Course Overview
Course Overview

Hi. My name's Mark Heath and welcome to my course, Azure Functions Fundamentals. I work as a software architect at NICE systems where I'm currently helping to create Azure-based digital evidence management systems for the police. Azure Functions opens the door to a style of programming known as serverless, which enables you to develop rapidly, scale out automatically, integrate easily with a host of other Azure services and all the while keep your costs to a minimum. In this course, I'm going to explain everything you need to know to get started with the new version 2 of Azure Functions. And we're going to see loads of demos of how simple it is to create your own functions. In particular, we'll create a simple pipeline of C# functions that demonstrates many of the supported triggers and binding types, including interacting with queues and blob storage, as well as sending emails. We'll also see how you can develop in Visual Studio or from the command line with a text editor if you prefer. We'll learn how to automate deployments, as well as how to debug and monitor our functions. By the end of this course, you'll be ready to create, deploy, and manage your own Azure Functions application. And I also hope that you'll have lots of ideas for how you can incorporate them into your own cloud-based applications as I really think they're applicable in a very wide range of scenarios. You'll be able to follow along with this course even if you're never used Azure before. And although I'll be using C# for most of the demos, you can actually use several different languages to create Azure Functions. So much of what we'll be learning is relevant whichever your language of preference is. So I hope you'll join me on this journey to learn about Azure Functions with the Azure Functions Fundamentals course at Pluralsight.

Introducing Azure Functions

Version Check

♫(Music)♫

Course Introduction

Hi. Mark Heath here, and welcome to this Azure Functions Fundamentals course on Pluralsight. This course covers the latest version 2 of Azure Functions. And in this module, I'll be introducing you to what Azure Functions are and what benefits they bring. The goal of this course is to teach you everything you need to know in order to get started using Azure Functions. But also I want you to get excited about the possibilities that Azure Functions open up because I think this is a technology that has the potential to transform the way you architect your applications. So I'm going to start off by explaining what Azure Functions are for and how they fit it with all the other multitude of services that Azure has to offer. So if you're already using Azure and you're wondering why would I need or want to use Functions, then I'll be explaining which scenarios would be a good fit for using them. But if you're completely new to Azure, don't worry. Azure Functions are really easy to get started with, and I'll be including plenty of step-by-step demos as we move through this course. Also in this module, I want to discuss the concept of serverless computing. Microsoft's own marketing material says

that with Azure Functions you can process events with serverless code. But what on earth does serverless mean? I'll be giving you a high-level overview of what this architectural style is all about. And we'll explore what benefits it can bring to your own applications. As we move through the rest of the modules in this course, we'll learn how to create functions in your language of choice, such as C# or JavaScript. And we'll be showing off the capabilities of Azure Functions, including various event triggers and bindings that they support. And we'll also see how you can deploy your functions, as well as how to manage and monitor them. So let's get started. What are Azure Functions, and what's the point of them?

Introducing Azure Functions

At the heart of Azure Functions is the idea of events and code. You simply supply some code, which is just a single function usually written either in C# or JavaScript, and you tell Azure Functions what event should trigger it. So for example, you can use Azure Functions to run a function every hour. So the event, or trigger in this case, is a scheduler. And I might use this sort of event trigger to run a nightly batch process that cleans up some data in my database. Another source of event that can trigger a function is new data becoming available. This might be a new message appearing on a queue or a new file being uploaded to Azure blob storage. And I might use this sort of event to trigger sending an email whenever a message appears on a particular queue. Another really useful example of an event that can trigger a function is an HTTP request. Whenever someone calls a specific URL, your function is executed and can respond to that request. So an example might be that I need to handle a web-hook callback from a payment provider like Stripe whenever I sell a product online. But we could also use this trigger type to implement a REST API. Now if you've used Azure before or any other cloud provider for that matter, you might be thinking hang on a minute. Can't we already do this? If Azure Functions is about letting your code run in response to event triggers, doesn't Azure already offer multiple ways of achieving exactly the same thing with offerings like virtual machines, web apps, and web jobs? After all, we can already use these to listen to queue events, respond to web-hooks, and do things on a schedule. So why would we need another way? Well, the best way of explaining this is to first consider the other ways of running your code in Azure, and then we'll be in a position to understand why functions are different and why you might want to use them instead. So bear with me for a few minutes. This won't take too long. But let's quickly discuss a few of Azure's existing offerings for running code in the cloud.

Running Code in Azure

Let's start off at the lowest level, which is using virtual machines. In Azure, I can spin up a virtual machine and install anything I want on it. I can install IIS and run an ASP.NET website on it. Or I could create Windows services to do some background work, like processing messages on queues or executing scheduled tasks. And this approach is known as IaaS, Infrastructure as a Service. And its great benefit is that it gives me complete control of the server. I can install whatever I want on it, including my choice of operating system, and I can fine-tune it to my exact needs, including choosing exactly how much RAM and CPU it has. But with this freedom comes several responsibilities. I must ensure that the operating system and the software I'm using is kept patched and up to date. If I want to scale out, I have to manage

that myself, deploying additional virtual machines and configuring my own load balancing between them. And this operational overhead comes at a significant cost. You need to have people dedicated to ensuring that your virtual machines in production are running smoothly. So wouldn't it be nice if we could hand that responsibility off to Microsoft and get them to manage our service for us so that we can just focus on writing our application. Another option for running code in Azure is using what's called Azure App Service. Azure App Service could be described as Platform as a Service or PaaS. In this model, unlike with virtual machines, the cloud provider takes responsibility for managing and patching the servers. All you need to do is provide the code for your website or background tasks. Azure Web Applications run on Azure App Service and are designed to make it super easy to deploy a website into Azure. You simply create a regular website, which might be using ASP.NET Core, but it also supports many of the most popular web development frameworks, including Node.js and PHP. And then you tell Azure to host your website in what's called a hosting plan. And we'll be talking more later about what a hosting plan is, but basically it gives you the freedom to host multiple websites on a single server for cost-saving purposes or to give your website its own dedicated server and turn on auto-scaling if your web application is in high demand. Azure App Service also supports the concept of web jobs. Web jobs offer a simple way to get your own background tasks, such as key processing, deployed to the same service in your hosting plan that are running the web application. So what we have with web applications and web jobs is a very programmer-friendly model that makes it super easy to create and deploy multiple websites along with background processing tasks and bundle them all up onto one server to keep your costs down, but with the flexibility to scale up as the demands of your application require. Now the reason I've taken the trouble to talk you through all this is that Azure Functions is actually built on top of the web jobs SDK and it's hosted on the App Service platform. So in many ways you can think of it as just another part of this same offering, but with a few additional powerful new capabilities that we'll see shortly.

What's Different About Azure Functions?

Let's talk now about Azure Functions, which were originally introduced as part of the App Service platform in 2016 and are now at version 2. What do Azure Functions offer that we can't already do with web applications and web jobs? Well first of all, Azure Functions offers a simplified programming model. Creating your first Azure function, as we're going to see in our next module, is trivially easy. All you have to write is the code that responds to the event you're interested in, whether that's an HTTP request or a queue message for example. And all the boilerplate code that you'd normally write to connect these events to the code that actually handles them is abstracted away from you, and this makes for a very lightweight and fast-moving style of development where you're really just focused on the code that meets your business requirements and eliminating a lot of repetitive boilerplate code. Another major benefit that Azure Functions offers is a consumption-based pricing model. With the more traditional Azure offerings we just discussed, virtual machines and web applications, you need at least one server running constantly, and you have to pay for that. But with Azure Functions, you have the option to select a pay-as-you-go pricing option. In other words, you only pay when your code is actually running. So if you're listening on a queue and no messages ever arrive, then you pay nothing. And the Azure Functions framework will automatically scale the number of servers running your functions to meet demand. So if

there's no demand, there might actually be no servers at all actively running your code. But the framework is able to spin one up very quickly if needed. And so this pricing model can result in dramatic cost savings compared to the more traditional approach of paying a fixed monthly amount to reserve one or more servers.

Azure Functions Pricing

Like Azure web applications and web jobs, Azure Functions run inside of what's called an App Service Plan. In addition to letting you use some of the existing pricing models for App Service, you also have access to what's called the consumption pricing plan. And here's how the consumption app service plan works. You're charged by two metrics, how many times your functions run, so the number of executions, and the time your function runs for in seconds multiplied by the RAM allocated. So the units of billing are GB seconds. Now the great news is that you get a very generous free grant, which is currently 1 million executions and 400 thousand GB- seconds per month. And this means you can achieve a lot with Azure Functions without paying anything at all. And even when you go over the free limit, the costs are very reasonable. And so with this pricing model, the main way to keep your costs down is by ensuring that when you do run a function, it completes as quickly as possible. And obviously, invoking the functions less frequently and keeping their memory requirements to a minimum will also help. In fact, the consumption plan actually limits function execution time to 5 minutes, and this is a good thing as it prevents costs from accidentally running away if you hit a deadlock in your code. And another nice feature is that you can set an optional maximum daily quota. Now you're not forced to use the consumption pricing tier. You can also host Azure Functions apps on a regular App Service Plan, which means you're back to paying for dedicated servers rather than paying for the duration of time your functions run. This makes your monthly costs predictable, and you can also choose from a variety of the pricing tiers that App Service offers depending on your needs. And it also means that you're free from that 5-minute execution time constraint as the length of time a function runs no longer has an impact on your costs. And you may also be interested to hear about the premium Azure Functions pricing plan. At the time of recording, this is only available in preview. But hopefully it will be publically available soon. And this offers advanced features like VNet connectivity, as well as improved performance. Of course, the pricing options do change from time to time, so make sure you visit this page on the Azure website, which has details of the pricing of the consumption plan and the free grant limits. And you can configure this page to show prices in your own local currency. If you want to see the pricing options for the dedicated App Service Plans, then you can visit this page on the Azure website. Although you need to be aware that even though there is a free and a shared tier, these aren't able to host Azure Functions apps. You'd need to choose the basic pricing tier or higher. Finally, I should mention that the Azure Functions runtime is available as a Docker container. This means that you can run an Azure Functions app on any computer that's able to run Docker, including on-premise datacenters or in other cloud providers. And so obviously if you chose this option, the pricing would simply be whatever you're paying for your container host. So as we've seen, there's a lot of choice about how you host your Azure Functions apps and how you pay for them. Well I suspect for most use cases that consumption plan is going to be the most attractive option.

Benefits of Azure Functions

So now we're in a position to understand the major benefits of using Azure Functions over the alternative options for hosting our code in Azure. First of all, it offers a very rapid and simple development model. In fact, as we're going to see in the next module, you can even create functions by coding directly in the Azure portal. And by eliminating all the boilerplate that you normally have to write just to listen for the event that triggers your function, you can focus instead on writing the code that really matters, the code that responds to that event. Second, because it's built on top of the Azure App Service, it comes with a very rich feature set, all of the core stuff from App service including continuous integration, the Kudu portal, Easy Auth, support for SSL certificates and custom domain names, easy management of application settings, and much more are all available to you with Azure Functions. And don't worry if you don't know what all of those are. We're going to be seeing several of them in action as we go through this course. Third, as we've just mentioned, there can be dramatic cost savings with the consumption plan as you're only paying for what you use. And this makes Azure Functions a very attractive proposition for startups. You pay almost nothing initially as you're within the free grant. And only when you attract a high volume of users do you need to start paying. Finally, this model takes away the need to manage and maintain servers that you have with virtual machines, and it even abstracts away scaling. You simply trust the framework to provide enough servers to meet the needs of your functions, which could be anything from zero to dozens or even hundreds of servers. And that brings us nicely on to the concept of serverless computing.

Introducing Serverless Architecture

Azure Functions is a serverless platform, or at least it's serverless when you choose the consumption pricing tier. And the concept of serverless has grown rapidly in popularity over recent years, and many cloud providers are offering similar serverless platforms, such as Amazon's AWS lambda. Now, of course, in one sense the name serverless is a bit silly because of course there are servers needed to run your code. But one of the key ideas behind serverless is that we want to delegate the management and maintenance of our servers to third parties so that we, as developers, can focus exclusively on the business requirements. So in a serverless architecture, you'd rely on third party platform or Backend as a Service offerings wherever possible, so for example using Azure Cosmos DB for your database instead of provisioning your own database server on a virtual machine or using Auth0 for your logging and authentication rather than hosting your own identity service. And there's a growing number of services that meet many of the common needs of modern cloud applications whether that be logging or email sending or search or taking payments. Now, of course, there will still be the need for you to write some of your own custom back end code, and that's where a framework like Azure Functions fits into serverless. Instead of provisioning a virtual machine to host your APIs or background processes, you simply tell Azure Functions which events you need to respond to, what to do when those events fire, and let the framework worry about how many servers are actually needed. And this model of lightweight hosting of functions is sometimes referred to as FaaS or Functions as a Service. On its own, Azure Functions isn't a serverless framework. But it fulfills the FaaS part of serverless. And so you could use it in conjunction with other cloud offerings to create an

overall serverless architecture. Now that was just a very brief introduction to serverless. So let's consider a simple, real-world example.

Serverless - A Real-world Example

A while ago, I needed to create a simple website on which I could sell a Windows application I'd written. The website was just some static HTML with a Buy Now button that integrated with a third party payment provider. So far, quite simple. I've got a web server responding to web requests from the browser and responding with HTML, CSS, and JavaScript. But I also needed to handle the web-hook callback from the payment provider when someone bought my product. So I added a web API method to handle that. And that needed to generate a license file and email it out. And so the web server posted messages to a queue. And because I needed somewhere to put the code that listened on the queues, well that ended up on the web server too. Before long, there were more and more little bits of code being added. The application let the user report an error, which called a web API. And the application needed to phone home to check that the license was valid. And then I wanted a nightly process to summarize all the activity from each day and email it to me. And so what had started out as a very simple website was growing into a full-blown application with lots of responsibilities. At one point, I was thinking about changing the website to use WordPress to make it easier for me to manage content. But I realized that I couldn't easily make that change now because there was so much custom .NET code that would need a new home if I switched the website over to PHP. And this is a familiar story to many developers. Your application starts small and so it only makes sense to have one server. But before you know it, that one server is doing everything, and you've ended up with a monolithic architecture that's hard to scale. Let's see how serverless could help us in this scenario. Serverless lets us break bits off our monolithic application. So let's start by saying that the new purchase web-hook is handled by an Azure function that posts a message onto a queue, and that message is handled by another Azure function that generates the license and puts it into blob storage. And that blog triggers another Azure function that emails it to the customer. The reporting error endpoint is another Azure function that writes a row into Azure Table Storage. The Validate License API is another Azure Function that performs a database lookup of that license code. And the nightly background process is another Azure function triggered by a timer that produces a report and sends it via email. Now we've still got our web server, and some serverless advocates might go even further and get rid of that too, replacing it just with static content hosted in blob storage. But even if we don't do that, our web server has now got a lot less to do. And we've managed to decompose our monolith into a set of loosely coupled functions. You could even think of them as microservices taken to the next level. And like with microservices, we've got freedom on deploy and scale them independently. Now there is one more concept that we need to understand, and that's the idea of function apps. Azure Functions lets you group related functions together into a function app, which allows them to share configuration settings and local resources. So in this application, it would probably make sense to put them all together into a single function app. And we'll be using function apps in our next module.

What Are Azure Functions Good For?

You might be wondering what sort of applications are Azure Functions and serverless architecture actually good for? Well, they're not necessarily the right choice for every application. But here's some examples of when they make a lot of sense. First of all, they're brilliant for experiments and rapid prototyping. It takes just a few minutes to get one up and running. So you can have a prototype application with a functioning back end in next to no time. They're also great for automating some of your internal development processes. So for example, you might want to use a web-hook to integrate a slack channel with your continuous integration server. And here it wouldn't make sense to deploy a whole web server just to handle a single web-hook. But with Azure Functions, you just need to write a single function, and you don't need to worry about what server that will run on. They're also great for decomposing or extending existing monolithic applications. You can break a bit out, say an existing queue handler into an Azure function. Or you can extend an existing application by adding on a couple of extra functions. And Pluralsight author, Troy Hunt, gave a great example of this a while back in a blog post where he shows how he uses Azure Functions to automate blocking and unblocking users of his web API to avoid server overloads. Just by adding a couple of Azure Functions to his existing architecture, he greatly reduced the load on his web server. And I've provided a link to this blog post in the course download materials along with a number of others that will give you inspiration of the types of scenarios in which Azure Functions are a good fit. Azure Functions are also great if you've got a number of queue handlers, but they might all need quite different scaling requirements. When you're managing servers yourself, it can be quite a headache to decide which handlers to group together onto the same server to balance the considerations of cost versus scalability. But with Azure Functions, you can let the runtime solve that for you. Another great use case is integrating systems together. Often you need to create intermediate adapters to connect together two systems, and Azure Functions is a really convenient place to do that. And of course finally, if the whole idea of serverless really appeals to you, then Azure Functions will let you build your entire application that way, composing the whole thing out of lots of loosely coupled functions.

Module Summary

We've covered a lot of ground in this module, so let's recap some of the key concepts that we've learned. An Azure function is simply a small piece of code in a language of your choice that runs in response to an event also called a trigger. An Azure function app is the unit of deployment and consists of one or more functions. Azure Functions is built on top of the existing Azure App Service and offers many of the same features and pricing options that you can use for Azure Web Apps. But it also offers what's called a consumption pricing plan where you pay only for what you use, and it has a generous free monthly grant, which means that during development, you're quite possibly paying nothing at all. Serverless describes a style of architecture where the service, scaling, and even wiring up of events are transparently managed for you so you can focus purely on writing the code that solves your business requirements. And Azure Functions doesn't force you to go all in on serverless. In fact, it's very easy to introduce a few Azure functions alongside a more traditional architecture, and they're also great for rapid prototyping. Now I know I've made you wait quite a while

to actually see an Azure function in action. So you'll be pleased to know that in the next module, we're going to see how easy it is to create your first Azure function.

Creating Your First Azure Function

Module Introduction

Hi, Mark Heath here. And in this module, we're going to see how easy it is to create your first Azure function. We'll start off by creating our first Azure function using a free trial experience designed to get you up and running really quickly. Our first function will be a simple web-hook, and we'll see how easy it is to code and test it within the portal. And you'll be able to follow along as we go, even if you don't have an Azure account. Then, we'll explore in more detail the options available in the Azure Functions section of the Azure portal, including how to create a function app and what options we can configure for our function apps. And I also want to give you a bit of a behind-the-scenes look at what's actually going on in your Azure Functions app. And so we'll take a quick look at what's inside the storage account that Azure Functions makes for us.

Creating Our First Function

We're going to jump straight into a demo now and create our first Azure function. So let's see how we can do that. Now you can actually try out Azure Functions for free without even needing an Azure account by visiting www.tryfunctions .com /try. So why don't you join me in your browser now, and let's create our first Azure function together. When we visit this page, we get offered the option to quickly start with a premade function. I can choose a webhook for a function that we can trigger with an HTTP request, but there are other options available as well. We can also choose between C# and JavaScript, which are the two best-supported languages on Azure Functions, but other languages are also supported. Let's accept the defaults, and you can see that it prompts me to log in. Now don't worry. We're not going to have to pay for anything. This just links your trial experience to a user account. I'm going to sign in with my GitHub credentials. And once that's completed, it's going to drop me into a window that shows my function app containing the function that we just created. As you can see, this Azure Functions trial experience just lasts an hour after which it will get completely reset. So this is a nice sandboxed environment that you can use to experiment without worrying about what you might break. And you can see over here on the left that it's showing me my function app, which has been given a randomly generated name, has got a single function in it called HttpTriggerCSharp 1. Now I know that's not the greatest name, but it'll do just fine for this demo. And here on the right, we get to see the code for this function, which is in a file called run.csx. Now I'm not going to go into too much detail here because later on in this course, I'm going to show you a better way to create C# functions. But the basics are that this run method is going to get run whenever our function is triggered. And when the function runs, it tries to find a query string parameter called name. And if that wasn't supplied, it looks to see if the body of the request contains a name. Then if no name was supplied, it will return a HTTP 400 Bad Request. But if you did supply a name, then we return 200 OK with a Hello message in the response body. Now one

really nice thing about this portal experience is that we've got a code editor built right in. So if we wanted, we could edit this function right here. Now obviously you wouldn't want to write your production code in the Azure portal. But as a quick way to try out Azure Functions, this is really convenient. If I expand this View Files tab over on the right, I can see the files that make up this function. There's the run.csx, which we were just looking at, a readme file that we don't need to concern ourselves with now, and there's also a function.json file. Now this is important as this is the file that defines each Azure function in a function app. It specifies important things like what event triggers this function. In this case, it's an HTTP trigger. It also specifies that we're going to support both the get and post HTTP methods. And it says that this function has an authorization level of function, meaning only people who know a special secret code can call it. Normally, you won't need to generate one of these function.json files yourself. In most cases, the tooling will generate it for you. But it's still really useful to know about it as it's a fundamental part of how Azure Functions are defined and can be useful for troubleshooting. And the editor actually allows us to add more code in here too. So we could put in more C# files if we wanted. Although the spirit of Azure Functions is to keep your function code as small and lightweight as possible. So hopefully you won't find yourself needing to do that too often. There's also a handy test window, which makes it really easy for us to trigger our function. I can choose what HTTP method to choose. Post is fine. And I can set up the request body. Let's set the name to Pluralsight, and we'll click Run. As you can see, quite soon it tells us that the function call succeeded with a 200 OK response. And it pops up this Logs window showing us the live log output from our function. We can see here that the log output shows us when the function started and stopped, as well as showing the actual log message that we omitted at the start of our function in the C# code. And if I scroll down in this test view, it also shows us the message that our function returned in its response body, Hello Pluralsight, which is what we were expecting. Of course, since this is an HTTP-triggered function, I don't have to use this test window. If I know the URL, I can call it from anywhere. If I click this Get function URL link, it shows me the URL for this function. And you can see in this dialog, the domain name is made up of the randomly generated function app name followed by azurewebsites.net. You can also see the function name in there, HttpTriggerCSharp1, and you can also see the security code that we do need to provide, or we'll get a 401 Unauthorized response. Let's copy this link to the clipboard, and I'll open a new browser tab and visit that link. As you can see, we get the warning message saying that we need to provide a name in the query string or in the request body. So let me do that by appending a query string parameter for the name with a value of Mark to the end of our URL. And sure enough, once we run this, we get Hello Mark back in the response body, which is defaulted to XML simply because there was no content negotiation header on the request. And so it show it chose the default of XML instead of JSON. We've just seen how we can use the tryfunctions.com website to get a free 1- hour trial experience. We saw how easy it is to create our first function based on a template, and we saw that we can view and edit code in the portal. We also have had our first look at the function.json file, which is a file that every Azure function will have. We saw how we can view the diagnostic log output from a function execution in the portal, and we also saw how we can test our function from within the portal, as well as how to access the URL for a HTTP-triggered function, including its secret code.

Creating Functions from Templates

In this demo, we'll create another function by using a different template. And this time, we'll also use a different language, JavaScript. So what happens when we want to create another function? Well, we can do that quite easily because what we've got here is an Azure function app. and as we said in the last module, an Azure function app can continue many functions. So back here in the portal, we just need to click this plus button to the right of Functions to create a new function. And you can see here that we've got a whole host of options for different types of functions we can create. So there's an HTTP trigger, which is what we used last time. But I can also choose queue triggers, blob triggers. And if I scroll down, you can see a whole bunch of other templates are available. And function templates serve as a great quick start for you to save you having to remember all the syntax for the various triggers and bindings that you might want to use. And we'll be making a lot of use of templates as we go through this course. You'll also notice that these templates have been offered to us in three languages, C#, F#, and JavaScript. And if we were to enable experimental languages, we'd see even more choices here. Azure Functions is designed to be extensible and to support many languages. And at the time of recording, there's also Python and Java support available. And a few other options, like TypeScript and PowerShell, are currently in development. Now in this course, we're going to focus mainly on C#. But I will show you in this demo how to create a JavaScript function. and so let's create a timer-triggered function we're just going to run on a schedule. This opens a dialog where I can choose the name for our new function. TimerTriggerJS1 is fine. And we can also set a cron expression that governs how often this function will run. If you've not see cron expressions before, they can look a little confusing. But this simply means that this function is going to run every 5 minutes, which is just fine for our purposes. Let's click Create to create this new function. Now one important thing that I should mention at this point is that the reason that we were able to create a JavaScript function alongside a C# one is that this trial experience website is still using version 1 of Azure Functions, which allows you to mix and match languages in the same function app. With version 2 of Azure Functions, the language extensibility mode has been rewritten, which brings better performance, but at the cost that you have to pick a single runtime for your function app. You'd either be using .NET and running C# or F# functions or Node if you were running JavaScript functions. And so that's why you can see this warning message telling me a new version of Azure Functions is available. Anyway, you can see that for a JavaScript function, we have an index.js file, which again we could edit in this window if we wanted. When the timer fires, our function is called. And all this function does is logs a message. It's also able to detect if the function invocation was running late for some reason. In JavaScript Functions, you get passed a context object, which you can use to log messages. And you call done on the context when your function is complete. If we go over to the View files tab, we can see that this function also has a function.json file in addition to index.js. As we said earlier, all Azure functions have a function.json file, and this one contains the same information. What triggers this function, in this case, is a timer trigger and any configuration specific to the trigger type, which for a timer-triggered function is the cron expression defining the schedule that we're going to run on. Now we could wait 5 minutes for this function to run, but we can also run it on demand by clicking the Run button. So let's do that. And as you can see, down here in the logs, we see messages indicating that our function has started and completed, as well as the log message we omitted in the function code itself. And there's still a couple of minutes to wait before our function is actually due to run. It runs every 5 minutes, so I'm going to fast-forward in time a

little. And if we watch the logs, we'll see that almost exactly 40 minutes past the hour, our function runs again. Obviously, a real-world scheduled function would do something a bit more interesting and interact with other services, and we'll be learning how to do that later on in this course. But I hope these quick demos have shown you how easy it is to try out Azure Functions and get your first function up and running. In that demo, we saw how easy it is to create additional functions in your function app based on a whole host of built-in templates, and we created a JavaScript timer-triggered function and saw how we could run it on demand.

Creating a Function App in the Portal

Our previous demos just used the completely free Try Azure Functions experience, which lets you play around for up to 1 hour with no financial commitment at all. But in this demo, we're going to create our own function app. And for that, we will need to have an Azure account, which is free to create. But we're going to be using an existing one. We'll visit the Azure portal and create a new function app. And we'll see that we've got exactly the same experience that we saw before with easy editing and testing of our code right within the portal. And we'll explore around a bit more to see what additional options we have at our disposal. So here we are in the Azure portal, which you'll be already familiar with if you've got an Azure subscription. You can visit the portal at portal.azure .com. And even if you don't have an Azure subscription at the moment, you can sign up for free, and that will actually give you some free credit to use in your first month. So how do we create a new function app? Well, we can press plus here in the top left to create a new resource. And in this window, we can either search for function app or navigate to it by looking in the Compute category for function app. And this brings up a window that lets us enter options for our function app. We need to give our function app a name, and this name is going to form part of a domain name so it does need to be unique. I was thinking of calling this function app Pluralsightfuncs, but you can see that that name has already been taken. So let's keep appending numbers until we find one that's free. And there we go, Pluralsightfuncs1234 is free. Next, I can choose which Azure subscription to use if I've got more than one. And I can put it inside an existing resource group. But here we're going to create a new resource group, and the default name it suggests is fine. You also notice that I can choose an operating system to run on. The original version 1 of Azure Functions was based on the .NET Framework, so it was tied to Windows. But version 2 of Azure Functions is based on .NET Core, which is cross-platform, so it's able to run on Linux. However, as you can see at the time of recording, Linux support is still in preview. So we'll choose Windows. Next, I can choose an app service plan. This allows me to choose between the consumption plan, which uses the serverless billing model, or I can just select App Service Plan, and this would allow me to host this function app on any app service plans that I've already created in this subscription. And as we discussed in the last module, during this we'll use the dedicated VMs that were already allocated to the App Service Plan to run the functions on top of whatever else they were already doing. However, in most cases, choosing the consumption plan makes a lot of sense as you can take advantage of the free grant and all the power and flexibility of the serverless model. If I choose the consumption plan, I need to pick a location, which really should be in the same region as any other Azure resources that I'm planning to use from within the function. And as you can see, there's a lot of supported regions. I'm going to pick West Europe. The next option is what runtime stack we're going to use. This is something we touched on earlier. With Azure Functions version 2, the expectation

is that you typically use the same language for all your functions. So if I'm writing C# or F# functions, then I pick .NET. If I'm writing JavaScript, then I'll pick the JavaScript option here, which will also be able to support functions written in TypeScript when that becomes available. And you can also see here that it's offering me Java as a preview. And by the time you watch this, there may be other runtimes available. For example, support for Python is well under way, so you might see that listed here. I'm going to be creating C# functions, so I'll pick .NET. Finally, a storage account will be created for you. This is for Azure Functions to store information relating to your function app. I'm going to be a bit nosy in a minute and take a look at what's inside here. You'll notice it does offer us an option to use an existing storage account, but I recommend that you let it create a new storage account. And you can see here that it's auto- generated a unique name that it can use for that storage account. You can also see that it's offered to create an Application Insights nstance for us, which is going to store log and diagnostic information. This isn't strictly required, but it does come highly recommended as it provides a much richer logging and diagnostics experience that we're going to be seeing later. So let's go ahead and create this function app by clicking Create. It will take about a minute to provision, so I'll fast-forward to when it's done. And we get a notification telling us that deployment is completed. When we click on it, it launches us into a function app navigation experience, which is actually the same as we saw earlier when we used the tryfunctions website. And on this same page, we get to see some basic information about our function app, including the URL, which will be the base address for any HTTP-triggered functions that we create. So we've seen that if you have an Azure account, you can create a new function app directly from the portal at portal.azure .com. And we saw that when you create a function app, you can specify the function app name, subscription, resource group. You can choose the operating system, Windows or Linux, the hosting plan, allowing us to choose between the serverless consumption plan or regular app service plan, and the runtime stack, which governs what languages we're writing our functions in. An Azure Functions app requires an Azure Storage account, and we usually create a dedicated one that can be associated with our function app. And we can choose whether to create application insights if we want the improved logging experience. So next, let's look around in a bit more detail at the function that we've just created.

Exploring the Function App

In this demo, we'll be exploring the function app that we just created in the Azure portal and seeing some of the settings and options that are available to us beyond simply editing the code for our functions. Let's start off by navigating into our resource group and seeing what got created. There are four resources here, the function app itself. But there's also a storage account, an Application Insights instance, and here's the App Service Plan, which governs pricing, and this is the consumption plan. Let's navigate back into our function app, and we'll take a look at some of the settings. If I visit the Platform features tab, you can see that there's a very large number of things that we can configure in here related to development, networking, monitoring, and much more. As we've said, Azure Functions is built on top of Azure App Service, and that means that the majority of features that you can use with Azure web apps are also available to function apps. Let's go in to look at the function app settings. This lets us access some key settings related to the function app as a whole. And you can see that here we're using version 2 of the Azure Functions runtime, which is the latest version, and I recommend you use it if possible. We've got the ability to set a daily usage

quota, which can protect you against accidentally running up a large bill. And the function app edit mode is shown here as read/write, meaning that we can create and edit functions in the portal if we want to. But we're actually not going to be doing that anymore in this course. Instead, we're going to write our function code in a regular code editing tool, and then we're going to push it to Azure when we're ready. In this view, we can also access or renew the secret keys that can be used to call our functions. Let's navigate to the application settings next. In here, we get to see a whole bunch of settings. And if you're wondering why there's things like PHP version, which isn't even currently a supported language for Azure Functions, that's because these settings come from the same underlying platform that web apps use, and they do support PHP. But generally, you won't need to modify most of the settings in here. The most relevant part of this page is the subsection called Application settings, and this contains settings that are used by the Azure Functions runtime, but you can also add your own settings in here as well, which can be accessed from within your functions. And anything set here will appear as environment variables that you can access in your code. And these settings include things like the Application Insights key, the version of the Azure Functions runtime, which has the value of ~2, which means use the latest release of Azure Functions versions 2, and the FUNCTIONS_WORKER_RUNTIME setting is set to dotnet as we're going to be writing C# functions for this function app. And you'll also see that the connection string to the storage account that we created is here as well. Now I'm not going to take you through all of the features shown here. That will take a long time and probably be quite boring. But it's worth you taking a little bit of time to look around here in the portal and familiarize yourself with what's available when you get the chance. And we will be returning here and using some of these other settings later on in this course.

Exploring the Storage Account

Now one final thing before we wrap up this module. Let's be nosy and take a peek at what's inside the storage account that got created for our app service. And just to make things a little bit more interesting, I've created two functions just like we did in the tryfunctions website, one is HTTP-triggered, and the other is a scheduled function, and both are in C#. So let's navigate to the storage account associated with our function app. We'll navigate to the resource group that contains our storage accounts and then into the storage account itself. And there's a really nice new feature in the Azure portal called Storage Explorer, which allows us to look inside a storage account. We can see that there are two blob containers, and these are used by the Azure Functions runtime for things like holding locks, timers, and secrets. But there's also a file share in here. Let's explore inside this file share, and we'll navigate into the site, wwwroot folder, and we can see that here's the code for our function app. There's a host.json file, which can be used to configure some global settings for your function app. And then we can see that each function has got its own folder. So if I navigate into the HTTP-triggered function folder, we can see the function.json file and the run.csx file are stored in here just like we saw in the tryfunctions website. Now there are no queues or tables in use currently in this storage account, but some of the more advanced features of Azure Functions, such as durable functions, can make use of these. Now most of the time, you won't need to look into this storage account. But it's nice to know that that option's available if you want it. In fact, now is a good time to mention that all of the code for Azure Functions is open source. It's all on GitHub in these repositories. So you can find out as much as you'd like about how it works by exploring these repositories. But you can also raise issues here if you're

having problems. And you'll find that the development team are very responsive. Even better, because Azure Functions is open source, it means that you can contribute your own features.

Module Summary

Okay, let's wrap up what we've learned in this module. We've seen that the try Azure Functions experience gives you a no commitment, one-hour, free trial where you can give it a go for yourself and code in a variety of languages directly in the portal. We've seen how if you do have an Azure subscription, you can create your own function app within the portal. We saw how you can access the Function App settings directly to modify your application settings and connection strings. And we also looked inside the storage account to get some low-level insight into what's going on in your function app. In the next module, we're going to start learning about all the triggers and bindings that Azure Functions support. And we're also going to see some better ways to write Azure functions that don't involve coding directly in the portal.

Understanding Triggers and Bindings

Module Introduction

Hi, Mark Heath here. And in this module, we're going to learn more about the various types of triggers and bindings available in Azure Functions. In the last module, we already used a couple of the available trigger types. We created a schedule function triggered by a timer and a web-hook triggered by a HTTP request. But there's actually loads of additional trigger types supported in Azure Functions, such as responding to queue messages or new items appearing in blob storage, and even new documents being added to a Cosmos DB database. In addition to triggers, Azure Functions provides input and output bindings, which make it really easy to connect your functions to all kinds of external resources. Whether your function needs to post a message to a queue or write a file to blob storage or send an email, bindings can greatly reduce the amount of code you need to write to achieve this. And in this module, we'll be making use of some of these triggers and bindings as we start to build a series of functions that will work together as an order processing pipeline. I'll also be taking the opportunity to introduce you to the Azure Functions core tools, which are a cross-platform command line set of tools that enable us to develop our Azure functions on any platform. And as we use these triggers and bindings, we'll see how they're represented in the function.json file. This documentation page on docs.microsoft .com is a great place to bookmark if you want to get an idea of all the various triggers and bindings supported by Azure Functions. If we navigate down to the Supported bindings section, we can see that there's lot of supported triggers and bindings, and there's a slight difference between what you can use in version 1 and version 2 of Azure Functions. And this is a very comprehensive list, especially when you consider that some of these triggers, like HTTP and Event Grid, are very general purpose in nature, allowing you to easily integrate with almost any external system, even if it isn't natively Azure Functions aware. From this page, you can drilldown into each of these different binding types individually and discover all of the options available for your language of preference. For example, if I navigate to the Cosmos DB binding, we'll see there's a page for

how bindings work with version 1 and version 2 of Azure Functions. And if we look at the version 2 page, we'll see that examples are provided, not just in C#, but also for the other main supported languages. So this really is a very comprehensive reference. Now in this course, we don't have the time to cover all of the permutations of different bindings, triggers, and languages because there's so many of them. But I do want to introduce you to some of the most important ones. If you remember earlier in this course, we looked at a real-world example of how we might refactor an existing ecommerce application to use Azure Functions. There was a web-hook from a payment provider that posted a message into a queue. And when that message was processed, it triggered the creation of a license file, which was stored in Azure blob storage. And then the creation of that license file blob resulted in an email being sent to the customer. And there were several other functions in this example as well. But in this module, we're going to create the first of these functions, which means we'll be working with the HTTP trigger and the queue output binding. And then later on in this course, we'll be building out more of this pipeline to make use of additional triggers, as well as input and output bindings, including blob storage, table storage, and sending emails. And we'll get to work creating this pipeline soon. But first, let's talk in a bit more detail about the language support in Azure Functions and the two different ways of writing C# functions.

Supported Languages in Azure Functions

I've mentioned a few times that Azure Functions supports several languages. And a good web page to check for the latest information is here on the official Microsoft documentation site. As you can see, there were some changes to the supported languages in the move from version 1 to version 2 with a few of the experimental options being dropped, but with Java support being newly added. And just like the rest of the Azure Functions runtime, the language workers are open source. So you can always visit the repositories on GitHub to find out the latest news and get involved in design discussions. In fact, if we visit the Azure organization page on GitHub and search for repositories with the word worker in their name, we can see the Python, Java, and Node.js workers here, as well as a PowerShell one that's still in its early stages at the time of recording. So there's plenty of language options. But I also want to highlight the fact that there are two ways of writing C# functions. And you can see here in the documentation table of contents that there's C# class library and C# script. So what's the difference? Well, C# script is the way of writing C# functions that we saw in our last module when we wrote code in the Azure portal. C# script files are compiled on the fly and lend themselves well to programming in the portal. However, this is not the most efficient approach to running C# functions. It's better to precompile your functions into a DLL, and that's what the class library approach is. In fact, you'll sometimes hear this approach referred to as pre-compiled functions. With the C# class library approach, your functions are created in a regular C# project that compiles to a .NET Core library. And with this approach, a function.json file will get automatically created for each of your functions and will point to the method in the DLL that implements your function, and this is the approach that we're going to be using for the rest of this course. Let's see next how we can install the tooling that helps us do this.

Azure Functions Core Tools

What do you need in order to develop C# class library Azure Functions? Well, there are two options. First of all, you can use Visual Studio, either 2017 or 2019, which is just about to be released at the time I'm recording this. And if you're using Visual Studio, you do need to make sure that you've selected the Azure development workload, which will enable all the tools you need. And then you'll be able to create an Azure Functions project and run and debug it locally. And this is a great option if you're on Windows and particularly if you're .NET developer already very familiar with Visual Studio. But Azure Functions development can also be done cross-platform with the aid of the Azure Functions core tools. And these give you a command line experience that simplify creating new Azure functions apps, adding new functions from templates, as well as other useful tasks, like deploying your application. And if you're planning to use these tools to develop class library C# function apps, then you will also need to make sure that you've got the .NET SDK installed, which is also cross-platform, and you can download that at dot.net. As you'd expect, the Azure Functions core tools are open source. And you can find the project here on GitHub in the azure-functions-core-tools repository. And there's a page here on the docs.microsoft .com site where you'll find instructions for installing the Azure Functions core tools on Windows, Mac, or Linux. I'm on Windows, and so I've used this npm install command to install it on my machine. And if you'd like to follow along with the demos in this module, then why don't you get the Azure Functions core tools and the .NET SDK installed now on your machine. I'll also be using Visual Studio Code, which is a really great open source cross-platform text editor that's got excellent support for Azure Functions development. So you might want to consider installing that too. Once you've installed the core tools, you can check that they're working by typing func version, and this will show us the version number of the core tools that we've got installed. If I just type func, then I get some helpful output showing me what subcommands are available. Let's use the Azure Functions core tools to create a new function app. I need an empty folder, so let's create one called Pluralsightfuncs. And we'll change directory into that folder, and we'll run the func init command, which creates a new function app. It prompts us to select which worker runtime we'd like, and so I'm going to chose dotnet. Now if we take a look at what it's created, we can see a csproj file, which is just a C# project file, a host.json file, which is where some configuration settings for your function app as a whole can be stored, a gitignore file, which is useful if we're going to initialize a Git repository here, and a local.settings .json file, which is a place to store the settings that you want to use when running locally. You'll also notice there's a vscode folder, and this contains some settings that will prompt Visual Studio Code to install some useful extensions that improve the Azure Functions development experience. So let's open this folder in Visual Studio Code. And if we take a look at the csproj file, we can see that it's a regular .NET Core project file with a few extra settings related to Azure Functions and what files should be copied into the output folder. We're referencing the Azure Functions SDK as well. And the host.json file and the local.settings .json file are populated with sensible starter values. You'll notice that Visual Studio Code pops up some messages because it's detected that this is an Azure Functions project and that we need to restore the NuGet references. So I'll accept both of these prompts. And if we take a look at my Visual Studio Code extensions and search for functions, you'll see that there's an Azure Functions Visual Studio Code extension, which I've already got installed. and if you don't have it, you'll be prompted to install it when you open this project in Visual Studio Code. And so this makes Visual Studio Code a great environment

for cross-platform Azure Functions development. So now we've got our development environment all set up, we're ready to create some functions.

Demo: Webhook Trigger

The first function we're going to create is for a webhook coming in from a third party payment provider, and we're going to handle that by posting a message onto a queue. The first step is to create a HTTP triggered function, and we're going to do that by making use of the Azure Functions core tools, and we're also going to see how we can test and run our function locally. Here we are back in our command prompt. And to add a new function to our function app, we simply need to type func new. And, you can see, this gives us the option to select from a whole bunch of example function templates. And the nice thing about these templates is that not only will they give us a quick start to show us the syntax needed for each of these trigger types, but they'll also install any necessary additional NuGet packages that are required to support the triggers. Now for this function, we just need an HTTP trigger, which we're going to use to receive the web-hook from our payment provider. So I'll select that. And it prompts us for a function name, so I'll call this function OnPaymentReceived. And now it's going to create some template code for our function. By the way, if you're wondering what's going on with my command prompt here, I've got a PowerShell extension called PoshGit installed, which gives me useful information about the Git repository in my working directory, such as what branch I'm on and how many files are currently modified. Let's jump back into Visual Studio Code, and we can see here that there's a new OnPaymentReceived C# file, and so let's take a look at what's inside there. We can see here we've got a static class called OnPaymentReceived, and there's a static method in here called Run. And here we can see the main difference to the C# script functions we created in the portal. This approach is using attributes to hold information about the triggers and bindings for the function. There's a FunctionName attribute. And since every Azure function needs to have a trigger, we've got a HttpTrigger attribute defined for the HttpRequest parameter. And this attribute lets us configure various settings for the trigger. We can choose the AuthorizationLevel. We're going to leave it as Function, which means that a secret code will be required in order to call this function when it's deployed in Azure. If we didn't want that and we wanted to make it publically available, we could just use the anonymous AuthorizationLevel. But with this option, we'd need to share the full URL of this function including the secret code with our payment provider to enable them to call our function. We can also specify which HTTP methods we want to accept. By default, it's allowing the get and post methods, but we only want to support post. So let's delete get. Finally, there's an optional Route template. By default, the URL for a HTTP-triggered Azure function will include /api and then the function name, which in our case is called OnPaymentReceived. But we could use the root parameter to overwrite this our own custom path. For example, if you had a function called getCustomer that took a customer ID, the default root would require you to pass the ID as a query string parameter. But you can also set up a root template that looks like this if you prefer with the customer ID as part of the URL path. If we jump back now to the code, we can see that this function also takes an ILogger, which allows us to write to the logs. Let's update the function body to say that we've received a payment. And instead of expecting a name in the query string, for now we're simply going to deserialize the body into a dynamic object and then respond with a Thank you for your purchase message. Now one really great thing about this Visual Studio integration is that we can run our function app directly from inside Visual Studio

Code. If I go to the Debug menu and select Start Debugging, it will compile a code and then start a local instance of the Azure Functions runtime to run our functions locally. By the way, if you're not running in Visual Studio Code and you want to start your function app from the command line, you can do that with the FuncHostStart command, which is part of the Azure Functions core tools. And you can see that once our code is complied, the local Azure Functions runtime starts up, and it helpfully tells us all the functions that it's found and what the URL is. Our PaymentReceived function is available at localhost port 7071 on the api/OnPaymentReceived root, which is, of course, the default root. So let's see if we can trigger this function. Now we could use something like Postman or cURL to do this, but I'm using PowerShell. So I'm going to use PowerShell's invoke web request command or iwr for short. We're going to post to that URL and specify that we're sending JSON. But we're just going to pass an empty JSON object in the body for now. And if you're wondering why we don't also need to provide the function's secret code, that's because when we're running locally, the code's aren't needed. But if we were running this function in Azure, then we would need it. And we can see that our function responds with the Thank you for your purchase message. If we jump back into Visual Studio Code, we can see that our Received a payment log message, which was emitted from our code, is visible alongside some other diagnostic output from the Azure Functions runtime itself. I'll stop the local Azure Functions runtime with Ctrl+C, and now let's take a quick look at what's in the bin folder. We can see that when we compiled our project, it copied a couple of files across, the host.json and the local.settings .json file. But there's also a folder OnPaymentReceived, which is named after a function. If we take a look inside, we can see there's a function.json file, and this has been autogenerated for us based on the attributes in our C# code. If we take a look at its contents, we can see it's similar to the function.json file that we looked at in the last module. It's got details of our httpTrigger, including the supported HTTP methods and the authorization level. But it also includes details of where the Azure Functions runtime can find the compiled code for our function. It's in Pluralsightfuncs.dll, and the method containing our function code is OnPaymentReceived.Run. Let's just make one quick enhancement to this function before we move on to look at output bindings. The real web-hook that we get from our payment provider contains details of the order that's been placed. And I've created a strongly typed order class, which will deserialize the data from the payment provider into. So let me paste that in here. And you can see we've got an OrderId, the ProductId of the item ordered, the customer email address, and the price paid. And now let's update the code to make use of this. The DeserializeObject method will now deserialize into a strongly typed Order object, and let's add another log statement to capture some details of the incoming order. Now we've made those changes, I'll press F5 to build and run this function app locally and wait for it to start up. And now let's return to PowerShell and send an example order. We'll use exactly the same invoke web request command as before, except that now in the body, we're going to send some dummy order data in JSON format. And when we run this, we get a 200 OK with Thank you for your purchase back again. And if we head back to Visual Studio Code, we can see that the log message has successfully picked out the details of the order that's been submitted. So that's our HTTP-triggered function created. Next, we're going to see how we can update it to post a message to a queue.

Demo: Queue Output Binding

The Azure function we just created is able to receive an incoming payment provider web-hook, but we also want it to post a message to a queue in order to trigger our license file generation code. Generally speaking, it's a good idea when you're receiving a web-hook to avoid taking actions that might take a while to complete as the sender of the web-hook might end up thinking that the call has failed and sending it again. So that's why we're going to post a message onto a queue, which will allow us to generate the license file asynchronously. So let's see how we can do that. As we said earlier, the docs.microsoft .com site is a great place to come to find details on all of the many bindings that are supported in Azure Functions. And so here I'm looking at the Azure Functions documentation in the Bindings section, and we're looking specifically at queue storage bindings. That is because we're going to send our message to Azure Storage queues. And the reason I've chosen that instead of say Azure Service Bus is simply that we've already got a storage account, which was created with our Azure function app and so it's really convenient just to use that. Now this is a very long and detailed article and explains how we'd use queue triggers and bindings from a whole host of languages. And they're actually many different ways of defining the binding. But I'm just going to point out two key bits of information. First, here in the Packages section for Functions 2.0, it tells us which NuGet package contains the binding attribute definitions for queue storage bindings. And it's Microsoft.Azure .WebJobs .Extensions .Storage. And we'll be adding that as a project reference shortly. Also, if we navigate all the way down to the Output section, we'll see a series of examples showing us how an Azure function can output a message to a queue. And the most important detail to notice is that it uses the Queue attribute. And the Queue attribute can be applied against a function parameter or to the return value of the function as this example is showing. In this example, the string that the function returns will be sent as a message to a queue called myqueue-items. But I don't want to bind to the return value of my function because I'm using a return value to send the Thank you for your purchase response to the webhook. And if we navigate to this Output-usage section, we can see all the other types that we can bind to. We can bind to an out parameter of our function, but that also isn't going to work because our function is asynchronous, and that means the C# compiler won't allow us to use output parameters. But there's another option here called IAsyncCollector of T. And this slightly oddly named type turns out to be really useful for output bindings in asynchronous functions, and that's what we're going to be using. So let's get to the code. First, we're going to add the NuGet reference to allow us to use storage bindings. I can do that in my csproj file by adding a new package reference to Microsoft.Azure .WebJobs .Extensions .Storage, and it's currently at version 3.0 .3. Visual Studio Code helpfully suggests that we do a NuGet restore to download this new dependency, so I'll do that. And obviously if you're working with Visual Studio, then you can use the NuGet Project References window to add this automatically for you. And now it's time to add the queue binding to our function, and we're going to use the Queue attribute to do that. The Queue attribute has one required parameter, which is the name of the queue to write to. Here I'm saying that we want to write to a queue called orders. And the documentation we looked at earlier told us that there were many different parameter types we could bind to here. But the one I've selected is an IAsyncCollector of type Order, and this will allow me to send messages to this queue from inside my function by calling the AddAsync method on the IAsyncCollector. Let's do that. And after we've deserialized the incoming order, we'll send it to the queue with await.orderQueue .AddAsync, passing in the order. And when this method is called, the queue binding will handle all the work of connecting to the

queue for us. And it will even create the queue on demand for us if it doesn't already exist. It will also serialize the Order object to JSON before placing it on the queue. So all the usual boilerplate code that you'd need to write to connect to a queue, ensure it exists, serialize your message, and send it has been abstracted away into this very simple syntax, which is one of the key benefits of using Azure Functions bindings. Now you might be wondering how Azure Functions knows which storage account to use. And the Queue attribute has an optional parameter called Connection, which lets us identify which storage account to use. The value of the Connection property isn't a connection string. You wouldn't want to put connection strings in your source code directly anyway. Instead, it's the name of the application setting that contains the storage account connection string. And this means that when you're running in Azure, it will use the app setting that we saw when we looked in the portal to resolve this connection string. But when you're running locally, it will find the one stored in your local.settings .json file. If we delete the connection property, it will look for a connection string called AzureWebJobsStorage, which points at the default storage account for our function app. If I take a look in our local.settings .json file, which contains the settings that will be used when we're running locally, we can see here that there's already an AzureWebJobsStorage connection string that got generated for us. And it's value, UseDevelopmentStorage=true, is a special connection string that refers to the Azure Storage Emulator. The Azure Storage Emulator is a free emulator that's brilliant for testing your Azure Functions locally. If you've installed the Azure workload in Visual Studio, then you've already got the storage emulator installed. However, it isn't cross-platform. And so that means if you're developing Azure Functions apps on a Mac or Linux, then you'll need to use a real Azure storage account. And we'll do that anyway here just for demonstration purposes to prove that Azure Functions runny locally can connect to a storage account running in the cloud. I'll jump over to the Azure portal and navigate into my storage account. And under Access keys, I can get hold of the connection string for this storage account. So I'll copy that to the clipboard and come back here to my local.settings .json file and paste it in. And by the way, the default gitignore file that gets created for you is set up to ignore this local.settings .json file. And so this file doesn't typically get stored in source control. And that's good because it prevents you from checking secrets like this into source control. But it does mean that you need to fill in the values again if you clone the repository to another machine. Okay, we're finally ready to test this function. I'll press F5 to compile our function app and launch the Azure Functions runtime. And once we can see that it's fully started up, I'll jump across to a PowerShell prompt. And once again, I'll use invoke web request to post a new order to our function. And of course if you are Mac or Linux, you could just use cURL instead. And I've just changed the order number and product ID for this example. And we can see that the order has been accepted. So I'm going to jump back into the Azure portal and take a look at that queue in the Storage Explorer. If I expand the QUEUES node, we can see that here's an orders queue that's been automatically created for us. And if I look inside, we can see that there's one message in the queue that's been serialized into JSON, and its properties match the order that we just submitted. Great, we've successfully created the first function in our pipeline. Let's recap what we've learned in this module.

Module Summary

In this module, we started off by learning about the supported languages in Azure Functions, which include C#, F#, JavaScript, and Java with more on the way. And we also

learned that there are two ways of writing C# Azure Functions, the C# script file way where you create CSX files that we saw in the last module and the precompiled way or the C# library way where you compile the function code into a DLL, and the function.json files get automatically created for you based on the attributes you decorated your C# code with. And that's the approach we used in this module. And unless you're just creating short-lived throwaway experimental functions, I recommend that you always use the precompiled approach for C# functions. We also used the Azure Functions core tools, which is an open source cross- platform command line tooling experience that allows you to easily create new function apps in a variety of languages. And it also simplifies creating new functions by providing you with a choice of templates. And it's a great option if you're not planning to use Visual Studio for developing your function apps. And we also saw how Visual Studio Code has got some really great Azure Functions tooling support, and so that's a great option again if you're not using Visual Studio. We discussed the broad range of triggers and bindings available in Azure Functions, and the function we created made use of an HTTP trigger and a Queue output binding. To indicate in C# that a function has an HTTP trigger, we apply the HttpTrigger attribute to a function parameter of type HttpRequest. And we can specify the authorization level, they're requiring a function-level code in this example, and the supported HTTP methods. Here's we're supporting post. And we can specify the route. And null means to use the default rate. So this function's rate will be /api/OnPaymentReceived. To use the queue output binding in C#, we need to reference the Microsoft.Azure .WebJobs .Extensions .Storage NuGet packet and then apply the queue attribute to a function parameter. And there are several different parameter types that you can bind to, but we chose to use the IAsyncCollector of T, which works well in asynchronous functions. We called the AddAsync method on it to post a message to a queue, and it automatically serialized the custom order DTO that we gave it into JSON for us. The Queue attribute also allows us to specify the queue name, which is orders in this example. And you can also configure the name of the connection string to the storage account, although we're just allowing it to use the default storage account in this example. And if the orders queue doesn't exist in the storage account, this output binding will create it automatically for us. Now I realize we've not got very far through our demo scenario yet. We've only created the first function, which receives the payment web-hook and writes a message to a queue. In the next module, we're going to continue to build our function pipeline by receiving the message we posted to a queue and creating a license file in blob storage, as well as sending an email. So join me in the next module to see some more triggers and bindings in action.

Building a Function Pipeline

Module Introduction

Hi, Mark Heath here. And in this module, we're going to continue our look at Azure Functions triggers and bindings by building more of the function pipeline that we started to create in the last module. If you remember, last module we built a simple web-hook that accepted order details sent to us by a payment provider, and we wrote a JSON message onto a queue whenever that function was called. In this module, we're going to create the next two steps in the function pipeline, first of all, handling that key message by creating a license file and storing it in blob storage and then responding to the creation of that license file by sending

an email to the customer with the license file attached. We'll also be upgrading our function pipeline to use Azure Table Storage as both an additional output of our first function and as an extra input to our last function. In this module, we'll be using the Queue trigger, the Blob output binding and Blob trigger, the SendGrid output binding for sending email, and the Table Storage input and output bindings. And we're going to be using Visual Studio in this module, so you can get to see the tooling support that it has for Azure Functions. We're also going to be learning about some more advanced binding techniques by using the IBinder interface. And the function pipeline that we're building is designed to demonstrate how you can chain together several small, single-purpose functions, which is a much more reliable and maintainable way than having one giant long-running function that tries to do everything. So let's get started by building the function that listens on a queue.

Demo: Queue Trigger, Blob Output

In this demo, we're going to listen on a queue for a message containing details of incoming orders and create license files, which we're going to place into a blob storage container. Let's see how we can do that. We're going to be using Visual Studio for the demos in this module. If you've got Visual Studio 2017 or newer with the Azure workload selected, then you're already good to go. And you can check if you've got it set up correctly simply by going to File, New project, selecting the Cloud category, and seeing if the option to create an Azure Functions app is offered. And here you can see that it's available. But we're actually going to carry on working on the project that we created in the last module with the Azure Functions core tools. So I'm going to cancel out of this dialog, and we're going to use the File, Open dialog to open an existing project. If I navigate to that project that we created in our last module, we can select the csproj file. And when it loads, we can see our function app complete with the OnPaymentReceived function that we already created. If you remember, this had a queue output binding, which allowed us to send a message to a queue. So now we're going to create a queue- triggered function to receive that message. I'll navigate to Project, Add New Item, and select Azure Function. And we'll call this function GenerateLicenseFile. And I'm going to select a queue trigger template. The connection string setting we're going to use is AzureWebJobsStorage, which is the default value. And I'm going to set the queue name to orders. So let's create this function now. I'm going to reformat the code a little bit just so we can see it. And we can see that our Queue attribute is bound to a string parameter. Now our queue message contains a JSON serialized order object. So let's update this parameter to be of type Order, and we'll also rename it to order. Another thing I'm going to do is jump back into my local.settings .json file, and we're going to update the value of the AzureWebJobsStorage connection string. Remember, this setting is only used when we're running locally. When we run in Azure, we're going to pick up the value of this connection string from an app setting. And in the last module, you might remember that we set this to a real connection string for an Azure storage account. So we could test this without needing to use the Azure Storage Emulator. But since we're running in Visual Studio, which means we have got the Azure Storage Emulator available, I'm going to update this to use the connection string for the emulator, which is UseDevelopmentStorage=true. Let's head back to our GenerateLicenseFile function, and we'll add a blob output binding. I'm going to paste in the definition for the binding here, and the attribute that defines the binding is a blob attribute. We're binding to a parameter of type TextWriter, and there are actually multiple types that we could bind to here depending on whether it's an input or an output

binding. Since this is an output binding, which means we're going to be creating a blob, and we're writing text to that blob, then a TextWriter is a good choice here. But if we take a quick look at the documentation, you can see there are many other choices for a blob output, including Stream, which you might use for writing binary data into your blob, and also CloudBlockBlob, which gives you the flexibility to set the blob's metadata if you need to. Now we're not using a blob input binding in this example, but you can see that a similar variety of choices is available. For example, if you choose string for a blob input binding, then the entire contents of the blob would be read into that string. But you could also get it as a byte array if, for example, it contained binary data. Back here in our code, I also want to point out that the Blob attribute takes a parameter that specifies the blob name. In this case, I want it to go into a blob storage container called orders, which is going to be created on demand for us. And for the filename, we know we're generating a license file. But what name should we give it? For now, I'm just showing how we can use a special syntax that lets me make use of a random GUID by using this rand-guid in curly brackets notation. And then after that, I'm putting the extension I want. And we'll see how to improve on this naming code a bit later on. Next, I'm going to paste in some code that actually writes to our blob. We're creating a license file that we're going to email to the customer, and so it's just going to be a text file containing details of their order, their email address, and so on. Often in a license file, there's a special code that we can use to prove that this is a genuine customer. So I've just created a very simplistic hash of their email address as a secret value, which is fine for demo purposes. But of course a real-world application would maybe use a more secure license code generation algorithm. And I've noticed that I didn't give the blob the extension I intended to, so let me fix that up to .lic. And while I'm hear, licenses would actually be a better name for the container. So let me change that as well. Great, we're ready to test this now. So I'm going to build and run my function app. It will take a moment to start up. But once it's ready, we're going to be able to submit another order using PowerShell just like we did in the last module, and this is going to test our pipeline end to end. We're going to be calling the OnPaymentReceived HTTP-triggered function, which posts the message to the queue. And then our new GenerateLicenseFile queue- triggered method should pick that up and write to blob storage. And it looks like that worked. So if we take a quick look at the functions runtime output, we can see that not only did our OnPaymentReceived function run successfully, but our GenerateLicenseFile method has also run. I'm going to use the free Microsoft Azure Storage Explorer tool to look inside the storage account and see what's in the licenses container. This tool is great for exploring Azure Storage accounts, but it also allows me to look inside the local storage emulator account if I navigate to Local and Attached, Storage Accounts, Emulator, Blob Containers, then we can see our licenses container. And notice that this licenses container automatically got created. And in here, we have a blob with a .lic extension and a GUID filename. I'm going to download that license file. And once it's downloaded, if I load it up in a text editor, we can see here that sure enough our license file has been successfully created and contains the details that we expected.

Queue Trigger, Blob Output Recap

Let's quickly recap the bindings we used in the last demo. We used a queue trigger so we could respond to a new order message in a queue and a blob output binding so we could write a license file to blob storage. To use the queue trigger in a function, we added the QueueTrigger attribute and specified the name of the queue we wanted to monitor for

messages. We were also able to provide the name of an application setting that contained the connection string to the storage account that this queue was in. And for local testing, we chose to use the Azure Storage Emulator so the connection string's value was UseDevelopmentStorage=true. And in our example, the QueueTrigger binding was applied to a parameter of type Order, which means that when the message is read off the queue, it's assumed to be a JSON serialized order. And so the binding automatically attempts to deserialize the message into an Order object. Then we set up a blob output binding, which we used the Blob attribute to define. A Blob attribute requires us to specify the name of the blob that we're writing to. And we made use of a special syntax that lets us use a random GUID as the filename, and we put it in a container called licenses. And in a bit, I'll be showing you other ways that we can control the filename. And just like we could with the QueueTrigger, we can also supply the name of an application setting that contains the blob storage connection string. We bound it to a TextWriter, which is a great choice if we want to write a text file. And anything we write to that TextWriter gets written to the blob in blob storage. But you don't have to use a TextWriter. As we saw in the documentation, there are several supported types that the binder will connect to making it easy to create JSON, plaintext, or binary blobs depending on your requirements. So we've completed the first two steps of our Azure Functions order processing pipeline. We've got the payment web-hook posting to the queue and our new function receiving from that queue and creating a license file in blob storage. So next up, when that license file is created, we want it to be automatically emailed to the customer. So let's do that next.

Demo: Blob Trigger, SendGrid Output

In this demo, we'll be creating the third step in our ordering processing pipeline. When the previous function creates a license file in blob storage, we'll trigger a new Azure function that emails the license file to the user. Let's head into Visual Studio and create our new function. As before, we're going to create a new function by selecting Project, Add New Item, Azure Function. And we'll name this function EmailLicenseFile. Next, let's select the template. This function will be triggered by a blob being created in the licenses container. So we'll choose the Blob trigger. And for the connection string setting, we'll again use the default value of AzureWebJobsStorage. And we'll set the path to simply licenses, which is the name of the container that we're watching for new blobs. So let's accept this and take a look at the function that gets created for us. Here we can see that it has the BlobTrigger attribute, and the path has been expanded automatically for us to licenses/name. And this name in curly bracket syntax simply allows us to easily get the actual name of the blob as another function parameter. Although in our case, it doesn't really matter as we don't need to use the blob name for anything here. It's bound to a Stream parameter. But in our case, it would be more useful as a string. So I'll make that change, and I'll call it licenseFileContents. Let's quickly jump into the documentation, and we're on the blob storage bindings page here. And it tells us that we can bind to a byte array, which would be good if our blob contains binary data, or to a string if the blob contains a text file, which is perfect for our needs. But you can also see we can bind to a string, which is great for large files that we don't want to read entirely into memory. And we can use ICloudBlob and its derived types, which would be great if our incoming blobs contain metadata that we needed to access. Great, what do we need to do inside this function? Well, we actually want to email this license file to the customer. So we need to know who the customer is. For now, I'm going to extract the customer email address

out of the license file with a simple regex. But we'll improve on this later on in the module. Now how we can actually send an email from an Azure function? To do that, we're going to use the SendGrid binding. SendGrid is a third party service that supports sending emails, and it has a binding extension for Azure Functions. And all the Azure Functions triggers and bindings with the exception of HTTP triggers and timer triggers are defined in NuGet packages. So let's go to Project and Manage NuGet Packages. And we'll search the NuGet repository for NuGet packages whose name starts with Microsoft.Azure .WebJobs .Extensions. And, as you can see, there's quite a lot. We're already using the storage extension. But you can see there are multiple services that we can integrate with. In our case, we're looking for the SendGrid extension. So I'm going to select that and install it. And so once we've accepted the license and it's finished installed, we can head back to our function code. And I'm going to add a new parameter to our EmailLicenseFile function. I'll just paste it in, and we need to add a few using statements, which Visual Studio will helpfully automatically fill in for me. We're using the SendGrid attribute, and we're binding to an object of type SendGridMessage. I've made this an out parameter. And so in the body of the function, we're going to need to create a new SendGrid message. And we're able to use an out parameter because this function is not asynchronous. If it was, we'd have to specify this parameter as an IAsyncCollector of SendGridMessage, similar to the technique that we used to post to a queue in a function earlier on in this course. You'll notice as well that we need to provide an API key. But, of course, this isn't the actual API key itself. Instead, it's the name of an application setting that contains the SendGridApiKey. And that allows us to keep the secret key out of source control. Of course, when we're developing locally, I will need to put my SendGripApiKey into my local.settings .json file, but that's a file that you typically don't check into source control. In the body of our function, all we need to do is create a new SendGridMessage object and configure it. And I've pasted some code in here that does exactly that. And it sets up who the email is from and to, and the recipient of this email is going to use the customer email address that we passed from the license file contents. And we're adding an attachment that contains the license file itself. We do need to base64-encode it first, so that's what these two lines are doing. Finally, we're setting the subject and content of the email, and that's all there is to it. By the way, just in case you're wondering what these little parameter labels are, this is actually a nice feature that comes from a Visual Studio extension called Resharper. And I think it's actually quite helpful for understanding code because we can see what each of the parameters of a method call actually are. So I've left it on here. You'll notice here that I've not hardcoded the EmailSender. Instead, I've asked for that to come from an environment variable. And that's a really easy way of accessing application settings in your code. All of your application settings will appear as environment variables. So it makes them very easy to access. Let's jump into our local.settings .json file so that we can configure the settings that are needed for testing locally. I need to add an EmailSender and a SendGridApiKey. And if you're trying to follow along with these demos, then you do need to create a SendGrid account to get your own key as I'm not going to be showing you mine. But there is a free trial option. So you shouldn't have to pay to try this out. And we'll be learning later on in this course how you can automate setting values like this on your function apps running in Azure without having to go into the portal and setting them manually in the applications settings view. Great, I'm going to put in my real SendGridApiKey now. And once I've done that, we're ready to test this. So I'm going to build my code and start the Azure Functions runtime app. Once again, we're going to test our whole function pipeline end to end. I'm going to call my OnPaymentReceived function in the same way that we've been doing throughout this course from PowerShell. But this time, I do need to

make sure that I provide a real email address. So I'm just going to send this to myself. When we send this payment web-hook, that's going to post a message to a queue, which will create a license file in blob storage, which triggers our new function and causes an email to be sent. And if we take a quick look at the log output from the local Azure Functions runtime, we'll see that our EmailLicenseFile function has been called. And soon enough in my email inbox, I'll get an email through that contains an attachment. And if I look at that, I can see the contents of the license file generated in blob storage.

Blob Trigger, SendGrid Output Recap

Let's quickly recap what we learned in the last demo where we used a blob trigger to trigger our function whenever a new license file was created in blob storage and a SendGrid output binding to send an email. To create a blob-triggered function, we used the BlobTrigger attribute and specified the name of the blobs we wanted to match. In our case, it was blobs in the licenses container, which had the .lic extension. And we used this special syntax to take the filename of the blob and to put it into another function parameter that had the name orderId. The blob attribute was applied to a string, meaning that the contents of that blob would be read out and written into that string. And that worked great for us as our blobs contained text and weren't particularly large. But we saw that you could bind to several other parameters besides string. You can bind to a custom object if your blob contains a JSON serialized version of that object just like we saw with the queue trigger. You can bind to a string, which is really great for large binary blobs that you don't want to load entirely into memory. And you can bind to a CloudBlockBlob, which is useful if you also want to access the blob's metadata, as well as its contents. And there are other options available too. We also saw how to use the SendGrid output binding for which we used the SendGrid attribute. We needed to provide an ApiKey, which was stored in our function app settings, and then we simply needed to assign a SendGridMessage object to our message parameter in order for it to get sent. And SendGrid is just one of many examples of really easy-to-use integrations with third party services in Azure Functions. For example, if we wanted to send a text message, we could've done that just as easily by using the Twilio binding. Now you might be wondering, what if I don't want to use SendGrid for my emails? Maybe you're using a rival email-sending service or you have your own email server you want to connect to directly. The important thing to remember is that you don't have to use output bindings. There's nothing stopping you just writing the code to directly call whatever external services you need to communicate with from within your Azure function. Output and input bindings are just provided for convenience. They make it really quick and easy to connect to other services. But if the service you need isn't supported, or if the existing bindings are too limited for what you need to do, then feel completely free to ignore the bindings and just write the code yourself. Great, we've successfully created the three functions we set out to build at the start of this module. But we're not quite done yet. In the next demo, we're going to see how we can also store details of our incoming order in Azure Table Storage.

Demo: Table Storage Output

The Azure Functions pipeline that we've created so far takes us all the way from receiving a payment web-hook to emailing the user. But we've still not stored the fact that we made a

sale in a database. And we'd probably want to do that in a real-world application. So how can we achieve that with Azure Functions? Well, there are some database bindings that we can use, including Azure Cosmos DB if you want a document database or Azure Table Storage if you just want something very lightweight and cheap. Unfortunately, there is no Azure SQL database binding at the moment. Of course, it's still possible just to use Entity Framework from inside your own functions, but you would need to bring your own model classes along. So it can be quicker and easier to use one of the supported bindings if that fits with your requirements. For our next demo, we're going to add an additional output binding to our payment web-hook. And it's completely fine to have multiple output bindings in a function. Although if the number of output bindings gets too high, that's probably a sign that your functions are breaking the single- responsibility principle. But we're going to add a table storage output binding to our payment web-hook function. Every sale we make will result in a new row in Azure Table Storage, as well as sending a message to a queue to trigger license file generation. And both of these options are going to be very quick to execute. So I'm not particularly worried about the extra output slowing down our web-hook. And the reason I've chosen table storage over a more powerful, fully featured database like Cosmos DB is simply because we've already got a table storage account. And the local storage emulator also supports emulating table storage. And so this is just going to make it really easy for us to test, and you'll be able to follow along easily if you're trying out these demos yourself. First, in our OnPaymentReceived function, let's add a new output binding. It will be almost identical to the queue output binding. We'll use the Table attribute, and the Table name is also going to be orders just like the queue name was. Again, we'll use the IAsyncCollector so that when we call the AddAsync method, we'll add a row to table storage. And I've even reused the same order type here. Now that is a little bit of a shortcut. In a real-world application, you might have several DTO classes, one that represents the incoming web-hook payload, one that holds the data we want to be serialized onto the queue, and one that represents the data we want in our database. But since in our example all three hold basically the same information, I'm just reusing the same class to cut down on code. However, I will need to add a couple of properties to the Order object to allow it to be stored in table storage. Table storage actually uses a composite key. Every row has got a PartitionKey and a RowKey, and a row in table storage can be uniquely identified by the combination of these keys. So in our code before we add the Order object to table storage, I need to set up these properties. I'm hardcoding the partition key to orders just to keep this example really simple. But in a real-world system, if you had potentially millions of orders, then you would want to spread them across more than one partition for efficiency. And for the RowKey, I'm just using the OrderId. And now just like we did with the queue, we can add the order to table storage with AddAsync on the IAsyncCollector. Now this function, which has an HTTP trigger, has two output bindings, queue output binding and a table storage output binding, and that's absolutely fine. Every Azure function must have exactly one trigger, but it can have zero or more bindings. Usually, you wouldn't have more than one output binding. But I did want to show that it's possible to have more than one. So let's try this out now. I'm going to start up the local functions runtime in the usual way, and we'll wait for it to start up. And, again, I'm going to use the same technique as before to test the function. I'll use PowerShell to post to the HTTP- triggered function. And this is the function that is now going to both send a message to the queue and add a row to our table storage order database. And it looks like everything's worked fine. So if we take a quick look at the log output from the Azure Functions runtime, we can see that everything looks good here too. Our functions pipeline has been successfully executed. Let's prove this worked by loading up the Azure Storage Explorer again. And sure enough, if we

navigate into the node for the local storage emulator, we can see that there's now an orders table, which got created automatically for us. And you can see here that there's a row with a Partition and RowKey as we expected, as well as columns for all the other properties of my Order object, which also got created automatically. So this is a super easy way for you to save data from an Azure function into table storage with the use of an output binding.

Table Storage Output Recap

Let's quickly recap how we added an additional output binding to our function to add a row to table storage. It was very straightforward to add the additional output binding. We just added a function parameter with the Table attribute, and we specified the name of the table we wanted to use. We bound to an IAsyncCollector of type Order, which allowed us to add a new row to table storage simply by calling AddAsync on the IAsyncCollector. And the IAsyncCollector of T interface allowed us to easily add a row in table storage, and it meant that we could pass in a strongly typed object, and the binding would automatically create a column in the table storage row for each property. But it did mean that we needed to add two special properties to our Order object, a RowKey and a PartitionKey because together they form the unique key for each row in table storage. So this is what our pipeline looks like now. We've upgraded it to write a row to table storage. But before we wrap up this module, I want to make a few final improvements, including using our first input binding, which is going to let us access the data that we've just stored in table storage.

Demo: IBinder and Table Storage Input

We've successfully achieved what we set out to do in this module, which was to create a pipeline that receives a payment web-hook, generates a license file, emails it to the customer, and stores it in a database, in our case table storage. But there are a few things I don't like about our code, and maybe you spotted some of these issues as we worked through the examples. And the first issue is when we create the license file, the filename we gave it consisted of a random GUID. Wouldn't it be nicer if the license file was named by something like the order number so it was order number .lic. And in this demo, I'm going to show how we can use the IBinder interface to take control of the filename of the license file we create in blob storage. The second issue is when we email out the license file, we're working out who to send it to by pausing the contents of the license file from blob storage. And that's a bit ugly, and it assumes that the license file actually contains the email address. Wouldn't it be better if we could look up the email address of the order from our table storage database. And we can do that by using our first input binding so that our license emailing function is not only triggered by blob storage, but also uses our orders table storage table as an input that we can query against to find the customer email address. And the third issue we're going to address is that it might be a bit annoying that we always have to send an email every time we test our function out locally. So I'll show you how by switching to use the ICollector of T interface, we can make email sending optional. Great, let's tackle our first issue. In the GenerateLicenseFile function, which has got a queue input and a blob output, I want to take full control over the output blob filename. And we've currently got a TextWriter parameter called outputBlob, and it currently uses a path of licenses/random-guid.lic. How can we ask it to use the order number instead? Well, what we can do is defer the

binding until we know what filename we want to use. I'll get rid of the TextWriter outputBlob parameter and replace it with a new parameter of type IBinder, which I'll just name Binder. And now I'll paste in a small code snippet to show how we can use this. I am also going to need to make this function asynchronous because we've used the await keyword. And that's easy enough to do. I replace void with async Task and make sure I add in the appropriate namespace. What we're saying here is the outputBlob is the result of binding to a TextWriter and passing in a BlobAttribute that has the exact filename we want including the container name. So it's going to be the licenses container, and the filename will be the order number, which we know from the incoming queue message with a .lic extension. We're also specifying the name of the app setting that contains the storage account connection string. In our case, it's just the default, which is AzureWebJobsStorage. The great thing about the IBinder interface is that it can be used for any binding that has a parameter value that we don't know the value of until runtime. When we call BindAsync on the IBinder, we need to provide two bits of information. The first is the binding attribute that normally would've been part of our function signature. In this example, it's a BlobAttribute. But this technique would also work for any other binding attribute, such as a QueueAttribute or a SendGridAttribute. Because we're constructing the attribute in the function body, we've got complete freedom to calculate the attribute parameters on demand. So not only could we customize the blob path parameter like we're doing here, but we could dynamically switch between different storage account connections as well if we wanted to. The other piece of information we provide is the type we're binding to. In this case, we're binding our BlobAttribute to a TextWriter. But as we saw in the Azure Functions documentation, most binding attributes support binding to a variety of different types. So again, you'd have the flexibility to pick the most appropriate one on demand in your function code if you needed to. And this flexibility makes the IBinder interface a very powerful way to customize your Azure Functions bindings to work just the way you need. So now our license file uses the filename that we want. Next, let's update our EmailLicenseFile function to make sending emails optional. If you remember, we used an out SendGridMessage parameter for the SendGrid binding. But quite often, if I'm testing, I might not want to always send an email. So I'm going to change this to an ICollector of SendGridMessage and call it sender. And ICollector is basically the same as IAsyncCollector, just it's a non- asynchronous version. I'll need to create a new variable to hold the message object. And finally, we'll only send the email if the email address doesn't end with test.com. So that's two of our issues resolved. Let's also make the final change. I don't want to have to pass the licenseFileContents with a regular expression. Instead, I'd like to look up the order in the database and use that to find out what the email address is. And this is where a table storage input binding can help us. First of all, I'm going to make use of the fact that now we can get the order number from the license filename. Let's change the BlobTrigger attribute to say that we're interested in blobs in the licenses container that have a name in the form orderId.lic. And that means we also need to update the name parameter to be orderId and update our email attachment name to make use of the orderId. And we'll also update this log message to show the orderId. And now I'm going to paste in a new function parameter that's going to perform the lookup of the order in our table storage database. The binding type is a Table attribute binding, and we're saying that we want to look in the table called orders for a row with a PartitionKey who has a value of orders because remember that's what we hardcoded it to and a RowKey that matches the orderId we extracted from the blob name. And this binding is applied to an object of type Order. And that means that we can now get the email address directly from the order that we found in table storage. So I can delete this ugly regular expression and replace it simply with

order.email. And, of course, now we've got access to other useful information about this order, which we might want to use to construct a more personalized email message. Now you might be thinking, well, wasn't that rather convenient that the blob filename just happened to contain all the information needed to perform the table storage lookup. Maybe in a real-world application it would be a bit more complicated than that. Well, we could, of course, have used the IBinder here as well and bound the table attribute after we'd worked out what the Row and PartitionKey we needed to perform the lookup were. So there really is quite a lot of flexibility available to you if your Azure functions need to perform some kind of database lookup. Great, we've changed quite a few things here. Let's make sure that it still works. I'm going to start up the local functions runtime again and check that everything still builds correctly. And once the Azure Functions runtime is running, we're going to go back to PowerShell again to test our pipeline. This time I'm going to say that the buyer has an email address ending in test.com, so it won't actually keep spamming my personal email inbox anymore. And I'm going to be using an order number in this example of 555, which we'll need to look out for later. And everything seems to have worked. So let's check the log output. Here we can see a message from the EmailLicense function showing us the email address that it looked up in table storage and the Order Id that it matched from the new filename in blob storage. And if we jump over into Storage Explorer and look at the blob storage account for our storage emulator in the licenses container, here we can see that alongside the old licenses we created earlier that just had random GUID filenames, there's one called 555.lic, which is the order number we just submitted. And you can also see that there's a 444.lic that I created earlier while I was testing.

IBinder and Table Storage Input Recap

In our previous demo, we made three improvements to our function pipeline. First, we used the IBinder interface to give us greater control over the name of the license file blob we created in blob storage. And that means that in the function body once we know the filename that we want to use, we can bind to a TextWriter with a Blob attribute that specifies the desired filename and connection string. And this is a very powerful technique that allows you to defer making decisions about what the properties of your binding attribute are or even what variable type you want to bind to. With IBinder, those decisions no longer need to be made in the function declaration, but can be made in the function body when you've got more information available to you. Second, we used a table storage input binding in our blob-triggered function, and this allowed us to perform an automatic lookup of a row in table storage. We used the table binding for this, specifying the name of the table we wanted to access and the Row and PartitionKey of the row we wanted to lookup. In our case, the PartitionKey was hardcoded to orders, and the RowKey was taken from the name of the license file blob that triggered our function. And this is another scenario in which IBinder might be useful. We might not always have enough information available about which row we want to look up from within the function declaration. But if we use IBinder to bind to a TableAttribute, then we could write additional code in the function body to determine the RowKey and the PartitionKey that we want it to look up. And third, we also switched the SendGrid email binding from using an out parameter to an ICollector of SendGridMessage, and this made sending an email optional. If we didn't want to send an email, we simply didn't need to call the Add method on the ICollector. Let's take one final look at the pipeline we created in this module. A web-hook from our payment provider results in a

message in a queue and a row in table storage. The message causes a license file to be created in blob storage, and it now has a filename that's a bit more meaningful to us. And the final function is triggered by that blob being created, but takes an additional input, which is the order row in table storage, which it uses to work out who the license should be emailed to.

Module Summary

Let's take a step back and look at what we've accomplished in the last two modules. The order processing function pipeline we created uses three different trigger types, four output binding types, and an input binding. And we've been recapping each of these triggers and bindings as we go. So I'm not going to cover that again now. But I do want to point out that we've only scratched the surface of Azure Functions. Azure Functions offers many more trigger types, input bindings, and output bindings that we haven't had the time to look at yet, including Service Bus, Cosmos DB, and third party services like Twilio. What's more, for each of the bindings that we did use, I only showed you one or two of the possible parameter types that you can bind to. So I really encourage you to bookmark the bindings documentation page that we've been referring to in this module as it's an invaluable resource for working out what you can do with each binding type. And, of course, we've been using C# for the functions in our pipeline, but you may prefer to use another of Azure Functions supported languages, like JavaScript or F# or Java. And each of those languages have got their own ways of connecting the function parameters to the corresponding bindings. One great way of getting up to speed quickly with all of these languages and the options for how you bind to parameters is to take advantage of the sample templates. Just create a new function from a sample template using the language and binding type you want, and you'll find that there's a readymade code snippet that will help you get started. So you don't need to memorize all the options. Up next, we're going to look at how we can deploy our function app to Azure when we finish developing and testing it locally.

Deploying Azure Functions

Module Introduction

Hi, Mark Heath here. And in this module, we'll be learning about the different ways you can deploy your Azure functions. Now so far in this course, we've done most of our testing locally. We've used the Azure Functions core tools and Visual Studio to run our function apps directly on our development machine. But of course at some point, we will want to get our functions running in Azure. So in this module, I'll be introducing you to the various ways to create function apps in Azure and publish your code to them. We already saw earlier on in this course how to create an Azure function app in the portal. But I'm also going to show you two ways to automate that with the cross-platform Azure CLI command line tool and by creating an ARM template that defines a function app. And then we'll also see some ways to publish your functions into that function app, first of all by using Visual Studio, but then by using the Azure Functions core tools. Finally, I'll show you how you can host your Azure function app in a Docker container, which opens the door to you deploying your function app in other

environments, such as to other cloud providers or to an on-premises data center. So let's get started.

Ways to Deploy Azure Functions

Azure Functions gives us a lot of choice over how we can deploy our functions. To deploy an Azure function app, there are two main steps. First, you need to create the infrastructure to run your functions on. This includes the App Service Plan that will host your function app. And, as we discussed earlier in this course, you've got a choice here of the consumption plan with its serverless per- second billing model. Or you can use a regular App Service Plan. But we also need to create additional infrastructure, such as a storage account for blogs, queues, and table storage and an Application Insights instance for monitoring and diagnostics. And then, of course, we need to create a function app itself. And we saw earlier on in this course how we could create all this infrastructure using the Azure portal. And you can also create a new function app and all its associated infrastructure from within Visual Studio. But I'm sure you know that when you have a production application, it's not a good idea to use manual processes to provision your cloud infrastructure. Instead, it's better to use an Infrastructure as Code approach that allows you to reliably and repeatedly create the Azure resources needed for your function app. And there's a few options available to us here. In this module, I'm going to be introducing you to the Azure CLI, which is an open source, cross-platform command line interface for automating the creation and management of Azure resources. And I'll also show you how to create an ARM template, which is a declarative JSON document that defines all the resources in an Azure resource group and can be provisioned using the Azure CLI. Or you can use PowerShell or even the Azure portal. Once you've got the necessary infrastructure in place in Azure, then you need to actually publish your function app code and perform configuration tasks, such as setting up application settings. Again, this can be done manually through the Azure portal, which we saw how to do earlier on in this course. And the Visual Studio tooling also makes it really easy to publish and configure your function app. But again, for a production application, you'd want to take steps to automate the publishing of your function app code. And here, there's a lot of options because Azure App Service, which Azure Functions is built on top of, has historically offered many different ways to deploy your web apps. And these are all available for function apps as well. And the available options include zipping up your application and using an API called Kudu to push that to your function app. You can also put the zip in Azure blob storage and then set up an application setting that tells your function app where it can find the code, and that option is called run from package. And you can even get your function app to automatically deploy whenever new code is pushed to a Git repository. Now the technique I'm going to be showing in this module is using the Azure Functions core tools to publish your application with the func azure functionapp publish command, which will use the run from package technique behind the scenes by default. But you can also use the Azure CLI with the az functionapp deployment source config-zip command to publish a zip file containing the function code. And, of course, if you want your CICD process to automatically publish new versions of your application, then you can configure your build server to use one of these techniques. For example, Azure DevOps pipelines has got some built- in capabilities that make it really easy to publish your function apps. And many build scripting utilities, such as Cake, which is a build automation system for C#, have also got add-ins that can easily publish function apps. So there really are loads of ways to manually provision infrastructure and publish application code, as well as

many options for automating those processes. And so we're going to be seeing several of these options in action as we go through the rest of this module.

Demo: Creating an Azure Function App with the Azure CLI

In this demo, we're going to create an Azure Functions app using the Azure CLI. We'll use it to create a resource group, a function app, a storage account, and an Application Insights instance. The Azure CLI is an open source, cross-platform command line tool that allows you to create and manage Azure resources really easily. It's got a very similar set of capabilities to the Azure PowerShell cmdlets. But I really like the simplicity of the Azure CLI even when I'm on Windows. So that's what I'm going to be using. And it also means that if you're on another platform, you'll be able to use exactly the same commands. To install it, you can find instructions on this page, on the docs.microscoft .com site. It's pretty easy to follow no matter what your operating system. So if you want to follow along, why not pause the video now and make sure it's installed on your system. Once you've installed it, you can check it's working by simply typing az. And the way it's structured is that it supports a whole bunch of subcommands. And you can see some of them listed here. So to manage function apps for example, we'd use az functionapp. And if you add the -h switch to ask for help, it shows you all of the subcommands for a function app. And we're going to be using az functionapp create later. So if we wanted to learn about all the options for creating function apps, we'd use the az functionapp create -h command. Now there are a couple of preliminary steps that you'll need to perform before using the Azure CLI. And I go into these in a lot more detail in my Azure CLI: Getting Started course here, on Pluralsight. So do check that out if you want a more in- depth guide to the Azure CLI. But the key thing for our purposes is that you need to log in, which you do with az login. And when you do this, it will open a browser window in order for you to sign in to Azure in the usual way. And once you've signed in, your command line prompt will be authorized. Secondly, if you've got multiple Azure subscriptions, then you should use the az account set command with -s and then your subscription name to ensure that you're using the correct subscription for creating all your resources. And I've already performed both of these steps to log in and select my subscription. So let's move on to create our function app. First of all, I'm going to create an Azure resource group to hold all of the resources associated with this function app. And this is a good idea to keep things organized. And it also allows us to delete everything in one go when we're finished simply by deleting the resource group. And so I recommend grouping together Azure resources that are going to share a common lifetime into a single resource group. I'll declare a couple of variables that we're going to be using later. The resource group is going to be called Pluralsightfuncs, and the location I'm going to use is westeurope. And so now I can create my resource group with the az group create command, specifying the resource group name and the location. And once that's completed, we're going to create a storage account for our function app to use. I need a unique name for our storage account, so I'll just call it Pluralsightfuncs2019, and then I can create a storage account with az storage account create, specifying the name of the storage account, the location, the resourceGroup I'm going to put it in, and the sku, which is the pricing tier. And standard locally redundant storage is fine for us. I'm also going to show you how to create an Application Insights instance, which unfortunately at the time of recording is a little bit tricky to create with the Azure CLI. But we can use the az resource create command to create one. And to make that a bit simpler for me, I'm first going to save a small piece of JSON into a file. So I'm just going to echo an object

where the Application_Type property has a value of web into a file called props.json, and we'll use that JSON file in a moment. Now we'll choose a name for our Application Insights instance. I'll also call this Pluralsightfuncs2019. And we can now create it with az resource create, specifying the resourceGroup we're putting it into, the name we want to give it, the resource-type, which is what sort of resource we're going to create, and the properties of the resource, and that's where we're using this special @ props.json syntax to say that the properties of the App Insights instance are defined in that json file we just created. Okay, so now we've got an Application Insights instance, and hopefully that step will become easier in the future. But with that, we are ready to create our function app. And we'll use the name Pluralsightfuncs2019 for our function app as well. Now normally at this point, you would need to also create an App Service Plan, but the Azure CLI has got a helpful shortcut to avoid us needing to do that when we're using the consumption plan. So we can just jump straight to creating an Azure function app with az functionapp create, specifying the name of the function app, the resourceGroup to put it in, the name of the associated storage account, and the name of the associated Application Insights instance. And we're saying we want to use the consumption plan in the same location as our resourceGroup, which in my case is West Europe. And I'm telling it that this function app is going to be using the dotnet runtime. Now this will take a moment or two to complete. But once this is in place, we will have successfully created and configured an empty Azure function app all ready for us to publish our code to. Let me just show you one more thing that will be very useful if you're using the Azure CLI to automate your deployments and that's how we can set up application settings. We can do that with the az functionapp config appsettings set command, specifying the functionAppName and the resourceGroup and then providing the settings in the format of setting name equals setting value with a space separating each setting that we want to configure. So this command would be really useful if we wanted to set that SendGridApiKey app setting, for example. Now when I run this command, we see that not only have those two settings been added, but the App Insights key and connection string for the storage account had already been automatically set up for us by the previous command when we created our function app. So we've created an Azure function app and the associated infrastructure with the Azure CLI. But next, let me show you another way that we can automate the same thing.

Demo: Create an Azure Functions App with ARM

In this demo, we're going to learn how to use Azure Resource Manager, or ARM templates, to create an Azure function app. An ARM template is a JSON document that defines all the Azure resources that you want to deploy to a resource group. So we can use one to define the App Service Plan, function app, storage account, and Application Insights instance all in a single JSON document. Not only can you specify what resources to create, but you've got high-level control over their settings. So we can choose the App Service Plan location, set up function app settings, and choose the pricing tier for the storage account all within the template. ARM templates support parameters allowing you to make the template reusable. For example, you might parameterize the resource location or the function app name. And ARM templates are idempotent, meaning that if you deploy one for a second time, then it will have no effect, assuming that all the resources it created the first time are still present. And that means if you want to update your resource group, you just need to change your template to include your new settings and redeploy it. So let me show you an ARM template that deploys an Azure function app. And the first thing to acknowledge is that

this might well seem rather intimidating if you've never seen an ARM template before. We've got 130 lines of JSON here, and some of the syntax is quite complex. The good news is that you'd never need to write one of these from scratch. Instead, you'd typically base a template off of an existing sample. And there's a GitHub project called azure- quickstart-templates that contains loads of example ARM templates, including several Azure function samples. And the template I'm showing you is based on one from that repository. Now I'm not going to go through the whole thing in great detail, but I do want to highlight some of the most important bits. First of all, here we're looking at the parameters section. This defines the ways in which we can make the template customizable. So users need to provide a name for the function app in the appName parameter, and there are some optional parameters as well, which have got sensible defaults. For example, if you don't specify a storage account pricing tier, it will pick Standard_LRS. and if you don't specify a location, it will just use the location the resource group is in. And if you don't specify what Azure Functions runtime you want, I've chosen dotnet as the default. Next is a section called variables. Variables allow us to perform operations on parameters and other data to calculate new values. For example, the storageAccountName uses a special function called uniqueString, which will generate a random unique name that will always remain the same when we redeploy this template. The main part of an ARM template is the resources section, and this defines what resources we want to create. And here, we're creating four resource types. First is the storage account whose name, location, and pricing tier are all derived from parameters or variables. Next is the App Service Plan, which was historically called a server farm. And we're again using variables and parameters to get its name and location, but we're also saying that it's using the dynamic pricing tier, which was the original name for the consumption plan. So we're saying here that we want to use the serverless pricing model. Next up is the function app, and a function app is an App Service app with a kind of function app. Its name and location come from the properties and variables, and we're using dependsOn to indicate that it should be created after we've created the App Service Plan storage account and App Insights instance. The function app has a bunch of properties that also need to be configured. We need to say which app service plan we're hosting it on, and this uses the serverFarmId property. In this siteconfig section, we're setting up the application settings that we want. And here, we're using a built-in function called listKeys to get ahold of the storage account key to the storage account that we created in this resource group. And there are several other settings here including FUNCTIONS_EXTENSION_VERSION, which needs to be set to ~2 to indicate that we're using version 2 of Azure Functions. The node version here would only be relevant if we were using a node runtime and creating our functions in JavaScript. And we're also using some built-in functions to get ahold of the Application Insights instrumentation key. And finally, the FUNCTIONS_WORKER_RUNTIME app setting contains the runtime that we've selected, which defaults to dotnet. Finally, we're creating an Application Insights instance, which is going to be used to store the logging and diagnostic information associated with our function app. Now again, I do have to admit that these templates can be quite intimidating for new users. However, the approach I tend to take is to initially just use the Azure portal or the Azure CLI to create my resources during the early prototyping and experimentation phase of application development. And then once my application is ready for production, I might invest the time to generate one of these ARM templates like this for much easier deployment. And the benefit of creating an ARM template is going to be seen when we come to deploy it as it really is very simple. I'm going to be using the Azure CLI again to deploy it. But we could also use the Azure PowerShell cmdlets or the Azure portal directly. Here I'm showing a PowerShell script I'm going to use to deploy the function app. And

there are only two commands needed. First, we'll create a resource group to host our function app using az group create just like we saw before. And we're going to call this resource group Pluralsightfuncsarm, and then we'll pick a name for our function app. I'll just use Pluralsightfuncsarm again. And then we deploy it with the az group deployment create command, specifying the resource group to deploy to, the path to the ARM template that we want to deploy, in this case azuredeploy.json, and then we can pass in any parameters that we want to customize. We only had one required parameter in our template, which was called AppName and is used to specify the functionAppName. So I'm passing that in here. Now I'm using Visual Studio Code here, which has got a nice PowerShell extension that allows me to just select some lines of script and easily execute them. So I'm going to do that. First of all, I'll create the resource group. And once that's finished, I'm going to deploy my ARM template. And deploying the ARM template does take a minute or two, but I'll fast-forward to when it's complete. And we can take a look at what got created with the az resource list command. And here you can see the four resources that we specified in our ARM template. If we wanted to check that all the function app settings we configured were present and correct, we could use the az functionapp config appsettings list command. And sure enough, we can see that the connection strings, App Insights keys, and function app version and worker runtime settings have all been set up correctly. So although creating an ARM template can seem a bit daunting, once it's done, it really is really quick and easy to deploy instances of your Azure resources with it. By the way, when you want to clean up a deployment, it's just a one-line command to delete a resource group. Here's how you can do it with the az group delete command, specifying the name of the resource group to delete and some optional arguments to make execute deletion immediately without asking if you're sure. But I'm not ready to delete this yet because we've not yet seen how to get our function code running. We've just created an empty function app with no functions. So next, let's see how we can publish code to these function apps.

Demo: Deploying from Visual Studio

In this demo, we'll see how we can publish our functions to a function app in Azure from Visual Studio. Here I am in Visual Studio, and I can bring up the publish dialog by selecting Build, Publish, from the menu. Here it gives me an option to create a new function app. So you can actually do all those steps that we just automated from within Visual Studio if you want. But we're going to deploy to an existing function app, so I'll choose Select Existing. You'll notice that there's an option to select Run from package file here, which is marked as recommended. When you choose this option, your code gets zipped up, and special application settings are configured telling Azure Functions where to find that zip file. And there are actually two different ways in which this can be set up to work, which can make it a little bit confusing. But I'm going to show you both ways in action in this module, and hopefully it will start to make a bit more sense. So I'll select this option. And now let's choose the function app we want to deploy to, and I'm going to pick Pluralsightfuncsarm, which is the one we just created using an ARM template. Now it takes us to this dialog where we can publish. But there's another useful option here that's worth pointing out called Manage Application Settings, and this gives me the chance to set up my application settings directly from within Visual Studio. If I click on this, it shows me all my application settings with a text box showing what my local development setting contains, which is coming from my local.settings .json file, as well as what the remote value

is, which is what our Pluralsightfuncsarm function app contains. Now the first two settings we can see here are the EmailSender and SendGridApiKey. These two settings exist in my local.settings .json file, but not on the remote Azure Functions app yet because they weren't in my ARM template. I want my Azure Functions application to use the same values. So I'll click this Insert value from Local link for both of them to copy the value from local to remote. If I scroll down a bit further, I can see the AzureWebJobsStorage setting. This points to the Azure Storage Emulator locally, but to a real storage account when I'm running in the cloud. So for some settings, I do want the values to be different. But for many other settings like FUNCTIONS_WORKER_RUNTIME, for example, the value should be the same for both. When I click OK here, it will actually update my function app in Azure with those new settings ready for when I push the new version of my application code. And I can do that with publish. What's happening behind the scenes here is that it's building my code in release mode, zipping it up, and then uploading that zip to my function app. And then it's configuring some more special application settings to point my function app at that zip. Once it's done, let's jump over to the Azure portal and take a look at the Pluralsightfuncsarm function app that we just published our code to. As you can see, if I look at the functions, we can see that the three functions in our function app are present and correct. If I wanted to test these functions, then I'd need to find out the URL of the OnPaymentReceived function. I can do that by navigating to that function and clicking on the Get function URL link. And you can see that this not only shows us the URL of that function, but also includes the secret function key that we'd need to successfully authorize a call to this API. Remember we didn't need that when we were testing locally. But in the cloud, it is required because we chose the authorization level of function for our HTTP-triggered function. So I've shown you how we can publish from Visual Studio, but that's still a manual process. How can be automate publishing archived? I'll show you how to do that next.

Demo: Publishing from the Command Line

In this demo, I'm going to show you two different options for publishing Azure function apps to Azure from the command line, and these are really useful if you want to automate the publishing of your Azure function code. Functions core tools with the func azure functionapp publish command. And then we'll see how we can use the Azure CLI with the az functionapp deployment source config-zip command. Both of these demos will use the run from package deployment mode that we talked about earlier, but we're going to use slightly different variations of it. So this will give us the opportunity to look a little bit behind the scenes at what actually happens when we use run from package. Here I am at a PowerShell prompt, and I'll very quickly create a new Azure function just so we can prove that we really have updated the application. I'll just use func new from the Azure Functions core tools to do that just like we did earlier on in this course. And I'll simply select a HTTP-triggered function, and we'll call it NewFunc1. And now let's use the Azure Functions core tools to publish this function app to Azure. We'll use the func azure functionapp publish command, and we need to specify the name of an Azure Functions app. I'm going to choose the Pluralsightfuncs2019 function app that we created earlier in this module. And currently, that doesn't have any functions deployed to it at all. when I execute this command, it builds my function app. It zips it up, and it pushes that zip up to Azure. And one nice thing that we get here when it completes is that it shows us all the functions in our function app including the URLs and secret codes for any HTTP- triggered functions, and we can see the new function

that we just added. But what happened behind the scenes here? Well, the zip file that got created was actually uploaded into a blob storage container in the storage account associated with our function app. And then a SAS URI was generated that pointed at that zip file. And a new application setting was added to our function app that contained that SAS URI. If we use the az function app config appsettings list command again from the Azure CLI to list all the app settings in our function app, we can see that new setting here. In this case, it's called WEBSITE_RUN_FROM_ZIP, and it contains the URL with the location of that zipped file, which includes a secret code, also called a SAS token, which is going to secure it from anyone else downloading it. Now two things to point out here. First of all, the app setting WEBSITE_RUN_FROM_ZIP was fairly recently changed to WEBSITE_RUN_FROM_PACKAGE instead. And unfortunately, it looks like the Azure Functions core tools version that I'm using hasn't been updated yet to reflect this change. I expect both application setting names will work for a while. But by the time you watch this, I would expect WEBSITE_RUN_FROM_PACKAGE to be what gets generated. But the second important thing to know about run from package is that the zip file doesn't actually have to be stored in blob storage. There's another variation of this technique that we're going to see shortly. Let's run at az functionapp config appsettings list command again. But this time, let's point it at the Pluralsightfuncsarm function app that we deployed to with Visual Studio in our previous demo. Remember that with Visual Studio, we did check the box that enabled the run from package option. Here we can see that this function has also got the WEBSITE_RUN_FROM_PACKAGE application setting. But this time, its value isn't a SAS URI. It's just the number 1. And what that means is that the zip file we uploaded can be found in a special location in the function apps file share, and I'll show you where that is in a minute. Let's just add another function to this project with the func new command again. And once again, I'll simply select a HttpTrigger, and this will now be called NewFunc2. And now I'm going to publish this function app with the Azure CLI. So we need to zip it up first. And the Azure CLI won't automatically zip it for us, so we're going to create the zip ourselves first. This is a .NET function app so we can use the .NET Core SDK and say dotnet publish with a -c release flag to say that we want to create a release build of our function app. And we can see here that it's showing us the folder that it published the application to. And if we examine the contents, here we can see all the files that we're going to need to zip up in order to publish this application to Azure Functions. Now if you were doing this as part of a CI build process, at this point you'd automate the zipping up of this folder. But to save us some time, I'm just going to zip it up manually, which I've done off screen. And here you can see the publish.zip file that I created. So we've got a zip of our function app, and now we can deploy it using the Azure CLI with the az functionapp deployment source config-zip command. We need to specify the resource group and function name, and the src argument points at the zip file that we just created. Once this is completed, our zipped up function app has been deployed. So let's check that it works by visiting the portal. And if we look here at the Pluralsightfuncsarm project, we can see that the new version of the function app has been successfully deployed including the NewFunc1 and NewFun2 functions that we just created. So let's quickly look behind the scenes at what's going on with this version of the run from package deployment technique. First, I'll look at the application settings again. And we can see that the WEBSITE_RUN_FROM_PACKAGE app settings still has a value of 1. That tells Azure Functions to find the zip file in a specific place on the function app's file share. And there's a handy way that we can explore the file share for our function app, and that's by using another App Service feature that we can find here in the Platform features tab. If we select Advanced tools (Kudu), it launches a website known as Kudu that gives us some low-level insight into the

environment that's running our function app. If I select this Debug console view, it takes me to an Explorer view, allowing me to browse around the disk of the app service, which is actually backed by an Azure file share. And if I navigate into the data, SitePackages folder, you'll notice a bunch of zip files in here. These are the zip files that you saw me upload through Visual Studio and just now from the command line plus a few others that I did as practice runs. And every time you publish a zip with WEBSITE_RUN_FROM_PACKAGE set to the value 1, it will simply copy the zip into this folder. And then it will update the contents of this packagename.txt file to contain the filename of the latest zip. And the Kudu website lets us look inside the packagename.txt file, and we could even update it here if we wanted. So one really nice thing about this feature is that now just by changing the contents of this file, we could roll back instantly to a previous version of our function app. If I navigate back down into the home directory and go to the site folder and then into wwwroot, this is where that zip file gets unpacked to. If we weren't using the run from package feature, then any function app code that we uploaded would just get written directly into this folder. So in this demo, we've seen a couple of ways we can automate the publishing of our function apps from the command line, as well as looking behind the scenes a little at how it's implemented. Let's seen next how we can host our Azure Functions apps in a Docker container.

Demo: Deploying as a Docker Container

In this demo, I'll show you how to create a Docker container image, allowing you to deploy your function app to any Docker host. Now we could do this by writing our own Docker file, but the Azure Functions core tools can generate one for us. So let me show you how to do that. I'll create a new folder called dockerfuncs, navigate into it, and then I'm going to use func init to create a new Azure Functions app. But I'm also going to provide the --docker argument to ask it to generate me a Docker file. This will prompt me for the WORKER_RUNTIME I want to use, and the Docker file will be customized to the choice I make here. I'm going to choose dotnet. And if we look on disk to see what files got created by this command, we can see that here is our Docker file. And this Docker file is a generic Docker file for .NET Azure function apps. So we could just copy it directly into the function app that we already created, and it would just work. If we take a look at the contents of the Docker file, it's quite simple and uses a two-stage Docker build process. First, it uses a container that's got the .NET Core 2.1 SDK installed to build our application. It copies our source code in and performs a dotnet publish command just like we did manually in our previous demo. And then it uses a base container that's just got the Azure Functions runtime installed. Then it sets an environment variable that's needed by the Azure Functions runtime. And then it copies the publish code out of the builder Docker container that we created in the first stage of this Docker file. Now if you've worked with Docker files before, hopefully that will be quite straightforward for you. But if you haven't, I know it can seem a little bit intimidating. And there are plenty of great introductory courses here on Pluralsight that will help you get started with Docker. And I've actually made one called Microsoft Azure Developer: Deploying and Managing Containers. And that not only gives you an introduction to Docker basics, but it goes on to show you the various ways that you can host your containers in Azure. So do check that out if you'd like to learn more about running Docker containers in Azure. To build a Docker image from this Docker file is very straightforward. But I will quickly create a new function first. And again, we'll just make an HTTP- triggered function, and we'll call it NewFunc1. And I'm able to build a Docker image because I've got Docker Desktop installed

here on my Windows machine. If I run docker version, you can see that I've got it configured in the Linux mode. So we'll actually be building a Linux Docker image. But because .NET Core and the Azure Functions runtime is cross- platform, I could also create a Windows container image. I'll build the image with docker build, giving it a tag of docker funcs v1 and saying that the Docker file is in the current directory. Now when we run this for the first time, it will take some time as it needs to pull down those base images. First, it needs to get the SDK image. So I'll fast-forward a bit. And once that installer image with the .NET Core SDK has downloaded, we can see that it's copying in our function app source code and building it with dotnet publish. And when it's finished doing that, it's going to download the other base image, the azure-functions dotnet runtime image. So, again, let's fast-forward to when that's complete. And when it's finished downloading it, it's going to copy in the publish code from the previous stage. So now we've got a Docker image containing our function app. And that means we can run this on any Docker host whether in Azure or in another cloud or in an on-premise data center or even on our own development machine. Now there are a couple of important things to be aware of. Obviously, we've seen that Azure function apps integrate with other Azure services, such as Storage Accounts and Application Insights. And so if you're not running your Docker container in Azure, you'll either need to allow it to connect out to Azure to access those services or simply not use the Azure Functions features that depend on those Azure services. Another thing to be aware of is there currently isn't an easy way to access the function keys for HTTP-triggered functions running in a Docker container. So it would be up to you to make sure that you chose a suitable security mechanism to protect your HTTP-triggered functions if you deployed them in containers. And we will be talking a bit more about ways of securing your functions in the next module. But first, let's wrap up what we've learned in this module.

Module Summary

In this module, we took a look at the various ways of creating Azure function apps in Azure and deploying our code to them. And we saw that there were loads of options available to us. We can create function apps with the Azure CLI using the az functionapp create command. And we can also define ARM templates to allow us to deploy the associated resources, like storage accounts and Application Insights instances, with a single command. Then we saw how we can publish our function app code to those function apps. We can use the Visual Studio publish command to publish to either a new or an existing function app, as well as update application settings all from within the development environment. And we saw that if we want to automate publishing the application code, we can either use the Azure Functions core tools with func azure functionapp publish or the Azure CLI with az functionapp deployment source config-zip. I chose to use the new run from package technique for these demos, and that creates an application setting called WEBSITE_RUN_FROM_PACKAGE. And the value of this setting is either a secure SAS URI pointing to a zip file in blob storage, or it just holds the number 1, in which case the Azure Functions runtime knows to find the zip file in the D:\home\data\SitePackages folder with the packagename.txt file containing the name of the zip file that's currently live. So now we know how to create Azure function apps and deploy our code to them. in our final module, I want to give you some pointers for working with Azure functions in production.

Working in Production

Module Introduction

Hi, Mark Heath here, and in this final module of our Azure Functions Fundamentals course, we're going to focus on working in production. So far in this course, we've learned how you can create functions, use triggers and bindings, and seen how to deploy our functions to Azure. So in one sense, we're ready to go live to production. But there are still a few questions that I've not directly answered so far in this course, so in this module, we'll be answering the following questions. First, how can I monitor my functions? I'll show you how we can view the invocation history of our functions, including their log output, and keep track of how much we're spending. Second, how can I secure my functions? We'll discuss a few of the options for securing functions and learn how to manage function keys. Third, how can I configure CORS for my functions, which I might need if I was calling them from JavaScript in a web page? Fourth, how can I integrate with other Azure services? Azure Functions integrates really nicely with many other Azure services, and in this module, we'll see how Azure Functions Proxies enable us to pass incoming HTTP requests onto other destinations, allowing us to host static websites. We'll also see how we can access secrets stored in Azure Key Vault and how to integrate our function app with Azure API Management, which gives us some additional security options. Fifth, how can I build workflows with my functions? We've already built a very simplistic workflow in this course, but we'll see what Durable Functions adds to the picture. And finally, just in case I didn't address your specific question, I'll point you at some recommended resources for learning more about Azure Functions. So we've got a lot to get through. Let's start off by looking at how we can monitor our functions.

How Can I Monitor My Functions?

The first question I want to address is how can I monitor my Azure functions? How can I find out how many times they ran, when they ran, and how long they took? And how can I look back at the log output for individual function invocations? And how can I track how much I'm spending? The recommended way to monitor your Azure functions and view their invocation history is to integrate your Azure function app with Application Insights. We've already seen how to create an Application Insights instance and link it up to our function app, so in a moment, we'll see what kind of information is available to us in Application Insights. Also, the Azure portal provides us access to lots of metrics, as well as alerts and billing dashboards that can help us keep track of how much we're spending. So in our next two demos, we're going to explore some of these features.

Demo: Monitoring Functions

In this demo, we'll see how to view our Azure function invocation history with Application Insights. Here we are in the Azure portal, and we're looking at one of the function apps we deployed in the last module. As you can see, here are the three functions in our order processing pipeline. If we expand one of these, we can see that there's a Monitor tab. If we select this tab, it loads a view showing us the details of the invocations of this function, and you can see here that there's been four calls to this function, which I simply ran recently while

I was testing. We can see the date and time that the function executed, whether it was successful or not, and the duration of the function call in milliseconds. And we can see that most of these executed very quickly, but the first one was a little bit slower, and that's because of what's called a cold start. If I've not used method Azure Functions app for a while, sometimes the first call can be a little bit slower, and that's because behind the scenes a new server needs to be quickly started up to run the function. Now for many serverless applications, cold starts aren't a problem, but they are something you need to be aware of, particularly if you're implementing a back-end API that a website is going to call. Let's look next at the GenerateLicenseFile, function, and as we'd expect, there are four calls here as well, each triggered by queue messages. The first three of these were a little bit slower, although, again, that might just be a cold start issue, as it seems to have sped up by the fourth invocation. And if we look at the EmailLicenseFile function invocations, again, we see a similar pattern. It sped up after the first couple of starts. If I select one of these invocations, it shows me the log output. We can see that we get some messages from the Azure Functions runtime, telling us when we started executing the function and completed it, as well as messages showing us the log output from our function code. If I return to the invocation list, you'll notice here that there's a link to run this query in Application Insights. If I click this, it jumps me over to the Application Insights query explorer site and preloads the query that returns the details of those invocations that we were just looking at in the functions part of the portal. I can expand one of these invocations to see more details about it. The query language used by Application Insights, which is sometimes called Kusto, is a very powerful language, and it does take a bit of getting used to, so it's really helpful that these sample queries are generated for you, and you can use them as the basis for customizing them to meet your exact needs. Back in the functions part of the portal, I can do the same for the log output. If I select an invocation and click on the Run in Application Insights link again, we see another query. This time it's a fairly complex one, but the key parameter is the operation Id, which is specific to this particular function invocation. And again, we can expand this table here to see the full details of the message. So as you can see, if you choose to enable Application Insights for your Azure function app, you get access to some quite powerful querying capabilities, allowing you to explore in detail how your Azure functions are performing.

Demo: Monitoring Cost

In this demo, I'll show you some of the ways that we can monitor and control the cost of our Azure function apps using the Azure portal. Now although we've said that the serverless pricing model can dramatically reduce your costs compared to a more traditional architecture, you probably still do want to keep a close eye on how much you're spending. So let's take a look at some of the capabilities available to us in the Azure portal. Here we're looking at the same function app we saw in our previous demo. I'll navigate into Platform features for my function app, and I'll select Metrics. And you may well already be familiar with Azure Metrics from other Azure resources, as this isn't specific to Azure Functions. Pretty much every type of Azure resource provides metrics that we can graph and create alerts from. In this view, I can easily create a graph, so let's pick what metrics we want to view. I'll select the resource group and the function app that I'm interested in, and there are loads of metrics available in here, but two that are particularly interesting for function apps are the Function Execution Count, which tells us how many times our function has been executed, and the

Function Execution Units, which are measured in gigabyte seconds, and this is interesting because these are the units that are used for the pricing calculation when you're using the Consumption plan. Another thing we can do with any of these metrics is set up alerts. This is really useful if you're concerned that you might accidently run up a large bill or if you just wanted some advanced warning that your function app was working harder than normal. If you are concerned at the possibility of running up a very large bill, there's another useful feature, which is the ability to set a daily usage quota. We can configure that in the Function app settings view for our function app. You provide a daily usage quota in gigabyte seconds, so here I'll set a maximum of 20000 GB-s per day. And when I set this, it explains to me that when the daily usage quota is exceeded, this function app is going to be stopped until the next day at midnight UTC. So this is a useful setting if you're creating experimental or prototype function apps, but for a production function app, I'd probably prefer to use alerts to let me know that there was a higher-than-normal usage, rather than allowing it to automatically disable my function app. Finally, if you want to know exactly how much you're spending, then searching for subscriptions in the Azure portal takes us to a view where we can see our current expenditure for a specific subscription. Here I've navigated into the subscription I'm using for this function app, and as you can see, on this particular subscription, I'm hardly spending anything at all. In fact, most of what I'm spending comes from Azure disks and container registries. If I click on one of the items in this pie chart legend, it takes us into another view that gives us a breakdown of exactly what we've spent so far. And as you can see, my costs for these test function apps I'm creating are extremely low, just costing me a couple of pennies a month. So this is a great place to come if you want to see the breakdown of exactly what's in your Azure bill.

How Can I Secure My Functions?

One very important question when you deploy your functions to production is how can you make sure that they're properly secured? And for most trigger types, your functions are already inherently secure by virtue of the fact that only authorized users can perform the action that triggers the function. For example, if I've got a queue-triggered function, then that can only be triggered by someone who's already got permission to post a message to that queue. But for HTTP-triggered functions, how can you prevent someone who knows the URL of your function from just calling it? Well, there are several options available to us. The most obvious one, which is built right into Azure Functions, is the concept of function keys. For each HTTP-triggered Azure function, you can choose an authorization level, and the three main options to choose between are anonymous, function, and admin. Anonymous means that the Azure Functions runtime will allow all calls to your function without any credentials required, so you'd only use this if you really were happy for anyone to call your function or maybe if you're going to implement your own custom validation logic in the function code itself. The function authorization level means that the caller must supply a secret code that's specific to that particular Azure function. It can be provided either as the code query string parameter or with the x-functions-key HTTP header. And the admin authorization level means that the caller must supply a secret code that's specific to the function app as a whole, and this is useful if you've created several HTTP-triggered functions that together make up an API and you don't want your caller to have to provide separate codes for each endpoint. The Azure Functions portal provides a user interface that allows you to generate new function and admin keys, as well as cycle or revoke them. And this means that it's possible to give each

client of your API their own personalized key. Now, function keys are fine for server-to-server calls where the caller can be trusted to keep a secret, but that's not always possible. Fortunately, there are some alternative ways you can secure your HTTP-triggered functions. For example, you can also enable an Azure App Service feature called Easy Auth, which requires a valid OAuth token before accepting a call. This feature applies to your entire function app, and it can be configured to use Azure Active Directory, Google, Facebook, or several other identity providers. If you're interested in seeing a demo of how to configure Easy Auth with Azure Functions, then do check out another course I've created here on Pluralsight. It's in the Microsoft Azure Developer: Create Serverless Functions course. And the demo can be found here. However, Easy Auth isn't always an ideal choice because it globally applies to all functions, and also because it redirects you to a login page whenever you make an unauthorized call, which is only really useful when your consumers are accessing your functions from a web browser. Other security options include simply validating the HTTP headers yourself, and this gives you complete freedom to perform your own custom bearer token validation. But of course, the usual security considerations apply here. Generally, it's very risky to implement your own security code, so you should only do this if you're really sure you know what you're doing. One other option worth considering is to put an Azure API Management service in front of your function app, and this gives you access to additional security features that API Management offers to authorize callers. And then API Management, once it's authorized the callers, can pass on the calls to your function app simply using the function keys, which never need to be revealed to the clients. So there are plenty of security choices available to you. Let's see in our next demo how we can configure function keys.

Demo: Working with Function Keys

In this demo, we'll see how to use function keys and manage them in the Azure portal. Here we are in the Azure portal, and we're looking again at the Pluralsightfuncsarm function app that we deployed in our previous module. I'll navigate into the OnPaymentReceived function, which, if you remember, was an HTTP-triggered function. Now, if I go to the Manage tab for this function, once this page is loaded, we can see the function keys that are currently defined for this function. There's currently just one function key, called default, and there are two host keys, called _master and default. And the host keys are the keys that can be used to call any function in this function app. If I want to, in this view I can click to show the value of a particular secret, as well as hide it again. And there's also a convenient way for us to copy it to the clipboard. Let me navigate back to the main view for the OnPaymentReceived function, and once it's loaded, I'm going to click this Get function URL link, which shows me the URL I need to call to trigger this function, including the function key. You'll see in this drop-down that I can choose which code I want to use. I can choose any of the function keys or host keys that we saw in the Manage tab. Let's copy this whole URL to the clipboard, and we'll use it to trigger our Azure function from PowerShell. I'll paste in a command line that we've seen before in this course. It's using the PowerShell invoke web request command, and we're calling the URL that we copied from the portal, complete with the secret code, and passing some order details in the request body. And this is executed successfully and accepted our order. But let's prove to ourselves that the function security really does work. What happens if we try the same thing, submitting another order, but this time without providing the secret code in the URL? This time, we get a 401 Unauthorized, which is of course exactly what we expected and wanted. Let me also show you another way of providing this secret code. You

don't have to pass it as a query string parameter. If you want to, you can pass the code as a HTTP header. Here you can see that I'm providing the code with the x-functions-key header, and if we submit this request, we can see that again it's successful. Let's jump back into the portal and look at the Manage tab for our OnPaymentReceived function again. If I click this Renew link, it will regenerate the secret key and invalidate the original value. It prompts me to confirm, and apologies that this prompt is appearing off-screen here. I unfortunately can't drag it down, but I'll just click OK. So now we've created a new key, which we can see here starts with 6x. Let's jump back to PowerShell, and we'll try to call our function again, but we're going to use the old key, which started with NE. And as we can see, we get 401 Unauthorized, which is exactly what we expect because this original function key has now been invalidated. Of course, we don't have to use the default key. Back in the Manage tab in the portal, I can create additional keys. For example, I could create a separate key for each client that I wanted to grant access to call my function. So I could create a new key called Client1, and we can either provide our own value or let it generate a value for us, which I'll do here. And again, from this view, I can show the key, I can renew it, or revoke it if I no longer want Client1 to have access to my API. So as we've seen in this demo, it's really straightforward to manage and use Azure function keys.

How Do I Configure CORS?

One issue you might run into if you use Azure Functions to implement an API is that you might need to configure the settings for cross-origin resource sharing, also known as CORS. This allows JavaScript hosted on one domain to make calls from within a web browser to APIs hosted on another domain. It's very easy to configure CORS for a function app in the Azure portal. Let's see how we can do that. Here I'm looking at my function app in the Azure portal again, and I'm going to navigate to Platform features. Now, I'll scroll down here and select the CORS option. This allows me to specify which domain names are allowed to make calls into my function app. You can see that there are already three configured, and these are used by the Azure portal, so I recommend you don't remove these. But in this view, it's quite straightforward to add additional domain names. For example, if I wanted to host a web page on my personal website at markheath.net, I could enter that domain like this, and click Save here. It will just take a few moments to update those settings, and now any JavaScript I host on the markheath.net domain will be allowed to make calls into this function app.

What Other Services Can I Integrate With?

We've already seen how bindings allow our Azure functions to easily integrate with Azure Blob Storage and queues and table storage, but there are many other Azure services you can combine your Azure function apps with to add on all kinds of additional capabilities. Let me just highlight a few particularly interesting possibilities. First, there's Azure Functions Proxies. These allow you to redirect any incoming HTTP requests to your function app to another endpoint, and you can optionally modify the requests and responses as they pass through. You could use this to pass certain requests onto other function apps or to pass requests through to blob storage, where you might have some static HTML, CSS, and JavaScript files hosted. And that's a great option if you want to create a single-page web application with a function app serving everything from the same domain. Second, you can

place an Azure API Management service in front of your function app. This acts as an API gateway that accepts incoming API calls and then routes them onto your back-end function APIs. It also provides additional security options. It's able to verify not just API keys, but JWT tokens or certificates. It can add a caching layer or enforce usage quotas or rate limiting, and the API Management service is a particularly good fit for integrating with function apps, as it now offers a serverless pricing model itself. Third, we've seen already that we can use application settings to store secrets like we did with the SendGrid API key, but you may prefer to store your secrets in Azure Key Vault. And you can enable your function app to access secrets from Key Vault by assigning a managed identity to the function app, which creates an Azure Active Directory service principal that you can grant key vault access to. And I'll be showing how we can set this up in a demo later on in this model. Fourth, in the last module, we looked at several ways to deploy our Azure function apps, including some tools that allow us to automate deployment. If you'd like to set up continuous delivery, where after your build server has built the function app code it publishes it to a staging environment in the cloud, then a great option is to use Azure DevOps Pipelines. This comes with a number of built-in actions that simplify publishing function apps, and it can also deploy ARM templates. So this can greatly reduce the amount of work required to automate the deployments of your function apps. Fifth, if you need to integrate with lots of different Azure services, then it's worth taking a look at Event Grid. Event Grid is a service that's able to subscribe to events raised by other Azure resources and pass them on to your function app. Almost every resource type in Azure publishes events to Event Grid, and so if you want to trigger functions from things like resource group deployments completing, then Event Grid is an excellent way to achieve that. Now, we don't have time in this course to go into great detail on all these integration options, but we will take a quick look at a few of them in action. So first up, let's see how Azure Functions Proxies work.

Demo: Azure Functions Proxies

In this demo, we'll see how we can configure Azure Functions Proxies to proxy incoming traffic to our function app through to some static web resources hosted in Blob Storage. Azure Functions Proxies are configured in a proxies.json file, and one of the easiest ways to create that proxies file is to use the Azure portal to configure the proxies manually and then take a look at the JSON that gets created. I've got a function app here called Pluralsightfuncs1234 that we created earlier on in this course. If I navigate to the Proxies view, it shows me that currently we have no proxies defined in this function app. I'll click this plus icon, which allows me to define a new proxy. This one I'll call proxyHomePage, and it will have a route template of simply a forward slash, meaning that requests to the route domain are going to be handled by this proxy. I'll only proxy GET requests through. And now in the backend URL, I'll specify where I want the requests to be proxied through to. In this case, it's to an index.html file that I've already uploaded to blob storage. You can see here that I can override the incoming HTTP request before proxying it onto the back end. I can override the HTTP method, add a query string parameter, or add a HTTP header. And there's also options to override what gets returned as well. But for this demo, I don't need to do any of that. So let's go ahead and create this proxy, and when that's completed, I'll create another one. This time, I'm going to call it proxyImages, and the route template is going to be /images/, *restOfPath. And that means that any incoming requests matching the route starting with images is going to match this proxy. I'll only proxy GET requests again, and

I'm going to proxy these requests through to some files stored in blob storage. Notice here that I'm taking the restOfPath parameter from the route template and using it in the backed URL, and this matches whatever was in the incoming request URL. So if a request comes in for /images/ elephant.jpeg, then the restOfPath parameter would contain elephant.jpeg. And so now we've created these two proxies, let's see if they're working by visiting the URL for our function app, which is Pluralsightfuncs1234.azurewebsites .net. And this is going to match the proxy homepage route. So when it loads up, it loads the HTML page that I have in blob storage, and it's used the other proxy to access the three images that I've got stored in blob storage. And this particular web page is a just a very simple site that I use in Durable Functions demos. And the nice thing about a setup like this is it's really easy for us to write some JavaScript in this web page that calls through into some HTTP-triggered functions in our function app. That way, because we're doing everything through the same domain, we don't even need to set up any CORS roles. Let's see what the underlying configuration looks like. If I click Advanced editor here, it takes us to this App Service Editor, which is a new preview feature that shows us the files that make up our function app. And here we can see what the proxies.json file that's been generated for us looks like. There's a proxies object that has an entry per proxy, and the information we entered in the portal is visible here in these matchCondition and backendUri settings. So once you've seen this syntax once, it's very easy for you to use this as the basis for creating your own proxies.json files directly without needing to use the portal. Let's see next how we can access secrets that are stored in key vault.

Demo: Accessing Secrets in Key Vault

In this demo, I'll show you how we can access secrets stored in Azure Key Vault from an Azure function app. We'll use it to protect a secret key that currently we're storing directly in an application setting. And we'll be creating a system-assigned managed identity for our function app to allow it to access the key vault. Here I'm showing you an Azure Key Vault that I've already created, and I've added a secret to it called SendGridApiKey. Since anyone who's got this key can send emails as though they were coming from me, I'd really like to protect this secret securely in Azure Key Vault. In Azure Key Vault, secrets have identifiers, and I'm going to need the identifier for this secret later. I can get it by navigating into the secret, selecting the current version, and copying the secret identifier here. Next, I want to grant my function app access to read the keys from this Key Vault, and I can do that in two steps. First, I'm going to give my function app a system-assigned managed identity. I can do that by going to Platform features for my function app and then selecting Identity. And all I need to do is turn this on, and my function app will be given an identity in Azure Active Directory that we can then grant access to our key vault. So I'll turn this on and save it, and it tells me that it's going to create an identity called Pluralsightfuncs1234. So I'll allow it to carry on, and when it's done, we need to grant this new identity read access to my Azure Key Vault. So back in my key vault, let's navigate to Access policies. I need to add a new access policy. First, I'll select the service principal that we just created, so I'll search for Pluralsightfuncs1234 and select that. The only thing my function app needs to be able to do is get a secret, so following the principle of least privilege, I'll only grant it permission to get secrets. And now let me complete creating this access policy. So now we've got two access policies for this key vault, and I can always come back here and revoke the access for my function app in the future if I want to. The final step is to set up the function app to be able to reference the

secret from key vault, rather than having the secret value stored directly as an application setting. So what we need to do in our application setting value is to use a special syntax. The syntax we need to use is @ Microsoft.KeyVault, and then SecretUri=, and then we provide the identifier of our secret that we copied from key vault. And this syntax is known as Key Vault References, and it's currently still a preview feature of Azure Functions at the time I'm recording this. So let's do this. In my function app, I'll visit the Application settings page, and I'll add a new setting. Its name will be SendGridApiKey, but the value, instead of being the actual API key, will use the special syntax that references our key vault secret. And now I'll save the application settings, and with that done, now whenever my function app code accesses the SendGridApiKey application setting, it will see the value from the key vault.

Demo: Integrate with API Management

In this demo, I want to show how easy it is to put an Azure API Management service in front of an Azure function app. First, let's click here in the Azure portal to create a new resource, and I'll search for API Management. We'll create a new API Management service, and I need to provide some configuration. I'll give it the name Pluralsightfuncs1234 and put it in the Pluralsightfuncs1234 resource group as well. I am required to provide an organization name, and that's used by a developer portal which API Management exposes to help developers who want to use our API. And we're also asked to provide an administrator email address. We need to choose a pricing tier, and until fairly recently, the options were Developer, Basic, Standard, and Premium, with each one adding additional capacity and features but going up in cost. But there's also a new tier called Consumption, which offers a serverless pay-for-what-you-use pricing model, and that makes it a great fit for combining with Azure Functions in a serverless architecture, so let's pick that option. We'll create our API Management service, and this will take a little while, but when the notification comes through that deployment is complete, we can navigate to it in the portal. Let's go into the APIs section, and we need to add a new API. A single API Management service can be placed in front of multiple function apps, but we're just going to connect to one. In the add new API view, I can select Function App, and this simplifies the process of adding an Azure function app. So let's search for our Pluralsightfuncs1234 function app. When we find it, it will automatically detect all the HTTP-triggered functions in our function app, and there's just one in this example. But if there were multiple HTTP-triggered functions, we could pick and choose which ones we wanted to expose through API Management. Next, we can configure various settings for this API like the display name and the API URL suffix, which controls the URLs that will be routed through to these function app endpoints. And the defaults it's suggested here are absolutely fine for our purposes, so let's create this now and wait for it to complete. And API Management offers us loads of flexibility to customize access to our API. For example, we can add inbound policies to all incoming requests, and here you can see some of the options. We can filter by IP address, we can rate limit the API, we can add additional headers on before passing the request on to our function app, or we could set up response caching. So there's a lot of power here to customize exactly how your API works, even before the request hits your Azure function app. And API Management also offers some great security features. For example, we can configure OAuth 2.0 for this API Management service, or OpenID Connect, or client-side certificates. Now in this course, unfortunately we don't have time to explore all the features of API Management, but I did want to show you just how easy it is to connect your Azure function app to an API Management service, and if you'd like to learn more about

Azure API Management, then here on Pluralsight there are some more courses that will introduce you to more of its capabilities.

How Can I Build Reliable Workflows with Azure Functions?

For the demos in this course, we've been building a very simple function pipeline. It starts with a HTTP-triggered function which writes a message into a queue, and that triggers another function that creates a blob in Azure Blob Storage, and that triggers another function that sends out an email. And we saw it was very easy to chain functions together like this to form a basic workflow. However, when you start creating more complex workflows, you will start running into some limitations with this approach. For example, it's hard to know what workflows are currently in progress and how far through them you've got. There's also no single place where you can write error handling for the workflow as a whole. You'd have to put error handlers in each function individually. There's also no easy way to retry steps in the workflow in case you encounter transient errors or if you wanted to implement some more advanced patterns like waiting for external events with timeouts or performing some of your tasks in parallel and resuming the workflow when all of those parallel steps have completed. Those types of things are quite hard to achieve just by chaining functions together. Fortunately, there's an extension to Azure Functions called Durable Functions that addresses all of these concerns, by allowing you to define workflows, also called orchestrations, that consist of multiple steps, each performed by a single Azure function. With Durable Functions, you define what's called an orchestrator function, which specifies what all the steps in your workflow are, and then activity functions implement each one of those steps. Now Durable Functions is too large a topic for us to cover in this module, but I have created a whole course about it here on Pluralsight called Azure Durable Functions Fundamentals, so do feel free to check that out if you want a detailed look into how Durable Functions work and what you can do with them. But I do want to quickly show you what our order processing pipeline would look like if we implemented it with Durable Functions. So let's see that next.

Demo: Durable Functions

In this demo, I'm going to show you what the order function pipeline we created in this course would look like if we used Durable Functions. It will still write the order to table storage, create a license file in blob storage, and send an email to the customer. But now each of those steps will be implemented as activity functions. And although we don't have time to go into the details of exactly how Durable Functions work, I hope this will give you a feel for what a durable workflow looks like in code. Here we are in Visual Studio, and I've already converted the workflow to use Durable Functions. So let's quickly look at each of the five new functions I've created. The first is the webhook that receives the payment notification from our external system, and I've called this OnPaymentReceived2, just so that it doesn't clash with our existing OnPaymentReceived function that we created earlier. And this is just a normal Azure function with a HttpTrigger. But it's also got a binding to a DurableOrchestrationClient, and we can use that to start a new durable workflow. And we do that here with the durableClient StartNewAsync method call. We pass it the name of our orchestrator function, which is called NewOrderWorkflow, and we can pass some information into our orchestrator. In this case, I'm

passing in the incoming order object. And StartNewAsync returns an orchestration identifier which we can use to track the progress of this orchestration if we want to. The orchestrator function in our example is very simple. It has an OrchestrationTrigger, which is bound to a DurableOrchestrationContext object, and the first thing it does is it gets hold of that input data that our starter function passed to our orchestration, which was simply an order object. Then we just call each of the three activities in turn by awaiting CallActivityAsync on the DurableOrchestrationContext. The first activity saves the order to the database, the next one creates a license file in blob storage, and the last one emails the license file to the customer. Each activity can have input data, and we're just passing the order object to each activity in this example. They can also return output data, which could be used to make decisions about what should happen next in the workflow. And although this workflow is extremely simple, Durable Functions allows us to evolve it to be more complex. Maybe we could run some of these steps in parallel, or we might want to put a top-level exception handler across the whole workflow. And here's one of our activity functions. This one saves the order to table storage. It's got an ActivityTrigger that binds to an order object, which will be populated with the activities input data. And it also has a Table storage output binding. That bit is exactly the same as we saw earlier on in this course. And Durable Functions can use any of the standard Azure Functions bindings just like regular Azure Functions can. If we look at the CreateLicenseFileActivity function, again, that takes the majority of the code from our queue-triggered function that we created earlier on in this course and simply moves it into an Activity- triggered function. It's using the IBinder technique that we learned about earlier to give us control of the blob storage file name. And the EmailLicenseFileActivity function again uses the same bindings that we've seen before in this course. So, converting existing Azure function pipelines to benefit from the added value of Durable Functions is very straightforward to implement. Let's quickly see this in action. I'm going to start up my function app, and once it's started up, you can see that now there's on OnPaymentReceived2 function. And when we call this, it's going to create a new Durable Functions orchestration. So, let's call this from PowerShell, just like we've done several times before in this course, and the only difference is that now I've updated the URL to point to OnPaymentReceived2. And this seems to have worked, so if we take a look at the functions runtime output, we can see that not only is it executing our orchestration function, which was called NewOrderWorkflow, but we can see it's already finished and called the EmailLicenseFileActivity function. And if you've got very keen eyesight, you might notice that we're actually sending two emails now because my original blob-triggered function has also picked up on the blob that was generated by my durable workflow, so I'd probably want to delete my original pipeline after migrating to Durable Functions. And we can check in Storage Explorer that the new order is present in the orders table, and here we can see it, order 909. And let's also check that the blob did get written to blob storage in the licenses container, and here we can see it, 909.lic. So I do highly recommend Durable Functions if you find that you're starting to chain together multiple Azure functions to create workflows.

How Can I Learn More?

In this module, we focused on running Azure Functions in production. We learned about several ways to monitor our functions, including using Application Insights to view the invocation history and log output, as well as using metrics in the Azure portal to help us keep track of costs and overall usage. We discussed various ways to secure our functions

and saw how we can manage function keys for HTTP-triggered functions in the Azure portal, as well as how to configure CORS. We saw that Azure Functions integrates really well with several other Azure services and saw demos of using proxies to send incoming traffic on to other destinations like blob storage, as well as how to access secrets stored in Azure Key Vault by configuring a Managed Identity for our function app. We also saw how easy it is to put Azure API Management service in front of our function app, which could be used to add additional security or other features such as caching or rate limiting. Finally, we took a very brief look at how Durable Functions add some powerful workflow orchestration capabilities to Azure Functions. And this is great for implementing complex and long-running business processes. In this course, I've tried to give you a big picture of the main features and capabilities of Azure Functions. But of course, there's still plenty more to learn. So let me wrap up by recommending a few resources for going into more depth. First of all, the official Azure Functions Documentation is very comprehensive and well written, so I do recommend you spend some time exploring it, particularly if you want to use some of the binding and trigger types that we didn't have time to cover in this course. Secondly, here on Pluralsight there are several additional courses that you can watch to learn more about Azure Functions. For example, in my Microsoft Azure Developer: Create Serverless Functions course, I show how you can use Azure Functions to build out a REST API. And in my Azure Durable Functions Fundamentals course, I go into a lot more detail on the inner workings of Durable Functions and the advanced workflow patterns you can implement with them. And you might also want to check out this course from Jason Roberts, which looks at how we can test Azure Functions. Finally, here on GitHub, I've created a large list of recommended Azure Functions resources, and this includes loads of articles about real-world use cases of Azure Functions, as well as many sample applications. I hope you've enjoyed this course and that it's inspired you to try out building serverless applications with Azure Functions. I'd love to hear about what you create with Azure Functions, so do feel free to get in touch with your questions or comments. Thanks for watching.