Course Overview
Course Overview

Hi. My name's Mark Heath, and welcome to my course, Microservices Fundamentals. I work as a software architect at NICE Systems where I'm currently helping to create Azure-based digital evidence management systems for the police. Microservices is a style of architecture where you create autonomous, independently deployable services that collaborate together to create a software application. Microservice architectures are popular because they allow us to build applications that scale, perform well, and enable us to adapt quickly to changing business requirements. In this course, we're going to learn about the key principles and practices that will enable you to be successful with microservices. And we'll be seeing how these principles are applied by exploring an example e-commerce microservices architecture. We'll start off by understanding what microservices are, along with the problems they solve and the challenges associated with them. Then we'll look at how we can architect microservices, making good decisions about service boundaries and data ownership. We'll move on to consider the practical challenges of building microservices, including how we can ensure that developers are as productive as possible, and we'll look at some options for how microservices can communicate with each other reliably. We'll also see how we can apply the defense in depth principle to secure our microservices and how to automate their deployment and monitor them in production. By the end of this course, you'll be ready to design and build your own microservices applications and identify which technologies and practices are a good fit in your context. The demo application that we'll be looking at uses Docker, and although you don't need to have Docker installed, if you do, you'll be able to follow along and try out the demo application for yourself. So I hope you'll join me on this journey to learn about microservices with the Microservices Fundamentals course, at Pluralsight.

Introducing Microservices

Course Introduction

When you're building a modern software application, you want it to perform well, to be resilient to errors, to have a flexible architecture that can adapt to new requirements, and to keep its data secure. Microservices is an architectural style that's grown rapidly in popularity over the last decade, and it has the potential to help us achieve these goals in modern distributed software applications. I'm Mark Heath, and in this course, we're going to explore what a microservices architecture looks like in practice and the benefits that it can bring. In this module, we'll start by making sure that we understand what the concept of a microservice is and what benefits it brings over a more traditional approach, often referred to as a monolith. We'll learn about the limitations of a monolithic architecture and see how microservices offer solutions to some of these problems. But we'll also learn about some of the challenges that adopting a microservices architecture can bring. Although microservices do solve many problems, they also introduce new ones. And so you do need to be aware of this before embracing microservices for your own applications. I'll also introduce you to a sample microservices application, which will serve as an example throughout this course, which illustrates many of the techniques and principles that we'll be learning about. As we move through the rest of this course, we'll look at architecting microservices. How do we decide the best way to divide overall system functionality into

individual microservices? Then we'll look at some of the practicalities of actually building microservices. How can we ensure that our team of developers is successful and productive as they implement a microservice architecture, including how to effectively debug and test those microservices? We'll also explore in a bit more detail the options for how microservices can communicate with each other. In particular, we'll look at the approaches of synchronous and asynchronous communications and how these choices relate to overall system reliability and performance. Of course, in an era of cloud-deployed applications and data breaches, it's essential that we secure our microservices and the data that they operate on. And we'll look at some key principles and techniques that we can apply to protect our system from attackers. And we'll finish by briefly looking at delivering microservices. How can we deploy them into production and monitor them effectively? Right, I've already said microservices lots of times, but I haven't yet explained what they are. So let's talk about that next.

## What Are Microservices?

What do we mean by microservices? Microservices is an architectural style where autonomous, independently deployable services collaborate together to form a broader software application or system. And that's it. Simple, right? You might even have worked on systems that take this approach without calling it a microservices architecture. Of course, the name microservices also suggests that these services should be small, but there isn't an official size limit. Some teams try to keep them extremely small, maybe just a couple of hundred lines of code but many of the benefits of microservices still apply even if they're a bit larger than that. But why? Why do we need microservices? Let's look at some of the problems that microservices are designed to address.

## The Problem of Monoliths

Microservice architectures are often contrasted with monoliths. A monolith is a software application that typically has all of its code in a single code base and which all developers collaborate together on that same source code repository. The build artifact is usually a single executable or process that runs on a single host server or virtual machine and persists all of its data into a single database. And the development environment typically uses a single consistent technology, such as what programming language or SDK you're using throughout the code base. And one of the major benefits of this approach is simplicity. All the code is in one place, so it's easy to find things. There's only one thing that you have to build and run, so it's straightforward for a developer to work on. And when you come to deploy it to production, well, there's just one application to update. And there's nothing inherently wrong with this approach, and if it's working well for you, that's probably a sign that you don't need microservices. Or perhaps I should say you don't need microservices yet. The problem with monoliths is one of scale. The monolith model works well for single developers or small teams of maybe two to three developers who are working together for just a few months maybe to build a small website that handles a modest number of visitors. But what happens when our projects grow much larger? What if we've got 20 to 30 developers, and they're working for a period of several years building a system that will have thousands of users and store massive amounts of data? When we get into this kind of situation, the monolith approach becomes a big problem. For one thing, as a code base grows larger, there's a tendency for it to become

more difficult to maintain due to growing complexity and the accumulation of technical debt. And even if you've made a concerted effort to keep your monolith modularized, often you'll find that these modules end up becoming very entangled and interdependent. When it comes time to deploy your monolith, even a single line code change requires the entire application to be deployed, which is risky and usually involves a period of downtime, something that's becoming less acceptable in the modern era of cloud services that are expected to maintain high availability. Scaling a monolith to meet demands of increased users or large amounts of data is also very difficult. Unless great care has been taken to make your monolith stateless, you probably can't scale it out horizontally. That's where you add additional servers. And so your only option is to scale vertically where you provision much more powerful and expensive servers. And monoliths require the entire application to be scaled out together rather than just calling the individual components that require additional processing power. Finally, with the monolith, you can very easily find yourself wedded to legacy technology. Whatever tech stack you built the original version with is going to be very hard to get away from as you have to upgrade the entire application to move to a newer framework. And this reduces your agility to adopt newer patterns and practices or to take advantage of innovation such as new tools and services that would benefit your application. Now perhaps you're thinking, Well, my application isn't a monolith. It's made up of several services that can run on different hosts and talk to each other over the network. However, it's very important to understand that just because you have an architecture based on services doesn't mean you're using microservices. It's possible to create a system where you have several services, but they all access data in a shared database, and they're all tightly coupled to each other in such a way that you have to deploy them all together, and any change you want to make to the system requires modifications to multiple services. Systems like this are sometimes referred to as distributed monoliths, and these things are big trouble. They combine all the downsides of monoliths with all the challenges of microservices and offer very few of the benefits of either approach. So it is important to have a clear understanding of microservices before diving into creating them or you could end up in a world of pain. Let's see next how a microservice architecture can offer a significant improvement over monoliths.

## The Benefits of Microservices

We've looked at some of the drawbacks of monoliths. So let's look now at how microservices can help us. By breaking our application into smaller pieces, each microservice can be owned by a small development team and is much easier to understand and work on. A good rule of thumb for the size of a microservice is that it's small enough to be thrown away and rewritten if necessary. Microservices allow us the freedom to adopt new technologies without needing to upgrade everything in one go and the flexibility to choose the right tool for the job. For example, one microservice might use a relational database to store its data while another uses a document database. One microservice might be written in a functional programming language while another might use an object-oriented approach. Of course, there are still benefits in using standardized approaches where possible, but there're no hard requirements. Individual microservice development teams have the freedom to use the most appropriate technology. When we create microservices, their loose coupling means that we can deploy them individually, and this is a huge benefit since it's much lower risk than upgrading everything in the entire system in one go, and it also helps us to achieve zero

downtime deployments. And that's because while we're upgrading an individual microservice, it's quite possible for the rest of the system to carry on running. Microservice architectures enable teams to deploy with much greater frequency. Rather than waiting months between upgrades, as is often common with monoliths, changes to individual microservices can be pushed to production as soon as they're ready, and it's even possible to deploy microservices multiple times a day. Microservices give us much more control over scalability. We can scale out each microservice individually, which is much more cost effective than scaling a monolith. Finally, microservices make us more agile. We can adapt more rapidly to changing business requirements. By keeping our microservices lightweight and decoupled, they're more easily adaptable to use in new scenarios. And these are just some of the reasons why microservices is becoming popular, particularly for very large cloud-deployed systems. But, unfortunately, microservices do come with some challenges of their own, so let's look at those next.

## The Challenges of Microservices

If microservices are so great, then why don't we just use them for everything? Well, it's not a free lunch. There are some things that can be more challenging with a microservices architecture. And it's important that you understand this in order to decide whether microservices is a good fit for your application and also so that you can be prepared to deal with these challenges as they arise. One of the first challenges you'll face is how can you ensure developers are productive working on the system? Do they have to download the code for every single microservice and run it individually and configure them all to talk to each other? That could be very error prone and time consuming. You also need to have a way of working that enables developers to easily work on an individual microservice in isolation, as well as testing their microservice in the context of the whole application. And we'll be seeing some ways to achieve this later on in this course. If we break an application up into dozens of microservices, then there's potential for the interactions between those microservices to get very complex making it hard to understand the behavior of the system as a whole, and it can also lead to performance issues. If you don't take care, you can find that a single operation requires lots of inefficient, verbose communications between many microservices. And while deploying an individual microservice is a smaller and safer task than deploying a monolith, when you have many microservices, automated deployment becomes critical because you're going to have many more deployments to do. Finally, if there's a problem in production, we don't want to have to connect individually to every single microservice to examine its logs and work out which one is the cause of our troubles. So it's essential that you have a great monitoring solution enabling you to view logs and telemetry for all of your microservices in a centralized place. The good news is that there's a growing body of tools and practices available to help you succeed with microservices and overcome these challenges and we'll be learning about many of these as we progress through this course.

## Introducing the Demo Application

Now, I don't want this course to be purely theoretical, so we're going to be referring to a real microservices application as we go through this course, which will illustrate the principles that

we're learning about. And, rather than me spending a few months creating my own microservices application, I've decided to use a preexisting reference microservices implementation called eShopOnContainers, which is open source and available here on GitHub. And there's a few reasons why I picked this example to use in this course. First of all, ecommerce is a well-known problem domain. We've all used a shopping website, and so it should be fairly straightforward for you to understand what the responsibilities of the various microservices such as the basket, catalog, and ordering services are. But it's also a non-trivial example. As you can see from this system architecture diagram, there are three client applications, two websites and a mobile app, and six microservices, making use of a few different database technologies and communicating via an event bus. And this sample also makes use of a pattern known as an API Gateway, or Backend for Frontend, that we'll discuss later on in this course. It's also being actively maintained and it's regularly updated with newer technologies and practices. In fact, it's had 96 different contributors at the time of recording, making it representative of what it's like to work on a real-world microservice system where you have many developers collaborating together. It's also containerized, which means it's really easy for you to run the code on your local development machine if you want to try it out. And there are detailed instructions provided on GitHub explaining how to run it locally, whether you're Windows or Mac-based. Finally, it's very well documented, including a couple of excellent free ebooks that you can download to help you fully understand the architecture, and they give the rationale behind many of the decisions that have been made in this project. And that's important because even if you choose microservices, you're still going to have a lot of choices ahead of you. Microservices don't dictate what programming language you use, what type of database you'll use, whether you communicate synchronously or asynchronously or what authentication mechanism you're going to use, and that means you're completely free to pick the ones that you're most familiar with and meet your needs. And we'll be discussing some of the factors that will help you make these decisions as we progress through this course. By the way, don't worry too much about the specific technology choices in the sample eShopOnContainers application. Because it's a reference application produced by Microsoft, it uses the .NET Framework and has scripts to help you deploy the application to Azure. But it doesn't matter if you're not a .NET developer or you use a different cloud other than Azure, we're going to be focusing in this course much more on the overall architecture rather than on the specific technology choices. So, you'll have no problem following along if you prefer to write your code in Java or Python or you like to host your applications in AWS or on Google Cloud Platform. Let's wrap up what we've learned so far. In this first module, we learned about what a microservices architecture is and how it differs from a monolith. We've seen some of the benefits, as well as challenges that microservices can bring. And in our next module, we're going to start looking at how we can architect a microservices solution.

## Architecting Microservices

### Module Introduction

Whether you're planning to migrate an existing application to microservices or adopt a microservices architecture from the start of a brand new project, there are some very important architectural decisions that you need to make up front. My name is Mark Heath,

and in this module, we'll discuss some of the key architectural principles for how we define a microservice, and its components. We'll begin by discussing the fact that you don't have to start with microservices. It's possible to evolve an existing application towards them. We'll look in a bit more detail at some of the key definitions of a microservice, including the fact that a microservice should be autonomous, which means it owns its own data. And a microservice should be independently deployable, which means it needs to have a clearly defined and backwards-compatible public interface. Finally, we'll discuss the tricky challenge of how we can identify suitable service boundaries for our microservices. We'll look at some guidelines for breaking applications into microservices, including the idea of identifying bounded contexts.

## Evolving Towards Microservices

It's quite possible that you already have an existing application with a monolithic architecture. Or maybe you've even got what we described in the previous module as a distributed monolith, where you've got several services that are tightly coupled together and not independently deployable. The good news is that it's perfectly possible to evolve towards a microservices architecture. You can augment a monolith with microservices where every time you add a new capability, you create a new microservice. And you can decompose a monolith into microservices where you identify existing capabilities that should be extracted out into their own independent microservices. In fact, some people argue that you should avoid starting a brand new project with a microservices architecture. The benefits of microservices are not typically seen in small, new projects. And it can be very hard to get your service boundaries right. So it might be better to allow a system to grow a bit until it becomes apparent what an appropriate segregation of responsibilities into microservices would be. Now, of course, when you are using microservices, at some point you're going to need to identify the responsibilities of a microservice and define what its public interface should be. So let's look next at some of the key principles of microservices that help us make these sorts of decisions.

## Microservices Own Their Own Data

Earlier in this course, we said that microservices are autonomous and independently deployable. And a key part of achieving these goals is ensuring that a microservice owns its own data. In other words, we're not allowed to have this sort of situation where multiple microservices both read or write from the same data store. Instead, each microservice should have its own data store, and any other microservice wishing to access needs to do so via the public API of the microservice that owns it. Now there are some obvious trade-offs here. By splitting our data out, it means that we can no longer perform database joins across the data that's owned by two different microservices. Instead, we're going to have to make separate calls to each database. And we can no longer update two tables owned by different microservices within a single database transaction. Instead, we'd either have to use distributed transactions, which are very complex to implement, or, more commonly, design our system to work in what's called an eventually consistent manner where we may have to wait a while for the overall state of the data to be fully consistent. What this means in practice is that when a single business operation requires updates to more than one

data store, there's going to be a short window of time when the change has been made in one data store but not the others. And so you need to develop your application in a way that is able to cope with this temporary inconsistency. Now there are ways to mitigate these issues. If we define our service boundaries well, we can minimize the need to aggregate data from multiple microservices to perform a single operation. Another approach is that one microservice might hold its own cache of a subset of the data that's owned by another microservice. An example of this might be that rather than having to constantly talk to a user profile service to get the name or email address of a user, we could maintain our own cached copies of this data. And this approach greatly improves performance because it removes the need for lots of chatty network communication and improves availability since now our microservice is able to do its job even when the user profile service is unavailable. It's important that when you're considering how to break your application into microservices, that you identify natural seams in the database schema, and this will help you avoid the performance penalties of having to involve multiple microservices in a single business operation.

EShopOnContainers Architecture

To illustrate how microservices should own their own data, let's take a look at the eShopOnContainers sample application that we introduced in the last module. We can see that the Catalog service uses a SQL Server database. The Ordering service also uses SQL Server, but it's a different database. Neither of these services can see or access each other's data by going directly to the database. The Basket service uses a Redis cache, and this is a more appropriate choice for short-lived data. We can also see that document databases such as Cosmos DB or Mongo have been chosen by some of the other microservices in this application. And so this architecture illustrates two key microservice characteristics. First, each microservice in eShopOnContainers owns its own data. And, second, each microservice is free to use the most appropriate database technology for the type of data it needs to store. This breakdown of microservices also highlights a way to approach one of the difficulties that we just discussed. The Ordering Microservice has got a concept of an OrderItem, which represents a single item in your order. The OrderItem entity contains a ProductId, a ProductName, and a UnitPrice. Now, hang on a minute, shouldn't the Catalog microservice be responsible for owning product names and prices? And yes, indeed, the Catalog microservice has got an entity called CatalogItem, which has an Id, a name, and a price. So when you order something, the Ordering microservice records not only the Id of the product you ordered, but also takes a copy of its name and price. And so we've actually got duplicate information, which is sometimes referred to as denormalization in databases. Now this means that if we were to update a product's name or price in the Catalog microservice, then the information in the OrderItem would become stale. So wouldn't it be better then if the Ordering microservice didn't store the product name and price, and instead asked the Catalog microservice whenever it needed it. Well, actually, maybe not. For one thing, that wouldn't perform quite so well as now we've introduced another network call. But another consideration is that it might actually be a good thing for your order to include the name and the price of the product as it was at the time when you placed the order. In other words, the Ordering microservice doesn't really care about what the current price and name of the product are. What's it cares about is what they were at the time you placed the order. And so, in fact, this isn't really duplicate information. These pieces of data have

different meanings within the context of each microservice. And so duplication of data between microservices and the inability to do a direct database join between the data owned by two microservices is not necessarily the problem that it might first appear to be.

## Components of a Microservice

The fact that a microservice owns its own data leads us to the realization that a microservice is not necessarily a single process running on a single host. Often at a minimum, there's your code, perhaps implemented as a web API, and the database. So that's at least two processes, and, often, they're not running on the same host. If we scale out, now we've got multiple replicas of our code running on different hosts and sharing access to a single database. And if we were to configure our database for replication or sharding, that too might be running on more than one host. A microservice might also have a cron job working on a schedule to perform some kind of data maintenance. And maybe it has another process that's listening for messages on a cue that trigger various other data updates. So, in fact, there may be several different hosts and processes involved in a microservice. But conceptually, together they form a single microservice, and it's only these components that are allowed to access the shared data. We can see this illustrated again in the eShopOnContainers sample application. If we look at the Ordering service, we can see that it not only contains a web API called Ordering.API, but there's a separate process to implement background tasks called Ordering.BackgroundTasks, which is used to provide a grace period feature. And this is a perfectly acceptable design for a microservice. You're not forced to have all of the microservice code running in a single process. The important thing is that conceptually, a microservice has a clearly defined boundary with a public interface, and its data can only be accessed through that public interface. Let's talk about microservice interfaces next.

## Microservices Are Independently Deployable

When we say that microservices are independently deployable, we mean that it's possible to upgrade a single microservice without requiring the clients of that microservice to be upgraded at the same time. And this requires great care because it means that you must never make a breaking change to the public interface of a microservice. In fact, it's a good idea to call the public interface a contract, which expresses the idea that it's an agreement between two parties that cannot be unilaterally changed. Now you might be wondering how is it even possible to never make a breaking change to your API? Well, the simplest solution is only to make additive changes to your APIs. You could add new endpoints, and you can usually add new properties to data transfer objects as most serializers can be configured to simply ignore additional fields that they aren't expecting. In some cases, you might find that you need to publish a version 2 of your public API, but if you do so, you still need to keep supporting version 1. Initially, your clients will carry on calling version 1 of your API, but over time, they can be updated to call version 2. Unfortunately, it's all too easy for developers to forget about this principle and make a breaking change to both the microservice API, but also update the clients at the same time. When they test this running locally on their development machine, everything works just fine. But now we've got ourselves into a situation where we're forced to upgrade both the microservice and its clients simultaneously. How can we avoid this? One approach that helps is by making individual teams responsible for

owning microservices rather than just allowing all developers to dive into the code for any microservice that they want to change. That way, a client must first ask the owners of the microservice to add a new capability, then wait for it to become available and deploy to production, and only then upgrade their client code to make use of that new feature. Another very helpful practice is to create automated tests around your microservices that explicitly attempt to call using any previous version of the clients that need to be supported. These tests need to be automated as part of a continuous integration build process. And if they fail, you should fail the build so everyone is aware that there's a problem with backwards compatibility. One pattern to be aware of is introducing shared code that both the clients and the microservice itself use. Now this can be very convenient for a microservice to produce a client package that simplifies the task of calling the service. But if you do take that approach, then you need to take great care to avoid any logic being added into that package that would result in tightly coupling the microservice to its clients and forcing them to be upgraded simultaneously.

## Identifying Microservice Boundaries

How can we work out the best way to break our system into microservices? This is a difficult task and easy to get wrong. Drawing service boundaries in the wrong place can result in poor system performance and are hard to change once those services are running in production. So it's worth dedicating some time to considering what your options are in this area. In some ways, your task is easier if you're starting from an existing monolithic application. Very often, there will already be some modularization, and some of those modules might be suitable candidates for converting into microservices, especially if they're loosely coupled from the rest of the system through a clear interface. Since microservices own their own data, looking for seams in the database is often a good idea. Are there certain groups of tables that conceptually belong together? If so, identify all the places in the code that will read and write from those tables and see whether they can be extracted to form a microservice. A helpful guideline is that microservices should be organized around business capabilities. You may be familiar with the concept of domain-driven design. This encourages us to identify bounded contexts within our applications. This is a way of breaking up the domain model for a large system with each bounded context having its own ubiquitous language and relating to one specific business capability or subdomain. And often these will map onto the organizational structure of the business. You then define some models that apply within your bounded context, and this means that different microservices will use different models and even might use different terminology for the same thing. We saw that earlier where the products that eShopOnContainers sells are called OrderItems by the Ordering microservice and CatalogItems by the Catalog microservice. Although they both relate to a product sold by the shop, they're being used in different contexts. And so they have different properties, and they're free to use different names for the same piece of information. Now it can be daunting to come up with good service boundaries. So a great approach to testing your ideas is to sketch them on a whiteboard and then run through some real-world use cases for your application and see which services would be involved. That can help you identify potential problems if too many services are required to collaborate together to achieve a single business capability. Now in just a couple of minutes, there's no way I can do justice to the concept of domain-driven design. And if you'd like to learn more about it, then there are several excellent courses available here on Pluralsight that cover it in a lot

more detail, including a whole learning path by Vladimir Khorikov. There are some pitfalls that you can run into when choosing your microservice boundaries. One is if you simply take all the nouns in your domain and turn them into services. This often ends up with you creating what are sometimes called anemic CRUD services, which are little more than wrappers around tables in a database with methods to add and update entities, while the logic related to those entities remains distributed across the rest of the system. Another sign that your boundary is not quite right is when you have circular dependencies between your microservices or clusters of microservices which all need to communicate frequently with one another.

EShopOnContainers Service Boundaries

Let's take a look at the choices made by the architects of the eShopOnContainers sample application to see some of these principles in action. There's a Catalog microservice responsible for storing details of the products available in the shop. Another microservice tracks what the customers have got in their basket. And then there's an Ordering microservice, which is responsible for handling the orders that we've placed in the shop. These three services relate to three distinct parts of the online shopping experience. The Catalog microservice supports browsing for products. The Basket microservice deals with preparing for a purchase. And the Ordering microservice deals with the business process of handling an order. And this separation of responsibilities makes a lot of sense. For one thing, it helps with resilience as customers will be able to browse for products to buy and put them in their basket, even if the Ordering microservice is temporarily unavailable. It also makes sense from a scalability perspective. Each of these microservices is dealing with very different volumes of data and access patterns. For example, the catalog is potentially very large and needs to support flexible queries as customers are searching for items that they want to buy based on various criteria. That means it makes sense for the Catalog microservice to store its data in a database that supports rich querying. And we can see here that it's using SQL Server, which makes a lot of sense for this kind of use case, whereas the Basket service deals with very short-lived data, so it may not even need to be written to a database. And, in fact, eShopOnContainers actually uses Redis, a memory cache to store basket data. The Ordering microservice is mostly writing new data to the database whenever a customer places an order. It's got very strict requirements for reliability. We certainly don't want to lose any orders. And it's also handling extremely sensitive data such as people's addresses and payment information. So its security requirements are very different from those of the Catalog service whose data is freely visible to everyone. There's another microservice here that handles identity, and this helps deal with authentication concerns. And we'll see why that choice has been made later on in this course when we look at security. The Marketing microservice relates to a completely different part of the business from the Basket and Ordering microservices. This microservice sends out email campaigns, which requires knowing where in the world our potential customers are located. And we can see here that there's a Location microservice that handles this responsibility. Obviously, the architects felt that this was a sufficiently complex component to warrant its own microservice. And, of course, this isn't the only way that you could break this domain down into microservices. But on the whole, I think that some good choices have been made here. And, in fact, this might be a good time for you to pause this video and take some time to think about the project that you're working on. How would

you break it up into microservices? What responsibility would you give to each one? And what data would each of those microservices own?

## Module Summary

In this module, we've looked at some of the key considerations involved in architecting a microservices application. You can evolve an existing system towards microservices by augmenting it with new microservices or by extracting existing functionality into microservices. We saw that microservices should own their own data, and they may consist of more than one process. Microservices need to be independently deployable. This requires taking care not to make breaking changes to the API or doing anything else that forces the clients of a microservice to be upgraded simultaneously with the service. The concept of bounded contexts from domain-driven design should be applied to identify suitable boundaries for a microservice. And getting those boundaries right is a critical factor in being successful with microservices. In the next module, we'll start to look at the practicalities of building microservices and what day-to-day development on a microservices project might look like.

# Building Microservices

## Module Introduction

What does the day-to-day development experience look like when you're working on microservices? I'm Mark Heath, and in this module, we'll look at some of the practicalities of building microservices. We'll start off by talking about some of the options for how microservices are hosted, as this has implications for development. In particular, we'll see why containerized microservices are so popular. We'll discuss the source control and build requirements for a microservice, as well as some of the standard features that should be present on every microservice. We'll look at the value of having a service template to work from, as well as a standard way of working that enables developers to productively debug and test their microservices in isolation, as well as in the context of the wider application.

## Hosting Microservices

When we develop a microservice application, we typically want to run it in a variety of environments. Developers want to be able to run and debug the code locally on their development laptop. Testers might want to run the application in a staging environment, maybe in the cloud. And, of course, we also want to be able to run our application in production. But how should we host the microservices? What should they run on? There are lots of choices here, and microservices don't dictate which option you go for. The traditional option is to use virtual machines. You could go for one virtual machine per microservice, but that can get very expensive, especially if you opt for having lots of small microservices. So, alternatively, you might decide to pack several microservices onto a single virtual machine, but that can get a bit messy as each microservice might have different

requirements for frameworks or dependencies that need to be installed. And there are several other operational challenges with using virtual machines, such as how each microservice can know where to find the others. Another option is to target Platform as a Service, or PaaS. Many cloud providers offer hosted services where you simply provide the code for your microservices and they take care of a lot of the management and infrastructure, providing you with automatic scale-out and DNS entries for each microservice with load balancing built in. They often also provide security and monitoring features as standard. Serverless platforms fall into this category and are especially good at supporting what might be called nano-services, which are extremely small microservices that just do a single task. And I've actually created several courses here on Pluralsight about serverless architecture and Azure Functions, which is a serverless platform for Azure. So feel free to check those out if you're interested in learning more about this approach. A third option that's emerged as one of the most popular approaches to implementing microservices, is to use containers. Containers let you package up an application along with all of its dependencies in such a way that it's easily portable to run on any container host. That container host could be running in the cloud, but it could also be your local development laptop, and this greatly simplifies the task for a developer who wants to get all of the microservices running locally. And tools like Docker Compose that we're going to be seeing later give us a really straightforward way of launching a group of microservices.

## Demo: Running Microservices Locally in Containers

Let's see how containerizing our microservices can make life much simpler for us as developers. We're going to be using the eShopOnContainers sample application that we've been looking at throughout this course to see how containerization greatly simplifies the development experience and enables us to easily run the whole application locally. First, let me quickly tell you how I've set up my development machine in order to run this demo. I've installed Docker Desktop, which allows me to run containers on Windows. And this actually supports running both Linux and Windows containers. We're going to be using Linux containers for this demo, and this means that if you're using a Mac, you can also install Docker Desktop and run exactly the same containers. I've cloned the eShopOnContainers source code from GitHub onto my local machine. And there were a couple of prerequisites steps on Windows that I needed to perform. One was that I enabled the WSL 2 based engine in the Docker Desktop General settings dialog. Another one was to add a Windows firewall rule that opens up some ports, and there's a helpful PowerShell script provided to do that. So, here I am at the Windows command prompt, and I'm going to use a tool called Docker Compose to work on this project. And the first thing I want to do is build all of the source code for my microservice into some Docker container images. And so I'm going to use the docker-compose build command to do that, and when I run this, it's going to go and build a Docker container image for each of the microservices in my application. And this is going to work regardless of whether I've got the .NET Core SDKs installed on my machine or not because it's going to build the code inside another Docker container that's got all of the necessary SDKs installed. Now, the first time you run this it is going to take quite a long time, it took about 10 minutes on my machine, and that's primarily because it's got to download a large number of container images that it's going to use as the basis for the containers that it's created. So I fast forwarded this to the point where it's finished building all of my microservice container images. And if we want to, we can use the docker image list

command to see the images that we've now got on our system, and you can see it's built a whole load of stuff, including images for each of the microservices. Now to actually run our application, we again can use Docker Compose, and Docker Compose has got this really handy docker-compose up command that will start a container for each of the services in the application. And again, the first time we run this, it will be a little bit slower as it needs to pull down some additional images for the third-party containers, like SQL Server, Redis, and RabbitMQ. But the great thing about this is I don't need to have any of those applications running locally on my development machine. My microservices are going to be storing their data in containerized databases. And once it's downloaded the images for all of those dependencies, it's going to start up a container for each microservice, as well as for each of those dependencies that we've containerized, and that will happen really quickly. So here you can see it started them all up, and it's also started to show us the log output from each of these containers. Now, if you're wondering how docker-compose up knew what services to start, well, all it does is it looks at a YAML file that lists all the containers that need to be started, along with any specific configuration or environment variables that they need. And, these files might look a little bit intimidating if you've never seen them before, but they're actually relatively straightforward and it's very easy to add in an additional service to the list that needs to be started. By convention, the file that Docker Compose looks for is called docker-compose.yml. And each of these containers actually comes with a variety of different override YAML files that can build on top of the base docker-compose.yml file which allows you to easily start the application in a variety of different configurations. As you can see here in this file, it lists all the services that make up the overall application. There's an entry here for each container, so here we can see a SQL Server and a MongoDB container, which are used for the databases when we're running locally. There's also a Redis Cache and RabbitMQ for the message bus. For production, we might not run any of these services in containers and prefer to point at cloud-hosted databases or event buses. But for local development, this is brilliant. If we scroll down a bit further, we can see the entries for each of our microservices, starting with the identity microservice and then the basket microservice. You can see that these point at the Dockerfile that can be used to create the image for each microservice if it's not already available. And that's what was happening when we ran the docker-compose build command earlier. Because it didn't have a local container image, it used that Dockerfile to build the container image. And if I scroll down a bit further, here we can see the catalog and ordering microservice definitions. Now, these containers should already have started up by now, so if I go to my web browser and we visit host.docker.internal on port 5100, we should see that the eShopOnContainers website is up and running. And we can see here on the shop home page that we can explore a range of programming-related merchandise. Let's log in, and the system comes preconfigured with a demo user that we can log in as, so I'll just copy in the username and password. And once we finish logging in, we can choose an item to buy and place it into our cart. Now let's go and look inside our shopping cart, and we'll click Checkout, we'll except these default shipping and payment details, and we'll place our order. And so now we've added an order to the system, and if we want to take a look at the order details, we can do so here. And so as you can see, it was very easy to get this up and running, and those steps we just ran through exercised a number of the microservices in our overall application. And if you want to try this out yourself, you can do that really easily whether you're on Windows or Mac, you just need to follow these getting started instructions that you can find on the eShopOnContainers GitHub page. And this has got detailed instructions of all the steps that you need to complete, as well as some troubleshooting hints as well for common problems that you might run into. Now, I

can't overemphasize how much of a game changer this sort of setup can be for developers. Imagine if I'd had to separately build and run each of those individual microservices, as well as install all of the third-party software like SQL Server and RabbitMQ and configure everything to talk to each other. That would be a really painful process and probably take the best part of a day to complete. But using containers and Docker Compose locally makes it really trivial to get started. We were up and running in just a few minutes. Another great thing about this setup is that many modern development tools will allow you to attach a debugger to code that's running inside a container, and so I can get a very rich local development experience with this kind of setup. Now obviously containers are not a requirement for microservices, but if you are planning to host your microservices on virtual machines or Platform as a Service, then you'll probably need to invest some time into automating the process for developers of getting everything up and running and configured on their local machines. Because if you don't, as the number of microservices grows larger, the productivity of developers will drop off dramatically. So having something like Docker Compose that gives us a one-command way to start everything up really is invaluable.

## Creating a New Microservice

What are the steps involved in creating a new microservice for your application? Well, first of all, you'll typically want to create a new source control repository for your microservice code. While it is technically possible to keep the code for all your microservices in a single repository, that can become very unwieldy if you have many microservices, and it also greatly increases the risk of introducing tight coupling between your microservices, which is something you really want to avoid. You should also set up a continuous integration build, which will build a new version of your microservice every time someone commits code into the source control repository. And this build should also run any automated tests you've created, and the build should fail if those tests don't pass. In fact, tests are a super important part of building microservices, so let's talk about them next.

## Testing Microservices

Automated tests are a critical part of being successful with microservices. We're not actually going to go into them in great depth on this course because other courses on the microservices learning path here on Pluralsight are going to cover them in a lot more detail, but I do want to quickly mention three very important types of tests that you should be aware of and be creating. The first is unit tests. These operate at the code level, and they're typically very fast to run. If you're following a test-driven development, or TDD process, then you'll already be creating these, and you typically want lots of unit tests and for them to have very good code coverage, especially any business-specific rules or logic. The second is service-level tests, or sometimes called integration tests. We want to be able to test an individual microservice in isolation from all the others. And these tests often require us to deploy an instance of the microservice to a host and configure it to talk to stubbed out or mocked versions of its collaborators, although it will talk to a real version of whatever database it's using to store it's data. Then you run a series of tests that call it's public API and verify that you're getting the expected responses from those end points. And

these sorts of tests are harder to write than unit tests, but they are tremendously valuable, so it's worth investing the time to create a framework that allows you to create these service-level integration tests for each of your microservices. And these tests too ought to be run as part of an automated build process. Finally, there are end-to-end level tests, and these are where you test all of your microservices running together in a production-like environment, and you automate some key journeys through your application from the user interface that exercise as many parts of the system working together as possible. And these types of tests can be much more challenging to write, to maintain, as they tend to be fragile and brittle and prone to failure due to transient problems. So try to verify as much functionality as possible with the other types of tests, but end-to-end tests are still very valuable even if you're just using them as what's called a smoke test that helps you determine that your system isn't completely broken, but is still able to provide some of the key bits of functionality end to end. Rather than starting from a blank sheet of paper

## Microservice Templates

for every microservice, it's a great idea to have some kind of standard template. This could be an actual template that you maintain in its own code repository that's used as the starting point for every new microservice you create, or it could be that you nominate a particular microservice as an exemplar. In other words, it demonstrates a pattern that other microservices should follow. And the reason for doing this is that there are many things that it's a good idea to standardize across all your microservices. Let me give you a few examples. Logging. All of your microservices should be emitting logs in the same format and sending those logs to a centralized location. Health checks. It's a great idea for every microservice to have a way of reporting its own health. Is it up and running yet, and can it talk to its downstream dependencies such as databases or maybe other microservices? Configuration. It's a good idea for all of your microservices to take the same approach to accessing sequence and configuration. Authentication. You might have some standard middleware that sets up the necessary authentication pipeline for your technology of choice, such as ASP.NET or Node.js, and by having a standard approach to that, you've reduced the risk of developers accidentally misconfiguring authentication and leaving a microservice unsecured. Build scripts. It's a good idea for there to be a standard way for developers to build the code. In the example we've been looking at, it's been using containers, so each microservice needs to have a Docker file that produces the container image. And, of course, there may be some additional things that you want every microservice in your application to have by default, so include those in your template or exemplar as well. Now by providing all of these things out of the box, you're going to greatly reduce the time needed to get a new microservice up and running, and you'll also ensure consistency across your product. Now, of course, this type of standardization might constrain some of the freedom that we discussed earlier that microservices give you to adopt new technologies. For example, you might decide that you're going to standardize on using one particular programming language across all your microservices, and that makes quite a bit of sense as it allows developers to more easily move between working on different microservices, but do try not to be so rigid with your standardization that you prevent teams from being able to choose the best tools for the job. The end goal of this is to provide a friction-free development environment. In other words, we want developers on our microservices application to be as productive as possible and to be able to spend their time

focusing on implementing the business requirements of the microservice rather than all the plumbing and infrastructure surrounding microservices. It should be possible for each developer to work on a microservice in isolation, running service-level integration tests against it to check that it still honors its contracts, but it should also be possible for each developer to test their work in the context of a full system. Depending on the size of your microservices application, it might be possible for developers to run everything locally on their development machine, just like we saw with the eShopOnContainers application in Docker Compose. Or you might decide to take a different approach where your developers connect to shared services running in the cloud. Whichever option you choose, try to ensure that the process is as streamlined and automated as possible.

## Module Summary

In this module, we looked at a few different options for how microservices can be hosted, and we saw how containers can offer us a particularly good local development experience. We discussed the fact that each microservice should have its own source control repository and continuous integration build configured. Automated tests are vitally important for microservices, and we said that each microservice should have its own unit tests, as well as service-level integration tests. And there's also the need for end-to-end tests that verify that the microservices integrate with one another successfully. We saw that there are several things that are worth standardizing across all your microservices, such as the approach to logging or authentication, and a great way to achieve that is by using service templates which serve as a starting point for creating a new microservice. In our next module, we'll explore the options for how microservices can communicate with one another.

# Communicating Between Microservices

## Module Introduction

How should our microservices communicate with each other? As always with microservices, there's a lot of choice available to us including both synchronous and asynchronous communication patterns. I'm Mark Heath, and in this module, we'll explore several approaches to reliable communication between microservices. We'll start off by looking at the big picture of communication between microservices. Should any microservice be allowed to communicate with any other? And what about client-facing applications such as websites and mobile applications? Can they simply call whatever microservice they want to? We'll learn about how API gateways can be used to decouple front-end applications from back-end microservices. We'll see that microservices can use either synchronous or asynchronous communication styles. And, in fact, many microservice architectures make use of a combination of both approaches. We'll look at RESTful APIs over HTTP, which is one of the most popular patterns for exposing microservice APIs. And we'll see how message brokers support asynchronous messaging patterns and why that's beneficial. For both synchronous and asynchronous communication styles, we'll also learn about some of the patterns that support resilient communications to ensure that even when things temporarily go wrong, your system is able to cope with the problem and recover from it. And, finally, we'll look at the

challenge of service discovery. How can microservices know where to find each other? We'll see how the service registry pattern can help us out here.

## Microservice Communication Patterns

Are there any rules that govern which microservices should be allowed to talk to each other? For example, if I've got a few microservices, is it okay for them all to just call each other whenever it's necessary? And if I've got a front-end application, like a website or a mobile application, should that also be allowed to call directly to any of the microservices it wants to? Well, once again, microservices doesn't stipulate any hard or fast rules here, but there are some difficulties with allowing a complete free-for-all situation like this. You can end up creating a mess of tangled dependencies between services that can result in cascading failures where one microservice failing causes the others to fail as well, and you can also end up with poor performance if making a call to one microservice ends up requiring multiple hops to call additional microservices. And if that's what's happening in your architecture, that might actually suggest that you've got some service boundaries in the wrong place. A better approach is to minimize calls between microservices. Instead, your microservices could publish messages to a shared event bus and also subscribe to messages on that bus. This promotes asynchronous communication between microservices, and we'll be discussing the benefits of that later in this module. When we're dealing with front-end applications, particularly with mobile applications or single-page applications where calls are coming into our microservices directly from a client-side application, then we might prefer to avoid the client application needing to know how to connect to all our different microservices. Instead, a pattern can be used called an API gateway where all the calls from the front-end client application go through the gateway and are routed through to the correct microservice. This offers several benefits. It means we can implementation authentication at the API gateway level, and generally it makes security easier to reason about if all the external traffic enters the system at a single point. This approach also decouples the client application from the specifics of our back-end APIS giving us freedom to be more flexible with changes to our microservices while maintaining a consistent public-facing API. And this is especially important if third parties are creating the client applications for our APIs, meaning that we aren't in control of their upgrade schedules. A closely related pattern called Back-end for Front-end builds on this idea. You can create an API gateway for each of your front-end applications, and this might even turn one incoming HTTP request into two back-end calls whose responses are aggregated or transformed into just the right format to meet the needs of the front-end application. Let's see what approach the eShopOnContainers sample app takes. First of all, we can see that they've embraced the concept of back-ends for front-ends, or API gateways, and both the mobile application and the two websites communicate through the API gateways. We can see that the microservices don't call each other but instead are raising integration events to an event bus, and this event bus is implemented either as RabbitMQ for when we're running locally or Azure Service Bus for when we're running in the cloud. And then these messages are subscribed to by the other microservices and handled asynchronously. And so this architecture is a good illustration of the fact that you can combine a few different approaches to communication patterns within the same system using each one where appropriate. So let's dive a little bit deeper now into our options for both synchronous and asynchronous communications between microservices.

## Synchronous Communication

By synchronous communications, we mean making a direct call to a microservice and waiting for it to respond to us. A good example of when we need to do this is when the front-end is making a query. Say, for example, in the eShopForContainers web page, we want to show the top 10 most popular products available. To implement this, we'd make a request through the API gateway and into the Catalog microservice, and then that microservice fetches data from its database which is a SQL database in eShopOnContainers and returns it all the way back to the web page. And because this is a synchronous communication, it's important that this operation is optimized to complete as quickly as possible. We don't want customers to be put off shopping with us by experiencing slow load times on our home page. And although there are many communication mechanisms available for calling microservices, HTTP is by far the most popular mechanism, and that's because it's an industry standard available in all mainstream programming languages including JavaScript, meaning that any web page or mobile application can easily call our microservices. In addition, we can take advantage of some of the other characteristics of HTTP such as standardized approaches to reporting errors and the ability to cache responses or use proxies. The payload of a HTTP request in response to our microservices is typically going to be in JSON, although other options are available including XML. One of the advantages of JSON is its wide support across the vast majority of programming languages, but also that it's the native format that JavaScript likes to work with, so it's really great when you're calling your microservices from web pages. Another pattern very often associated with microservice architectures is to implement your API as a set of RESTful resources. The RESTful approach to API design encourages you to represent the information that your service works with as resources. In eShopOnContainers, a resource might be a CatalogItem or an Order. Each of those resources is then manipulated using the standard HTTP verbs such as GET to retrieve a resource, POST to create a new resource, or PUT to update a resource. RESTful API design also encourages making good use of HTTP features such as meaningful response codes and using media types to specify how our resources should be represented. Now REST is a very big topic, and we don't have the time in this course to explain all of the philosophy behind it. But, fortunately, Pluralsight has several excellent courses in its library that will explain to you how to design a RESTful API and implement it in your technology of choice. Some of my favorite courses on this topic in the Pluralsight library are these ones by Kevin, which will give you a very thorough overview of working with REST and are especially useful if you're planning to implement your RESTful APIs in ASP.NET Core.

## Asynchronous Communication

Sometimes we might want to ask a microservice to do something that we don't need to wait for. A good example in the eShopOnContainers website would be the process of submitting an order. When we submit an order, there might be all kinds of tasks that need to be done by the back-end systems including taking payments from a payment provider, maybe ordering additional stock from a supplier, and then initiating the shipping process from the warehouse. And this process might take a few days to complete, and so we can't expect the user of the website to wait all that time to get an order confirmation. They just need to know that their order has been accepted and they can come back later to check its progress. We might even send them email updates to keep them updated on their order status and let

them know once it's out for delivery. And we can achieve this by making use of asynchronous communication patterns. It is possible to implement asynchronous communication patterns over HTTP. For example, when a customer submits a new order, our Ordering microservice could return a 202 Accepted HTTP response code rather than a regular HTTP 200 OK response code. 202 means that you've accepted a request to do some work but you haven't carried it out yet. And in the response body, you often send an ID of some sort so that the client can then pull for the status of that work to find out if it's completed. In our example, that ID might just be the order number. You might also want to combine this technique with the use of webhooks where the microservice itself reports back when it's completed the task. In this model, when a client calls your microservice, they can register a callback URL on which they'd like to receive any notifications. But another very common pattern for asynchronous communications is for microservices to publish messages to an event bus. Rather than directly calling the other microservices, they simply create a message and send it to a message broker, which serves as an intermediary. Other microservices then subscribe to those messages. And this pattern has a number of advantages. First of all, it completely decouples the microservices from each other. So instead of one service directly calling the next service, instead it simply talks to the message broker. This means that if the second service is temporarily unavailable, then the first service is still able to function, and the second service can just process any queued-up messages once it's online again. This approach is also very beneficial for supporting more advanced scaling patterns. If the number of unprocessed messages for a microservice is building up, you can scale that microservice out to multiple instances to help work through the backlog of messages quickly. And many serverless hosting platforms do this automatically for you. But with containerized solutions, it's also possible to achieve the same thing by configuring some automatic scaling roles for the cluster of virtual machines that your containers are running on. There are several different types of messages that you can use. Two of the most important types of message are commands and events. A command message is a request for a particular action to be performed. You can think of a command message as saying, Do this please. Imagine you have a microservice that sends out emails. That doesn't necessarily need to work synchronously, so it's a good fit for asynchronous patterns. We could post a command message that specified the details of the email that should be sent, and the email microservice could pick that up and send the message. Maybe the email microservice could get a bit behind if we posted large numbers of these command messages to its queue but it would eventually catch up. The other type of message is an event. An event message is simply a way of announcing that something has happened. Events are in the past tense and are not directed at any one microservice in particular. When you publish events, you're allowing any other microservices in your system that are interested to subscribe to that event and perform their own custom actions when it occurs. For example, if there was an OrderPlaced event in an e-commerce system like eShopOnContainers, then several actions might need to occur as a result of that event. For example, charging your credit card, sending a confirmation email, or checking on stock levels. And so each microservice that needs to perform an action when an order is placed will be triggered by that single event being posted to the event bus. And if you're interested in learning more about messaging patterns, then a great book to look at is called Enterprise Integration Patterns, and there are also several practical courses here on Pluralsight that will show you how to implement those patterns using various message brokers such as RabbitMQ or Azure Service Bus. So do check those out if you're interested in learning more about messaging.

Resilient Communication Patterns

We can't assume in a distributed system that all the microservices will be up and running the whole time. We can't even assume that the network is going to be reliable. So we must expect from time to time that communication from service to service will fail due to transient issues. And it's very important that we handle these failures well as they can result in cascading failures that are very difficult to recover from. And there are several techniques and patterns that we can apply to increase the resilience of our service-to-service communication. I'll just mention a few of them. If you're making any kind of network call, such as a regular HTTP request to another service or maybe making a query to a database, then it's a great idea to build in retries with back-off. This means that if your first attempt fails, then you just wait a few seconds and try again, and if that fails, then you wait a bit longer and make another attempt. And this simple technique can often be all that's necessary to handle various transient service outtakes. And you might find that your programming framework of choice has got built-in support for this functionality. For example, in .NET there's a great library called Polly that makes this very easy to implement. However, if too many retries are attempted, you can end up making the situation worse, and potentially you end up effectively issuing a denial of service attack on your downstream services because you're calling them so frequently. And so another pattern that can help us with this is called a circuit breaker. A circuit breaker sits in between the client and the server, and initially it allows all calls through. And we call this being in the closed state. However, if it detects any errors, maybe the server is returning error codes or it isn't responding at all, then the circuit breaker opens, which means now whenever the client makes a call, it's going to fail fast rather than passing on that request to the server. But after a configured timeout has elapsed, the circuit breaker allows a few calls through again to see if the downstream service has recovered. If so, then the circuit break closes, and it allows all calls through again. If the downstream service hasn't recovered, then it remains in the open state a bit longer, again quickly rejecting any incoming requests. And this is a very simple but powerful technique. And, again, many programming frameworks will have ready-made implementations of a circuit breaker that you can take advantage of. Caching can also be a valuable part of a resilient strategy. If you cache the data that you receive from a downstream service, then if that service is unavailable for a short period, you might be able to fall back to the data in your cache. Of course, that assumes that you have thought about the implications of using stale data. One of the reasons message brokers are so popular in microservice architectures is that they have inherent support for resilience. We can post messages to the message broker, and it doesn't matter if our downstream services are not currently online. When they start up again, they can catch up on the backlog of messages that are waiting for them. Message brokers also often support the ability to retry sending a message. If a message handler fails for any reason, the message can be returned to the message broker to be redelivered later, and this effectively means that you get a certain number of retries for free simply by using messaging patterns. Of course, you don't want to get stuck in an infinite loop attempting to process the same message again and again. So usually after a certain number of retries, the message broker will consider this message to be poisoned and put it into a dead-letter queue. Now there are some challenges associated with messaging that you do need to consider. One is it's very hard to guarantee that messages will always be received in order. So you need to take care that your message handlers will do the right thing even if they receive messages out of order. It's also possible to receive a message more than once. One example of when this can happen is when

you receive a message for the first time, maybe you only got part way through handling it and something went wrong, and so the message got returned to the queue. And now later, you receive that same message again as a retry. And this scenario is why it's really important to ensure that your message handlers are idempotent. A message handler is idempotent if calling it twice with the same message has essentially the same effect as calling it once. A good example in our e-commerce sample application would be handling an order. We only want to charge your credit card once, and we only want to ship your order once. So if the order received message gets processed twice for whatever reason, we need to make sure that we've got good checks in place in our code to protect us from charging you twice or shipping your order twice.

## Service Discovery

For our microservices to be able to communicate with each other, each microservice needs to have an address. But how can we find out what address all the other microservices are located at? Let's imagine we've got three virtual machines and we've got various microservices running on each machine. Maybe we've even got multiple instances of some of the microservices running on different machines in our cluster of virtual machines. Now we don't want to have to hardcode IP addresses in as that gives us poor flexibility to dynamically move our microservices between virtual machines in the cluster, which is necessary in the cloud as machines do need to be taken down from time to time for servicing purposes. Now there are a few ways that we can solve this problem. And one is to make use of what's called a service registry. This is a central service that knows where all the other microservices are located. Sometimes this works by each service reporting its location to the service registry when it starts up and keeping in touch to allow the service registry to know that it's still available at that address. That means that whenever you want to communicate with another microservice, you can ask the service registry to tell you where that microservice can be found. And the service registry itself is typically distributed across all the machines in your cluster, which makes it simple for you to be able to contact the service registry. But you don't necessarily need to build your own service registry. A number of microservice hosting platforms solve this tricky problem for us by means of DNS. For example, if you're using a PaaS platform to host your microservices, then often each microservice you deploy will automatically be allocated a DNS name that points to a load balancer sitting in front of the actual instances of your microservice. And that means you can just use the DNS name to communicate with your microservice and not have to worry about the actual IP addresses of the individual virtual machines that are running the service and might change over time. Or if you're using a container orchestration platform, like Kubernetes, then you can simply refer to the other services just by their service name. Kubernetes has got its own built-in DNS, and that means you don't need to know the IP addresses of each container. You just need to know the name of the service you want to talk to, and it takes care of routing traffic to the container that's running your service and load balancing if necessary. So the challenge of service discovery tends to be much easier if you're planning to host your microservices application on a modern microservice-friendly platform.

## Module Summary

In this module, we've learned about several different ways that microservices can communicate with each other. There are no hard and fast rules about what you have to use, but two of the most common mechanisms are implementing RESTful APIs, which allow your clients to make synchronous HTTP calls to your microservices, and publishing messages to an event bus, which allows asynchronous communication patterns. And these two approaches are not mutually exclusive, and it's quite acceptable to use them both in the same system. We also learned about the API gateway or back-end for front-end pattern where we put an intermediate proxy between our front-end applications and our back-end services. We also looked at some techniques that support resilient communication patterns, including the use of retries, circuit breakers, and caching with synchronous communication. And for asynchronous communication, we saw how we can take advantage of message retries and that we need to make sure our message handlers are idempotent. Finally, we learned that service discovery can be implemented using a service registry and that many modern PaaS microservice hosting platforms and container orchestrators like Kubernetes can solve this problem automatically for us making our lives much simpler. Now we've not yet discussed the very important problem of how to secure our microservices. So let's look at that next.

# Securing Microservices

## Module Introduction

Modern distributed cloud applications often store and process large amounts of sensitive data, and it's of the utmost importance that we protect that data ensuring that only authorized users can access it. I'm Mark Heath, and in this module, we'll be learning about some techniques and principles that we can apply to ensure that the microservices we create are properly secured. We'll start off by discussing the importance of encryption. We want sensitive data to be encrypted in transit and at rest. Next, we'll look at authentication. When a microservice receives an incoming HTTP request, how does it know that the request has come from a trusted caller? And just because we know who the caller is doesn't necessarily mean that they're allowed to perform the action that they've requested. So we'll also be looking at the related concept of authorization. And we'll see how industry standard protocols such as OAuth 2.0 and OpenID Connect can help us with authenticating and authorizing the callers of our microservices. Next, we'll explore some of the options for securing the network, including the use of virtual networks, IP whitelisting, and firewalls. And, finally, we'll learn about the implications of the defense-in-depth principle where we apply several levels of security rather than just relying on one. And some of the additional security mechanisms that we can add are arranging for penetration testing, setting up alerts when attacks are detected, and auditing everything.

## Encrypting Data

Modern microservice applications often store and process large amounts of data. Not all of that data will be sensitive. For example, in our eShopOnContainers sample application, there's

a Catalog microservice. And the Catalog microservice deals with all the products that the store sells, but this isn't sensitive data. We're happy for all of our customers to be able to explore all of the catalog data. However, the Ordering microservice holds details of which customer has ordered which items. It may also hold details of their shipping address or payment details. And these are very sensitive pieces of information. And to leak them would have very serious consequences. So it's imperative that steps are taken to secure any microservice that deals with sensitive data. And encryption plays an important role in protecting our data. At a minimum, we want to ensure that data is encrypted in transit. That means that as it's transferred across the network, it should be encrypted so that no intermediate party is able to listen in. Of course, the encryption algorithms you use should be up-to-date, industry standard algorithms. You should never attempt to invent your own encryption algorithms. For HTTP communications, encryption can be achieved by configuring Transport Layer Security, or TLS, so that all requests are made using HTTPS. Now this does require you to configure SSL certificates, which can be a complex process since you need to get one issued by a trusted certificate authority for each of your services and to have a mechanism for updating them when they expire. Fortunately, many cloud microservice hosting platforms provide mechanisms that simplify the process of configuring HTTPS for your services. You may also find that your customers require data to be encrypted at rest. This means that whenever it's stored on disk, those disks should be encrypted. This can also be complex to configure as you'll need a secure way to manage the encryption keys. But an increasing number of cloud providers are offering encryption at rest as a standard feature for file storage in databases. If you're considering using encryption at rest, don't forget that any backups you make of the data from your system should also be encrypted.

## Authentication

Just because you're using encryption in transit with HTTP requests being made using TLS doesn't mean that you can trust the caller of your service. Each incoming request needs a way to tell us who the caller is so we can make a decision as to whether they're authorized or not. And there are several different ways we can achieve this. With HTTP, the most common approach is to include an authorization header in each HTTP request. For example, this might contain a username and password, which is a technique known as basic authentication. This approach is sometimes used when the front-end client application provides the ability for the user to log in. And so we send the credentials that they've entered in the header in order to authenticate them. The trouble with this approach is it means our microservices now have the responsibility of securely storing user passwords, and if we're doing that, we need to take great care to follow the best practices for hashing those passwords. And that's probably something we'd rather not have to do ourselves if possible. Alternatively, when one microservice calls through to another, the authorization header might contain an API key. With this approach, you give each valid client of your microservice their own API key. However, that's again more secrets that you need to manage and potentially rotate if they get compromised. Another option that you might see is the use of client certificates. By using public-key cryptography, a certificate gives us a very secure way to allow a caller to prove their identity. However, certificates also can be quite complex to install and manage. So is there a better way? Well, one of the most promising approaches is for microservices to build on some industry-standard protocols such as OAuth 2.0 and OpenID Connect. And this introduces the concept of an authorization server. The authorization server, which is called

the identity server in our eShopOnContainers sample application, so I'll be referring to it as an identity server in this module, takes care of a lot of the complexities of authenticating and authorizing the callers of our microservices. The way this works is that, first, the clients of our microservices authenticate themselves with the identity server by sending credentials of some sort, and then the identity server returns an access token which has got a limited lifetime. That access token can then be passed in an authorization HTTP request header as part of the call to the microservice. And these tokens are signed using public key cryptography, meaning that it's possible to verify that the token really was issued by the identity server. And this approach, which I've only really given you a simplified explanation of here, turns out to have a lot of benefits. For one thing, it means that only one service has the job of authenticating users and managing their credentials securely. And because the identity server is implementing industry standard protocols, like OpenID Connect, that means that you don't need to write this component yourself, you can make use of a third-party solution, which is great because writing your own identity server is a lot of work. And that's why the eShopOnContainers sample application makes use of an open source product called IdentityServer4 which is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Now if you're planning to adopt this approach to microservice authentication and authorization, I recommend that you invest some time in learning more about the underlying protocols we've mentioned such as OAuth 2.0 and OpenID Connect, as well as how to best make use of them in your own language of choice. And there's plenty of excellent courses here on Pluralsight that can help you with this. A good starting point would be the Getting Started with OAuth 2.0 course by Scott Brady. And if, like me, you're a .NET developer, then these two courses on securing ASP.NET Core replications would be excellent choices to follow up with.

## Authorization

When a user is authenticated, that simply means that we know who they are. But we also need to check that they're authorized to perform the action that they've requested. For example, if I log in to eShopOnContainers, then I'm authorized to view my own order history, but I shouldn't be authorized to view your order history. Many web API frameworks, like ASP.NET Core, which eShopOnContainers uses, provide built-in mechanisms that help you perform authorization by checking various things about the user such as whether they've got a particular role or not. And it's your responsibility as a developer to think very carefully about every single endpoint of the public API of your microservices and consider what actions each potential user of the service should or shouldn't be allowed to perform. There's one authorization gotcha that I particularly want to highlight as it applies especially in microservices scenarios, and this is a problem sometimes known as the confused deputy. Suppose I, as an end user, make a call to a microservice. Let's say it's the Ordering microservice. But that microservice, as part of what it's doing, makes an ongoing call to another microservice. Let's say the Payment service. Now when I call the Ordering service, the credentials or access token that I supply are going to indicate that it's me, Mark, who's making the call. The Ordering service can then decide whether or not I'm allowed to perform the particular action that I've requested. However, when the Ordering service calls the Payment service, it might send credentials that simply say that this is the Ordering service making the call. And because the Payment service trusts the Ordering service, it assumes that it's safe to perform whatever action has been requested. But this could be risky because what if I could somehow trick the Ordering service to ask the Payment service to pay

for my order but using someone else's credit card details? Now there are a few ways to address this issue, but one solution is by issuing on behalf of access tokens. This way, when the Ordering service calls the Payment service, it still identifies itself as the Ordering service, but it also tells the Payment service that this is a call being made on behalf of Mark. And that gives the Payment service more information to go on, and it can reject the request if it recognizes that I'm asking to pay with a credit card that doesn't belong to me.

## Securing the Network

Another important form of defense is to make use of networking features such as firewalls, IP whitelisting, and virtual networks to secure your microservices. In this diagram, we can see three microservices communicating with each other. And if we can place them inside a virtual network, then we have the ability to reject any incoming traffic from outside the virtual network and only allow them to communicate with each other. And this type of approach is a great idea to help protect your back-end services, which may need to communicate between themselves, but you don't necessarily need access to them from outside. But what about those situations where you do need to accept traffic from the internet? For example, if you've got a website implemented as a single-page application, then it will want to make requests to your microservices. So does that mean you have to forego any kind of network restrictions and just open everything up? Well, this is actually an example of where the pattern that we discussed earlier in this course, the API gateway or back-end for front-end, can be really helpful. We can have an API gateway that accepts incoming requests from the public internet but is also connected to the virtual network and so is able to pass on incoming requests to the back-end microservices. And this means we can be very selective about exactly which back-end APIs are able to be called from the outside. It also means that we've got a single point of entry. Some cloud API gateways can also be configured with a firewall or other types of attack protection such as guarding against distributed denial-of-service attacks or SQL injection attacks. Often, in a microservices application, there's a public-facing website, like the eShopOnContainers store website. But there may also be additional websites intended for use by a smaller targeted group of users such as system administrators. For example, the eShopOnContainers application supports creating marketing campaigns, which are sent out to customers by email. And if there was a marketing website that allowed them to create these campaigns, the only people authorized to access that website are staff from our company. So we might decide to apply IP address whitelisting so that this website rejects any traffic that doesn't come from certain known IP addresses such as the IP addresses of the marketing officers. Now so far in this module, we've talked about quite a few different techniques for securing our microservices, and you might be wondering whether you need to use all of them or whether you can just pick one or two. And this is where the defense-in-depth principle comes in. This principle encourages us to avoid relying solely on a single layer of protection and to use multiple layers of security wherever possible. So let's discuss the idea of defense in depth in a bit more detail.

## Defense in Depth

We've discussed a number of techniques in this module to secure our microservices. By using encryption in transit with TLS, the traffic to and from our microservice can't be listened into

by unauthorized third parties. By using access tokens issued by an identity server to authenticate ourselves in the requests, our microservices can reject any incoming calls from unauthorized clients. And by using virtual networks and IP whitelisting, we can reject any calls to our microservices that come from an unauthorized network location. And the defense-in-depth principle states that you shouldn't rely entirely on a single technique to secure your application, and that's because if that one defense is breached, then everything is lost, and your attacker has gained free access to everything. So by combining several different layers of security, we can significantly reduce the possibility of a data breach. And the more sensitive the data that you're dealing with, the more important it is that you apply several layers of protection. And certainly these three techniques that we've just discussed I would consider to be essential for securing any microservices application that deals with sensitive data. But there are some additional defensive measures that we can take. For one thing, modern hackers have a variety of very sophisticated tools and techniques at their disposal, and so often software developers aren't aware of all of these attack techniques and don't have the knowledge of how best to mitigate against them. And for that reason, it's a very good idea to arrange for penetration testing to be performed by a team of infosec experts who can test the defenses of your application against state-of-the-art hacking techniques and can also give you some expert advice on how to increase the security of your applications. I also recommend that you create automated tests that verify that your security is working as expected. Don't just assume that you've configured your virtual networking or access token validation correctly. Run some tests that prove that unauthorized users are not able to call your APIs. It may also be possible to detect when an attack is in progress. Plot scanning, repeated login attempts, HTTP requests, phishing for sensitive files, SQL injection attack attempts, all of these things can be detected in real time. And you should configure alerts so that you know when they're happening, and this allows you to proactively make the decision either to block traffic from the attacker's IP address or maybe even to temporarily shut down part of your system to protect against the attack. Finally, you should have good logging and auditing of all of the actions being performed in the system. This way it's possible to review exactly who did what and when they did it. And that way, if there is a data breach, you'll have a much better chance of understanding how it happened, as well as exactly what data has been compromised.

## Module Summary

Securing our microservices should be one of our highest priorities. The stakes can be very high when you're working on a system that deals with sensitive data. And that's why, in this module, we emphasized a defense-in-depth approach. And the techniques that we should consider applying wherever possible include encrypting data in transit with TLS, encrypting data at rest in files and databases, authenticating users, ideally by means of industry standard protocols such as OAuth 2.0 and OpenID Connect, and ensuring that we also correctly authorize access to our microservices so that users can only perform the actions that they have permission for. In addition, we can configure various network level security such as virtual networks, IP address whitelisting, firewalls, and making use of API gateways that can create a single point of entry for external traffic. It's also a great idea to make use of security experts to perform penetration testing and provide feedback on our system design. And we should configure alerts for attack detection and regularly review the system audit logs to identify any suspicious access patterns or behavior. In our next

module, we're going to look at how we can deliver microservices focusing in particular at the challenges of deploying and monitoring our microservices.

## Delivering Microservices

### Module Introduction

In this course, we've talked a lot about how to architect and build microservice applications. But, of course, before long, you are going to want to deploy your microservices to production. And while they're in production, you'll also need a way to monitor them and ensure that they're functioning correctly. I'm Mark Heath, and in this module, we're going to learn about some good practices for how you deliver your microservices into production. We'll start off by looking at how we deploy microservices. It's really important that we set up an automated process that can repeatedly and reliably upgrade individual microservices. And we can achieve this by setting up a pipeline that can deploy code into one or more environments. It's also important that once our microservices are running in production that we have a good way of monitoring them, and we'll see how having standardized health checks and aggregated logs can benefit us greatly when we're diagnosing issues in production.

### Automated Deployment

In the traditional world of monolithic applications, it's not uncommon for deployments to be a manual process often where you follow a step-by-step Word document that explains how to upgrade to a new version of your software. But once you've moved to microservices, a manual process simply isn't going to work anymore. You've got far more things to deploy, and you also want to deploy much more frequently. And so microservices pushes us very strongly in the direction of automating our deployments, and that's a good thing. If we've got a reliable and repeatable deployment process for our microservices, that allows us to upgrade them frequently as soon as new features or bug fixes become available. In fact, some teams using microservices deploy multiple times a day. So how can we achieve this? Well, we've already discussed how a continuous integration server can be used to automatically build our code whenever we check it in, and it's also very common for that CI server to run unit tests as well. But we can take this idea further. The CI server can trigger what's sometimes called a release pipeline. A release pipeline starts off where the CI build has finished. It's built the code and run the unit tests. But, next, we want to actually deploy the microservice. maybe to a virtual machine running on-premises or to a cloud-deployed resource. Once we've deployed the microservice, this is a great opportunity to run the service-level integration tests that we discussed earlier on in this course where we're testing that microservice in isolation. If it passes those tests, the next step might be to deploy the microservice into a QA environment where it can be tested as part of the entire system. And this is the point at which you'd run any automated end-to-end tests. Now at this point, there's often what's called a release gate. Just because our microservice has successfully passed all of its automated tests doesn't necessarily mean that we're ready to push it straight to production. There may be the need for some additional manual testing or some kind of risk assessment or customer sign off

before, finally, we're ready for this code to go to production. Once you reach the stage of being ready to push a microservice into production, it's really important that you use exactly the same procedure that you used to deliver it to the QA or staging environment for pushing it to production.

## Deployment Environments

It's very common that we want to be able to deploy our microservices to different environments. A developer might have their own virtual machine that they want to run a microservice on during development and that will allow them to debug their code locally. And as we've just said, you might have a QA environment in which you're going to run your end-to-end tests or perform some manual testing. Sometimes you might have other environments dedicated to penetration testing or performance testing. And, of course, another really important environment is production where our code is actually being used by the end users. And in some scenarios, you have multiple production environments, maybe one for each of your customers or one running in each of a few geographical regions. So it's really helpful to parameterize our automated deployment scripts making the process of deploying to any of these environments as simple as possible. A common approach to dealing with this is to define each environment in a configuration file, and JSON or YAML are very popular choices here. These configuration files essentially express what's different about the environment we're deploying to. So here I'm showing an imaginary configuration file that expresses that the QA environment should be deployed in the West Europe cloud region, it should use medium-sized virtual machines and scale between a minimum of 2 and a maximum of 5 instances of the virtual machine. And configuration files like this are often used in conjunction with other deployments scripts and manifests to allow you to deploy a specific version of a microservice into a specific environment. For example, it would be really nice if we would run a command like this where we're asking to deploy version 1.0 .4 of the Ordering microservice into the environment that's defined in the qa.yaml file. But, of course, this is just a simplified made-up example here, but there are all kinds of tools that you could use to achieve this including Docker Compose that we saw earlier, or if you're using Kubernetes, that's got its own tool called kubectl that can apply the contents of a configuration file with a command that looks a bit like this. There are also technologies like Terraform and Azure ARM templates, which allow you to template the creation of the cloud infrastructure you need to host your microservices, and that might include virtual machines, databases, or blob storage accounts and virtual networks. And these too allow you to define a hierarchical set of scripts with base templates that are overridden with environment-specific configuration.

## Artifact Registries

To allow the deployment scripts to easily deploy any version of our microservices, we need the build artifacts to be stored in some kind of registry. That way we can easily deploy the latest version of our microservice or roll back to a specific previous version. In the eShopOnContainers application that we've been looking at in this course, they've made the choice to build every microservice as a Docker container image. And using containers has many benefits, but one of them is the way that they are named and versioned in a standard way and can be stored in a container registry. And that means it's really easy when you're

using Docker to identify which container you want to be deployed. Here's an example of what name and tag we might give to version 1.3 .1 of the Ordering service in our e-commerce example. Because the eShopOnContainers application is using containers as a delivery mechanism, it's ideal for deploying to a container orchestrator such as Kubernetes. If you have a Kubernetes cluster, then you can define all the microservices in your application with some YAML manifest files. Kubernetes actually operates using a desired state strategy. Your manifest files define what microservices should be running and how they should be configured. And then Kubernetes compares what your manifests have asked for with what's actually happening right now on the cluster. If there's a difference, then it will add or remove containers until what's running matches what you asked for. And this means that upgrading to a new version of a service or, indeed, changing any other property of a service such as an environment variable or the number of replicas simply becomes a case of changing the manifest files. And if you'd like to dive deeper into Kubernetes, I highly recommend that you watch some of the excellent courses here on Pluralsight that will guide you through how it works. A really good place to start would be this Kubernetes for Developers course by Dan Wahlin.

## Independent Upgrades

We've mentioned before in this course how important it is that we can independently upgrade microservices. We don't want them to be tightly coupled in such a way that upgrading a single microservice requires all the others to be upgraded at the same time. And that means our automated process needs to be able to update just one microservice at a time even if it can deploy them all at once, for example, if we were doing a fresh deployment of the entire application. So it's very important that you think about how you're going to upgrade an individual microservice. You might think that it's a simple case of just stopping the old one and starting the new one, and that's certainly a valid approach. However, it does cause a small period of service unavailability. And depending on what that microservice is responsible for, that may or may not be acceptable. But there are a number of other techniques that you can use to reduce downtime when you're upgrading a service. One is called a blue green swap where you run the old and new versions of the service simultaneously and then swap traffic from one to the other by means of a load balancer. And that means you don't have downtime while you're waiting for the new version of the service to start up, and so you should be able to get near-instantaneous swap over. Another option, if you've got multiple replicas of a service running, is to replace the instances one by one. So if I've got three instances of version 1 all load balanced, I might, one at a time, add new instances of version 2 and remove instances of version 1 until all three are upgraded. Now this does assume that you've written your code in such a way that having both versions running at the same time doesn't cause any problems. If you are using a platform designed to host microservices, such as Kubernetes, then many of these more advanced upgrade strategies are available to you out of the box. Another thing to consider is how seamlessly you can roll back to the previous version if anything goes wrong. This is another thing that a container orchestrator like Kubernetes can help you with since rolling back is simply a matter of updating our manifest files to point at the previous tagged version of our microservice. Now, of course, you're only going to do a rollback if you've been able to detect that something about the new version isn't working correctly. And so that brings us onto another really important topic of how we can monitor our microservices.

## Monitoring Microservices

One of the big challenges of microservices is that there are more things for us to monitor. Your microservices might be running as dozens of different processes across many different host machines, and you don't want to have to individually connect to every worker node in your virtual machine cluster and examine individual log files for every microservice. Instead, what we want is a system where all of our monitoring, telemetry, and logs are available in a centralized place. Ideally, we want a dashboard that gives us an instant view of the overall health of the system and the ability to dive in deeper if a problem is detected. Let's have a think about some of the things that we want to be able to monitor. First of all, we want to be able to capture host metrics. This includes things like CPU and memory usage. By tracking these metrics, we can detect if we need to scale out to meet increased demand. Many cloud providers offer the ability to set up alerts based on these sort of metrics so you can be notified if there are any potential problems. Secondly, we want to track metrics at the application level. So for a web API, this would be tracking the number of HTTP requests, including the number of failures, and tracking what the errors codes were. For example, if there were lots of 401 Unauthorized response codes, then maybe our system is under attack by hackers or maybe we've just misconfigured something. If there are lots of 500 errors, then that points to a bug somewhere in our code. If you're using a message broker, then you'll want to track whether large numbers of messages are backing up in the queues, in which case, again, we might need to scale out. And you'll also want to track whether messages are being sent to a dead-letter queue, which indicates that you've got a problem processing your messages. One particularly helpful approach is for each of your microservices to support a health check endpoint. This is a web API that can be periodically called to check your service is running and functioning correctly. It might be as simple as just reporting OK to indicate that it started up successfully. But it can also be useful for your health check endpoints to return additional information such as whether any downstream dependencies like databases are currently accessible. Third, we need the logs to be easily accessible. Each microservice should be emitting logs, and if those logs are aggregated into a centralized place, that will make your life a whole lot easier. One nice thing about using containers is that they already have a standardized approach to capturing logs. So it's relatively straightforward to aggregate the logs from all the containers into a single place. And there are many products that can help you achieve this. A couple that I've used in the past are sending logs to Elasticsearch and then using Kibana to view them, or in Azure, using Application Insights. Many microservice hosting platforms or the serverless PaaS platforms like Azure Functions or container orchestrators like Kubernetes already come with lots of easily enabled monitoring and logging capabilities. And that's another reason why it's a great idea when you're using microservices to take advantage of these kind of observability features rather than having to spend a lot of time creating your own customized solution to these common problems that everyone who's building microservices is going to need a solution to. Let's take a look next at how the eShopOnContainers sample application allows us to easily monitor what's going on with our microservices.

## Demo: Centralized Logging and Health Checks

In this demo, we'll see how the eShopOnContainers sample application that we've been looking at throughout this course sends its logs to a centralist service called Seq and

implements health check APIs for each microservice that are aggregated together into an overall health dashboard. I'm going to start up the eShopOnContainers application locally with the docker-compose up command that we saw earlier in this course, and this will start up a container for each of our microservices along with some additional containers for the databases and other supporting services. And we can see that these containers start very quickly because the images are all available locally on my machine. And, docker-compose up, by default, will actually emit all the container logs to the console, so we can already see some of the logging here. But if I quickly go to my browser and visit localhost port 5107, then what we can see here is a health check dashboard. And what this is doing is calling the health check endpoints on each microservice in the system. Of course, because we've only just started up, not all of the microservices have finished their startup yet. And this page is supposed to auto refresh, but for some reason it's not working at the moment. So I'll manually refresh, and we can see that nearly everything is up and running now with the exception of WebMVC. What's even nicer about this dashboard is that I can drill down into each of the microservices and it will tell me whether it's able to contact the dependencies of that microservice or not. And this is incredibly helpful as it can be possible for a microservice to start successfully but for it not to be able to do its job properly because it can't communicate with a downstream service, and that might be due to a configuration problem or maybe a transient network issue. Here we can see that the WebMVC service, which is the main website, has started up okay. But it also needs to be able to communicate with the identity microservice, which currently isn't responding. But pretty soon, if I refresh again, we'll see that everything is now working. Not only are all our services up and running, but they're also able to communicate with their dependencies. And so a dashboard like this is a really great way to get a quick, high-level view of the overall health of your microservices application. What about the logs? Well, I can visit localhost port 5340, and we can see here a dashboard showing the aggregated log output from all of our microservices. And this is possible because all of the containers have been configured to send their logs to an application called Seq. Now, this is a commercial product, so you do need to license it if you want to use it in production, but it is free for use in development scenarios. And from this page, I can drill down to see more details of each log message. Seq is designed to support structured logging. So the logs aren't just lines of text, but they can contain additional key value pairs of useful information. So here it's showing us which specific event was being handled when this log message was emitted. And Seq also supports many additional useful features, such as the ability to search and filter the log messages. And Seq is just one example of a product that helps us explore the logs from all of our microservices in a centralized place. And there are several other alternatives that you could choose, but whichever one you select, it will be a huge timesaver compared to having to access the logs for each microservice in a different place.

## Course Conclusion

In this module, we looked at some of the challenges of getting our microservices running in production. We saw how important it is to automate our deployments as when you've got lots of microservices, a manual process is just going to be too cumbersome. We saw that you ideally want to set up a release pipeline that allows you to deploy your code into different environments, and run your automated tests in those environments, and then, finally, release it into production once you're ready. We saw that microservices should be independently

deployable, and there are several approaches to how you upgrade a microservice such as rolling upgrades or blue green swaps that allow you to achieve minimal or zero downtime and the ability to roll back if something goes wrong. Finally, we looked at how important it is that our microservices are observable. We need to be able to monitor host and application metrics, set up alerts to warn us if things go wrong, and to easily examine the logs of each microservice in a centralized place. And we also saw how implementing health check endpoints on each microservice is a great way to enable you to get a quick picture of the overall health of your system. And that brings us to the end of this course. I hope I've been able to give you lots of practical ideas for how you can be successful with microservices in your own projects. Before we wrap up, let me just end with a few final thoughts. First, as we've said before, microservices give you lots of options. In this course, we've talked about things like containers, RESTful APIs, and back-end for front-ends. But none of these technologies are required for microservices. You're free to choose the ones that make the most sense for your context, and that's important because every context is different. Your task now, if you want to adopt microservices, is to identify the patterns and technologies that make the most sense for the application that you're building. And that's going to take into account the skills and experience of the team that are going to be developing it. Third, in this course, we've only had time to scratch the surface of microservices. And it's also an area that's changing very fast with lots of rapid innovation, and many new tools and frameworks are being designed with the goal of improving the experience of building microservices. So I do encourage you to keep learning. Do check out the rest of the courses on the microservices learning path here at Pluralsight, and I can also recommend this excellent book on Building Microservices by Sam Newman, which has recently been released in a second edition. And, finally, it's likely that your first attempts at building microservices are not going to be perfect, but don't be discouraged about that. One of the best ways to be successful with microservices is to keep evaluating the architecture that you've currently got and gradually evolving it towards some of the improved patterns and practices that we've been learning about in this course.