AZURE SERVERLESS

SUCCINCTLY

BY **SANDER ROSSEL**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Azure Serverless Succinctly

By
Sander Rossel

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

ISBN: 978-1-64200-201-0

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

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

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

Information is plentiful but harder to digest

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

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

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

The *Succinctly* series

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

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

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

The best authors, the best content

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

Free forever

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

Free? What is the catch?

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

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

Let us know what you think

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

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

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Sander Rossel is a Microsoft-certified professional developer with experience and expertise in .NET and .NET Core (C#, ASP.NET, and Entity Framework), SQL Server, Azure, Azure DevOps, JavaScript, and other technologies. He has an interest in various technologies including, but not limited to, cloud computing, NoSQL, continuous integration/continuous deployment, functional programming, and software quality in general. In his spare time, he writes articles for MSDN, CodeProject, and his own blog, as well as books about object-oriented programming, databases, and Azure.

Chapter 1 What Is Serverless?

Serverless computing is a cloud service where the cloud provider dynamically allocates infrastructure to run your code. Your provider scales hardware dynamically based on your usage and deallocates hardware when your code isn't running. Pricing is based on your usage.

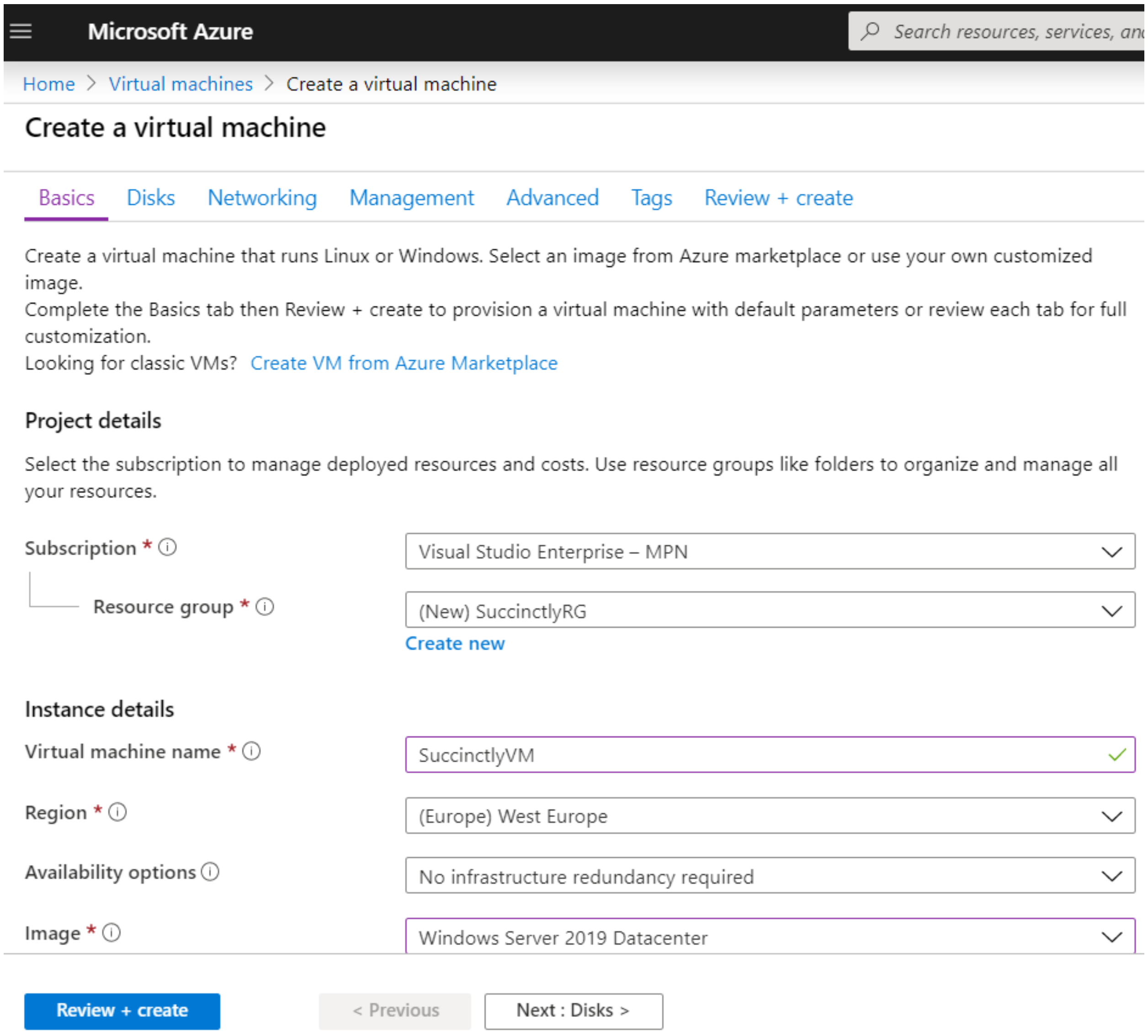
Businesses worldwide are rapidly embracing cloud computing. That means our software and hardware environments are changing. Companies no longer need on-premises hardware, or need significantly less of it. Infrastructure and software are deployed using scripts and continuous deployment. However, whether you have a physical server, an on-premise virtual machine (VM), or a VM in the cloud, you still need to maintain the server, install updates, and troubleshoot issues. No matter how or where you host your infrastructure, you'll always need maintenance.

The cloud has four types of cloud services. Hosting infrastructure in the cloud is also called infrastructure as a service (IaaS). It's easier to run applications on a predefined host and leave the infrastructure to your cloud provider, but options are more limited. They provide a middleware on top of your infrastructure that makes software run out of the box. Such platforms are named platform as a service (PaaS). Another step "up" is using software in the cloud, such as Office 365 or Dynamics 365. This is called software as a service (SaaS), and is not offered in Azure directly. The fourth and last cloud service is serverless. In this chapter we're going to briefly explore IaaS and PaaS, and then move on to serverless for the remainder of the book.

Infrastructure as a service (IaaS)

One reason to use VMs in the cloud is that it's easy to grab your on-premises VM and move it to the cloud. You can run all of your on-premises workloads, except they now run in the cloud. VMs give you the freedom to install whatever you want, but at the cost of having to maintain your own server.

Let's look at the VM creation blade in Azure. We can see that we have to plan for availability, and we need to select a region (which can affect the available machines), an image, and a size. We need to provide administrator credentials, and on the next blade we need to configure hard disks, networks (or virtual networks (VNETs), network interface cards (NICs), and IP addresses), security rules (for example, open port 3389 for remote desktop connections), and diagnostic tools.


Microsoft Azure

Search resources, services, an

Home > Virtual machines > Create a virtual machine

Create a virtual machine

Basics | Disks | Networking | Management | Advanced | Tags | Review + create

Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image.
Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization.
Looking for classic VMs? Create VM from Azure Marketplace

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Visual Studio Enterprise – MPN
Resource group *	(New) SuccinctlyRG
	Create new

Instance details

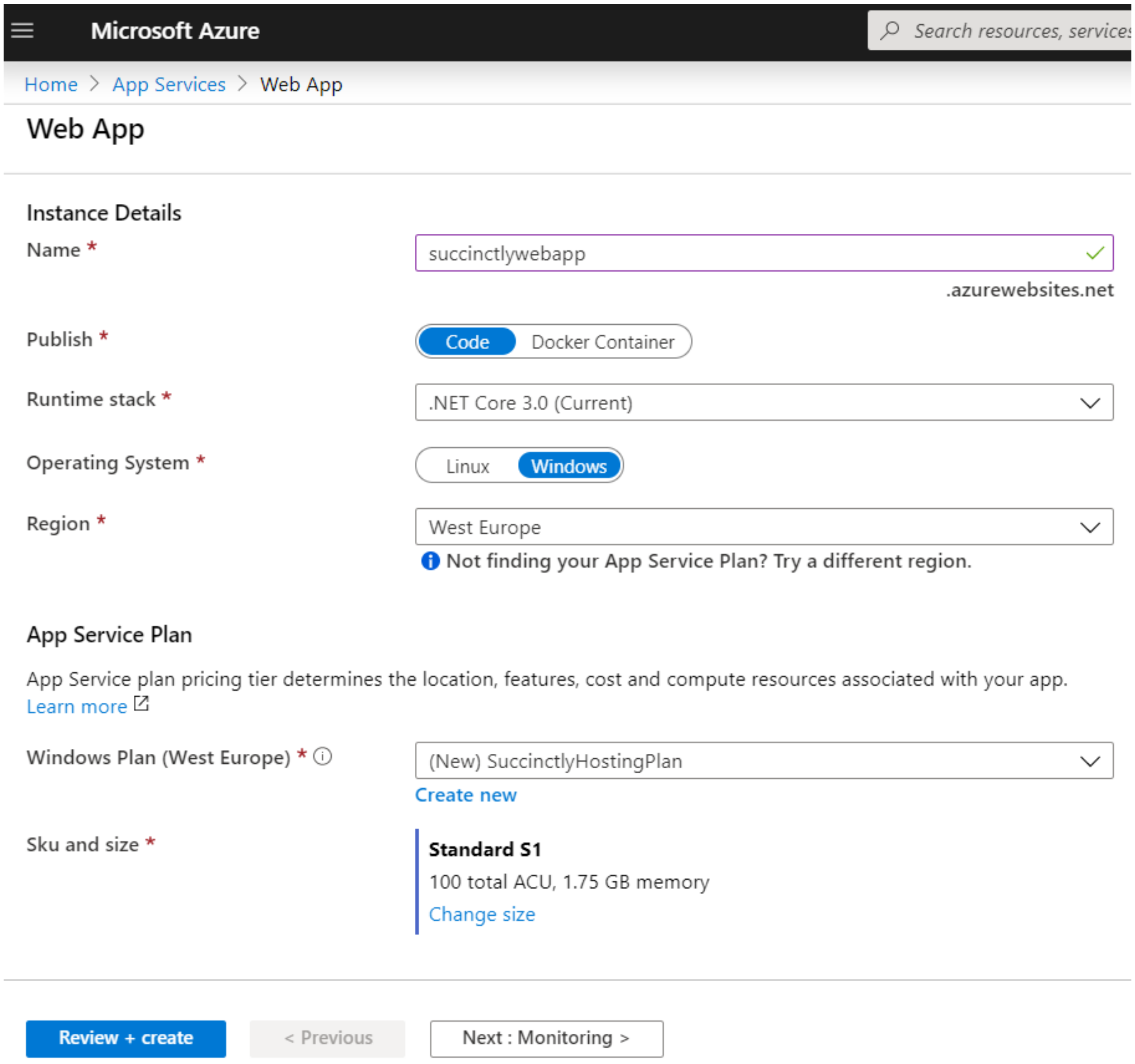
Virtual machine name *	SuccinctlyVM
Region *	(Europe) West Europe
Availability options	No infrastructure redundancy required
Image *	Windows Server 2019 Datacenter

Figure 1: Creating a VM

Luckily, the default settings provide a sensible standard for trying out VMs in Azure, so we don't need to know all that stuff when we're playing around. Still, it's quite a lot to manage, and if you just want to host your .NET application in Azure, there are simpler and cheaper options.

Platform as a service (PaaS)

With platform as a service, the maintenance effort is significantly reduced. In Azure this would typically be an app service with a hosting plan. Instead of hosting software on a server or VM, you can host software in the cloud and not worry too much about server maintenance. You still rent some server space for your application to run on, but the server is maintained by your cloud provider. You must decide on a server plan, for example, one with at least 2 GB of memory and four cores (or compute units). After that, you have to monitor if your service plan is enough and plan for upscaling (increase memory and/or compute units) or out-scaling (adding additional instances) if it isn't. Additionally, you can set some settings like Always On, HTTPS Only, certificates and authentication, and authorization using the intuitive Azure portal.



Web App

Instance Details

Name * succinctlywebapp .azurewebsites.net

Publish * [✓ Code] Docker Container

Runtime stack * .NET Core 3.0 (Current)

Operating System * Linux [✓ Windows]

Region * West Europe

Not finding your App Service Plan? Try a different region.

App Service Plan

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app.
Learn more

Windows Plan (West Europe) * (New) SuccinctlyHostingPlan
Create new

Sku and size * **Standard S1**
100 total ACU, 1.75 GB memory
Change size

Figure 2: Creating an App Service

With an app service you can select either a Linux or Windows OS, but no distro or version. There are no networks or IP addresses to specify. The most important feature is the App Service plan, but that can be changed after creation. The only additional option we must specify is whether we want monitoring, which is as easy as choosing a name for our Application Insights instance that is going to monitor this App Service.

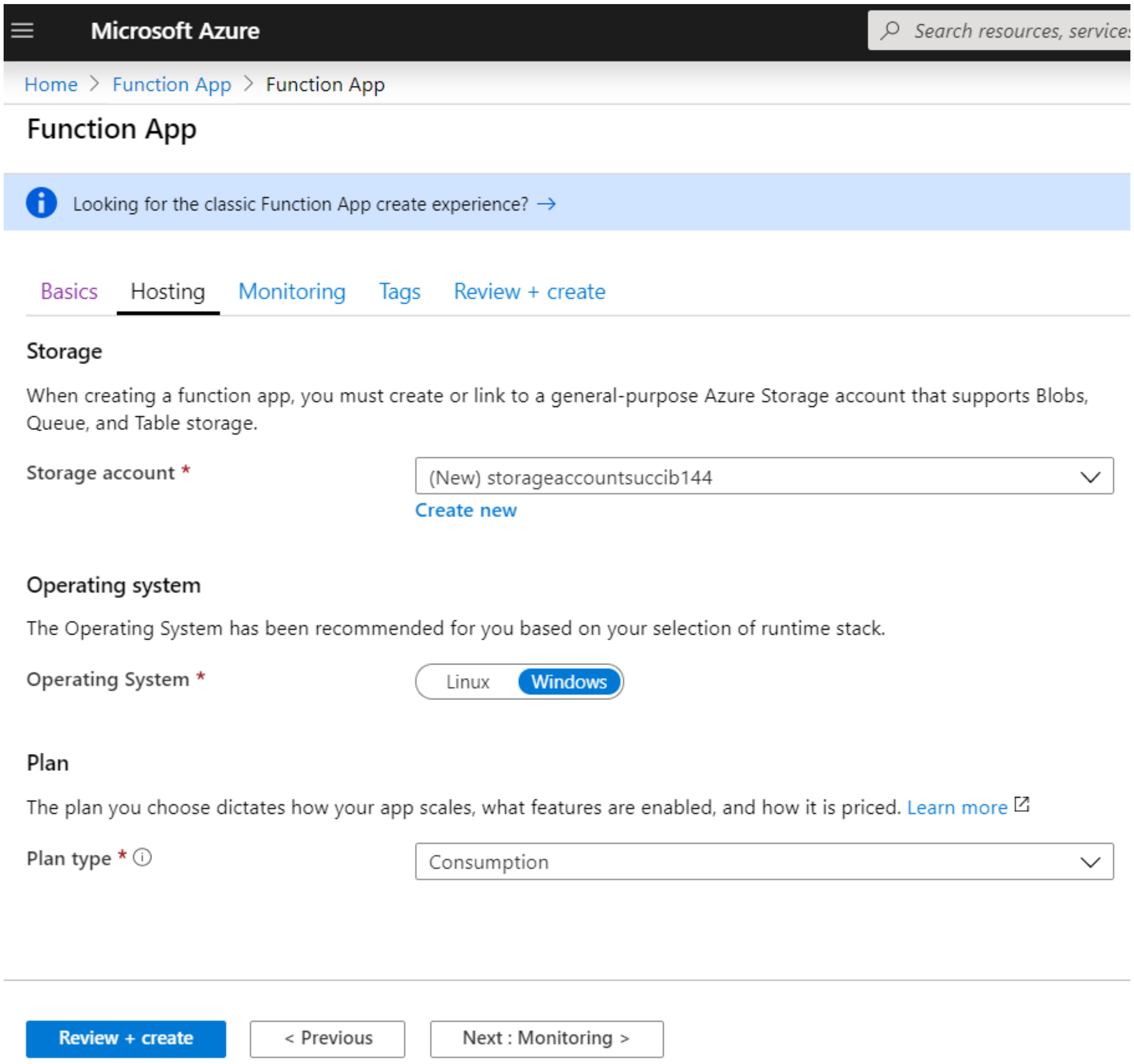
Once you have created an App Service, you can use Visual Studio to directly publish a web app to Azure, which will then be hosted on https://[*your-app-service-name*].azurewebsites.net. You can specify custom domain names. There's now even a service in preview that lets you create managed SSL certificates for your own domain names, which are automatically renewed using Let's Encrypt.

While PaaS is easier than IaaS, it does have its downsides. On a VM you can install anything you want, but on an App Service, not so much. For example, if you're using SQL Server on-premises, you can most likely use the Azure SQL PaaS offering, although it doesn't support functionality such as SSRS and the SQL Agent. When you want an Oracle database in Azure, you have no other option than to create a VM and install it there. When your application depends on software such as Microsoft Office being installed on the server that it runs on, you're out of luck—and PaaS isn't for you.

Serverless

Serverless is the next step in this evolution. You get even less control over your environment, but with some great benefits. First, serverless is a bit of a misnomer. Of course, your code still runs on servers, but you have minimum control over them. In fact, when your code isn't running, the server is allocated to other Azure users who need to run their code. The code for your serverless applications is stored in an Azure storage account and started up when it is somehow triggered.

To run code in a serverless solution, you're going to need an Azure function app to hold your functions. Functions can be triggered by various events, such as a timer, an HTTP call, a message on a storage account queue, or a message on a Service Bus queue or topic.



Function App

Looking for the classic Function App create experience? →

Storage

When creating a function app, you must create or link to a general-purpose Azure Storage account that supports Blobs, Queue, and Table storage.

Storage account * (New) storageaccountsuccib144

Create new

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System * Linux [✓ Windows]

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. Learn more

Plan type * Consumption

Figure 3: Creating an Azure Function App

The creation of an Azure function app seems a bit more complicated than that of a web app, but it's just a name, a development stack (.NET Core, Node.js, Python, Java, or PowerShell Core), a storage account to store your code, an operating system, a plan type (which I'll be discussing later), and whether or not you want to use Application Insights.

Another serverless solution is Azure Logic Apps, which allows you to click together a workflow using a designer in Azure. Logic apps are triggered by various events, like functions, but they have additional events, such as receiving an email in Outlook or a tweet on Twitter. A logic app can trigger a function, and vice versa (through HTTP).

Other serverless offerings in Azure include Service Bus, Event Grid, Azure SQL, and API Management. We'll see them all in the remainder of this book.

Cheap scaling

Because you do not have a dedicated server at your disposal, the use cases for serverless are limited. For example, consider the startup time that's needed if your function triggers create some latency. Furthermore, the default timeout for a function is five minutes, but can be increased to 10 minutes.

One of the biggest pros is that you also don't need to maintain a server, and you don't pay for a server that you're not using. Since servers are allocated when you need them, you aren't limited to a single server, either. When you have some heavy workload, for example, you need to process hundreds or even thousands of images at a couple of minutes per image, both functions and logic apps will scale accordingly. So, when Azure notices that your allocated server is at full capacity, but new triggers are still incoming, Azure will automatically scale out new servers to run your additional workloads. When a server has been idle for a few minutes, Azure can also automatically scale down server instances.

The pricing model for functions and logic apps is a bit difficult and hard to measure, but it's a mix of number of executions and resource consumption measured in gigabyte seconds. However, the first million executions and the first 400,000 GB-s of resource consumption are free. That means you probably don't pay anything at all for small and medium-sized businesses with small workloads. If, however, your function is working the whole day, every day, on multiple instances, the price goes up quickly, and you may be better off with an app service, or even VM. Of course, when Azure scales out some servers, your number of executions and GB-s are increased as well, and your bill will go up accordingly. You can find more information on Azure Functions pricing here.

In the next chapter, we're going to dive into Azure Functions and serverless.

Note: Serverless isn't limited to the Azure cloud. For example, the AWS equivalent of Azure Functions is AWS Lambda, and in the Google Cloud it's called Google Cloud Functions. There is nothing stopping you from using functions in one cloud and calling functions in another. For example, say you want to do some processing in Go, which isn't supported in Azure, but is supported in AWS. Multicloud environments are becoming increasingly popular, but this book will focus on Azure.

Summary

Serverless is becoming increasingly popular. It's potentially cheap and has automatic scaling out of the box. With serverless, you don't have to worry about infrastructure, operating systems, updates, or installed software. While this is a strength of serverless computing, it also limits the possible use cases. Luckily, serverless can complement your other computing infrastructure.

Chapter 2 Azure Functions

In the previous chapter we briefly touched upon the concepts of serverless and functions. In this chapter we're going to write Azure functions and see them in action. There are various ways to create and deploy functions to Azure. The easiest is by creating a function app in the Azure portal and creating new functions there. It's also possible to write and deploy functions using Visual Studio. Lastly, it's possible to deploy functions using Azure DevOps. We're going to explore all three. We'll see triggers as well as input and output bindings. Finally, we're going to write a durable function.

Creating a function app using the portal

The easiest way to create an Azure function app is through the portal. First, we need a function app, a specific sort of App Service tailored to host functions. While function apps are shown in the App Service overview, you can't create a function app from here. Instead, click **+ Add** in the upper-left corner, or go to **Create a resource** and search for **Function App**.

The page that opens is the same one we saw in the previous chapter. It's straightforward. Assign or create a resource group, assign a unique (across Azure) name for your function, specify that you want to publish code rather than containers, select .**NET Core** as your runtime stack, and select a region that's closest to you (or your customers). The function app name will be part of the function's URL, **[function name].azurewebsites.net** (which is the same for an app service).

On the **Hosting** blade, we need to specify a storage account. For this purpose, I recommend creating a new one, which will be placed in the same resource group as your function app, so you can delete the entire resource group once we're done. Choose **Windows** for your operating system and **Consumption** for your plan type.

Tip: Once your function and storage account are created, head over to the storage account. The storage account was probably created as general purpose v1, which is outdated. Go to the *Configuration* ***blade in the menu and click*** *Upgrade*. ***This will upgrade your account to general purpose v2, which is faster and cheaper, and has additional functionality. That said, all the samples in this chapter can be completed with a general purpose v1 storage account.***

On the **Monitoring** blade, enable **Application Insights**. You can skip tags altogether. Go to **Review + create** and click **Create**. This will deploy an empty function app.

The plan types need a bit of explanation.

Plan types

Currently, there are three plans that decide how your functions will run and how you will be billed. The Consumption plan is the default, but you can also opt for an app service plan or a premium plan.

The Consumption plan is the default serverless plan: you get servers dynamically allocated to you, providing automatic scaling, and you pay per execution and GB-s. This is potentially the cheapest plan and the plan that needs the least configuration. This is the only true serverless plan in the sense that if your function is not running, you won't get servers allocated to you—and you'll pay nothing for it.

With an App Service plan, you can use a hosting plan that you may already have in place for an (underutilized) app service. With this option you get a regular service plan that can run your workloads, but doesn't scale as much. You can still scale up (upgrade your plan, for example from Basic to Standard) or scale out (add additional instances, either manually or automatically). The pros to this solution are that you have predictable costs, which are the monthly costs for your plan, and the timeout is set to a default of 30 minutes, but can be extended to unlimited execution duration, meaning you have no timeout at all. On top of that, you can enable "Always On" in the function app settings, which eliminates the latency at startup, because it doesn't have to start every time. Obviously, this isn't serverless, but it has some use cases when you still want to use functions.

The Premium plan combines the Consumption and App Service plans, but is also potentially the most expensive. Billing is based on number of core seconds and memory used. A so-called "warm instance" is always available to eliminate startup latency, but that also means there's a minimum cost starting at about €70. Furthermore, you also get unlimited execution duration, you can use it for multiple function apps, and you can connect to VNETs or VPN. You still enjoy the scaling benefits that come with the Consumption plan.

Which plan type to choose is up to you and your specific situation. For the remainder of this chapter, we'll use the Consumption plan.

Creating a function in-portal

When the function app is created, we can start to write functions. Go to the function app and click the **+** symbol next to **Functions**.

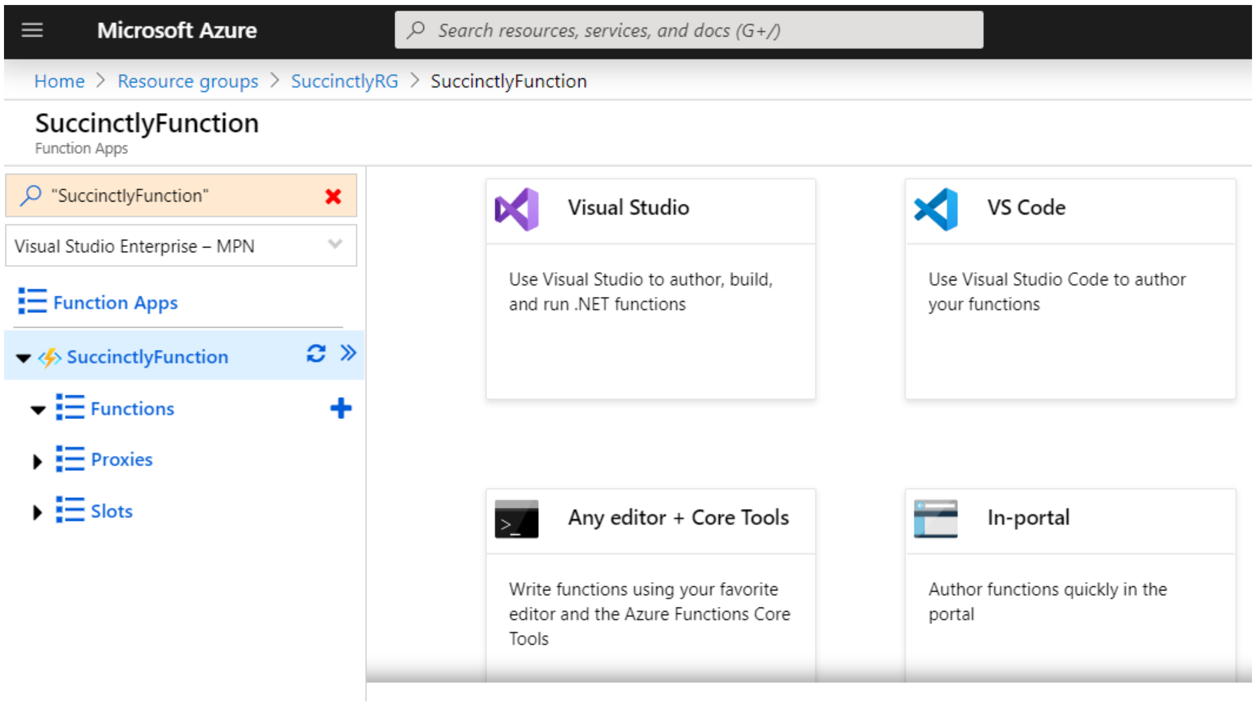

Microsoft Azure
Search resources, services, and docs (G+/)
Home > Resource groups > SuccinctlyRG > SuccinctlyFunction
SuccinctlyFunction
Function Apps
"SuccinctlyFunction"
Visual Studio Enterprise – MPN
Function Apps
SuccinctlyFunction
Functions
Proxies
Slots
Visual Studio
Use Visual Studio to author, build, and run .NET functions
VS Code
Use Visual Studio Code to author your functions
Any editor + Core Tools
Write functions using your favorite editor and the Azure Functions Core Tools
In-portal
Author functions quickly in the portal

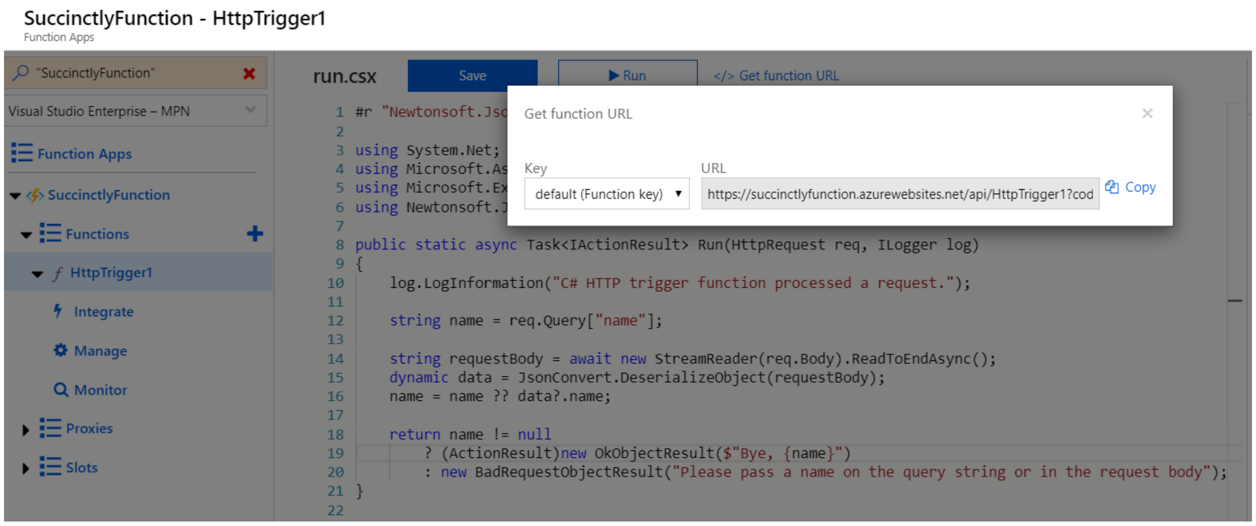
Figure 4: Creating an Azure Function

Here we can choose how we want to write our function, using Visual Studio or Visual Studio Code (the open source, lightweight, multiplatform code editor from Microsoft), any other editor, or in-portal. We're going to use the **In-portal** option, so we don't need any editors right now. Choose **In-portal** and click **Continue** at the bottom of the page.

Next, we need to select a template. There are two default templates to choose from: **Webhook + API** and **Timer**, but we can choose **More templates**. Here we find Azure Queue Storage trigger, Azure Service Bus Queue and Topic triggers, Blob Storage trigger, Event Hub trigger, Cosmos DB trigger, Durable Functions, and more. If you click on **More templates**, you can choose the **Webhook + API** template or the **HTTP trigger** template (which is the same). If you click on the first, you'll immediately get a function named HttpTrigger1. If you click the latter, you get to choose a name and authorization level (leave the defaults).

This should leave you with a function named HttpTrigger1 and some default sample code that echoes a name either from a URL parameter or a request body. You can test the function right away, which lets you choose **GET** or **POST** for the HTTP method. You can add headers and query parameters, and there's a default request body. The default will print **Hello, Azure**, as you would expect from the code. Try changing the code a bit, for example by changing **Hello** to **Bye**, and see that it works by running the test again.

Another way to test this is by getting the URL and pasting it in your browser. You can get the URL at the top by clicking **</> Get function URL**. It will look like: https://[your-app-service-name].azurewebsites.net/api/HttpTrigger1?code=[code].



```
#r "Newtonsoft.Jsc

using System.Net;
using Microsoft.As
using Microsoft.Ex
using Newtonsoft.J

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

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

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

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

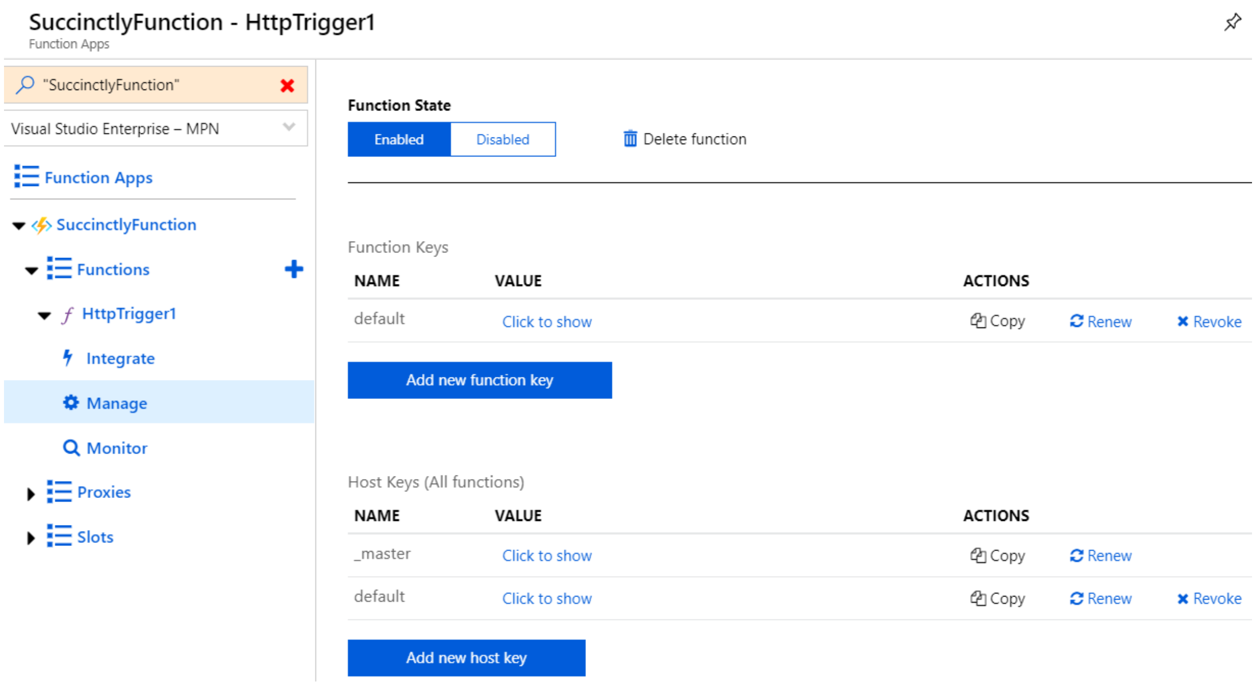
Figure 5: Testing an Azure Function

If you try it out, you'll get an error saying a name should be provided, so simply add **&name=Azure** at the end of the URL, and it will work again. That's it: you can now change the function as you please, and see it work. The sample shows how to read query and body parameters. Including packages, like Newtonsoft.Json, is done by using **#r** and the package name. Other than that, it's just C# code. The **`HttpRequest`** and **`ILogger`** arguments are injected by the Azure Functions runtime, but we'll see more of that later. Let's first dive into a bit of security for your functions.

Securing your HTTP functions

When you got the URL from your function, you may have noticed the **code=[code]** part. If you created the function from the **HTTP trigger** template rather than the **Webhook + API** template, you could even choose an authorization level. Basically, your function is public to anyone, and the only way to limit access is by securing it with a key. There are three kinds of keys: function keys, host keys, and master keys. There are also three levels of authorization: function, admin, and anonymous.

The authorization level of a function can be changed under the **Integrate** menu option of a function; we'll look at that in a minute. Function keys can be managed from the **Manage** menu option under a function.


SuccinctlyFunction - HttpTrigger1
Function Apps

Function State

Function Keys

NAME	VALUE	ACTIONS		
default	Click to show	Copy	Renew	Revoke

Host Keys (All functions)

NAME	VALUE	ACTIONS		
_master	Click to show	Copy	Renew	
default	Click to show	Copy	Renew	Revoke

Figure 6: Managing Function Keys

The default authorization level is **function**, which means you need any of the three keys to access the function. The function key is specific to the function. The host key can be used to access any function in the function app. Function and host keys can be created, renewed, and revoked. The master key cannot be revoked or created, but it can be renewed.

The **admin** authorization level needs the master key for access. Function and host keys cannot be used on this level. Function keys can be shared with other parties that need to connect to your function, as can host keys. Giving out the master key is a very bad idea—it cannot be revoked, and there is only one.

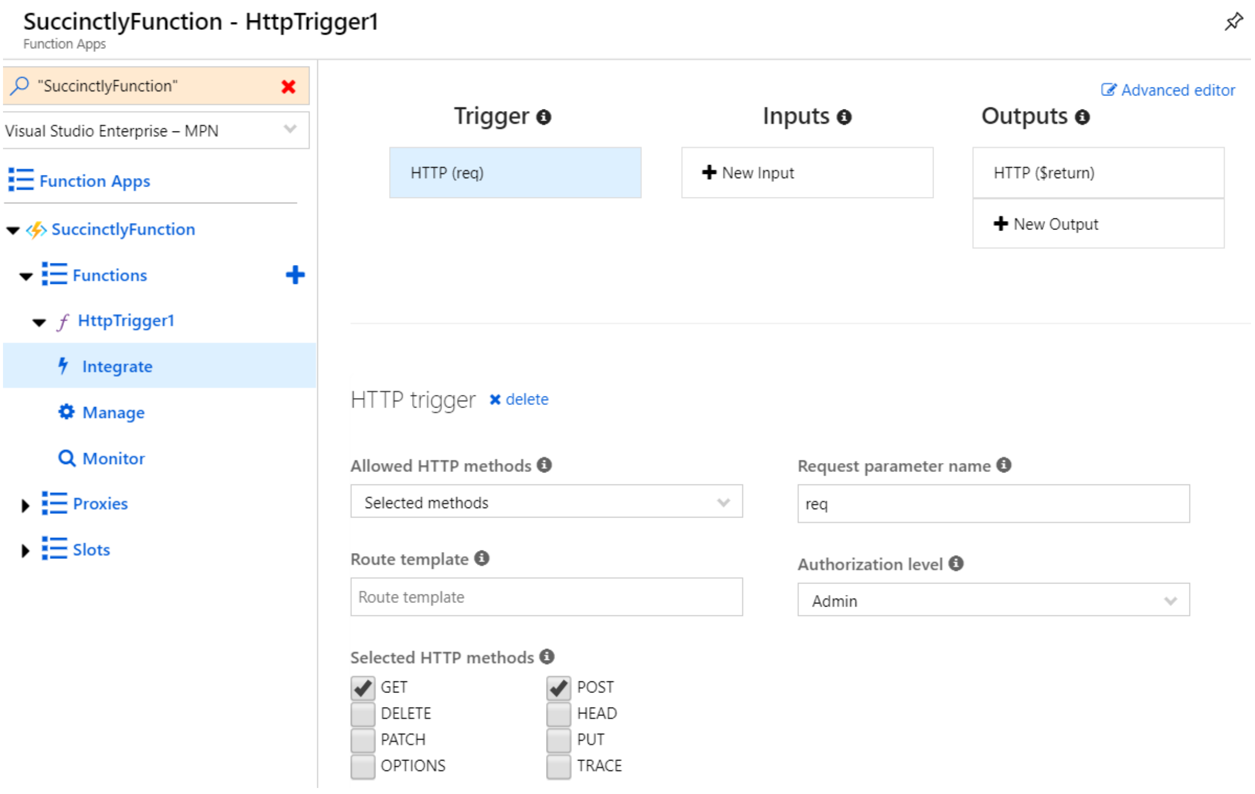
For example, say the Contoso and Acme companies need access to your function app. You can create two new host keys and give each one of them to one of the companies. If access needs to be revoked for one of the companies, you can simply revoke its key. Likewise, if any of the keys get compromised, you can renew it for that company. If a company only needs access to one or two functions, you can hand out the specific function keys.

The third authorization level is **anonymous**, and it simply means no keys are required and everyone with the URL can access your function.

Authorization levels are important for HTTP triggers. Other trigger types, such as timers, can never be triggered externally. Those functions do not have function keys, but they still have host and master keys because those are not specific for a function.

Triggers and bindings

An important aspect of functions is triggers and bindings. We've seen some of the trigger templates, like the HTTP trigger, the timer trigger, and the Service Bus trigger. Next to triggers, a function can have bindings that translate function input to .NET Core types, or output to input for other Azure services, such as blob storage, table storage, Service Bus, or Cosmos DB. The output of a function, an `IObjectResult` in the default example, is another example of a binding. A trigger is a specific sort of binding that differs from input or output bindings in that you can only have one. You can find triggers and bindings within the **Integrate** menu option under your function.



Allowed HTTP methods: Selected methods
Request parameter name: req
Route template: Route template
Authorization level: Admin
Selected HTTP methods: [✓ GET] [✓ POST] DELETE HEAD PATCH PUT OPTIONS TRACE

Figure 7: Function Triggers and Bindings

On the **Integrate** screen, you can change the behavior of triggers. For example, you can change the authorization level or the allowed HTTP methods. In the case of a timer trigger, you can change the CRON expression, which indicates at what times or intervals the function triggers. For Cosmos DB triggers, you can specify what collection and database name you want to connect to. Every trigger has its own configuration.

Output bindings

Let's start by adding an output binding, since that's the easiest one to work with. You'll notice that the return value is an output binding as well. If you click it, you'll notice that **Use function return value** is checked. This can only be checked for one output binding per function.

Let’s create a new output binding. In the **Integrate** window, click **+ New Output** and select **Azure Blob Storage**. You have to select it and then scroll down with the *outer* scroll bar to click **Select**. (It sounds simple, but it’s not obvious on small screens.) On the next screen, you probably need to install an extension, so simply click **Install**. After that, click **Save**, and the output binding will be created. The defaults for the binding dictate that the contents of the blob are specified in an output parameter named **`outputBlob`**, and the blob itself is created in the same storage account as the function in a container named **`outcontainer`**, with a random GUID as a name. We now need to change the code of the function so that it sets the **outputBlob** parameter.

Code Listing 1

```
#r "Newtonsoft.Json"

[…]

public static IActionResult Run(HttpRequest req, out string outputBlob,
ILogger log)
{
    […]
    outputBlob = $"Hello, {name}";

    return name != null
        […]
}
```

If you now run the function and then head to your storage account, you’ll notice a container with a file that has the contents **`Hello, [name]`**.

We’ll use output bindings in Visual Studio later, which has a couple of ways to define them. The main point to take away is that there’s a binding that binds, or converts, a string to a blob in a storage account.

Input bindings

Next is the input binding. This works pretty much the same. Click **+ New Input** and select **Azure Blob Storage**. To make this more real-worldly, we’re going to change the **Path** property to **incontainer/{customer}.txt**. Click **Save**. Now, head over to your trigger and add **{customer}** to your **Route template**. The URL for your function is now https://[your-app-service-name].azurewebsites.net/api/{customer}?code=[code]&name=[name]. Now refresh your browser with **F5**, or the changes won’t be visible to the debugger.

Open your code again and make sure you get the input blob as an argument to your function.

Code Listing 2

```
#r "Newtonsoft.Json"
```

```
[…]

public static IActionResult Run(HttpRequest req, string inputBlob, out
string outputBlob, ILogger log)
{
    […]

    return name != null
        ? (ActionResult)new OkObjectResult($"Bye, {name} ({inputBlob})")
        […]
}
```

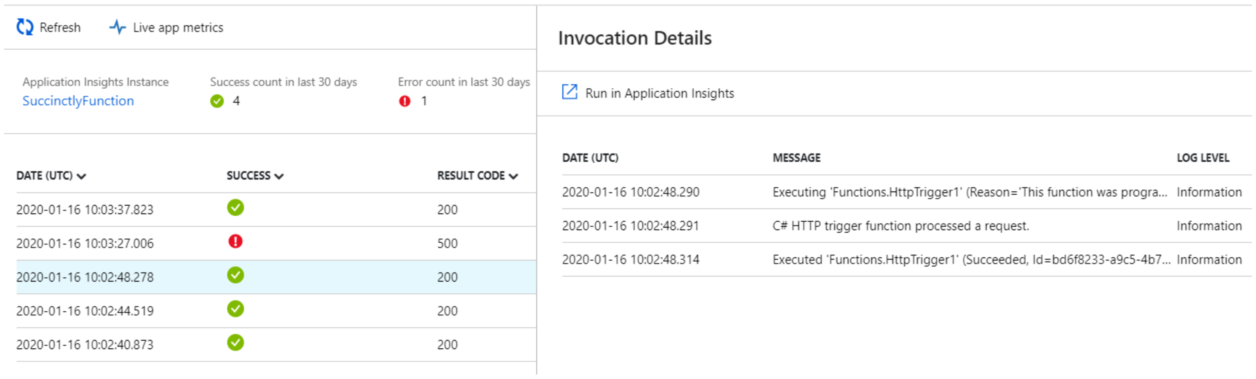
Next, make sure the input blob is available in your storage account. Go to your storage account and create a new container named **incontainer**. On your local computer, create a file named **customer1.txt** and add some text to it. Upload it to your container. Now, head back to your function and test it. Make sure you add **`customer`** to your query with the value of **`customer1`**. This should load **customer1.txt** from **incontainer** and pass its contents to **`inputBlob`** in your function. If all went well, you should see the text inside your text file printed to the output.

Using input and output parameters, it requires almost no code to transform string (or sometimes slightly more difficult classes) to blobs, records in table storage or Cosmos DB, messages on queues, or even emails and SMS messages. We'll see more of these bindings in a later example in Visual Studio.

Monitoring

A very important part of any application, and far too often an afterthought, is logging and monitoring. While regular app services have diagnostics logs, these are disabled for functions. In the code sample we already saw an **`ILogger`** and a call to **`log.LogInformation`**, but this isn't the **`ILogger<T>`** that you are probably using in your ASP.NET Core applications. Even with functions, you can still use the log stream and alerts and metrics options of your app service, though.

You can monitor your functions by using the **Monitor** menu option. Functions use Application Insights for monitoring, and as such, you can get very detailed information about your functions and individual invocations. When you open the **Monitor** blade, you can see all function invocations of the last 30 days with a slight delay of five minutes. Clicking an invocation shows some details about it. The information in your code's **`log.LogInformation`** is shown here as well. The **`ILogger`** can be used to log messages with various log levels to this blade. An error is shown when your functions throw an exception and you don't catch it in the function. This is important, because it means that if you write a try-catch block and don't re-throw your exception, the function will always succeed, while that may not actually be the case.


Refresh Live app metrics

Application Insights Instance SuccinctlyFunction | Success count in last 30 days ✓ 4 | Error count in last 30 days ● 1

DATE (UTC)	SUCCESS	RESULT CODE
2020-01-16 10:03:37.823	✓	200
2020-01-16 10:03:27.006	●	500
2020-01-16 10:02:48.278	✓	200
2020-01-16 10:02:44.519	✓	200
2020-01-16 10:02:40.873	✓	200

Invocation Details

Run in Application Insights

DATE (UTC)	MESSAGE	LOG LEVEL
2020-01-16 10:02:48.290	Executing 'Functions.HttpTrigger1' (Reason='This function was progra...	Information
2020-01-16 10:02:48.291	C# HTTP trigger function processed a request.	Information
2020-01-16 10:02:48.314	Executed 'Functions.HttpTrigger1' (Succeeded, Id=bd6f8233-a9c5-4b7...	Information

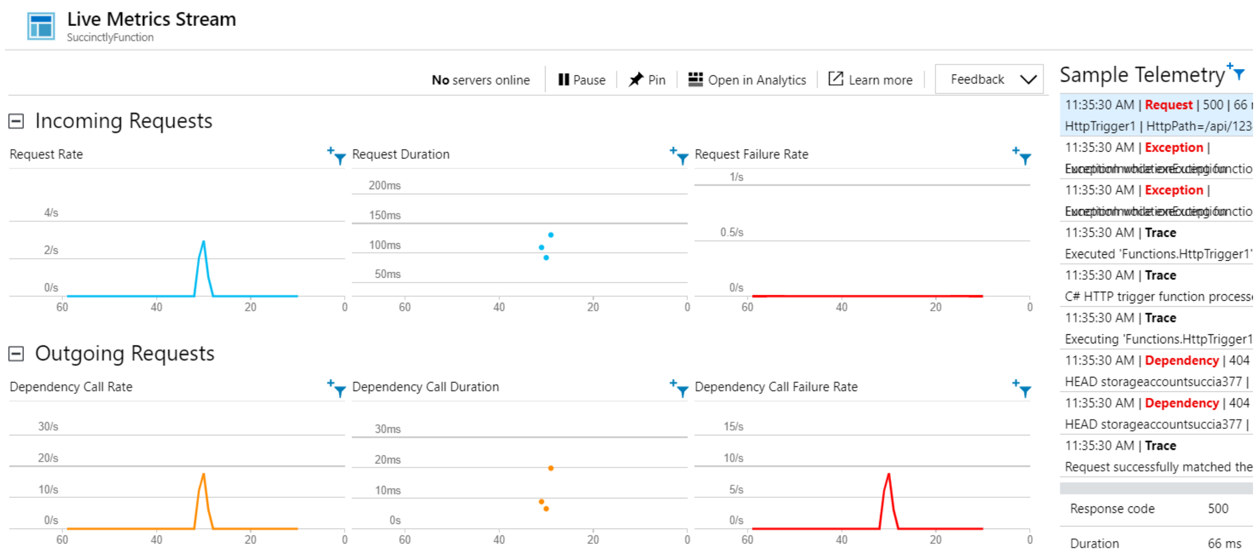
Figure 8: Monitoring in Azure Functions

By clicking **Run in Application Insights**, you can go further back or narrow down your search criteria. Application Insights works with a SQL-like syntax named Kusto Query Language (KQL), and the blade that opens gives you a couple of tools to help you write your queries. KQL is not in scope of this book, but you should be able to figure out some basics. For example, the query for the last 60 days is made by simply editing the default query to 60 days.

Code Listing 3

```
requests
| project timestamp, id, operation_Name, success, resultCode, duration,
operation_Id, cloud_RoleName,
invocationId=customDimensions['InvocationId']
| where timestamp > ago(60d)
| where cloud_RoleName =~ 'SuccinctlyFunction' and operation_Name ==
'HttpTrigger1'
| order by timestamp desc
| take 20
```

It is also possible to get live feedback from your function while it is running. Go back to the **Monitor** blade of your function and click **Live app metrics** to get live feedback. This is especially useful if someone is using your function right now but isn't getting the expected result. The blade that opens is somewhat big and convoluted, but it shows a lot of information, and you should be able to find what you need. Especially when you're trying to debug a single request, this is indispensable.



```
11:35:30 AM | Request | 500 | 66 
HttpTrigger1 | HttpPath=/api/123
11:35:30 AM | Exception |
Exception while executing functio
11:35:30 AM | Exception |
Exception while executing functio
11:35:30 AM | Trace
Executed 'Functions.HttpTrigger1'
11:35:30 AM | Trace
C# HTTP trigger function process
11:35:30 AM | Trace
Executing 'Functions.HttpTrigger1
11:35:30 AM | Dependency | 404
HEAD storageaccountsuccia377 |
11:35:30 AM | Dependency | 404
HEAD storageaccountsuccia377 |
11:35:30 AM | Trace
Request successfully matched the
```

Response code	500
Duration	66 ms

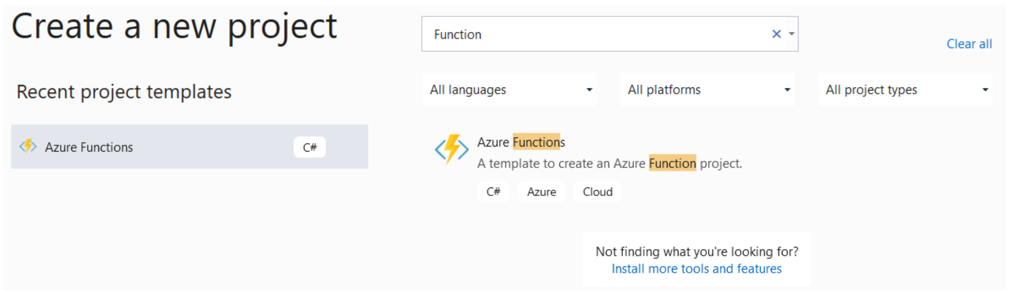
Figure 9: Live Metrics Stream

Application Insights offers a lot of tools for monitoring, but they are not within the scope of this book. However, it is possible to monitor availability, failures, and performance; to see an application map that can show interaction with other components such as databases, storage accounts, and other functions; and to send notifications, such as emails, on failures.

Creating a function using Visual Studio

Now that we've seen functions in the portal, we're going to write some functions in Visual Studio. While the portal has its merits—it's easy and a low barrier to entry—it's hard to write any serious code, and you can't utilize tools such as source control and continuous integration. I'm using the latest version of Visual Studio 2019 Community Edition (which you can download here) with .NET Core 3.1 (which you can download here), but the examples should work with Visual Studio 2017 and .NET Core 2.1 as well. When installing Visual Studio, you can select **.NET Core workload**, and .NET Core will be installed for you. In order to be able to develop functions using Visual Studio, you need to select the **Azure development** tools in the Visual Studio installer.

Once you've got everything set up, open Visual Studio, create a new project, and search for **Function**. This should bring up the Azure Functions project template. If you can't find it, you can click **Install more tools and features**, which opens the installer and allows you to install the Azure development tools.


Create a new project

Function

Clear all

Recent project templates

All languages | All platforms | All project types

Azure Functions C#

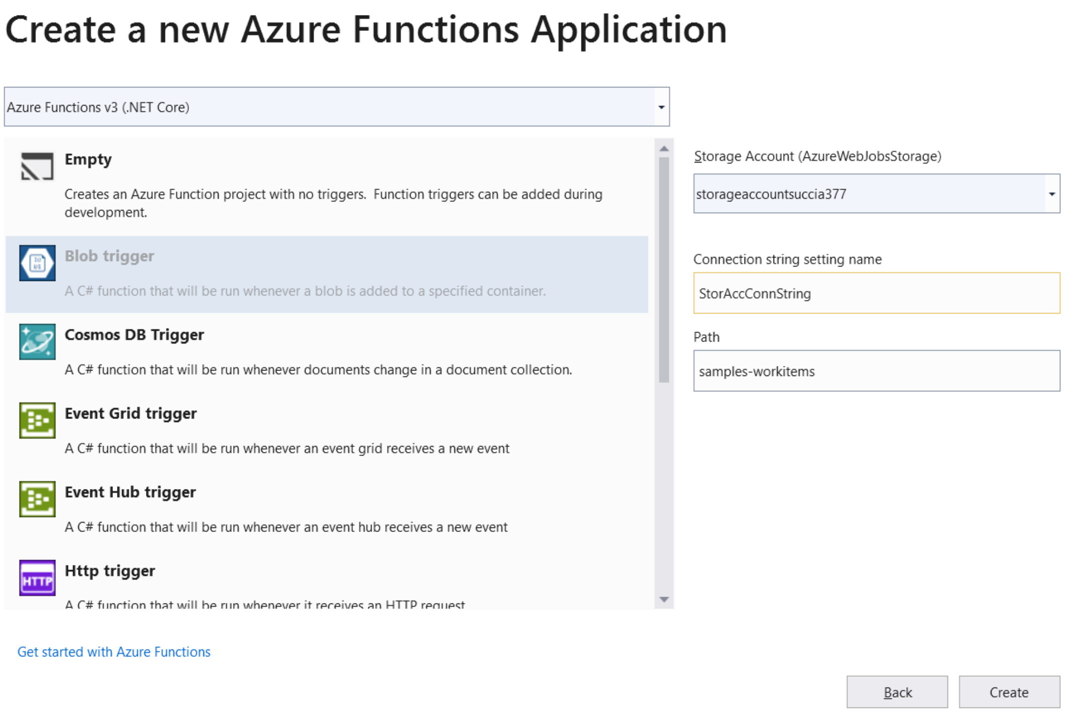
Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

Not finding what you're looking for?
Install more tools and features

Figure 10: Creating an Azure Functions Project in Visual Studio 2019

Once you click the **Azure Functions** project template, Visual Studio will ask you for a project and solution name, like you're used to. I've named mine **SuccinctlyFunctionApp**, but you can name yours whatever you like. Once you click **Create**, you can choose your runtime: Azure Functions v1 for the .NET Framework, or Azure Functions v2 and v3 for .NET Core. I'm going for **Azure Functions v3 (.NET Core).** You also have to pick a trigger, like we had to do in the portal. Let's choose the **Blob trigger** option this time. On the right side, you have to choose a storage account, either an actual Azure storage account or a storage emulator. This is the storage account for your function to be stored, not the storage account of the blobs that will trigger your function. The emulator, unfortunately, does not support every trigger. It's fine for a timer trigger, but not for our blob trigger. So, click **Browse** instead, and select the storage account that you used in the previous examples.

We must also enter a connection string name and a path, which will contain the connection string of the storage account that will trigger our function, and the path within that storage account that will be monitored for new blobs. Name the connection string **StorAccConnString** and leave the path at the default **samples-workitems**. Then, click **Create**.


Create a new Azure Functions Application

Azure Functions v3 (.NET Core)

Empty
Creates an Azure Function project with no triggers. Function triggers can be added during development.

Blob trigger
A C# function that will be run whenever a blob is added to a specified container.

Cosmos DB Trigger
A C# function that will be run whenever documents change in a document collection.

Event Grid trigger
A C# function that will be run whenever an event grid receives a new event

Event Hub trigger
A C# function that will be run whenever an event hub receives a new event

Http trigger
A C# function that will be run whenever it receives an HTTP request

Storage Account (AzureWebJobsStorage): storageaccountsuccia377

Connection string setting name: StorAccConnString

Path: samples-workitems

Get started with Azure Functions

Back | Create

Figure 11: Configuring an Azure Functions Project in Visual Studio 2019

When you try to run the generated sample code, you'll get an error that the **`StorAccConnString`** is not configured. Open the **local.settings.json** file, copy the **`AzureWebJobsStorage`** line, and change it to **`StorAccConnString`**. Then head over to the storage account in the Azure portal, go to your blob storage, and create a new container and name it **`samples-workitems`**. Run your sample function and upload any random file to the container. Your function should now trigger, and it should print something like:

```
C# Blob trigger function Processed blob
Name: dummy.txt
Size: 10 Bytes
```

Tip: You'll notice that when you stop your function using Visual Studio, the debug window doesn't close. There's an option in Visual Studio, under *Tools* ***>*** *Options* ***>*** *Debugging* ***>*** *General****, to*** *Automatically close the console when debugging stops****. It seems to work in the latest version of Visual Studio, but in the past, I've had problems with this option. If your breakpoints aren't hit during debugging, make sure this option is turned off.***

There are a few things to notice in the example. First, the blob is represented in code as a stream. Second, the **`blobPath`** variable has a **`{name}`** variable, which is also passed to the function. The name is handy, but not mandatory.

Let's say we upload text files to the blob storage, and we only want to know the contents of the file. I already mentioned that many triggers can translate to various C# objects. Instead of the stream, we can have the file represented as a string. Change the type of **`myBlob`** to **`string`** and remove the **`name`** variable. Instead of logging the name and the length, log **`myBlob`** directly. The code should now look as follows.

Code Listing 4

```
[FunctionName("Function1")]
public static void Run([BlobTrigger("samples-
workitems/{name}", Connection = "StorAccConnString")]string myBlob, ILogger
 log)
{
    log.LogInformation($"File contents: {myBlob}");
}
```

If you now upload a simple text file to the blob container, the function will trigger and print the contents. If you upload an image or something that doesn't contain text, it still tries to convert the bytes to text, and ends up with some weird string like **IDATx??• L#O??.**

Tip: A blob can be represented by stream, TextReader, string, byte[], a Plain Old CLR Object (or POCO) serializable as JSON, and various ICloudBlobs. The different triggers all have various code representations, and it's not obvious from looking at

the code or the trigger. The documentation is pretty good though, so be sure to check it. For example, you can find the various usages of the blob trigger here.

Bindings in code

The Azure portal was a bit limited in bindings options. For example, it wasn't possible to create multiple bindings to the **return** statement. For the next example we're going to quickly create a Service Bus. We're going to look at the Service Bus in more depth in Chapter 4.

Go to the Azure portal, create a new resource, and search for **Service Bus**. In the Service Bus creation blade, enter a name that's unique for Azure, for example **SuccinctlyBus**. Choose the **Basic** pricing tier, your subscription, resource group, and location. Once the Service Bus is created, look it up, find **Queues** in the left-hand menu, and create a new queue. Name it **myqueue** and leave all the defaults. When the queue is created, click it, and it will show 0 active messages. Next, find your **Shared access policies** in the **Service Bus** menu, click **RootManageShareAccessKey**, and copy one of the connection strings. We'll need it in a minute.

Next, go back to the code in Visual Studio. The first thing we need to do is install the **Microsoft.Azure.WebJobs.Extensions.ServiceBus** package using NuGet. Now, go to the **local.settings.json** file and create a new setting under **StorAccConnString**, name it **BusConnString**, and copy the Service Bus connection string as its value.

We can now use the Service Bus binding in the code. Above the function, add a **ServiceBusAttribute** to the return value and pass to it **myqueue** as queue name, and **BusConnString** as connection. To make things more fun, we're going to add an additional output binding with the **BlobAttribute** so we can copy our incoming blob. You can add this attribute to the return value as well, and it will compile, but only one attribute will work. The function should now look as follows.

Code Listing 5

```
[FunctionName("Function1")]
[return: ServiceBus("myqueue", Connection = "BusConnString")]
public static string Run([BlobTrigger("samples-
workitems/{name}", Connection = "StorAccConnString")]string myBlob,
    [Blob("samples-workitems-
copy/{name}_copy", FileAccess.Write, Connection = "StorAccConnString")]out
string outputBlob,
    ILogger log)
{
    log.LogInformation($"File contents: {myBlob}");
    outputBlob = myBlob;
    return myBlob;
}
```

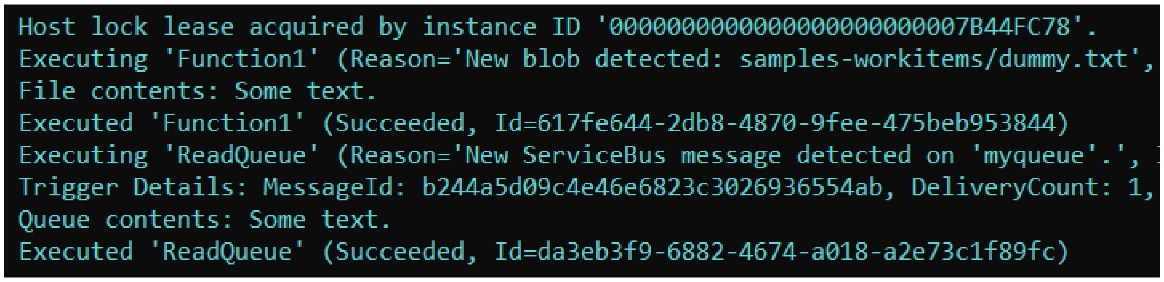
When you now upload a text file to the blob container, the function will trigger, and it will return the file contents as string. The **`BlobAttribute`** on the out parameter will now translate this string to a new blob with **`_copy`** appended to the original name: **`myfile.txt_copy`**. We have to do this in another container, or the copy will trigger the function, and we'll be stuck in an endless loop with **`_copy_copy_copy_copy`**. The Service Bus binding on the return value will put the string as a message on the **`myqueue`** queue that we just created. We can see this in the portal if we open the queue. It should now say there is one active message on the queue.

Now, let's create a new function that responds to the queue.

Code Listing 6

```
[FunctionName("ReadQueue")]
public static void DoWork([ServiceBusTrigger("myqueue", Connection = "BusCo
nnString")]string message, ILogger log)
{
    log.LogInformation($"Queue contents: {message}");
}
```

The **`ServiceBusTrigger`** looks the same as the **`ServiceBus`** binding to the return value. We specify a queue name and a connection name. The variable can be a string, a byte array, a message, or a custom type if the message contains JSON. The **`FunctionName`** attribute is used to specify the function's display name in the Azure portal. When you place a blob in the container, you can see that the first function will read it, create a copy, and put a message on the queue. The second function will trigger instantly and log the message.



```
Host lock lease acquired by instance ID '0000000000000000000000007B44FC78'.
Executing 'Function1' (Reason='New blob detected: samples-workitems/dummy.txt',
File contents: Some text.
Executed 'Function1' (Succeeded, Id=617fe644-2db8-4870-9fee-475beb953844)
Executing 'ReadQueue' (Reason='New ServiceBus message detected on 'myqueue'.',
Trigger Details: MessageId: b244a5d09c4e46e6823c3026936554ab, DeliveryCount: 1,
Queue contents: Some text.
Executed 'ReadQueue' (Succeeded, Id=da3eb3f9-6882-4674-a018-a2e73c1f89fc)
```

Figure 12: Read the Blob, Queue the Message, and Read the Bus

We're now going to change the first function a bit so that it will place a JSON string on the queue. We can then read it in the second function, and read the blob copy as input parameter. The first thing we need to do is create a class that we can write to and read from the queue.

Code Listing 7

```
public class BlobFile
{
    public string FileName { get; set; }
}
```

Next, we need to return a JSON string instead of the file contents in our first function. We only need to pass the name of the file to the queue because we can read the contents using an input parameter.

Code Listing 8

```
using Newtonsoft.Json;

[…]

log.LogInformation($"File contents: {myBlob}");
outputBlob = myBlob;
return JsonConvert.SerializeObject(new
{
    FileName = name
});
```

Now, in the second function, we can add an input parameter, bind the Service Bus trigger to **`BlobFile`**, and use the **`FileName`** property in the input parameter.

Code Listing 9

```
[FunctionName("ReadQueue")]
public static void DoWork([ServiceBusTrigger("myqueue", Connection = "BusCo
nnString")]BlobFile message,
    [Blob("samples-workitems-
copy/{FileName}_copy", FileAccess.Read, Connection = "StorAccConnString")]s
tring blob,
    ILogger log)
{
    log.LogInformation($"Queue contents: {message.FileName}");
    log.LogInformation($"Blob contents: {blob}");
}
```

When you now upload a text file to your blob container, the first function will read it, create a copy, and place a JSON message containing the file name on the queue. The second function then reads the message from the queue and uses the file name to read the copy blob, and passes its contents as an input parameter. You can then use the object from the queue and the blob from the storage account.

Dependency injection

When you're on a consumption plan, your app isn't running all the time, and you get a small startup delay if the function is first executed after it's gone idle. With that in mind, it's important to keep your startup time short. So far, we haven't seen any startup logic. We have some static functions in a static class, so the class can't have state or constructors. However, they don't have to be static. This is especially useful when you want to use dependency injection.

The first thing we need to do is create an interface and a class we want to inject.

Code Listing 10

```
public interface IMyInterface { string GetText(); }
public class MyClass : IMyInterface
{
    public string GetText() => "Hello from MyClass!";
}
```

Next, we need to install the **Microsoft.Azure.Functions.Extensions** package using NuGet. Once that is installed, we can create a new class that will have some startup logic.

Code Listing 11

```
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection;

[assembly: FunctionsStartup(typeof(SuccinctlyFunctionApp.Startup))]

namespace SuccinctlyFunctionApp
{
    public class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            builder.Services.AddTransient<IMyInterface, MyClass>();
        }
    }
}
```

The one thing to note here is the **assembly** attribute that specifies the **Startup** class. The class itself inherits from **FunctionsStartup** and overrides the **Configure** method, which lets you build your dependency injection container. It uses the default .NET Core DI libraries.

You can now remove all the static keywords from the **Function1** class and add a constructor that takes **IMyInterface** as argument.

Code Listing 12

```
public class Function1
{
    private readonly IMyInterface myInterface;

    public Function1(IMyInterface myInterface)
    {
        this.myInterface = myInterface;
    }

```

```
[…]
```

You can then use **`myInterface`** in any function in the **`Function1`** class.

Code Listing 13

```
log.LogInformation(myInterface.GetText());
```

It's perfectly valid to use DI in your function classes. However, you can't use state to "remember" values from one function to another. Due to the dynamic scaling of functions, you're never sure if your functions will run in the same instance. Write your functions as if they were static, except for some injected variables.

Deploying your function app

Once your function is done, you need to deploy it to Azure. There are various ways to deploy your app. The easiest method of deploying your function app is through Visual Studio. Another method is using Azure DevOps. While a complete discussion of Azure DevOps is not within the scope of this book, we're going to explore both methods of deployment in the following sections.

Deploy using Visual Studio

In Visual Studio, in the Solution Explorer, simply right-click on the solution that has your function app and click **Publish**. After that it's self-explanatory. You can choose the Consumption plan, Premium plan, or an App Service plan with Windows or Linux. You can also choose an existing Azure function or create a new one.

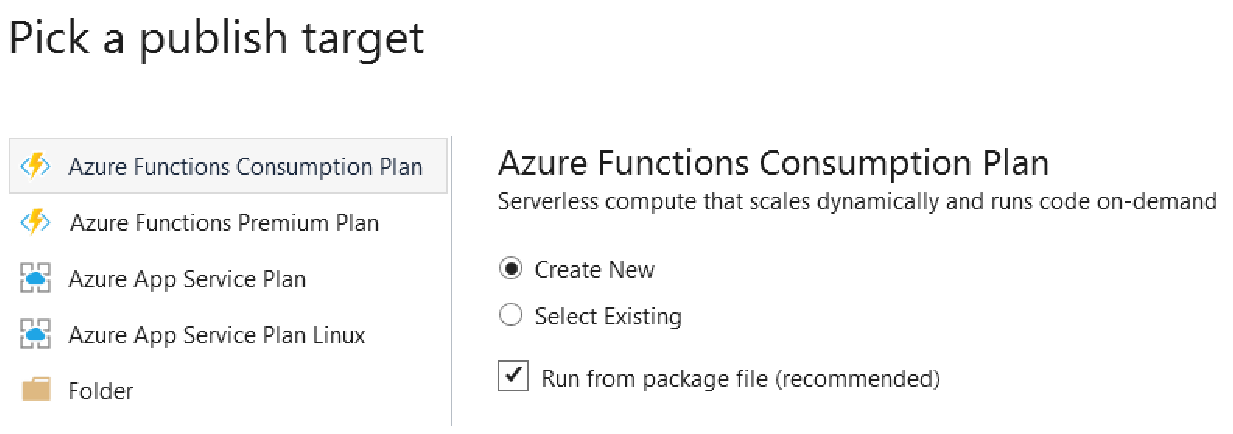
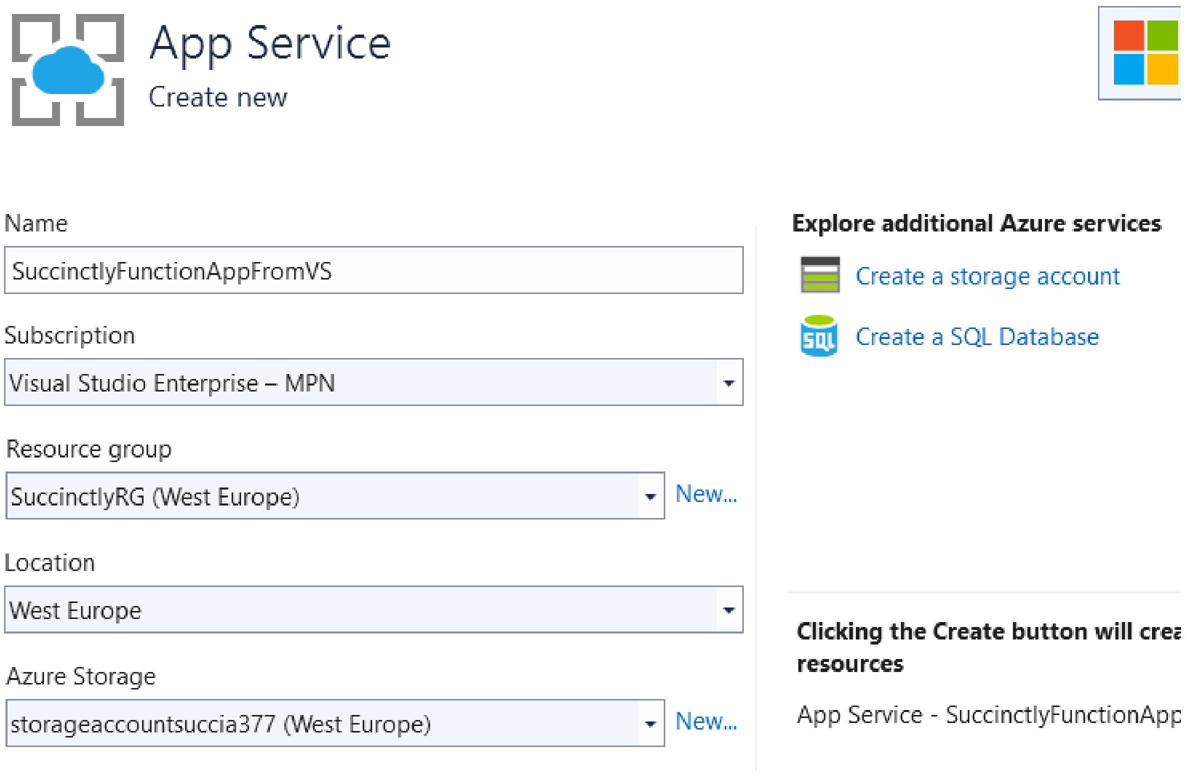

Pick a publish target
Azure Functions Consumption Plan
Azure Functions Premium Plan
Azure App Service Plan
Azure App Service Plan Linux
Folder
Azure Functions Consumption Plan
Serverless compute that scales dynamically and runs code on-demand
Create New
Select Existing
Run from package file (recommended)

Figure 13: Picking a Publish Target Using Visual Studio

The **Run from package file (recommended)** option needs some explanation. Normally, a web app would run files from the wwwroot folder. With **Run from package**, your web app is read from a zip file that is mounted as a read-only file drive on wwwroot. That means deploying is as easy as uploading a new zip file and pointing to it. You'll never have locked or conflicting files. That leads to atomicity and predictability: a function app is deployed at once, and all files are included. Versioning also becomes easier if you correctly version your zip files. Cold starts also become faster, which becomes noticeable if you run Node.js with thousands of files in npm modules. The downside to running from packages is that it only runs on Windows, so Linux is not supported, and your site becomes read-only, so you can't change it from the Azure portal.

Back to the actual deployment. We're going to create a new Azure function. I've had some bad experiences with overwriting an existing function with a new packaged function; it doesn't always work. So, you can create a publishing profile and fill out the data to create a new function app, and then just click **Create**.


App Service
Create new

Name
SuccinctlyFunctionAppFromVS

Subscription
Visual Studio Enterprise – MPN

Resource group
SuccinctlyRG (West Europe)

Location
West Europe

Azure Storage
storageaccountsuccia377 (West Europe)

Clicking the Create button will crea
resources

App Service - SuccinctlyFunctionApp

Figure 14: Creating a New Resource Using Visual Studio

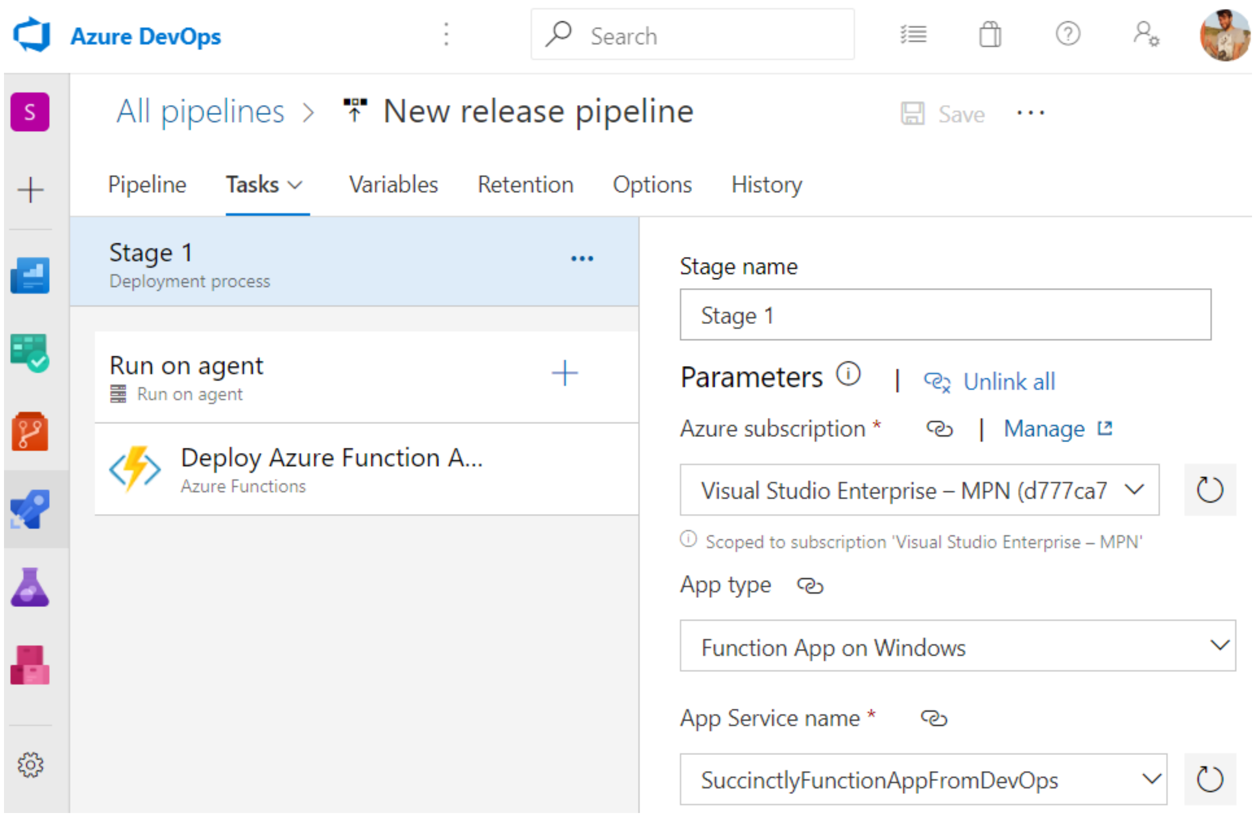
The plus side of using Visual Studio is, of course, that it's easy to deploy. The downside, however, is that you always have to deploy a local copy. When deploying software, you often need to do some additional work, like change some configuration, change a version, or run some tests. That means you must always remember to do those steps. What's more, you can now deploy from your machine, but a coworker may be using another editor, or might somehow be unable to build the software while they still need to be able to do deployments. Deployments may now suffer from the "it works on my machine" syndrome.

Deploy using Azure DevOps

As I mentioned, DevOps is not within the scope of this book, but I wanted to mention it and give you a quick how-to guide in case you're already familiar with Azure DevOps, so prior knowledge is assumed here. First, you need a Git repository in DevOps. Copy the solution to your repository and commit and push it to DevOps. For simplicity, be sure to add the **local.settings.json** file to source control (which is excluded from Git by default).

Next, you need to create a build pipeline in DevOps. I'm using the classic editor without YAML with the **Azure Functions for .NET** template. Pick the repository you just created and click **Save**. You shouldn't have to change anything.

The same goes for the release pipeline. Create a new pipeline and choose the **Deploy a function app to Azure Functions** pipeline. In this step you need an existing function app, so create one in the Azure portal. It is possible to create function apps from DevOps by using ARM templates (or PowerShell or the Azure CLI), but I won't cover that here. We'll see some ARM templates in the next chapter, though. Once the function app is created, you can return to DevOps and enter the necessary values.



Stage name: Stage 1

Azure subscription *: Visual Studio Enterprise – MPN (d777ca7

Scoped to subscription 'Visual Studio Enterprise – MPN'

App type: Function App on Windows

App Service name *: SuccinctlyFunctionAppFromDevOps

Figure 15: Deploying a Function App Using DevOps

Set the artifact to the latest version of the build pipeline you just created and save the pipeline. If you now create a new build and a new release, the function app should be released to Azure.

Note: Most resources in Azure can be deployed using ARM templates. In the next chapter, we'll see an example of an ARM template. When ARM isn't available for a certain resource or setting, you can use PowerShell or the Azure CLI. Sometimes it's easier (and safer) to manage resources manually, like users and access.

Durable functions

Durable functions are an extension to the Azure Functions runtime. They allow for stateful functions in a serverless environment. Durable functions can be written in C#, JavaScript in version 2, and F# in version 1, but the long-term goal is to support all languages. The typical use cases for durable functions are function chaining, fan-out/fan-in, async HTTP APIs, monitoring, human interaction, and aggregating event data. I'll briefly discuss some scenarios in the following sections.

The default template is for the async HTTP API scenario. You can initiate some process by doing a HTTP request. This returns a response with some URLs that can get you information about the initiated task. Meanwhile, the process is run asynchronously. With the provided URLs you can request the status of the process, the result if applicable, when it was created and last updated, or you can cancel it. You can read more about the various scenarios in the documentation.

Durable functions share the runtime of regular functions, but with a lot of new functionality, and restrictions as well. Discussing all possibilities and restrictions of durable functions could be a book on its own, but in the following sections we're going to look at some basics that should get you started. There is also plenty of official documentation on durable functions, so you should be good to go.

Creating a durable function

You can create durable functions in the Azure portal or in Visual Studio. When you're creating them in the portal, you'll need to install an extension, which you're prompted for, and which takes about two minutes. In the portal you'll find three sorts of durable functions: starter, activity, and orchestrator. In Visual Studio, these are created for you in a single file. We're going to use Visual Studio to create our durable functions.

Open Visual Studio, create a new solution, and choose the **Functions** template. Pick an empty function and browse for your Azure storage account. When the project is created, right-click the project and select **Add** > **New Azure Function**, pick a name (or leave the default), click **Add**, and then choose the **Durable Functions Orchestration** function. The functions that are created need some explanation.

There are three functions: **`Function1`**, **`Function1_Hello`**, and **`Function1_HttpStart`**. As the name suggests, **`Function1_HttpStart`** sets everything in motion. This creates a new context with an ID by calling **`starter.StartNewAsync(“Function1”, null)`**, and returns that ID to the caller. You can see this in action by starting the application, opening your browser, and browsing to **http://localhost:7071/api/Function1_HttpStart** (your port may vary; see the console output). This returns a JSON.

Code Listing 14

```
{
    "id": "1341f5df5d8149679acdbd5e0fd07043",
    "statusQueryGetUri": "[URL]",
    "sendEventPostUri": "[URL]",
    "terminatePostUri": "[URL]",
    "rewindPostUri": "[URL]",
    "purgeHistoryDeleteUri": "[URL]"
}
```

The next step, in code, is that **`Function1`** is executed asynchronously. This is the orchestrator function that determines the status and output of the function. You can get the status of the function by using the **`statusQueryGetUri`** URL from the response of **`Function1_HttpStart`**. If you’re fast enough, you’ll see the status **`Pending`** or **`Running`**, but you’ll probably see **`Completed`**. If the functions raise an exception, you’ll see status the **`Failed`**. Other than that, you’ll see the function name, ID, input, output, the creation time, and when it was last updated.

```
{
    "name": "Function1",
    "instanceId": "1341f5df5d8149679acdbd5e0fd07043",
    "runtimeStatus": "Completed",
    "input": null,
    "customStatus": null,
    "output": ["Hello Tokyo!", "Hello Seattle!", "Hello London!"],
    "createdTime": "2020-01-23T10:13:27Z",
    "lastUpdatedTime": "2020-01-23T10:14:09Z"
}
```

The actual work takes place in **`Function1_Hello`**, the activity function, which is called from **`Function1`** three times with **`CallActivityAsync`**. The result from **`Hello`** is added to the output, which is returned by the orchestrator function. You can test the asynchronous character a bit better by adding **`Thread.Sleep(3000)`**; to the activity function. This allows you to comfortably check out the status of the orchestrator.

Updating to v2

Unfortunately, the Durable Functions template is not up to date, and some manual work is required to get it to the latest. First, go to NuGet and update the **Microsoft.Azure.WebJobs.Extensions.DurableTask** package. Next, in your code, add a **using** statement for the **Microsoft.Azure.WebJobs.Extensions.DurableTask** namespace. Next, in the starter function, replace the **OrchestrationClient** attribute with the **DurableClient** attribute, and **DurableOrchestrationClient** with **IDurableClient** or **IDurableOrchestrationClient**. In the orchestrator function, replace **DurableOrchestrationContext** with **IDurableOrchestrationContext**. You could also replace the **Function1_Hello** string with a constant string to avoid typos or missing one if you ever decide to rename.

Version 2 has some new features like durable entities, which allow you to read and update small pieces of state, and durable HTTP, which allows you to call HTTP APIs directly from orchestrator functions and implement automatic client-side HTTP 202 status polling, and has built-in support for Azure managed identities.

Human interaction

Let’s look at another scenario: human interaction. A common scenario is an approval flow. An employee requests something, perhaps a day off, and a manager has to approve or deny the requests. After three days the request is automatically approved or denied. We can use events for this.

Let’s first create the starter function.

Code Listing 15

```
[FunctionName("StartApproval")]
public static async Task<HttpResponseMessage> StartApproval(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get")]HttpRequestMessage re
q,
    [DurableClient]IDurableOrchestrationClient client)
{
    string name = req.RequestUri.ParseQueryString()["name"];
    string instanceId = await client.StartNewAsync("ApprovalFlow", null, na
me);
    return client.CreateCheckStatusResponse(req, instanceId);
}
```

The only thing we do here is get a **name** variable from the request URL and pass it to the **ApprovalFlow** orchestrator function.

The orchestrator function is a bit of a beast. First, we create a cancellation token that we pass to a timer. The timer indicates when the approval request expires, which we set to a minute from the approval time. The real magic is in the **context.WaitForExternalEvent** function, paired with the **Task.WhenAny** function. **WaitForExternalEvent** returns a task that completes when the event is raised. **`Task.WhenAny`** waits until either the task from the external event returns or the timer task completes (meaning it goes off). If the **`approvalEvent`** completes first, we cancel the **`cancelToken`** so that the timer stops. Next, we check to see whether the approval event returned true (approved) or false (denied). When the timer completes first, the request is automatically denied. With **`context.GetInput<string>`** we can get the input that was passed to the function from the starter function, in this case the value of the **`name`** URL parameter.

Code Listing 16

```
[FunctionName("ApprovalFlow")]
public static async Task<object> ApprovalFlow([OrchestrationTrigger]IDurabl
eOrchestrationContext context)
{
    // Possibly write the request to a database here.
    //await context.CallActivityAsync("RequestToDb", context.GetInput<strin
g>());
    using (var cancelToken = new CancellationTokenSource())
    {
        DateTime dueTime = context.CurrentUtcDateTime.AddMinutes(1);
        Task durableTimeout = context.CreateTimer(dueTime, cancelToken.Toke
n);

        Task<bool> approvalEvent = context.WaitForExternalEvent<bool>("Appr
ovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTimeo
ut))
        {
            cancelToken.Cancel();
            if (approvalEvent.Result)
            {
                return await context.CallActivityAsync<object>("ApproveActi
vity", context.GetInput<string>());
            }
            else
            {
                return await context.CallActivityAsync<object>("DenyActivit
y", context.GetInput<string>());
            }
        }
        else
        {
            return await context.CallActivityAsync<object>("DenyActivity",
context.GetInput<string>());
        }
    }
}

[FunctionName("ApproveActivity")]
public static object ApproveActivity([ActivityTrigger] string name)
```

```
{
    // Probably update some record in the database here.
    return new { Name = name, Approved = true };
}

[FunctionName("DenyActivity")]
public static object DenyActivity([ActivityTrigger] string name)
{
    // Probably update some record in the database here.
    return new { Name = name, Approved = false };
}
```

The **ApproveActivity** and **DenyActivity** functions are the activity functions, and they simply return whether a request was approved or denied, but in a real-world scenario you'd probably do some updates in a database, and possibly send out some emails.

That leaves two more functions: one for approval, and one for denial. Or alternatively, one function with a parameter approved or denied. In any case, here's the function for approval.

Code Listing 17

```
[FunctionName("Approve")]
public static async Task<HttpResponseMessage> Approve(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get")]HttpRequestMessage re
q,
    [DurableClient]IDurableOrchestrationClient client)
{
    string instanceId = req.RequestUri.ParseQueryString()["instanceId"];
    await client.RaiseEventAsync(instanceId, "ApprovalEvent", true);
    return req.CreateResponse(HttpStatusCode.OK, "Approved");
}
```

For this to work, we need to provide the durable function's instance ID in the query parameters. With this instance ID, we can raise an event by using **client.RaiseEventAsync**. This will complete the approval event in the orchestrator function, so it approves the request.

You can test this by starting up the function app, browsing to the URL of the **StartApproval** function, and providing a name in the parameters, like **http://localhost:7071/api/StartApproval?name=Sander**. This returns a JSON with the **statusQueryGetUri** and the **id**. Open the get URI in a separate browser tab to see that the status is currently running. Next, browse to the URL for the approval function and supply the ID that's returned from the initial start: **http://localhost:7071/api/Approve?instanceId=[ID]**. Now, refresh the status tab and you should see the status is completed, the input was **Sander** (or whatever name you provided), the output JSON with a name, and whether the request was approved. If you wait longer than a minute to approve the request, it's denied.

Summary

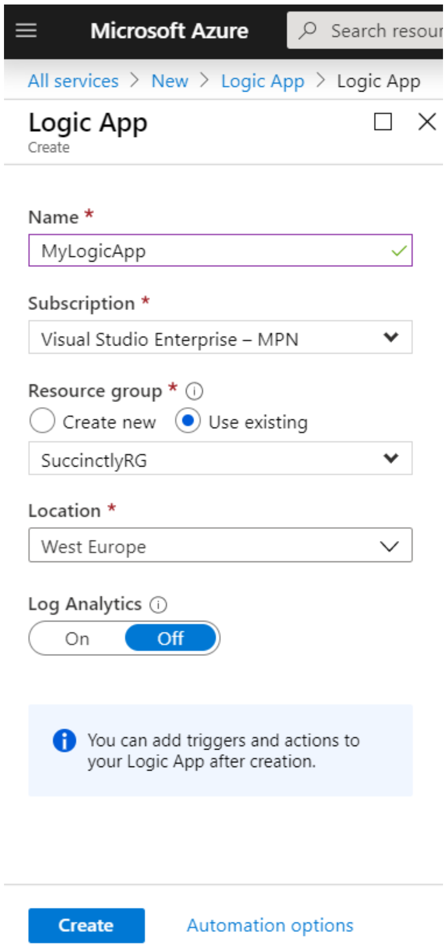
Azure Functions is a powerful tool in your toolbox. With numerous triggers and bindings, it integrates with your other Azure services almost seamlessly. With durable functions and keeping state, your serverless options greatly expand. There are limitations though, so functions aren’t always a good option. If you need a quick function, you can use the Azure portal, or if you’re building some serious, big, serverless solutions, you can use Visual Studio and Azure DevOps.

Chapter 3 Logic Apps

In the previous chapter, we explored Azure Functions in depth. We wrote functions using the Azure portal and using Visual Studio. We also deployed them using Visual Studio and Azure DevOps. While Azure functions are a great way to run some code on specific triggers, there is an easier way to create workflows. Azure Logic Apps provides a sort of no-code version of functions. Logic apps are serverless and are initiated using various triggers, just like functions. Logic apps can also trigger functions and other logic apps using HTTP. In this chapter we're going to build logic apps using the Azure portal.

Creating a logic app

To create a logic app, go to **Create a resource** or **All services** in the Azure portal, and search for **Logic App**. Click **Create** and fill out the blade that appears. You need a name, subscription, resource group, and location. Leave **Log Analytics** set to **Off**.



Name *
MyLogicApp

Subscription *
Visual Studio Enterprise – MPN

Resource group *
Create new [✓ Use existing]
SucinctlyRG

Location *
West Europe

Log Analytics
On [✓ Off]

You can add triggers and actions to your Logic App after creation.

Figure 16: Creating a Logic App

Once the logic app is created, go to the resource, and Azure will suggest some common triggers and templates, as well as an introduction video. You'll see some triggers that can also be used with functions, like when a message is received in a Service Bus queue, recurrence (timer), and when an HTTP request is received. You'll also see some new triggers, like when a new tweet is posted on Twitter, when a new email is received in Outlook.com, and when a new file is created on OneDrive. Since a logic app can trigger functions, this effectively allows you to use some new triggers for your functions. You'll also see some templates that contain entire workflows, like deleting old Azure blobs, daily email reminders, and even some advanced ones, like posting to Slack if a new tweet matches with some hashtag, and when a new member is added to a MailChimp list, asking me if I also want to add them to a SharePoint list. It's also possible to react to events from Azure DevOps, but they're not visible in the common triggers and templates.

For this demo we are going to use some triggers and actions that aren't available for functions. Pick a blank Logic App from the templates. This allows you to pick your own triggers in the Logic App designer. In the form that opens you'll see the designer. First, you'll need to pick a trigger. The default selection is **For You**, but you probably haven't used any triggers yet, and so you don't get any recommendations either. Select **All** instead, and marvel at the many triggers. This is where the Azure DevOps triggers are, as well as Gmail, Dropbox, Dynamics 365, Ethereum Blockchain, GitHub, Instagram, Jira, Salesforce, Trello, YouTube, and many others. We're a bit limited in options because I can't assume my readers have any of those services. We're going to use a SQL Server trigger, but before we can do that, we first have to create an Azure SQL Server instance.

Creating a SQL database

1. In **All services**, search for SQL database and choose **SQL databases**.
2. In the SQL database creation blade, pick a resource group and enter a database name.
3. Create a new server, which needs a name that is unique across Azure.
4. Enter a server admin login name (for example, your first name) and a password (for example **Welcome1!)**. It's just a temporary server, so it doesn't matter too much. Just be sure you remember both your username and password.
5. Under **Compute + storage**, choose the **Basic** tier.
6. Click **Review + create** and then **Create** to create the database.
7. Find the server in your Azure resources, go to **Firewalls and virtual networks** in the menu on the left, click **Add client IP**, and enable **Allow Azure services and resources to access this server**.
8. Click **Save**.

Note: If you're having trouble creating a database, skip to Chapter 6 where databases are explained in more detail. You may opt for a serverless database, but this demo was created using the Basic tier instead of Serverless.

Next, we need to create a table in the database. You can do this from either SQL Server Management Studio (SSMS) or from the query editor in Azure. You can find the query editor in the menu of the SQL database. Log in using **[myserver].database.windows.net** (when using SSMS) and your username and password. Then, create the table by running the following query.

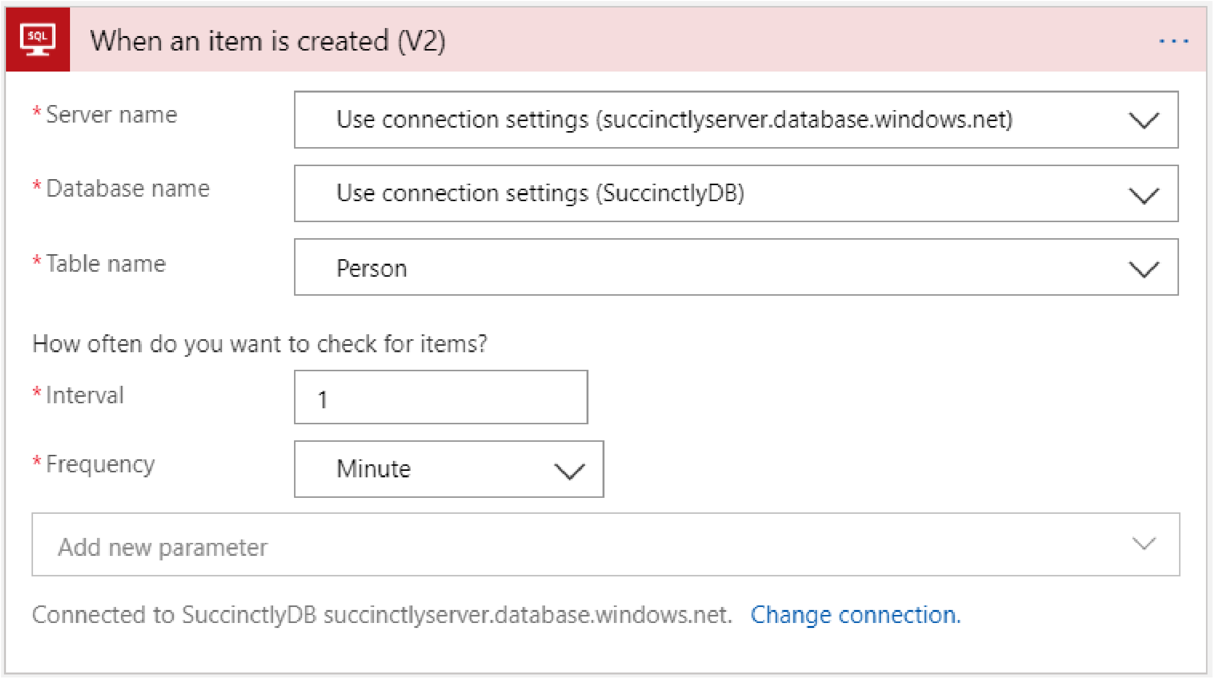
Code Listing 18

```
CREATE TABLE [dbo].[Person](
      [Id] [int] IDENTITY(1,1) NOT NULL,
      [Name] [varchar](50) NOT NULL,
 CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED
(
      [Id] ASC
)WITH (STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Next, we can go back to our logic app and add a trigger.

Building a logic app with SQL Server

Go back to the logic app and choose a blank template. In the designer, search for the SQL Server trigger and then select **When an item is created (V2)**. Next, you can specify your server, database, and table. Select the server, database, and table that we just created, and set the interval to **1**.



When an item is created (V2)

* Server name: Use connection settings (succinctlyserver.database.windows.net)

* Database name: Use connection settings (SuccinctlyDB)

* Table name: Person

How often do you want to check for items?

* Interval: 1

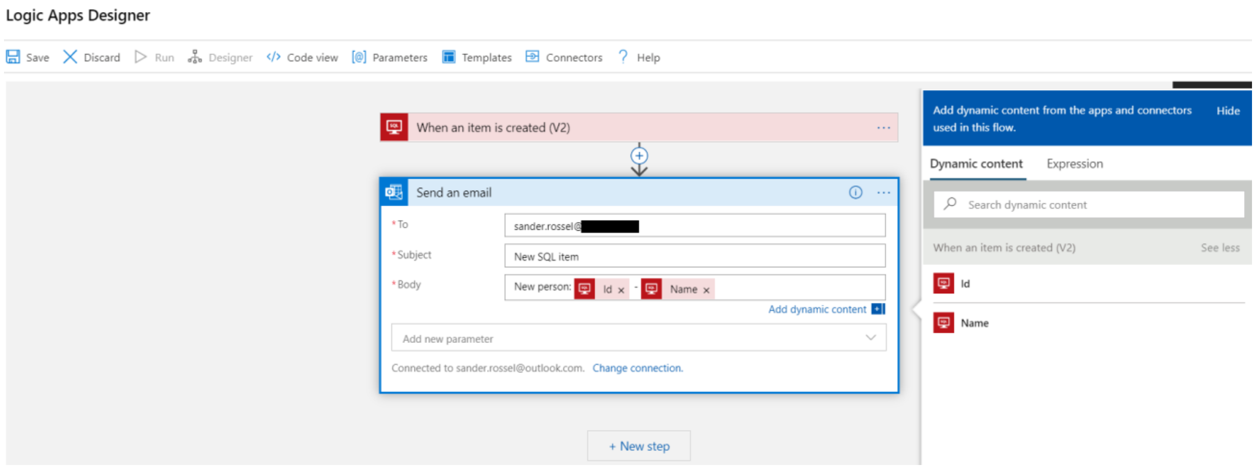
* Frequency: Minute

Add new parameter

Connected to SuccinctlyDB succinctlyserver.database.windows.net. Change connection.

Figure 17: Creating a SQL Trigger

After that, you can add a follow-up action. For this, we are going to select **Send an email** from **Outlook.com**. Alternatively, you can pick Gmail or one of the other email alternatives (**Send Email** from **SMTP** for a generic mail client). When you choose the **Outlook.com** option, you need to sign in, and after that you can simply add a To, a Subject, and a Body. You can use dynamic content from the previous actions, in our case the trigger. We can use this to get the inserted ID and name from the Person table.



*To: sander.rossel@[redacted]
*Subject: New SQL item
*Body: New person: Id - Name

Connected to sander.rossel@outlook.com. Change connection.

Figure 18: Configuring a Logic App

When the SQL trigger and the email action are configured, click **Save** in the upper-left corner to save the logic app.

After that, it's time to test the logic app. Go to SSMS or to the query editor in Azure and insert some names.

Code Listing 19

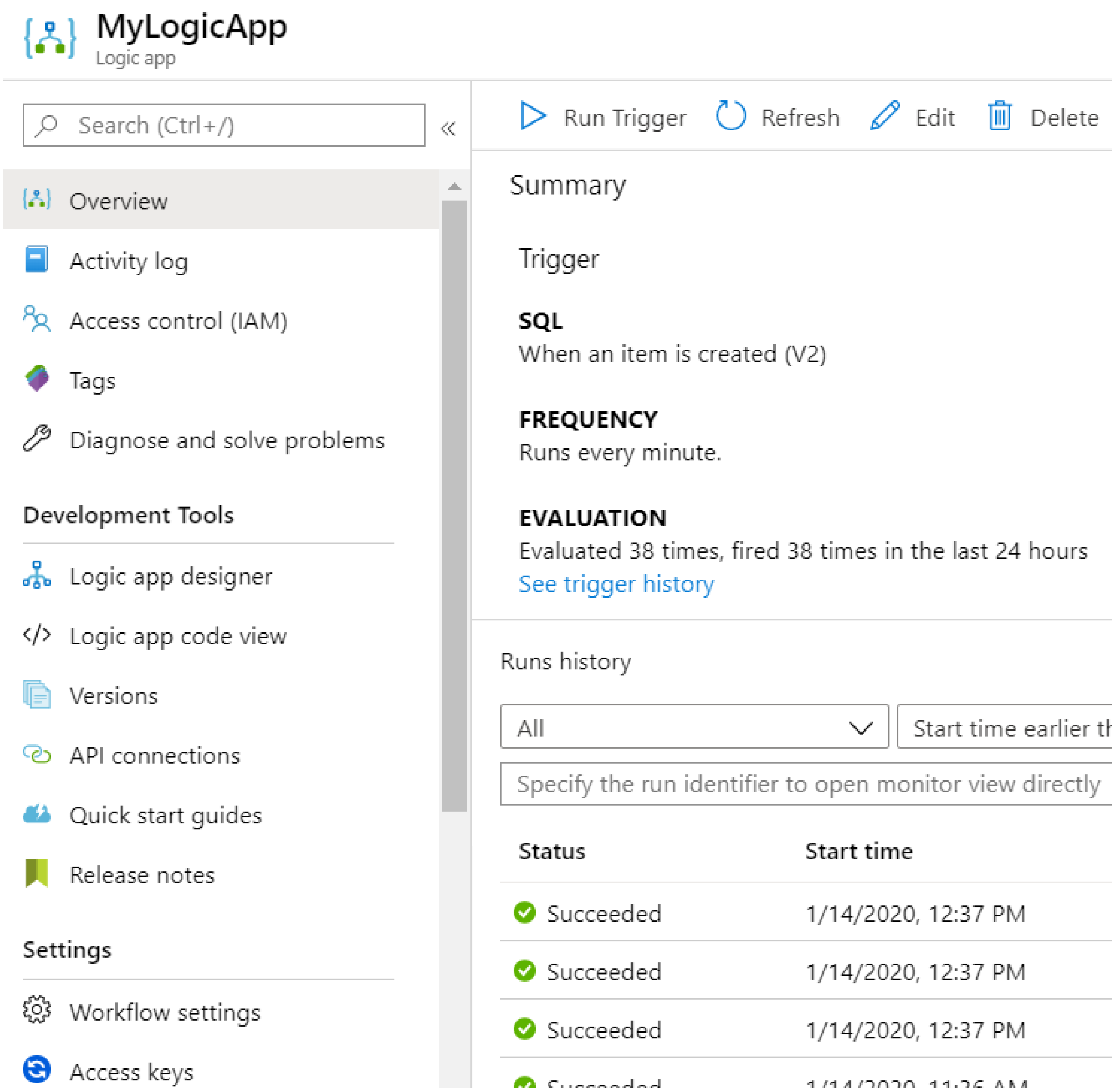
```sql
INSERT INTO Person ([Name])
VALUES ('Bill'),
       ('Steve'),
       ('Satya')
```

Wait a minute, and check if you get three emails with the inserted IDs and names.

Note: The logic app we just created makes use of two external connections: the database connection, and the Outlook connection. These are stored as separate resources called API connections, and can be found under *All services* ***if you search for*** *API connections*. ***API connections can be reused in other logic apps.***

Monitoring logic apps

You can monitor your logic apps in the **Overview** tab of the resource. This shows the type of trigger, the frequency, and how often it was triggered, along with the status of each run.



Summary

Trigger

SQL
When an item is created (V2)

FREQUENCY
Runs every minute.

EVALUATION
Evaluated 38 times, fired 38 times in the last 24 hours
See trigger history

Runs history

All
Start time earlier th
Specify the run identifier to open monitor view directly

Status	Start time
Succeeded	1/14/2020, 12:37 PM
Succeeded	1/14/2020, 12:37 PM
Succeeded	1/14/2020, 12:37 PM

Figure 19: Logic App Overview

It's possible to drill down on a run and see detailed logging. You can see the input and output of each step and whether the step succeeded. Even when you're using a **`ForEach`** step, you can see the status for each individual iteration. You can also resubmit specific runs here.

It is possible to be notified when a run fails, for example through email. This can be done through the **Alerts** function in the logic app menu. This works the same as for your other Azure resources.

Flow in logic apps

Let’s create another logic app to see some other steps that you can use. In the example, we’re going to create an HTTP trigger. Unfortunately, it’s not straightforward to get query parameters from your URL, so we’re going to create a `POST` request with a JSON body as input. To do this, create a new logic app and choose the **When a HTTP request is received** template.

In the request, you can specify a request body JSON schema. I strongly suggest you do this, because it allows you to directly use any values from the body in your next steps in the logic app. Click **Use sample payload to generate schema** and enter the following JSON.

Code Listing 20

```
{
    "firstName": "Sander",
    "lastName": "Rossel"
}
```

This will create the schema with `firstName` and `lastName` as strings in an object.

Note: If you really want to make `GET` ***requests, that’s possible. In your HTTP trigger, click*** *Add new parameter* **>** *Method* **>** *GET*. ***To access parameters from your URL query, use*** `@triggerOutputs()['queries']['firstName']`***, where you can now simply put in*** `firstName`***, and do the same for*** *lastName*.

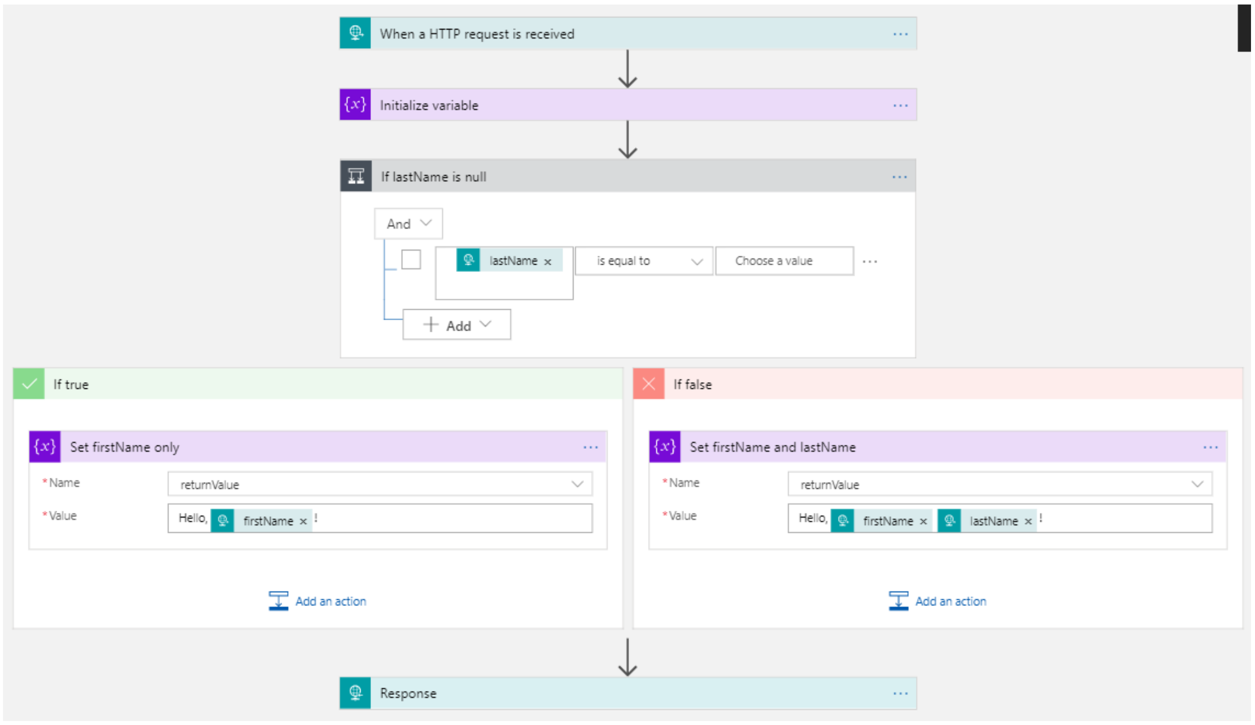
For the next action, we’re going to work with a variable. Add an action and look for the **Variables** action, and then choose **Initialize variable**. This action requires a name and a type, so enter **returnValue** and **String**, respectively. We’ll leave the (initial) value blank.

Add another action and look for **Control**. These are your standard programming control statements, such as if and foreach. Pick **Condition**, which is a standard if statement. This is where things get interesting. You’ll get three new boxes, a condition, an “If true,” and an “If false.” What we’re going to do is check for the value of `lastName`, and if it’s not set, we’re going to set the variable we declared in the previous step to `Hello, [firstName] [lastName]!`, but if it is, we’re going to use the `firstName` only. So, in the condition part, set **[lastName] [is equal to] [leave blank]** (just don’t type anything there). When you enter one of the fields, a pop-up menu allows you to choose some predefined values and functions. In our case we should see Body, Path Parameters, and Queries, which are added by our HTTP trigger, as well as the `firstName` and `lastName` values, which also come from the trigger because we added them. The `returnValue` can also be selected here. You can also click on the three dots in the upper-right corner of any action and rename your action. So instead of **Set variable** and **Set variable 2**, you can rename them to **Set firstName only** and **Set firstName and lastName**, for example. You can also set some additional settings here, but you probably won’t need them a lot.

Then in both the true and false conditions, we can add a **Set variable** action. These are simple. The variable we want to set is `returnValue` and the value we want to give it is either `Hello, [firstName]!` or `Hello, [firstName] [lastName]!`.

Both the true and false conditions can have multiple actions, which can form a complete workflow on its own. We need to break out of the `if` statement though, so we can just add another action at the bottom. Here, we'll pick the **Response** action. The status code is 200, and the body is `[returnValue]`.

If you save the logic app, your trigger will generate a URL that you can copy.



Set firstName only
*Name returnValue
*Value Hello, firstName !

Set firstName and lastName
*Name returnValue
*Value Hello, firstName lastName !

Figure 20: Logic App Condition

To test this, we need an application that can test **`POST`** requests. For this, you can use Postman. Postman is fairly straightforward for simple stuff. Simply copy the URL from your trigger, set the method to **POST**, and in the **Body** tab, set the type to **`raw`** and **`JSON`**. For the body, specify a JSON with a **`firstName`** and a **`lastName`**, and click **Send**. The request should now show **`Hello, [firstName]!`** or **`Hello, [firstName] [lastName]!`**, depending on whether you set a **`lastName`**.

Deploying logic apps

The strongest point of Azure Logic Apps—having a relatively easy designer that less technical people can work with—is also its weakest point. To build a logic app, you're basically just doing a lot of button clicks. That's hard to automate. So, whenever you're building a logic app in a test environment, and you want to bring it to a staging environment and then a production environment, you're going to have a bit of a challenge.

There are some options: an easy manual method, and a more complicated automated method. Unfortunately, neither work well with changes. Let's take our first example that works with an

Azure SQL database and an Outlook account, specifically because I want to point out some things there.

Cloning logic apps

The first way to move a logic app to the next environment is by cloning it. On the **Overview** page of your logic app, there's a **Clone** button in the top menu. You can specify a new name for the cloned logic app, as well as whether it should be enabled or disabled. Add something like **Staging** or **Prod** to the name, and clone it in the disabled state. It should take a few seconds to clone. Once it's cloned, go to the logic app's **Overview** page and change the **Resource group** of the logic app. This may take a while.

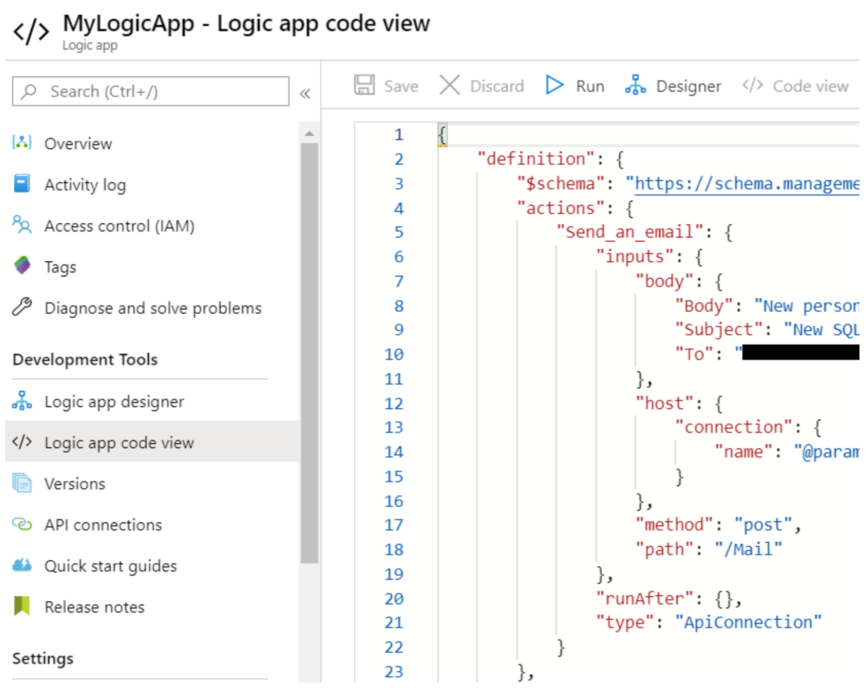
Once the logic app has moved to the new resource group, there are a few problems you have to solve. Once you check the designer, you'll notice some errors. Your connections have become invalid. That's because the connections are stored as API connections, which you can find in the menu on the left, or by going to **All Azure resources** and searching for **API connections**. Because API connections are separate resources, they are still shared with the original logic app, and they are also still in the original resource group. So, you'll have to create a new connection for both your database and your Outlook account.

Be aware that you have to completely refresh the page for the API connections tab to correctly remove the old connections. This is a bit of a hassle, but since this is another environment, you probably needed to create new connections anyway—the production database instead of the test database, for example. Once the manual changes are done, you can enable the logic app on the **Overview** page. So, this method has some problems, but it's relatively easy, and someone who is not technical could still do this.

ARM templates

The other option for deploying logic apps is using Azure Resource Manager (ARM) templates. Almost all Azure resources can be deployed using ARM templates, which are declarative JSON documents that describe the resources you want to deploy. An ARM template can be deployed more than once, and always gives the same result. ARM templates can contain more than one resource, and they can be deployed using the Azure portal, PowerShell, Azure CLI, or Azure DevOps. The exact structure and syntax of ARM or JSON is not within the scope of this book, but I want to look at some logic app-specific "gotchas."

In the portal, go to **MyLogicApp** and select **Logic app code view** in the menu. This is where you can view the definition of the logic app in code. Be aware that this is only the part you created in the designer; the code to deploy the logic app itself is not included.



```
{
    "definition": {
        "$schema": "https://schema.manageme
        "actions": {
            "Send_an_email": {
                "inputs": {
                    "body": {
                        "Body": "New person
                        "Subject": "New SQL
                        "To": "[redacted]
                    },
                    "host": {
                        "connection": {
                            "name": "@param
                        }
                    },
                    "method": "post",
                    "path": "/Mail"
                },
                "runAfter": {},
                "type": "ApiConnection"
            }
        },
```

Figure 21: Logic App Code View

Microsoft has a couple of examples of logic app ARM templates on its GitHub account for Azure. The easiest is the one with a simple timer trigger that checks a URL every hour. A slimmed-down version looks as follows.

Code Listing 21

```
{
      "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
      "contentVersion": "1.0.0.0",
      "parameters": {
            "logicAppName": {
                  "type": "string",
                  "metadata": {
                        "description": "The name of the logic app to 
create."
                  }
            }
      },
      "variables": {},
      "resources": [{
            "type": "Microsoft.Logic/workflows",
            "apiVersion": "2019-05-01",
            "name": "[parameters('logicAppName')]",
            "location": "[resourceGroup().location]",
            "tags": {},
            "properties": {
                  "definition": {}
            }
      }],
      "outputs": {}
}
```

The **`definition`** part should be replaced with the code view from the logic app. However, this poses some problems. For example, the **`parameters`** section contains hard-coded references to the Outlook and SQL connections.

Code Listing 22

```
"parameters": {
      "$connections": {
            "value": {
                  "outlook": {
                        "connectionId": "/subscriptions/[subscription
ID]/resourceGroups/[resource
group]/providers/Microsoft.Web/connections/outlook",
                        "connectionName": "outlook",
                        "id": "/subscriptions/[subscription
ID]/providers/Microsoft.Web/locations/westeurope/managedApis/outlook"
                  },
                  "sql": {
                        "connectionId": "/subscriptions/[subscription
ID]/resourceGroups/[resource
group]/providers/Microsoft.Web/connections/sql",
                        "connectionName": "sql",
                        "id": "/subscriptions/[subscription
ID]/providers/Microsoft.Web/locations/westeurope/managedApis/sql"
                  }
            }
      }
}
```

You can fix this by using the **`subscription().id`** function instead of the actual subscription ID, and possibly using **`resourceGroup().name`** or passing in the subscription ID and/or resource group name as parameters to the template. You can also pass in the API connection name as a parameter.

To create the API connections using ARM, you can view them in the portal and select **Export template** from the menu. The email API connection template, for example, looks as follows.

Code Listing 23

```
{
    "type": "Microsoft.Web/connections",
    "apiVersion": "2016-06-01",
    "name": "[parameters('connections_outlook_name')]",
    "location": "westeurope",
    "properties": {
        "displayName": "[concat('[email]@',
parameters('connections_outlook_name'), '.com')]",
        "customParameterValues": {},
        "api": {
            "id": "[concat('/subscriptions/d777ca79-3991-4e0b-a7b0-
271ed6a3f69c/providers/Microsoft.Web/locations/westeurope/managedApis/',
parameters('connections_outlook_name'))]"
        }
    }
}
```

Also, you’ll probably want to parameterize some other values in the logic app template, like the email address and subject, which are also hard coded.

Code Listing 24

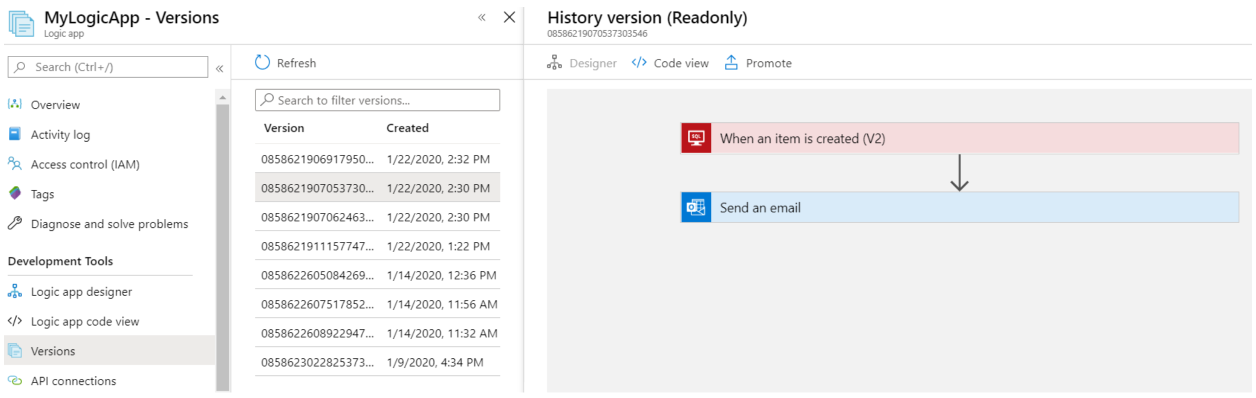
```
"actions": {
      "Send_an_email": {
            "inputs": {
                  "body": {
                        "Body": "New person: @{triggerBody()?['Id']} -
@{triggerBody()?['Name']}",
                        "Subject": "New SQL item",
                        "To": "[email]@[domain].com"
                  },
```

Once you have the complete ARM template, which can take some puzzling out to get right, you can deploy it using PowerShell, Azure CLI, the Azure portal, or Azure DevOps. To deploy your ARM template using the portal, go to **+ Create a resource** and look for **Template deployment (deploy using custom templates)**, and then chose **Build your own template in the editor**. Here you can simply copy and paste your ARM template, click **Save**, fill out the settings, and then click **Purchase**. In Azure DevOps, you can use the **Azure resource group deployment** task.

You can also change a logic app in the logic app code view directly. Instead of making changes in the designer, you can write them here and click **Save** at the top. For example, change the mail subject and see how it changes in the designer as well.

Versioning

When you keep your ARM templates in source control, you can always restore a previous version of your ARM template and redeploy it. On top of that, logic apps keep track of their own changes. In the menu, go to **Versions** and see all the various versions of the logic app. Every time you make a change using the designer, the code view, or an ARM template, a new version is added here. By clicking on a version, you can see the logic app as it was at that date and time. You can look in the designer or in the code view, but you can’t make changes. You can also “promote” this version, which overwrites the current version with this version. Of course, you can always go back to the current version by promoting that.



Version	Created
0858621906917950...	1/22/2020, 2:32 PM
0858621907053730...	1/22/2020, 2:30 PM
0858621907062463...	1/22/2020, 2:30 PM
0858621911157747...	1/22/2020, 1:22 PM
0858622605084269...	1/14/2020, 12:36 PM
0858622607517852...	1/14/2020, 11:56 AM
0858622608922947...	1/14/2020, 11:32 AM
0858623022825373...	1/9/2020, 4:34 PM

Figure 22: Logic App Versioning

All in all, logic apps have various methods of development and deployment, some more technical than others. Depending on what you do with it, your manager can play around in the designer, which may be enough, or you can use ARM templates in source control to deploy to various environments automatically.

Summary

Logic apps make it easy to click together workflows. Even your non-technical manager can create automatic tasks when they receive a new email or tweet. There are lots of triggers, endpoints, and control statements to create advanced workflows. With cloning, versioning, and code view, you get the tools to deploy your logic apps to different environments with relative ease.

Chapter 4 Service Bus

In Chapter 3, we looked at logic apps. Both logic apps and functions can be triggered by Service Bus messages, and have built-in support for putting messages on a Service Bus. In this chapter we're going to take a closer look at queues and topics with Service Bus. The Azure Service Bus is your go-to cloud messaging service in Azure, also called messaging as a service (MaaS).

Asynchronous processing

A lot of modern applications use some form of queueing. Queueing is a technology that allows for the asynchronous processing of data. For example, you place an order in a web shop. Once you click the order button, a message is put on a queue. Another process picks up this message and updates the stock for this item, places the item in your history, and sends out an email that your order is being processed. You could argue that this should be done immediately, but if a business receives lots of orders at the same time, the customer may have to wait for the processing, and you don't want to keep customers waiting. Queueing scales a lot better. Just put an order on a queue so it can process it in the next minute or so, and show the customer a message that the order is being processed.

Using a queue has another advantage, especially in a microservices environment. Consider for a moment that the processing fails and throws an exception. Normally, you'd log the error and show the user that something went wrong, and that they should try again. Service Bus, and queues in general, have retry mechanisms that allow you to retry the processing a few times (with a delay) so that it may be processed after all, without any intervention. But let's assume the data can't be processed at all, due to a bug. Messages that cannot be processed can be placed on a so-called "dead letter" queue. This allows you to send out a notification when something goes wrong, but it also makes it very easy to retry the process, because you still have the original message. Once you've fixed the bug, you can simply put the message back on the queue, and it will be processed just as it normally would.

This also means that you can decouple processes and applications. For example, to stay with the web shop example, you can keep your web shop online while updating your processing logic. New orders will simply be placed on the queue and are processed once the processing application is started again. This also means you get load balancing, as you can have multiple listeners for the same queue.

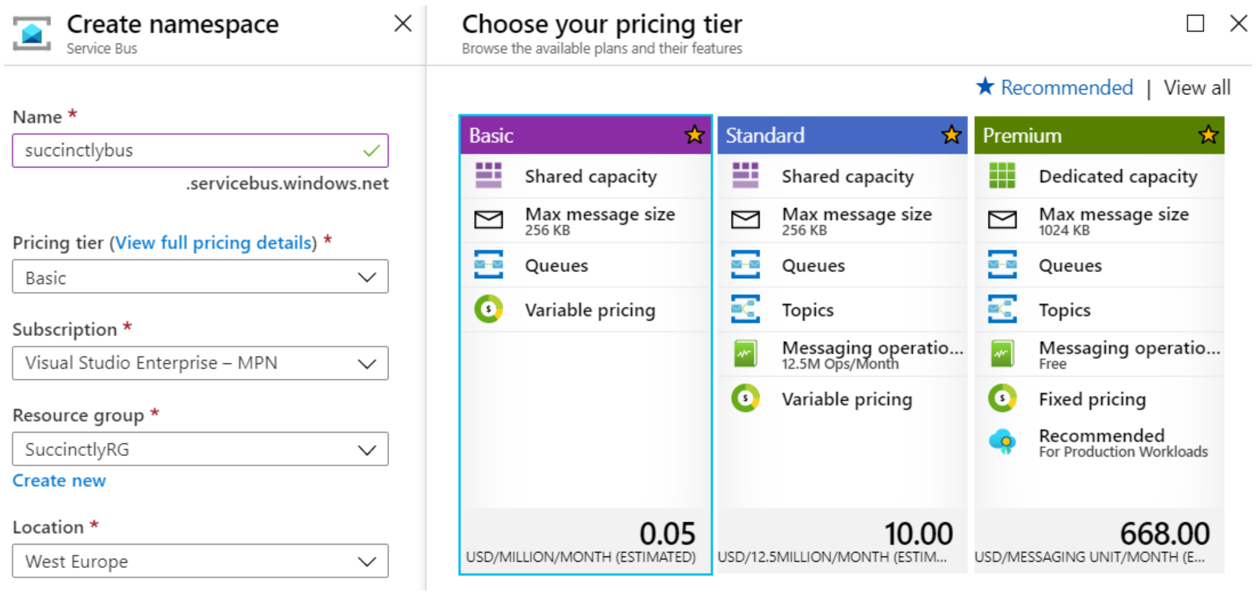
With queues, messages are processed one-to-one using the FIFO (first in, first out) principle. Next to queues, Service Bus offers topics. This is a broadcasting, or publish/subscribe pattern, which offers one-to-many messaging. An application places a message on a topic, and the topic notifies all subscribers of the topic. We'll go into topics in greater detail in the next sections.

Creating a Service Bus namespace

Let's create a Service Bus namespace first. The namespace is where your queues and topics live. We already created one in Chapter 2, but we can now look at it in more detail. In the Azure portal, find all services and search for **Service Bus**. Click **Add**, and the **Create namespace** blade appears. The name for your Service Bus namespace has to be unique across Azure. The subscription, resource group, and location are self-explanatory. Select the **Basic** pricing tier and create the namespace.

The pricing tier needs some explanation. There are three tiers: Basic, Standard, and Premium. As you can see, the price goes up quickly. The Standard tier is 20 times more expensive than the Basic tier, and the Premium tier is almost 67 times more expensive than the Standard tier, and 13,360 times more expensive than the Basic tier! Those are some serious differences.

The Standard tier offers more operations per month than the Basic tier, but the biggest difference is that it allows topics, whereas the Basic tier only allows for queues. So, if you need a publish/subscribe pattern, you need at least the Standard tier. With Premium you get dedicated infrastructure, instead of sharing with complete strangers. Furthermore, you get unlimited operations per month. It also allows for bigger messages to be put on your queues and topics. Where the Basic and Standard tiers allow for messages of up to 256 KB, the Premium tier allows messages of up to 1024 KB. Not mentioned on the pricing blade is that the Standard and Premium tiers offer transactions, de-duplication, sessions, and forward to/send via. The Premium tier also offers geo-disaster recovery. The Premium tier is recommended for production workloads, but the Standard tier will suffice for most small and medium-sized companies.


Create namespace
Service Bus

Name * succinctlybus .servicebus.windows.net

Pricing tier (View full pricing details) * Basic

Subscription * Visual Studio Enterprise – MPN

Resource group * SuccinctlyRG

Create new

Location * West Europe

Choose your pricing tier
Browse the available plans and their features

★ Recommended | View all

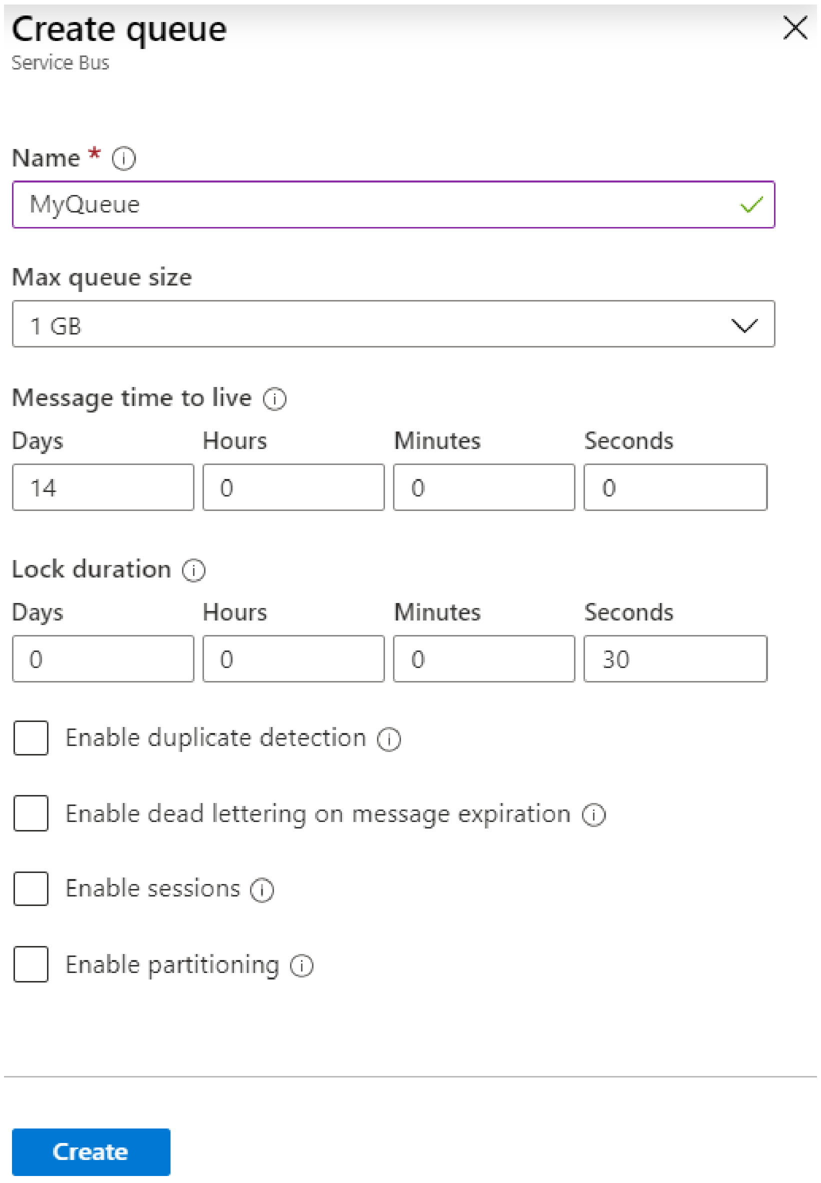
Basic	Standard	Premium
Shared capacity	Shared capacity	Dedicated capacity
Max message size 256 KB	Max message size 256 KB	Max message size 1024 KB
Queues	Queues	Queues
Variable pricing	Topics	Topics
	Messaging operatio... 12.5M Ops/Month	Messaging operatio... Free
	Variable pricing	Fixed pricing
		Recommended For Production Workloads
0.05 USD/MILLION/MONTH (ESTIMATED)	10.00 USD/12.5MILLION/MONTH (ESTIM...	668.00 USD/MESSAGING UNIT/MONTH (E...

Figure 23: Creating a Service Bus Namespace

The actual pricing for each tier is calculated differently. With the Basic tier, you pay €0.043 per million operations, so that's roughly five cents per million operations. The reason that the price goes up for the next tiers is because you pay per hour. That means that the Basic tier is the only true serverless Service Bus solution. The €10.00 includes 12.5 million free operations, and after that you start paying per million operations. The price per million operations differs though; you pay €0.169 from 13M to 87M operations, but €0.675 per million for 87M to 100M. If you need that many operations, I suggest you read the Service Bus pricing documentation. The Premium tier has a fixed price, so you just pay €668.00 a month.

Creating a queue

Creating a queue is very easy. Just go over to your Service Bus namespace, find **Queues** in the menu, and click **+ Queue**. Enter a name and click **Create**.


Create queue
Service Bus

Name *: MyQueue

Max queue size: 1 GB

Message time to live

Days	Hours	Minutes	Seconds
14	0	0	0

Lock duration

Days	Hours	Minutes	Seconds
0	0	0	30

☐ Enable duplicate detection

☐ Enable dead lettering on message expiration

☐ Enable sessions

☐ Enable partitioning

Figure 24: Creating a Queue

You can enable some extra options, which are explained in the information circles. As for the size of the queue, one GB is 1,024 MB, or 1,048,576 KB. A single message can be up to 256 KB. So, a queue of 1 GB can have at least 1,048,576 / 256 = 4,096 messages, which is more than enough for our test case.

Messages are kept in the queue for 14 days, so if it isn't picked up after 14 days, it's discarded. Instead of discarding, you can check **Enable dead lettering on message expiration**. A dead-letter queue is a special queue where all messages go that can't be processed. So, if your software throws an exception during message processing, the message is sent to the dead-letter queue. There, you can inspect it and either delete it, or move it back to the queue so it's re-processed.

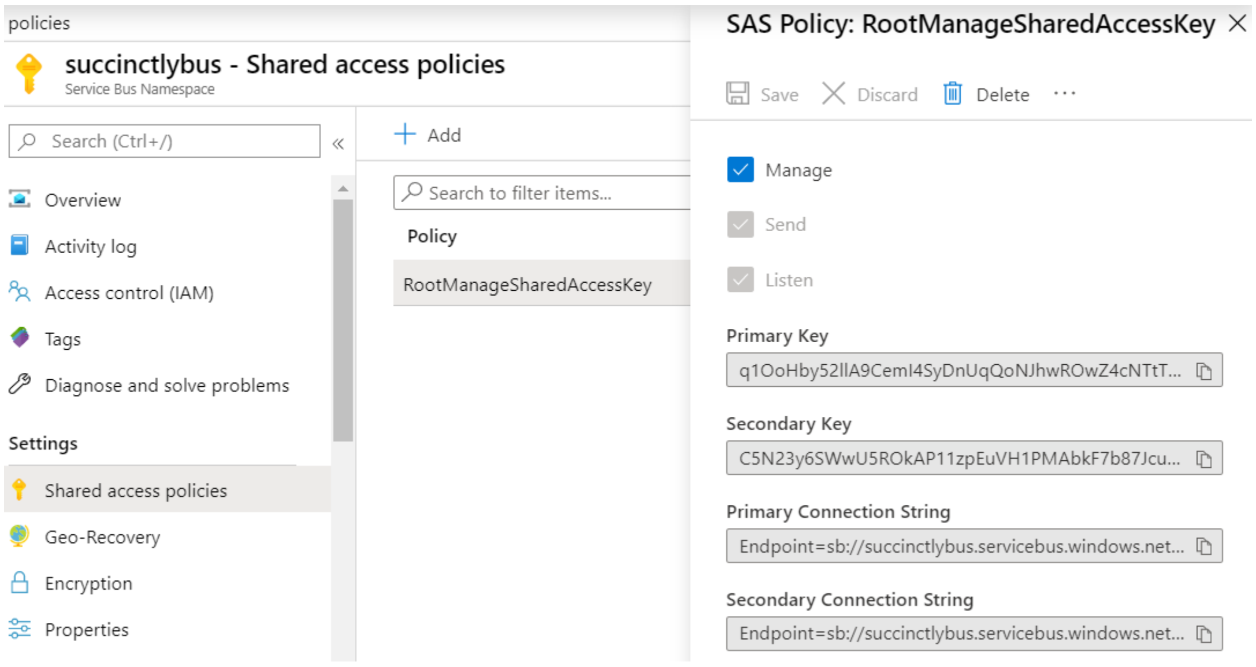
Consuming a queue

We've seen how a queue works with functions, but you can use a queue from regular .NET Core projects as well. Open Visual Studio and start a new .NET Core console application. Using NuGet, install the packages **Microsoft.Azure.ServiceBus** and **Newtonsoft.Json**. Next, you need a couple of **`using`** statements.

Code Listing 25

```
using Microsoft.Azure.ServiceBus;
using Newtonsoft.Json;
using System;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

Next, we're going to create a little helper method to create a **`QueueClient`**. We need a connection string for this. You can get this in the **Shared access policies** of the namespace in the Azure portal. The root access key is already created, but you can create new keys with separate access for managing, sending, and listening for messages. You get two keys and connection strings so that if one is compromised, you can switch to the second one before refreshing the first one. For this example, we can get any of the two root connection strings.



Manage ✓
Send ✓
Listen ✓

Primary Key
q1OoHby52lIA9CemI4SyDnUqQoNJhwROwZ4cNTtT...

Secondary Key
C5N23y6SWwU5ROkAP11zpEuVH1PMAbkF7b87Jcu...

Primary Connection String
Endpoint=sb://succinctlybus.servicebus.windows.net...

Secondary Connection String
Endpoint=sb://succinctlybus.servicebus.windows.net...

Figure 25: Shared Access Policies

In code, we can use the **`QueueClient`** and pass in the connection string and the name of the queue you want to connect to.

Code Listing 26

```
static IQueueClient CreateQueueClient()
{
    return new QueueClient("[ConnectionString]", "MyQueue");
}
```

Next, we need two small helper methods. In order to put something on the queue, you need a **`byte`** array. A generic method of putting anything on the queue is by serializing any object to JSON and using an encoding to get the byte representation. For the reverse, turning bytes into an object, you can use the encoding to get the string from the bytes and then deserialize the JSON into an object. We can use the **`Newtonsoft`** library for the serialization of JSON.

Code Listing 27

```
static byte[] ObjectToBytes(object obj)
{
    string content = JsonConvert.SerializeObject(obj);
    return Encoding.UTF8.GetBytes(content);
}

static T BytesToObject<T>(byte[] bytes)
{
    string content = Encoding.UTF8.GetString(bytes);
    return JsonConvert.DeserializeObject<T>(content);
}
```

Next, we can create two clients: one for putting new messages on the queue, and one for listening to new messages. In theory, this can be the same **`QueueClient`**, but in practice this probably isn't the case. Let's start with putting messages on the queue. We're going to put **`Person`** objects on the queue, but you can use this method for any object. The **`Person`** class is simple.

Code Listing 28

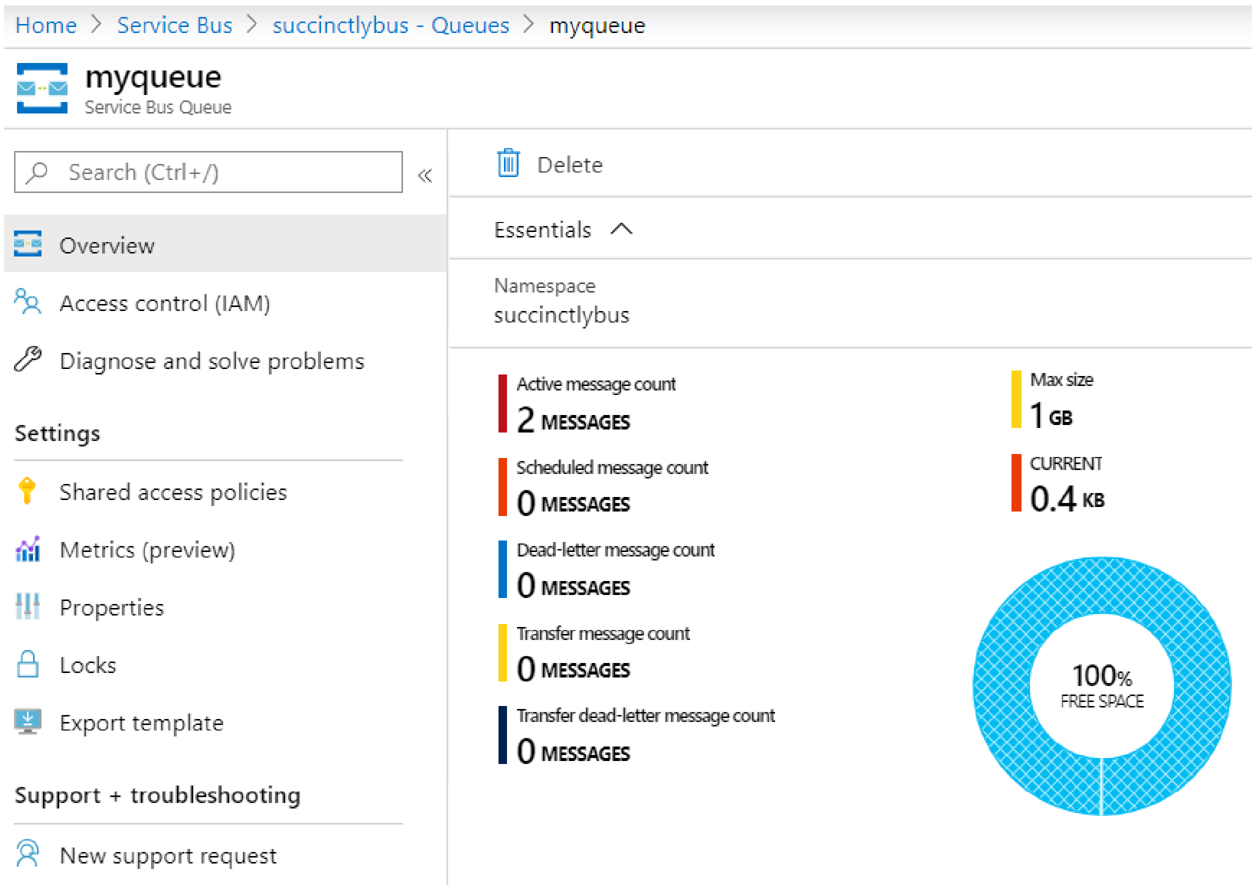
```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Next, in the **`Main`** method of your console application, create a new client, create a person, convert it to bytes, put it in a Message object, and send it to the queue.

Code Listing 29

```
static async Task Main()
{
    // Send messages.
    var client = CreateQueueClient();

    Person bill = new Person
    {
        FirstName = "Bill",
        LastName = "Gates"
    };
    byte[] billBytes = ObjectToBytes(bill);
    Message billMsg = new Message(billBytes);
    await client.SendAsync(billMsg);

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
```

You can repeat this code at your own leisure. I copied it, so it also adds Satya Nadella on the queue as well. To see if it worked, go to the Azure portal, look at your queues, and click the **myqueue** queue. This opens the queue overview where you can see how many messages are in the queue or in the dead-letter queue, how much of your queue's size is used up, and some other statistics.



Namespace
succinctlybus

Active message count
2 MESSAGES

Scheduled message count
0 MESSAGES

Dead-letter message count
0 MESSAGES

Transfer message count
0 MESSAGES

Transfer dead-letter message count
0 MESSAGES

Max size
1 GB

CURRENT
0.4 KB

100%
FREE SPACE

Figure 26: Queue Overview

Next, we need a listener. A listener needs two methods: a method that handles the received message, and a method that handles any exceptions. In the handler, we get the body, a byte array, from the received message. We can use the **`BytesToObject`** method to convert the bytes to a **`Person`** object. Since the console is thread safe, we can simply use **`Console.WriteLine`** and write the **`FirstName`** and **`LastName`** to the console.

Code Listing 30

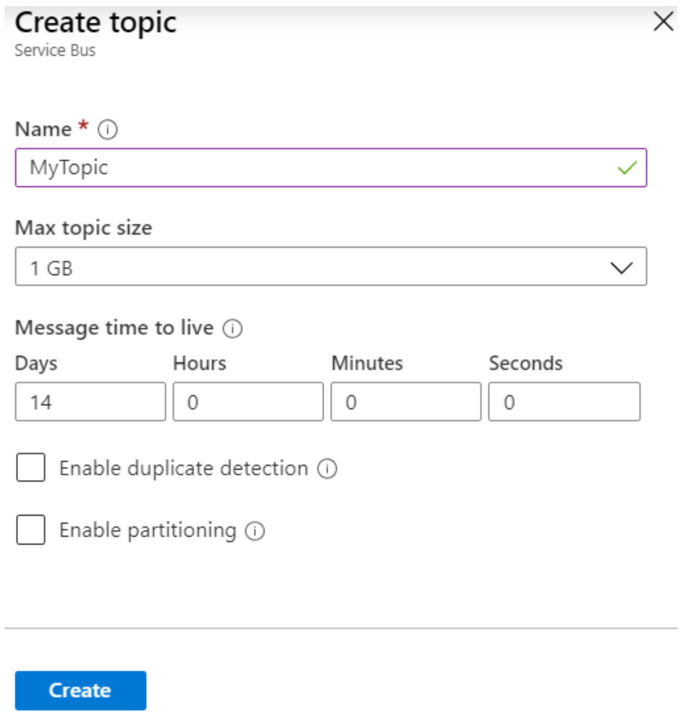
```
static async Task Main()
{
    // Receive messages.
    var receivingClient = CreateQueueClient();
    receivingClient.RegisterMessageHandler(HandleMessage, ExceptionReceived);

    // Send messages.
    // [...]

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

static Task HandleMessage(Message message, CancellationToken token)
```

```
{
    Person person = BytesToObject<Person>(message.Body);
    Console.WriteLine($"Received {person.FirstName} {person.LastName}.");
    return Task.CompletedTask;
}

static Task ExceptionReceived(ExceptionReceivedEventArgs e)
{
    Console.WriteLine($"Exception occurred: {e.Exception.Message}");
    return Task.CompletedTask;
}
```

When you run this, you should see **`Bill Gates`** (and in my case, **`Satya Nadella`**), printed to the console. One thing to notice, is that **`Press any key to exit`** may be printed before any person in the queue. This is because the messages are received on separate threads, and there's probably a slight delay in receiving them. Another thing to note is that you can create a second **`receivingClient`**, but any person will only be printed once because queues have one-to-one communication, meaning that if one client receives the message, any other clients will not receive it anymore.

Creating a topic

Next, we are going to create a topic. If you followed my example, you've chosen the Basic pricing tier, but topics can only be created in the Standard or Premium tiers. Go to your namespace overview and click on the pricing tier. Here, you can switch to Standard tier. You can't change to the Premium tier because that has a different infrastructure. If you want to go to Premium, you can select the **Migrate to premium** menu item. When you switch to the Standard tier, you need to refresh the page. You'll now see **Topics** in the menu. Click **Topics** and add a topic. The blade for a topic is pretty much the same as for a queue, but even simpler. So, type a name and create the topic.


Create topic
Service Bus

Name * MyTopic

Max topic size: 1 GB

Message time to live

Days	Hours	Minutes	Seconds
14	0	0	0

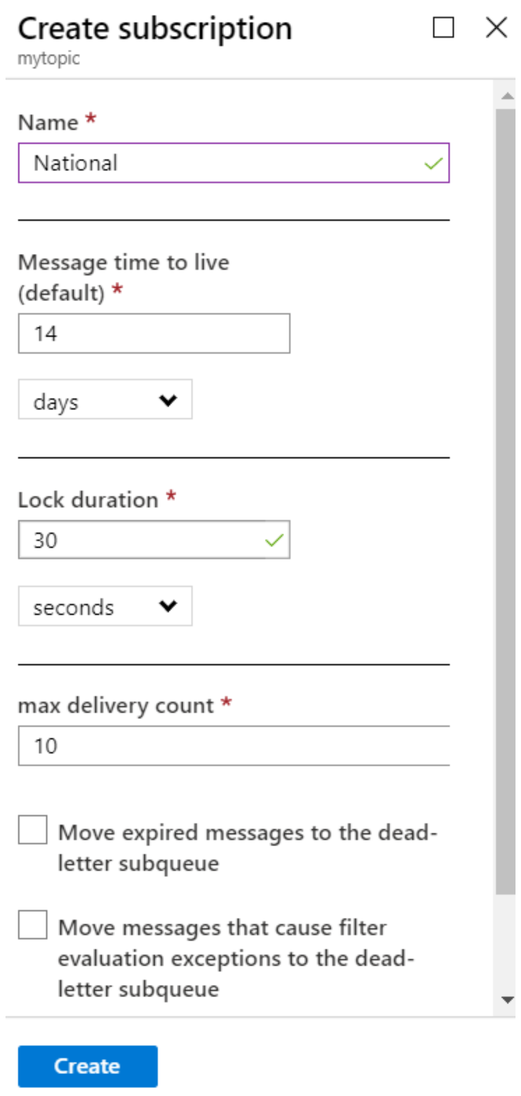
☐ Enable duplicate detection

☐ Enable partitioning

Figure 27: Creating a Topic

A topic alone does nothing. Additionally, we need to create a subscription. This is where a topic differs from a queue. Just like with a queue, messages are sent to a topic. However, where listeners receive messages directly from a queue, a topic sends them to subscriptions, which then notify listeners. A topic can have more than one subscription. Messages are sent to specific subscriptions through filters. Topics can then apply further filters to messages. Unfortunately, the Azure portal has very little tooling for managing Service Bus, and managing subscriptions and filters is currently not possible.

We're first going to create two subscriptions, and then we're going to add and remove some filters using the Azure CLI. In the next section, we'll use the Service Bus Explorer to manage Service Bus resources. For now, go to the Service Bus namespace in the portal and browse to the topic we just created. There, click **+ Subscription** and enter a name. For this example, create two subscriptions named **National** and **International**.


Create subscription

mytopic

Name *

National

Message time to live (default) *

14

days

Lock duration *

30

seconds

max delivery count *

10

☐ Move expired messages to the dead-letter subqueue

☐ Move messages that cause filter evaluation exceptions to the dead-letter subqueue

Figure 28: Creating a Subscription

Once the two subscriptions are created, we can add some filters. We're going to send orders to the topic; orders with a country NL (the Netherlands) are national, while orders with any other country are international. So, open the Azure CLI in the portal. If you're using a different subscription you need to select that first, using your subscription ID.

Code Listing 31

```
az account set --subscription xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx
```

After that, you can create the rules for your subscriptions. The filter syntax is SQL-like and **`Country`** is a property of your topic message, as we'll see in the code sample. Be sure to substitute any names you've changed.

Code Listing 32

```
az servicebus topic subscription rule create --name CountryNL --namespace-
name succinctlybus --resource-group SuccinctlyRG --subscription-name
National --topic-name MyTopic --filter-sql-expression "Country = 'NL'"

az servicebus topic subscription rule create --name CountryNotNL --
namespace-name succinctlybus --resource-group SuccinctlyRG --subscription-
name International --topic-name MyTopic --filter-sql-expression "Country !=
'NL'"
```

As for the code, we can reuse some code from the queue example, like **`ExceptionReceived`**, **`ObjectToBytes`**, and **`BytesToObject`**. Other than that, the code looks very similar, but we're using some different classes. Instead of **`Person`**, we're going to use **`Order`**.

Code Listing 33

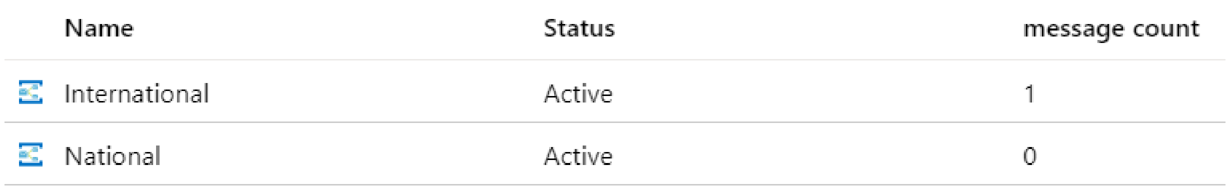
```
class Order
{
    public string Country { get; set; }
    public decimal Amount { get; set; }
}
```

Since the client for sending and receiving messages is different for topics and subscriptions, I've put the connection string and topic name in a string constant that we can reuse.

Code Listing 34

```
const string ConnString = "[ConnString]";
const string Topic = "MyTopic";

static async Task Main()
{
    // Send messages.
    var client = new TopicClient(ConnString, Topic);

    Order intOrder = new Order
    {
        Amount = 1000,
        Country = "US"
    };
    byte[] intBytes = ObjectToBytes(intOrder);
    Message intMsg = new Message(intBytes);
    intMsg.UserProperties.Add("Country", intOrder.Country);
    await client.SendAsync(intMsg);

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
```

The code is almost the same as for the queue, except of course the **`TopicClient`** instead of **`QueueClient`**. The most important difference is the **`UserProperties.Add`**. Remember that the message you send to the queue or topic is just a byte array—it's not necessarily an object with properties. So, for a subscription to evaluate a filter, it needs user properties. By adding the country to the user properties, the message can be filtered. You could see if it worked by going to your topic in the Azure portal. If it worked, your International subscription has a message while your National subscription is empty.



Name	Status	message count
International	Active	1
National	Active	0

Figure 29: View Subscriptions

Now for the handler part. This is almost the same as the queue example. We need a **`SubscriptionClient`** to which we can register a message handler. These are somehow registered in the application, so having multiple handlers for the same subscription in the same application will not work. Instead, you'll need to create a second application, run that first, and then start this one, and you'll see that the message is handled by both applications. Of course, we'll need to change **`HandleMessage`** as well, because it now needs to handle orders instead of persons.

Code Listing 35

```
static async Task Main()
{
    // Receive messages.
    var subClient = new SubscriptionClient(ConnString, Topic, "International
");
    subClient.RegisterMessageHandler(HandleMessage, ExceptionReceived);

    // Send messages.
    // [...]

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

static Task HandleMessage(Message message, CancellationToken token)
{
    Order order = BytesToObject<Order>(message.Body);
    Console.WriteLine($"Received {order.Country} {order.Amount}.");
    return Task.CompletedTask;
}
```

Try to add a national order as well to see if that works, and if it's not handled by the national subscription handler. The downside to subscriptions is that if you have a message that absolutely has to be handled by two applications, and one of the applications is not running at the time of publishing, the message is only handled by one application—and after that, it's gone. Any other rules, like dead-letter queues, still hold true for subscriptions.

Consuming subscriptions in functions

In the first chapter of this book, we looked at functions and used the Service Bus trigger to read messages from a queue and send output as a message to a queue. This is also possible for topics and subscriptions, but it looks a little different.

Code Listing 36

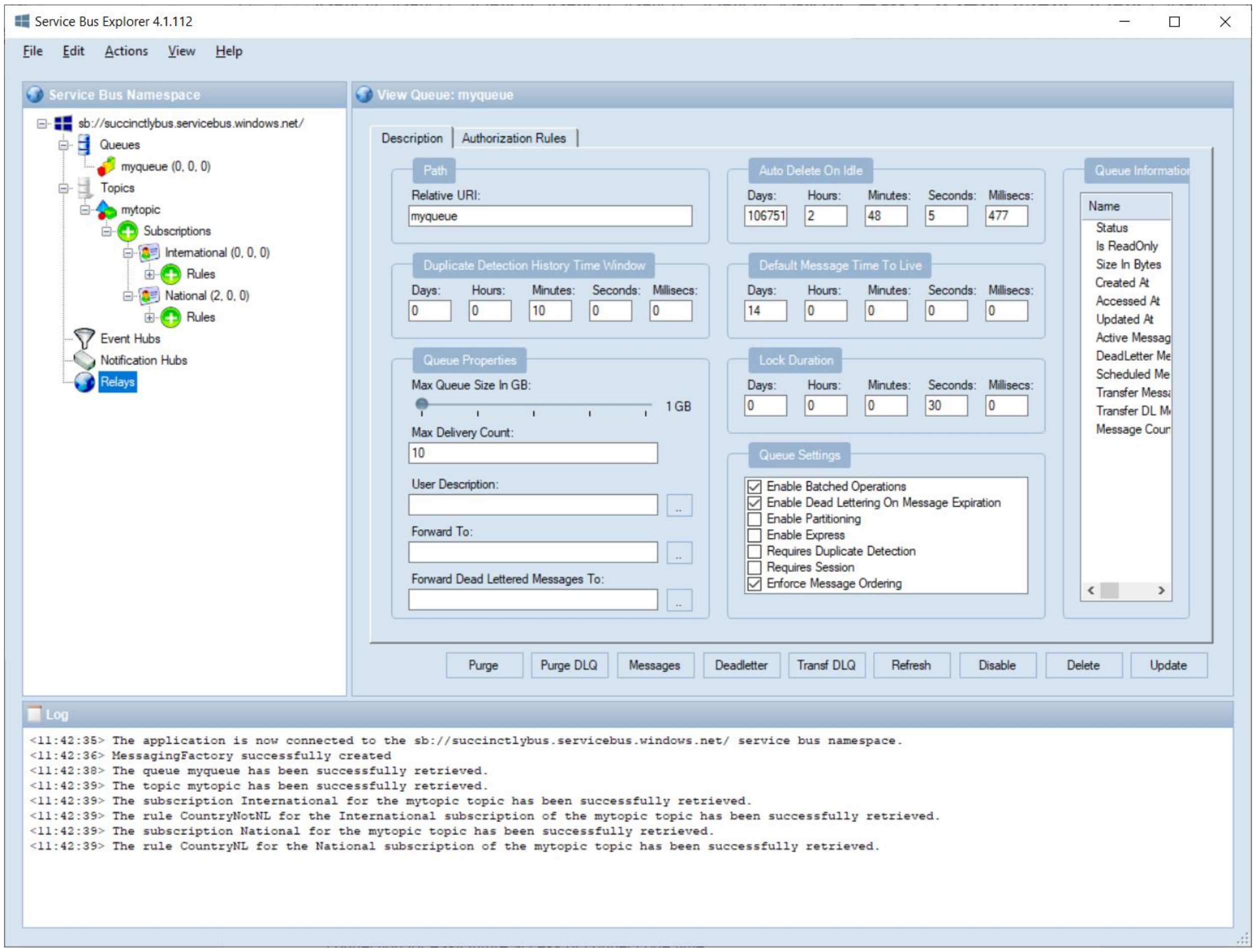
```
[FunctionName("ReadTopic")]
public void DoWork([ServiceBusTrigger("mytopic", "International", Connection
 = "BusConnString")]string message,
    ILogger log)
{
    var order = JsonConvert.DeserializeObject<Order>(message);
    log.LogInformation($"Received {order.Country} {order.Amount}.");
}
```

You can still use the **`ServiceBusTrigger`** that we also used for queues. Instead of the queue name, you use the topic name and, additionally, you have to specify the subscription name. The message you receive can be bound to a string, so you don't have to convert byte arrays, but we covered that in Chapter 1. The return attribute doesn't change because messages are sent to a queue or topic in the same way.

Service Bus Explorer

Unfortunately, the Azure portal offers little in ways of working with queues and topics. For something as fundamental as filters on topics, we had to resort to the Azure CLI. Luckily, there's a tool that can make working with Service Bus a lot easier. It allows you to peek at messages; remove messages; work with the dead-letter queue; re-queue messages; add and remove filters; create and remove queues, topics, and subscriptions; and directly place messages on queues and topics. To do any serious work with Service Bus, I really recommend getting the Service Bus Explorer.

Service Bus Explorer is an open-source tool that was developed using GitHub, so that's where you can find the code, the documentation, and the releases. Get the latest release as a .zip file from the releases page and unzip it somewhere on your computer. You can then simply start the .exe file. In the application, go to **File** > **Connect**. Get a connection string from the portal and enter it in the connection string field. You can save the connection for easy future access or connect one time.



Path
Relative URI: myqueue

Auto Delete On Idle
Days: 106751 Hours: 2 Minutes: 48 Seconds: 5 Millisecs: 477

Duplicate Detection History Time Window
Days: 0 Hours: 0 Minutes: 10 Seconds: 0 Millisecs: 0

Default Message Time To Live
Days: 14 Hours: 0 Minutes: 0 Seconds: 0 Millisecs: 0

Queue Properties
Max Queue Size In GB: 1 GB
Max Delivery Count: 10
User Description:
Forward To:
Forward Dead Lettered Messages To:

Lock Duration
Days: 0 Hours: 0 Minutes: 0 Seconds: 30 Millisecs: 0

Queue Settings
- [x] Enable Batched Operations
- [x] Enable Dead Lettering On Message Expiration
- [ ] Enable Partitioning
- [ ] Enable Express
- [ ] Requires Duplicate Detection
- [ ] Requires Session
- [x] Enforce Message Ordering

```
<11:42:35> The application is now connected to the sb://succinctlybus.servicebus.windows.net/ service bus namespace.
<11:42:36> MessagingFactory successfully created
<11:42:38> The queue myqueue has been successfully retrieved.
<11:42:39> The topic mytopic has been successfully retrieved.
<11:42:39> The subscription International for the mytopic topic has been successfully retrieved.
<11:42:39> The rule CountryNotNL for the International subscription of the mytopic topic has been successfully retrieved.
<11:42:39> The subscription National for the mytopic topic has been successfully retrieved.
<11:42:39> The rule CountryNL for the National subscription of the mytopic topic has been successfully retrieved.
```

Figure 30: Service Bus Explorer

As you can see, the tree view on the left side of the Explorer makes it easy to see your queues, topics, and subscriptions, and how many messages are on the queues and dead-letter queues. Right-clicking on a queue, topic, or subscription opens a context menu that lets you delete, disable, rename, export, send, receive, and purge.

Choose **Send Messages** on your queue. A new window opens, and you can now enter some text to send to the queue. Since we don't have a listener now, it doesn't really matter what you enter. Just enter some text and click **Start**. It's not directly evident, but this puts a message on the queue. Close the window and refresh your queue, and you'll see a message in the queue. You can remove it from the queue by purging it or by receiving messages. Click **Receive Messages**, and another window opens. You can now enter the number of messages you want to receive, but the default is 10. You can also select **Peek** or **Receive and Delete**. With Peek, you can read the messages, but they won't be removed from the queue. Receive and Delete lets you read the messages and removes them from the queue, but you can resubmit them if you like.

For topics, you can send messages and create subscriptions. For this, we had a user property that decided whether a message was sent to the National or International subscription. Next to the message text field are message properties. The properties **`MachineName`** and **`UserName`** are added by default, but you can add properties however you see fit. So, add **`Country`** with the value **`NL`** (for national order), or whatever other value for international orders. Click **Start**, and you'll see the message added to the proper subscription. If you right-click the subscription, you can also add rules. You can't change existing rules, only delete and add again. If you click a rule, you can read the filter expression. With actions, you can set additional properties on a message, for example **`SET MyProperty = 'MyValue'`**. Unfortunately, these features aren't very well documented by Microsoft. Other than that, you can receive messages on a subscription just like you could with a queue.

The Service Bus Explorer really is an indispensable tool when working with the Azure Service Bus. It has some tooling for Event Hubs, Notification Hubs, and Azure Relays as well, but they are not within the scope of this book. Remember that you can create all resources, including subscriptions and filters, using ARM templates as well. For manually reading, deleting, and sending messages, you're stuck with the Service Bus Explorer.

Summary

Azure Service Bus is a great tool to have in your arsenal. It allows you to decouple logic and applications and adds scaling to your applications. Queues are great for FIFO applications, while topics and subscriptions are great for publisher/subscriber patterns. Service Bus integrates well with Azure Functions, but the SDK allows you to work with it from any sort of application. With the Service Bus Explorer, you have all the tools you need to build and maintain your Azure Service Bus solutions.

Chapter 5 Event Grid

In the previous chapter, we looked at Azure Service Bus, a messaging as a service offering in the Azure ecosystem. Before that, we looked at the Azure Logic Apps and Azure Functions services. Azure Event Grid routes events from one source to the next, at massive scale. These events can come from various sources, such as storage accounts or subscriptions, and can be routed to various targets, like an Azure function, a logic app, or a Service Bus queue. Event Grid works with subscriptions, topics, and domains.

Pricing

No matter what kind of Event Grid resource you create, you'll never have to specify a pricing plan. With Event Grid, like with all serverless offerings, you pay for usage. The first 100,000 operations a month are free of charge. After that, you pay €0.506 per million operations. For most small and medium-sized applications, that means free to 50 cents a month. Don't underestimate the number of events you'll generate, though. I recently created a small app that reacted to a daily export of a third-party application. The application generated a new event, which would then be handled, and that was it. It turned out to be about 800 events from the export per day in a 20-day month, or 20 * 800 * 2, which adds up to 32,000 events a month. And that's just one small application. That's still free, but it quickly adds up. The following demos will be free in any case.

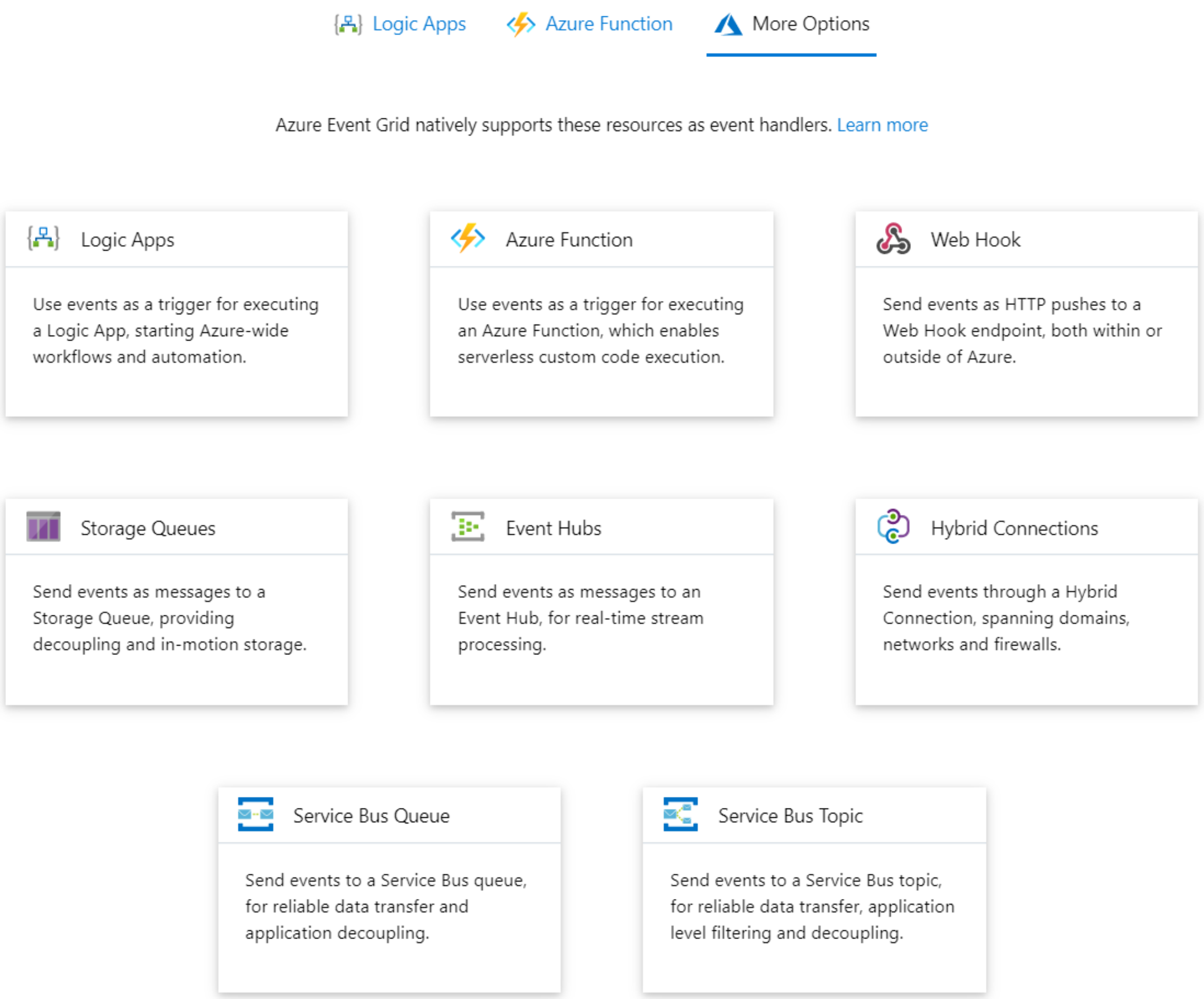
Creating a subscription

Event Grid works a little different than other Azure resources. You don't create an event grid like you'd create a web app, for example. Instead, you create event subscriptions, topics, and domains. Subscriptions are at the base of Event Grid. You can create Event Grid subscriptions in two ways. First, go to all services and search for **Event Grid**. You'll find Event Grid domains, subscriptions, and topics. Select **Subscriptions** there and click **+ Event Subscription** at the top. This will take you to the creation blade where you can pick your subscription details, including topics, events, and endpoints (event handlers).

There is a second way to create subscriptions. You can browse to whatever resource you want events from and create it there. This works, for example, for subscriptions, resource groups, storage accounts (either general-purpose v2 or blob storage), Key Vault (currently in preview), and Service Bus (Premium). For other resources, read the documentation.

Let's pick an example that we're familiar with: we're going to get events from a storage account to a Service Bus queue. So, go to a (general-purpose v2) storage account and select **Events** in the menu. Alternatively, create a subscription from the **Event Subscriptions Overview** page. For this demo, we're using the storage account. A **Get Started** page appears, which already gives you some options for creating a subscription with an event handler. The default is a logic app that triggers when a blob is uploaded, but you can also choose when an image or video is uploaded. The **Azure Function** option just takes you to your functions overview. Under **More**

Options you can also see Web Hook, Storage Queues, Event Hubs, Hybrid Connections, Service Bus Queue, and Service Bus Topic. All these buttons currently do the same thing, which is what also happens when you click **+ Event Subscription** at the top: you get a subscription creation blade for the storage account topic, and no endpoint details (event handler).



Azure Event Grid natively supports these resources as event handlers. Learn more

Logic Apps
Use events as a trigger for executing a Logic App, starting Azure-wide workflows and automation.

Azure Function
Use events as a trigger for executing an Azure Function, which enables serverless custom code execution.

Web Hook
Send events as HTTP pushes to a Web Hook endpoint, both within or outside of Azure.

Storage Queues
Send events as messages to a Storage Queue, providing decoupling and in-motion storage.

Event Hubs
Send events as messages to an Event Hub, for real-time stream processing.

Hybrid Connections
Send events through a Hybrid Connection, spanning domains, networks and firewalls.

Service Bus Queue
Send events to a Service Bus queue, for reliable data transfer and application decoupling.

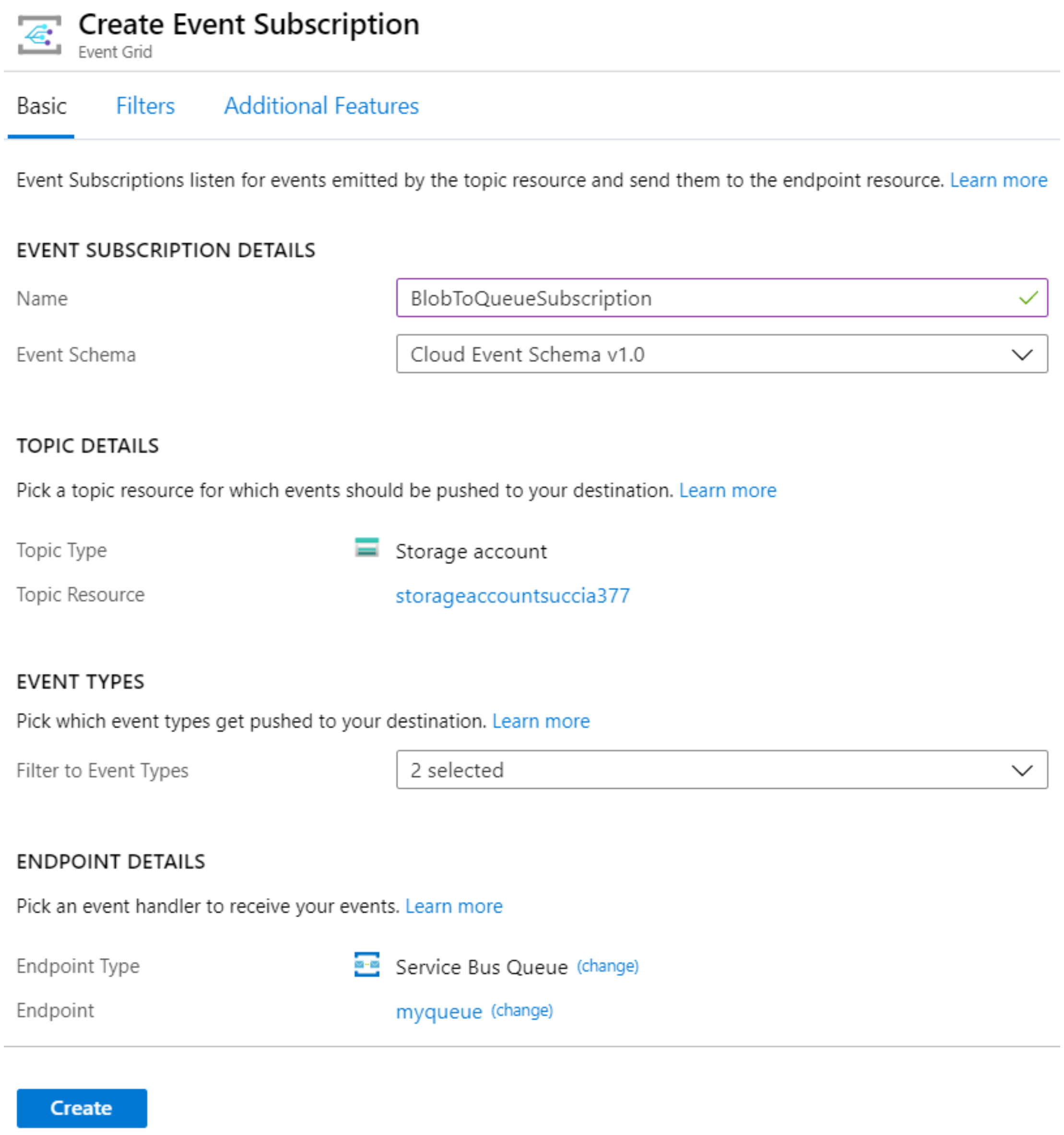
Service Bus Topic
Send events to a Service Bus topic, for reliable data transfer, application level filtering and decoupling.

Figure 31: Subscription Event Handlers

In the creation blade, enter a name for your subscription. The topic type and resource are already filled out, which are **Storage account** and the storage account that you selected. You can filter some event types. The events **Blob Created** and **Blob Deleted** are selected by default, but there's also a **Blob Renamed** event, and those same events exist for directories.

The endpoint type is the event handler, and we've already seen those. Select the **Service Bus** queue type and then select the endpoint. You can only select an existing queue, so if you don't want to use a queue that we've used in the previous examples, you should create a new queue first.

The **Event Schema** property needs a little more explanation. If an event is triggered, you'll receive event data. The data received and the format of the data depend on the event schema. The default is the **Event Grid Schema**, which contains data such as the Azure tenant and subscription, authorization information, and claims. This data is not very useful for our example. Instead, select **Cloud Event Schema v1.0**, which contains the type of event, a link to the created or deleted blob, the content type, the length of the blob, and the eTag that uniquely identifies a blob. Next to the basic subscription information, you can add additional filters and features, such as specific storage containers, dead-lettering, and retry policies. You can add or change these options later.



Create Event Subscription
Event Grid

Basic | Filters | Additional Features

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. Learn more

EVENT SUBSCRIPTION DETAILS

Name	BlobToQueueSubscription
Event Schema	Cloud Event Schema v1.0

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. Learn more

Topic Type	Storage account
Topic Resource	storageaccountsuccia377

EVENT TYPES

Pick which event types get pushed to your destination. Learn more

Filter to Event Types	2 selected

ENDPOINT DETAILS

Pick an event handler to receive your events. Learn more

Endpoint Type	Service Bus Queue (change)
Endpoint	myqueue (change)

Create

Figure 32: Creating an Event Subscription

Click **Create**, and your topic and subscription are created. If you now look at the **Events** blade in your storage account, you can see the event subscription. If you remember, we saw event subscriptions under all resources, too. However, you won't see your subscription there because of the default filter. It makes little sense, but you need to select your **Topic Type** (Storage Accounts), your **Subscription**, and your **Location** before you see anything. When you change the filter, your subscription is there, though.

Let's see our subscription in action. Simply go to your storage account and upload a file. Now, connect to your queue using Service Bus Explorer and read your queue. You should see a message with text that looks like the following.

Code Listing 37

```
{
    "id": "036490ca-801e-001f-35f3-dcf675062001",
    "source": "/subscriptions/[…]/storageAccounts/storageaccountsuccia377",
    "specversion": "1.0",
    "type": "Microsoft.Storage.BlobCreated",
    "dataschema": "#",
    "subject":
"/blobServices/default/containers/mycontainer/blobs/dummy.txt",
    "time": "2020-02-06T13:45:04.2806789Z",
    "data": {
        "api": "PutBlockList",
        "clientRequestId": "ab9f3d2b-410c-4f48-903e-b43d11028067",
        "requestId": "036490ca-801e-001f-35f3-dcf675000000",
        "eTag": "0x8D7AB0AC5333D0B",
        "contentType": "text/plain",
        "contentLength": 19,
        "blobType": "BlockBlob",
        "url": […]/mycontainer/dummy.txt",
        "sequencer": "0000000000000000000000000003CFB00000000038b29d3",
        "storageDiagnostics": {
            "batchId": "d4c88382-9006-002c-00f3-dca9de000000"
        }
    }
}
```

The **`source`** and **`type`** properties are useful for filtering in applications that need to respond to events. The **`data`** object has more useful information, such as the URL and eTag. The subject is useful too, since you can filter based on subject. In my case, I have a container named **`mycontainer`**, and I really don't want to get events from other containers. Go back to your event subscription and select **Filters**. Enable subject filtering and enter **/blobServices/default/containers/mycontainer** in **Subject Begins With** to only get events for **`mycontainer`**. There's also a **Subject Ends With** filter, which is useful if you want to filter for specific file types.

Note: Do you get a lot of events from your storage account? If you have an Azure function that's triggered by a timer, it will create a file every time the timer goes off.

You can fix this by either using dedicated storage accounts or by using a filter on your event subscription.

Azure function event handler

Most Event Grid event handlers are easy to implement, but since you'll probably need it, and because it doesn't work all that well, I'm going to shortly discuss functions that are triggered by Event Grid.

When you create a new function in Visual Studio and choose the Event Grid trigger, you'll end up with the following code.

Code Listing 38

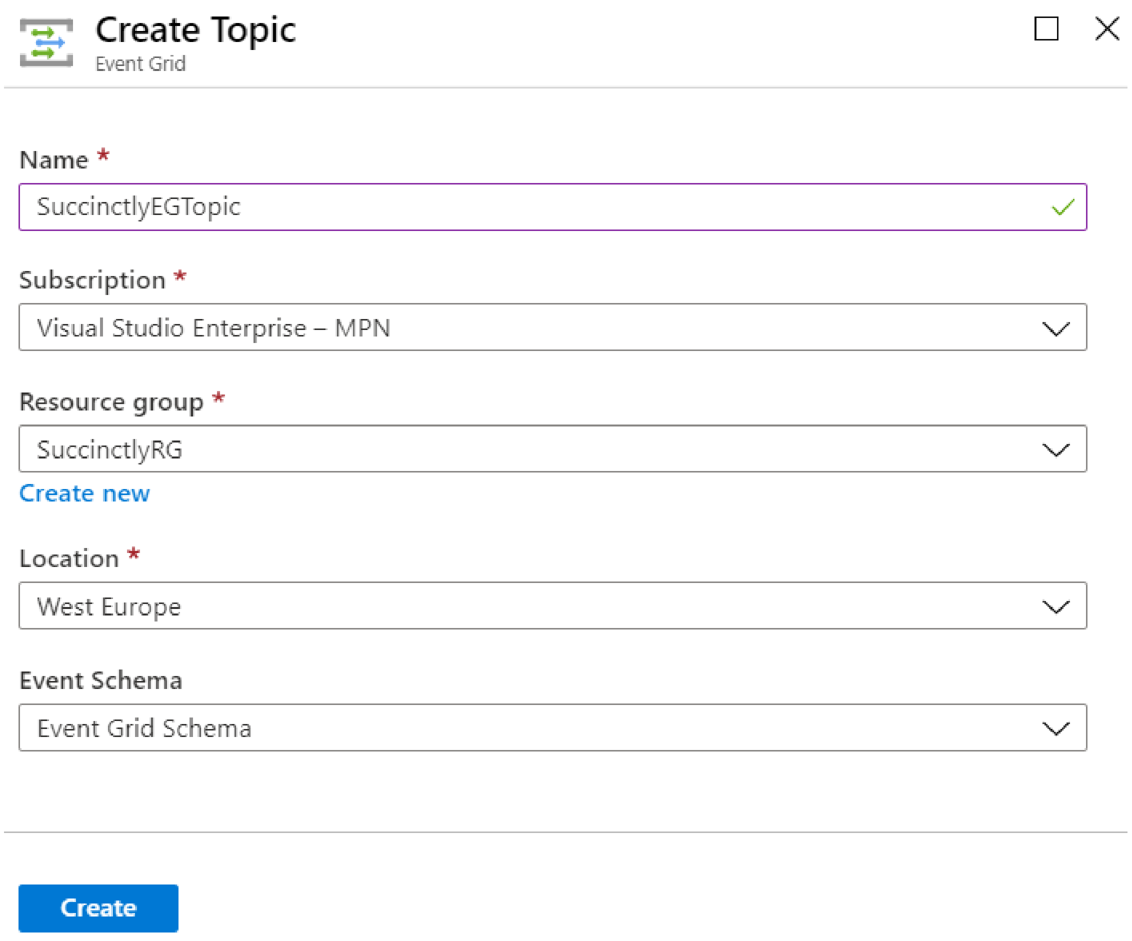
```
// Default URL for triggering Event Grid function in the local environment.
// http://localhost:7071/runtime/webhooks/EventGrid?functionName={functionname}
using Microsoft.Azure.EventGrid.Models;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.EventGrid;
using Microsoft.Extensions.Logging;

namespace FunctionApp1
{
    public static class Function1
    {
        [FunctionName("Function1")]
        public static void Run([EventGridTrigger]EventGridEvent eventGridEvent, IL
ogger log)
        {
            log.LogInformation(eventGridEvent.Data.ToString());
        }
    }
}
```

The default URL sort of works, but you'll get a message that the runtime failed to bind to the **`EventGridEvent`** class. When you create the function in the portal and try to run it there, you get a **`NotImplementedException`**. However, when you try to use it directly from the topic, it works like you'd expect.

You can work around these issues by creating an HTTP trigger instead and handling the event with a web hook.

Creating an Event Grid topic

Now that we've seen how Event Grid handles various events and sends them to event handlers, we can create our own topic. We can send out events to our topic, which Event Grid can then send to various event handlers using a subscription.


Create Topic
Event Grid

Name *
SuccinctlyEGTopic

Subscription *
Visual Studio Enterprise – MPN

Resource group *
SuccinctlyRG

Create new

Location *
West Europe

Event Schema
Event Grid Schema

Create

Figure 33: Creating an Event Topic

Once your topic is created, click **+ Event Subscription** at the top of the **Overview** blade. Here you can create a subscription like we did in the previous section. So, create a subscription and hook it up to a queue so we can test our custom topic.

To trigger events using our custom topic, we need to call it using an HTTP `POST` request. We can use Postman for this. You can find the endpoint on the Overview blade of your topic. Mine is **https://succinctlyegtopic.westeurope-1.eventgrid.azure.net/api/events**, which is the name of my topic, followed by the location code, followed by **eventgrid.azure.net/api/events**. To call the URL, append the API version to it: **?api-version=2018-01-01**. You'll also need to include a header with an access key. The access keys can be found under **Access keys** in the menu of your topic. For the body of your request, select `raw` and set the type to **JSON**. In the **Body** field enter the following JSON.

Code Listing 39

```
[
  {
    "id": 1,
    "eventType": "mytype",
    "subject": "mysubject",
    "eventTime": "2020-02-06T16:36:00+00:00",
    "data":{
      "Type": "OrderCreated",
      "Price": 100,
      "Country": "NL"
    },
    "dataVersion": "1.0"
  }
]
```

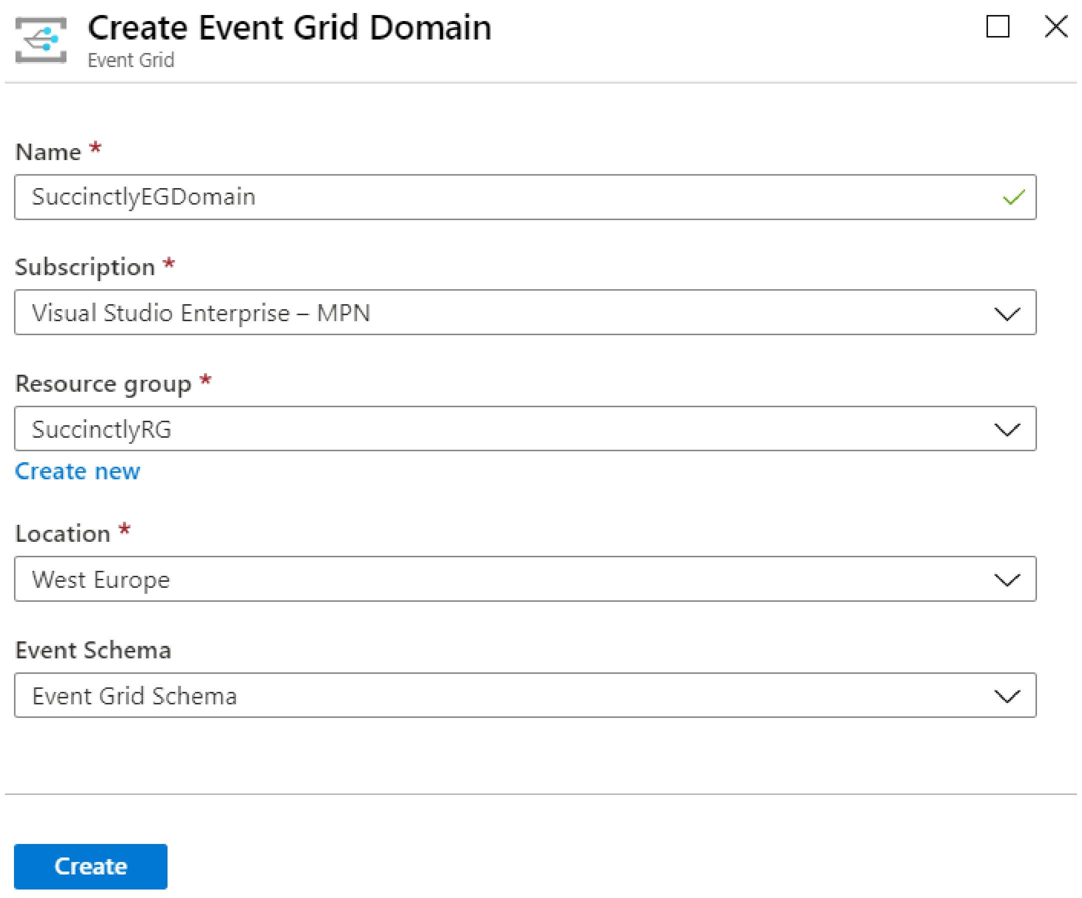
Last, set the method to **`POST`** and send the request.

If you now look in your queue, you should have a message that resembles the request, but with an added **`topic`** property. A topic can have multiple subscriptions so you can broadcast the event to multiple handlers.

You can also raise multiple events at once. The JSON payload is an array, so just adding multiple objects to the array will trigger multiple events. This is preferred over sending multiple events in separate requests.

Creating an Event Grid domain

We're now going to create an Event Grid domain. An Event Grid domain is a collection of topics and subscriptions. It allows you to publish events to a single endpoint, after which the domain will distribute it among the various topics. Domains also allow you to manage authorization and authentication among topics. To create an Event Grid domain, go to all services and search for **Event Grid**. Choose the Event Grid domain and fill out the creation blade that appears. Enter a name, subscription, resource group, and location. Leave the **Event Schema** on the default **Event Grid Schema**. Click **Create**.



Name *

SuccinctlyEGDomain

Subscription *

Visual Studio Enterprise – MPN

Resource group *

SuccinctlyRG

Location *

West Europe

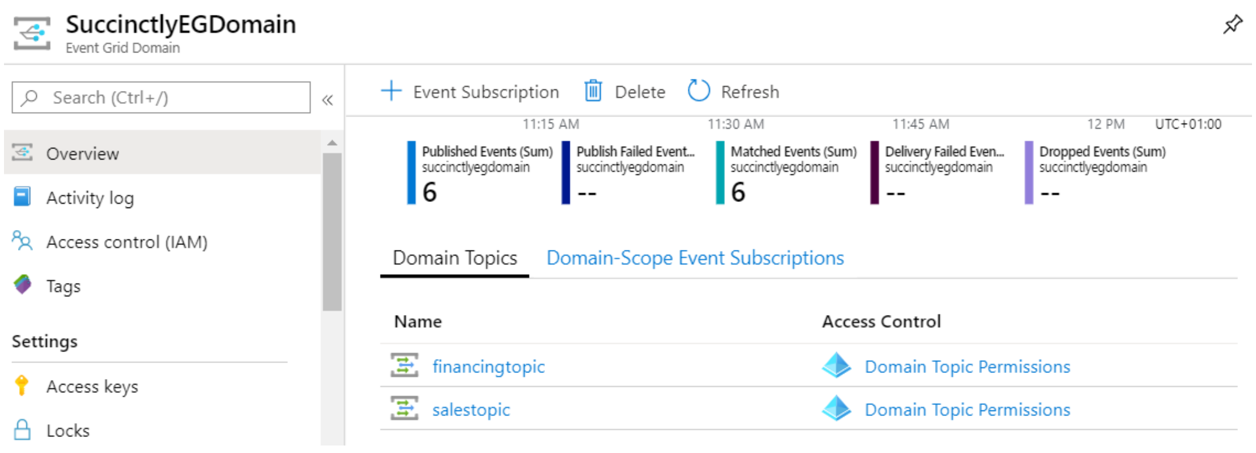
Event Schema

Event Grid Schema

Figure 34: Creating an Event Grid Domain

In the Event Grid domain, click **+ Event Subscription**. In the next blade, simply create a subscription, like we did before. There's one difference though: the creation blade asks you for a **Domain Topic**. You can unselect **Scope to domain topic**, which means all events from this domain will be published to this subscription. But if you do create a new domain topic, you can publish to that topic specifically.

Let's say your company has a sales department, a finance department, and an IT department, and they all need to subscribe to various events. So, you create a domain with three topics, a separate topic for each department. You can manage permissions per topic and create various subscriptions for each topic. However, you only need to publish events to one domain. With that in mind, create another subscription for a separate topic.


SuccinctlyEGDomain
Event Grid Domain
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Settings
Access keys
Locks
Event Subscription
Delete
Refresh
11:15 AM
11:30 AM
11:45 AM
12 PM
UTC+01:00
Published Events (Sum) succinctlyegdomain 6
Publish Failed Event... succinctlyegdomain --
Matched Events (Sum) succinctlyegdomain 6
Delivery Failed Even... succinctlyegdomain --
Dropped Events (Sum) succinctlyegdomain --
Domain Topics
Domain-Scope Event Subscriptions

Name	Access Control
financingtopic	Domain Topic Permissions
salestopic	Domain Topic Permissions

Figure 35: A Domain with Multiple Topics

Publishing events to the domain works the same as publishing events to a topic, with one minor difference: you have to specify the topic in the JSON payload.

Code Listing 40

```
[
  {
    "topic": "salestopic",
    "id": 1,
    "eventType": "mytype",
    "subject": "mysubject",
    "eventTime": "2020-02-06T16:36:00+00:00",
    "data":{
      "Type": "OrderCreated",
      "Price": 100,
      "Country": "NL"
    },
    "dataVersion": "1.0"
  }
]
```

This event is now handled by all subscriptions in the **`salestopic`**, and by the domain-scope event subscriptions.

To make handling various events more manageable, consider using the Event Type filter. This is like subject filtering, which we've seen in the storage account example, but a little less fine grained. Like the subject, the event type is part of the JSON you post to your endpoint, so it's up to you which types you give to your events. Using the type filter when creating a subscription prevents certain types of events from being published to the event handler. That also means that you can introduce new event types without them being handled by the subscription. This saves you from having to create a topic for each separate event concerning sales orders, or whatever business object you have.

Conclusion

Azure Event Grid allows you to create event-based applications. It allows you to react to events from Azure itself, like blobs or resources being created, but you can also add your own events using topics. With subscriptions, you can route events to specific handlers. Each topic can have multiple subscriptions, so each event can be handled by multiple handlers. By using an event domain, you can manage your custom topics and subscriptions more easily, and manage authorization and authentication in a single place. Using custom events with webhook handlers allows you to handle any event how you want to.

Chapter 6 Azure SQL

In the previous chapters, we've seen functions, logic apps, Service Bus, and Event Grid, which are all services that provide some sort of event-based computing or messaging. When you think of serverless, you're probably not thinking of any sort of database, yet Azure SQL has a serverless offering. In Chapter 2, we briefly used Azure SQL with Azure Logic Apps. In this chapter, we're going to further explore Azure SQL and its serverless offering.

Azure SQL tiers

Before we go and create a serverless SQL database, let's talk about the different tiers and how they're different from serverless. Creating a serverless Azure SQL instance works the same as creating a regular Azure SQL instance, except for the service tier. Understanding the different tiers is key to understanding serverless SQL. The default tiers when creating a new database used to be the Basic, Standard, and Premium tiers, which work with Database Transaction Units (DTUs). The default tiers are now based on vCores. One of these vCore offerings is serverless.

A DTU is a combination of CPU, memory, and IO resources. That means having more DTUs makes your database perform faster. Having more DTUs also means you can store more data. In DTU terms, a database with 20 DTUs is about twice as fast as a database with 10 DTUs.

A vCore, or virtual core, on the other hand, is just CPU, and lets you pick other characteristics of the hardware independently; although currently, the Gen5 configuration seems to be the only option for general workloads. The FSv2 series is compute optimized and offers 72 vCores, but less memory, at a better price than Gen5. There's also the M series, which offers massive memory, but also at a price. Whether you choose the general workload, compute optimized, or memory optimized, you can always pick your number of vCores independently.

Other than the difference in maximum number of DTUs or vCores, the different tiers offer other features, such as longer backup retention and high availability options. A single vCore is equal to 100 DTUs for the Basic and Standard tiers, and 125 DTUs for the Premium tier.

Let's get back to serverless, which is a vCore solution. Since serverless services are supposed to autoscale, you don't configure a number of vCores, but a minimum and a maximum. This is also linked to your minimum and maximum amount of memory, which is three times the minimum and maximum number of cores. Compare that to DTUs, which have CPU, memory, and IO resources in one, and you'll see why that would be hard to scale—your entire performance would drop like a brick.

With vCores, your computing power scales up and down, but your memory doesn't. When your serverless Azure SQL instance is idle for a specified amount of time, the database is paused, your resources are reclaimed, and you stop paying for your vCores and other infrastructure. The database is resumed after a small startup delay, when activity on the database occurs. Some actions and options, like geo-replication and long-term backup retention, prevent the database from auto-pausing.

The use case scenario for a serverless database is when you have a database that is accessed infrequently, or when usage is unpredictable for short periods of time. When your database is accessed constantly throughout the day, I'd recommend a regular tier, such as Standard or General Purpose Provisioned (or one of the high-availability tiers if you need that).

Billing

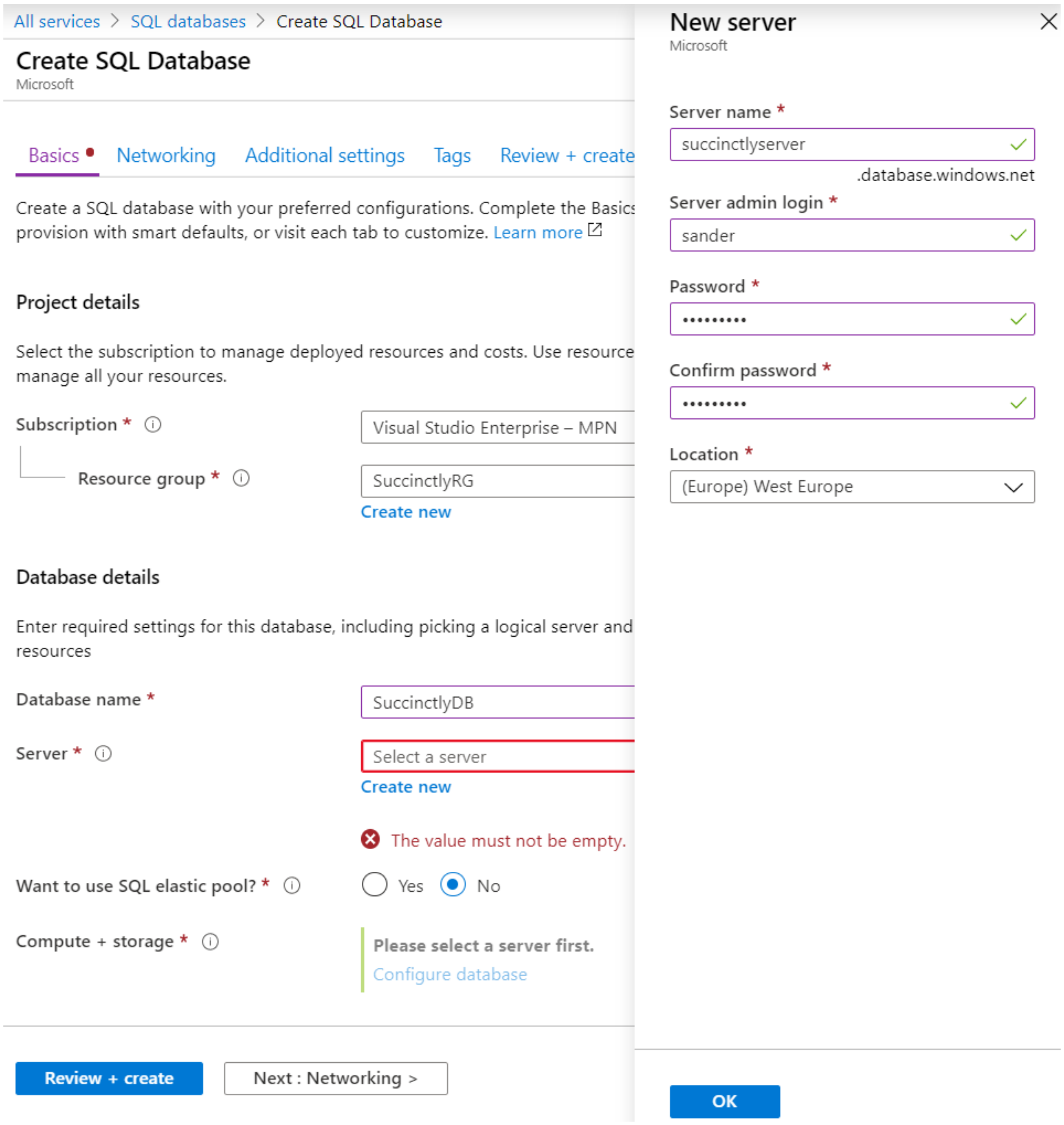
The serverless tier is billed by the maximum of CPU and memory used each second. If the used resources are less than the provisioned amount, then the provisioned amount is billed. So, let's say you have a database with a minimum of 0.75 vCores and a maximum of 3, and it's running a query using 1 to 2 vCores (let's say an average of 1.5 per second) for a minute straight. After that it goes idle for an hour, and the next 23 hours it's paused. You now pay for (60 * 1.5) + (3600 * 0.75) = 2790 vCore seconds. The price per second is €0.000134, so that's €0.37386 in total for the vCores that day.

Now, let's say the memory used in that minute is 6 GB on average. GBs are normalized into units of vCore by dividing them by 3. In that case, the maximum vCore usage would not be 1.5, but 6 / 3 = 2. The price would then be (60 * 2) + (3600 * 0.75) = 2820, so €0.37788. If this is the exact daily usage for a month you pay about €11.71 for that month.

As you can see, it's cheaper to use the database three times in an hour, have the hour of inactivity, and then pause for 22 hours, than it is to use the database three times throughout the day and have three hours of inactivity. It's possible to stretch the time before your database goes to auto-pause, but the minimum inactivity time is an hour.

Creating a serverless Azure SQL instance

Now that that's clear, let's create a serverless Azure SQL database and connect to it using SQL Server Management Studio. In the portal, go to SQL databases and click **+ Add.** Pick a subscription and resource group, and enter a database name. You may reuse the server we created in Chapter 2, or you may create a new one. If you create a new one, pick a server name that's unique across Azure. The server admin login can be your own name. The password needs to be at least eight characters and has some additional rules, which are explained when you enter the field. For our use case, **Welcome1!** is a good enough password (and easy to remember).



All services > SQL databases > Create SQL Database

Create SQL Database
Microsoft

Basics • | Networking | Additional settings | Tags | Review + create

Create a SQL database with your preferred configurations. Complete the Basics
provision with smart defaults, or visit each tab to customize. Learn more

Project details

Select the subscription to manage deployed resources and costs. Use resource
manage all your resources.

Subscription *	Visual Studio Enterprise – MPN
Resource group *	SuccinctlyRG
	Create new

Database details

Enter required settings for this database, including picking a logical server and
resources

Database name *	SuccinctlyDB
Server *	Select a server
	Create new
	The value must not be empty.
Want to use SQL elastic pool? *	Yes [✓ No]
Compute + storage *	Please select a server first. Configure database

Review + create | Next : Networking >

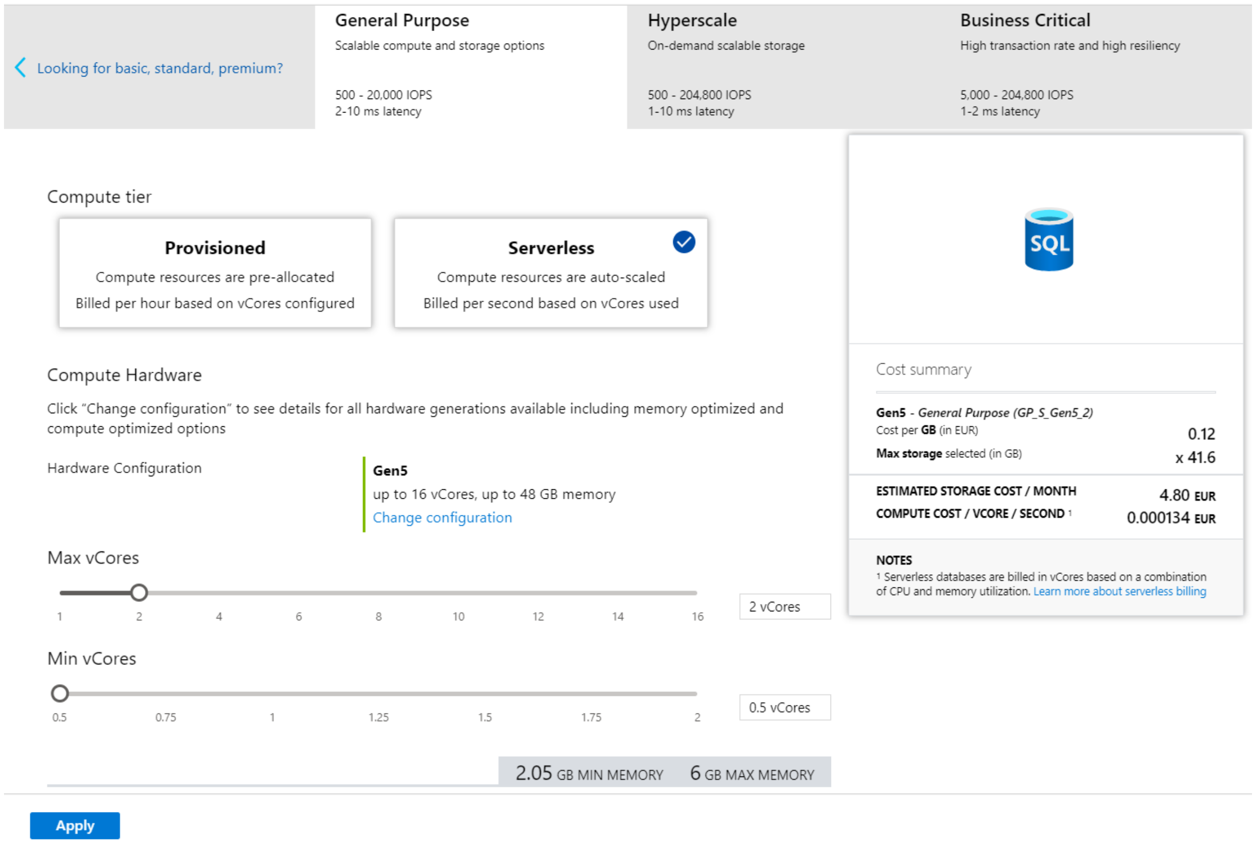
New server
Microsoft

Server name *	succinctlyserver .database.windows.net
Server admin login *	sander
Password *	•••••••••
Confirm password *	•••••••••
Location *	(Europe) West Europe

OK

Figure 36: Creating a New Azure SQL Database and Server

After you fill out all the fields and create the server, it's time to pick our serverless tier by clicking **Configure database** next to **Compute + storage**. The blade that opens has all the pricing tiers I discussed earlier. Select the **Serverless** compute tier and choose a minimum and maximum amount of vCores. Set the max vCores to **2** and leave all the other settings on the default value.



Looking for basic, standard, premium?
General Purpose
Scalable compute and storage options
500 - 20,000 IOPS
2-10 ms latency
Hyperscale
On-demand scalable storage
500 - 204,800 IOPS
1-10 ms latency
Business Critical
High transaction rate and high resiliency
5,000 - 204,800 IOPS
1-2 ms latency
Compute tier
Provisioned
Compute resources are pre-allocated
Billed per hour based on vCores configured
Serverless
Compute resources are auto-scaled
Billed per second based on vCores used
Compute Hardware
Click "Change configuration" to see details for all hardware generations available including memory optimized and compute optimized options
Hardware Configuration
Gen5
up to 16 vCores, up to 48 GB memory
Change configuration
Max vCores
2 vCores
Min vCores
0.5 vCores
2.05 GB MIN MEMORY
6 GB MAX MEMORY
Apply
SQL
Cost summary
Gen5 - General Purpose (GP_S_Gen5_2)
Cost per GB (in EUR) 0.12
Max storage selected (in GB) x 41.6
ESTIMATED STORAGE COST / MONTH 4.80 EUR
COMPUTE COST / VCORE / SECOND 1 0.000134 EUR
NOTES
1 Serverless databases are billed in vCores based on a combination of CPU and memory utilization. Learn more about serverless billing

Figure 37: Azure SQL Pricing Tiers

I couldn't fit the auto-pause delay and data max size on the screenshot in Figure 37, but they're under the minimum vCores, so don't forget to scroll to check them out. Then, click **Apply**.

Back in the SQL database creation blade, click **Next : Networking** > **Next : Additional settings**. Set **Enable advanced data security** to **Not now**. Then, click **Review + Create** and create the SQL database. The creation may take a few minutes.

Connecting to your database

Once the creation is complete, head over to your SQL Server (not your SQL database, which is a different resource). Then go to **Firewalls and virtual networks**. At the top of the blade, click **Add client IP** (never do this on a public wireless network). The IP address you're currently on is added to the list of allowed IP addresses, which means you can now access the SQL Server, and with that, the SQL database. Be sure to change the added rule name to something meaningful, like **Home** or **Office**. Save the changes.

Now that your IP address is whitelisted, you can access your database using SQL Server Management Studio. When you are prompted to connect to a server, use **[myserver].database.windows.net** with **SQL Server Authentication** and your username and password (in my example, that's **sander** and **Welcome1!**). You can find the server name on the **Overview** blade of your SQL database in the Azure portal. Under the **Connection strings** blade, you can find the connection string for the database, with only the password missing. If you forgot your password, you can go to the SQL Server in Azure and click **Reset password** on the **Overview** blade. Alternatively, you can set an Active Directory Admin. From here on, everything for your serverless database works exactly the same as for your regular Azure SQL database.

Tip: If you don't want to get SQL Server Management Studio, or if you have to look up something quick, you can use the Query Editor from the SQL database in the Azure portal. It's very limited in functionality, but you can run any query in it.

Summary

Serverless Azure SQL isn't for everyone, but when you have a database that's doing nothing most of the time, serverless is probably a cheap alternative for your database. Especially in the world of microservices, where each service can have its own database, this is an interesting alternative. Other than the pricing plan and the billing strategy, serverless Azure SQL works the same way as your regular Azure SQL databases.

Chapter 7 API Management

In the previous chapter, we saw serverless Azure SQL, a bit of an odd one out in the serverless Azure offerings. In this chapter we're going to have a look at API management, which may also not be your first thought when you think of serverless. However, as we'll see, serverless API management makes a lot of sense. As an added bonus, whether you use the serverless or the regular pricing tier, API Management has native support for your serverless APIs.

What is API Management?

Like the name implies, API Management enables you to manage APIs. It lets you track usage and configure access and rules, subscriptions, and products. So, let's say you have three APIs: an API for managing master data, an API for managing sales orders, and an API for managing stock. In your API Management service, you can now define products. A product is a logically grouped set of APIs.

So, say you have two products: one product for managing sales orders (which also includes the master data API, because that contains information on products and customers) and the stock API so you know what to sell. There's also the light version of the product, which only lets you keep track of the stock, but also requires the master data API. The following illustration hopefully makes this clear. The arrows have a "serving" relationship, so the master data API is serving the stock API.

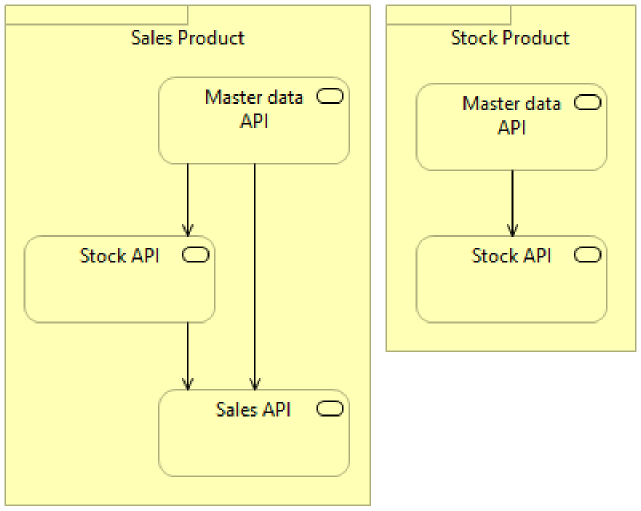


Figure 38: Two Products with the Same APIs

You now want to make these products available for customers. Let’s say you have two customers. Contoso wants the sales product, while ACME wants the stock product. You can create two subscriptions and associate the products with the subscriptions. The subscription offers two keys that you can send to your customers, who can then access the product the subscription is associated with. While subscriptions are normally scoped to products, you can scope a subscription to an API or to all APIs. Of course, you should be very careful with allowing access to all APIs.

In terms of serverless and microservices (and even SOA), API Management can have another function. Back in the day you had your one monolith that had everything you needed, but now you have lots of endpoints, all with different URLs. API Management can act as a gateway to allow uniform access to all your APIs.

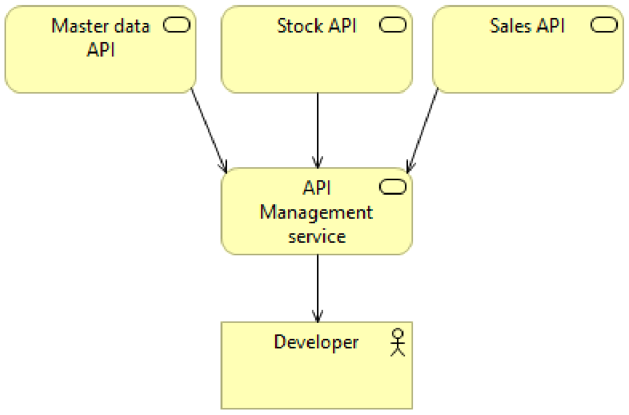


Figure 39: API Management as Gateway

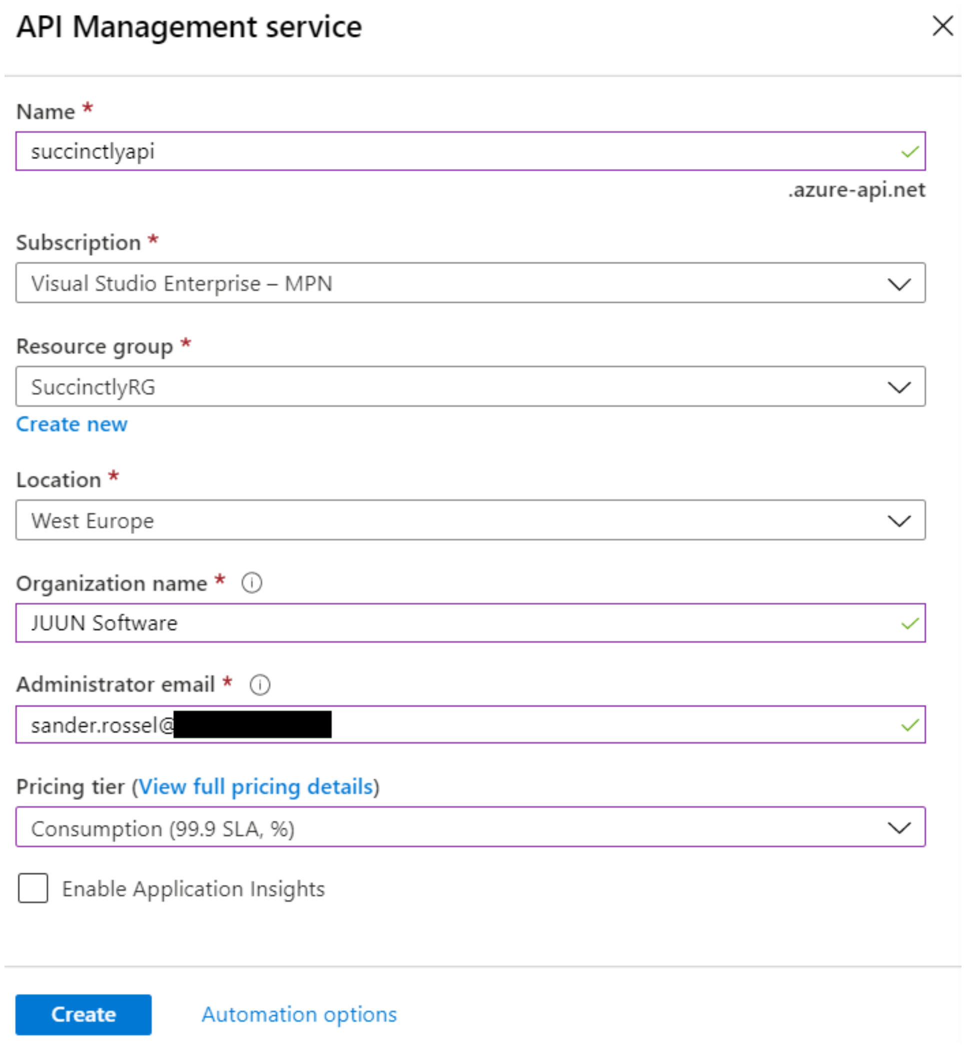
Instead of having to manage three different endpoints (which are tens to hundreds in real-life scenarios), you now only have to worry about one.

The serverless API Management offering is a bit more limited in functionality than the other tiers. User management and identities were removed from the serverless tier. In the other tiers you can add users to groups and give groups access to products and subscriptions. Also, serverless doesn’t have settings for notifications. A very important change in the serverless tier is that it doesn’t support the developer portal. With this, users could log in to a (customized) portal generated by your API Management service, which would let them browse your APIs and try them out. Despite the limitations of the serverless tier, it still offers plenty of functionality and at a good price.

Creating an API Management service

We’re now going to create an API Management service. In the Azure portal, go to all resources and find the API Management services, then click **Add**. The API Management creation blade has some fields that we haven’t seen before, your organization name and administrator email. You can put anything in organization name, so I’m putting in my own company, JUUN Software. The administrator email should be your own email address.

Creating an API Management service can take up to an hour, and when it's deployed an email is sent to your email address. Other API Management notifications will also be sent to this email address. The company name is used in the developer portal and in email notifications. Other than that, enter a name that's unique across Azure, select a subscription, pick a resource group, and set a location. Optionally, you can enable Application Insights. While I'd recommend it in production environments, leave it off for this demo unless you want to do some exploring on your own. Set the pricing tier to **Consumption**, which is the serverless offering.



Name * succinctlyapi .azure-api.net

Subscription * Visual Studio Enterprise – MPN

Resource group * SuccinctlyRG

Location * West Europe

Organization name * JUUN Software

Administrator email * sander.rossel@[redacted]

Pricing tier (View full pricing details) Consumption (99.9 SLA, %)

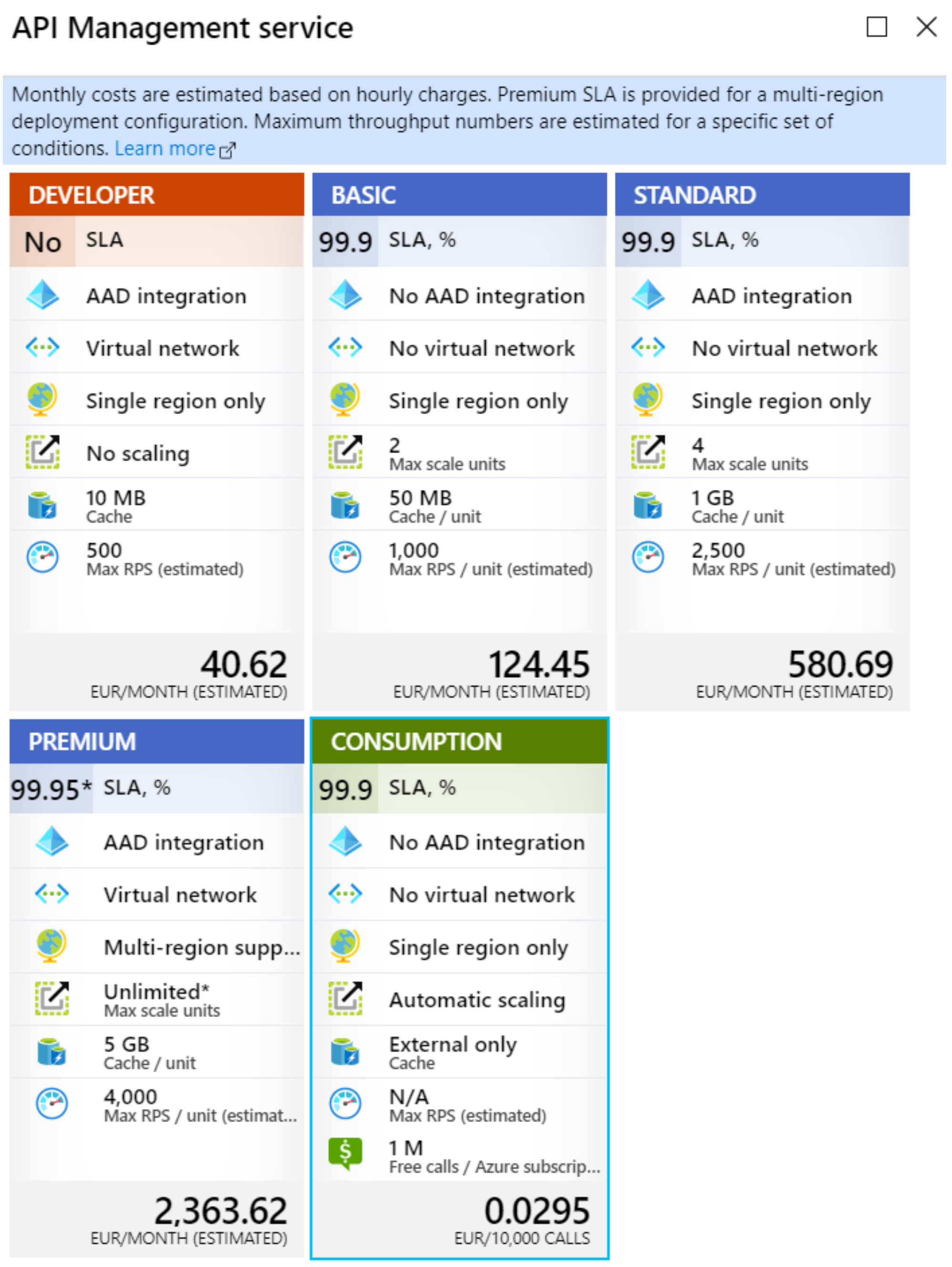
☐ Enable Application Insights

Figure 40: Creating a New API Management Service

After you've filled all the fields, click **Create**, and your API Management service is deployed. Since that may take a while (I once waited one and a half hours), we'll look at the pricing tiers while the service is being deployed.

Pricing tiers

The pricing for the API Management service differs greatly per tier and, unfortunately, none are cheap. The Developer, Basic, Standard, and Premium tiers are charged hourly, while the Consumption tier is the serverless offering that's charged by usage.



API Management service

Monthly costs are estimated based on hourly charges. Premium SLA is provided for a multi-region deployment configuration. Maximum throughput numbers are estimated for a specific set of conditions. Learn more

DEVELOPER
No SLA
AAD integration
Virtual network
Single region only
No scaling
10 MB Cache
500 Max RPS (estimated)
40.62 EUR/MONTH (ESTIMATED)

BASIC
99.9 SLA, %
No AAD integration
No virtual network
Single region only
2 Max scale units
50 MB Cache / unit
1,000 Max RPS / unit (estimated)
124.45 EUR/MONTH (ESTIMATED)

STANDARD
99.9 SLA, %
AAD integration
No virtual network
Single region only
4 Max scale units
1 GB Cache / unit
2,500 Max RPS / unit (estimated)
580.69 EUR/MONTH (ESTIMATED)

PREMIUM
99.95* SLA, %
AAD integration
Virtual network
Multi-region supp...
Unlimited* Max scale units
5 GB Cache / unit
4,000 Max RPS / unit (estimat...
2,363.62 EUR/MONTH (ESTIMATED)

CONSUMPTION
99.9 SLA, %
No AAD integration
No virtual network
Single region only
Automatic scaling
External only Cache
N/A Max RPS (estimated)
1 M Free calls / Azure subscrip...
0.0295 EUR/10,000 CALLS

Figure 41: API Management Service Pricing Tiers

The Developer tier is awesome, and has all features for the relatively low estimated price of €40.62, which already isn't cheap. Unfortunately, this tier isn't meant to be used in production, if only because it has no SLA (meaning no uptime guarantees). The Basic tier can handle more requests than the Developer tier, and supports some scaling and extra caching, but has no Azure Active Directory (AAD) integration and cannot be placed inside virtual networks. From this tier up, an uptime of 99.9% is guaranteed, which means the service can have an expected downtime of a little under nine hours a year (or about a minute and a half a day). This costs you about €80 a month compared to the Developer tier, though.

The Standard tier has even more scaling and caching options, as well as a higher throughput, and adds AAD integration, but it still can't be placed inside virtual networks. The price increase from the Developer tier is a whopping €460, or almost four times more expensive. The price doesn't get much better with Premium, which adds the ability to place your API management in your virtual network, allows it to connect to your on-premises network, and adds support for multiregion deployment, as well as more scaling and throughput, and an even higher SLA (if you deploy to multiple regions). But the price is an estimated €2,363.62 a month—not for smaller businesses or casual students. Needless to say, if you offer various APIs to a multitude of enterprise customers worldwide, the Premium tier is the one you want.

The Consumption tier, the serverless offering, has the 99.9% SLA, no AAD integration, and no virtual network, and can only be deployed in a single region. In this respect it's equal to the Basic tier. However, because the service scales automatically, when it comes to throughput, it virtually has no limits. The Consumption tier has no built-in caching, but every tier supports external Redis-compatible caches. This tier only supports external caching, which can make it a little more expensive if you want caching. The service itself has a million free calls per Azure subscription, and after that you pay €0.029516 per 10,000 calls. When an API Management service instance scales down to zero, billing stops until an instance is needed again. When the API Management is scaled down to zero, there's the usual startup delay when you make a new request. So, once again, check your usage and decide for yourself if this is the tier for you.

Configuring API Management

Let's configure API Management to get a basic understanding of the service. First, start by making three function apps through the portal and name them something like **succinctlymasterdata**, **succinctlystock**, and **succinctlysalesorders**. Next, create an HTTP function in each app. I've named them **`CreateCustomer`**, **`GetStock`**, and **`GetSalesOrders`**, respectively. Leave the default implementation.

Now, head back to your API Management service and go to **APIs**. As you can see right away, there's built-in support for Azure Logic Apps and function apps. OpenAPI, WADL, WSDL, and App Services are also supported. API discovery is supported through Swagger definitions. Click **Function App** and add your apps one by one.

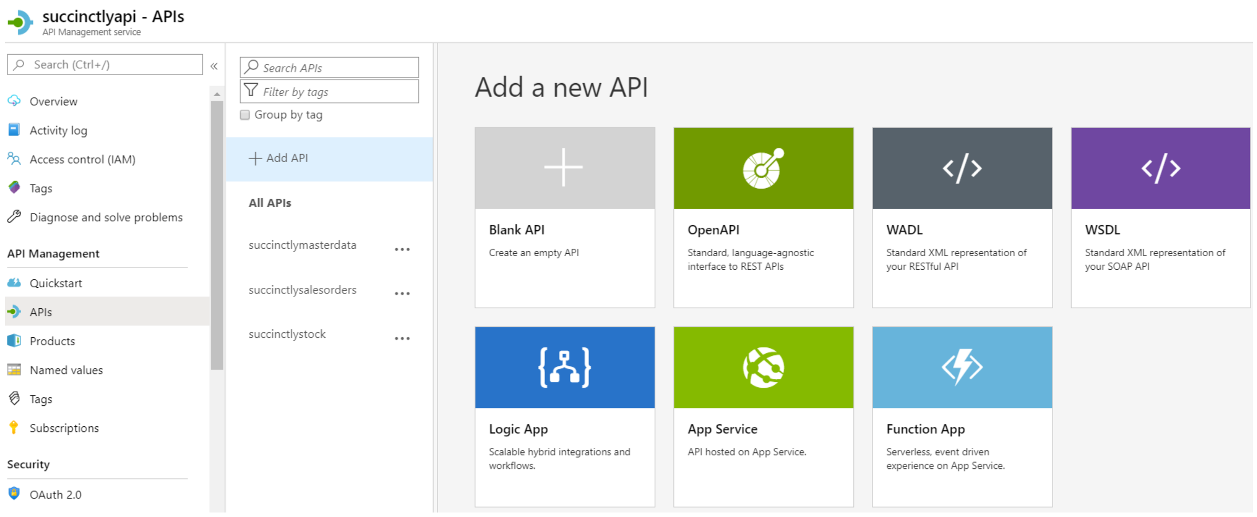
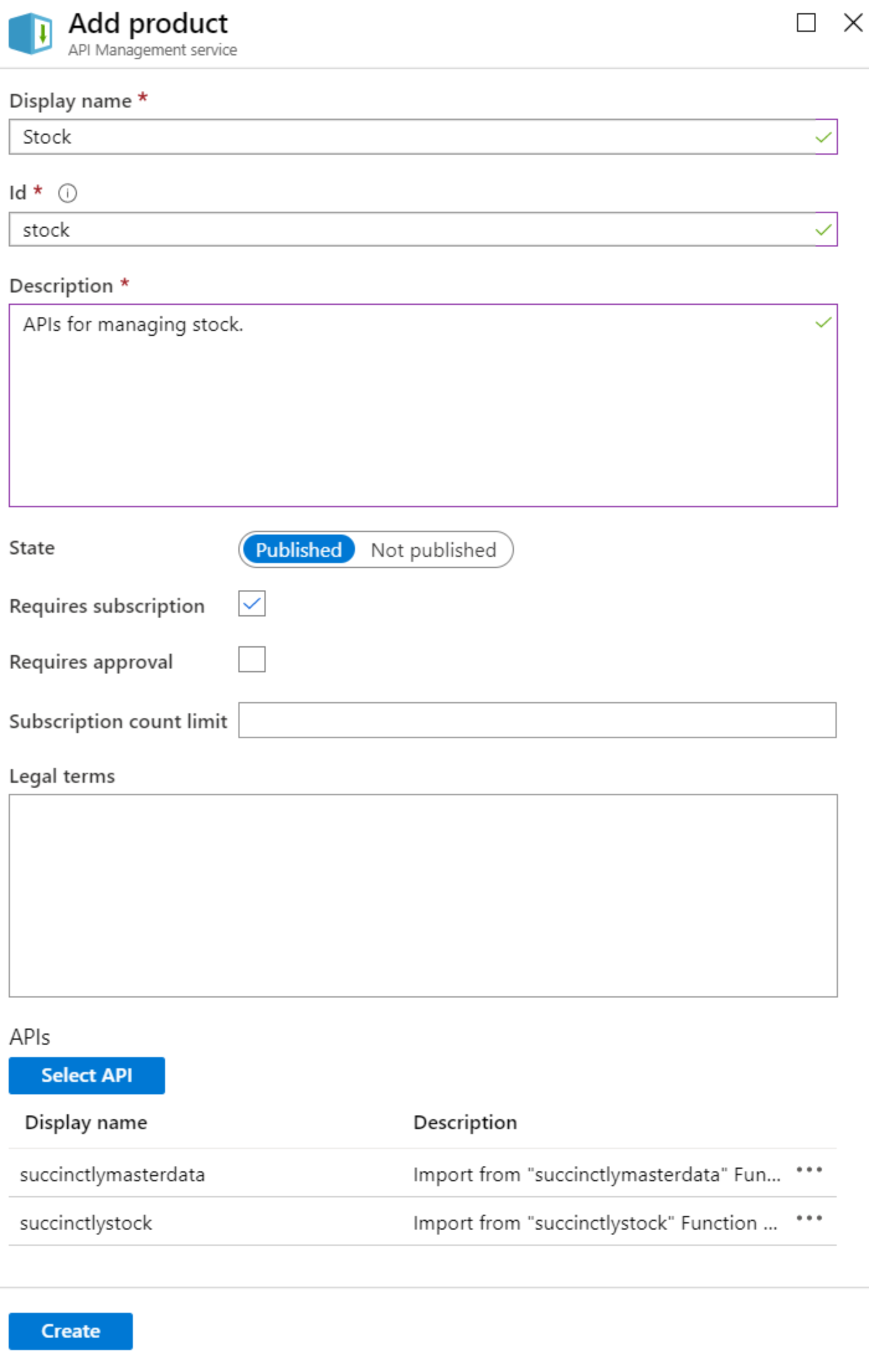


Figure 42: Add API

Now that your APIs are added, you can access them using your browser. Your API settings pages contain a base URL for the API. Normally, this would be https://[myapimanagement].azure-api.net/[myapi], so in this example, it's **https://succinctlyapi.azure-api.net/succinctlymasterdata**. We need to add the function name to that, so for example **CreateCustomer**. Of course, the default function implementation needs a name parameter, so use **?name=APIManagement**. On top of that, API Management needs a subscription key. Go to the **Subscriptions** blade in the portal, find the **Built-in all-access subscription**, click on the three dots, and click **Show/hide keys**. Copy the primary or secondary key and add it to the URL, like **&subscription-key=[your key].** The complete URL is now **https://succinctlyapi.azure-api.net/succinctlymasterdata/CreateCustomer?name=APIManagement&subscription-key=[key**]. When you put this in a browser window, you should see `Hello, APIManagement`.

Because we don't want to give our customers access to all our APIs, we're going to create a subscription with a product. First, create a new product. Go to the **Products** blade and click **+ Add**. Name the new product **Stock**. The ID will fill automatically. Provide a description, something like **APIs for managing stock**. Set the state of the product to **Published**. Click **Select API**, and then select the master data API and the stock API.



Display name *: Stock

Id *: stock

Description *: APIs for managing stock.

State: [✓ Published] Not published

Requires subscription: ✓

Requires approval:

Subscription count limit:

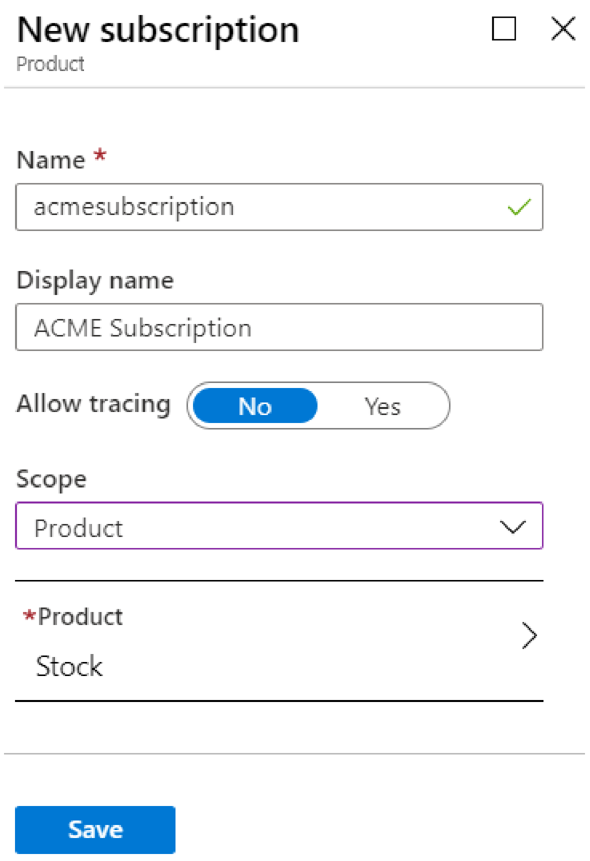
Legal terms:

APIs

Display name	Description
succinctlymasterdata	Import from "succinctlymasterdata" Fun...
succinctlystock	Import from "succinctlystock" Function ...

Figure 43: Creating a Product

Next, create a subscription. Enter **acmesubscription** as the name and **ACME Subscription** as the display name. Choose **Project** for the **Scope** field, and choose **Stock** for the **Product** field.


New subscription

Product

Name *

acmesubscription

Display name

ACME Subscription

Allow tracing [✓ No] Yes

Scope

Product

*Product

Stock

Figure 44: Creating a Subscription

To test if it works, use the same URL that we just used, but replace the key with the key of the ACME subscription. Now try to access the sales orders API by replacing **succinctlymasterdata** with **succinctlysalesorders** and **CreateCustomer** with **GetOrders** in the URL. You should get a 401, Unauthorized.

Code Listing 41

```
{
    "statusCode": 401,
    "message": "Access denied due to invalid subscription key. Make sure to
provide a valid key for an active subscription."
}
```

Unfortunately, it's not possible to switch products on a subscription once the subscription is created. Let's assume ACME wanted the sales orders API after all. Go back to your product and click **+ Add API**, and add the sales orders API. Wait a few seconds so the product can update, and then try the sales orders URL again. Instead of the 401 you should now get **`Hello, APIManagement`**. Obviously, you've now added the sales orders API to the stock product, which means everyone with that product can now access the sales orders API. You have two options to prevent this. One option is to create a new subscription, give ACME the new key, and don't delete the current subscription until they've updated all of their services with the new key (if you delete the subscription before that time, they won't be able to access your APIs at all). You can also cancel or suspend a subscription so you can activate it again later. The other option is to create a product per subscription, so you can always change the product without affecting other customers.

Note: I've mentioned that not all resources can be deployed using ARM templates. The API Management service can be deployed using ARM, but APIs, products, and subscriptions cannot. You can create some of these using PowerShell or the Azure CLI, but I haven't found a way to programmatically import your functions into API Management. However, since we're talking about securing APIs and managing access, which can change quite often, it may be best to manage this manually.

Summary

API Management is useful for grouping your APIs and managing access. With the serverless offering, you get a stripped-down version of the API Management service, but it still offers plenty of functionality for configuring access to your APIs through products and subscriptions.

Chapter 8 Conclusion

In this book we've seen various serverless offerings in Azure. With Azure Functions and Azure Logic Apps, we can run some code or workflow, triggered by various events, and we're only charged when the application runs. Service Bus offers queues and topics, which allows for decoupled and highly scalable applications. Event Grid offers subscriptions, topics, and domains, which allow for highly scalable, event-driven applications. All these services work well together. For example, all of these can connect with Service Bus, and Event Grid can send events to any of these services.

The Azure SQL and API Management serverless offerings are the odd ones out. They don't offer the cross-compatibility that the other services offer, and you'll always need to pay for your storage. These services weren't built to be serverless, but serverless tiers were added to them much later.

If you need to churn out CPU or memory 24/7, non-serverless resources are always the cheaper choice, if they are available. Yet, with the right use case, serverless is a potentially cheap and powerful tool in your toolbox.

In this book, I've wanted to give you better insight into serverless computing. Most blogs and articles don't get past a basic Azure Functions example, but serverless is more than that. There are other services that offer pay-per-use subscriptions, such as Azure Cognitive Services and Serverless Kubernetes, and more serverless services are being added. Hopefully, this book gave you the tools and insights to use them in your own applications.