Course Overview

Hi, my name's Mark Heath, and welcome to my course, Dapr 1 Fundamentals. I'm a software architect working at NICE. One of the things I really like about Dapr is that it has the potential to greatly accelerate and simplify the process of developing microservices. In this course, we're going to learn about several of the key building blocks that Dapr provides. We'll see how Dapr helps us with state management, service invocation, pub sub messaging, managing secrets, and dealing with the challenge of observability in a microservices application. By the end of this course, you'll know how to add the Dapr building blocks to your own microservices. As part of this course, I'll be demonstrating how to install a Dapr application onto Kubernetes, and so if you have some prior experience with Kubernetes, that will be useful, but I'll also be showing you how to use Dapr without Kubernetes. I hope you'll join me in this journey to learn about Dapr with the Dapr 1 Fundamentals course, here at Pluralsight.

Getting Started with Dapr

Course Introduction

Hi, my name's Mark Heath, and welcome to the Dapr 1 Fundamentals course. In this course, we'll learn about how Dapr can help us to build distributed applications. Over recent years, microservice architectures have established themselves as one of the primary approaches to building modern, scalable, cloud-native applications. They provide an elegant solution to the problems of scale and maintainability that a more traditional monolithic architecture can run into. But adopting microservices comes with some considerable challenges, as there's a lot of capabilities and infrastructure requirements that you're going to need to implement in order to successfully build and run a microservices application. And although there are many excellent existing runtimes and frameworks specifically designed to help simplify the process of microservice development, there's still plenty of room for improvement. And that's where Dapr comes in. Dapr is an abbreviation of the phrase distributed application runtime, and it takes an innovative approach to provide many of the building blocks that you need to create a microservices application. On its official website, which you can find at dapr.io, Dapr describes itself as "APIs for building portable and reliable microservices." Dapr isn't intended as a replacement for other frameworks and runtimes that you might already be using. In fact, it can work well alongside them. And Dapr is not language specific, so whatever programming language you use, you can make use of Dapr. In this course, I will be using C#, which is the language I'm most familiar with, but we're going to be mostly focusing on the capabilities of Dapr that you can apply to any language. Now, I'm not going to spend any time in this course arguing for why you should use Dapr, and that's because there's also a Dapr 1 big picture course here on Pluralsight that gives a high-level introduction to what Dapr is and why you want to use it. And I can highly recommend that you watch that course in order to get the background that you need to fully understand everything that we'll be doing in this course. But I also think that the demos you're going to see in this course will speak for themselves, and as you see Dapr in action, you'll begin to understand the value that it could add to your own microservice applications. In this module, you'll learn how to get started with Dapr. At its heart, Dapr is a collection of building blocks that help you to build distributed applications, and Dapr makes

use of what it calls sidecars to implement those building blocks. So I'll explain what Dapr sidecars are, and we'll also learn about the two modes in which Dapr can run in, self-hosted and Kubernetes, both of which we'll be using in this course. I'll also show you how to install the Dapr CLI and the Dapr runtime, and you'll want to do this if you're planning to follow along with the demos. And finally, I'll introduce you to the GloboTicket demo application, which we're going to be using to illustrate the benefits of using Dapr in a microservices application. Later on in the rest of this course, we're going to be making use of several of the Dapr building blocks within our GloboTicket demo application. We'll see how to store state, how to perform service-to-service invocation, how to communicate asynchronously with pub-sub messaging, how to access secrets securely, how bindings can simplify integration with external services, and how the observability capabilities that are built into Dapr can give us powerful insights into what's going on in our microservices application. And I'll also show you how to run a Dapr-enabled application on Kubernetes.

Version Check

Let's quickly take a look at the versions of software that this course is applicable. To this course was made using this version of Dapr, version 1.5, which was the latest version at the time of recording, and the demo application is using .NET version 6. So if you'd like to build and run the demo code, then you will need to have version 6 of the .NET SDK installed. Of course, Dapr itself doesn't require .NET 6. You can use any version of .NET with Dapr. And in fact, Dapr isn't a .NET-specific technology. It's actually written in Go, and you can use it from any programming language. New versions of Dapr are released fairly regularly, and here you can see the versions of Dapr that the material in this course applies to. And if at the time that you're watching this there's a newer version of Dapr available, then it's very likely that the demo application will be able to work on that version as well. And the reason I say that is because the Dapr development team take great care to avoid making breaking API changes when they release new versions, and this means that all of the techniques that I'm showing in this course should continue to work even when new capabilities are added to Dapr.

Dapr Building Blocks

In essence, Dapr is a collection of what it calls building blocks, and each of these building blocks implements a capability that is commonly needed when you're building a distributed application. And you can see the current list of Dapr building blocks here at the docs.dapr.io website. And this is the official Dapr documentation site, and this site will be extremely valuable to you as you build Dapr applications, so make sure you bookmark this and take some time exploring it. The building blocks that we're going to be using in this course are the service-to-service invocation, state management, publish and subscribe, resource bindings, observability, and secrets building blocks. And these building blocks greatly simplify several of the most common tasks involved with creating a microservices application. And you can also see here that there are other building blocks, such as one that lets you use a virtual actor programming model, and there's a configuration building block, which, at the time I'm recording this, is still in a preview state. Now, the great thing about Dapr building blocks is that they're all completely optional. If you've already got something that implements state management or pub-sub messaging and you prefer to use that, then that's absolutely

fine. With Dapr, you can simply choose the building blocks that are going to be most useful for your application. And this means that Dapr can be very easily introduced into an existing microservices application, as well as being a great way to significantly reduce the amount of work involved in creating a brand-new microservices application. Many of the Dapr building blocks take advantage of third-party services, and by that, I mean that the Dapr building blocks are exposing capabilities of very popular industry-standard services such as Redis, for example. And Dapr actually gives you the freedom to pick between different underlying implementations of each building block, and you can even switch between them easily. We're going to be taking advantage of this capability in our demo application to use different building block implementations, depending on whether we're running locally or in production.

Dapr Sidecars

I've said that Dapr consists of a number of building blocks, but let's talk a bit now about how those building blocks are implemented, because Dapr takes quite an innovative approach, compared to many of the other frameworks that you might be used to. Dapr isn't actually implemented as a component or a library that you load into your microservice process. Instead, Dapr actually runs alongside your microservice as what's called a sidecar, and this sidecar is a separate process. There's one Dapr sidecar for each one of your microservices. And when your application wants to make use of any of the Dapr building blocks, it does so by making a straightforward network request to its sidecar. Let me explain this with a couple of simple examples. In this diagram, I've got a simple microservice, and it wants to make use of the Dapr state store building block to store some state. And the way our microservice does that is it makes a network call to the Dapr sidecar process, and then the Dapr sidecar uses a component definition file which tells it where it's actually going to go in order to store and retrieve the state. And in this example, the Dapr building block is configured to store the state in Redis. Or let's consider a slightly more complicated example of service-to-service invocation. Imagine that we have two microservices, and Service A wants to talk to Service B. How Dapr allows us to do this is that Service A simply talks to its sidecar and tells it that it wants to invoke a method on Service B. And then the sidecar from Service A talks to the sidecar for Service B, which then forwards the request onto Service B. And then the response from Service B flows all the way back through the sidecars to Service A. And there's actually several benefits to this approach, because the Dapr sidecars are able to implement additional capabilities for us. First of all, the Dapr building block can automatically retry if it encounters any transient network issues. Second, it can also automatically encrypt the traffic between the microservices using mutual TLS, including automatic certificate rollover, which is something that can be very challenging if you try to implement it yourself. Another security benefit is that Dapr can enforce access control policies that ensure that only the services that are allowed to talk to each other can actually do so. And this approach also allows the Dapr sidecar to capture traces and metrics, which can provide us additional insights and diagnostics about what's going on with our microservices. And because all of these benefits are implemented inside the sidecar, it keeps your microservice code very simple and straightforward. And one of the key benefits of this sidecar approach is that it means that Dapr can be language agnostic. It really doesn't matter at all what language your microservice is written in; as long as you can call the Dapr sidecar with an HTTP request, you're able to take advantage of all of the Dapr building blocks. Now, you might be a little bit concerned about the performance implications of inserting sidecars

into the communication between services, and the good news is that Dapr has been extensively optimized to keep any performance impact to a minimum. And it's also worth noting that Dapr makes use of gRPC, which is a protocol that's been optimized for performance. And the Dapr sidecars use gRPC for their communications between each other.

Self Hosted and Kubernetes Modes

There are two main ways in which you can run Dapr distributed applications. The first is known as self-hosted. In self-hosted mode, you take responsibility for running a Dapr sidecar process alongside each of your microservices, and the sidecar runs on the same machine as the microservice. And the Dapr command-line tool includes the dapr run command that you can use to easily launch your application along with a sidecar. Now, obviously, any dependencies for the building blocks, such as Redis, for example, to support the state store or pub-sub building blocks do need to be running somewhere and available for the Dapr sidecar to access. And for local development, it's quite common for those dependencies to be run as Docker containers, and I'll be showing a demo of how to set that up later on. And self-hosted mode is very often used for local development, although it's perfectly valid to use it in production, for example, if you've chosen not to containerize your microservices. The second way to run your Dapr application is on a Kubernetes cluster, and this would be a very common choice for production scenarios, but again, there's no reason why you can't also use this as an option for local development and testing if your development team are comfortable with working with Kubernetes. When you deploy a Dapr application to Kubernetes, you first install Dapr onto the cluster. Once you've done this, any microservices that you install need to be annotated in a special way that causes the Dapr runtime to automatically create a Dapr sidecar alongside each of your microservice containers, and this means that your Kubernetes Pod will contain both your microservice container and a Dapr sidecar container. And if you've used service meshes in Kubernetes before, you'll recognize this kind of approach of using sidecars. And in this course, I'm going to be showing you how we can run the demo application in both modes, self-hosted and on Kubernetes.

Demo: Installing the Dapr CLI

In this demo, we'll see how to install the Dapr CLI. Here we are on the official Dapr website at dapr.io, and if I follow the Get Started link, it takes us to the documentation, which includes up-to-date installation instructions for a variety of platforms. If I look at the instructions for installing the Dapr CLI, it shows me the commands necessary for Linux, Windows, or macOS. I'm on Windows, so I'm going to copy the Windows install command to my clipboard, and I'm going to run it inside an administrative command prompt. When I run this, it downloads the Dapr CLI tool and updates my path to make that tool available. And so if I open a new command prompt and enter the dapr -v command, here I can see the version that I've got currently installed, and it's showing that the runtime version is not applicable because we haven't yet installed the Dapr runtime. We're going to be doing that in a moment. And if I just enter the dapr command on its own, this is going to give me a summary of the subcommands that we have at our disposal. Now, one quick note for Windows users, if you're trying to install the Dapr CLI, you may need to run Set-ExecutionPolicy RemoteSigned with the scope of CurrentUser in order to allow PowerShell to actually run the installation script. And another

quick tip for you, if you've already installed an earlier version of the Dapr CLI and you'd like to upgrade, then the approach I tend to take to do that is just to run the install script again, and that upgrades you to the latest version.

Installing the Dapr Runtime in Self-hosted Mode

In this demo, we'll install the Dapr runtime in self-hosted mode. Again, the most up-to-date instructions for this can be found on the official Dapr website, which I'm showing here. And one important thing to note is that by default, installing the Dapr runtime locally does expect that you've got Docker already installed on your machine. If you don't have Docker installed, then I recommend downloading Docker Desktop for Windows or Mac, although, please note that you may need to purchase a license for Docker Desktop, depending on the size of your company. Now, on my machine, I've got Docker Desktop for Windows installed, and I've set it to the Linux container mode, so I'm ready to install the Dapr runtime. Here I'm in an elevated command prompt, and I'll enter the dapr init command, and this might take a couple of minutes, as it needs to download some container images. But once it's completed, our machine is ready to run Dapr applications in self-hosted mode. There are three Docker containers that will have been started by this command, and we can see them if we run the docker ps command. There's a Redis container, which is used for the state store and pub-sub messaging Dapr building blocks; there's a Zipkin container, which is used for observability; and there's also a dapr_placement container, which is there in case you want to use the actor building block. By the way, it's not essential that you have these containers running to use Dapr in self-hosted mode. You're completely free to define your own Dapr components that point to services that aren't running locally as Docker containers. Another thing that happened when we ran dapr init is that it created a default components folder, which can be found in the .dapr folder in my user profile. So if I look inside that folder on my computer, you can see that there are two YAML files, and these define the state and pub-sub Dapr building blocks. And these default components make it even easier for us to get started with local development. And both of these components make use of that Dapr Redis container that got created for us with the dapr init command.

Demo: Installing the Dapr Runtime on Kubernetes

Let me also quickly show you how we can install the Dapr runtime onto a Kubernetes cluster. Now, before you do this, you do need to have set your kubectl context to point to the Kubernetes cluster that you want to install Dapr onto. And assuming that you're familiar with Kubernetes and the kubectl command-line tool, then you will know how to do that. But once you've done that, you can install the Dapr runtime with the dapr init -k command. So here I am at a command prompt, and I've currently got a Kubernetes cluster that's running in Azure that my kubectl context is pointing at, and so if I run dapr init -k, then we can see that it deploys the Dapr runtime into my Kubernetes cluster. And once that's completed, it prompts me to verify it with the dapr status -k command. And so here we can see that it lists all the containers that have been created as part of the Dapr control plane installation on Kubernetes. Now, they've not all started up yet, but if I wait a while and issue this command again, we can see that all of these containers are up and running on my Kubernetes cluster. By the way, if you've already initialized a previous version of Dapr and you want to

upgrade to the latest version, then, in self-hosted mode you run dapr uninstall --all and then follow that with another call to dapr init. And in Kubernetes, you can use the dapr upgrade -k command, passing in the desired runtime version with the --runtime-version flag, as you can see with this example here.

GloboTicket Demo Application

Let's take a quick look at the demo application we'll be working with in this course. The code is available both in the Pluralsight course download materials, but you can also find it on GitHub at the markheath/globoticket-dapr repository. GloboTicket is a company that sells tickets for music and theater events. They have an online website which allows customers to browse the available tickets and to place orders. And they've adopted a microservices architecture, enabling several software development teams to work independently on different areas of the application. In this course, we're going to see how Dapr building blocks can be used to simplify many of the microservices development tasks that the GloboTicket development team are going to need. We'll see how Dapr can be added to the existing GloboTicket microservices, allowing them to take advantage of Dapr's capabilities incrementally. And to keep things as simple as possible, our demo scenario is just going to have three microservices. By the way, although I'll talk a lot about microservices in this course, Dapr doesn't assume that you're necessarily using a microservices architecture, so it calls the services applications, with each application having its own sidecar and having a unique name, which is known as the app id. Let's look at the GloboTicket microservices. First is the front-end shop website. This is an ASP.NET Core website, and this is the website that GloboTicket customers visit to browse the available tickets and place orders. The second is the event catalog microservice. The front-end microservice calls the catalog microservice whenever it needs to retrieve the list of available tickets. Third is the ordering microservice. This service has the responsibility of performing the tasks that need to be done after an order is placed, such as emailing the customer with an order confirmation message. Let's see how we'll be introducing the Dapr building blocks into the GloboTicket architecture in this course. For the communications between the front-end website and the event catalog microservice, we'll be using the service-to-service invocation building block, which gives us service discovery, retries, and encrypted communication. The event catalog microservice needs a connection string in order to access a database, and so we'll see how the secret building block enables us to fetch that connection string. The front-end website allows customers to put items into a shopping basket. In the future, GloboTicket would like to create a dedicated shopping basket microservice, but for this first version, we're simply going to use the Dapr state store building block directly, which gives us a really easy way to save and load the contents of users' shopping baskets. When an order is placed, we're going to use the pub-sub messaging building block, which will allow the front-end website to publish a message and the ordering microservice to subscribe to it. And this keeps the ordering microservice decoupled, making for a more resilient architecture, and it also keeps the customers' online shopping experience as responsive as possible. We also have a scheduled task that we want to run periodically against the catalog microservice, and we're going to use the cron input binding to call an endpoint on a schedule. And the ordering microservice needs to send an email, so we'll use a Dapr output binding to send an email to an SMTP server. Finally, we'll see how the Dapr observability features enable us to understand the flow of calls between our microservices. We'll be making use of the C#

Dapr SDK to implement these features. Although you don't need to use the Dapr SDK, since you can communicate with the Dapr sidecars by just using regular HTTP requests, the Dapr SDKs simplify this task and ensure that we get the URLs and the request payloads right. If you download the demo code, you'll be able to build and run the GloboTicket demo application, either in self-hosted mode or on Kubernetes, so feel free to do that. Or you might actually prefer instead to take a microservices application of your own, and as you follow along with this course, try to add each of the Dapr building blocks into that application.

Module Summary

In this module, we've learned about some of the key concepts in Dapr and seen how we can set up a development machine in order to start using it, by installing the Dapr CLI and initializing the runtime. We've learned about many of the key building blocks that Dapr provides, including service-to-service invocation, state management, and pub-sub messaging. And each of the upcoming modules in this course is going to demonstrate the use of one of those building blocks. We saw that the Dapr architecture makes use of sidecars, which are processes that sit alongside each microservice. And we also learned about the two modes that Dapr can run in, known as self-hosted, which is quite commonly used for local development, and also running on Kubernetes. And I introduced the GloboTicket demo application that we'll be using throughout this course to put into practice the Dapr concepts we're learning about. In the next module, we'll actually get started using Dapr by using the state management building block to implement the shopping basket feature.

Implementing State Management

Module Introduction

Hi, my name's Mark Heath, and in this module, we're going to use our first Dapr building block and see how Dapr can help us implement state management for our microservices in a simple and elegant way. We're going to see what capabilities the Dapr state management building block has to offer and why we might want to use it, instead of some of the other options available to us. As this is the first Dapr building block that we're going to cover, we'll also look at the concept of Dapr components. We'll see how to configure a state store and how the Dapr components model allows us to seamlessly switch out the underlying back-end store, depending on what environment we're running in. We'll look at the URLs that we can call on the Dapr sidecar to access the state management building block easily, no matter what programming language we're using. I also want to introduce the Dapr SDKs, which make it even easier to access the Dapr building blocks from your language of choice. And in this course, we're going to be using the .NET Dapr SDK. And we'll see the Dapr state store building block in action as we implement a shopping basket feature in our GloboTicket demo application.

The Dapr State Management Building Block

The Dapr state management building block is designed for those situations where your microservices need to store state as key-value pairs. It's extremely common for services in a distributed application to have some state that they need to store, and often, you need to share that state between multiple instances of the same service running on different machines. And this means that you need a shared store that all of the instances can access. Now, of course, many microservices will already have a database, perhaps a relational or a document database, that they can use to store state, and if that's the case, then, of course, you can use that. Dapr doesn't force any of its building blocks on you, and if you've already got a solution that you're happy with, then that's fine. But the particular type of data that the Dapr state management building block helps you to manage is key-value pairs, and this might include things like temporary data, user preferences, configuration settings, or the status of currently logged-in users. And this sort of data often doesn't warrant the additional cost and complexity of being stored in a traditional database. And a better choice might be to store it in a cache like Redis. And what we want for this sort of data store is for it to be as simple, cheap, and fast as possible. The example we're going to be using in this module is to store the contents of the users' shopping basket for our GloboTicket store. Shopping baskets do need to be persisted short term, but they don't necessarily have the same high business value as the actual orders that are placed in the GloboTicket store, and so having a nice, easy to use key-value state store to keep those shopping baskets in is quite an attractive option.

Benefits of Dapr State Management

You might be thinking, well, I understand why I might want to use a key-value state store like Redis, but why don't I just use it directly? And of course you're free to do that, but let me mention a few of the benefits of using the Dapr state management building block in order to take advantage of a key-value store like Redis. First, the Dapr state management building block supports swappable back-end stores. The actual state can be stored in not just Redis, but MongoDB or Azure Blob Storage or AWS DynamoDB. There's a whole host of supported state stores. And here on the Dapr docs website, there's a list of all of the available options, and as you can see there's a really good amount of choice. The ability to switch between back-end stores is particularly useful if you want to use one back-end store for local development but a different one when you're in production. You can also configure multiple state stores. This allows different types of state to be stored in the most appropriate back-end store but doesn't require the application developer to have to learn how to use the SDKs of each one of those different back-end stores. It's also very easy to use, thanks to a simple API. You just issue HTTP requests to the Dapr sidecar in order to get and set state. By default, the state is isolated to the microservice, or what Dapr calls the application, that sets the state. But you can also configure a Dapr state store that can share state between multiple applications. And in addition to basic getting and setting of state by key, the Dapr state management building block offers several useful added value capabilities, such as control over the time to live, concurrency control using ETags, support for querying and bulk operations, and automatic state encryption with key rotation. So as you can see, Dapr offers a very powerful and flexible state management capability that's easy to use.

Demo: Using the State Management Building Block

Let's actually see the Dapr state management building block in action by accessing it directly with HTTP requests. Here I am in a command prompt, and because I've installed the Dapr CLI and I've initialized Dapr in self-hosted mode on this computer, I can start a Dapr sidecar with the dapr run command, and I'm using the dapr-http-port switch to say that we want the Dapr sidecar to listen on port 3500. Normally, we'd pass several additional arguments to dapr run, and we're going to be seeing those later, but for now, we're simply running a Dapr sidecar on its own. You can see here that there's a log message saying that the Dapr sidecar is up and running. Let's make some HTTP requests to this Dapr sidecar. And to make it easy for me to show you issuing HTTP requests, I've decided to use this Visual Studio Code extension called REST client, and this just allows me to show you in text format the URLs we're calling and any payloads of the requests and see the response. So the first one is a POST to localhost port 3500, which, if you remember, is the port that the sidecar is listening on. And we're going to call the v1.0/state/statestore endpoint. I'll be explaining more about how this URL works in a little bit. The body is a JSON array of objects with keys and values. We're only setting one state value here with a key of color and a value of red. I'll send this request, and we can see that it's been accepted with a 204 response code. And now, I'll issue a GET request to the same URL but with the key name of color appended to the end. As you can see, we get a 200 OK response, and the body contains the value of the state, which was red. Now, at this point, you might be wondering where this state is actually being stored. How did the Dapr sidecar know what back-end store it should use? And to answer that question, we need to learn about components, so let's talk about those next.

Dapr Component Definitions

How did the Dapr sidecar know that we wanted our state to be stored in the Redis Docker container that's running locally? Well, when we installed Dapr in self-hosted mode, some global components were set up for us. I can find those global component definitions on Windows in my user profile directory under a folder called .dapr. Here we can see inside that folder there's a config.yaml file, which includes the Dapr configuration for tracing. But the thing of interest to us is this components folder. If I look inside, I can see pubsub.yaml and statestore.yaml. These are the two global components that automatically got set up for us. Let's look inside statestore.yaml since that's the one we're using. This file says that this is a Dapr component, and it has a name of statestore. You can actually have multiple state stores and give each one its own name. The type says that for this state store, we want to use Redis as the backing store. And when we installed Dapr in self-hosted mode, a Redis container was automatically created for us, and this component is set up to use that Redis container, which is exposed on localhost port 6379. And if I run docker ps, I can see that Redis container is running on my PC and has the name dapr_redis. Back in the configuration file, we can see that a configuration file can contain additional settings, such as the password needed to connect, which is blank for this local Redis instance. Now, you might be wondering how you would know what the correct syntax of this component configuration file is and what the required values are. The answer is that we can get this information from the official Dapr documentation website. So, here we are looking at the list of supported state stores again. Let's navigate into the entry for Redis. Here we can see a fuller example of the settings that we can configure for a Redis state store. You can see that we can enable TLS or configure

the maximum number of retries. And if I scroll down, we can see additional help for each of the fields, including whether or not they're required. So you could use this information to point your component configuration to a completely different Redis cluster if you wanted. Or let's suppose that we wanted to use MongoDB instead. Let's navigate into the Component format page for MongoDB, and we can see the type value that we'd need to set, which is state.mongodb, and we can see the settings that are relevant for this backing store. When we started Dapr with dapr run earlier, we didn't specify a path to a components folder, and so it just used the globally defined Dapr components. This is great because it gives you a nice convenient quick start to using Dapr. But, for real-world applications, you'd typically create dedicated YAML component definition files that were just for that application. For example, in the GloboTicket demo application that we're going to be using in this course, here's the state store component definition that we use when we're running locally in self-hosted mode. It's almost identical to the global one, but notice that the store name is different. I've decided to call this state store, shopstate. And, let me also show you the one that I use when I deploy GloboTicket into Azure running on Kubernetes. Here, I'm defining that the state store called shopstate is actually using Azure Blob Storage as the backing store, and there's different metadata here that allows the Dapr sidecar to know how to connect to my Azure Blob Storage account. And we'll be seeing more of these component definition files in future demos in this course.

Dapr State Store URLs

Let's quickly review the URLs that we saw in the last demo, remember that when we use Dapr building blocks were talking to the sidecar so it's always available on localhost, whether we're in self-hosted mode or whether we're running on Kubernetes. The default port number used by a Dapr sidecar is 3500. But when you're running in self-hosted mode, there are potentially many sidecars running on the same computer, and so each one will need its own unique report number. And you can discover the port number that your Dapr sidecar is using by inspecting the Dapr HTTP port environment variable that the Dapr runtime sets up for you. The state part of this URL means that we want to use the state management building block. The other Dapr building blocks use a similar URL structure, but will have something different here. And the v1.0 simply means that we're using version one of the state management building block API. And having a version in the URL enables Dapr to offer backwards compatibility if they ever needed to introduce a new version of the state building block API that wasn't backwards compatible. Finally, the statestore part of this is the name of the statestore as it's defined in the component definition YAML file. For the GloboTicket examples that we just looked at, this would be set to shopstate. Now, the reason I took the time to show you how we can call the Dapr Sidecar directly by constructing these URLs is that you're free to interact with Dapr simply by using whatever mechanisms you already have available to you for making HTTP requests in your programming language or framework of choice. For example, in ASP.NET, I could simply use the HttpClient class and call these endpoints myself and serializing my state in and out of JSON. And this is one of the great advantages of Dapr, it really does have a minimal barrier to entry and really doesn't matter what programming language you use. As long as that programming language can make HTTP requests, you can use the Dapr building blocks. However, it would be nice if we didn't have to remember how to construct these URLs ourselves and do the work of JSON

serialization and deserialization each time we use the Dapr building blocks. And that's where the Dapr SDKs for different languages come in. So let's talk about those next.

Dapr Language SDKs

Dapr provides client SDKs for several of the most popular programming languages, and these simplify accessing the main Dapr building blocks. This page on the Dapr Documentation site is a good place to visit to check on the current availability of Dapr SDKs. At the time of recording, there are stable Dapr client SDKs available for .NET, Python, Java, Go, PHP, and JavaScript, and there are others in development. You'll also notice that some of the languages have server extensions available, such as an ASP.NET Core extension that we'll be using later on in this course. In our next demo, we're going to be using the state store in GloboTicket. And, even though we could do that without using the .NET Dapr SDK, we're increasingly going to see the value of using the SDK as we move on to some of the more complex building blocks. So, I'm going to be showing you how we can use the Dapr .NET SDK to implement the GloboTicket shopping basket. But before we're ready to integrate the Dapr state store, let me show you first how we can run the GloboTicket demo application in self-hosted mode with the dapr run command.

Demo: GloboTicket Dapr Run Startup Scripts

In this demo, I'll show you the scripts that I've created to start GloboTicket in self-hosted mode. These scripts use the dapr run command, and we'll see several of the arguments that can be used to configure the Dapr sidecar. The code for the Dapr GloboTicket demo is available from GitHub at the Mark Heath GloboTicket Dapr repository, and if you clone this, you'll have the full solution, including all the Dapr integration code, and so you can explore that in more detail and experiment with it. There are also branches in that Git repository that include a version of GloboTicket before Dapr has been added, so you can use that as a starting point if you want to follow along and add Dapr to it. The README for the repository describes a number of ways that you can run GloboTicket locally, including with Docker Compose. Docker Compose is quite an attractive option for running microservices locally, but in this course, we're not going to be using Docker Compose, and I'm just going to show you how we can start each of our microservices in self-hosted mode from the command line. Here we are in Visual Studio Code, and I've loaded up the demo project, and you can see that there are folders for each of the three microservices, frontend, catalog, and ordering. For this demo, we're only going to need to have the frontend and catalog services running. If we look inside the frontend folder, you'll see a start-self-hosted PowerShell script that has a dapr run command that starts up the Dapr sidecar for the frontend microservice and launches the frontend web server. Let's look at the additional arguments we're passing to dapr run. First, we're specifying an app-id. Each application needs a unique ID, and here you would typically just use the name of the microservice. This one is called frontend. Second, we need to tell the Dapr sidecar what port our microservice is listening on. For this application it's 5266, which you can find in the launchSettings.json file in the Properties folder, if you're wondering where I got that value from. Third, we need to specify the port that the Dapr sidecar should listen on. For this application, I'm using the default Dapr port of 3500. Fourth, we're telling the Dapr sidecar where our component definitions can be found. They're in the Dapr components

folder. If we look at this folder, you'll see a few YAML files for various building blocks that we're going to be using throughout this course. But the only one that matters for now is stateStore.yml, which we've seen earlier. It's using Redis as the backing store and giving it the name, shopstate. Back in the dapr run command script, the final parameters are dotnet run, and here, you'd put whatever command is needed to start your microservice, which for .NET Core is simply the dotnet run command. We also need to start the event catalog microservice. So, if we look in that folder, we can see another start-self-hosted.ps1 file. The command in here is almost exactly the same, except we've got a different app id, a different app port, the catalog service is listening on port 5016, and I do need a unique port number for the Dapr sidecar to use, so I've chosen 3501. So, now I've shown you these dapr run startup scripts, in the next demo we'll actually run these scripts and explore the GloboTicket website.

Demo: Running GloboTicket in Self-hosted Mode

In this demo, we're going to use the dapr run commands that we looked at previously to start up the frontend and catalog microservices in self-hosted mode and see the problems with the current in-memory basket implementation. In Visual Studio Code, I've opened up two terminal windows, which I've renamed to catalog and frontend, just for convenience. And, in the frontend window, I've navigated into the frontend folder. Let's run start-self-hosted.ps1 to run that dapr run command. You'll see there's a fair bit of logging output from the Dapr sidecar itself, but we can also see some log messages that start with APP, and these are the log messages that are coming from our frontend microservice. And, I can see here that our microservice has started up successfully. So, let's jump to our other terminal window, in which I've navigated into the catalog folder, and we'll start the event catalog service in exactly the same way with our start-self-hosted.ps1 script that runs the dapr run command. So now these two microservices are running, let's navigate in a web browser to localhost port 5266, and we can see the frontend website. Now, notice, we're not going through the Dapr sidecar here. We are talking directly to our frontend website microservice. And the ticket details that you can see here have been retrieved from the event catalog microservice, although that communication isn't actually using Dapr yet. Let's put something into the shopping basket. I'll add a ticket for this concert. And, back on the home page, you can see that the basket is showing that I've currently got tickets for one concert. Now, let's imagine that our frontend web server restarts at this point, and I'll simulate that by quickly stopping and restarting it. And now, let's go back and we'll reload the home page. As you can see now, I've got 0 tickets in my shopping basket. Well, why did that happen? Well, currently, the frontend is just using in-memory shopping baskets, which is not really a very good practice. It can't survive the restart of the service, and it also won't let us scale out because each instance of the frontend service won't know about the shopping basket contents on the other instances. So let's see how the Dapr state store building block enables us to fix this in our next demo.

Demo: Using the Dapr State Store Building Block

In this demo, we're going to change the shopping basket from using a simple in-memory cache to use the Dapr state store building block. Here we are in Visual Studio Code, and the shopping basket functionality currently resides inside the frontend website. Now, maybe in the future, GloboTicket would want to break this out into its own shopping

basket microservice, but for now, we're just keeping things very simple. The first thing we need to do is to reference the Dapr.AspNetCore NuGet package, and this package references the Dapr.Client NuGet package, which is the main thing we need for this demo. But, Dapr.AspNetCore adds in some additional capabilities relevant to ASP.NET Core that we're going to be using throughout this course, so it makes sense for us to reference this package. And I'm referencing the latest version of this package at the time of recording. The next thing to show you is in the Program.cs file for the frontend project. And this is where all of the startup code for the web server belongs. And there's only really one thing we need to do here, and that's to ensure that we call builder.Services.AddDaprClient, and this registers that Dapr client into the inversion of control container so it can be easily used in any of our services and controllers. Here you can see I've defined a simple interface that specifies the capabilities of the shopping basket. You can see here that we've got methods that let us add, update, or remove items from the basket. Now, the current implementation that the frontend microservice is using is in this InMemoryShoppingBasketService that you can see here. Here we've got some in-memory dictionaries that are storing the shopping basket contents and they're also caching details of the events that the tickets are for. And we need that because whenever we view the shopping basket in the website, we also want to see the name and date of the events that our tickets relate to. So the shopping basket actually has got two types of data that it needs to temporarily store. It needs to store the contents of the basket and also some details of the events that are in the basket. Here, you're looking at an implementation of the IShoppingBasketService that makes use of the Dapr state store building block. There's a couple of important things to note here. First of all, you can see that in the constructor, we're injecting an instance of DaprClient, and this is the class that we're going to use in order to access the Dapr state store building block functionality. Also, I've defined a constant called stateStoreName, which holds the name of the Dapr state store that we want to use, which if you remember from our YAML files is shopstate. Now, most of the implementation of this class is exactly the same as for the in-memory implementation. The only difference is how it stores the basket contents and event details cache. Let's take a look at how we save a basket into the state store in the SaveBasketToStateStore method. It's really very easy. I just call SaveStateAsync on the daprClient, passing in the stateStoreName, the key that I want to use, which I'm building from the BasketId, and the thing that I want to store as the value of that state. In this example, it's a data transfer object called StateStoreBasket, and the Dapr client is automatically going to serialize this to JSON for me. And there's a similar method here called SaveEventToStateStore, which does the same thing, but with event details. Now let's see what happens when we want to get a basket from the state store. Here in the GetBasketFromStateStore method, we simply call daprClient.GetStateAsync, specifying the stateStoreName and the key we want to retrieve, as well as the type to deserialize the state into. Most of the code here is actually dealing with what happens if the basket isn't in the state store, which prompts us to create a new basket. There's a similar method called GetEventFromStatStore, and that again uses the daprClient.GetStateAsync method to fetch details of an event from the state store. If it doesn't find it, it makes a call to the catalog microservice to fetch the data so that it can cache it in the state store. Okay, so that's the Dapr implementation of the state store, and I can quickly switch over my code to use this by changing my startup code in Program.cs to register the Dapr client state store shopping basket instead of the InMemoryShoppingBasket service. And, I was also able to change this registration from AddSingleton to AddScoped because the state is no longer stored in memory, so it doesn't matter if this class is discarded and recreated. So now we've seen the code changes needed to implement the Dapr state store building block, let's try this out.

Demo: Testing the Dapr State Store Building Block

In this demo, we'll test the changes that we just made to make use of the Dapr state store building block in GloboTicket. First of all, I'm going to do a dotnet build to ensure that we've built the changes for our frontend microservice and that everything is compiling correctly. And, now I'll run my PowerShell script again to start this service in self-hosted mode. And, the catalog microservice is actually still running on my machine, so we can try this out immediately in the browser. The first time in, as expected, my shopping basket is empty. So, let's put something into the shopping basket, and because we're using the Dapr state store building block, this now should be persisted in the Redis state store. And to prove that that's the case, I'm actually going to go and restart my frontend service again. I'll stop it, and start it up with the PowerShell script. And now, if we go back to the web page and reload, we can see that our ticket is still in the basket and we can check that all of the contents are still there and correct by looking inside the basket. Now, if you're interested in what's actually going on behind the scenes, you can run a command inside the Redis container. I've used Docker Desktop to open a bash shell into my Redis container, and if I run the redis-cli KEYS '*' command, which lists all of the keys that are stored in this instance of Redis, we can see that the keys include the state key name that we generated in code, but they're also all prepended with the application name. And that's because a single state store might be holding state from multiple applications, and this protects them of accidentally interfering with each other's state.

Demo: Using a Different State Store Component

For our final brief demo in this module, I want to quickly show you how we can use a completely different backend store for the state. And, in this example, we'll be using Azure Blob Storage. In the deploy folder, you can find the Dapr component definitions that are used when I run GloboTicket in Azure on the Azure Kubernetes Service. We'll be talking more about this later, but for now, we're just going to focus on the state store part of this setup.
In azure-statestore.yaml, we can see that we're defining a state management component, which has the same name, shopstate, but its type is state.azure.blobstorage. And, I've filled in the name of my Azure Blob Storage account, as well as the Azure Blob Storage container name that I want the state to be stored in, and I do also need to provide an account key, which is a secret. So I've used this syntax to say that the account key that's needed to connect to this storage account can be found in a Kubernetes secret. Again, that's something we'll look at in more detail later on in this course. And there's a file here called aks-deploy.ps1, which shows all the steps that you need to go through to get GloboTicket set up and running in Azure. Again, we're going to be looking at this in a lot more detail later on. But for now, I just want to focus on this line where we use kubectl apply to deploy this state store component definition file onto our AKS cluster. And this cluster has already been initialized with Dapr, as you can see here, using the dapr init -k command, so it understands what these Dapr component definitions are. And I've also set up that Kubernetes secret that we're using here with the call to kubectl create secret, which contains the storage account key. So, if I visit my cloud instance of GloboTicket, which is running in Azure on AKS and it happens to be at this IP address, you can see basically the same website and everything is working just fine. I can put items into my basket and everything seems to be persisted, as expected. If I want to check what's happening behind the scenes, I can actually go into the

Azure Portal and look at that blob storage account that I've set up as the backing store for my state store, and I can look at the blobs inside of the container that we set up to use for the state store. And, if we look at those blobs, we can see that actually the way that this Dapr component has been implemented is that for each of the keys that we store in the state store, it's created a file in blob storage. And obviously, every implementation of the Dapr state store building block is going to have its own way of implementing the state store behavior, depending on the capabilities of the service that you're using.

Module Summary

In this module, we learned about how the Dapr state management building block can be used to greatly simplify the task of adding state storage to a microservice. We learned about the concept of components in Dapr and saw the default global components that are configured when we install Dapr in self-hosted mode, as well as how we can pass in a path to a components folder when we call dapr run to allow us to specify our own component definitions. We saw that the state store building block supports pluggable backends and demonstrated this in action with both Redis and Azure Blob Storage state store component definitions. We learned that the way this works is that our application makes a call to the Dapr sidecar, and then the Dapr sidecar manages the details of how to actually talk to the underlying backend provider, whether that's Redis, or Azure Blob Storage, or something else. We used the .NET Dapr SDK to make it really easy to get and set state values from inside an ASP.NET Core application. And, of course, if you're not using C#, you can use the Dapr SDK for your programming language of choice. And, in this module, I focused on how to get started using the state management building block, and it may well be that the capabilities I've demonstrated are perfectly sufficient for you if all you want to do is just get and set state by key. However, if you'd like to dive deeper into some of the more advanced features of the state management building block, such as how to query the state, or how to set time to live on your state, then a great place to start exploring that is this page in the official Dapr documentation. Next up, we're going to learn about the service invocation building block in Dapr and see how we can use it to call the event catalog service.

Invoking Services

Module Introduction

Hi, my name's Mark Heath, and in this module, we'll see how Dapr can help us with invoking other services. In the last module, we looked at our first Dapr building block, the state management building block, which we used within a single microservice. And in this module, we're going to be using the service invocation Dapr building block to allow us to communicate between our microservices. We'll start by learning about some of the key reasons why you might want to use the service invocation building block. And these benefits include service discovery, encrypted communication, built-in tracing, and automatic retries. I'll explain how the service invocation building block actually works behind the scenes, and we'll see how to construct the URLs that allow Dapr to direct your requests to the correct

service. We'll also update our GloboTicket demo application to use the service invocation building block to support the frontend microservice calling the event catalog. microservice.

Service Invocation Challenges

When you build a distributed application, you're likely to want to make a call from one microservice to another. On the surface of things, this seems like it ought to be relatively straightforward, but there are some common concerns that need to be addressed. First of all is the challenge of service discovery. How can I find out the IP address of the server that's hosting my other service? Ideally, I'd just like to address the microservice by name, for example, the catalog service, and for my request to get automatically routed to the correct place. Obviously, there are other technologies apart from Dapr that can solve this problem, but one of the nice things about using Dapr is that it's going to work in exactly the same way, whether we're running on Kubernetes in production or running locally in Dapr's self-hosted mode. Second, whenever we make calls between microservices, we want that communication to be secure. Often in a microservices architecture, your microservices may all be hosted on a virtual network, which protects you from malicious callers outside the network perimeter being able to call your microservices or listen in to traffic between the microservices. But nevertheless, it's considered best practice in microservices to use a defense-in-depth approach to security where we make use of multiple levels of security. We'd ideally like our communication between microservices to be encrypted and to restrict communications between microservices to only allow the ones that are supposed to be talking to each other. Another thing we'd like in a microservices environment is observability of calls between microservices. It can be really hard to debug problems when there's communication between microservices involved. And one advantage of sending all of your service-to-service traffic through the Dapr sidecar is that it can capture tracing information using the open telemetry standard, allowing you to view that telemetry in tools like Zipkin. The final benefit I'll mention is the need to retry when there's a transient error. In distributed applications, you can be sure that from time to time there are going to be network connectivity issues, or maybe the service that you're trying to reach is temporarily unavailable. Of course, you can implement your own service invocation retry logic yourself; however, you'd have to remember to do that everywhere where you were making a service-to-service call. With the Dapr service invocation building block, you can get retries automatically.

How it Works

Let's see how service invocation works behind the scenes. And a great place for us to learn about service invocation is, of course, the official documentation here on docs.dapr.io. Here, we're looking at an overview of the service invocation building block. In particular, let's focus on this diagram. This diagram is showing two microservices, Service A and Service B, and Service A wants to make a request to Service B. Without Dapr, Service A would need to find out the IP address of Service B and then make a direct call and handle any concerns like encryption and retries itself. But when we're using the Dapr service invocation building block, instead of Service A talking directly to Service B, it simply talks to its sidecar, which is available on localhost. The Dapr sidecar then uses Dapr's name resolution component to find out the network address of the target service. Now, if you're running in Kubernetes, the

name resolution component is just the standard Kubernetes DNS service. And if you're running in self-hosted mode, Dapr uses something called mDNS, or multicast DNS. And Dapr actually supports pluggable name resolution components, so you can swap this out for something else if you have the need to. However, for the vast majority of cases, these default name resolution components will work just fine. Now we're up to step three in this diagram, which is where the Dapr sidecar for Service A passes the invocation request onto the Dapr sidecar for Service B. You'll notice on this diagram that it shows that this call is encrypted using mutual TLS, and you might also notice here that it tells us that the communication between Dapr sidecars always uses gRPC, which offers improved performance compared to a regular HTTP request. Step four is where the Dapr sidecar for Service B passes the request on to Service B. Service B then responds to its own Dapr sidecar, and that gets proxied back, again, encrypted and using gRPC for performance over to the sidecar for Service A. Finally, in step seven here on the diagram, the response from Service B is passed back to Service a. And so essentially, what Dapr is doing is acting as a proxy. We just need to communicate with our sidecar, and we allow Dapr to take control of the rest. By the way, if you're concerned that this proxying might introduce a performance degradation, then you can take a look at this helpful performance summary for Dapr provided in the official documentation. Now there's a lot of useful information here, but the short summary is that you can expect the overhead to be in the region of one to two additional milliseconds on each request, which actually compares quite favorably with many commonly used service meshes on Kubernetes. And, of course, you need to remember that you're getting quite a lot of added value for that small addition to the time of each request.

Service Invocation URLs

We've just seen how the service invocation building block works under the hood, but how do the URLs that we call when we want to talk to another microservice need to change if we want to use that building block? Well, currently in the GloboTicket demo application, the frontend service makes a call to the event catalog microservice to ask for details of an event, and the URL that it calls might look something like this. We're requesting details of event number 12345, and in this example, I'm assuming that we've got some kind of DNS resolution already set up that knows where the event catalog service is located. So how does that change with Dapr? Well, instead of talking directly to the event catalog service, we instead talk to the sidecar of the service that's making the request, in this case the frontend service, and here you can see the URL that we called. We're talking to our Dapr sidecar, which is available on localhost, and in this example, I'm assuming it's using the default Dapr port of 3500, and we're telling Dapr that we want to invoke another service, and the name of the service that we're invoking is the catalog service, or in Dapr terminology, the catalog application. We also need to specify what method we want to call. The method is simply the endpoint on the service we're calling, and in this example, it's event/12345, the same endpoint that I showed you in the previous example. Now, although this URL is a bit longer and more complex than the one that you'd call to make a direct request to another microservice, in practice, it doesn't really matter because this whole section on the left up to /method remains constant, and so in most programming frameworks, you can simply set this up as the base address, and then your code just needs to add on the actual endpoint that you want to call. However, if you don't like this long prefix, Dapr actually offers an alternative approach where we just make a call to the sidecar as though it were the target service. So

with this approach, the URL we'd call looks like this. Of course, now the URL doesn't contain any information about which microservice we want to talk to, and so you have to supply a Dapr ID header with the name of the target microservice for this technique to work. Let's see this in action now, and I'll show you both types of URL.

Demo: Invoking a Service Through the Dapr Sidecar

In this demo, we'll see the service invocation capabilities of Dapr by directly making a HTTP request to the Dapr sidecar. First, let's start the event catalog microservice in self-hosted mode just like we did before with the start-self-hosted PowerShell script file inside the catalog folder. This starts both the event catalog microservice and the Dapr sidecar for that microservice, which is listening on port 3501. Now, I'm going to start by just calling the event catalog microservice directly, and this is so that you can see the URLs that it's expecting. Once again, I'm using the REST client extension in Visual Studio Code, which gives us a nice, easy way to issue HTTP requests that I've defined in a simple text file that we're looking at now. The event catalog microservice is listening on port 5016. So if I call the event endpoint, you can see that I get a JSON list of events back, and I can also pass one of the event IDs in to get details of a specific event. So now, instead of calling the event catalog microservice directly, I'm going to talk to the Dapr sidecar, which is listening on port 3501. I'm saying that we want to invoke the event catalog service, and the method is just /event, which is to get the full list of events. As you can see, the Dapr sidecar simply passed that method on to the actual event catalog microservice and proxied the response back to us. So we've got exactly the same response as we did when we called the event catalog /event endpoint directly. Let's try the same thing, but with an event ID as well. And you can see that once again we get the same result, as if we'd gone directly to the event catalog microservice. Now, in these examples, I'm talking to the sidecar of the event catalog microservice because that's the only sidecar that I've got running at the moment. But I could actually send this request to any Dapr sidecar, and it would work out where to route the request to. Finally, before we move on to using this in code, let's look at the alternative syntax I mentioned that gets rid of the v1.0/invoke/catalog/method part of the URL. Here, I'm still talking to the sidecar on port 3501, but you can see that the rest of the URL is just asking for specific event details, and then I'm passing in a header called dapr-app-id with the value of catalog, which tells the Dapr sidecar that I want to invoke this method on the application with the name catalog, and you can see that this works as well. Now all the examples that I've just shown you happen to be using the GET http method, but the Dapr sidecar can use any HTTP method, so we could use POSTs, and PUTs, and DELETE requests, and they would all work in exactly the same way. So now we've hopefully got an idea of how the service invocation building block works and what the URLs that we need to call look like. Let's update the GloboTicket demo application to use the service invocation building block.

Demo: Update GloboTicket to Use Service Invocation

In this demo, we'll update the GloboTicket application to use the Dapr service invocation building block to allow our frontend microservice to call the EventCatalog microservice. First, let me show you how the frontend application currently makes requests to the EventCatalog microservice. Here in the EventCatalogService class in the frontend

application, you can see that it's very simple. We get a HttpClient passed into our constructor, and this has already been set up to point to the correct base address for the EventCatalogService. The method to get all events is called GetAll, and we simply make an HTTP get request by calling GetAsync on the event endpoint and then deserializing the response. And to get details of a specific event, we call GetAsync on the event/ event id endpoint. And, of course, there's nothing in here at all about where the EventCatalog/Service is located, and that's because it's all been configured in the startup. So here in Program.cs, which is where the code that runs when our service starts up is, you can see that we're using AddHttpClient to set up the HttpClient that's being used by the EventCatalogService, and we're setting the BaseAddress property of that HttpClient to a value that we're reading out of Config. And if I jump into my appsettings.Development.json file for the frontend project, you'll see that we've got the EventCatalog's base address set up here for when we're running locally in a development environment, which is localhost port 5016. And this is quite typical for what you'll see in many real-world microservice applications. They need to be told through configuration where to find the other services and often that's different depending on what environment you're running in, and this can be a pain to set up correctly and a source of frustrating errors. Let's switch this over to use Dapr. It's really quite straightforward to do. All we need to do is change the BaseAddress of this HttpClient to the address of the Dapr sidecar. The Dapr sidecar port is available to us using the DAPR_HTTP_PORT environment variable. So I am fetching that with a call to GetEnvironmentVariable, and we can use that port number to construct the service invocation BaseAddress, which is following the pattern that we looked at earlier in this module. The BaseAddress is now http://localhost on the daprPort number, and then v1.0/invoke/catalog, and catalog obviously is the name of the service that we want to talk to, and then method. And this means that because our EventCatalogService class was just using relative endpoints, those are just going to get appended onto our BaseAddress, so this call to event actually results into a call to this full URL, which includes the BaseAddress of the Dapr sidecar. And because we're now using Dapr for service discovery, we don't need these settings anymore, so I can delete them, and I'll save my changes. Let's run this and check that it's all working. I'll start the EventCatalog microservice first using the start-self-hosted script in the catalog folder, and I'll start the frontend microservice, again, using the start-self-hosted script in the frontend folder. Now if we visit the website, which is on localhost port 5266, everything looks exactly the same and we can see a list of events, but this time, rather than directly knowing where to find the EventCatalog microservice, we've retrieved this data through the Dapr sidecar, and that means we're already benefiting from the additional security, retries, and observability that the Dapr service invocation building block has to offer, and we only needed to make a very minimal code change.

Demo: Using the Dapr SDK for Service Invocation

In our last demo, we used Dapr service invocation without using the Dapr SDK at all, and that's because service invocation in Dapr is so straightforward to use that you might not feel that the Dapr SDK offers any benefits. However, the SDK does provide us a feature that helps us to do service invocation, and you may prefer that. So in this short demo, let's switch over to use the Dapr SDK instead. What the Dapr SDK does is that, instead of us having to know how to construct this special URL and having to get hold of the Dapr HTTP port ourselves, instead, we're going to register our EventCatalogService as a singleton, and into

the constructor, we'll pass a HTTP client that's been created with the DaprClient.CreateInvokeHttpClient method, and we need to pass in the name of the target microservice or application, which is catalog, and this is going to create an HttpClient that's already set up for us with the correct base address, and this simplifies our code even further. Let's quickly test and make sure it's still working as expected. I'm going to rebuild the frontend service and start it up again. And now when I load the website in my browser at localhost port 5266, we can see that our event catalog is still loading correctly.

gRPC and Access Control Lists

Before we wrap up this module, I did want to mention a couple of additional capabilities of Dapr's service invocation building block that you might be interested in. And the first of those is that, as well as accessing regular HTTP APIs that we've seen so far, you can also use Dapr to call services exposed via gRPC. Now we've already said that Dapr uses gRPC internally for sidecar-to-sidecar communications, which offers improved performance, but this expands on that to allow Dapr service invocation to be used to call APIs that are exposed using gRPC. Now, this feature is still in preview at the time I'm recording this. And, of course, gRPC is still a relatively new technology, but it is growing in popularity due to the performance benefits it offers. So if you're considering Dapr and you'd like to use gRPC for your own microservice APIs, then you may want to explore this option. The second feature I want to mention is the ability to configure access control for service invocations. By default, Dapr will allow you to make service invocations to any other service. But Dapr offers the concept of access control lists, which allows you to choose which services are allowed to make requests, what endpoints they can call, and you can even go to the level of specifying what HTTP verbs such as GET or POST are allowed. And this page in the official documentation gives some examples of how to configure common scenarios. So you can use these as starting points for configuring your own access control rules. Access control lists give you the ability to adhere to the principle of least privilege, which is a security best practice where you only allow services to perform the specific actions that they ought to be allowed to perform. And that's another potential reason why you might want to adopt Dapr if you'd like to have this level of control..

Module Summary

In this module, we've seen how Dapr gives us a simple way to make requests to other services. We saw that the benefits include service discovery, built-in retries, traceability, and encryption with mutual TLS. We saw the structure of the URL that you need to use in order to use the service invocation building block and how the Dapr SDK can construct that for us. We also learned a bit about how service invocation works behind the scenes, proxying the requests between the Dapr sidecars of both services. Of course, in a microservices architecture, sometimes rather than making a direct service invocation to another service, we'd like to use asynchronous messaging instead. And that's what the Dapr pub/sub messaging building block can offer and we'll be looking at in the next module.

Communicating using Pub Sub Messaging

Module Introduction

Hi. My name is Mark Heath, and in this module, we'll see how the Dapr pub/sub messaging building block enables microservices to communicate with one another asynchronously using messaging. We'll start out with a quick look at how pub/sub messaging works and why we might want to use it in a microservices architecture. Then we'll see some of the advantages of choosing the Dapr publish and subscribe building block and learn a bit about how it works behind the scenes. In particular, we'll see that there's a few different choices for how we can tell Dapr that we want to subscribe to messages on a particular topic, and we'll actually use the pub/sub building block in our GloboTicket demo application to send a message to our ordering microservice whenever a new order is placed in the front-end website.

Pub Sub Messaging

In our last module, we saw how one microservice can directly call another microservice using the Dapr service invocation building block, and the example we looked at where the front-end website needed to get a list of events from the event catalog microservice was a good example of where this makes sense. We need that information immediately, and so we make a direct synchronous call using HTTP, which is, of course, proxied through the Dapr sidecars. But there are many other scenarios in microservices where it might be better to use messaging instead. For example, when the customer places an order in the website, we need to pass details of the order onto the ordering microservice, which might have a whole host of responsibilities, such as ensuring that payment is taken and emailing a confirmation to the purchaser. However, from a user interface perspective, all that we need to do on the website is say to the customer, thank you for your order. They don't need to sit and wait for all the back-end tasks associated with that order to complete. Those can be done asynchronously in the background, which we can accomplish by sending a message. There are several reasons why using messaging for communication between microservices is beneficial. With messaging, rather than communicating directly to the target service, you publish a message to a message broker. Then that message broker can either push that message to the recipient or store it temporarily until the recipient asks to receive it. And this means that if the recipient of your message isn't currently available, you can still send the message knowing that it will get picked up later. Another advantage of this pattern is that it can be expanded to support multiple subscribers. In other words, you can publish one message to what's known as a topic, and then multiple services can register their interest in receiving that message by creating what's known as a subscription. In the example we're looking at, you could imagine that when an order is placed, the two activities of taking your payment and emailing you your tickets might be handled by different microservices, and so a single message indicating that a new order has been placed could be published to a topic called orders, and then the orders topic could have two subscriptions, one by a payments microservice and one by an email microservice. Notice that even though we only send one message to the topic, both subscribers get a copy of that message. And this is a very extensible model. If in the future there's something else we want to do whenever a new order is placed, we can simply create a

new subscription on that topic. So having a message broker that supports pub/sub messaging allows us to write our software in an event-driven manner where microservices simply publish what are sometimes called domain events, messages that indicate that something of interest has happened in the system and other microservices subscribe to those domain events and carry out different activities. This greatly reduces the coupling between services since the publisher of a message doesn't need to know anything about who the subscribers are. Using messaging also has scalability benefits because if we build up a backlog of messages, we can scale out additional instances of the subscriber microservice in order to catch up. And it has resilience benefits as subscribers can catch up on messages they've missed if they have any temporary downtime. Now that we've understood the basics of pub/sub messaging and the reasons why we might want to use it, let's see how Dapr makes this capability available to us.

Dapr Pub Sub Building Block

Let's take a look at how we can publish messages and subscribe to them using the Dapr pub/sub messaging building block. A microservice that wants to publish a message makes an HTTP post to the Dapr sidecar using this URL format, which should be looking quite familiar to you by now. This is saying that we're accessing the publish capability of Dapr, and we also need to specify the name of the pub/sub component that we want to interact with. In this example, it's just called pubsub. And in most systems, you'll probably just have a single Dapr pub/sub component, but you can have more than one if you want. The final portion of this URL is the topic name that we want to publish to. In this example, we're publishing to a topic called order, and you can define as many topics as makes sense for your application. The body of the request should contain the message payload that is going to be received by the subscribers. In this example, I'm just showing some of the details of the order. What happens on the subscriber end? Let's imagine that we've published a message to the orders topic on the message broker, and let's imagine that we've got two microservices, the payments and email microservices that both want to subscribe to the orders topic. What happens is that the Dapr sidecar for both of our subscriber microservices connects to the message broker and registers to receive messages on that topic. Then, whenever a message is posted to the topic, the Dapr sidecar receives that message and then passes it on to the microservice. And the way it does that is it calls an endpoint that you've exposed on your microservice. The name of the endpoint is configurable, but let's say for this example that it's called order. The Dapr sidecar can then make an HTTP request to your microservice, and the body of the request contains the message payload. A nice thing about this approach is that from your microservices perspective, it's a push model. Any polling of the message broker that might be required can be handled by the Dapr sidecar. Now you might be wondering at this point, how does Dapr know which of your microservices want to subscribe to a particular topic, and how does it know what endpoint to call them on with those messages? Dapr actually offers two ways of setting up subscriptions, so let's talk about those next.

Declarative and Programmatic Subscriptions

We need a way for our subscribers to register that they're interested in receiving a message posted to a topic, and Dapr allows us to do that in two ways known as declarative and

programmatic. Declarative subscriptions are where we create a YAML file similar to the component definition YAML files we've already seen, and programmatic subscriptions are where we define our subscriptions in code. Let's start by looking at declarative subscriptions. Here, we're looking in the official Dapr documentation at an example of a declarative subscription file. We can see that the kind is set to subscription, and we can give a name to this subscription, which here is just called order_pub_sub. Then in the spec, we're going to say what topic we're subscribing to, what route we want to be called with each message, which is checkout in this example, and the name of the pubsub component that the topic exists on. Notice here is the Scope section which lists two microservices. Obviously, not every microservice wants to subscribe to all messages, and so this limits the services that will attempt to subscribe. Only the microservices with the Dapr application names, orderprocessing and checkout, are going to create subscriptions and get called on the /checkout endpoint with new messages. The second method of subscribing is called programmatic. With this approach, your microservice is able to report which topics it wants to subscribe to. The way that this works is that the Dapr sidecar attempts to call the /dapr/subscribe endpoint, and if your microservice is listening on that endpoint, you can respond with details of which topics you want to subscribe to, as well as which endpoint that you want to be called whenever a new message is received. Now, if that sounds like it's a bit more complicated to set up than the declarative mode, you're right. However, the Dapr SDKs provide quite a lot of help to simplify the process, and in our demo, I'll show you how easy it is to configure the programmatic technique in .NET. While we're talking about subscriptions, it's important that you understand the concept of competing consumers when you're working with pubsub messaging. Let's imagine we have a situation where there are two instances of the payments microservice. The payments microservice is configured to receive messages on the orders topic, but what happens if we have two instances of the payments microservice running for resilience reasons or maybe we've scaled out to handle additional load. Will both instances receive a copy of the message? The answer is no because, although there are two instances of the payments microservice, there is only one subscription, and so the two instances are what's known as competing consumers. That means that the message will only be delivered to one of those instances. Let's look next at how we can configure the underlying service that acts as a message broker.

Pub Sub Component Definitions

How does the Dapr sidecar know which message broker we're using and where to find it? Well, it works just the same as we saw for the state management building block. Dapr supports many different pub/sub message brokers that can be used to implement the pub/sub messaging building block. You create a component configuration file that defines which one you want to use, along with the details of how to connect to it. Here, we're looking at the pubsub component definition file that I'm using for running locally in self-hosted mode. We can see that I'm giving this component a name of pubsub, which is the name that you put after /publish in the request to the Dapr sidecar to publish a message, and the component type is pubsub.redis. And this is in fact the default pub/sub implementation that gets set up when you install Dapr in self-hosted mode because, as we've already seen, it creates a Redis container that can be used not just for the state store, but also for the pub/sub building blocks. So in the metadata, we simply need to provide the network address of our local Redis server, which is running as a Docker container, and the password, which is blank. If we take a

look here at the official documentation for Dapr, we can see the current set of supported pub/sub brokers, along with their status of whether they're stable or still considered to be in alpha or beta. As you can see, many of the most popular message brokers are supported, including Apache Kafka, NATS, and RabbitMQ. And if I scroll down, we can see that there's also support for the messaging services that are available in various cloud providers, such AWS, Google Cloud Platform, and Azure. I'm deploying GloboTicket on Azure Kubernetes Service, so it makes sense for me to pick an Azure message broker for my pub/sub building block in production. I've chosen Azure Service Bus, so let's look at the configuration for that. Here we can see that I've given my component definition the same name, pubsub, but it's of type pubsub.azure.servicebus. For the connection string, I've chosen not to hard code my connection string in the component definition file, but instead reference a Kubernetes secret. And this is considered a best practice from a security perspective, and we'll be looking in more detail at the Dapr support for secrets later on in this course. Now we've learned about how to configure pub/sub messaging, we're ready to update the GloboTicket application to take advantage of the Dapr pub/sub building block.

Demo: Publishing a Message

In this demo, we're going to update the GloboTicket front-end web application to publish a message to a topic whenever a new order is placed. Here, we're in the components folder that we use when we're running in self-hosted mode. We already looked earlier in this course at the stateStore.yml file, which set up the shop state stateStore using our Redis Docker container as the backing store. And here, in the pubsub.yaml file, we're doing something very similar. This component is called pubsub, and it's also using Redis as the backing implementation for the pubsub messaging block. And the config file you're seeing here is exactly the same as the default global component that gets set up when you install Dapr in self-hosted mode. All of our startup scripts for running in self-hosted mode points to this components folder, which means that all of our services will be able to publish messages to a topic on the pubsub component. Now let's look at the code that actually publishes a message. Here, we're looking at the CheckoutController at the purchase endpoint, and this is called when the customer actually purchases something. You can see here that we call the SubmitOrder method on the orderSubmissionService and pass it this checkout object, which contains details of the order that's just been placed. The IOrderSubmissionService interface has just got a single SubmitOrder method. And in the demo code base, I've provided an implementation that doesn't use Dapr called HttpOrderSubmissionService, but we're going to look at the implementation that uses Dapr pub/sub messaging, which is here in DaprOrderSubmissionService. You'll notice that in the constructor, we're asking for an instance of daprClient, which is part of the Dapr .NET SDK. Then, in the SubmitOrder method, the vast majority of the code that you can see here is simply constructing the message that we're going to publish. The message object we're constructing is of type OrderForCreation, and we're setting up details of the customer and the items in the order, which we fetched from the shoppingBasketService. And you may remember that earlier in this course, we saw that the shoppingBasketService is making use of the Dapr stateStore building block. Then, to actually publish the message, we're calling PublishEventAsync on the daprClient, passing in the name of the pubsub messaging component, which, in our case is pubsub, and the name of the topic that we're publishing to, which is orders. Note that we don't have to do anything to create this topic ourselves. It will get automatically generated for

us. Finally, we pass in the message itself, which is an instance of OrderForCreation, and this is going to get serialized to JSON behind the scenes to be passed to the message broker. One important thing to note that when you're using messaging is that the format of the messages that you send between microservices is essentially a contract. Both the sender and receiver have to understand the message format, and so you should take care not to make breaking changes to the format of a message. Okay, that's all we need to do to publish a message, but we haven't written any code yet to subscribe to the message, so let's do that in our next demo.

Demo: Receiving a Message

In this demo, we'll update the ordering microservice to subscribe to the message on the orders topic that we published a message to in our previous demo. We're going to use the programmatic subscription method making use of the Dapr SDK for ASP.NET. Here, we're looking at the project file for the ordering microservice, and you can see here that we've referenced the Dapr.AspNetCore NuGet package. This not only contains the Dapr client, which we've used before, but it has some additional helpful support specifically for ASP.NET Core that we're going to be making use of. Let's take a look at the startup code in Program.cs. You can see here the AddDapr method that registers the Dapr client in our microservices inversion of control container. Again, we've done that before in this course, but there are two new things that we've needed to add. First is UseCloudEvents. This is needed because Dapr takes advantage of the cloud events specification as a message format, and this means that the messages we publish get wrapped into a cloud events message, which adds additional metadata, and what this UseCloudEvents method does is it allows us to easily unwrap those cloud events and get back to the underlying data type that our code is expecting. Another thing we've added here is in the call to app.UseEndpoints. This call to endpoints.MapSubscribeHandler is what sets up the endpoint that the Dapr sidecar is going to call to find out what topics our application wants to subscribe to, and this means that the Dapr sidecar can call our application on the /dapr/subscribe endpoint, and it will respond telling the sidecar what topics we want to subscribe to. So those are a few things we needed to set up in the startup of our microservice, but now we've done that, we can create the endpoint that's actually going to receive the messages. The method that we want to be called whenever a new message is published on the orders topic is here in the order controller. We have an endpoint called Submit, and it accepts HttpPost requests, and that's important because the Dapr sidecar will post to this endpoint whenever a new message is available, and this HttpPost attribute is simply saying that this endpoint is listening on the base route for this controller, which will be /order and accepting post requests. The way that we ensure that Dapr knows that we want to create a new subscription is by adding this topic attribute to the method. We're saying that we want to subscribe to messages posted to the orders topic on the pubsub component. The Dapr.AspNetCore SDK looks for methods that have this topic attribute and then uses that information to construct the response to the method that asks for the details of the subscriptions we'd like. The other thing to notice for now about this method is that we're expecting an OrderForCreation object to be in the message body, and this is, of course, the same type that we sent from the frontend microservice. However, remember that we said that this gets wrapped using the cloud events message format, and that was why we needed the UseCloudEvents method in our startup. With that present, the Dapr SDK is able to unwrap the payload of the cloud event into the type that we're expecting. Inside this method,

for now, we're just going to log a message to say that we have received the order, but we'll be updating this later in the course to actually send an email. Let's test this out. Here you can see the start self-hosted script for the ordering microservice, and we've seen this before with the other two microservices, so the contents won't be a surprise to you. It's exactly the same, except that we've picked the next free port number for the Dapr sidecar 3502, and we've told Dapr which ports our ordering microservice is listening on port 5293. It's pointing at the Dapr components folder, which as you remember, includes that pubsub.yml file which configures the pubsub messaging component and enables our sidecar to create a subscription and receive messages to pass on to our microservice. And so here in a Visual Studio Code terminal window, I'm going to run that start self-hosted script. And I need to do exactly the same thing for the other two services starting the event catalog microservice and the frontend microservice because all three are going to be involved in the purchase process. Now in a browser, I'm going to visit the GloboTicket application, we'll pick an event, add a ticket to our basket, and I'll go to the checkout, I'll fill in some customer details about this order, and make the purchase. And if everything has worked correctly, the frontend microservice should have published a message to the orders topic. Let's jump back and take a look at the log output for our microservices. Here, we can see that the frontend has posted an order message using the Dapr pubsub building block. And if I look at the log output for the ordering microservice, I can see that we've received that message, and we know that we've been able to deserialize it correctly because we can see the customer name.

Additional Pub Sub Building Block Characteristics

Before we wrap up this module, I did want to briefly discuss a few additional characteristics of the Dapr pub/sub building block that are worth being aware of. First, Dapr offers an at least once delivery guarantee for messages. In other words, Dapr will ensure that every published message is received at least once by every subscriber. Of course, in normal operation, messages should be delivered exactly once, but what this means is that you should try to write idempotent message handlers that are able to cope with the possibility that on occasions the same message might be delivered more than once. And by the way, this isn't a Dapr-specific thing. Lots of message brokers operate in exactly this way. Another capability that you might be aware of in existing message brokers is the ability to set a time to live, or TTL, for each message. This allows you to say that if a message is not read after a certain period of time, it can be discarded, and you can configure this on a per message or a per topic basis. Finally, there's also an event routing feature currently in preview. This allows different message types to be sent to different endpoints This page in the documentation explains more about this feature. It includes information about how you can set it up declaratively in a YAML file, as you can see here with these match rules, and also how you can set it up programmatically, which you can see here in this C# example, which uses the topic attribute that we saw in our earlier demo, but with an additional parameter that sets up the routing rule. This example shows how we could have one endpoint for messages of type widgets and another endpoint for messages of type gadgets. And this makes it much easier in a language like C# to use strongly-typed objects to deserialize those messages into, rather than having to try to write one generic handler for both types of message. Further down on this page, you can see some examples of the types of expressions that you can configure to select the messages that you want to route to different endpoints. One

particularly helpful use case is if you need to support multiple versions of a message, allowing you to receive old versions on one endpoint and the current versions on a different endpoint.

Module Summary

In this module, we've learned that the Dapr pub/sub building block makes it really simple and easy to add asynchronous messaging to your microservice application. We saw how a simple YAML configuration file allows us to point at any one of a number of well-known message brokers and that we can switch easily between them using one for local development and a different one in production. We saw that Dapr offers two different ways to subscribe to messages on a topic, declarative or programmatic, and that if you choose programmatic, the Dapr SDK for your language of choice is able to simplify the work required to configure your subscriptions. In the next module, we're going to look in a bit more detail at how Dapr can help us to manage secrets securely, and we're going to focus in a bit more detail on how we can run Dapr on Kubernetes.

Accessing Secrets Securely on Kubernetes

Module Introduction

Hi. Mark Heath here, and in this module, we're going to learn about how Dapr can help us access secrets securely, and we'll be particularly focusing on running Dapr on Kubernetes in this module. Often in distributed applications, we need to deal with secrets such as connection strings to databases or API keys to access third-party services. And I'm sure you're already aware that it's considered bad practice to hard code secrets or to check them into source control. And of course, there are plenty of existing tools that are designed to help us to store secrets securely. Kubernetes has secret support, and cloud providers like Azure offer services like Key Vault that can store secrets, and programming frameworks like ASP.NET also have tooling to support user secrets in local development. And you might wonder whether we really need anything additional from Dapr. However, Dapr's secret management support is worth considering for a few reasons. First, it's not necessarily a replacement for those services I just mentioned. Dapr builds on top of existing secret stores behind the scenes and provides us with a simplified API to access those secrets. Second, as we've already seen in this course, very often we need to make use of secrets inside our Dapr component definition YAML files, and the Dapr secrets building block makes it very easy to reference a secret from those component files. And third, one of the key benefits that the Dapr building blocks offer is swappable component implementations. By using the Dapr API to access secrets, your code for running locally can work in exactly the same way that it runs in production, even though they are storing their secrets in completely different places. In this module, I'll show you how we can configure secret stores both for local development, and we'll see how secrets work in Kubernetes. We'll see how to configure a local secret store and use it to directly fetch a secret from within our GloboTicket application. We'll also look in a bit more detail at how a secret can be referenced in a component configuration file. Finally, we'll see GloboTicket running in Kubernetes to demonstrate both ways of accessing secrets in production. As well as showing

you how to access secrets with Dapr on Kubernetes, I'll be walking you through all the steps that you need to follow if you'd like to run the demo app on a Kubernetes cluster yourself.

Dapr Secrets Management Building Block

The Dapr secrets management building block works in a very similar way to the building blocks we've already seen. It exposes a simple API that we can call to access a secret and supports several backing secret stores. The URL that we'd call to ask the Dapr sidecar to fetch a secret looks like this. Here, we're asking to retrieve a secret called mysecret from a secret store called vault. Here in the Dapr documentation, we can see the list of supported secret stores. The two that we'll be using in this course are local file for local development and Kubernetes for running on Kubernetes in production. But you can see that there's also support for HashiCorp Vault, which is a very popular secret store, as well as specific cloud stores for different cloud providers, such as Azure Key Vault. As usual, we can click on any one of these supported secret stores to get some help on the format of the configuration definition file. Here's an example of the local file SecretStore configuration file, which you might use for development purposes. This Dapr component has the name secretstore, and its type is secretstores.local.file. The metadata indicates the location of the secrets file, but be careful with the path here. The working directory is not necessarily going to be the folder that this configuration file is in. Instead, it's the folder that we launch Dapr from. I'm pointing at a local file in the same folder called secrets.json. And if we look inside, we can see that I've simply provided an example secret. Obviously, if you do use this approach for local development and your secrets.json file contains high-value secrets, then do be careful not to check that file into source control. And of course, although this is convenient for local development, you are free to use any of the Dapr supported secret stores. For example, maybe you've got an API key that you want all of your developers to share when they're working locally, but you don't want to store that in a local JSON file. In that case, you might decide to define an additional Dapr secret store that points at a secure location, such as an Azure Key Vault, and store that high-value secret in there while still using the convenience of a local JSON file for using things like the connection strings to your locally hosted databases.

Demo: Accessing Secrets Programmatically

In this demo, we're going to update the catalog microservice to request a secret programmatically by calling the Dapr secrets API. Let's start by reminding ourselves of the configuration for the localSecretStore that we looked at a moment ago. It's pointing to a local secrets file called secrets.json, which contains a single secret called eventcatalogdb. And the value I've put in here is just an example string that we can use to check that we've successfully retrieved a secret. We're not actually going to be connecting to a real database for this demo. Let's look at the code that we'd used to access a secret programmatically. Here in the event catalog microservice, we're looking at the EventRepository class, and this is responsible for fetching the list of events. In a real-world scenario, this would likely connect to a database, but in our demo it's just returning an in-memory list. However, for the purposes of demonstrating the secrets building block, let's imagine that we do need to get hold of a connection string to allow us to access a database. And you can see here I'm calling a method called GetConnectionString, which we'd like to implement using Dapr to fetch the secret from

the secret store. So let's see what we need to do to implement this functionality. To do this, we need to make use of the Dapr .NET SDK again. Even though we could just construct the URL and talk to the Dapr sidecar directly, the Dapr SDK simplifies our code by constructing the URL for us and handling the process of correctly parsing the JSON response. Here in the catalog microservice csproj file, you can see that I'm referencing the Dapr.AspNetCore NuGet package. And in the startup code in Program.cs, were calling AddDapr to register the Dapr client in the inversion of control container. And this means that back in my EventRepository class I can inject an instance of DaprClient into the constructor. And in the GetEvents method, before returning the events, I'm asking for that connection string, which we'd need if we were talking to a real database, and I'm writing it to the logs just so that we can prove this works. Let's look at my GetConnectionString method. Now I've decided to make the secret store name configurable with an environment variable, and the reason for that is simply that when we run on Kubernetes there's a default secret store called Kubernetes, and we're going to be using that later. But when I'm running locally, it would be a bit strange to use a secret store called Kubernetes, so we're going to use the local file secret store that we've configured, which is simply called secretstore. As well as a secret store name, there's a secret name. Our secret is just called eventcatalogdb. Now, here's where things can get a little confusing. An individual Dapr secret can contain multiple key-value pairs, and this is because some of the secret stores support multiple keys in a secret. Kubernetes is actually an example of a secret store that can do this. And this means that to get an actual secret value, such as a connection string, we need the secret store name, the secret name, and the key of the actual secret that we want. To actually access the secret, we call the GetSecretAsync method, passing in the secretStoreName and the secretName. Behind the scenes, this is going to make a call to the Dapr sidecar with a URL that looks like this. The GetSecretAsync method returns a dictionary so that secret stores that support multiple keys for a single secret are supported. However, our dictionary will only have one key in it, which is the secret name again, eventcatalogdb. Okay, we're ready to try this out. I'll need to start both the catalog and front-end microservices, and I'll do that in the usual way with my script that uses dapr run to start each service. Once both of those are up and running, I'm going to visit the GloboTicket home page, and that will make a call to the catalog service, which, behind the scenes, will fetch the secret that it needs for the database. And we can check that it works by looking at the log output for the catalog microservice. And sure enough, here we can see the log message which includes the secret value that came from the local file.

Demo: Running On Kubernetes Part 1 - Creating Azure Resources

For our next demo, we're going to see how to use secrets with Kubernetes, but I've not yet shown you in detail how I've got GloboTicket running on Kubernetes. And so in this first part of the demo, I'm going to show you how I created the Azure resources that I'm using, which are an Azure Kubernetes cluster, an Azure storage account for the state store to use, and Azure Service Bus for the pub/sub messaging. Now, none of these are requirements for working with Dapr, and so if you're not interested in Azure or maybe you've already got your own Kubernetes cluster that you want to use, feel free to skip on to the next part of the demo where I'm going to install the GloboTicket application onto Kubernetes. The main tool that I'm using to create my resources is the Azure CLI. This provides a really simple command line interface for creating and interacting with resources in Azure. You can download it here, and it's cross platform, so you can use these commands on any operating system. We're

looking here at the aks-deploy PowerShell script that's in the root of the demo project. With the Azure CLI, you start off by calling az login, which will prompt you to log into Azure if you haven't already, and if you have multiple Azure subscriptions, be sure to call az account set to select the one you want to use, as you can see with this example. I've already done this, so let's move on. Next, I'm declaring a few variables that we'll need. My Azure RESOURCEGROUP, which will contain all of the Azure resources I'm creating, will be called globoticket-dapr; and I've chosen the Azure westeurope region, as that's near to where I live; and I'm going to use globoticketdapr as the name for my AKS cluster. Now I'm going to run these in my PowerShell session, as we're going to be using them for some later commands. And I'm making use of a nice feature in Visual Studio Code where I can just press F8, and the lines that I've got selected will get sent into my PowerShell terminal. Now let's look at the commands that create the Azure resources. I've actually already created these in Azure with the names that I want, so I don't need to run these commands again, but I will show you what each one does. First I create the RESOURCEGROUP with az group create with the name and location I've chosen. I can run this one again, and it will do nothing because the group already exists. Then I create the AKS cluster using the az aks create command, and I'm putting it inside my RESOURCEGROUP with the name that I've chosen. To keep costs down, I've specified an initial node-count of just 1 node, and I've also chosen to enable the http_application_routing add-on and to generate ssh keys. The next command, az aks get-credentials, will update my local kubectl configuration file to allow me to use the kubectl CLI tool to manage this cluster. I've said that I want to overwrite any existing entries for this AKS cluster name. Let's run this command to ensure that my kubectl context is set up correctly. If I now call kubectl config current-context, we can see that the current context is now pointing at my GloboTicket cluster. By the way, kubectl is of course the standard command line tool for managing a Kubernetes cluster, and if you've used Kubernetes before, you'll be very familiar with it. Conveniently, if you've installed Docker Desktop, then kubectl will be automatically available to you, but you can install it separately. There are two other Azure resources that GloboTicket depends on. The first is a storage account that's going to be used for the Dapr state store building block. It needs a name, and this has to be unique across Azure, so if you're following along, you'll have to pick your own name here. I've called mine globoticketstate. I then create the storage account with az storage account create, specifying the account name, the resource group and location to put it in, and what pricing tier I want, and I've chosen the Standard_LRS. Again, I don't need to run this now because I've already created my storage account. The next two commands you see here are how I can get the storage account connection string and key. We're going to need the account key for one of the secrets later on, and in order to call the next command, we need to set up an environment variable called AZURE_STORAGE_CONNECTION_STRING, and I'm setting that up to use the STORAGE_CONNECTION_STRING that we just retrieved, and this allows me to create an Azure Blob storage container inside the storage account, and this container is going to be used by the Dapr state store. I've already done this, but here you can see the az storage container create command that I used to create a container with the name statestore. There's one final Azure resource we need to create, our Dapr pub/sub component is going to use an Azure Service Bus namespace, so I've picked a name for that, globoticketpubsub, and again, if you're following along, you'll need to provide your own unique name here. The command to create the namespace is az servicebus namespace create, and I pass in the name, the resource group and the location to put it in, as well as the pricing tier I want, which is Standard. Next, I need to retrieve the service bus connection string, which is a bit trickier, but this az servicebus namespace authorization-rule keys list command can find it for us. I'm

saying that I want the RootManageSharedAccessKey, and I'm filtering to just get the primaryConnectionString, and again, this is because we're going to use this in a secret later on. Now I appreciate that if you've not done a lot with Azure, then maybe some of that went a bit over your head, but don't worry, as none of this is necessary to use Dapr. You can work with the cloud providers and tools that you're most familiar with and create your own Kubernetes cluster using whatever backing services you need for the Dapr components that you're using. But now that I've shown you how I created my Kubernetes cluster in Azure, along with the backing services for the Dapr components, let me show you how we can install GloboTicket onto this Kubernetes cluster next.

Demo: Running on Kubernetes Part 2 - Installing GloboTicket

In the previous demo, I showed you how I created my AKS cluster and updated my kubectl context to point at it. In this demo, we're going to begin installing GloboTicket onto our Kubernetes cluster, starting by initializing Dapr on Kubernetes with dapr init -k, and we'll be building our container images and looking at the Kubernetes deployment YAML files for the GloboTicket services, and we'll see how they're annotated to ensure that they have the Dapr sidecar injected correctly. The first thing we need to do, if we haven't already, is to install Dapr onto our Kubernetes cluster. Because we've connected kubectl to the cluster, we can just use the dapr init -k command, which we saw in the first module of this course. I've actually already installed Dapr onto my Kubernetes cluster, so it will error if I call this again. But I can verify that Dapr is installed correctly by calling dapr status -k, and this shows me a summary of the Dapr containers that are running on my cluster. This includes the dapr-sidecar-injector, which is the component that ensures that all of my microservices have a Dapr sidecar running alongside them. Now we do need to containerize our GloboTicket services. The way that you containerize your application will vary depending on the programming language and framework you're using. But here you can see the Dockerfile for the front-end microservice. This was auto generated by Visual Studio for me, and it's completely standard. There are no special customizations for Dapr necessary. I've got some docker build commands here that will build each of the three microservice containers using their Dockerfiles and tag them. I'm prefixing the tags with my Docker Hub username, which is markheath. So now, the front-end service has been tagged markheath/globoticket-dapr-frontend. And I'll do the same for the other two microservices, building the catalog and ordering microservice containers and tagging them. By the way, in a real production environment, you'd likely have your own private container image registry that you push your microservice container images to. In Azure, that would typically be the Azure Container Registry. But I'm using Docker Hub here just to keep things simple, and it also allows you to use my container images if you'd prefer not to have to build them yourself. I can push a container image to Docker Hub with the docker push command using that tag that I just created. So, let me push each of these three images to Docker Hub. And once that's completed, you can see here on my Docker Hub page those images that I've just pushed. Now we need to tell Kubernetes to run these containers. In the deploy folder of the sample code, I have all of our Kubernetes resource definition YAML files. This includes the definitions of the Dapr components that we're using and the definitions of the microservices. Let's focus on the microservices first. In frontend.yaml, I'm defining a Kubernetes service called frontend, which will be exposed on port 8080 and is of type LoadBalancer. By the way, if you're not familiar with Kubernetes resource definition YAML files, then these will

probably seem a bit intimidating. I'm not going to be explaining every part of this file, but I do want to point out a few of the most important things. This file also defines a Kubernetes deployment called frontend. Let's start by looking at these annotations on the template metadata. The dapr.io/enabled annotation, which is set to true, is what tells Dapr to automatically inject a sidecar into our Pod. The app-id property is the application ID that's needed for service-to-service invocation. The app-port property tells the Dapr sidecar which ports our microservice is listening on, which is port 80, in this case. And the final annotation tells the Dapr sidecar to use a configuration called appconfig, and this is where we set up the tracing, which we're going to be looking at in more detail later on in this course. Further down, you can see that the container image I'm using is markheath/globoticket-dapr-frontend, which is the image I just pushed to Docker Hub,and I've set the ImagePullPolicy to Always, which is convenient for development, as I might quite frequently push new versions of this container image. And we're saying here that our container is listening on port 80, which is the same port that we specified in the annotation above. Now let's look at the catalog microservice YAML file. This is simpler. I didn't declare a service resource, just a deployment. It's got the name catalog and has also got the dapr.io annotations. But you'll also notice that I've set an environment variable on this container, which sets the SECRET_STORE_NAME. And that's because, if you remember earlier, we decided to make the SECRET_STORE_NAME configurable by an environment variable so that when we run on Kubernetes, we can choose to talk to the default Kubernetes secret store that gets created automatically. And there's also a YAML file here for the ordering microservice, and it's very similar to the other two that we just looked at. In the next part of this demo, I'm going to show you how we can use the kubectl apply command to install these microservices onto our Kubernetes cluster. But before we do that, we're going to create some secrets in Kubernetes and see how our Dapr component definition files can reference those secrets.

Demo: Running On Kubernetes Part 3 - Secrets In Component Config

In this demo, we're going to create some Kubernetes secrets, and then we're going to see how we can access those secrets from a Dapr component configuration file. And we'll wrap up by actually deploying our microservices and our Dapr components into our Kubernetes cluster using the kubectl apply command. First of all, let me show you the commands that I used to create my Kubernetes secrets. I've created three with the kubectl create secret generic command. The first is called blob-secret, and it contains a single key value pair where the key is account-key, and the value is the STORAGE_ACCOUNT_KEY that we retrieved earlier on with the Azure CLI. And we're doing a similar thing with the service bus secret with the connection string of the service bus being made available in a key called connection-string. And finally, the secret that we're accessing programmatically from the catalog service just has an example value. I've put Event Catalog Connection String from Kubernetes, and I've used the same value of eventcatalogdb for both the secret name and the key because that's consistent with how the local file secret store works which were using when we're running locally. With these secrets in place, our Dapr component configuration files can make use of these secrets. For example, here in the azure-pubsub.yaml file where we set up Azure service bus as the message broker for the pubsub building block, rather than hardcoding the connection string, under value, we have this secretKeyRef property that has a name of servicebus-secret, which is the Kubernetes secret name and a key of connection-string, which is the name of the key inside that Kubernetes secret. Remember, we said a single Kubernetes

secret can contain multiple keys, even though we're only using one in this example. We also need to tell it which secret store to use. So here, we're just saying Kubernetes, and this secretStore is set up by default for us when we enable Dapr on a Kubernetes cluster, which makes it really convenient to use, but obviously, if you prefer to use a different secret store, you could point it there instead. Now currently, we haven't pushed any of these component definition files into our Kubernetes cluster, and we can do that quite simply with the kubectl apply command. Here, you can see the command to apply the pubsub and statestore component definitions. By the way, this script also contains code to deploy a number of additional components for things like bindings and observability that we'll be using later. And if you want to install GloboTicket yourself, then you will need to complete these steps here as well that installs Zipkin, the cron component, and the MailDev service, and I'll be explaining more about those later on in this course. The final thing we need to do to install GloboTicket onto our Kubernetes cluster is to install the three microservices themselves, so we'll do that with kubectl apply pointing at the YAML files for each of those three microservices that we looked at earlier. And I've shown some additional helpful commands here that you can use to check that everything is running correctly on your cluster, as well as examples of how to do things like restart one of the microservices or examine the logs, which can be very useful when you're troubleshooting. Let me just run this command to see the Kubernetes deployments, for example. And here, you can see the three microservices that I've installed, as well as the MailDev and the Zipkin containers that we'll see in action later on in this course when we implement bindings and observability. And if like me you're using the Azure Kubernetes service to host your Kubernetes cluster, then this command will launch the Azure portal and navigate to the AKS cluster, which allows you to explore all of the resources configured in your Kubernetes cluster and to check on the health. Okay, now I do appreciate that was quite a lot to go through, and if you're new to Kubernetes, it might seem overwhelming as Kubernetes itself is a very powerful platform with lots of capabilities, but once you do become familiar with Kubernetes resource definitions, then you'll recognize that adding Dapr to a Kubernetes microservices application is actually relatively straightforward. In the next demo, we're going to check that this is all working as expected and that we can access secrets.

Demo: Running on Kubernetes Part 4 - Accessing Kubernetes Secrets

In this final part of our demo, we'll see the Dapr secret access in action on Kubernetes and test that it's all actually working. Basically, if we've set up everything right, our state store and pub sub components will be able to use the Azure resources that they're pointing at, and our catalog service will be able to fetch the connection string secret from the Kubernetes secret store. I've included a useful script that helps us find the IP address of the frontend service running in Azure, and I'm using PowerShell to launch a browser to that address on port 8080. And we can see it's loaded successfully, I can see the list of events, and I'll put something in the basket, and I'll quickly go through the whole purchasing process entering some data about who has placed the order. And once I've completed that, we're going to go back and view the logs to check that our secrets are working as expected. In Kubernetes, to get the logs, you need to know the pod name, and again, I've added some helpful commands here to get the pod names for each of the microservices. So I'll run these, and now I can use the kubectl logs command to ask to see the logs from the catalog pod, and in particular, from the container in that pod called catalog. If we look at these logs, we can see that the catalog

service was able to programmatically access the Connection string secret that we added to the Kubernetes secret store. And if we view the log output from the ordering microservice, we can see that we received a message about the new order, and that proves that the pubsub component was successfully able to connect to Azure Service Bus to publish and subscribe to that topic, which means that it was able to access the connection string secret. That's almost it for this module, and if you've been following along in Azure, you may now want to delete the resource group that you created with the az group delete command, and that will mean that you're no longer paying for any of the Azure resources that we created in this demo.

Secret Scoping and Module Summary

Before we wrap up what we've learned in this module, I want to highlight one additional feature of the Dapr secrets building block, and that's secret scoping. In a large application, we might have many secrets and some of those secrets are specific to a single microservice. Not every microservice will need access to every secret, and if we want to follow the security best practice of principle of least privilege, then we only want to allow microservices to access the secrets that they actually need, and this page in the official Dapper documentation gives some helpful examples showing how to achieve different scoping rules. As you can see in this example, we're only allowing access to two specifically named secrets and denying access to all other secrets, so for production scenarios, you might want to consider adding this additional level of security to your secret configuration. In this module, we looked at the Dapr secrets management building block. We saw how Dapr supports a number of underlying secret stores that your component definition can switch between. In particular, we used the local file secret store for local development and the Kubernetes secret store when we were running on Kubernetes. We saw how secrets can be accessed either from within a component definition file with the secretKeyRef property or programmatically by calling the Dapr sidecar and asking for them, and we used the .NET Dapr SDK to simplify doing that. We also spent quite a bit of time looking in more detail at exactly how we can get a Dapr application running on Kubernetes using kubectl apply to configure our microservices and Dapr components. In the next module, we're going to see how the Dapr bindings building block can greatly simplify integrating with other services.

Integrating with External Services Using Bindings

Module Introduction

Hi. Mark Heath here, and in this module, we'll see how the Dapr bindings building block can greatly simplify the task of connecting to third-party services. To help us understand what Dapr bindings do, let's remind ourselves of what happens when we use the pub/sub building block. Suppose microservice A wants to publish a message and microservice B wants to subscribe to it. Thanks to the pub/sub building block, all microservice A has to do is make a simple HTTP request to its Dapr sidecar, and the Dapr sidecar handles all the complexity of actually sending that message to a message broker, whether that be RabbitMQ or Azure Service Bus or any one of the supported pub/sub message brokers. And this has the benefit of greatly simplifying the code in our microservice. We don't need to know about all the

complexities of how to use the SDK for the particular message broker we're using. And what about the code that subscribes to the topic? Well, we saw that, again, what happens is that the Dapr sidecar takes care of actually communicating with the message broker and subscribing to the topic. And when it receives a message, it passes it on to microservice B by calling an HTTP endpoint. And this keeps microservice B very simple. It only needs to expose an endpoint and doesn't need to deal with any of the complexities of connecting to the message broker. Now, what's this got to do with bindings? Well, what bindings in Dapr do is it takes this general principle of the Dapr sidecar communicating with external services on our behalf and expands it to support a much wider range of services. For example, there are bindings for services that can send emails. So if we use that binding, our microservice could send emails very easily just by posting a message to the Dapr sidecar. And just like we saw that the Dapr sidecar can call us when a message appears on the service bus, this principle can also be expanded so that Dapr bindings can trigger requests to our microservices whenever something of interest happens in an external service. So in this module, we're going to look at the Dapr bindings building block and understand the concepts of input and output bindings. We'll look at some examples of how to configure bindings to various external services, and then we'll see them in action by using them in GloboTicket. We'll configure an output binding to allow us to send an email whenever we receive a new order, and we'll also look at the cron input binding, which gives us a really convenient way of implementing scheduled tasks.

Supported Dapr Bindings

If we take a look here at the official Dapr documentation, we can see a list of the currently supported bindings. It's quite a long list, and one of the things that you might notice is that bindings can be input or output. And output bindings are the most common. An output binding means that we make a request to the Dapr sidecar, and it forwards on that request to the external service. And an input binding means that the Dapr sidecar will make a call to our microservice letting us know whenever something interesting has happened on the external service. So we can be notified about events and they can be used to trigger endpoints on our microservices. You can see here that in addition to multiple generic bindings, there's also several bindings supporting services on specific cloud providers, like Azure or AWS. Many of the most popular external services that you might use when building microservices are already in this list. And given that Dapr is an open source project, this list continues to grow as members of the community contribute bindings for additional services. To help us understand how you'd use these bindings, let's quickly take a look at some component configurations.

Output Binding Example (SendGrid)

for our first example binding, let's choose an output binding. We're going to look at the SendGrid binding. SendGrid is a service that allows you to send emails, and here we can see the example component definition. Like all Dapr components, it's got a name and the type is bindings.twilio.sendgrid. The metadata shown here allows us to optionally configure some default values, such as the email sender and recipient addresses and subject, but the one required configuration property is your SendGrid API key, and this is of course a secret, so it

would be a good idea to set this value using a Dapr secret reference like we saw earlier in this course. How would we actually send an email using this binding? Well, we'd need to send a request to our Dapr sidecar, which would have a URL looking a bit like this calling the Dapr sidecar on localhost, using version 1 of the bindings building block, and talking to the binding named sendgrid. And then for the body of the request, this documentation gives us an example of the data that we need to pass to the output binding. The request payload contains a data property, which for this binding is the HTML body of the email that we want to send. And there is also some optional metadata we can add, and this example is showing setting the recipient of the email and the email subject, and although the documentation doesn't show it, we also need to specify an operation in this payload. As you can see here, the sendgrid binding only supports the create operation, which you used to create an email. And once again, the Dapr SDK can help us out a lot here by formatting the URL for us and correctly constructing the request payload with the data, metadata, and operation.

Interchanging Bindings

One really nice capability that many of the Dapr building blocks offer is the ability to interchange between different backing implementations. We saw this with the state store building block, where in local development we used a Redis instance running in Docker, and when we ran in production, we used Azure Blob storage for our state store. And the microservice using the Dapr state store building block doesn't need to know or care what the actual backing store is. The code remains exactly the same even when we switch component definitions. Now, does this apply to Dapr bindings as well? Well, actually, in the case of email bindings that we're going to be using in this module, it is a very similar story. We might bind to SendGrid for production and to the SMTP binding for local development, and the same message could be sent to the Dapr sidecar and the bindings would deal with the complexities of actually forwarding those emails on to the email sending services. However, it's not necessarily the case that all external services offer the same set of capabilities. The Dapr bindings for different services might expect different metadata in the messages that we send to them, so be careful of assuming that you can always trivially switch between bindings. The important thing to remember is that the benefits of Dapr bindings are the simplification of the code that we need to write to connect to those external services. All the complexities of connecting and the knowledge of how to follow best practices when you're talking to those services is contained within the binding, and so even when you might want to switch between binding to two services, but don't have exactly the same capabilities, your code can still benefit greatly from the simplicity that comes from using Dapr bindings. Let's look next at an example of a Dapr input binding.

Input Binding Example (Twitter)

Now let's look at an example of an input binding, and I've picked Twitter. The Twitter binding is actually both an output and an input binding. When you're using it as an output binding, you can use the binding to send tweets, but for an input binding you can configure search criteria that's looking for all new tweets that contain a specific phrase, and then it can notify you whenever someone posts a tweet that matches that phrase. And the documentation page that we're looking at here shows us the component definition format. There's a bunch of

Twitter keys and secrets that we need to provide in order to be able to connect. But let's look at the input binding part. All you do is you provide another metadata property called query, and the value is set to the term you're searching for. In this example, we're searching for all tweets that mention Dapr. And what happens is that the Dapr sidecar connects to Twitter using the credentials that you supplied and registers for a webhook callback from Twitter whenever a tweet containing this term is posted. And then the webhook will be received by your Dapr sidecar and then it will pass it on to your service by calling an endpoint that matches the name of your component, so if our component was called Twitter, then the Dapr sidecar would call an endpoint of /twitter. The format of the body of the request is defined by the binding itself. In this case, it's simply passing on the definitions of the tweets received from Twitter, which is actually a fairly complex structure that's defined here in the Twitter developer documentation. So we've seen that Dapr bindings can simplify the task of connecting to external services, as well as receiving notifications from external services that can trigger behavior on our microservices. It's important to remember though, just like with all Dapr building blocks, bindings are optional. Just because a Dapr binding is available, doesn't mean you have to use it. Sometimes, particularly if you're an advanced user, you may still prefer to directly use the SDK of the service that you're talking to, as the Dapr bindings won't always necessarily expose the full capabilities of the services that they are connecting to. Let's put Dapr bindings into practice by using them in the GloboTicket application.

Demo: Output Binding - Sending Email with SMTP

In this demo, we'll see how a Dapr output binding can be used to send emails. We're actually going to use the Dapr SMTP binding to implement this. We'll update the ordering microservice to send an email whenever tickets are purchased. First, let's take a look at the component definition we're going to use. Here in email.yml, I'm defining a component whose type is bindings.smtp, and the name I'm giving to my component is sendmail, and we need to supply a few details. First, what's the host name of the SMTP server that we want to connect to, which in our case is localhost, and the port it's listening on, which is 1025. We need to supply the username and password, which normally would be set up as secrets, but because this is the component definition for local development and we're not actually connecting to a real email sending service, in this case, hardcoded values are acceptable. Notice the scopes section at the bottom. Sometimes in Dapr, we want our components to be available to all our microservices, but in this case, we only want the ordering microservice to access this component, so I've set the scope to be simply ordering. For local development, it would be really useful to have an SMTP server we can use, and I've chosen to use an open source component called MailDev. This combines an SMTP server and a web interface that allows us to view the emails that have been sent, and this is really convenient for local development. We can start it locally by using the docker run command to start the maildev/maildev image, and mapping ports 80 and 25 to 3 port numbers. In this example, I'm using 1080 and 1025. Now let's look at the code to actually send an email using the Dapr output binding. I've put all the code that sends an email here in the EmailSender class. As you can see, in the constructor, we're taking a DaprClient instance as we want to make use of the Dapr C# SDK to simplify sending the email. Then in the SendEmailForOrder method, we construct a dictionary containing the metadata for this email which includes the sender and recipient email addresses and the email subject line. You might notice that these are the

exact same metadata names for the sendgrid binding, which is nice. So in this particular example, we could swap out an SMTP binding component definition for a sendgrid component binding definition, and there'd be no required code changes at all. And the body contains the HTML formatted message content. Finally, here we're sending the request to our Dapr sidecar for it to pass it on to the SMTP server, and we're using the InvokeBindingAsync method on the Dapr client. The first parameter is the binding name which we called sendmail and then the operation which is simply create. Then we pass in the data, which for this binding is the email body, and finally, any additional metadata, which is our dictionary in C#. And so the Dapr client has saved us from having to construct the URL and HttpRequest payload ourselves. If we now take a look in the OrderController class which contains the endpoint that receives a new order from the Dapr pubsub component, we're now making a call to the SendEmailForOrder method on the emailSender. Okay, so now I've shown you all the code that will send an email for an order, let's test it out.

Demo: Testing Sending an Email

In this demo, we'll put the code that we just looked at for the output binding into action by sending an email and then checking that our maildev Docker container picks it up. And I'll also show you how I've configured the same maildev service in Kubernetes. I've already started the maildev container locally using the command that we saw earlier. And so we need to start all three microservices, which we'll do in the same way that we've done before, with the start-self-hosted PowerShell script in each microservice folder. Once they've started up, let's visit the GloboTicket home page, and we'll add something into our basket and we'll complete the whole checkout process. And once we've got to the point where we've submitted the order, a message is going to be sent to the ordering microservice using the pub/sub building block, and that's going to trigger our output binding that should have sent an email. So let's check it worked by visiting our MailDev web server. We exposed it on localhost port 1080, and here we can see that the email has been passed on to our SMTP server by the Dapr binding, and it's got the correct subject, body, and recipient. By the way, if you want to try this out in Kubernetes, you can also install the maildev container onto a Kubernetes cluster. And if we take a look in the aks-deploy script that we've gone through in some detail earlier in this course, you can see here this kubectl create deployment command that deploys the maildev container image and the kubectl expose command that makes ports 25 and 80 available. And the binding component definition for maildev that we're using in Kubernetes, you can see here in the email.yaml file in the deploy folder, and it's exactly the same as the one we used in self-hosted mode, except now that the host is maildev rather than localhost. Finally, back in the aks-deploy script, here's where we deploy that email.yaml component into our Kubernetes cluster. And I've also included a helpful kubectl port-forward command that you can use if you want to access the maildev web server on a Kubernetes cluster from your local development machine. You just run the kubectl port-forward command, mapping the maildev service onto a local port, I've chosen 8081, and then if you visit localhost port 8081, you'll see the same maildev UI that we just saw. Let's quickly run that port-forwarding command, and if I visit port 8081, then I can see the MailDev instance that's running on my Kubernetes cluster in Azure, and you can see that there's a few email notifications in here from tests that I've done previously. Of course, in production you wouldn't use something like maildev. You'd either point this SMTP binding at a real SMTP server that was going to send emails, or you'd make use of an email sending service

like the SendGrid output binding that we looked at earlier. And the great thing is that we can change the binding destination with no code changes whatsoever. In this demo, our code is simply talking to a binding called sendmail, but it doesn't need to know or care what the actual email sending service behind the scenes is.

Demo: Cron Input Binding

For our input binding demo, we're going to use the cron binding, and this allows us to run scheduled tasks. Now this is a bit of a special case, because this binding isn't connecting to any specific external service. However, scheduled tasks are an extremely common requirement in microservice applications, and so the fact that Dapr provides a way for us to implement them with bindings is potentially a big time saver. We're going to create a very simple scheduled task in the catalog microservice as an example. Let's imagine that GloboTicket want to randomly pick an event to go on special offer every hour in order to encourage customers to keep logging onto the website to see what the current special offer is. Let's start by looking at the component definition file for our input binding. It's pretty straightforward. Our component is of type bindings.cron and it has a name of scheduled. The name is important, because the Dapr sidecar is going to call an endpoint with this name on our microservice every time that the scheduler fires. We obviously need to tell it how often to fire, and the cron binding supports regular cron job expressions, which are very flexible. And there's also a simplified syntax available that I'm taking advantage of here. I've set mine up to run every 5 minutes, as I don't want to wait ages while I'm testing this. Earlier in this module, we talked about the scopes section of the component definition file, which lists the microservices that this component applies to. It's only the catalog microservice that has an endpoint for the scheduler to call, so I've just put catalog here. Remember that when we start our microservices in self-hosted mode, we're pointing at this components folder to say that this is where all of the component definitions for our microservices are, and so even though all three of our microservice sidecars will see this YAML file because it's in the folder, only the catalog microservice will actually use it because the application name of the catalog service is listed in the scopes here. Now let's look at the endpoint that's going to be called when the scheduler fires. I've created a new ASP.NET controller in the EventCatalog microservice called ScheduledController. It's very simple to set this up. The Dapr sidecar will simply make a post request to the /scheduled endpoint, and so I've said that this controller is listening on the base path of scheduled, and this OnSchedule method will accept HttpPosts to that endpoint. Inside this method, I can then do whatever I want to on the schedule, such as triggering the logic to put one of our products on special offer, as you can see in this example, and I'm also writing a message to the log so that we can easily check that this endpoint is getting called correctly. We're ready to test this, and we don't even need all of our microservices running. I'll just start the catalog microservice, and I'll wait a couple of minutes for the scheduler to fire. And sure enough, here in the logs we can see the message that shows our scheduled endpoint has been called. And if I start the frontend service and visit the home page, we can test that it's all working by waiting a few minutes, and now when I refresh the page, we should see some prices change as a different event has been placed on special offer. One nice thing about the cron binding is that to run it on Kubernetes, we don't need to make any changes whatsoever to our component definition file. So here in the deploy folder, which defines the components we're using on AKS, I've got exactly the same cron.yaml file.

Module Summary

In this module, we looked at Dapr bindings, which can greatly simplify the code required to communicate with external services. We saw that there's two types of binding, input and output bindings. Input bindings allow events that happen in external services to trigger requests into our microservices. And we looked at examples of Twitter search results and scheduled tasks. And, output bindings allow us to send data to external services, and the examples we looked at were sending emails with SendGrid and with an SMTP server. And, these are only a small selection of examples. Dapr comes with many bindings to popular services, and because it's an open source project that accepts community contributions, that list is continuing to grow. Right, we're almost at the end of this course now, but there is one more building block that I want to show you and that's how Dapr can help us to monitor what's happening in production with the observability building block.

Monitoring in Production with Observability

Module Introduction

Hi. Mark Heath here, and in this module, we're going to learn how the Dapr observability building block can help us understand exactly what's going on in our microservices application. One of the biggest challenges associated with operating a microservices applications in production is that you need to be able to understand the big picture of the overall health and performance of the system. You need to be able to both detect problems quickly so that you can fix them and to troubleshoot issues to understand afterwards what went wrong. And once again, Dapr is able to help us by providing a variety of capabilities to give us greater insight into the health and performance of our overall application. In this module, we'll start off by taking a look at several helpful observability capabilities provided by Dapr, including distributed tracing, the Dapr dashboard, health checks, metrics, and sidecar logging, and then we'll see how to configure our GloboTicket application to collect tracing information that we can view with Zipkin. And we'll wrap up by taking a look at some of the additional capabilities of Dapr that you might want to explore, as well as learning about some resources that will help you in your journey to adopt Dapr for your own applications. So let's get started.

Distributed Tracing

We'll start off by looking at the distributed tracing capability of Dapr. Tracing allows us to understand the way that our microservices are interacting with each other, and it also gives us a way to correlate the logs from different microservices in order to piece together what happened when we're trying to understand an operation that involves more than one microservice. By default, Dapr uses the Zipkin protocol for distributed traces and metrics collection. The Zipkin protocol is widely supported, meaning that you can choose from multiple different back ends, including Zipkin, Stackdriver, and New Relic. Dapr also supports the OpenTelemetry protocol, allowing you to export your traces to many other back ends, including Azure Monitor, Datadog, and SignalFx. And the way it works is that the Dapr sidecar contains tracing middleware. This means that all communications via the Dapr side car will

automatically be eligible for tracing. That means if you're using the Dapr building blocks that we've looked at in this course, you just need to turn on tracing, and you'll get a very comprehensive picture of what's going on in your application with minimal effort. Obviously, if you're connecting to other services without using Dapr, then it's up to you to ensure that you configure tracing for those communications. The way you set up tracing in Dapr is via a configuration file similar to the component definition files that we've looked at before. Here we can see an example of the configuration that I'm using in Kubernetes. This is a Configuration definition, and I've given it the name appconfig. The key part here is in the spec section where we've set up tracing with a sample rate of 1. This is the probability that any given tracing span will be sampled. One is the maximum value, so this means capture all traces, which is good for development, but might result in too large a volume of tracing data in production, so you might set this 0.01 sample only 1%. And of course, we could set it to 0 to completely disabled tracing. We've also configured it to use the Zipkin protocol and told it where the endpoint is that it needs to send the tracing information to. In this case, it's a Zipkin container running on the kubernetes cluster.

W3C Trace Context

As well as sampling tracing information, Dapr is able to propagate W3C tracing context information. What does that mean? Well, let's consider what happens when we load the GloboTicket home page. First, there's a request to the frontend microservice. That in turn makes a request to the catalog microservice to fetch the list of events, and the catalog microservice uses the Dapr sidecar to fetch a connection string secret and then talks to a database. And the frontend microservice also uses the Dapr state store building block to find out if there are any items in your shopping basket. As you can see, even in the context of simply loading a single web page, several different services and external components are involved. And that picture could get even more complicated in the future as additional features are added to GloboTicket. And this means that if something goes wrong during the loading of that page, we need to piece together all of the logs from the different components to understand what happened. And what the W3C trace context does is it allows us to pass around a unique identifier that ties together all of these operations. Dapr is able to both generate trace context and to propagate already existing trace context using the industry standard W3C trace context specification. And this means that any other services you interact with that can interoperate with this standard can also tie their logs into the same trace context. It also means that in your own logging code, you can access the trace context and ensure that it's recorded alongside each of your log messages. And the way this capability works is with HTTP headers. The traceparent header contains a unique identifier for the incoming request and can be used to tie information from all systems together. And there's also a tracestate header that can propagate additional state information between services. And if you're interested in learning more about this capability, there's an article here on the Dapr documentation website that includes code samples in various languages to show you how you can access these headers, as well as how you can create your own trace context.

Health Checks

In a microservices architecture, it's generally considered best practice for your microservice to expose a health check endpoint, and this can be used to report whether your microservice is running correctly or not. For example, has it started up, and is it able to successfully communicate with its downstream dependencies? And one of the key benefits of having health check endpoints is that container orchestrators like Kubernetes can use them to know whether a newly started microservice is ready to start receiving traffic, and they can also use them to detect when your microservice has a problem and needs to be restarted. Of course, when you're running Dapr on Kubernetes, for every microservice, there are two containers, your microservice itself and the Dapr sidecar, and they both run inside a Kubernetes Pod. It's your responsibility to create the health check endpoint on your microservice if you'd like Kubernetes to monitor it, and the Dapr sidecar comes with a built-in health check endpoint. This can be accessed at v1.0/healthz, and this allows Kubernetes to determine the overall health of the Pod by calling the health check endpoints for the sidecar and for your microservice. And Kubernetes will only consider a Pod to be healthy if all of the health check endpoints for the containers running inside it report that they're healthy.

Dapr Metrics

Another observability benefit of Dapr is that it exposes a Prometheus metrics endpoint. This allows you to gain better insight into how Dapr is behaving and create alerts that warn you if there's a problem in production. This endpoint is configured by default and has a default port number of 9090. And if we look here on GitHub, we can see a list of the metrics that Dapr is able to track for us. These include counting various failure conditions, such as component initialization problems or MTLS authentication failures. And it also includes several performance-related metrics, such as latencies for HTTP requests, and so this can be a helpful way of detecting performance issues in production.

Dapr Sidecar Logging

Another way that Dapr helps us with observability is that each sidecar emits logging messages. You've already seen these if you've run Dapr in self-hosted mode. For the most part, you can safely ignore these messages. They're simply providing low-level information about what the Dapr sidecar is doing. But one situation in which I find the Dapr sidecar log messages very helpful is when you have problems with component configuration. Maybe you made a mistake in the YAML configuration for a component, or perhaps the resource you're trying to connect to is unavailable. And these are the types of error that the Dapr sidecar is able to report for you. So if you're encountering issues, whether running in self-hosted mode or in Kubernetes, make sure that you also examine the Dapr logs, as well as the logs from your own application. The logs from Dapr can be configured to be logged in JSON format, and you can direct them to log collectors such as Fluentd or Elasticsearch. And once again, the Dapr documentation is the place to go to find out how to configure this for the systems that you want to integrate with.

Demo: Dapr Dashboard

Another great feature that we can use to help us with observability is the Dapr dashboard. This is a web-based user interface for Dapr, allowing you to see useful information about the Dapr components, configuration, and applications that you're running. The Dapr dashboards, like the rest of Dapr, is open source, and its repository can be found here on GitHub, which is a good place to start learning more about its capabilities and make any feature requests. It's very easy to get started with, and you can see here on the GitHub page the instructions for launching the dashboard. We've already got the Dapr CLI installed, so all we need to do is run the dapr dashboard command. If you want to run the dashboard against Kubernetes, you also need to pass the -k switch and, optionally, the -n switch to specify the Kubernetes namespace you installed Dapr in. Let's take a look at the Dapr dashboard on a Kubernetes installation of GloboTicket. I'll launch it with dapr dashboard -k, and it shows me the URL it's running on, which is localhost 8080. And, of course, what's happening here is that I'm essentially port forwarding from my local machine onto the Kubernetes cluster. Here we can see it's showing me the version of the Dapr control plane that I have installed and that my Dapr installation is healthy. It's also showing me the Dapr applications that I have running. These are our three microservices, the catalog, frontend, and ordering microservices. I can navigate into one of these Dapr applications, such as the catalog microservice, and see some useful information about it, such as its application ID and Dapr HTTP port number. And you can see that there are several other views, such as Metadata, which is showing us the annotations on this application. The Configuration tab shows us the full Kubernetes Pod configuration, which includes lots of useful information, such as all of the environment variables that have been configured for our containers. But perhaps most useful of all for troubleshooting is the Logs tab. Here we can access the logs from both the sidecar, which is this container called daprd, and our microservice itself, which we can access by changing the container to catalog. And we can even get it to highlight a keyword that we're looking for in the logs. So I can look for the word scheduled, for example. Let's go back to the home page and follow this More Information link. This takes us to a view that shows us the Dapr services that are running in our cluster and allows us to check what version they are and ensure that they're all healthy. Another view we can access is the Dapr Components view. This shows us all of the components that we've configured. And you can see here that we've got an Azure Service Bus pub/sub component, a cron binding for our scheduled task, an SMTP binding to send email, and an Azure Blob storage binding for our state store. And I can drill into one of these bindings to see additional information, including the YAML component configuration. Finally, there's a view of Dapr Configurations. In here, we can see not only the default daprsystem configuration, but also the appconfig configuration file that I showed you earlier. Here we can see that all three of our microservices have been configured to use this appconfig configuration. And if we look at the YAML for the component configuration, we can see that this includes setting up the Zipkin tracing, which is what we're going to be using in our next demo. As you can see, the Dapr dashboard is a really powerful way of getting insight into the overall health of both Dapr and your own microservices.

Demo: Distributed Tracing with Zipkin

In this demo, we'll see how the distributed tracing in Dapr works by configuring tracing and viewing some of those traces in Zipkin. Let's start off by looking at how I've configured tracing

with Dapr. Here, we're looking at my appconfig file, which we've discussed already. And I explained that the samplingRate of 1 means that we're going to capture all traces, and we're using the Zipkin protocol and sending these traces to the zipkin container that's running on our cluster. And here in the aks-deploy.ps script, which has got all of the commands that I used to deploy GloboTicket into Azure, you can see here is the bit that creates our Zipkin container. We're using kubectl create deployment to create a Kubernetes deployment running the openzipkin/zipkin container image, and then we're using kubectl expose deployment to make that deployment available locally in the cluster on port 9411. And then this call to kubectl apply is deploying the configuration file that we just looked at. I've also included a helpful command here. In order to access the Zipkin instance that's running on our Kubernetes cluster, I can use kubectl port-forward to make the service we exposed available on localhost port 9412. And by the way, the reason I didn't choose 9411 is because I've already got another local Zipkin instance running on my machine on port 9411, which was created when I installed Dapr in self-hosted mode. So let me run this command to set up the port forwarding to Kubernetes. And now I'll visit the Zipkin instance in my Kubernetes cluster by visiting localhost port 9412. And here we can see the Zipkin user interface that lets us search the traces that have been captured. I'll click RUN QUERY, and you can see that it shows me a list of recent traces. And we can also see the duration of those traces, which is really helpful to quickly identify where there might be some performance issues. Let's take a look inside this top one by clicking SHOW. And here, it gives us a breakdown of all of the individual requests involved in that trace. We can see that we access the state store to fetch a shopping basket, and we published an event, which the ordering microservice picked up, and we called invokebinding to send an email, and it breaks it down to show us the time taken by each step in the trace. We can also look in more detail at individual requests in the trace such as the call here to fetch the basket from the state store. Let's also take a look in this Dependencies view. If I perform a search for some recent traces, we can see that it draws a graph showing the Dapr applications, which are our three microservices. And then these animated dots moving across the lines show us the communication between those services. As we can see, we're making a lot of requests from the front end to the catalog microservice and occasionally one to the ordering microservice, which is what we'd expect.

Additional Dapr Capabilities

We're close to the end of this course now, and we've explored several of the most important Dapr building blocks. The ones we've used in our GloboTicket demo application are state management, service invocation, pub/sub messaging, secrets, input and output bindings, and observability. And that's a pretty powerful set of capabilities that will be useful to anyone who's building a microservices application. But it's not all that Dapr has to offer. In particular, I want to highlight three capabilities that we've not used in this course, but that you might want to explore and consider whether they're a good fit for your application. The first is actor support. Dapr comes with a very powerful building block that supports using the virtual actor architectural pattern. With the actor pattern, you write code in self-contained units called actors, which receive and process messages one at a time. And actors can send messages to other actors, as well as create new actors. And it's possible to support a very large number of actors simultaneously. And although perhaps it's not a very widely used pattern, there are some types of problems that it's ideal for. And the great thing with Dapr is that you have

this building block available to you, and so if at any time you identify a problem as being a good fit for the actor pattern, you can easily make use of it. The second is the configuration building block. The reason we didn't focus on this in this course is at the time of recording it was still in alpha and subject to change, but it helps with another very common problem with microservices, which is having somewhere to store and manage all of the configuration settings for your microservices and getting notified when those settings are modified. Finally, Dapr offers some middleware components that can run as a pipeline inside your Dapr sidecar, transforming requests and responses as they go through. The tracing functionality that we looked at earlier is an example of some middleware that's enabled by default. One particularly noteworthy middleware component is the OAuth 2 middleware, which enables the OAuth 2 authorization code flow on a web API. Again, this has the potential to take some complexity out of your microservice code and move it into the Dapr sidecar. And there's also a rate-limiting middleware component, which is another good example of the kind of added value that middleware can offer.

Dapr Community Resources

Before we wrap up this module and the course, I want to point you to some of the key community resources that you'll want to be aware of as you continue your journey of evaluating Dapr and incorporating it into your own microservices. And some of these we've seen already. First of all, the official Dapr documentation site is your best starting point for learning about building blocks and component configurations. And the Dapr GitHub repository is a great place to report bugs and make feature requests, as well as learn about the roadmap for future versions of Dapr. Another great thing about Dapr is that approximately every two weeks they host a Dapr community call that you can watch on YouTube. This gives you updates on the features that they're currently working on, and there's an opportunity for you to ask your own questions. Dapr also have a community on Discord which is a very active community with discussion channels for all of the building blocks, as well as individual programming language support. It's a great place to ask for help with issues you're facing, and I really like the show and tell channel where people share the cool things that they've built with Dapr. It's a great way to learn about things that can be achieved using Dapr. And finally, Microsoft have made a free Dapr for .NET Developers e-book available, which is an excellent resource that works you through building a traffic control sample application and uses many of the Dapr building blocks.

Module Summary

In this module, we learned about several of the powerful features that Dapr offers for observability. We saw how tracing can be captured using the Zipkin protocol and that Dapr supports W3C tracing headers. We also learned about additional observability metrics, sidecar logging, health check endpoints, and the Dapr dashboard. And we also explored some of the additional Dapr capabilities that you might wish to explore after watching this course, such as the actors and configuration building blocks, as well as middleware for things like OAuth and rate limiting. And I pointed you in the direction of some additional learning resources that will help you to keep up to date and get help as you adopt Dapr, including the community call videos and the Discord channel. And this is also a good time to remind you that the sample

code for this course is available here on GitHub The main branch contains the complete GloboTicket application that uses Dapr and includes scripts that help you run it locally or in Azure. And although we didn't use it in this course, there are also some Docker Compose files, which demonstrate how you could run it locally using Docker Compose. And if we look at the branches, there's a before branch, which has the starting point of the GloboTicket without Dapr, and I've also got some branches in there for specific versions of Dapr, and I hope to be able to create more of these as new versions of Dapr are released, although I found so far that the changes required to move between Dapr versions are very minimal. Thanks for watching this course, I hope you found it helpful, and feel free to provide feedback on the Pluralsight discussion board or on the GloboTicket Dapr GitHub repository.