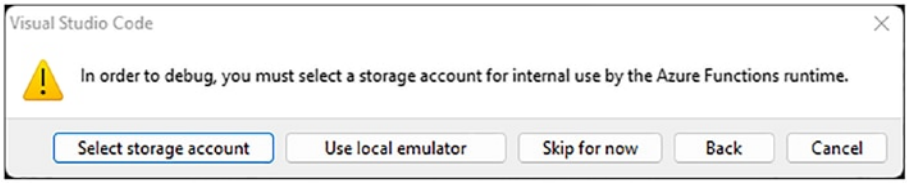

Figure 3-12. *Selecting storage account for Azure Functions runtime*

Add your project to a workspace in Visual Studio Code for easy access, as shown in Figure 3-13.

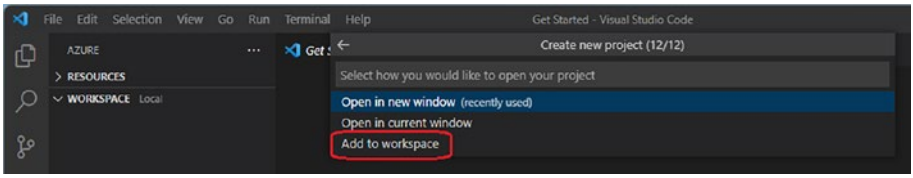

Figure 3-13. *Adding the function to a workspace*

Your function is set up, as shown in Figure 3-14.

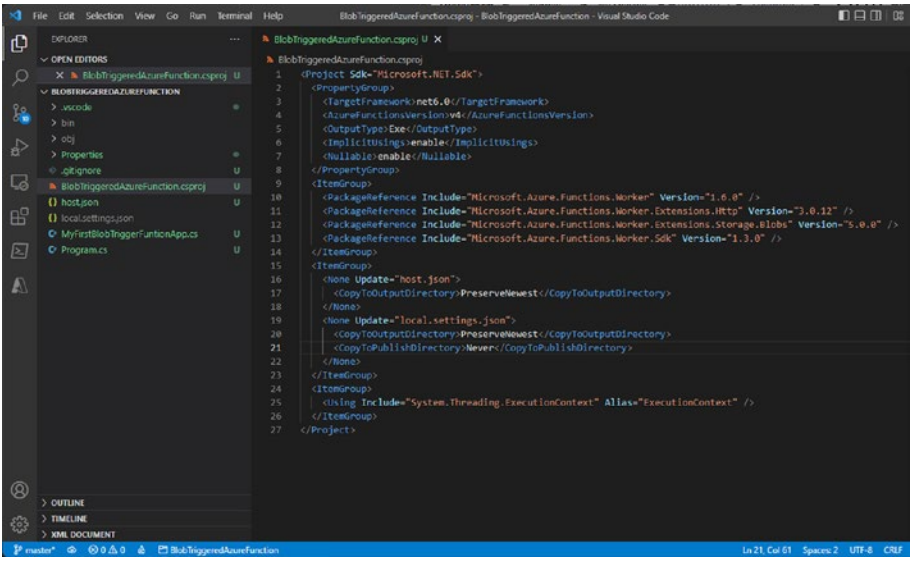

Figure 3-14. *Completed function*

Now you need to add a package named `SixLabors.ImageSharp` to the project. To do that, type **`dotnet add package SixLabors.ImageSharp`** in the terminal. Visual Studio Code installs the package.

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

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats;
using SixLabors.ImageSharp.Processing;

namespace AzureFunctionV4Book.Function
{
    public class MyFirstBlobTriggerFuntionApp
    {
        private readonly ILogger _logger;
```

```
public MyFirstBlobTriggerFuntionApp(ILoggerFactory
loggerFactory)
{
    _logger = loggerFactory.CreateLogger<MyFirstBlobTri
    ggerFuntionApp>();
}

[Function("MyFirstBlobTriggerFuntionApp")]
[BlobOutput("output-blob/{name}_resized", Connection
= "functionv4book_STORAGE")] // Output Container and
connection string of the storage account
public byte[] Run([BlobTrigger("image-blob/{name}",
Connection = "functionv4book_STORAGE")] byte[]
inputImage, string name)
{
    _logger.LogInformation($"C# Blob trigger function
    processed blob\n Name:{name} \n Size: {inputImage.
    Length} Bytes");

    IImageFormat format;
    var width = 100;
    var height = 200;

    using (Image input = Image.Load(inputImage, out
    format))
    {
        var output = new MemoryStream();

        input.Mutate(x => x.Resize(width, height));
        input.Save(output, format);
        return output.ToArray();
    }
```

```
        }
    }
}
```

In the top menu, click Debug ➤ 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, go to https://docs.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-portal to create containers.

Once the container is created, upload the blob as shown in the previous link in the `image-blob` container. You see the function being triggered. Once the function runs, the resized image appears in `output-blob`, as shown in Figure 3-15a and Figure 3-15b.

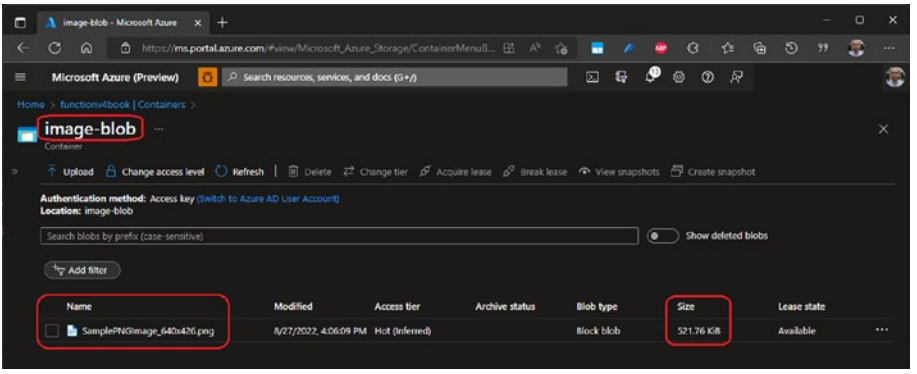

Figure 3-15a. *File size before resizing the image*

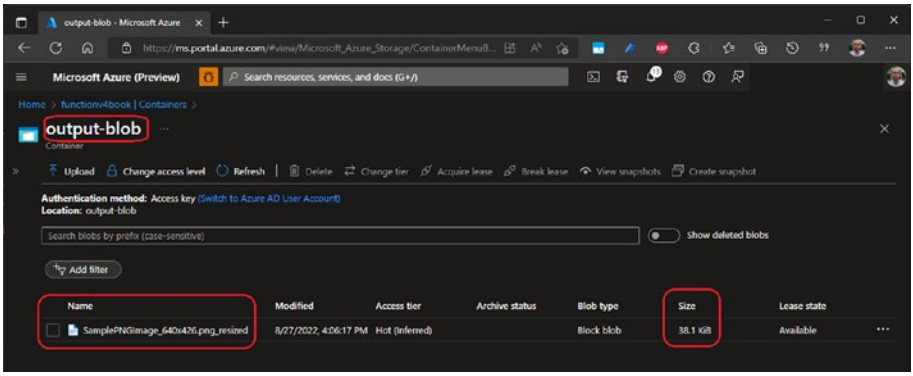

Figure 3-15b. *File sizes after resizing the image*

You have created a blob-triggered function using C#.

Next, let's look at creating a blob-triggered function using Node.js.

Blob-Triggered Function Using Node.js

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

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#.

Select JavaScript as the language, as shown in Figure 3-16.

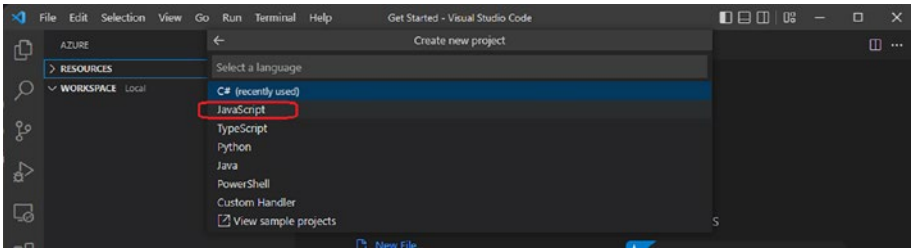

Figure 3-16. *Selecting the language*

Select the "Azure Blob Storage trigger" template, as shown in Figure 3-17.

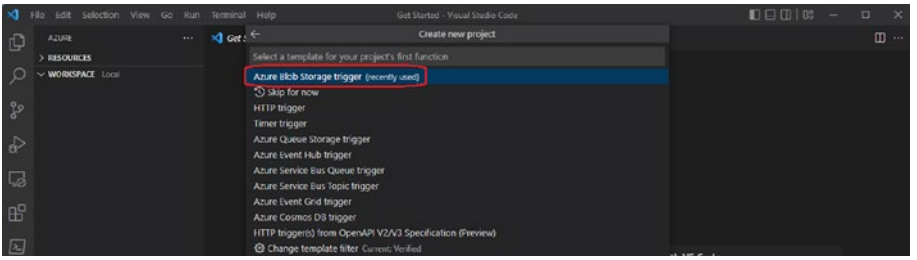

Figure 3-17. *Selecting the template*

Provide the function name. By default, it is `BlobTrigger`. For this function, set the function name to `BlobTriggerJs`, as shown in Figure 3-18.

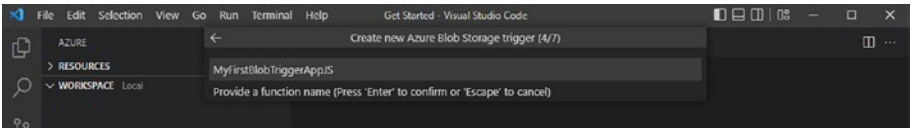

Figure 3-18. *Naming the function*

Click "Create new local app setting," as shown in Figure 3-19.

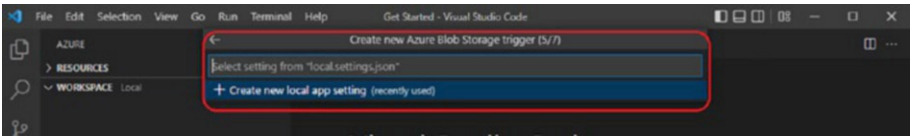

Figure 3-19. *Creating a new local app setting*

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, and select the Azure storage account with which you want this function to connect, as shown in Figure 3-20.

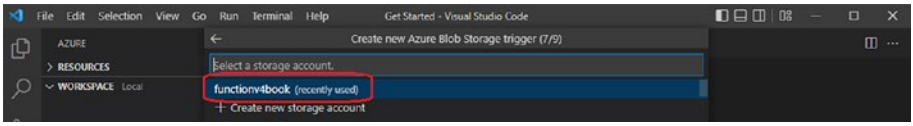

Figure 3-20. *Selecting the Azure Storage account*

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

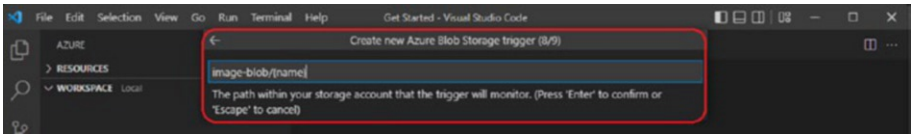

Figure 3-21. *Naming your blob*

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

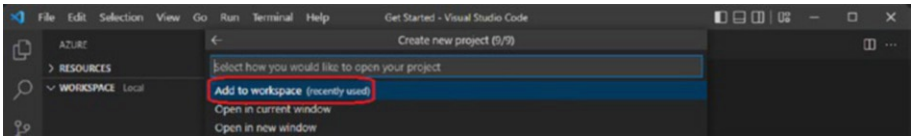

Figure 3-22. *Adding to the workspace*

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

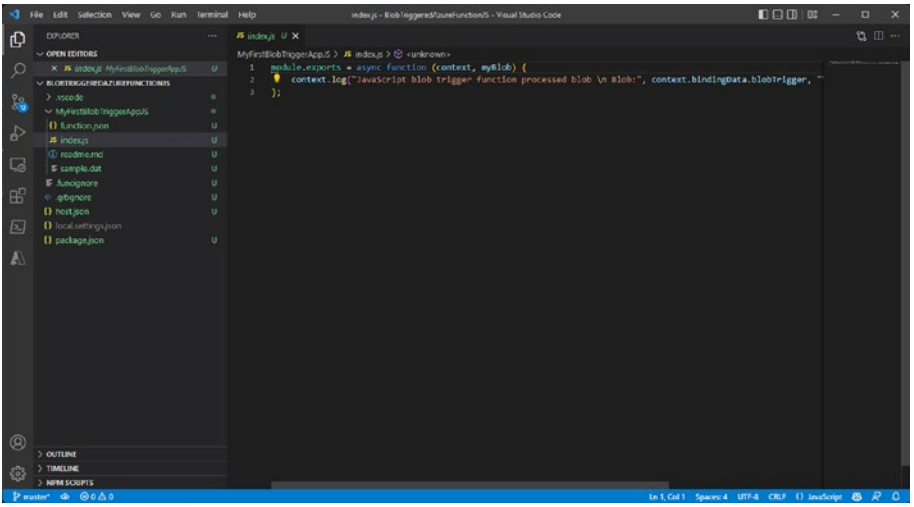

Figure 3-23. *Function files*

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

Once the packages are installed, copy and paste the following code.

```
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('image-
            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.createBlockBlobFromStrea
                    m(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.generateSharedAccessSignature(
    container, blobName, sharedAccessPolicy);
    return {
        token: sasToken,
        uri: blobService.getUrl(container, blobName,
        sasToken, true)
    };
}
```

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 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"
}
```

Summary

You should now understand the concept of Azure triggers and bindings, and you have created a blob-triggered function. The next chapter looks at creating serverless APIs using Azure Functions.

CHAPTER 4

Serverless APIs Using Azure Functions

Before you start creating APIs with Azure Functions, you must understand where Azure Functions as a serverless API fit into the current system architecture you plan to build for 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 each business scenario was considered a mammoth task, but the monolithic approach became less desirable with the advent of cloud computing and agile methodologies. Developers started looking at microservice architecture because cloud platforms such as Microsoft Azure, AWS, and GCP made building and managing microservices easy.

In this chapter, we 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

Next, let's look at monolithic and microservice architectures and determine which architecture to use in specific circumstances and where Azure Functions fits in.

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_4

Monolithic Architecture vs. Microservice Architecture

The *monolithic* approach used to be one of the most popular approaches to building applications. 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 compared to the agile development model. Monolithic applications are more vulnerable to bugs and deployment issues. For example, if there is a bug in the client-side code, you still deploy all the server-side code after fixing the bug since everything resides in one codebase. This means frequent downtimes for the application and expensive hardware infrastructure required to ensure high availability.

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

A *microservice architecture* breaks a complex monolithic application into small and independent applications. With a microservice architecture, deploying and scaling individual applications becomes easier and less expensive and makes DevOps less time-consuming. If you further break down microservices, they are called *nano services*.

The following lists the benefits of microservices.

- A small and independent codebase for each component is easy to maintain.
- Onboarding new developers becomes easy.
- Each service can be scaled individually in the cloud based on its consumption.

- Multiple teams can work in parallel on different microservices.
- Microservices can be written based on the programming language that 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 application.
- Communication within the APIs can become slow and difficult to troubleshoot if not developed correctly.
- If DevOps is not properly set up, deployment can become messy and create many issues (but if done properly, it becomes easy to maintain). An 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 becoming your biggest blocker.

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

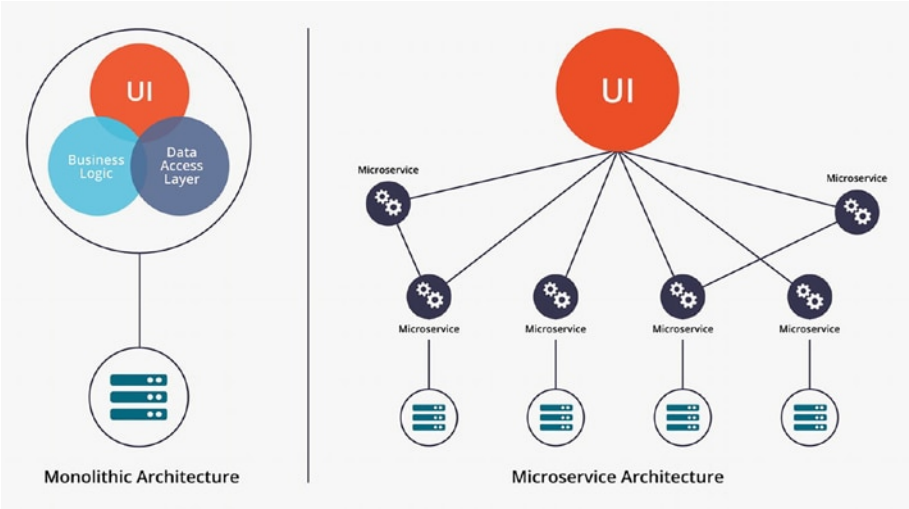

Figure 4-1. *Monolithic architecture vs. microservice architecture*

Converting Monolithic Applications to Highly Scalable APIs Using Azure Functions

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

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

- Customer service, which includes 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 your front-end user interface can easily consume.

Now, let's see how microservices help you scale the application with a minimum cost. Let's say you have a holiday season sale coming up, and you estimate that you have double the amount of traffic on the website during this time.

After some analysis, you find that the product/service and the payment and checkout services 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, payment, and checkout) and leave the other services as is. In contrast, if you had a monolithic application, you would have to scale out the complete application, which would greatly increase your operating costs.

Since now you know why you want to convert a monolithic application to microservices, let's examine 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 as a separate function, and on days when 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 reduces the team's workload and helps them concentrate on the business.

Azure Functions provides a free monthly grant of 400,000 GB/s (a unit of resource consumption) of execution time and one million executions, 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 shown in Figure 4-2, you can create separate function(s) for each of the microservices and expose them as REST APIs.

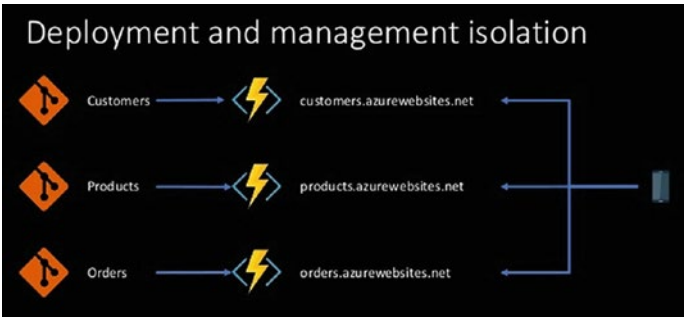

Figure 4-2. *Isolated functions*

To expose functions as REST APIs, you must create HTTP-triggered functions so that you can use them. HTTP-triggered functions work like any other API where you call an endpoint, and it returns the results. 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.

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

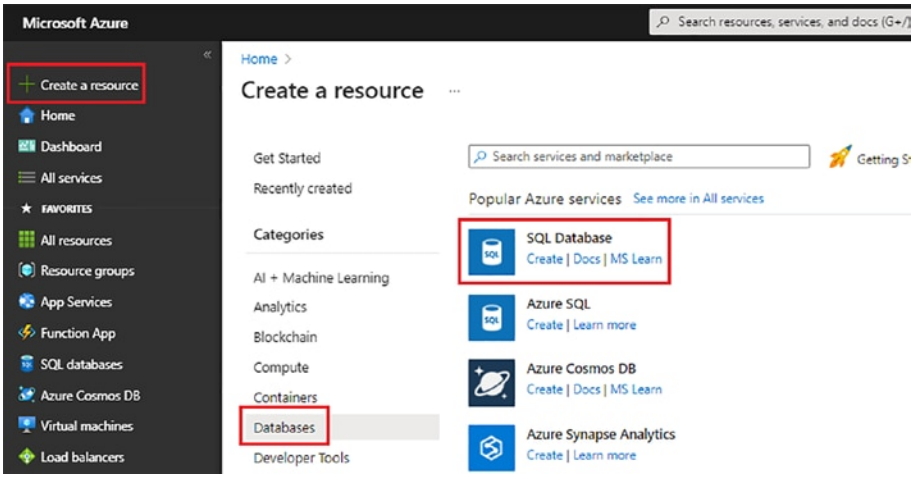

Figure 4-3. *Selecting SQL Database*

Provide a name for the database. Select a server or create one if it does not exist. Configure the other settings as shown in Figure 4-4a and Figure 4-4b.

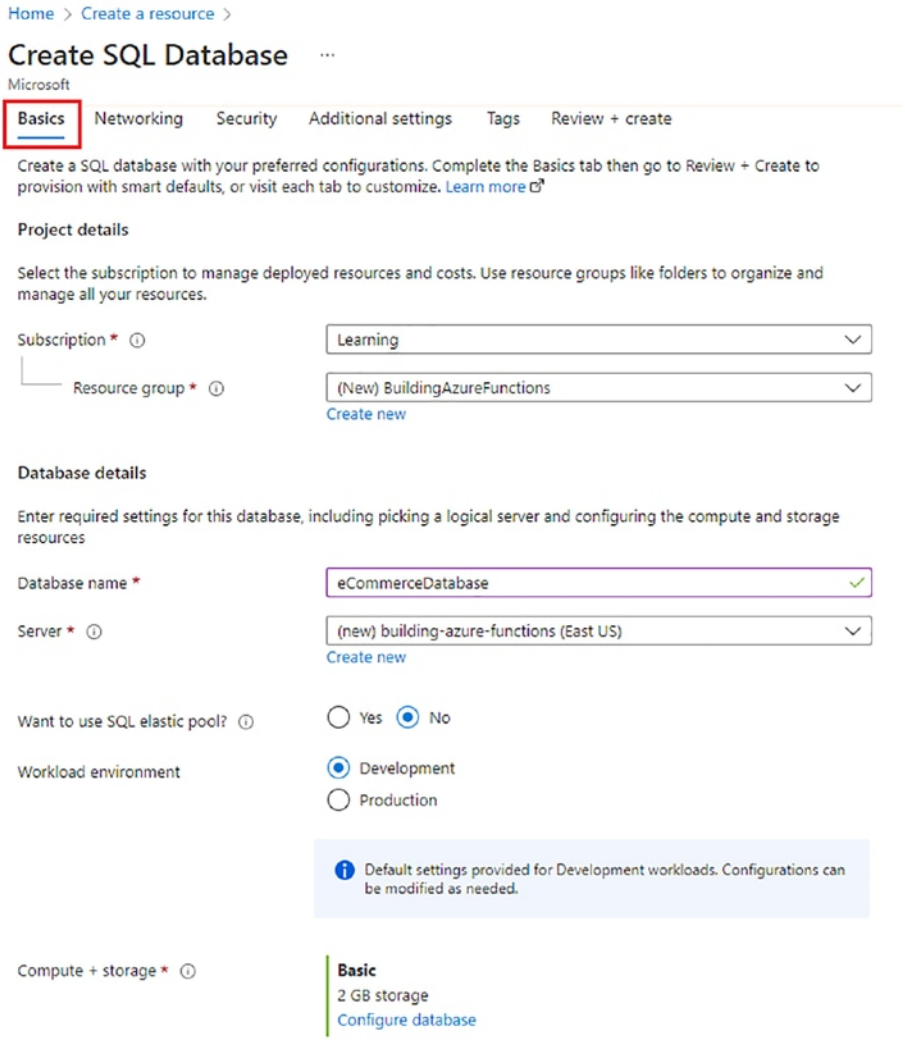

Figure 4-4a. *Creating the database*

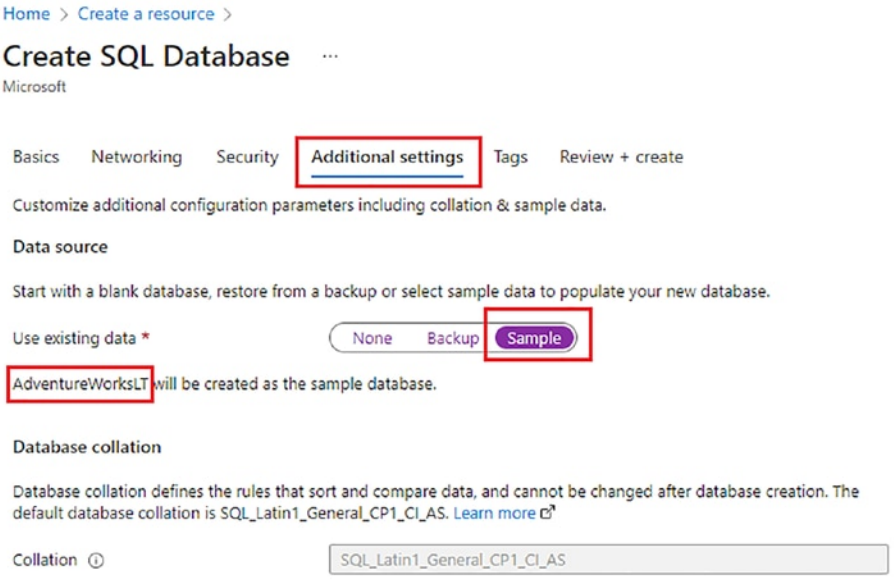

Figure 4-4b. *Creating the database from the sample*

For the data source, you could choose None and create the tables from scratch. In this case, let's use the sample database AdventureWorksLT as shown in Figure 4-4b, because it has tables pre-populated with sample data.

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

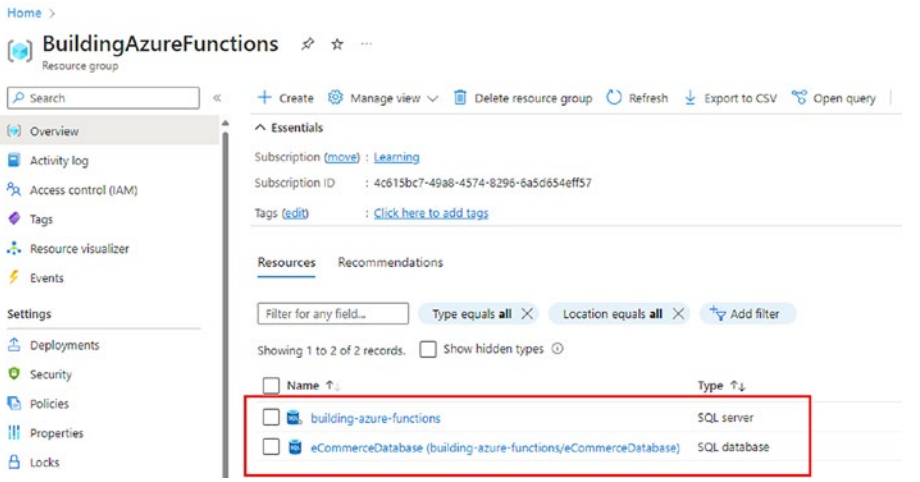

Figure 4-5. *Database created*

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

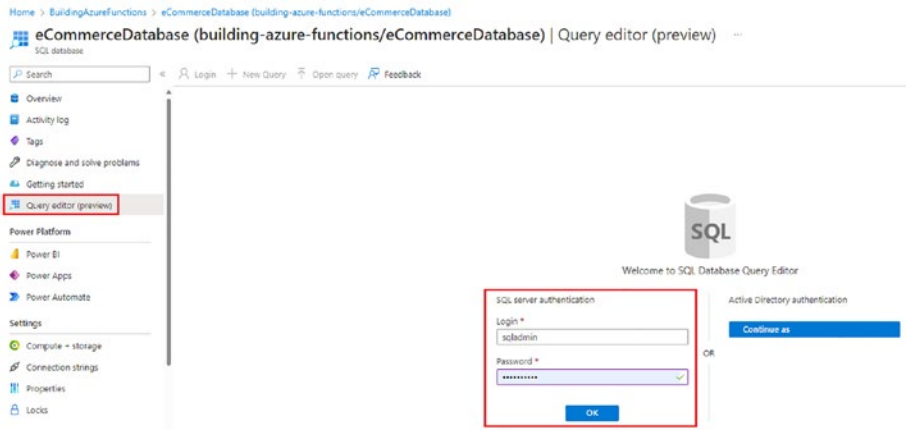

Figure 4-6. *Connecting to the database*

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

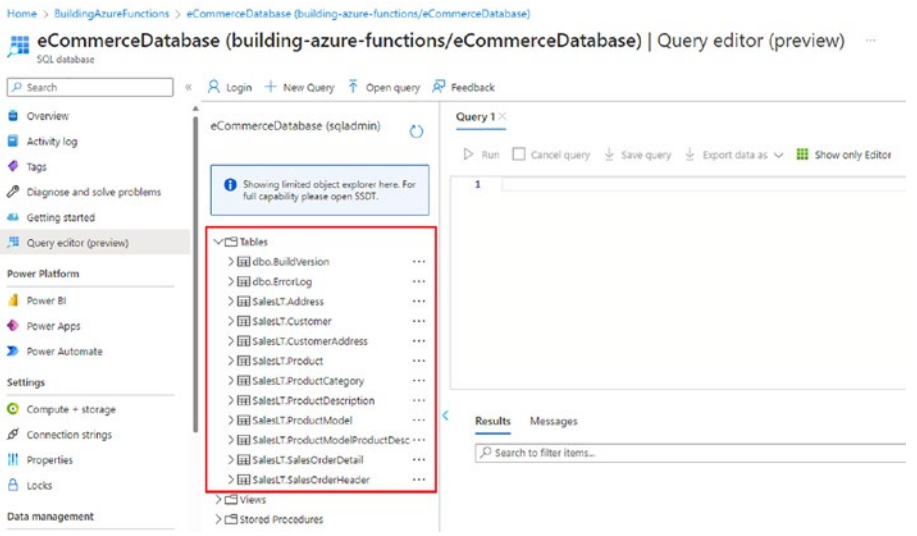

Figure 4-7. *Tables being created*

You can query the tables similar to the way it is done 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.

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

Open Visual Studio Code, click the Azure icon, and Create a New Function as shown in Figure 4-8. Create a folder named `HTTP-Triggered-Function` to save your code when prompted.

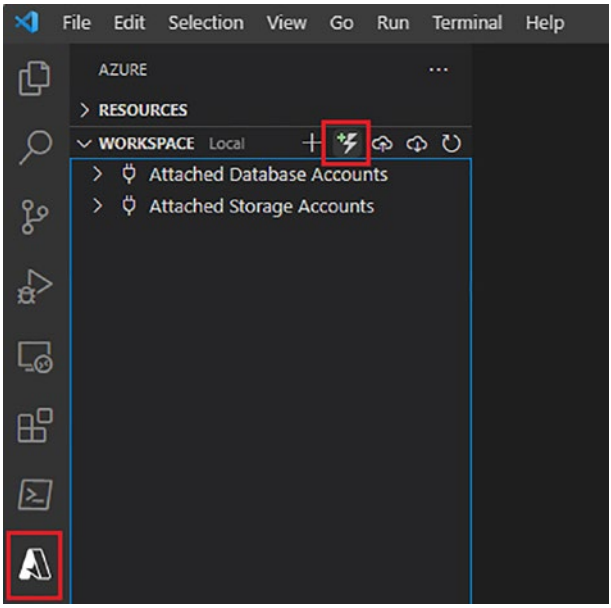

Figure 4-8. *Creating the function*

Select the language to be used for the Azure function. C# is used for this example, as shown in Figure 4-9.

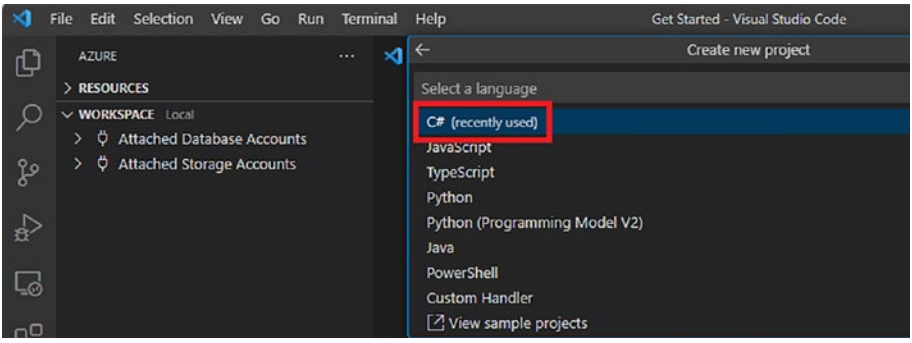

Figure 4-9. *Selecting the language*

Select the .NET 6.0 Isolated LTS runtime version, as shown in Figure 4-10.

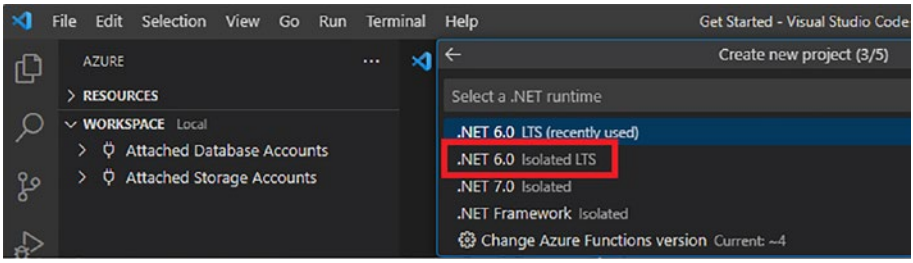

Figure 4-10. *Selecting the .Net version*

Select the "HTTP trigger" template for the function, as shown in Figure 4-11.

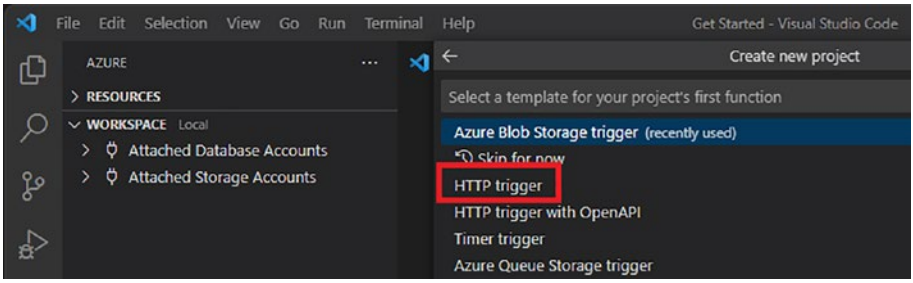

Figure 4-11. *Selecting the HTTP trigger template*

Provide a name for the function, as shown in Figure 4-12.

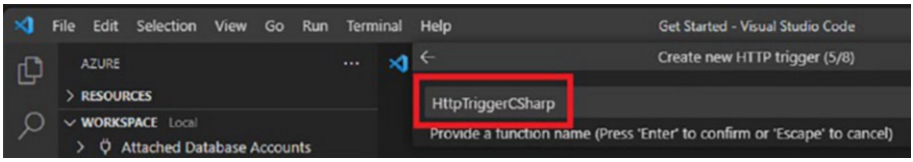

Figure 4-12. *Naming the function*

Provide the function's namespace. By default, it is `Company.Function`, but for this function, set it to `AzureFunctionBook.Function`, as shown in Figure 4-13.

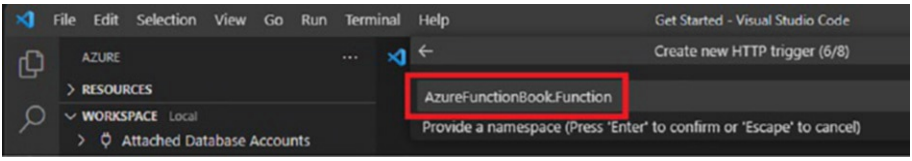

Figure 4-13. *Providing the namespace*

Set the access rights for this function. You see three options: Anonymous, Function, and Admin. The access rights determine which keys are required to invoke the function.

- **Anonymous** means no API key is required.
- **Function** is the default setting if nothing is selected, which means a function-specific API key is required.
- **Admin** means a master key is required.

Use Anonymous as the access right for this function, as shown in Figure 4-14.

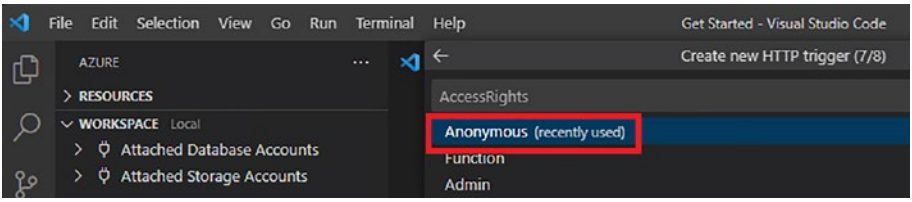

Figure 4-14. *Setting the access rights*

Let's add the `SqlClient` package to the solution. To do this, go to the terminal and select New Terminal in the top menu. Type **`dotnet add package System.Data.SqlClient --version 4.5.1`** and press Enter. This installs the package required for your solution. To verify, go to the `.csproj` file. The package was added, as shown in Figure 4-15.

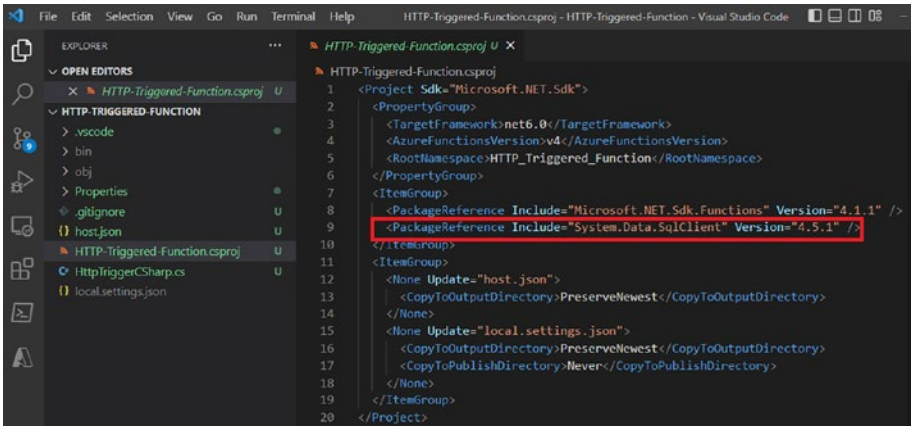

Figure 4-15. *Package added*

Now everything is set up. You follow a code structure similar to what you would follow in normal projects. Create two folders: `Helper` and `Models`. In `Models`, create the `CustomerModel.cs` file, and in `Helper`, create the `SqlClientHelper.cs` file. You use them for your function, as shown in Figure 4-16.

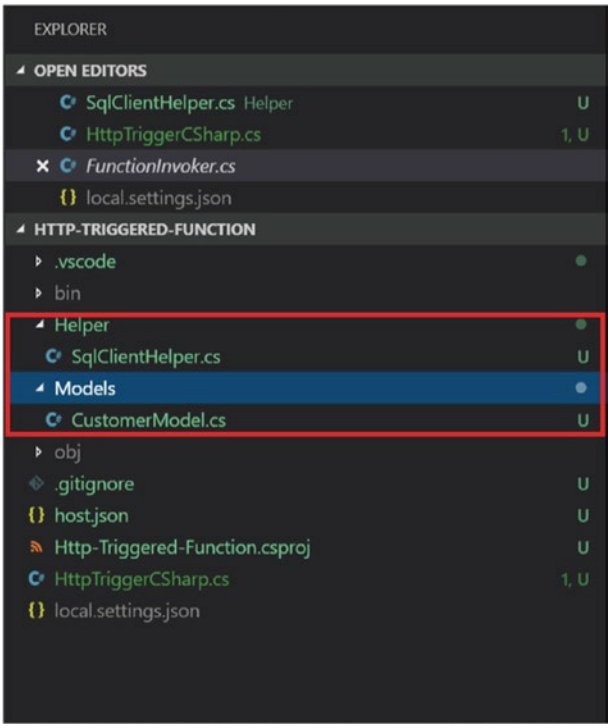

Figure 4-16. *Folders created*

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; }
    }
}
```

Click the `SqlClientHelper.cs` file and paste the following code, which calls the SQL Server database and gets 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.GetEnvironmentVariable
            ("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;
        }
    }
}
```

Go to the main `HttpTriggerCSharp.cs` file and paste the following code, which is the function that is triggered when you call it. It first tries 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, you'll see you have created a basic customer profile function where you get customer details based on `CustomerID`.

Select Debug in the top menu and click Start Debugging. Once the function is compiled, you see the local URL of the function, as shown in Figure 4-17.

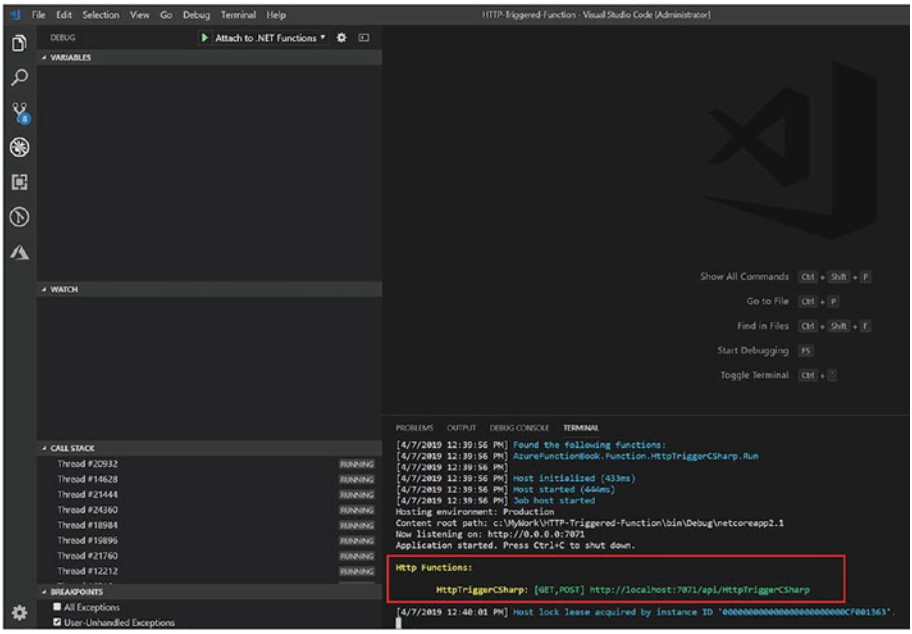

Figure 4-17. *Local URL*

Copy this URL, append `?customerId=1` to it, and hit Enter. You should see the output shown in Figure 4-18.

Figure 4-18. *Output*

Now you should understand how to create an HTTP-triggered API using Azure Functions. In the next section, you 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. It is short 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 the steps in the previous section. The only change here is that instead of C#, JavaScript is the language for the function.

Once the function is created, install the following `npm` packages.

1. `Azure-odata-sql`
2. `Async`
3. `Tedious`
4. `Tedious-connection-pool`

To install the packages, open the terminal and type the following, which installs the package for you.

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

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

Go to the main `index.js` file and paste the following code. In the following code, you are first configuring the table and schema 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();
        });
}
```

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

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

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

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

Once you run the previous query, you get the result shown in Figure 4-19.

Figure 4-19. *Query output*

You have created two HTTP-triggered functions: a normal HTTP-triggered function with C#, and an advanced HTTP-triggered function using OData with Node.js.

Summary

In this chapter, you learned the differences between monolithic and microservices architectures. You also created serverless APIs using Azure Functions. The next chapter looks at the Durable Functions extension and how you can use durable functions for long-running tasks.

CHAPTER 5

Azure Durable Functions

Over the course of this chapter, we will cover the following durable function topics.

- Overview
- Bindings
- Performance and scale
- Creating through the Azure portal
- Disaster recovery and geodistribution

Overview of Durable Functions

Durable Functions is an extension to Azure Functions that helps you build reliable, stateful apps in a serverless environment and define workflows in your code. The Azure Functions platform manages checkpoints, states, and restarts for you.

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

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

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_5

Types of Functions

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

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

There are three types of functions.

- A **client function** is the entry point for creating an instance of a durable function. They are triggered functions that create a new instance of an orchestration process. Any available trigger in Azure Functions can trigger client functions. Also, client functions have an orchestration binding that allows them to manage durable orchestrations.
- An **orchestrator function** describes 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.
- An **activity function** is the basic unit of work in durable function 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 is an activity function, and the output of one function can be used as the input of another. An activity trigger must trigger an activity function.

Application Patterns

The Durable Functions extension 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 sequential order. Also, you use function chaining when the output of one function must 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 process can be done using function chaining, as shown in Figure 5-2.

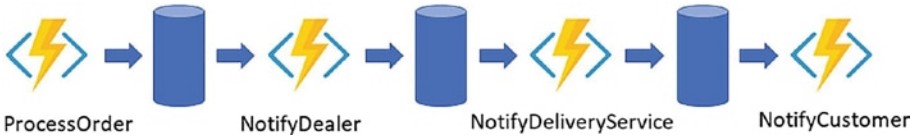

Figure 5-2. *An example of function chaining*

The following code calls the function chaining pattern using C#.

```
public static async Task<object>
Run(DurableOrchestrationContext context)
{
try
{
    var orderProcessedResult = await context.CallActivityAsync
    <object>("ProcessOrder");

    var dealerNotificationResult = await context.CallActivity
    Async<object>("NotifyDealer", orderProcessedResult);

    var deliveryServiceResult = await context.Call
    ActivityAsync<object>("NotifyDeliveryService",
    dealerNotificationResult);

    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 them to execute. Usually, 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 is complicated because you must track when the message is picked up and processed and store the result. This is a difficult task 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.

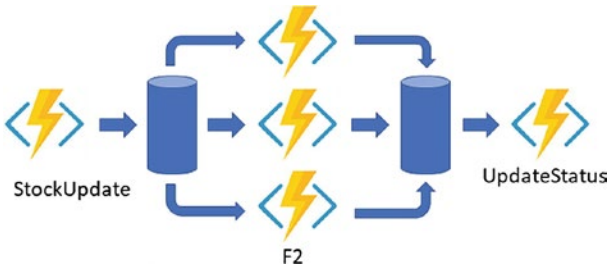

Figure 5-3. *Fan-Out/Fan-In example*

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 sends an email, another sends an SMS message to the customer, and one stacks 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 updates the notification status corresponding to each user who opted for notification, as shown in the following code.

```
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 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 polling the long-running job's state.

The Durable Functions extension provides a built-in capability that simplifies the code you 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 order's status. The first state is whether the restaurant accepts the order. Once the order is accepted, it starts showing you the time it takes for the order to be prepared by the restaurant. 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.

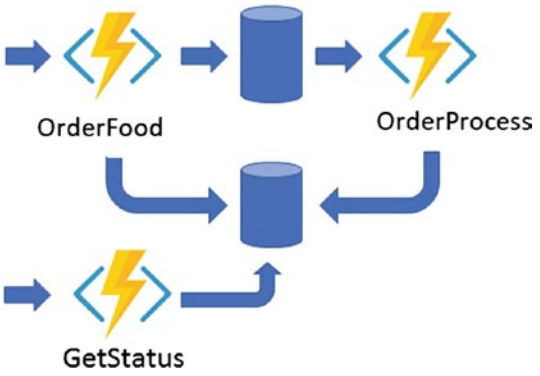

Figure 5-4. *Async example*

In Figure 5-4, the `OrderFood` function act as an HTTP API that is called once the end user clicks Order Food. The `OrderFood` function checks for the validity of the order and calls the `OrderProcess` function. This function keeps on updating the status of the order.

You redirect the end user to the order-tracking page, which polls the `GetStatus` function and shows 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("OrderProcess",
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. A regular *timer trigger* (which 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.

On the other hand, the Durable Functions runtime 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).

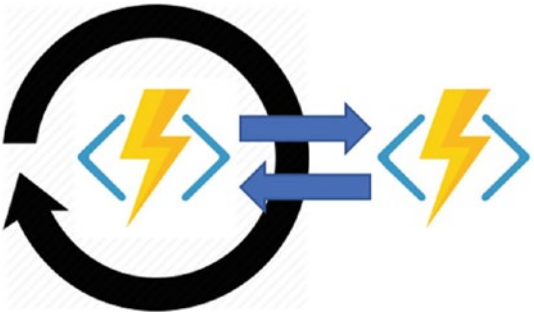

Figure 5-5. *Monitoring example*

Human Interaction

Usually, you 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 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 is 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. *An example of a human interaction pattern*

The following 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.WaitForExternalEvent
    <bool>("ApproveLeaveEvent");
    if (approveLeaveEvent == await Task.WhenAny
    (approveLeaveEvent, durableTimeout))
    {
        timeoutCts.Cancel();
        await context.CallActivityAsync("ProcessLeaveApproval",
        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 functions. The Durable Functions extension also introduces one output binding that triggers 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 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 following JSON object in the bindings array defines the activity trigger.

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

The following describes the trigger behavior.

- **Threading** is an activity trigger like any other function in Azure Functions that you code and has no limitation on threading or I/O.
- **Message visibility** is 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** are JSON serialized and 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"
}
```

The following describes the trigger behavior.

- **Single threading**: A single dispatcher thread is used for all orchestrator functions running on a single host instance. For this reason, the orchestrator function code should not perform any I/O. Also, this thread should not do async work except when awaiting 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"));

    return outputs;
    // returns
    // "I am travelling to Hyderabad"
    // "I am travelling to New York"
    // "I am travelling to Delhi"
}
```

As you can see, the previous orchestrator function is calling the activity function `City_Travel` and passing the city's name to it. From how it is written, it looks like the orchestrator function is calling the City_Travel activity function directly, but it is actually 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 work-items queue, as shown in Figure 5-7.

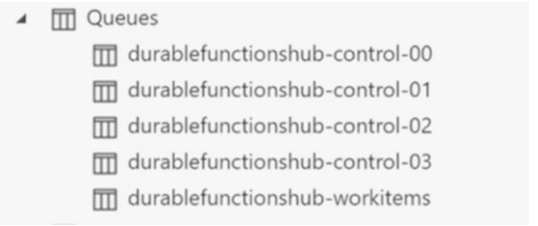

Figure 5-7. *Durable function queues*

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 allows you to write functions in Azure Functions that interact with orchestrator functions.

The following JSON object in the bindings array defines the orchestrator client trigger.

```
{
  "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 scale and improve performance. To understand the scaling behavior, you must first understand some of the underlying details of the Azure storage provider.

History Table

The history table contains the history of 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 the 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 historical events are then replayed in the orchestrator function to return to the previous checkpoint state. The event sourcing pattern influences this.

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 is influenced by the *command and query responsibility segregation* (CQRS) pattern.

Internal Queue Triggers

Activity and orchestrator functions are 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.

- The **control queue**: There are multiple queues in a 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.

- The **work-item queue**: Each task hub has one work-item queue. This queue behaves like a normal queue. This queue triggers the stateless activity functions by dequeuing 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.

Now that you understand 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; by default, the partition count is four. The number of control queues is defined in the `host.json` file for a function running on the 2.0 runtime, as follows.

```
{
"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 has 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 advisable to create no more than two to five control queues.

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

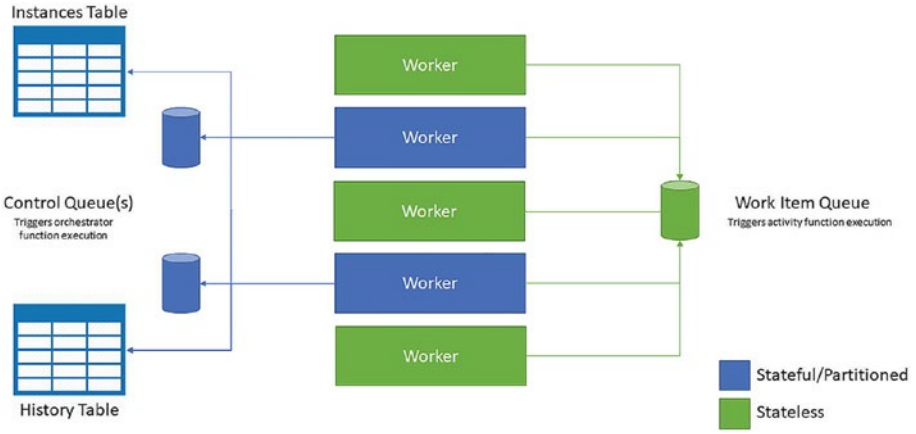

Figure 5-8. *Azure Functions' behavior when scaling out*

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 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, the scale controller keeps adding the instances until it reaches the partition count.

In work-item queues, the scale controller keeps 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 that high concurrency results in high per-VM memory usage.

Orchestrator and activity functions 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 messages 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 keep the idle session in the memory, you use the `extendedSessionIdleTimeoutInSeconds` setting in `host.json`, as follows.

```
{
    "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 also a downside.

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

Performance Targets

If you plan to use durable functions in a production application, consider the performance requirements early in the process because they 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 the Azure Portal

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

Creating a Durable Function

Open the Azure portal and click "Create a resource." Next, select Compute ➤ Function App, as shown in Figure 5-9.

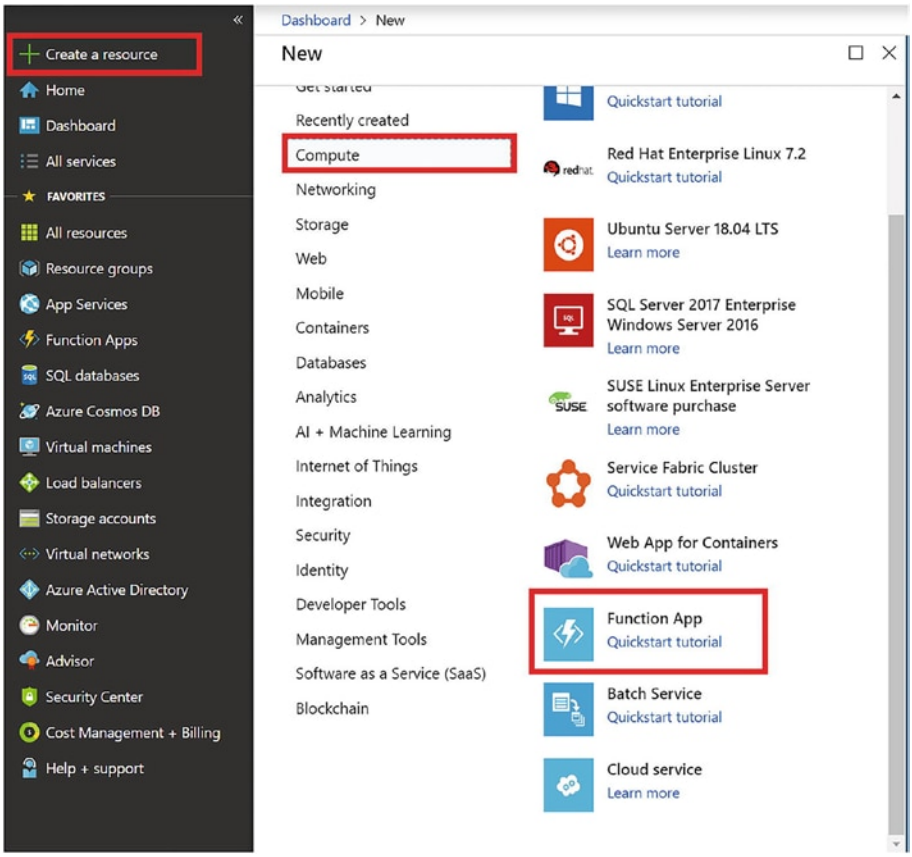

Figure 5-9. *Starting a durable function*

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

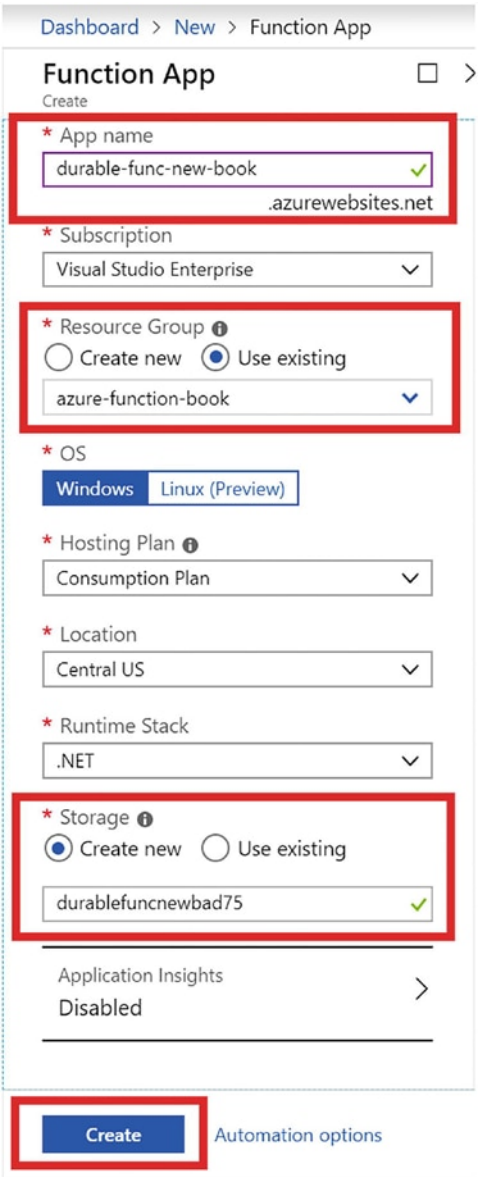

Figure 5-10. *App details*

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

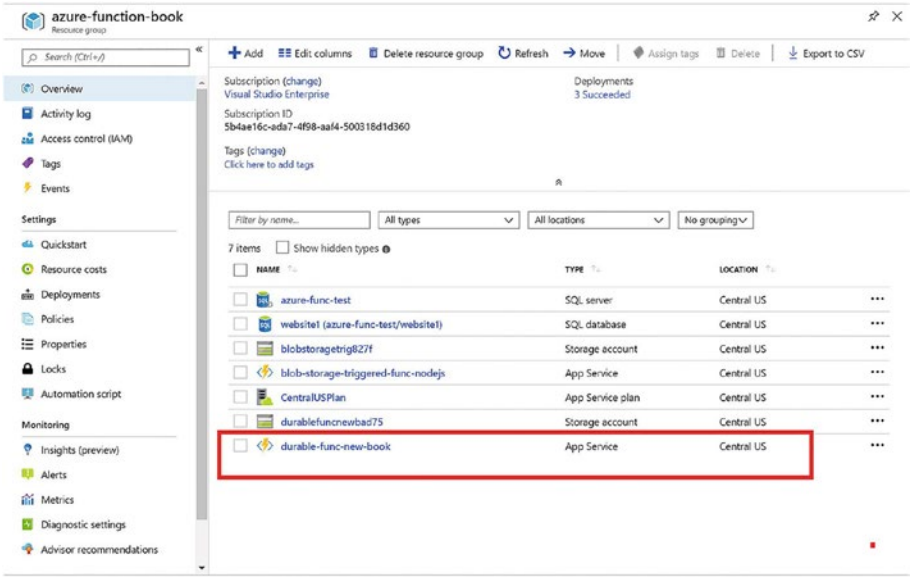

Figure 5-11. *Selecting the function*

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

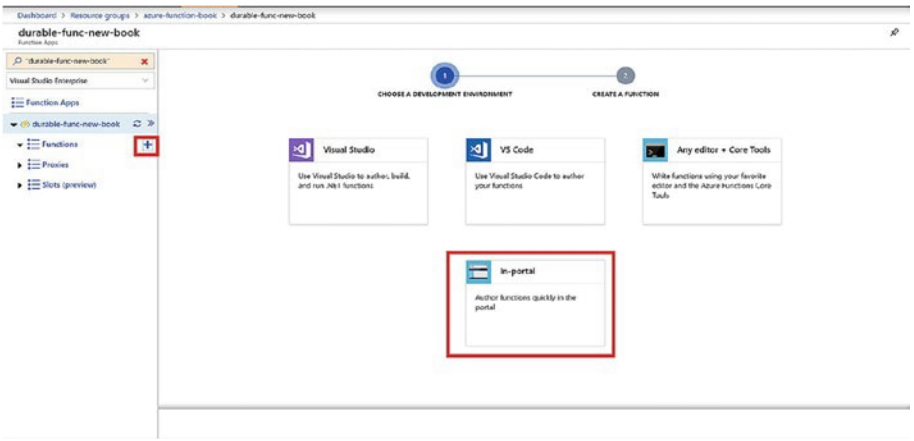

Figure 5-12. *Select the environment and continue*

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

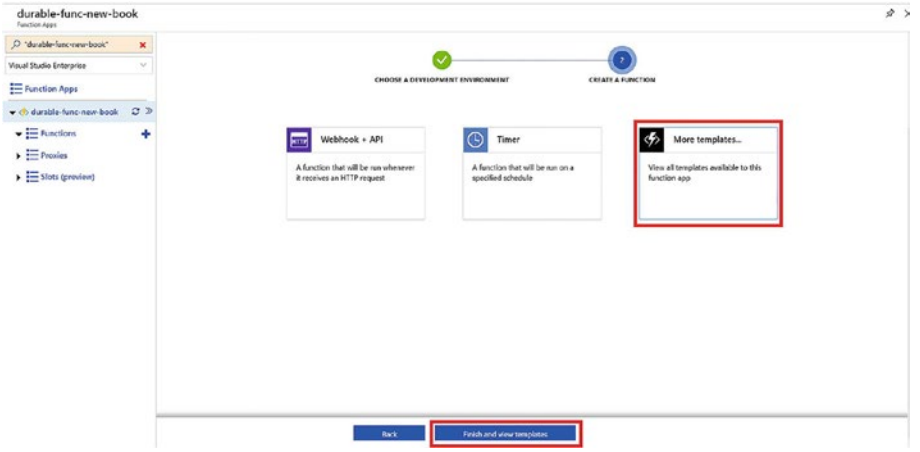

Figure 5-13. *Choosing more templates*

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

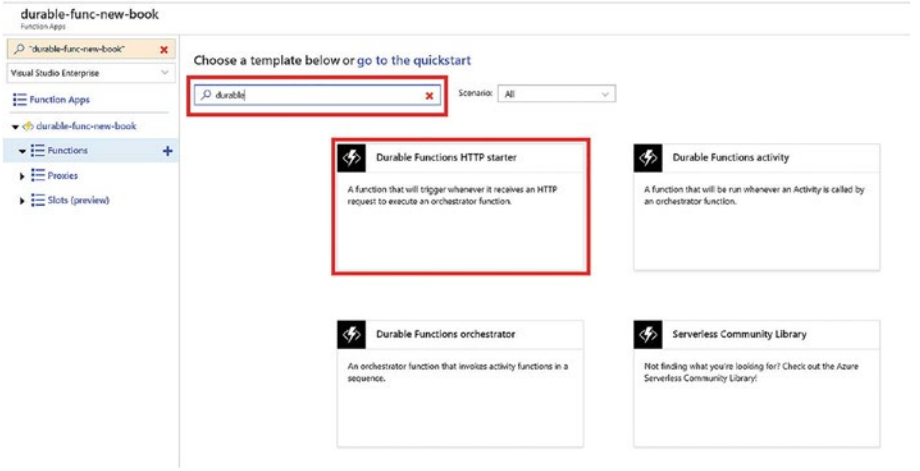

Figure 5-14. *Selecting the starter template*

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

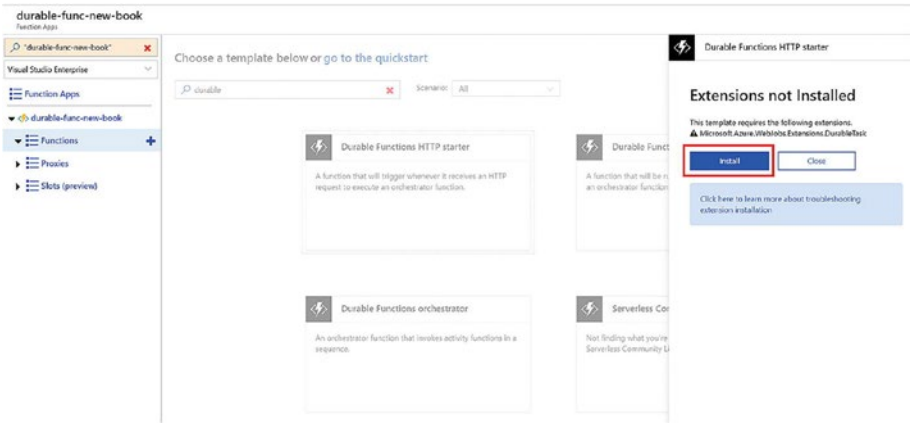

Figure 5-15. *Starting the installation*

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] DurableOrchestrationClient
            starter, ILogger log)
```

```
{
    string instanceId = await starter.StartNewAsync
    ("Orchestrator_City", null);
    log.LogInformation($"Running orchestration with
    ID = '{instanceId}'.");
    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

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

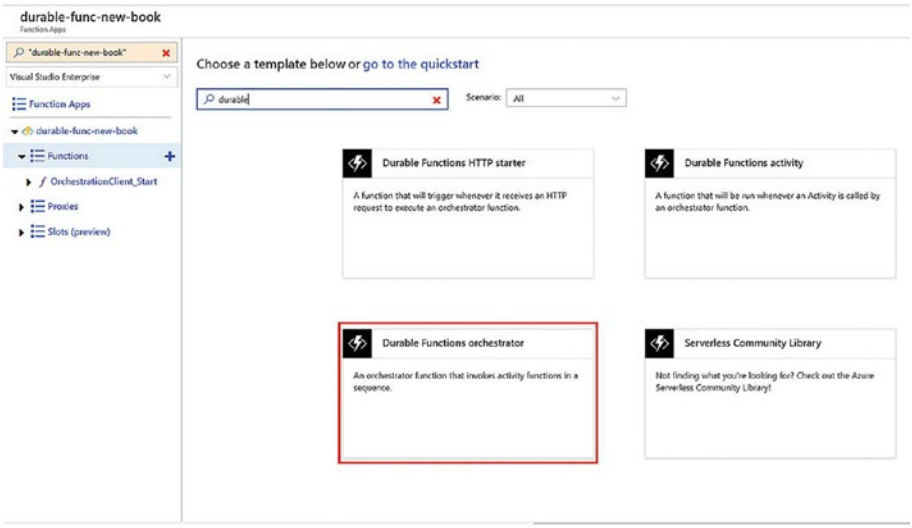

Figure 5-16. *Selecting the orchestrator*

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

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
using System.Collections.Generic;
using System.Threading.Tasks;

[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"));

    return outputs;
    // returns
    // "I am travelling to Hyderabad"
    // "I am travelling to New York"
    // "I am travelling to Delhi"
}
```

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

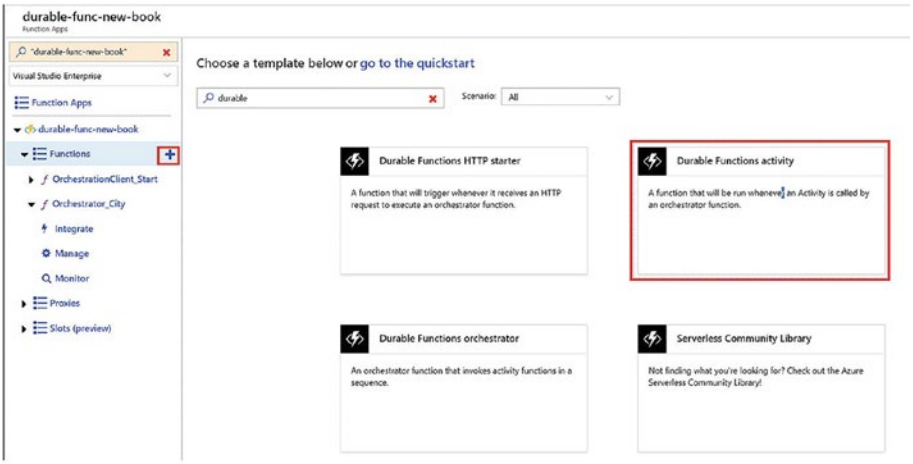

Figure 5-17. *Selecting the activity*

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}!";
}
```

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

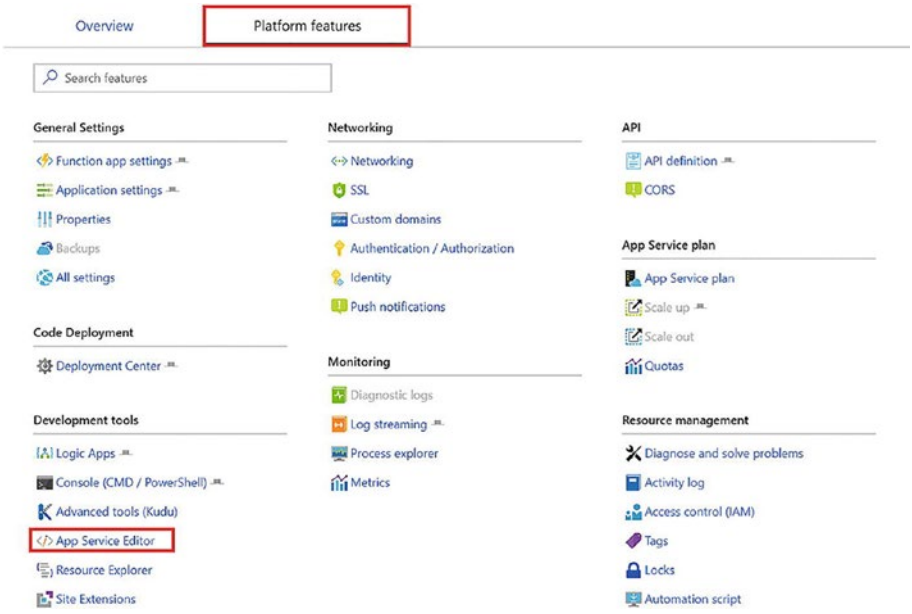

Figure 5-18. *Selecting App Service Editor*

The App Service Editor opens in a new tab. Now, select the `host.json` file and copy and paste the following code into it.

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

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

Figure 5-19. *The code*

Next, return to the function app and go to the `OrchestrationClient_Start` function. Click Get Function Url and copy the URL. The URL is in the format of `https://<function-http-instance>/api/orchestrators/{functionName}?code=#code`. Replace `{functionName}` with the name

of the orchestration HTTP trigger, which is `OrchestrationClient_Start`. Paste the URL in the browser and press Enter to execute a new orchestrator function instance.

Disaster Recovery and Geodistribution of Durable Functions

Since you have deployed your Azure durable function and want 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, make sure that the data to be stored to make this application run successfully is properly georeplicated.

To enable disaster recovery for durable functions, you should first ensure that your 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 ensure that your data is available in multiple regions. This helps you make your function available quickly in a disaster.

Summary

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 event of a disaster.

The next chapter looks at deploying functions to Azure using a CI/CD pipeline.

CHAPTER 6

Deploying Functions to Azure

This chapter covers the following topics.

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

This chapter walks 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 Continuous integration/continuous deployment (CI/CD) 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 on the same codebase, and changes to the code repository are frequent.

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_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. Once the continuous deployment is enabled, the access. to the function app is set to read-only in the 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 the function requires to run, as shown in Figure 6-1.

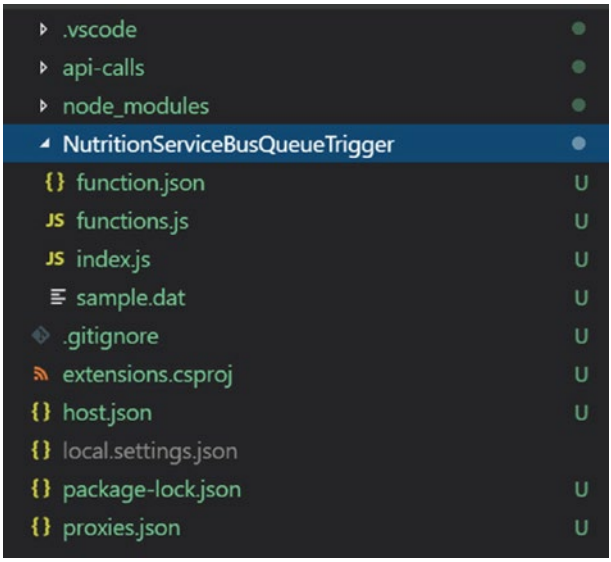

Figure 6-1. *Project organization*

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 to connect it to the Azure Functions service.

Set up your Azure DevOps account by following these steps.

Go to the Azure portal (`https://ms.portal.azure.com/`) and click DevOps. Select "Azure DevOps organizations," as shown in Figure 6-2a and Figure 6-2b.

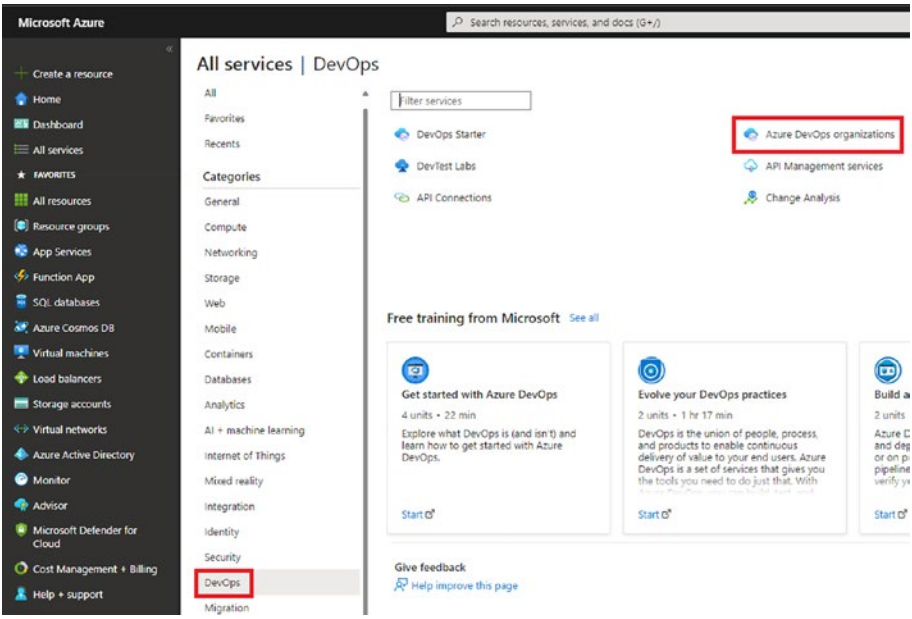

Figure 6-2a. *Finding Azure DevOps organizations resource*

If you do not already have an Azure DevOps organization, this page presents you with options to create one.

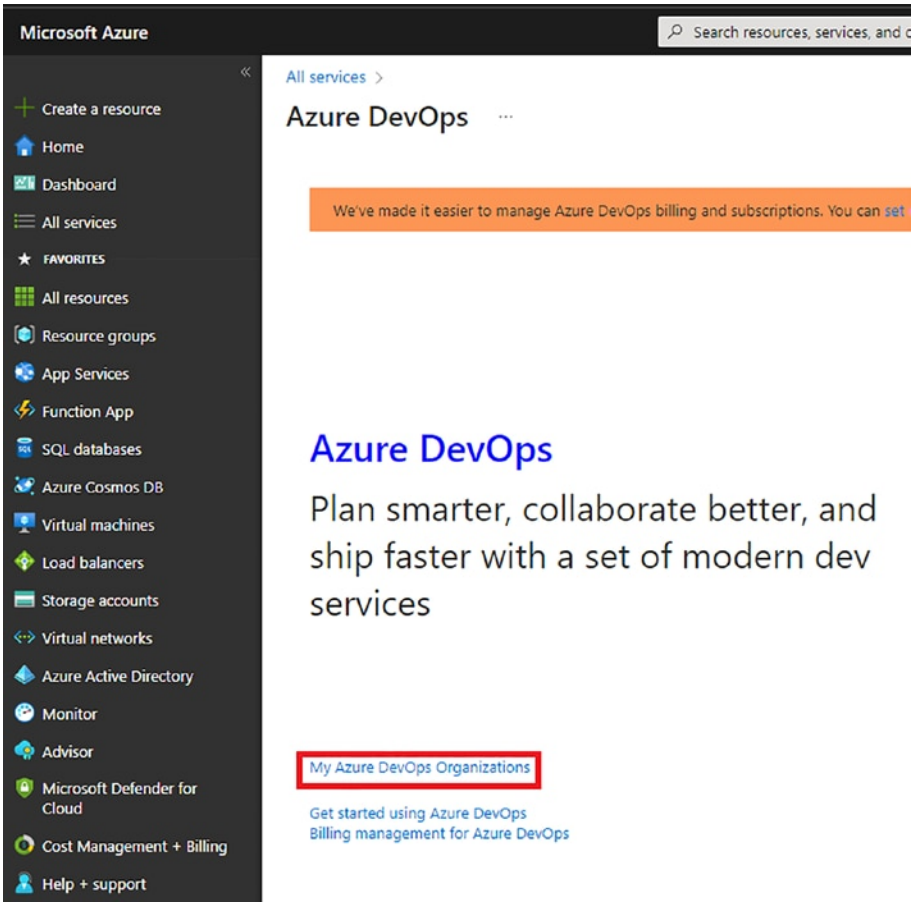

Figure 6-2b. *List your Azure DevOps organizations*

Once created, you should see the list of organizations, as shown in Figure 6-3.

Note Skip the next steps if the AzureDevOps organization does not show up here after creating one. This can happen if you are using a free Azure subscription. You can still set up continuous integration. The steps are covered in the next section.

Figure 6-3. *Organizations list*

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.

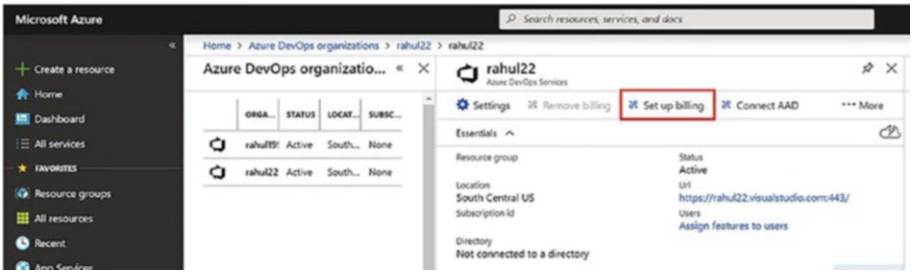

Figure 6-4. *Setting up billing*

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

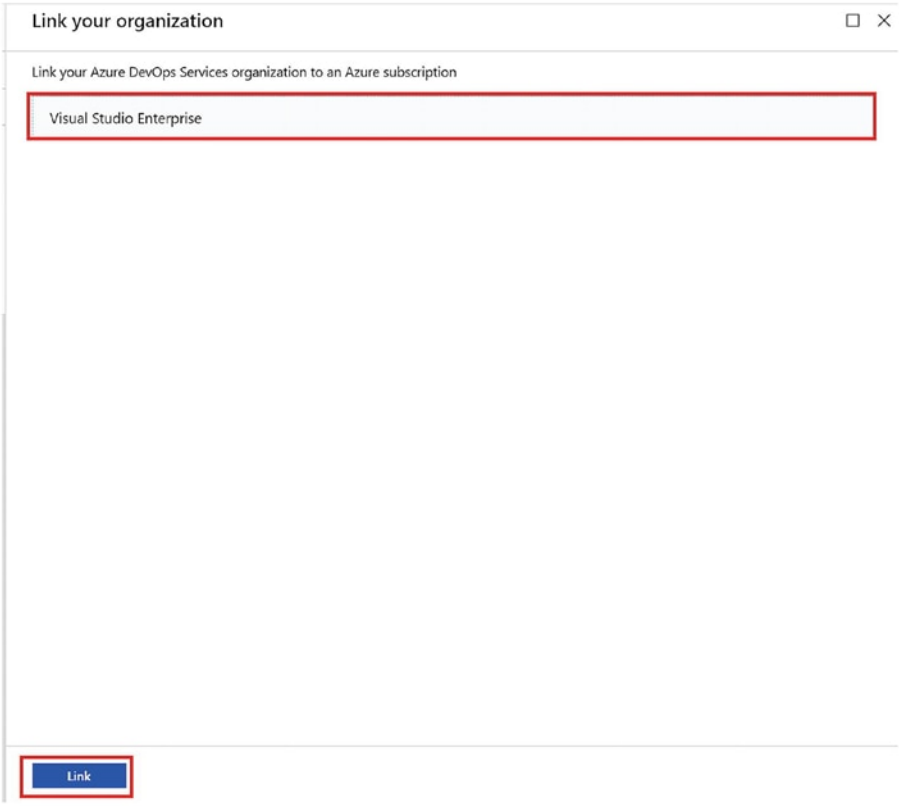

Figure 6-5. *Linking to an Azure subscription*

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

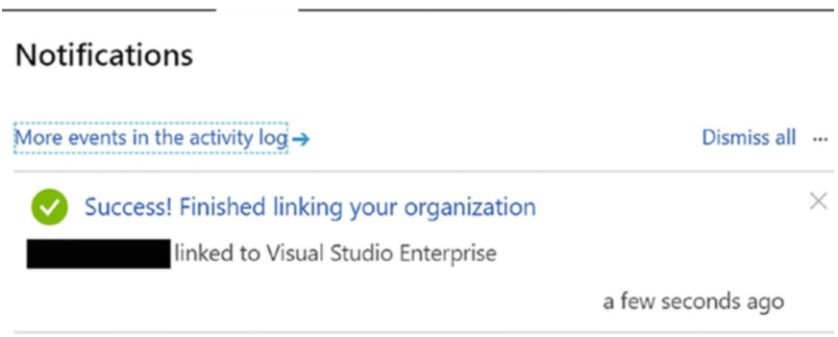

Figure 6-6. *Success*

Enabling Deployment Slots

Azure allows you to create *deployment slots* for Azure Functions and web apps. Think of a deployment slot as a clone of the production environment where you can deploy your app first, and test it before deploying it to the production slot. And if everything looks good, you can just swap the staging slot with the production slot without downtime. This helps 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.

Enable deployment slots on your Azure function app by following these next steps.

On the Azure portal, navigate to your Azure function, select Deployment Slots under the deployment section, as shown in Figure 6-7, and click Add Slot. The production slot corresponds to the function app.

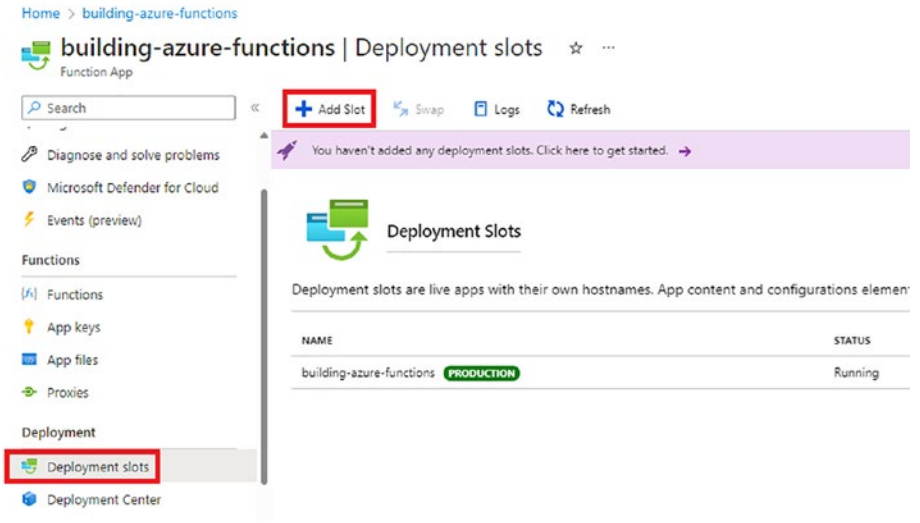

Figure 6-7. *Deployment slot*

Specify a name for the slot and click Add, as shown in Figure 6-8.

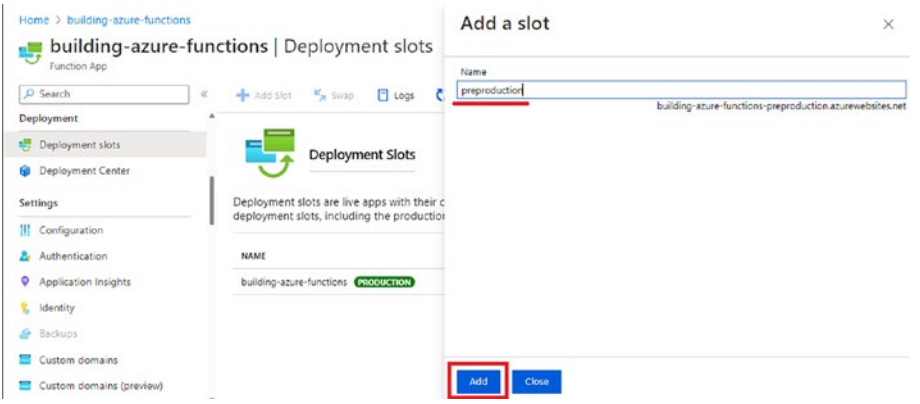

Figure 6-8. *Add deployment slot*

Once the slot is created, navigate to the resource group where your Azure function is deployed. You should see an identical App Service (Slot) and the original function app, as shown in Figure 6-9.

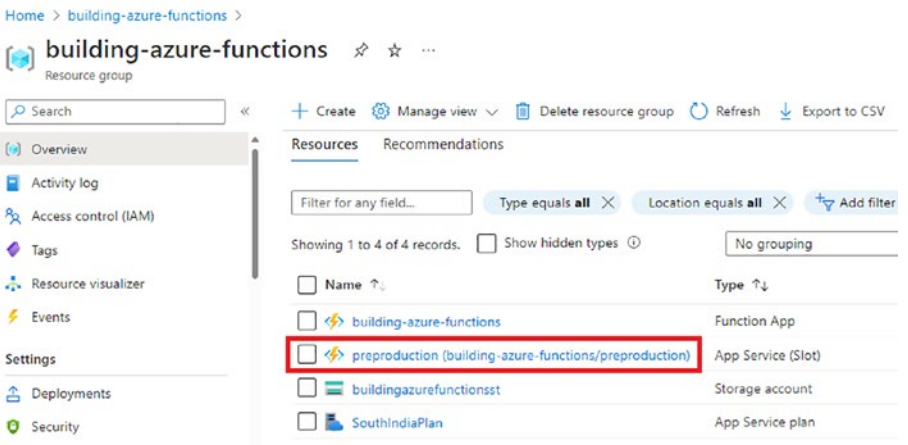

Figure 6-9. *App Service (Slot)*

This new slot is now practically identical to the production app. You can create up to five deployment slots to test various functionalities and configurations before deploying to the production slot.

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.

Go to the Azure portal, navigate to the resource group where the function app is created, select the preproduction slot, and select Deployment Center, as shown in Figure 6-10.

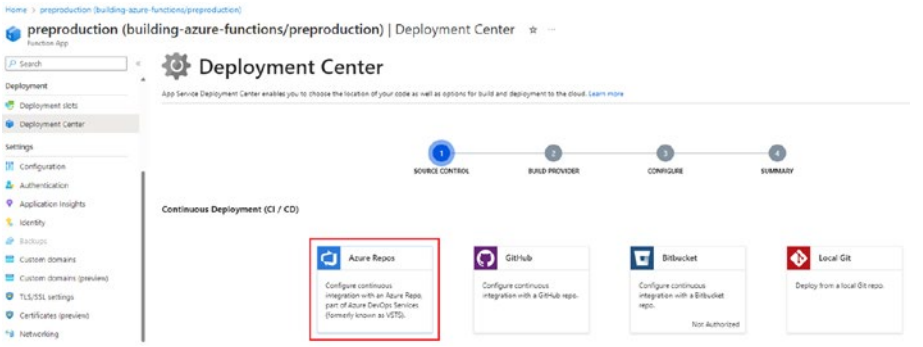

Figure 6-10. *Deployment Center*

Now you should see many options, such as Azure Repos, GitHub, Bitbucket, and so on, from where you can set up CI/CD. For this example, select Azure Repos and click Next.

Azure now presents you with a choice of selecting a build provider. Azure App Service build service should be good for basic scenarios, but Azure Pipelines lets you customize the build artifacts accordingly if you need more control over the build process.

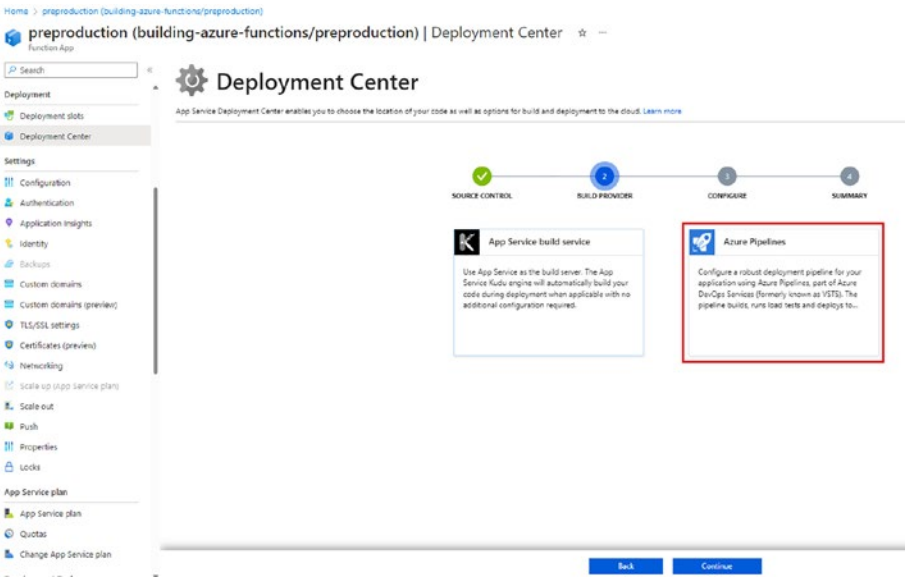

Figure 6-11. *Azure build providers*

Select Azure DevOps Organization, Project, Repository, and Branch menus, and click the Continue button, as shown in Figure 6-12.

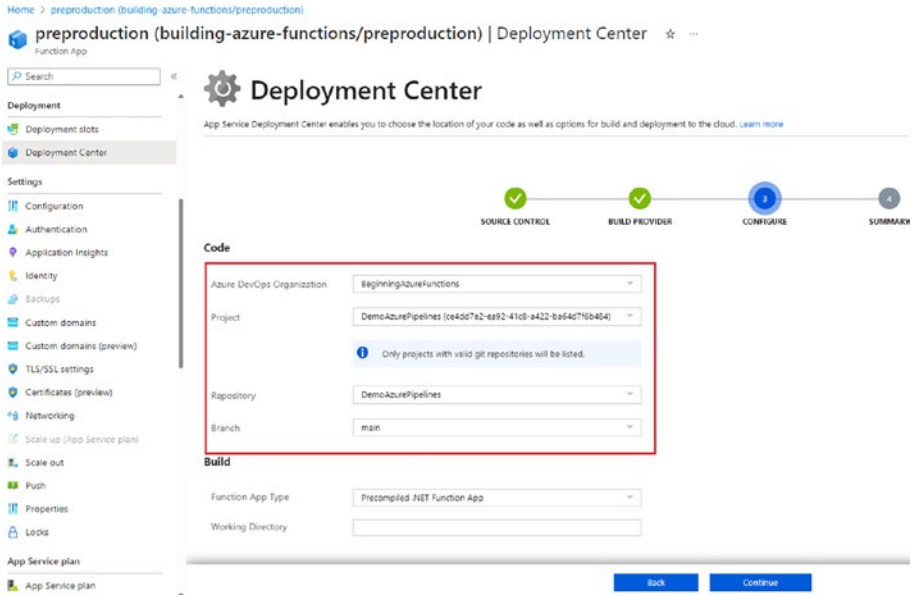

Figure 6-12. *Adding the Azure DevOps configuration*

Check the summary and click Finish, and your continuous deployment is now ready and set up, as shown in Figure 6-13.

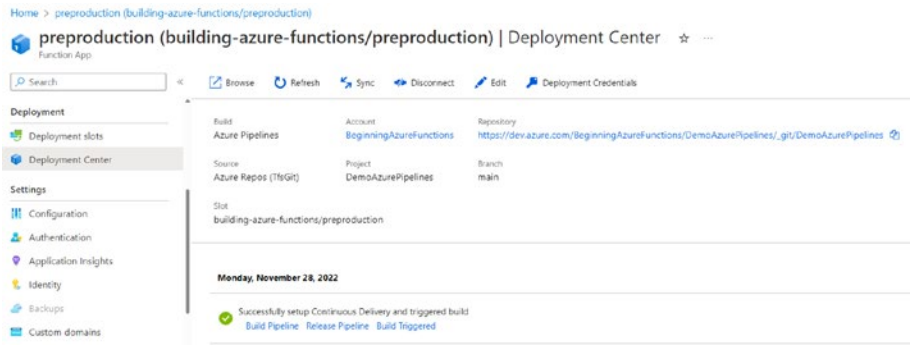

Figure 6-13. *CD now ready*

Click Build Pipeline.

You are taken to the build pipeline of your function, as shown in Figure 6-14.

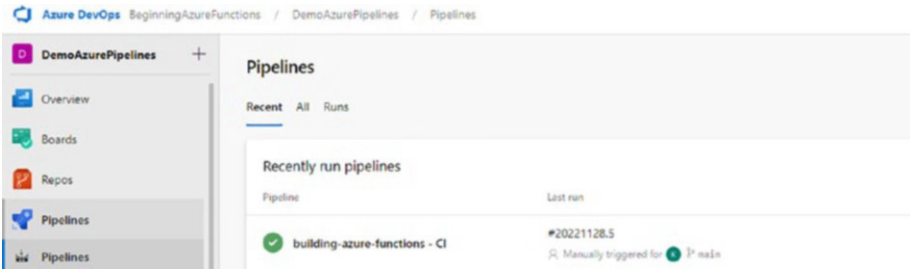

Figure 6-14. *Build pipelines*

To check the release pipeline, click the Release Pipeline link, as shown in Figure 6-14. You are taken to the release pipeline of the function in Visual Studio Team Service (VSTS), as shown in Figure 6-15.

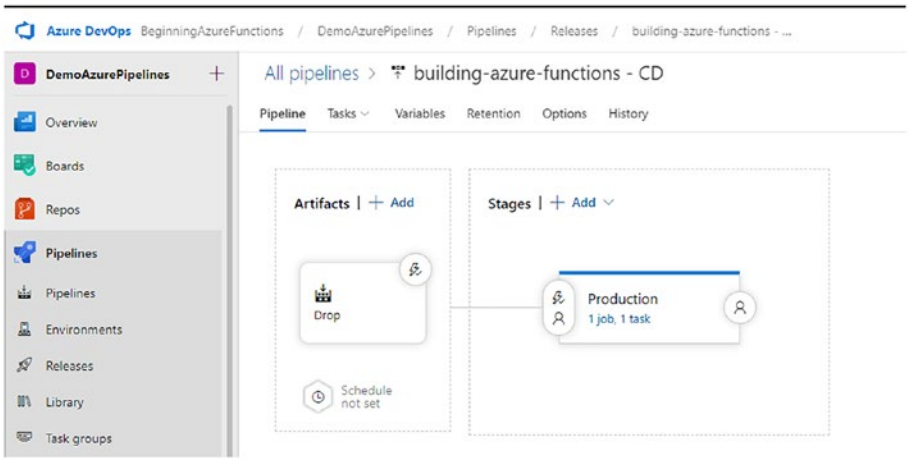

Figure 6-15. *Release pipeline*

Azure has made configuring CI/CD intuitive and easy to set up with just a few clicks.

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 look at the required parameters and resources that enable you to deploy functions with ARM templates.

You need the following resources to start deploying functions using ARM templates.

- An Azure storage account
- A hosting plan
- A function app

Let's set up everything using ARM templates.

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

The following code snippet creates 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 next.

```
"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 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. It dynamically allocates compute power when your code is running. It scales out to handle the extra load and then returns to normal when it lessens. So, if Azure Functions is not running, you are not paying anything for idle virtual machines. Also, you don't have to worry about peak load in advance because the Consumption plan takes 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, the Consumption plan requires two more settings: WEBSITE_CONTENTAZUREFILECONNECTIONSTRING and WEBSITE_CONTENTSHARE. 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
              ('storageAccountid'),'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(resourceGro
  up().id), 'azfunctions')]",
  "storageAccountid": "[concat(resourceGroup().id,'/
  providers/','Microsoft.Storage/storageAccounts/', variables
  ('storageAccountName'))]",
  "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('applicationInsights
        Name')]",
        "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 follows.

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

The following is the complete ARM template for Azure Functions.

```
{
  "$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 is deployed to the staging slot. The following steps take you to the staging slot.

Go to the Azure portal and click Function Apps in the menu, as shown in Figure 6-16.

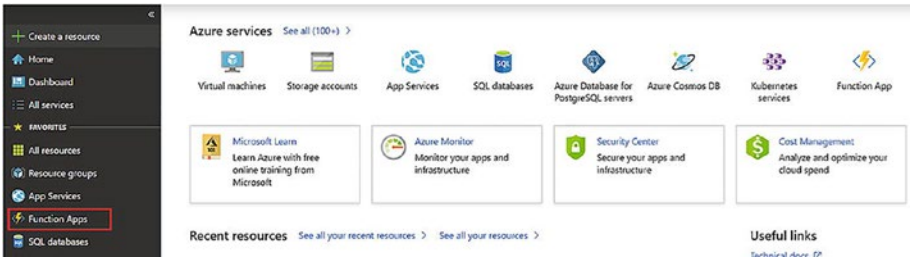

Figure 6-16. *Selecting Function Apps*

Select the function for which you created the CI/CD pipeline. A pipeline was created for `durable-func-new-book`, as shown in Figure 6-17.

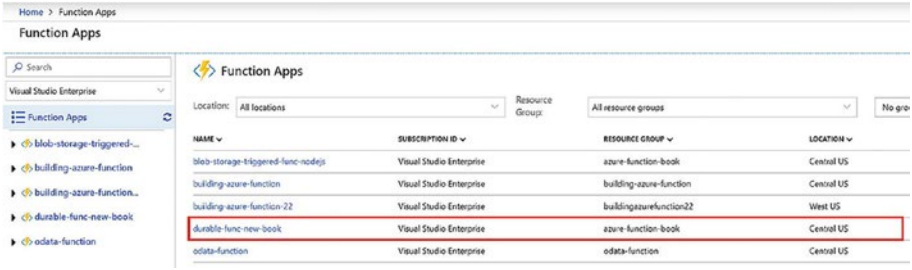

Figure 6-17. *Pipeline*

Click "Platform features" and select Deployment Center, as shown in Figure 6-18.

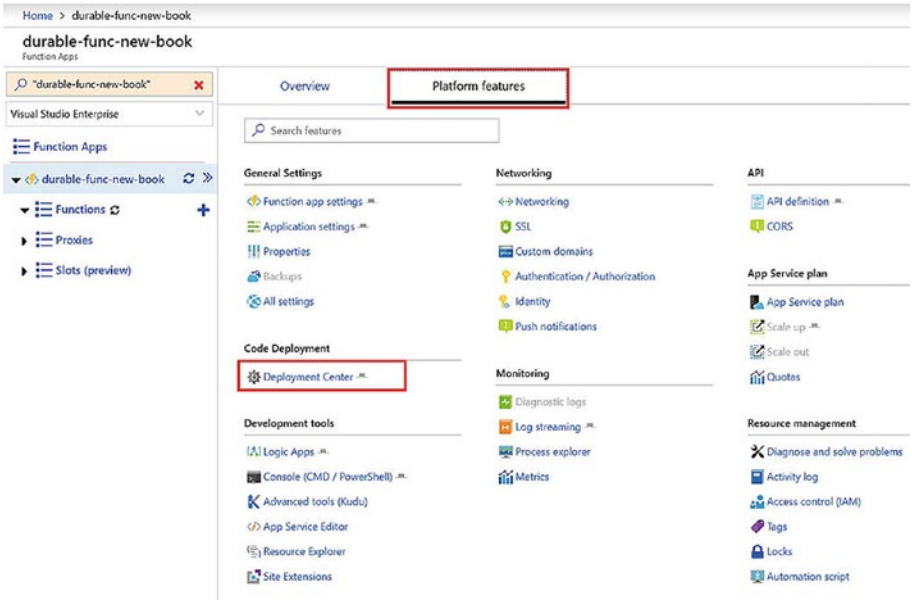

Figure 6-18. *Selecting Deployment Center*

In the Deployment Center, click the slot, as shown in Figure 6-19.

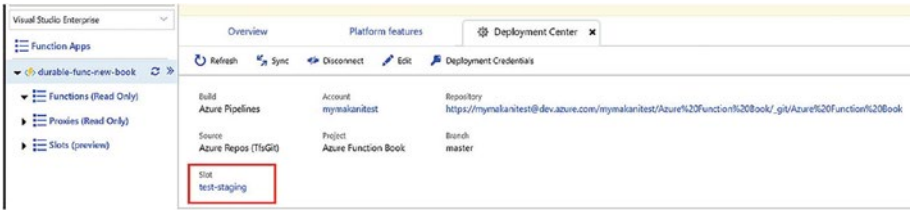

Figure 6-19. *Selecting the slot*

In the slot, you see the URL of the staging slot and the Swap option, as shown in Figure 6-20. Now, you can test your function in the staging slot. Once you are satisfied that things are running fine, you can swap the slot.

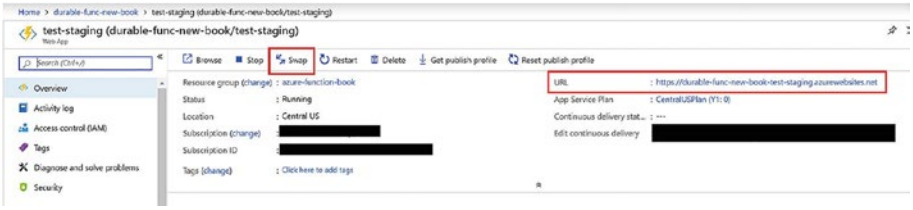

Figure 6-20. *Testing*

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

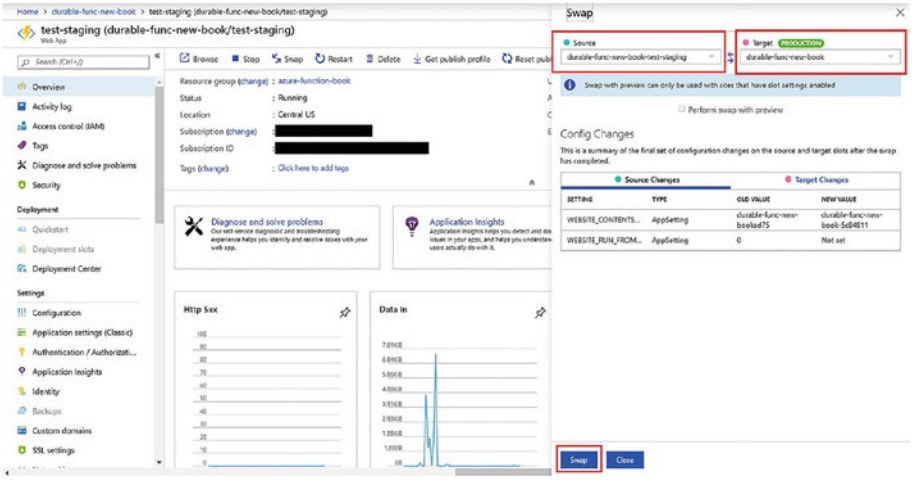

Figure 6-21. *Clicking Swap*

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/`.

Summary

This chapter examined setting up a CI/CD for your Azure function using Azure DevOps, enabling deployment slots for testing your functions before making changes to your functions in production. You also learned about deploying functions using ARM templates. In the next chapter, you look at what's required to make functions production-ready.

CHAPTER 7

Getting Functions Production-Ready

This chapter explains logging, Application Insights, securing functions, and using cross-origin site scripting, and by the end, your function will be production-ready.

This chapter covers the 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 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 has 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.

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_7

```
[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*

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 methods from `Microsoft.Extensions.Logging` to log events. The methods exposed are `LogTrace, LogDebug`, `LogInformation`, `LogWarning`, `LogError`, and `LogCritical`.

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

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

Figure 7-2. *The JavaScript function*

The context passed in Figure 7-2 has a `log` function, which you can use to log at different levels. The `log` function has similar levels, such as Trace, Debug, Information, Warning, Error, Critical, and None.

The host and function logs are stored in a file share in the storage account associated with the Azure function under `/LogFiles/Application/Functions`.

Using Application Insights to Monitor Azure Functions

Application Insights is an extension of Azure Monitor that provides capabilities to monitor your application's performance, availability, and usage. Azure Functions offers seamless integration with Application Insights for logging details such as incoming HTTP requests, response times, failure rates, exceptions, and more.

Application Insights Settings for Azure Functions

To connect, Azure Functions needs to know the Application Insights connection string key. The `APPLICATIONINSIGHTS_CONNECTION_STRING` key must be set in the Azure Functions app settings.

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

- Automatic integration while creating a function app
- Manual connection to an existing Application Insights service

Integrate Application Insights During New Azure Function Creation

Let's see how this is done.

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

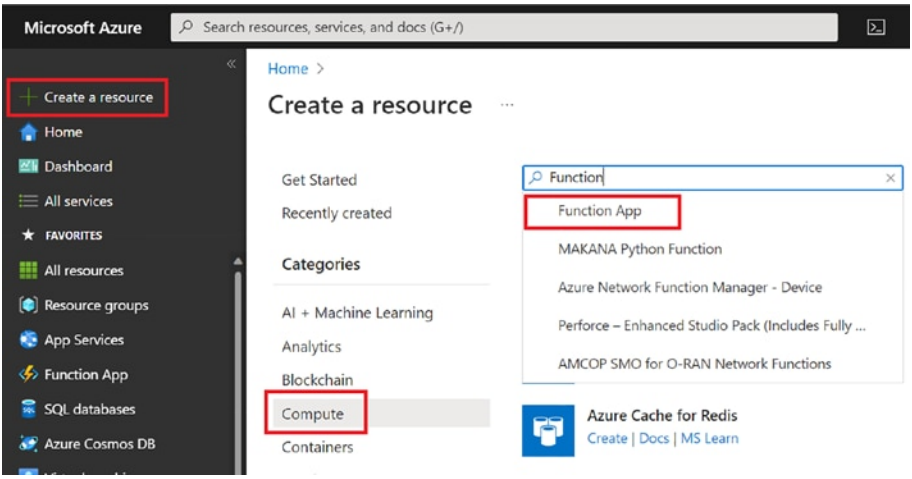

Figure 7-3. *Starting the function app*

In the Basics tab, provide details about the function app, as shown in Figure 7-4. (This was covered in Chapter 2.)

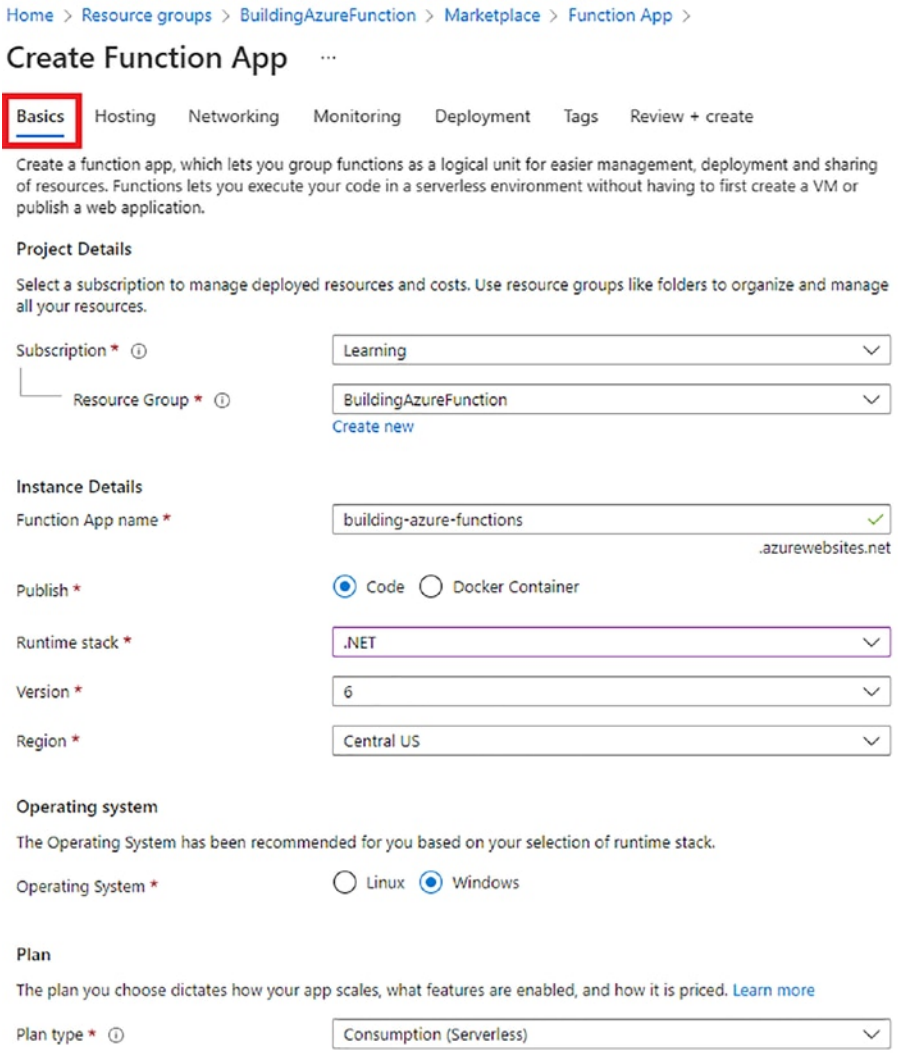

Figure 7-4. *Creating an Azure function*

In the Monitoring tab, select Enable Application Insights. Next, select either "Create new" or "Select existing resource" and set up Application Insights, as shown in Figure 7-5.

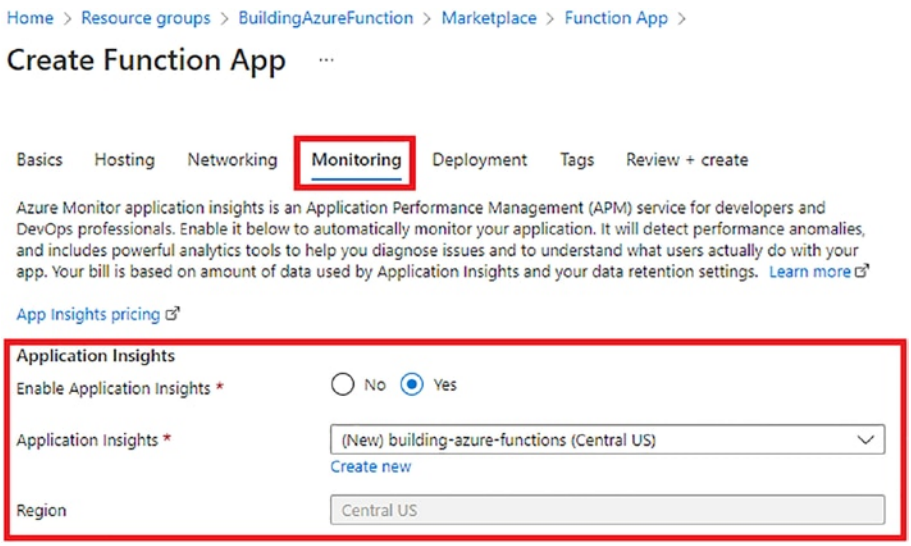

Figure 7-5. *Setting up Application Insights*

Your new function is now integrated with Application Insights.

Manually Connecting Application Insights to Azure Functions

If you already have an Azure function created without Application Insights, you can manually connect them. Let's look at that process now.

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

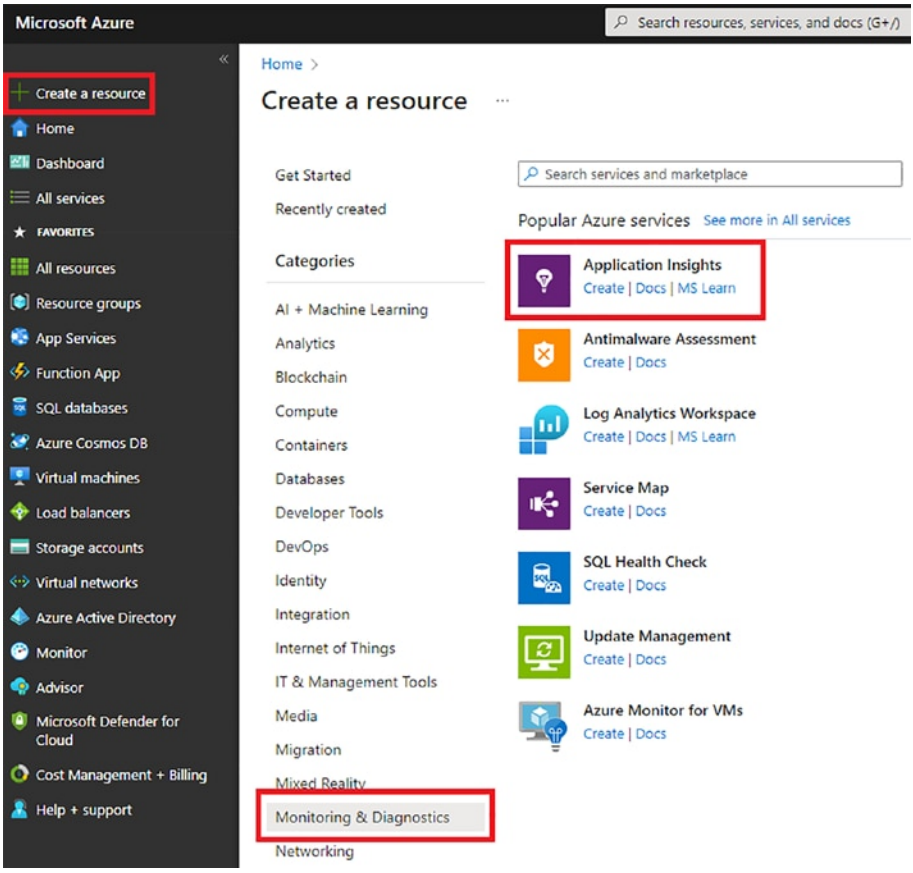

Figure 7-6. *Creating Application Insights*

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.

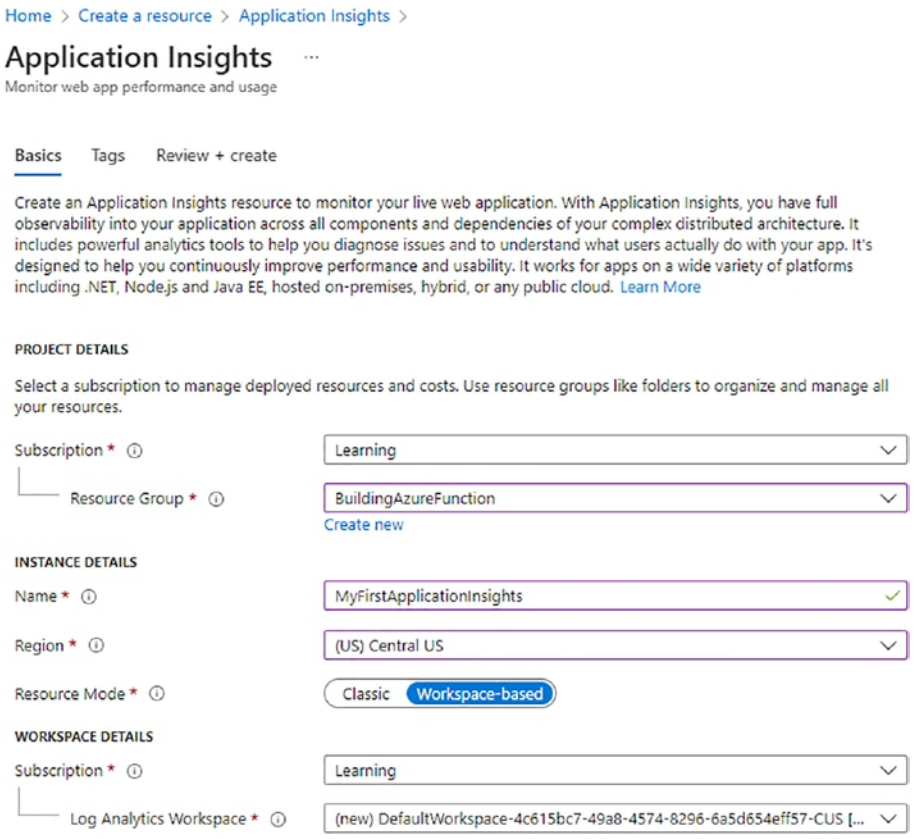

Figure 7-7. *Application Insights properties*

Once Application Insights is ready, navigate to the resource and copy the connection string, as shown in Figure 7-8.

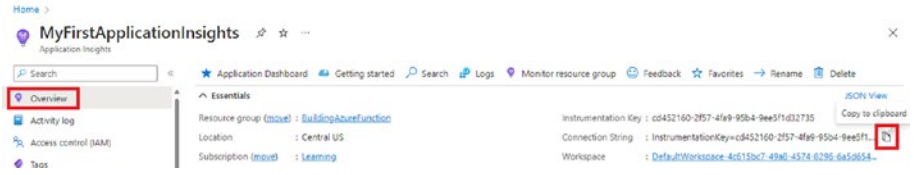

Figure 7-8. *Locating the instrumentation key*

Navigate to the Azure Functions and select Configuration, and then Application Settings. Click "+ New application setting" and add a new key `APPLICATIONINSIGHTS_CONNECTION_STRING` (see Figure 7-9).

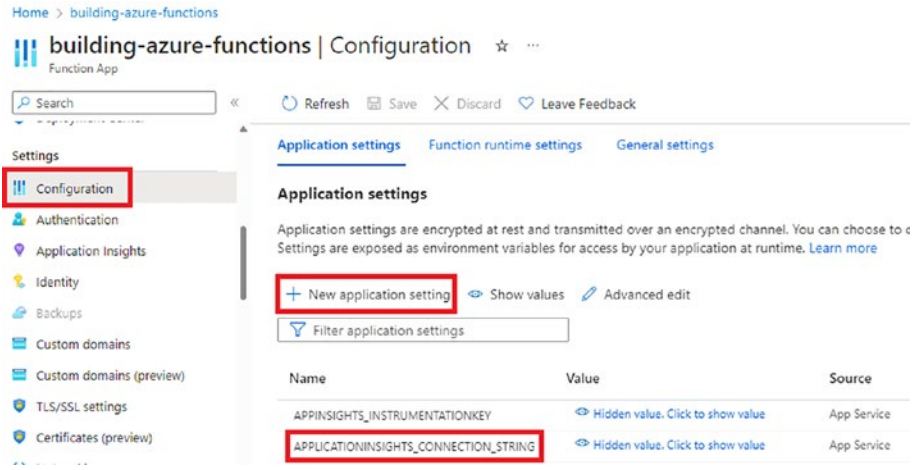

Figure 7-9. *Adding the key*

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

Note You might find another application setting named `APPLICATIONINSIGHTS_INSTRUMENTATIONKEY`. This setting has been deprecated and will be removed in the future.

Disabling Built-in Logging

Because you have enabled Application Insights for your Azure Functions app, it is imperative to disable the built-in logging that uses Azure Storage. The built-in logging is good for lightweight 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 other applications.

Configuring Categories and Log Levels

Application Insights is like a plug-and-play service for Azure Functions. But when used with the default configuration, it could result in high-volume data, and you may hit your data cap for Application Insights.

To avoid that, you can customize the configuration and ingest only the required logs. To do that, you must first understand the log categories 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 `Host.Executor` category.
- The logs that you write in your function have the `Function` category.

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

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

In the preceding configuration, you are specifying the following.

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

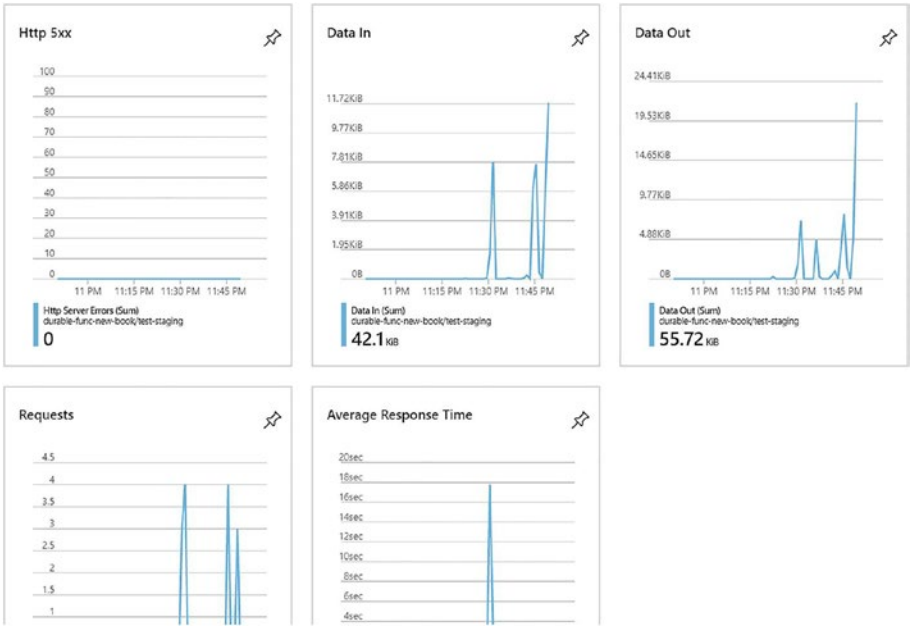

Figure 7-10. *Function in action*

Monitoring functions in production is a necessity. It lets you monitor load, errors, and requests and debug issues in production.

Securing Azure Functions

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

The good thing about Azure Functions is that it provides 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.

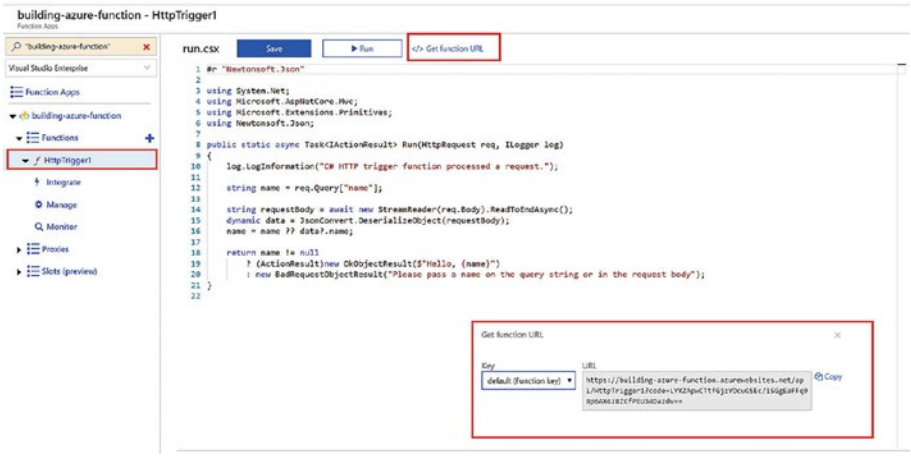

Figure 7-11. *Copying the function*

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 you created in Chapter 4. Now, if your endpoint is not secured (i.e., it does not require any authentication/authorization), you invite hackers to easily get your data.

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

Navigate to the function app you want to secure, click Authentication, and click the "Add identity provider" button, as shown in Figure 7-13.

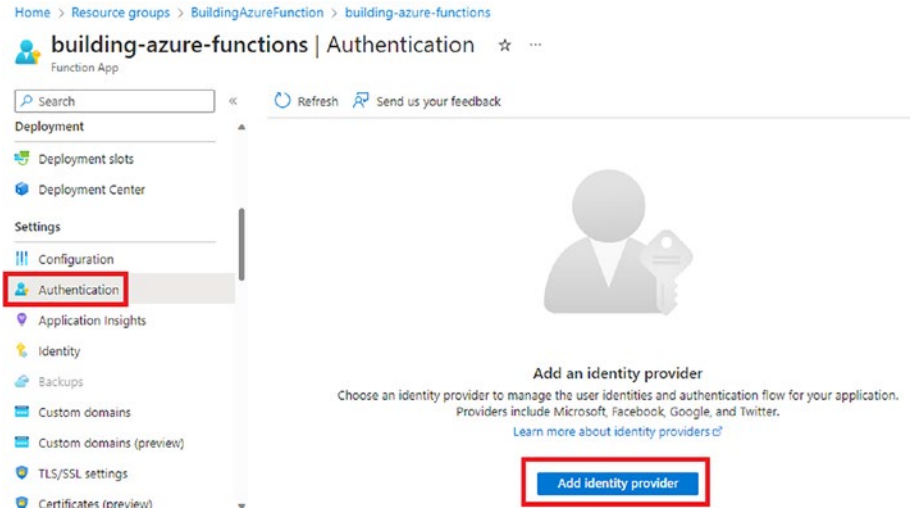

Figure 7-13. *Adding Authentication*

Choose an identity provider from the drop-down menu. Microsoft is used for this example, as shown in Figure 7-14.

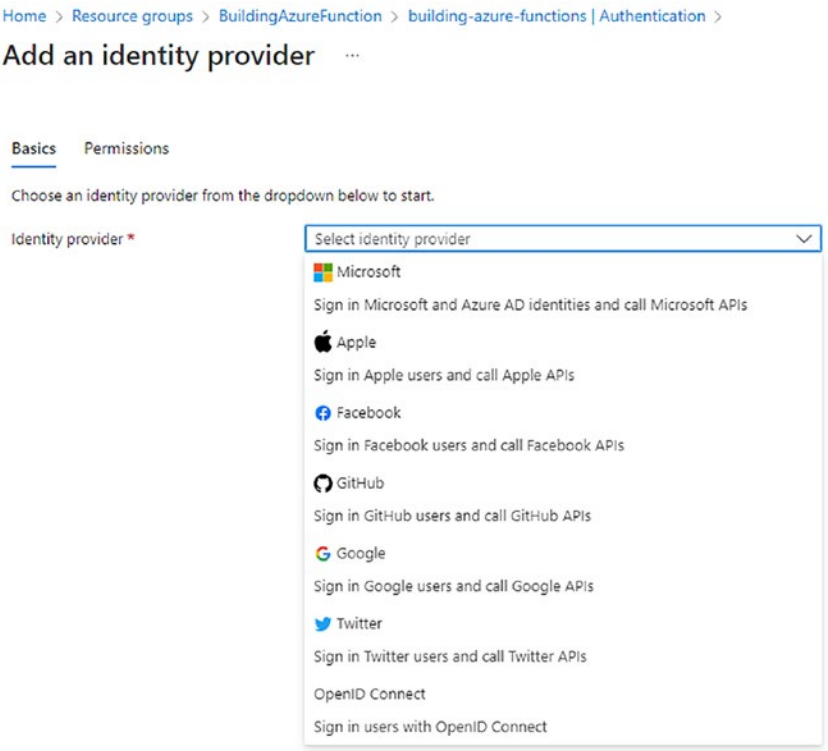

Figure 7-14. *Setting "Action to take when request is not authenticated"*

On the next page, you can create a New AD app. Choose what accounts are allowed to log in and how unauthenticated requests should be handled. This creates the AD app and enables authentication/authorization for this function, as shown in Figure 7-15.

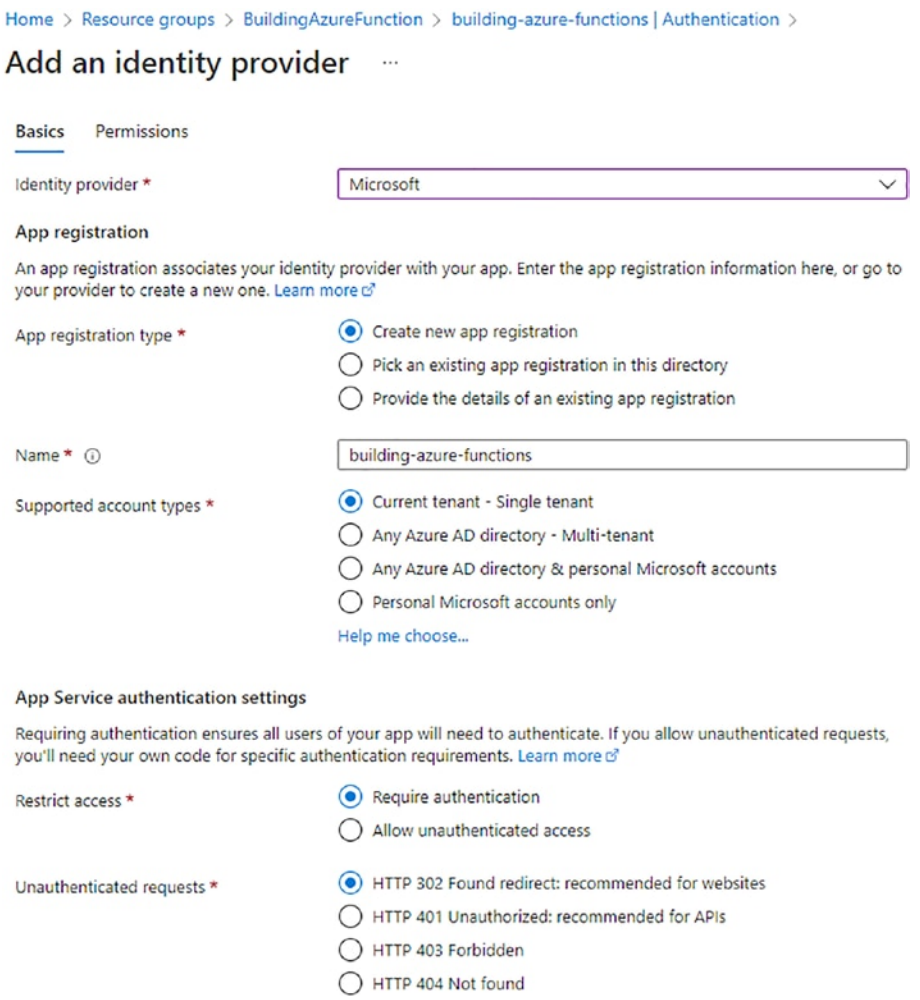

Figure 7-15. *Creating the AD app ID*

Notice how Azure makes adding authentication to your function app easy without writing any code to integrate with the identity provider.

Let's use the URL in Figure 7-12. You see that now it asks you to log in before showing the result, as shown in Figure 7-16.

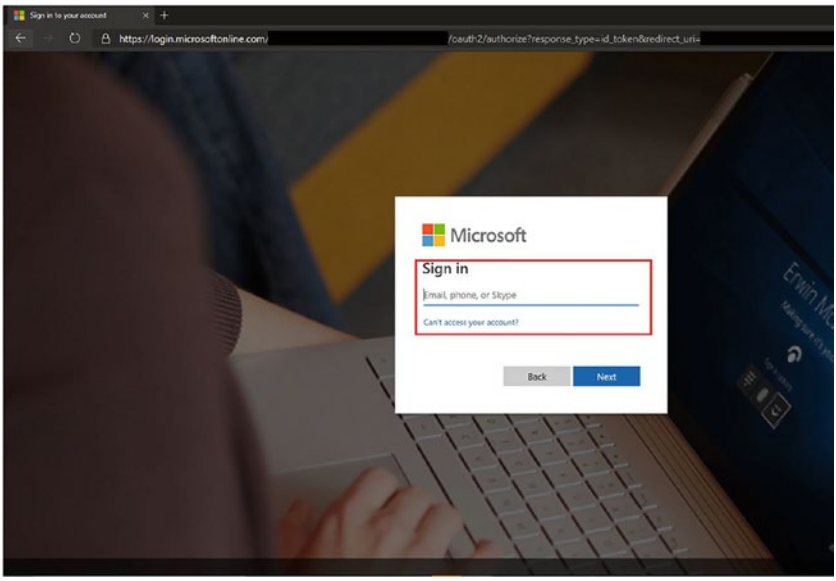

Figure 7-16. *Requesting a login*

You have now secured your function, and only the users in your AD tenant can access this function.

Configuring CORS on Azure Functions

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

If that is the case, you must 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.

Let's go back to the function and select CORS within the API section, as shown in Figure 7-17.

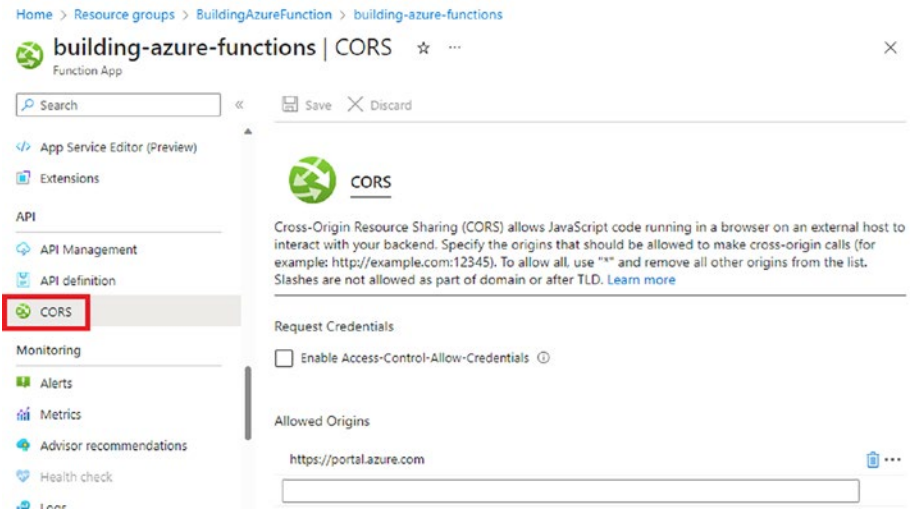

Figure 7-17. *Selecting CORS*

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.

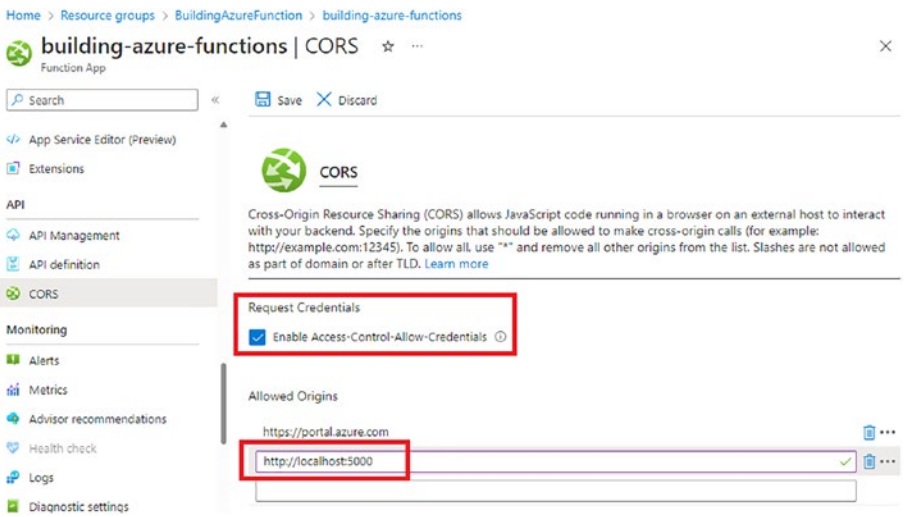

Figure 7-18. *Setting the URL*

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

Summary

In this chapter, you learned about making your Azure Functions production-ready by using the built-in logging capabilities, monitoring using Application Insights, and securing your functions by configuring authentication and CORS. The next chapter discusses some best practices for working with Azure Functions.

CHAPTER 8

Best Practices for Working with Azure Functions

In this chapter, you learn about general best practices for working with Azure Functions, including organizing your functions based on various criteria, developing robust functions, and managing your Azure spending.

Organize Your Functions

Depending on the size and scale of your project, you could build several functions to perform various operations. You can group them into one or more function apps or deploy each function as a separate function app. Before organizing your functions, you must consider aspects like configuration, deployment, performance, scaling, and security of the overall solution. Here are some commonly used patterns and practices.

Based on Business Function

Grouping functions based on a department or business function, such as human resources and payroll, is a common practice among many organizations. This pattern allows deployments and monitoring to be

R. Sawhney and K. Chanumolu, *Beginning Azure Functions*,
https://doi.org/10.1007/978-1-4842-9203-7_8

independently carried out by the respective teams within the organization. Each function app can be assigned to a Consumption plan or a Premium plan and scaled individually according to the requirement.

For Performance and Scaling

A function should ideally be designed for short executions, performing a small unit of work. This ensures that the function has a low memory footprint. Several functions in the same app can still spike up the overall memory usage. Dependencies for every function must be initialized and loaded, which could lead to a slower app startup. This problem is also referred to as the Azure Functions *cold start* problem and is apparent with functions in the Consumption plan.

If you run multiple function apps in a single Premium plan or App Service plan, these apps share the same resources allocated to the plan. If you have one function app with a much higher memory requirement than the other, it uses a large amount of memory on each instance to which the app is deployed. Because this could leave less memory available for the other apps on each instance, you might want to run a high-memory-using function app like this in its own separate hosting plan.

By Configuration and Deployment

Functions within a function app inherit configuration from the `host.json` file, which is used to configure the advanced behavior of function triggers and the Azure Functions runtime. If you have functions that need similar custom configurations, consider moving them into their own function app.

Functions that consume data from other functions or are dependent on other functions to get triggered are good candidates to be grouped together so that changes made to one function do not break functionality on a dependent function and are always deployed together.

By Privilege

Connection strings and other credentials stored in application settings give all functions in the function app the same set of permissions in the associated resource. Consider minimizing the number of functions with access to specific credentials by moving functions that don't use those credentials to a separate function app. You can always use techniques such as function chaining to pass data between functions in different function apps.

By App Visibility

Azure Functions, especially HTTP-triggered public-facing APIs, have different authentication, security, and availability requirements than internal and private-facing APIs. A firewall usually protects public APIs. It is a good practice to group such functions together.

Separate Function Apps for Each Function

While this pattern provides the most isolation from any other processes or inter-service dependencies, it comes with a lot of operational overhead because each function requires its own configuration, deployment, and monitoring. Over time, the number of function apps could become unmaintainable.

The Consumption plan pricing includes a monthly free grant of 1 million requests and 4,00,000 GB/s of resource consumption per month per subscription in pay-as-you-go pricing across *all* function apps in that subscription. You can deploy several function apps and still take advantage of the pricing benefits provided by Azure.

Develop Robust Functions

These are some important practices to consider while developing for scale and performance.

Design for Statelessness

Functions should be designed to perform a small unit of work without dependency on the outcome of a previous execution. For instance, a function to email order summaries to customers should be able to resume from where it was left in the previous occurrence. The function should rely on the state of the order to determine if an email needs to be sent. Here, the state belongs to the data being processed, not the function itself.

Leveraging a durable function when maintaining a state is necessary, and there is a need to execute one function after the other in a particular order across multiple functions.

Use Messaging Services

Use messaging services like Azure Queue Storage or Azure Service Bus to relay messages between functions instead of invoking a function from another function to transfer data. For instance, a function that processes orders in an e-commerce system should place a message on a storage queue with the data required by another function for sending a push notification to the customer. Such a design allows the entire system to scale out and scale when there is a sudden surge or decline in the number of orders being processed. Messaging services also provide features like delivery and ordering guarantees to ensure that the consuming function receives and processes the message.

Integrate with Monitoring Tools

The Azure portal provides basic metrics like memory utilization and function execution count to monitor your functions. But these are insufficient to determine if your app is experiencing performance issues or throwing exceptions. APM (*application performance monitoring*) tools like Application Insights can integrate easily with Azure Functions and collect detailed information on function invocations, execution times, and resource utilization and capture the stack trace of exceptions in your code for troubleshooting. It also allows you to configure email and text alerts when performance deteriorates beyond a certain threshold.

If you are already using other APM tools like AppDynamics and New Relic, Azure Functions integrates with them to provide a consolidated monitoring experience for all your applications.

Use Separate Storage Accounts

When you create an Azure Functions app, it also creates Azure storage to store logs and execution schedules. This is metadata required by the function to work. Data is being written to this storage account throughout the function execution.

If you need to use storage blobs or queues with your function, you should create separate storage accounts instead of reusing the one bound to the function app. This is because storage accounts have their own concurrency, throttling, and throughput limits and could seriously limit the performance of your function app.

Account for Delays with Function Cold Starts

This problem is relevant to the Consumption plan. Due to the serverless nature of Azure Functions, when functions are not invoked for an extended duration (approx. 20 minutes), Azure deallocates the resources assigned to

the app. Next time the function is triggered, the function's runtime needs to start and load dependencies before responding to the event. The total time elapsed from the event occurring until the function responds to the event is called *cold start time* (usually up to 10 seconds).

Implement a retry pattern in the calling app to wait and retry for a response if it receives a time-out on the first attempt.

You can also create a function to periodically poll and invoke the other functions to keep them "warm" even if they are not doing any actual work.

When a function needs to run, all the code is eventually loaded into memory, which takes longer with a larger application. So, if the function has many dependencies, the cold start times are longer due to increased time for I/O operations from Azure Files and the longer time needed to load them into memory.

For functions that cannot afford this latency, choose a Premium or App Service plan so that the function's host is always running.

Use Asynchronous Patterns

Make your code as asynchronous as possible. Avoid blocking calls and ensure that any tasks, callbacks, threads, and processes initiated by the function are completed before your function code returns. Functions don't track these background threads; site shutdown can occur regardless of background thread status, which can cause unintended behavior in your functions.

For example, suppose a function starts a background task and returns a successful response before the task completes. In that case, the Azure Functions runtime considers the execution as having been completed successfully, regardless of the result of the background task. If this background task is performing essential work, it may be preempted by site shutdown, leaving that work in an unknown state.

Manage Costs

Azure Functions' instances in the Consumption plan automatically scale to zero when there are no triggers for the functions. However, you are billed for Application Insights depending on the volume of events collected. The monitoring costs can become substantial and exceed the cost of Azure Functions. Configure the log level to exclude Informational events if they are not necessary for production. Application Insights also allows you to specify sampling settings to reduce traffic and storage costs.

While inbound or ingress traffic to Azure is free, outbound or egress traffic from Azure is billed. So, networking costs will apply if your Azure Functions serve data to the outside world. Depending on the traffic and volume of data transferred, these costs can add to your overall application costs.

The storage account associated with the function is also charged for the storage used. Although these charges are insignificant, they get added to your total Azure spend.

Make sure that you understand the pricing model and closely monitor your usage and Azure bill.

Summary

With this, you have come to the end of the book. We hope this book helped you start your journey with building serverless applications on Azure. You will learn more as you dig into the concepts explained in this book and start building your own functions. Happy learning!